****

2021年秋季学期

操作系统课程设计报告

组长：\_\_\_\_李坤璘\_\_\_\_\_

成员：\_\_\_\_张杰宁\_\_\_\_\_

成员：\_\_\_\_\_方凡\_\_\_\_\_\_

课程设计选题一 多线程单词统计

一．设计要求

1.数据要求

输入数据：若干篇英文小说

输出数据：各篇小说各自单词数、单词总数、所有单词出现的频率、top20热词

进阶要求：将无用单词筛选后的top热词；采用多线程/单线程以及不同文件读取操作所耗时间的输出比较

2.功能要求

（1）用多线程统计单词出现频率并从中找出Top-N热词

（2）使用相关提示语以使多线程程序中直观展示出多线程并发调度的过程。

（3）用单线程实现以上功能，并比较单线程和多线程的时间效率

（4）实现细节处单词分割功能

（5）学会使用linux中文件夹的管理操作

（6）使用多种文件读写方式，并与单/多线程结合使用。

二．背景知识及设计思路

1.背景知识

**（1）Linux 的线程创建**

使用pthread\_create 函数创建用户级线程

**函数定义：**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr，void \*(\*start\_rtn) (void\*)，void \* arg);

**参数说明：**

thread: 待创建线程的 id 指针

attr: 创建线程时的线程属性

v(\*start\_rtin)(void\*): 返回值是 void\*类型的指针函数

arg: start\_routine 的参数

**返回值：**

成功返回 0

失败返回错误编号

 EAGAIN：表示系统限制创建新的线程，如线程数目过多

 EINVAL：代表线程属性值非法

**（2）线程同步终止**

**函数定义：**

int pthread\_join( pthread\_t thread, void \*\* rval\_ptr); **功能：**

①调用者将挂起并等待新进程终止

②当新线程调用 pthread\_exit()退出或者 return 时，进程中的其他线程可通过pthread\_join()获得进程的退出状态

**返回值：**

成功结束返回值为 0,否则为错误编码

**（3）带有线程程序的编译**

**基于 POSIX 标准的线程编程接口：**

 包括一个 pthread.h 头文件和一个线程库

**编译方法：**

 g++ \*\*.c -o \*\*\* –lpthread

**（4）TOP-N热词统计设计**

C++ 中有一个<map>容器类，其实就是封装了一个红黑树（近似平衡二叉树），可以调用map容器来实现基于平衡二叉树的字典结构。

map<string,int> word\_count

map<string,int>::iterator iter

string word

word\_count.insert(pair<string,int>(word,1))

iter = word\_count.find(word)

word\_count[iter]+=1

**（5）读取文件夹的目录和文件**

**函数定义：**

struct dirent \* rea

.ddir(DIR \* dir);

**函数功能：**

返回参数dir ⽬录流的下个⽬录进⼊点。

**结构dirent 定义：**

struct dirent{

ino\_t d\_ino; //d\_ino 此⽬录进⼊点的inode

ff\_t d\_off; //d\_off ⽬录⽂件开头⾄此⽬录进⼊点的位移

signed short int d\_reclen; //d\_reclen \_name 的长度, 不包含NULL 字符

unsigned char d\_type; //d\_type d\_name 所指的⽂件类型

har d\_name[256]; //d\_name ⽂件名

};

**返回值：**

成功则返回下个⽬录进⼊点. 有错误发⽣或读取到⽬录⽂件尾则返回NULL.

2.设计思路

（1）采用STL标准库中的map来记录不同单词的个数，本质是近似的平衡二叉树。

（2）通过逐个字符扫描，判断当前扫描的字符是否为字母，以此决定是否在此处进行单词分割。

（3）采用vector存储要扫描的文件夹中所有的小说名，在操作系统中可以认为是一个共享的buffer数组。

（4）可以设计一个集合，存储着那些没有意义的单词，在我们后期输出的时候，一旦待输出的单词在这个集合中，说明它不是有意义的单词将被舍弃。

（5）核心单词查找函数find，在一开始的时候，程序会判断目前容器中是否还有元素，通俗讲就是还有没有没有被线程处理的小说，因为当线程选中某一篇小说后，就会将这篇小说的名字从容器中去除，同时在选中的过程中也涉及到了PV操作。

三．设计方案

1.数据结构设计

const int MAX=1000;//定义最大路径名称长度和单词的最大长度

char dir[MAX];//存放.txt文件的文件夹名

int thread\_num=0;//线程的个数，也就是文件夹里.txt文件的数目

vector<string> txtfiles;//保存文件的路径

map<string,int>mp;//创建结果的map

set<string> st;//存储意义不大的介词、冠词等单词

pthread\_mutex\_t mutex\_txt;//对访问txt文件互斥的信号量

pthread\_mutex\_t mute\_ans;//对计算写入结果互斥的信号量

const int print\_num = 5;//定义每行打印多少单词

int \*Count;//统计单篇文章的单词数目

int sum\_num=0;//单词结果的总数

2.程序结构设计（模块图，流程图，函数调用关系图）

主进程实现：

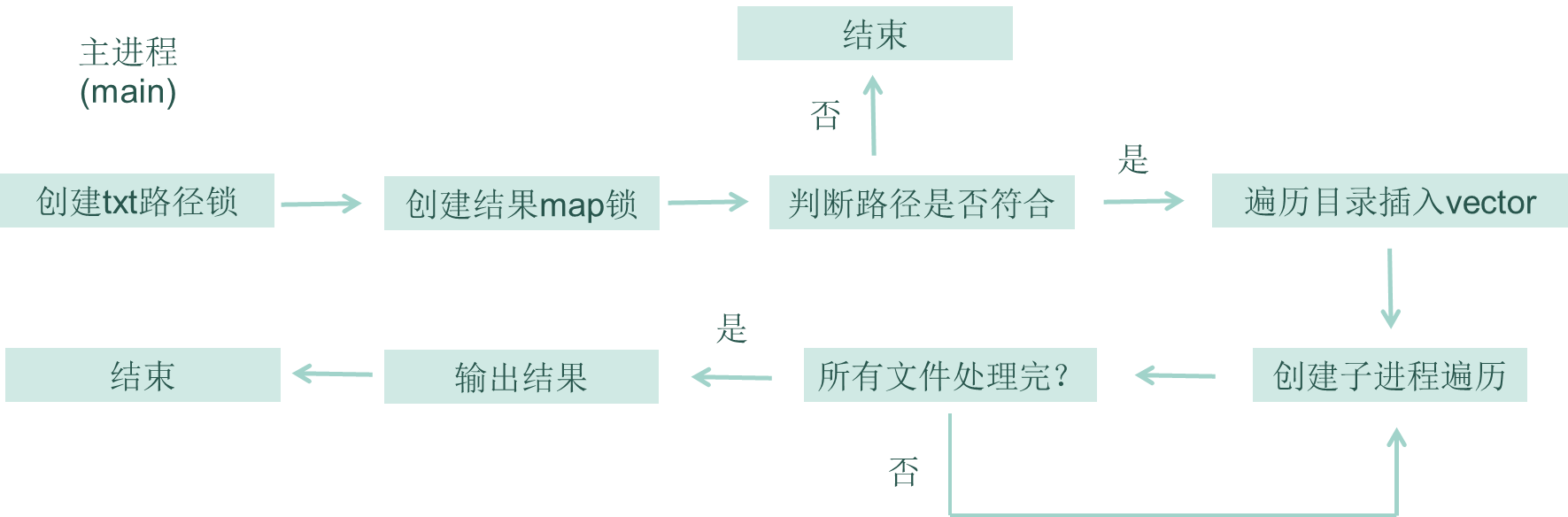


图3.1 主进程实现流程

子线程实现：

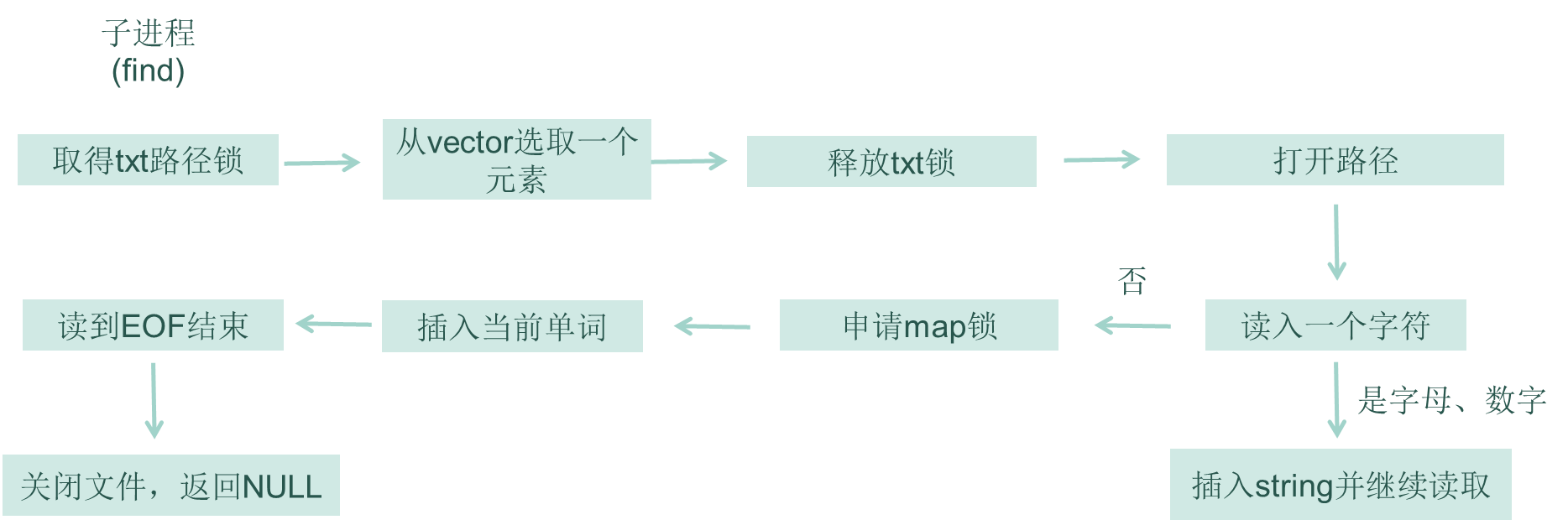


图3.2 子线程实现流程

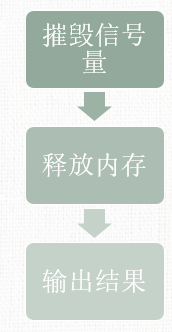
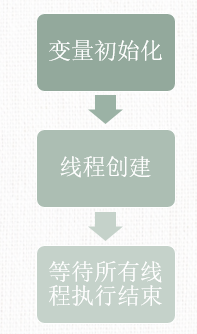
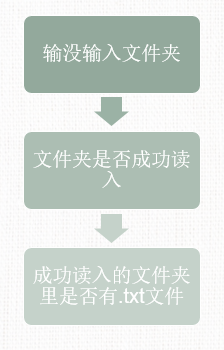


图3.3 主进程中四阶段的流程

阶段一：try-catch-throw进行一系列异常检测

阶段二：多线程进行计数

阶段三：摧毁信号量，释放内存

阶段四：输出统计结果，记录所用时间

3.用到的Linux系统调用函数说明

**（1）opendir函数：**

打开一个目录并建立一个目录流

DIR \*opendir(const char \*name); //参数name 文件夹的名字

如果打开成功的话返回一个DIR结构的指针,该指针用于读取目录数据项。

如果失败的话返回一个空指针

如果文件中的文件过多也可能打开失败

**（2）readdir函数**

struct dirent \*readdir(DIR \*dirp)

函数成功的话返回一个指针，该指针的结构里保存着目录流中下一个目录项的资料，如果错误或者到达目录尾，返回NULL，但不改变error的值，只有发生错误的时候才会设置error;

struct dirent 结构体中有两个元素

ino\_t d\_ino: The inode of the file//文件的inode节点

char d\_name[]: The name of the file//文件名

**（3）chidr函数**

int chdir(const char \*path );

用于改变当前工作目录。调用参数是指向目录的指针，调用进程需要有搜索整个目录的权限。每个进程都具有一个当前工作目录。在解析相对目录引用时，该目录是搜索路径的开始之处。如果调用进程更改了目录，则它只对该进程有效，而不能影响调用它的那个进程。在退出程序时，shell还会返回开始时的那个工作目录。

**（4）lstat函数**

int lstat(const char \*pathname, struct stat \*statbuf);

获取文件属性，比如文件权限，文件属主，文件大小等属性。

第一个参数为传入参数，pathname为绝对路径或相对路径。第二个参数为传出参数，它将文件所具有的属性信息存放在该结构体指针所指的结构体中。接收该属性信息时，需要提前定义一个结构体变量struct stat st，或者结构体指针struct stat\* pst = &st。

**（5）pthread\_mutex\_init函数**

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,const pthread\_mutexattr\_t \*restrict attr);

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

该函数用于C函数的多线程编程中，互斥锁的初始化。以动态方式创建互斥锁的，参数attr指定了新建互斥锁的属性。如果参数attr为空(NULL)，则使用默认的互斥锁属性，默认属性为快速互斥锁 。互斥锁的属性在创建锁的时候指定，在LinuxThreads实现中仅有一个锁类型属性。

**（6）pthread\_create函数**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr，void \*(\*start\_rtn) (void\*)，void \* arg);

用于普通用户创建线程，成功返回 0，失败返回错误编号

 EAGAIN：表示系统限制创建新的线程，如线程数目过多

 EINVAL：代表线程属性值非法

**（7）pthread\_join函数**

int pthread\_join( pthread\_t thread, void \*\* rval\_ptr);

①调用者将挂起并等待新进程终止

②当新线程调用 pthread\_exit()退出或者 return 时，进程中的其他线程可通过pthread\_join()获得进程的退出状态，成功结束返回值为 0,否则为错误编码。

**（8）pthread\_mutex\_lock函数**

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

对互斥量加锁，如果该互斥量已经被锁住，函数调用失败，返回EBUSY，否则加锁成功返回0，线程不会被阻塞。

**（9）pthread\_mutex\_unlock函数**

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

对互斥量解锁。

四．设计源代码

1.多线程 + 逐字符文件读取(fgetc)

//fgetc+多线程

/\*需要用到的C/C++头文件\*/

#include<iostream>//用于cout和异常输出的cerr

#include<cstdio>

#include<fstream>//文件流读取

#include<map>//STL标准map库模拟平衡二叉树

#include<vector>//向量存储.txt文件的名称

#include<cstring>//使用了string流和标准字符串

#include<ctime>//统计时间

#include<exception>//便于C++的try-catch-throw的异常检测

#include<algorithm>//最后使map按照value排序

#include<set>//集合用于存储意义不大的介词、冠词等单词

using namespace std;

/\*用到的linux系统内核/系统调用函数头文件\*/

#include<cstdlib>//使用了malloc申请全局内存，以及调用了信号量互斥访问的函数

#include<unistd.h>//系统调用API接口用于获取当前目录

#include<pthread.h>//线程库函数

#include<dirent.h>//用于遍历linux系统中当前路径文件

#include<sys/stat.h>//定义linux文件状态

/\*一些全局变量的初始化\*/

const int MAX=1000;//定义最大路径名称长度和单词的最大长度

char dir[MAX];//存放.txt文件的文件夹名

int thread\_num=0;//线程的个数，也就是文件夹里.txt文件的数目

vector<string> txtfiles;//保存文件的路径

map<string,int>mp;//创建结果的map

set<string> st;//存储意义不大的介词、冠词等单词

pthread\_mutex\_t mutex\_txt;//对访问txt文件互斥的信号量

pthread\_mutex\_t mute\_ans;//对计算写入结果互斥的信号量

const int print\_num = 5;//定义每行打印多少单词

int \*Count;//统计单篇文章的单词数目

int sum\_num=0;//单词结果的总数

/\*一些需要用到的函数声明\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2);//排序所用的比较函数

int scandir(char \*);//扫描文件夹搜索出所有的txt文件

bool Istxt(char \*);//判断是否是txt文件

bool Isword(char ch);//判断是否是能组成单词的字符

void \*find(void \*id);//线程运行的函数

void Print();//打印结果

void InitWord();//词库初始化

bool Ismeaningful(string ch);//检查该单词是否是有意义的

int main(int argc,char\* argv[]){

    /\*----------阶段一：try-catch-throw进行一系列异常检测----------\*/

    /\*检测是否正确传入参数：程序+文件夹\*/

    try{

        if(argc!=2) throw argc;

        strcpy(dir,argv[1]);

    }

    catch(int){

        cerr<<"检测到输入参数有异常"<<endl

            <<"正确使用：./xx程序 ./xx文件夹"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*检验当前文件夹是否能成功读入\*/

    DIR \*tem;

    try{

        if((tem=opendir(dir))==NULL) throw 1;

        scandir(dir);//扫描文件夹读取.txt文件进入vector

    }

    catch(int){

        cerr<<"文件读取失败或文件夹不存在"<<endl

            <<"请检查正确文件夹姓名"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*检验当前扫描的文件夹里是否有可读入的.txt文件\*/

    try{

        if(txtfiles.size()==0) throw txtfiles.size();

    }

    catch(int){

        cerr<<"没有可读入的.txt文件"<<endl

            <<"请检查文件夹中的文件内容是否准确"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*----------阶段二：多线程进行计数----------\*/

    /\*信号量初始化+时间初始化+无意义词库初始化\*/

    while(pthread\_mutex\_init(&mutex\_txt,NULL)!=0);//信号量初始化

    while(pthread\_mutex\_init(&mute\_ans,NULL)!=0);//信号量初始化

    InitWord();//词库初始化

    int start,end;//统计运行时间

    start = clock();//开始计时

    /\*为线程开辟空间\*/

    pthread\_t \*tid=(pthread\_t\*)malloc((thread\_num+1)\*sizeof(pthread\_t));//为线程分配空间，thread\_num个线程就分配thread\_num块

    int\* id=(int\*)malloc((thread\_num+1)\*sizeof(int));//存储子进程的id

    Count=(int\*)malloc((thread\_num+1)\*sizeof(int));//为统计单词数开辟空间

    for(int i=1;i<=thread\_num;i++) Count[i]=0;

    /\*遍历txt文件创建线程\*/

    while(txtfiles.size()){

        /\*创建线程\*/

        for(int i=1;i<=thread\_num;i++){

            id[i]=i;//记录线程id

            if(pthread\_create(&tid[i],NULL,find,&id[i])==-1)

                printf("线程创建失败\n");

        }

        /\*等待所有线程均执行结束\*/

        for(int i=1;i<=thread\_num;i++)

            pthread\_join(tid[i],NULL);

    }

    /\*----------阶段三：摧毁信号量，释放内存----------\*/

    pthread\_mutex\_destroy(&mutex\_txt);

    pthread\_mutex\_destroy(&mute\_ans);

    free(tid);  //释放内存

    free(id);   //释放内存

    /\*----------阶段四：输出统计结果，记录所用时间----------\*/

    Print();//输出计算结果

    end=clock();//结束计时

    printf("多线程运行时间为: %d 微秒\n", end-start);//输出运行时间

    return 0;

}

/\*自定义cmp用于后期排序使用\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2){

    if(p1.second==p2.second)

        return p1.first<p2.first;

    return p1.second>p2.second;

}

/\*扫描文件夹\*/

int scandir(char \*dir){

    DIR \*current\_dir;

    struct dirent \*catalogue;//当前所处的目录结点

    struct stat buffer\_status;//暂存文件信息状态

    current\_dir=opendir(dir);//打开文件夹

    int ignore\_a=chdir(dir);//将当前的工作目录转换到dir存储的文件夹下

    /\*不断读取目录中的信息直到没有文件/目录可以遍历（ls）\*/

    while((catalogue=readdir(current\_dir))!=NULL){

        lstat(catalogue->d\_name,&buffer\_status);//获取文件信息

        /\*如果当前读到的是目录（文件夹）\*/

        if(S\_ISDIR(buffer\_status.st\_mode)){

            /\*如果目前访问目录是文件夹本身(".")或目录即将是上一级("..")\*/

            if((!strcmp(".", catalogue->d\_name) || (!strcmp("..",catalogue->d\_name))))

                continue;//跳过这条目录

            /\*目前扫描到目录是非自身的文件夹\*/

            else

                scandir(catalogue->d\_name);//递归重新进行新文件夹的扫描

        }

        /\*当前不是文件夹（是单个文件）\*/

        else{

            /\*如果是txt文件就读入\*/

            if(Istxt(catalogue->d\_name)){

                char current\_path[MAX];//存储目前的路径

                /\*写出将txt文件的完整路径\*/

                char \*ignore\_c = getcwd(current\_path,sizeof(current\_path));//写入目前的绝对路径

                strcat(current\_path,"/");

                strcat(current\_path,catalogue->d\_name);//将目录里的txt文件名插入到目前路径形成完整的txt绝对路径

                /\*将路径写入vector\*/

                string s(current\_path); //将current\_path转换为字符串

                txtfiles.push\_back(s);

                thread\_num++;//是txt文件就+1

            }

        }

    }

    chdir("..");//返回上一级

    closedir(current\_dir);//关闭文件夹

    return 0;

}

/\*判断是否为txt文件\*/

bool Istxt(char \*ch){

    int len=strlen(ch);

    /\*如果最后四个字符时.txt说明是txt文件\*/

    if(strcmp(&ch[len-4],".txt")==0)

        return true;    //是txt文件

    /\*不是txt文件的话默认为false\*/

    return false;

}

/\*判断是否是能组成单词的字符\*/

bool Isword(char ch){

    if((ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z'))//只有大小写字母才可以组成单词

        return true;

    return false;

}

void InitWord(){

    ifstream fp("words.txt");//打开文件

    char s[100];

    while(!fp.eof()){

        fp.getline(s,100);

        st.insert(s);

    }

}

bool Ismeaningful(string ch)

{

    /\*如果单词不在集合里，说明是有意义的\*/

    if(st.count(ch)==0) return true;

    return false;

}

/\*核心函数：统计单词个数的find函数\*/

void \*find(void \*id){

    /\*容器没有元素了说明遍历完了\*/

    if(!txtfiles.size())

        return NULL;

    int num=\*(int\*)id;

    printf("线程开始执行...,它的id为: %d\n",num);

    /\*利用信号量进行加锁保证互斥\*/

    while(pthread\_mutex\_lock(&mutex\_txt)!=0);//P操作

    string s = txtfiles[txtfiles.size()-1];

    txtfiles.pop\_back();//路径已经被选择，为了防止被重复取则移出容器

    pthread\_mutex\_unlock(&mutex\_txt);//V操作

    /\*打开相应的txt文件\*/

    char current\_path[MAX];

    strcpy(current\_path, s.c\_str());//转为字符串

    FILE \*p=fopen(current\_path,"r");//打开文件

    /\*开始读入字符\*/

    string word="";//记录单词

    char c;

    c=fgetc(p);//从文件里逐字读入字符

    while(c!=EOF){

        if(Isword(c))

            word+=c;

        else{

            /\*如果当前单词串为空,直接读入字符但是不插入word（因为不是字母）\*/

            if(word==""){

                c=fgetc(p);

                continue;

            }

            /\*统计单词的时候需要信号量互斥操作\*/

            while(pthread\_mutex\_lock(&mutex\_txt)!=0);//P操作

            mp[word]++;

            sum\_num++;

            word="";//重置单词

            pthread\_mutex\_unlock(&mutex\_txt);//V操作

            Count[num]++;//单篇单词数+1

        }

        c=fgetc(p);

    }

    fclose(p);//读完这个文件了就关闭文件

    printf("id为 %d 的线程统计结束,单词总数为%d\n",num,Count[num]);

    return NULL;

}

/\*输出统计结果\*/

void Print(){

    printf("单词总数 : %d\n",sum\_num);

    /\*默认的map是按key排序的,输出之前换用value排序\*/

    vector<pair<string,int>> arr;

   for(map<string,int>::iterator it=mp.begin();it!=mp.end();++it)

    arr.push\_back(make\_pair(it->first,it->second));

   sort(arr.begin(),arr.end(),cmp);//对vector排序

    /\*遍历vector等效作为遍历map容器\*/

    printf("现展示出现频率top20的单词如下:\n");

    FILE \*fp;

    fp=fopen("res1.txt","w");

    int i,k=1;

    for(vector<pair<string,int>>::iterator it=arr.begin();it!=arr.end();++it){

        i++;

        if(k<=20){

            if(Ismeaningful(it->first)){

                printf("%04d : %s ", it->second, (it->first).c\_str());

                for(int j = 20 - (it->first).size();j > 0;j--) putchar(' ');

                if(k % print\_num == 0) puts("");

                k++;

            }

        }

        fprintf(fp,"%05d : %s ", it->second, (it->first).c\_str());

        for(int j = 20 - (it->first).size();j > 0;j--) fputs(" ",fp);

        if(i % print\_num == 0) fputs("\n",fp);

    }

    if(i % print\_num != 0) fputs("\n",fp);

    printf("具体所有单词统计结果请查看本目录下'res1.txt'文件\n");

}

2.单线程 + 逐字符文件读取(fgetc)

//fgetc+多线程

/\*需要用到的C/C++头文件\*/

#include<iostream>//用于cout和异常输出的cerr

#include<cstdio>

#include<fstream>//文件流读取

#include<map>//STL标准map库模拟平衡二叉树

#include<vector>//向量存储.txt文件的名称

#include<cstring>//使用了string流和标准字符串

#include<ctime>//统计时间

#include<exception>//便于C++的try-catch-throw的异常检测

#include<algorithm>//最后使map按照value排序

#include<set>//集合用于存储意义不大的介词、冠词等单词

#include<cstdlib>//使用了malloc申请全局内存

using namespace std;

/\*用到的linux系统内核/系统调用函数头文件\*/

#include<unistd.h>//系统调用API接口用于获取当前目录

#include<dirent.h>//用于遍历linux系统中当前路径文件

#include<sys/stat.h>//定义linux文件状态

/\*一些全局变量的初始化\*/

const int MAX=1000;//定义最大路径名称长度和单词的最大长度

char dir[MAX];//存放.txt文件的文件夹名

int thread\_num=0;//文件夹里.txt文件的数目

vector<string> txtfiles;//保存文件的路径

map<string,int>mp;//创建结果的map

set<string> st;//存储意义不大的介词、冠词等单词

pthread\_mutex\_t mutex\_txt;//对访问txt文件互斥的信号量

pthread\_mutex\_t mute\_ans;//对计算写入结果互斥的信号量

const int print\_num = 5;//定义每行打印多少单词

int \*Count;//统计单篇文章的单词数目

int sum\_num=0;//单词结果的总数

/\*一些需要用到的函数声明\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2);//排序所用的比较函数

int scandir(char \*);//扫描文件夹搜索出所有的txt文件

bool Istxt(char \*);//判断是否是txt文件

bool Isword(char ch);//判断是否是能组成单词的字符

void find(int id);//统计文章单词的函数

void Print();//打印结果

void InitWord();//词库初始化

bool Ismeaningful(string ch);//检查该单词是否是有意义的

int main(int argc,char\* argv[]){

    /\*----------阶段一：try-catch-throw进行一系列异常检测----------\*/

    /\*检测是否正确传入参数：程序+文件夹\*/

    try{

        if(argc!=2) throw argc;

        strcpy(dir,argv[1]);

    }

    catch(int){

        cerr<<"检测到输入参数有异常"<<endl

            <<"正确使用：./xx程序 ./xx文件夹"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*检验当前文件夹是否能成功读入\*/

    DIR \*tem;

    try{

        if((tem=opendir(dir))==NULL) throw 1;

        scandir(dir);//扫描文件夹读取.txt文件进入vector

    }

    catch(int){

        cerr<<"文件读取失败或文件夹不存在"<<endl

            <<"请检查正确文件夹姓名"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*检验当前扫描的文件夹里是否有可读入的.txt文件\*/

    try{

        if(txtfiles.size()==0) throw txtfiles.size();

    }

    catch(int){

        cerr<<"没有可读入的.txt文件"<<endl

            <<"请检查文件夹中的文件内容是否准确"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*----------阶段二：单进程while循环进行计数----------\*/

    /\*时间初始化+无意义词库初始化\*/

    InitWord();//词库初始化

    int start,end;//统计运行时间

    start = clock();//开始计时

    /\*为统计单篇文章开辟空间\*/

    Count=(int\*)malloc((thread\_num+1)\*sizeof(int));//为统计单词数开辟空间

    for(int i=1;i<=thread\_num;i++) Count[i]=0;

    /\*遍历txt文件反复循环\*/

    int Id=1;

    while(txtfiles.size()){

        find(Id);

        Id++;

    }

    /\*----------阶段三：输出统计结果，记录所用时间----------\*/

    Print();//输出计算结果

    end=clock();//结束计时

    printf("单进程运行时间为: %d 微秒\n", end-start);//输出运行时间

    return 0;

}

/\*自定义cmp用于后期排序使用\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2){

    if(p1.second==p2.second)

        return p1.first<p2.first;

    return p1.second>p2.second;

}

/\*扫描文件夹\*/

int scandir(char \*dir){

    DIR \*current\_dir;

    struct dirent \*catalogue;//当前所处的目录结点

    struct stat buffer\_status;//暂存文件信息状态

    current\_dir=opendir(dir);//打开文件夹

    int ignore\_a=chdir(dir);//将当前的工作目录转换到dir存储的文件夹下

    /\*不断读取目录中的信息直到没有文件/目录可以遍历（ls）\*/

    while((catalogue=readdir(current\_dir))!=NULL){

        lstat(catalogue->d\_name,&buffer\_status);//获取文件信息

        /\*如果当前读到的是目录（文件夹）\*/

        if(S\_ISDIR(buffer\_status.st\_mode)){

            /\*如果目前访问目录是文件夹本身(".")或目录即将是上一级("..")\*/

            if((!strcmp(".", catalogue->d\_name) || (!strcmp("..",catalogue->d\_name))))

                continue;//跳过这条目录

            /\*目前扫描到目录是非自身的文件夹\*/

            else

                scandir(catalogue->d\_name);//递归重新进行新文件夹的扫描

        }

        /\*当前不是文件夹（是单个文件）\*/

        else{

            /\*如果是txt文件就读入\*/

            if(Istxt(catalogue->d\_name)){

                char current\_path[MAX];//存储目前的路径

                /\*写出将txt文件的完整路径\*/

                char \*ignore\_c = getcwd(current\_path,sizeof(current\_path));//写入目前的绝对路径

                strcat(current\_path,"/");

                strcat(current\_path,catalogue->d\_name);//将目录里的txt文件名插入到目前路径形成完整的txt绝对路径

                /\*将路径写入vector\*/

                string s(current\_path); //将current\_path转换为字符串

                txtfiles.push\_back(s);

                thread\_num++;//是txt文件就+1

            }

        }

    }

    chdir("..");//返回上一级

    closedir(current\_dir);//关闭文件夹

    return 0;

}

/\*判断是否为txt文件\*/

bool Istxt(char \*ch){

    int len=strlen(ch);

    /\*如果最后四个字符时.txt说明是txt文件\*/

    if(strcmp(&ch[len-4],".txt")==0)

        return true;    //是txt文件

    /\*不是txt文件的话默认为false\*/

    return false;

}

/\*判断是否是能组成单词的字符\*/

bool Isword(char ch){

    if((ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z'))//只有大小写字母才可以组成单词

        return true;

    return false;

}

void InitWord(){

    ifstream fp("words.txt");//打开文件

    char s[100];

    while(!fp.eof()){

        fp.getline(s,100);

        st.insert(s);

    }

}

bool Ismeaningful(string ch)

{

    /\*如果单词不在集合里，说明是有意义的\*/

    if(st.count(ch)==0) return true;

    return false;

}

/\*核心函数：统计单词个数的find函数\*/

void find(int id){

    /\*容器没有元素了说明遍历完了\*/

    if(!txtfiles.size())

        return ;

    int num=id;

    printf("第 %d 篇文章开始统计...\n",num);

    /\*扫描路径\*/

    string s = txtfiles[txtfiles.size()-1];

    txtfiles.pop\_back();//路径已经被选择，为了防止被重复取则移出容器

    /\*打开相应的txt文件\*/

    char current\_path[MAX];

    strcpy(current\_path, s.c\_str());//转为字符串

    FILE \*p=fopen(current\_path,"r");//打开文件

    /\*开始读入字符\*/

    string word="";//记录单词

    char c;

    c=fgetc(p);//从文件里逐字读入字符

    while(c!=EOF){

        if(Isword(c))

            word+=c;

        else{

            /\*如果当前单词串为空,直接读入字符但是不插入word（因为不是字母）\*/

                if(word==""){

                    c=fgetc(p);

                    continue;

                }

            mp[word]++;

            sum\_num++;

            word="";//重置单词

            Count[num]++;//单篇单词数+1

        }

        c=fgetc(p);

    }

    fclose(p);//读完这个文件了就关闭文件

    printf("第 %d 篇文章统计结束,单词总数为%d\n",num,Count[num]);

}

/\*输出统计结果\*/

void Print(){

    printf("单词总数 : %d\n",sum\_num);

    /\*默认的map是按key排序的,输出之前换用value排序\*/

    vector<pair<string,int>> arr;

   for(map<string,int>::iterator it=mp.begin();it!=mp.end();++it)

    arr.push\_back(make\_pair(it->first,it->second));

   sort(arr.begin(),arr.end(),cmp);//对vector排序

    /\*遍历vector等效作为遍历map容器\*/

    printf("现展示出现频率top20的单词如下:\n");

    FILE \*fp;

    fp=fopen("res2.txt","w");

    int i,k=1;

    for(vector<pair<string,int>>::iterator it=arr.begin();it!=arr.end();++it){

        i++;

        if(k<=20){

            if(Ismeaningful(it->first)){

                printf("%04d : %s ", it->second, (it->first).c\_str());

                for(int j = 20 - (it->first).size();j > 0;j--) putchar(' ');

                if(k % print\_num == 0) puts("");

                k++;

            }

        }

        fprintf(fp,"%05d : %s ", it->second, (it->first).c\_str());

        for(int j = 20 - (it->first).size();j > 0;j--) fputs(" ",fp);

        if(i % print\_num == 0) fputs("\n",fp);

    }

    if(i % print\_num != 0) fputs("\n",fp);

    printf("具体所有单词统计结果请查看本目录下'res2.txt'文件\n");

}

3.多线程 + 数据块文件读取(fread)

//fgetc+多线程

/\*需要用到的C/C++头文件\*/

#include<iostream>//用于cout和异常输出的cerr

#include<cstdio>

#include<fstream>//文件流读取

#include<map>//STL标准map库模拟平衡二叉树

#include<vector>//向量存储.txt文件的名称

#include<cstring>//使用了string流和标准字符串

#include<ctime>//统计时间

#include<exception>//便于C++的try-catch-throw的异常检测

#include<algorithm>//最后使map按照value排序

#include<set>//集合用于存储意义不大的介词、冠词等单词

using namespace std;

/\*用到的linux系统内核/系统调用函数头文件\*/

#include<cstdlib>//使用了malloc申请全局内存，以及调用了信号量互斥访问的函数

#include<unistd.h>//系统调用API接口用于获取当前目录

#include<pthread.h>//线程库函数

#include<dirent.h>//用于遍历linux系统中当前路径文件

#include<sys/stat.h>//定义linux文件状态

/\*一些全局变量的初始化\*/

const int MAX=1000;//定义最大路径名称长度和单词的最大长度

char dir[MAX];//存放.txt文件的文件夹名

int thread\_num=0;//线程的个数，也就是文件夹里.txt文件的数目

vector<string> txtfiles;//保存文件的路径

map<string,int>mp;//创建结果的map

set<string> st;//存储意义不大的介词、冠词等单词

pthread\_mutex\_t mutex\_txt;//对访问txt文件互斥的信号量

pthread\_mutex\_t mute\_ans;//对计算写入结果互斥的信号量

const int print\_num = 5;//定义每行打印多少单词

int \*Count;//统计单篇文章的单词数目

int sum\_num=0;//单词结果的总数

/\*一些需要用到的函数声明\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2);//排序所用的比较函数

int scandir(char \*);//扫描文件夹搜索出所有的txt文件

bool Istxt(char \*);//判断是否是txt文件

bool Isword(char ch);//判断是否是能组成单词的字符

void \*find(void \*id);//线程运行的函数

void Print();//打印结果

void InitWord();//词库初始化

bool Ismeaningful(string ch);//检查该单词是否是有意义的

int main(int argc,char\* argv[]){

    /\*----------阶段一：try-catch-throw进行一系列异常检测----------\*/

    /\*检测是否正确传入参数：程序+文件夹\*/

    try{

        if(argc!=2) throw argc;

        strcpy(dir,argv[1]);

    }

    catch(int){

        cerr<<"检测到输入参数有异常"<<endl

            <<"正确使用：./xx程序 ./xx文件夹"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*检验当前文件夹是否能成功读入\*/

    DIR \*tem;

    try{

        if((tem=opendir(dir))==NULL) throw 1;

        scandir(dir);//扫描文件夹读取.txt文件进入vector

    }

    catch(int){

        cerr<<"文件读取失败或文件夹不存在"<<endl

            <<"请检查正确文件夹姓名"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*检验当前扫描的文件夹里是否有可读入的.txt文件\*/

    try{

        if(txtfiles.size()==0) throw txtfiles.size();

    }

    catch(int){

        cerr<<"没有可读入的.txt文件"<<endl

            <<"请检查文件夹中的文件内容是否准确"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*----------阶段二：多线程进行计数----------\*/

    /\*信号量初始化+时间初始化+无意义词库初始化\*/

    while(pthread\_mutex\_init(&mutex\_txt,NULL)!=0);//信号量初始化

    while(pthread\_mutex\_init(&mute\_ans,NULL)!=0);//信号量初始化

    InitWord();//词库初始化

    int start,end;//统计运行时间

    start = clock();//开始计时

    /\*为线程开辟空间\*/

    pthread\_t \*tid=(pthread\_t\*)malloc((thread\_num+1)\*sizeof(pthread\_t));//为线程分配空间，thread\_num个线程就分配thread\_num块

    int\* id=(int\*)malloc((thread\_num+1)\*sizeof(int));//存储子进程的id

    Count=(int\*)malloc((thread\_num+1)\*sizeof(int));//为统计单词数开辟空间

    for(int i=1;i<=thread\_num;i++) Count[i]=0;

    /\*遍历txt文件创建线程\*/

    while(txtfiles.size()){

        /\*创建线程\*/

        for(int i=1;i<=thread\_num;i++){

            id[i]=i;//记录线程id

            if(pthread\_create(&tid[i],NULL,find,&id[i])==-1)

                printf("线程创建失败\n");

        }

        /\*等待所有线程均执行结束\*/

        for(int i=1;i<=thread\_num;i++)

            pthread\_join(tid[i],NULL);

    }

    /\*----------阶段三：摧毁信号量，释放内存----------\*/

    pthread\_mutex\_destroy(&mutex\_txt);

    pthread\_mutex\_destroy(&mute\_ans);

    free(tid);  //释放内存

    free(id);   //释放内存

    /\*----------阶段四：输出统计结果，记录所用时间----------\*/

    Print();//输出计算结果

    end=clock();//结束计时

    printf("多线程运行时间为: %d 微秒\n", end-start);//输出运行时间

    return 0;

}

/\*自定义cmp用于后期排序使用\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2){

    if(p1.second==p2.second)

        return p1.first<p2.first;

    return p1.second>p2.second;

}

/\*扫描文件夹\*/

int scandir(char \*dir){

    DIR \*current\_dir;

    struct dirent \*catalogue;//当前所处的目录结点

    struct stat buffer\_status;//暂存文件信息状态

    current\_dir=opendir(dir);//打开文件夹

    int ignore\_a=chdir(dir);//将当前的工作目录转换到dir存储的文件夹下

    /\*不断读取目录中的信息直到没有文件/目录可以遍历（ls）\*/

    while((catalogue=readdir(current\_dir))!=NULL){

        lstat(catalogue->d\_name,&buffer\_status);//获取文件信息

        /\*如果当前读到的是目录（文件夹）\*/

        if(S\_ISDIR(buffer\_status.st\_mode)){

            /\*如果目前访问目录是文件夹本身(".")或目录即将是上一级("..")\*/

            if((!strcmp(".", catalogue->d\_name) || (!strcmp("..",catalogue->d\_name))))

                continue;//跳过这条目录

            /\*目前扫描到目录是非自身的文件夹\*/

            else

                scandir(catalogue->d\_name);//递归重新进行新文件夹的扫描

        }

        /\*当前不是文件夹（是单个文件）\*/

        else{

            /\*如果是txt文件就读入\*/

            if(Istxt(catalogue->d\_name)){

                char current\_path[MAX];//存储目前的路径

                /\*写出将txt文件的完整路径\*/

                char \*ignore\_c = getcwd(current\_path,sizeof(current\_path));//写入目前的绝对路径

                strcat(current\_path,"/");

                strcat(current\_path,catalogue->d\_name);//将目录里的txt文件名插入到目前路径形成完整的txt绝对路径

                /\*将路径写入vector\*/

                string s(current\_path); //将current\_path转换为字符串

                txtfiles.push\_back(s);

                thread\_num++;//是txt文件就+1

            }

        }

    }

    chdir("..");//返回上一级

    closedir(current\_dir);//关闭文件夹

    return 0;

}

/\*判断是否为txt文件\*/

bool Istxt(char \*ch){

    int len=strlen(ch);

    /\*如果最后四个字符时.txt说明是txt文件\*/

    if(strcmp(&ch[len-4],".txt")==0)

        return true;    //是txt文件

    /\*不是txt文件的话默认为false\*/

    return false;

}

/\*判断是否是能组成单词的字符\*/

bool Isword(char ch){

    if((ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z'))//只有大小写字母才可以组成单词

        return true;

    return false;

}

void InitWord(){

    ifstream fp("words.txt");//打开文件

    char s[100];

    while(!fp.eof()){

        fp.getline(s,100);

        st.insert(s);

    }

}

bool Ismeaningful(string ch)

{

    /\*如果单词不在集合里，说明是有意义的\*/

    if(st.count(ch)==0) return true;

    return false;

}

/\*核心函数：统计单词个数的find函数\*/

void \*find(void \*id){

    /\*容器没有元素了说明遍历完了\*/

    if(!txtfiles.size())

        return NULL;

    int num=\*(int\*)id;

    printf("线程开始执行...,它的id为: %d\n",num);

    /\*利用信号量进行加锁保证互斥\*/

    while(pthread\_mutex\_lock(&mutex\_txt)!=0);//P操作

    string s = txtfiles[txtfiles.size()-1];

    txtfiles.pop\_back();//路径已经被选择，为了防止被重复取则移出容器

    pthread\_mutex\_unlock(&mutex\_txt);//V操作

    /\*打开相应的txt文件\*/

    char current\_path[MAX];

    strcpy(current\_path, s.c\_str());//转为字符串

    FILE \*p=fopen(current\_path,"r");//打开文件

    /\*获取文件的大小\*/

    fseek(p,0,SEEK\_END);

    long lSize=ftell(p);

    rewind(p);

    /\*分配能完整读取文件的缓冲区大小\*/

    char \*buffer=(char\*)malloc(lSize\*sizeof(char));

    /\*开始用fread将字符全部读入缓冲区\*/

    fread(buffer,sizeof(char),lSize,p);

    fclose(p);//读完这个文件了就关闭文件

    /\*开始读入字符\*/

    string word="";//记录单词

    char c;

    for(int ii=0;buffer[ii]!='\0';ii++){

        if(Isword(buffer[ii]))

            word+=buffer[ii];

        else{

            /\*如果当前单词串为空,直接读入字符但是不插入word（因为不是字母）\*/

            if(word==""){

                continue;

            }

            /\*统计单词的时候需要信号量互斥操作\*/

            while(pthread\_mutex\_lock(&mutex\_txt)!=0);//P操作

            mp[word]++;

            sum\_num++;

            word="";//重置单词

            pthread\_mutex\_unlock(&mutex\_txt);//V操作

            Count[num]++;//单篇单词数+1

        }

    }

    // fclose(p);//读完这个文件了就关闭文件

    printf("id为 %d 的线程统计结束,单词总数为%d\n",num,Count[num]);

    return NULL;

}

/\*输出统计结果\*/

void Print(){

    printf("单词总数 : %d\n",sum\_num);

    /\*默认的map是按key排序的,输出之前换用value排序\*/

    vector<pair<string,int>> arr;

   for(map<string,int>::iterator it=mp.begin();it!=mp.end();++it)

    arr.push\_back(make\_pair(it->first,it->second));

   sort(arr.begin(),arr.end(),cmp);//对vector排序

    /\*遍历vector等效作为遍历map容器\*/

    printf("现展示出现频率top20的单词如下:\n");

    FILE \*fp;

    fp=fopen("res3.txt","w");

    int i,k=1;

    for(vector<pair<string,int>>::iterator it=arr.begin();it!=arr.end();++it){

        i++;

        if(k<=20){

            if(Ismeaningful(it->first)){

                printf("%04d : %s ", it->second, (it->first).c\_str());

                for(int j = 20 - (it->first).size();j > 0;j--) putchar(' ');

                if(k % print\_num == 0) puts("");

                k++;

            }

        }

        fprintf(fp,"%05d : %s ", it->second, (it->first).c\_str());

        for(int j = 20 - (it->first).size();j > 0;j--) fputs(" ",fp);

        if(i % print\_num == 0) fputs("\n",fp);

    }

    if(i % print\_num != 0) fputs("\n",fp);

    printf("具体所有单词统计结果请查看本目录下'res3.txt'文件\n");

}

4.单线程 + 数据块文件读取(fread)

//fgetc+多线程

/\*需要用到的C/C++头文件\*/

#include<iostream>//用于cout和异常输出的cerr

#include<cstdio>

#include<fstream>//文件流读取

#include<sstream>// istringstream

#include<map>//STL标准map库模拟平衡二叉树

#include<vector>//向量存储.txt文件的名称

#include<cstring>//使用了string流和标准字符串

#include<ctime>//统计时间

#include<exception>//便于C++的try-catch-throw的异常检测

#include<algorithm>//最后使map按照value排序

#include<set>//集合用于存储意义不大的介词、冠词等单词

#include<cstdlib>//使用了malloc申请全局内存

using namespace std;

/\*用到的linux系统内核/系统调用函数头文件\*/

#include<unistd.h>//系统调用API接口用于获取当前目录

#include<dirent.h>//用于遍历linux系统中当前路径文件

#include<sys/stat.h>//定义linux文件状态

/\*一些全局变量的初始化\*/

const int MAX=1000;//定义最大路径名称长度和单词的最大长度

char dir[MAX];//存放.txt文件的文件夹名

int thread\_num=0;//文件夹里.txt文件的数目

vector<string> txtfiles;//保存文件的路径

map<string,int>mp;//创建结果的map

set<string> st;//存储意义不大的介词、冠词等单词

pthread\_mutex\_t mutex\_txt;//对访问txt文件互斥的信号量

pthread\_mutex\_t mute\_ans;//对计算写入结果互斥的信号量

const int print\_num = 5;//定义每行打印多少单词

int \*Count;//统计单篇文章的单词数目

int sum\_num=0;//单词结果的总数

/\*一些需要用到的函数声明\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2);//排序所用的比较函数

int scandir(char \*);//扫描文件夹搜索出所有的txt文件

bool Istxt(char \*);//判断是否是txt文件

bool Isword(char ch);//判断是否是能组成单词的字符

void find(int id);//统计文章单词的函数

void Print();//打印结果

void InitWord();//词库初始化

bool Ismeaningful(string ch);//检查该单词是否是有意义的

int main(int argc,char\* argv[]){

    /\*----------阶段一：try-catch-throw进行一系列异常检测----------\*/

    /\*检测是否正确传入参数：程序+文件夹\*/

    try{

        if(argc!=2) throw argc;

        strcpy(dir,argv[1]);

    }

    catch(int){

        cerr<<"检测到输入参数有异常"<<endl

            <<"正确使用：./xx程序 ./xx文件夹"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*检验当前文件夹是否能成功读入\*/

    DIR \*tem;

    try{

        if((tem=opendir(dir))==NULL) throw 1;

        scandir(dir);//扫描文件夹读取.txt文件进入vector

    }

    catch(int){

        cerr<<"文件读取失败或文件夹不存在"<<endl

            <<"请检查正确文件夹姓名"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*检验当前扫描的文件夹里是否有可读入的.txt文件\*/

    try{

        if(txtfiles.size()==0) throw txtfiles.size();

    }

    catch(int){

        cerr<<"没有可读入的.txt文件"<<endl

            <<"请检查文件夹中的文件内容是否准确"<<endl

            <<"程序正在退出..."<<endl;

        exit(1);

    }

    /\*----------阶段二：单进程while循环进行计数----------\*/

    /\*时间初始化+无意义词库初始化\*/

    InitWord();//词库初始化

    int start,end;//统计运行时间

    start = clock();//开始计时

    /\*为统计单篇文章开辟空间\*/

    Count=(int\*)malloc((thread\_num+1)\*sizeof(int));//为统计单词数开辟空间

    for(int i=1;i<=thread\_num;i++) Count[i]=0;

    /\*遍历txt文件反复循环\*/

    int Id=1;

    while(txtfiles.size()){

        find(Id);

        Id++;

    }

    /\*----------阶段三：输出统计结果，记录所用时间----------\*/

    Print();//输出计算结果

    end=clock();//结束计时

    printf("单进程运行时间为: %d 微秒\n", end-start);//输出运行时间

    return 0;

}

/\*自定义cmp用于后期排序使用\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2){

    if(p1.second==p2.second)

        return p1.first<p2.first;

    return p1.second>p2.second;

}

/\*扫描文件夹\*/

int scandir(char \*dir){

    DIR \*current\_dir;

    struct dirent \*catalogue;//当前所处的目录结点

    struct stat buffer\_status;//暂存文件信息状态

    current\_dir=opendir(dir);//打开文件夹

    int ignore\_a=chdir(dir);//将当前的工作目录转换到dir存储的文件夹下

    /\*不断读取目录中的信息直到没有文件/目录可以遍历（ls）\*/

    while((catalogue=readdir(current\_dir))!=NULL){

        lstat(catalogue->d\_name,&buffer\_status);//获取文件信息

        /\*如果当前读到的是目录（文件夹）\*/

        if(S\_ISDIR(buffer\_status.st\_mode)){

            /\*如果目前访问目录是文件夹本身(".")或目录即将是上一级("..")\*/

            if((!strcmp(".", catalogue->d\_name) || (!strcmp("..",catalogue->d\_name))))

                continue;//跳过这条目录

            /\*目前扫描到目录是非自身的文件夹\*/

            else

                scandir(catalogue->d\_name);//递归重新进行新文件夹的扫描

        }

        /\*当前不是文件夹（是单个文件）\*/

        else{

            /\*如果是txt文件就读入\*/

            if(Istxt(catalogue->d\_name)){

                char current\_path[MAX];//存储目前的路径

                /\*写出将txt文件的完整路径\*/

                char \*ignore\_c = getcwd(current\_path,sizeof(current\_path));//写入目前的绝对路径

                strcat(current\_path,"/");

                strcat(current\_path,catalogue->d\_name);//将目录里的txt文件名插入到目前路径形成完整的txt绝对路径

                /\*将路径写入vector\*/

                string s(current\_path); //将current\_path转换为字符串

                txtfiles.push\_back(s);

                thread\_num++;//是txt文件就+1

            }

        }

    }

    chdir("..");//返回上一级

    closedir(current\_dir);//关闭文件夹

    return 0;

}

/\*判断是否为txt文件\*/

bool Istxt(char \*ch){

    int len=strlen(ch);

    /\*如果最后四个字符时.txt说明是txt文件\*/

    if(strcmp(&ch[len-4],".txt")==0)

        return true;    //是txt文件

    /\*不是txt文件的话默认为false\*/

    return false;

}

/\*判断是否是能组成单词的字符\*/

bool Isword(char ch){

    if((ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z'))//只有大小写字母才可以组成单词

        return true;

    return false;

}

void InitWord(){

    ifstream fp("words.txt");//打开文件

    char s[100];

    while(!fp.eof()){

        fp.getline(s,100);

        st.insert(s);

    }

}

bool Ismeaningful(string ch)

{

    /\*如果单词不在集合里，说明是有意义的\*/

    if(st.count(ch)==0) return true;

    return false;

}

/\*核心函数：统计单词个数的find函数\*/

void find(int id){

    /\*容器没有元素了说明遍历完了\*/

    if(!txtfiles.size())

        return ;

    int num=id;

    printf("第 %d 篇文章开始统计...\n",num);

    /\*选择路径\*/

    string s = txtfiles[txtfiles.size()-1];

    txtfiles.pop\_back();//路径已经被选择，为了防止被重复取则移出容器

    /\*打开相应的txt文件\*/

    char current\_path[MAX];

    strcpy(current\_path, s.c\_str());//转为字符串

    FILE \*p=fopen(current\_path,"r");//打开文件

    /\*获取文件的大小\*/

    fseek(p,0,SEEK\_END);

    long lSize=ftell(p);

    rewind(p);

    /\*分配能完整读取文件的缓冲区大小\*/

    char \*buffer=(char\*)malloc(lSize\*sizeof(char));

    /\*开始用fread将字符全部读入缓冲区\*/

    fread(buffer,sizeof(char),lSize,p);

    fclose(p);//读完这个文件了就关闭文件

    /\*模拟从缓冲区读字符\*/

    char c;

    string word="";

    for(int ii=0;buffer[ii]!='\0';ii++){

        if(Isword(buffer[ii]))

            word+=buffer[ii];

        else{

            /\*如果当前单词串为空,直接读入字符但是不插入word（因为不是字母）\*/

            if(word=="")

                continue;

            mp[word]++;

            sum\_num++;

            word="";//重置单词

            Count[num]++;//单篇单词数+1

        }

    }

    /\*以下注释是另外两种统计单词的方法，一种是string流，一种是遍历字符串迭代器\*/

    // char c;

    // for(string::iterator it=buffer.begin();it!=buffer.end();++it){

    //  c=\*it;//从缓冲区里逐字读入字符

    //  if(Isword(c))

    //      word+=c;

    //  else{

    //      /\*如果当前单词串为空,直接读入字符但是不插入word（因为不是字母）\*/

    //      if(word==""){

    //          continue;

    //      }

    //      mp[word]++;

    //      sum\_num++;

    //      word="";//重置单词

    //      Count[num]++;//单篇单词数+1

    //  }

    // }

    // istringstream input(buffer);

    // vector<string> vec;

    // string temp;

    // while(getline(input,temp,' '))

    //  vec.push\_back(temp);

    // for(string word:vec){

    //  mp[word]++;

    //  sum\_num++;

    //  Count[num]++;

    // }

    printf("第 %d 篇文章统计结束,单词总数为%d\n",num,Count[num]);

}

/\*输出统计结果\*/

void Print(){

    printf("单词总数 : %d\n",sum\_num);

    /\*默认的map是按key排序的,输出之前换用value排序\*/

    vector<pair<string,int>> arr;

   for(map<string,int>::iterator it=mp.begin();it!=mp.end();++it)

    arr.push\_back(make\_pair(it->first,it->second));

   sort(arr.begin(),arr.end(),cmp);//对vector排序

    /\*遍历vector等效作为遍历map容器\*/

    printf("现展示出现频率top20的单词如下:\n");

    FILE \*fp;

    fp=fopen("res4.txt","w");

    int i,k=1;

    for(vector<pair<string,int>>::iterator it=arr.begin();it!=arr.end();++it){

        i++;

        if(k<=20){

            if(Ismeaningful(it->first)){

                printf("%04d : %s ", it->second, (it->first).c\_str());

                for(int j = 20 - (it->first).size();j > 0;j--) putchar(' ');

                if(k % print\_num == 0) puts("");

                k++;

            }

        }

        fprintf(fp,"%05d : %s ", it->second, (it->first).c\_str());

        for(int j = 20 - (it->first).size();j > 0;j--) fputs(" ",fp);

        if(i % print\_num == 0) fputs("\n",fp);

    }

    if(i % print\_num != 0) fputs("\n",fp);

    printf("具体所有单词统计结果请查看本目录下'res4.txt'文件\n");

}

本质上说，代码的区别仅在于主函数的第二阶段以及find函数中。



图4.1 多线程方式代码特征

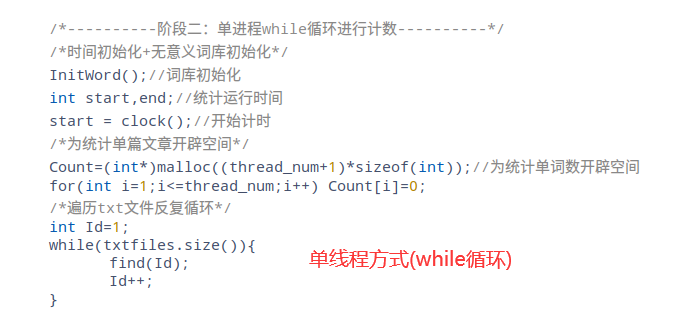


图4.2 单线程方式代码特征



图4.3 逐字符读取文件方式

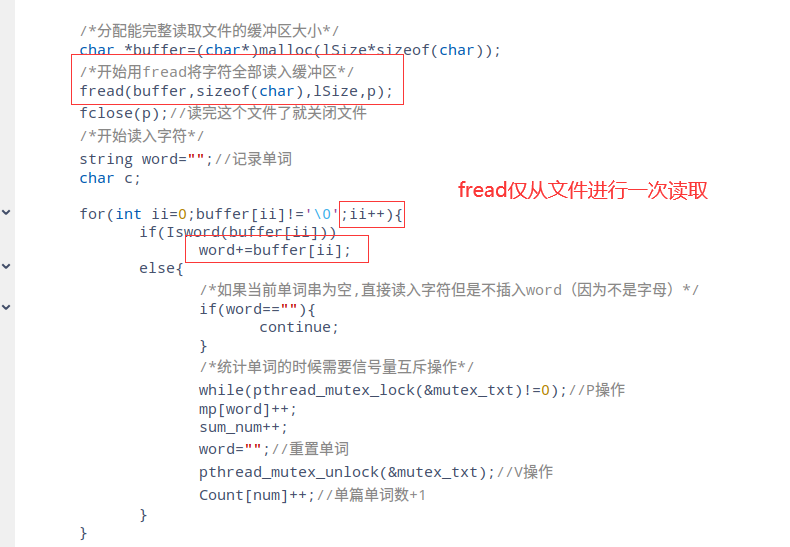


图4.4 数据块读取文件方式

五．调试方案及调试过程问题分析

四个代码文件的运行方式均书写了一份操作手册：

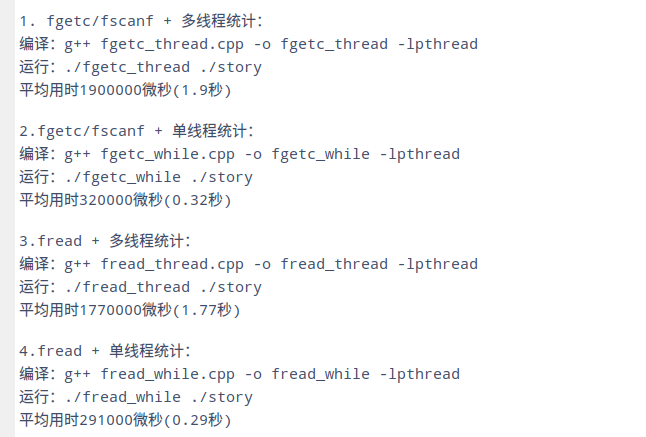


图5.1 代码操作手册

六．运行结果与分析

1.测试用例及各功能运行结果截图

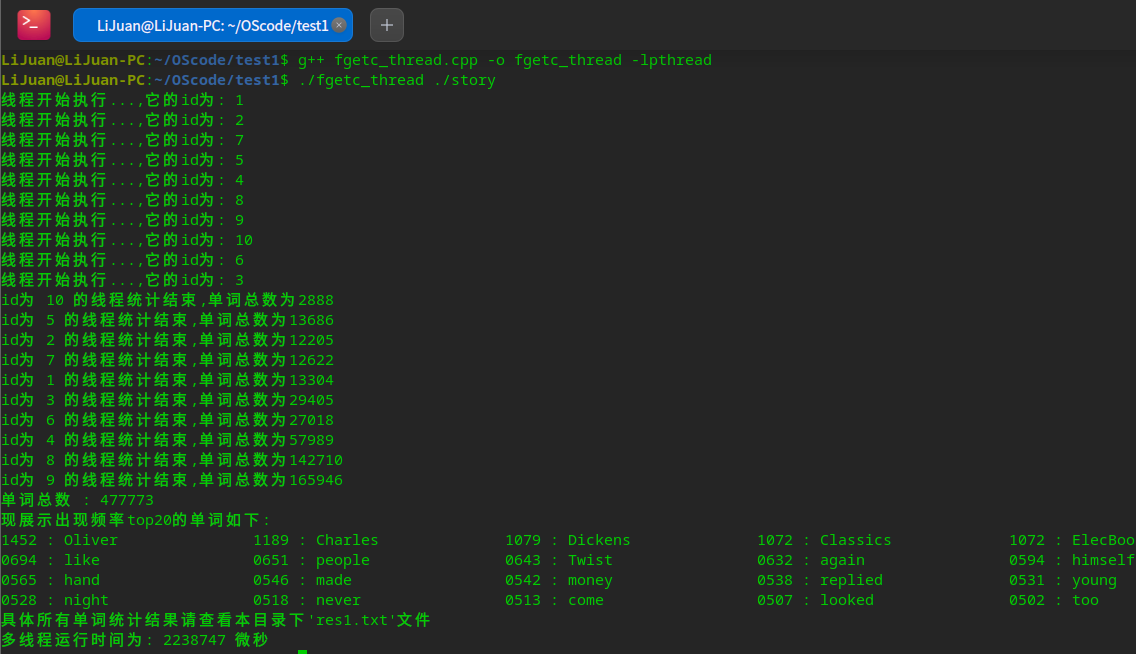


图6.1 多线程+fgetc运行结果

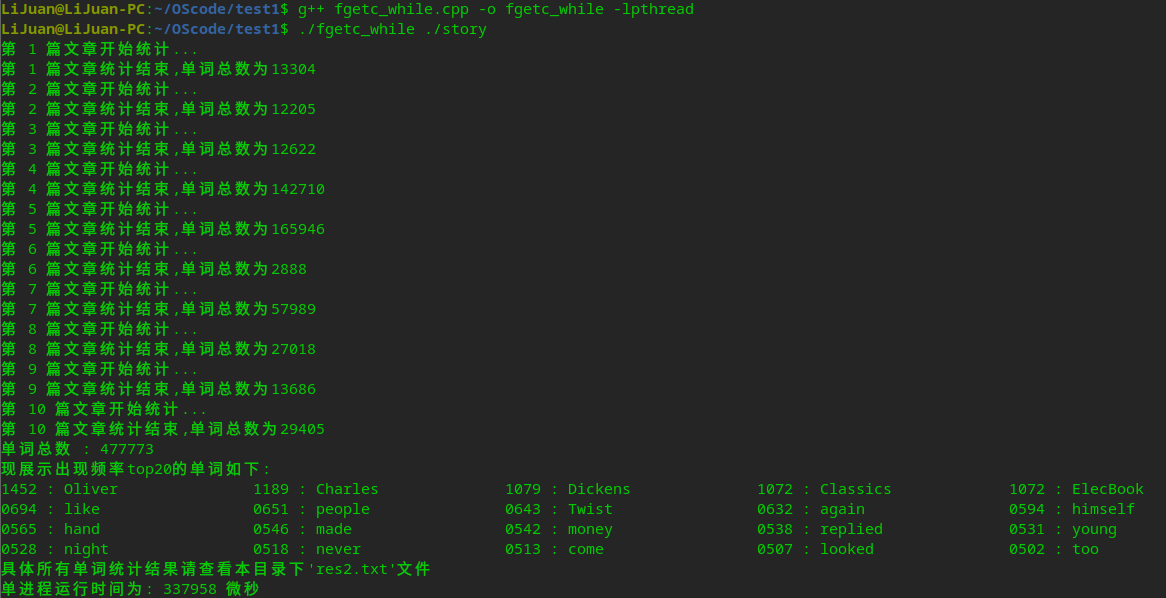


图6.2 单线程+fgetc运行结果

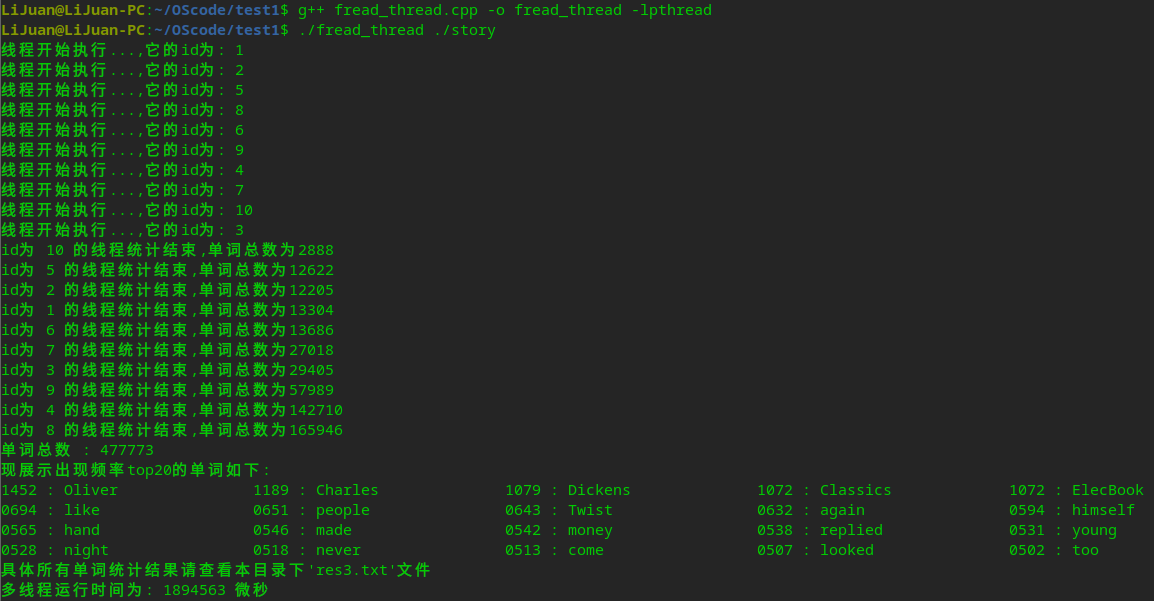


图6.3 多线程+fread运行结果

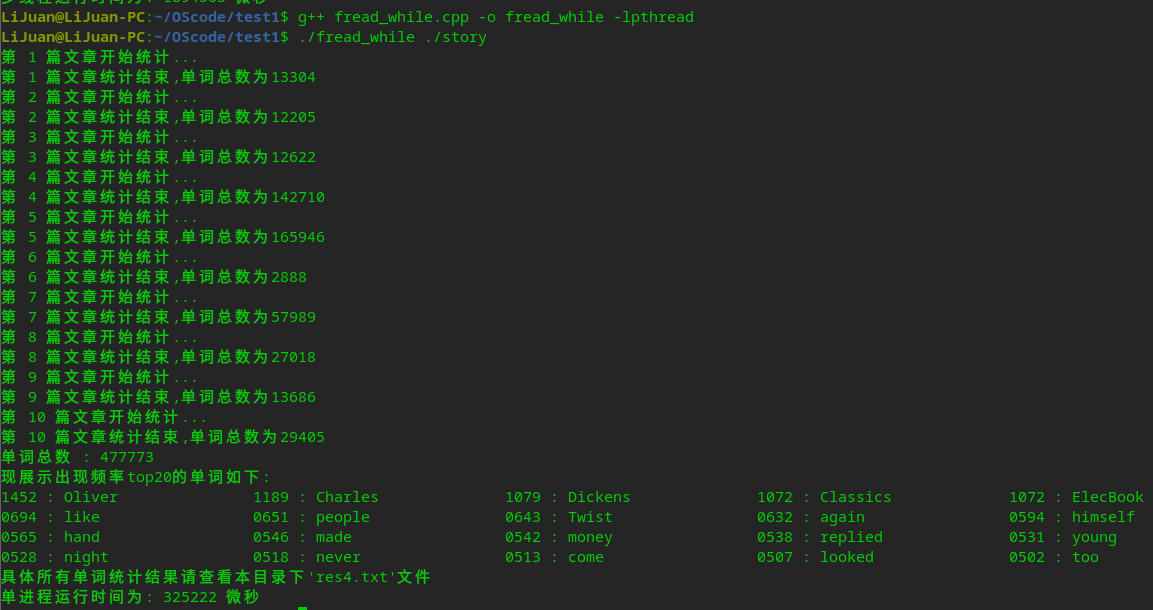


图6.4 单线程+fread结果

2.运行结果分析

对比可得，当小说篇数不算太多时（本次实验测试篇数为10），采用多字符读取，单线程统计的方式效率最高。

但当篇数较多时，预测用多字符读取，多线程统计效率较好。

七．总结

1.设计中的经验体会

经过小组的总结反思，我们认为出现上述的运行结果的原因实质上是：

fgetc是一个一个字符从文件也就是外存读取，而fread函数从文件进行一次读取后，一直是从内存读取；而从内存读数据速度是远快于外存的。在单线程中没有CPU的切换线程，而线程过多时，CPU会由于反复调度线程切换浪费大量的时间。

因此总的来说，可以推测出最合适的方式就是采用2-3个线程+fread文件读取。

2.后续设计设想

（1）联系起生产者消费者问题，在读入小说名的时候增添“生产者”，实现多线程读入小说名，进一步提高效率。

（2）自行学习多种语言的接口设计，尝试用Python等方式绘制单词云图

（3）将多线程单词统计程序与（2）的云图联系起来，设计真正具有使用价值的单词统计可视化系统。

课程设计选题二 生产者-消费者问题

一．设计要求

1.设计任务一

（1）利用信号量模拟“多生产者-多消费者”对同一缓存区的访问

（2）数据要求

输入数据：依靠编写函数随机生成英文字母

输出数据：线程用时、生产者线程操作（生产装入/阻塞等待）、消费者操作（消费取出/阻塞等待）、生产者/消费者同类之间互斥状态。

（3）功能要求

①用多线程实现生产者生产字符，并将商品存入buffer的过程；实现消费者将商品从buffer取出，并消费的过程。

②为生产者/消费者运行过程增添状态提示，使用相关提示语动态显示具体工作以使多线程程序中直观展示出多线程并发调度的过程。

**③进阶要求：**不直接调用linux中与线程同步互斥的内核函数，而是自己手写PV操作，简单实现阻塞队列，并格式化输出相关提示语句。

2.设计任务二

（1）用“生产者-消费者”模型实现选题1的单词统计功能

（2）数据要求

输入数据：长篇英文小说

输出数据：线程用时、生产者线程操作（生产装入/阻塞等待）、消费者操作（消费取出/阻塞等待）、生产者/消费者同类之间互斥状态、单词数、所有单词出现的频率、top20热词。

（3）功能要求

①确定合适的产品单位，将小说分割成buffer能容纳的数据块单位。

②用多线程实现生产者生产出数据块单位，并将商品存入buffer的过程；实现消费者将商品从buffer取出，并消费的过程。

③生产者/消费者运行过程增添状态提示，使用相关提示语动态显示具体工作以使多线程程序中直观展示出多线程并发调度的过程。

④用多线程统计单词出现频率并从中找出Top-N热词

⑤使用相关提示语以使多线程程序中直观展示出同步/互斥线程调度的过程。

**⑥进阶要求：**手写PV操作模拟同步/互斥线程的状态变化以及相关工作。

二．背景知识及设计思路

1.背景知识

**（1）Linux 的线程创建**

使用pthread\_create 函数创建用户级线程

**函数定义：**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr，void \*(\*start\_rtn) (void\*)，void \* arg);

**参数说明：**

thread: 待创建线程的 id 指针

attr: 创建线程时的线程属性

v(\*start\_rtin)(void\*): 返回值是 void\*类型的指针函数

arg: start\_routine 的参数

**返回值：**

成功返回 0

失败返回错误编号

 EAGAIN：表示系统限制创建新的线程，如线程数目过多

 EINVAL：代表线程属性值非法

**（2）线程同步终止**

**函数定义：**

int pthread\_join( pthread\_t thread, void \*\* rval\_ptr); **功能：**

调用者将挂起并等待新进程终止

当新线程调用 pthread\_exit()退出或者 return 时，进程中的其他线程可通过pthread\_join()获得进程的退出状态

**返回值：**

成功结束返回值为 0,否则为错误编码

**（3）带有线程程序的编译**

**基于 POSIX 标准的线程编程接口：**

 包括一个 pthread.h 头文件和一个线程库

**编译方法：**

 g++ \*\*.c -o \*\*\* –lpthread

1. **循环队列**

**循环队列定义：**

循环队列就是将队列存储空间的最后一个位置绕到第一个位置，形成逻辑上的环状空间，供队列循环使用。在循环队列结构中，当存储空间的最后一个位置已被使用而再要进入队运算时，只需要存储空间的第一个位置空闲，便可将元素加入到第一个位置，即将存储空间的第一个位置作为队尾。 循环队列可以更简单防止伪溢出的发生，但队列大小是固定的

**循环队列数据结构定义：**

int front;//指向队列头，指向第一个数据节点

int rear;//指向队列尾（并不是指向最后一个数据节点，而是最后一个数据节点后面的位置）

char data[];//节点数据，根据实际需要可以是不同的数据类型，但是因为是数组，需要在声明时指定大小

**循环队列大小：**

规定循环队列最多只能有MaxSize-1个队列元素，当循环队列中只剩下一个空存储单元时，队列就已经满了。因此，队列判空的条件是front=rear，而队列判满的条件是front=（rear+1)/MaxSize。

**（5）读取文件夹的目录和文件**

**函数定义：**

struct dirent \* rea

.ddir(DIR \* dir);

**函数功能：**

返回参数dir ⽬录流的下个⽬录进⼊点。

**结构dirent 定义：**

struct dirent{

ino\_t d\_ino; //d\_ino 此⽬录进⼊点的inode

ff\_t d\_off; //d\_off ⽬录⽂件开头⾄此⽬录进⼊点的位移

signed short int d\_reclen; //d\_reclen \_name 的长度, 不包含NULL 字符

unsigned char d\_type; //d\_type d\_name 所指的⽂件类型

har d\_name[256]; //d\_name ⽂件名

};

**返回值：**

成功则返回下个⽬录进⼊点. 有错误发⽣或读取到⽬录⽂件尾则返回NULL.

1. 设计思路

任务一设计思路：

（1）生产者生产产品，然后将产品放置在一个空缓冲区中。消费者进程从缓冲区中获得产品，然后释放缓冲区

（2）生产者生产产品时，先要 P(空缓冲区信号量）：如果没有空缓冲区，那么生产者就阻塞等待消费者释放出空缓冲区；如果有空缓冲区，生产者就放入产品，然后V(满缓冲区信号量)，唤醒等待满缓冲区的消费者。

（3）当消费者消费产品时，先要P(满缓冲区信号量)：如果没有满的缓冲区，那么消费者将被阻塞，直到新的产品被生产出来；如果有满的缓冲区，那么消费者就取出产品处理，同时V(空缓冲区信号量)，唤醒等待空缓冲区的生产者

（4）多个生产者消费者线程之间进行互斥

（5）采用循环队列存储生产或者消费的数据（可能是数字，字母，单词）

（6）加入计时，以便观察生产与消费运行时间

任务二设计思路：

1. 生产者生产单词存储到buffer中，buffer中存储单词
2. 消费者取出单词，统计单词频率及文章单词总数
3. 设计一个集合，存储着那些没有意义的单词，在我们后期输出的时候，一旦待输出的单词在这个集合中，说明它不是有意义的单词将被舍弃。
4. 采用STL标准库中的map来记录不同单词的个数，本质是近似的平衡二叉树
5. 总体的生产者消费者的定义和实现与任务一相同。
6. 设计方案
7. 数据结构设计

任务一：

//宏定义一些变量和信号量，方便进行调试

#define N 10             //缓冲区的大小

#define ProNum 3         //生产者的数量

#define ConNum 3         //消费者的数量

#define SleepTime 3      //生产者消费者操作完后的暂停时间

typedef int semaphore;   //定义信号量的类型均为整数

typedef char element;    //定义生产者生产的元素都是字符

element buffer[N] = {0}; //定义长度为 N 的缓冲区，内容均是字符型

int in = 0;              //缓冲区下一个存放产品的索引

int out = 0;             //缓冲区下一个可以使用的产品索引

int ProCount = 0;        //生产者序号统计

int ConCount = 0;        //消费者序号统计

double StartTime;        //程序开始时间记录，方便输出执行语句的时间

semaphore mutex = 1, empty = N, full = 0;

    mutex 用来控制生产和消费每个时刻只有一个在进行，初始化为 1，用来实现互斥

    empty 表示剩余的可用来存放产品的缓冲区大小，用来实现同步，刚开始有 N 个空位置

    full 表示当前缓冲区中可用的产品数量，用来实现同步，刚开始没有产品可以使用

任务二：

//宏定义一些变量和信号量，方便进行调试

#define N 10             //缓冲区的大小

#define ProNum 3         //生产者的数量

#define ConNum 3         //消费者的数量

#define SleepTime 0      //生产者消费者操作完后的暂停时间

/\*因为任务三注重的是结果，所以中间并不需要去暂停\*/

typedef int semaphore;   //定义信号量的类型均为整数

typedef string element;  //定义生产者生产的元素都是string字符串

element buffer[N]; //定义长度为 N 的缓冲区，内容均是字符型

set<string> st;//存储意义不大的介词、冠词等单词

int in = 0;              //缓冲区下一个存放产品的索引

int out = 0;             //缓冲区下一个可以使用的产品索引

int ProCount = 0;        //生产者序号统计

int ConCount = 0;        //消费者序号统计

double StartTime;        //程序开始时间记录，方便输出执行语句的时间

map<string,int>mp;//创建结果的map

int sum\_num=0;//单词结果的总数

char \*Buffer;//模拟从文件读取的数据块放入的磁盘缓冲区

int pointer=0;//上一行Buffer数组的索引指针

bool flag=false;//判断磁盘缓冲区的字符是否读完，读完为true，未读完为false

const int print\_num = 5;//定义每行打印多少单词

与任务一相比最大的区别：添加实验一中单词统计的一些数据结构如vector、set、以及string串，还有对txt文件互斥的信号量以及结果输出容纳的map红黑树。

1. 程序结构设计

任务一：

生产线程：

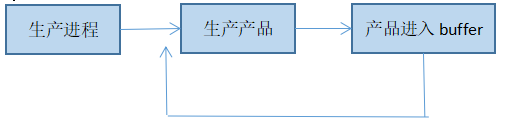


图3.1 生产者线程流程

消费线程：

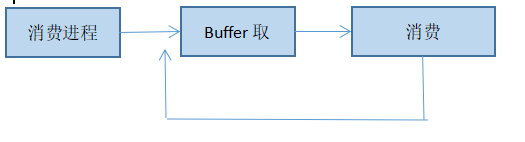


图3.2 消费者线程流程

主程序：

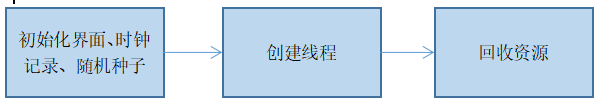


图3.3 主程序流程

任务二：

主进程：

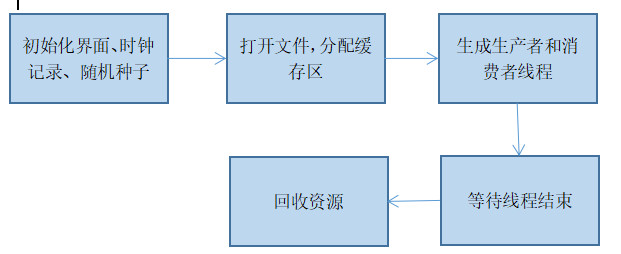


图3.4 主程序流程

生产者线程：

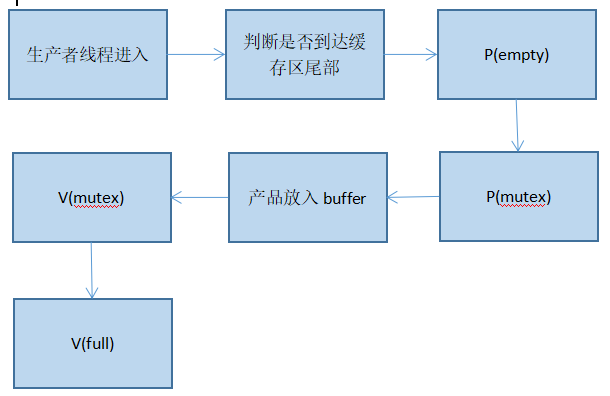


图3.5 生产者线程流程

消费者线程：

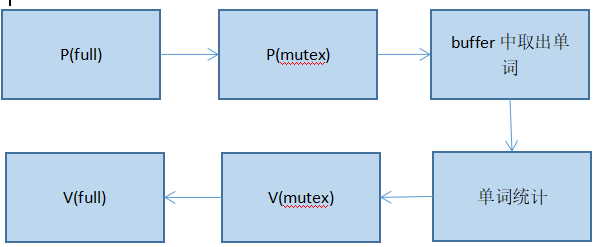


图3.6 消费者线程流程

1. 用到的Linux系统调用函数说明

**（1）clock函数**

clock\_t clock(void) ;

clock()是C/C++中的计时函数，而与其相关的数据类型是clock\_t

1. **CLOCKS\_PER\_SEC**

CLOCKS\_PER\_SEC是标准c的time.h头函数中宏定义的一个常数，表示一秒钟内CPU运行的时钟周期数，用于将clock()函数的结果转化为以秒为单位的量

1. **rand函数**

rand()函数用来产生随机数，用来生产一个产品

1. **sleep函数**

Sleep函数可以使计算机程序（进程，任务或线程）进入休眠，使其在一段时间内处于非活动状态，让生产者和消费者完成生产或消费后，进入短暂休眠

1. **pthread\_create函数**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr，void \*(\*start\_rtn) (void\*)，void \* arg);

用于普通用户创建线程，成功返回 0，失败返回错误编号

 EAGAIN：表示系统限制创建新的线程，如线程数目过多

 EINVAL：代表线程属性值非法

1. **pthread\_join函数**

int pthread\_join( pthread\_t thread, void \*\* rval\_ptr);

调用者将挂起并等待新进程终止

当新线程调用 pthread\_exit()退出或者 return 时，进程中的其他线程可通过pthread\_join()获得进程的退出状态，成功结束返回值为 0,否则为错误编码。

四．设计源代码

1、利用信号量模拟“多生产者-多消费者”对同一缓冲区的访问

/\*

    任务一：利用信号量模拟“多生产者-多消费者”对同一缓冲区的访问

    使用到自己改编重写的P/V操作

\*/

#include<iostream>

#include<cstdio>//基本格式的输入输出

#include<cstdlib>//使用了malloc申请全局内存，以及调用了信号量互斥访问的函数

#include<pthread.h>//线程库函数

#include<unistd.h>//系统调用API接口用于获取当前目录

#include<ctime>//统计时间以及生成种子以便随机生成生产者待生产的字符

#include<exception>//便于C++的try-catch-throw的异常

using namespace std;

//宏定义一些变量和信号量，方便进行调试

#define N 10             //缓冲区的大小

#define ProNum 3         //生产者的数量

#define ConNum 3         //消费者的数量

#define SleepTime 3      //生产者消费者操作完后的暂停时间

typedef int semaphore;   //定义信号量的类型均为整数

typedef char element;    //定义生产者生产的元素都是字符

element buffer[N] = {0}; //定义长度为 N 的缓冲区，内容均是字符型

int in = 0;              //缓冲区下一个存放产品的索引

int out = 0;             //缓冲区下一个可以使用的产品索引

int ProCount = 0;        //生产者序号统计

int ConCount = 0;        //消费者序号统计

double StartTime;        //程序开始时间记录，方便输出执行语句的时间

/\*

    定义并初始化信号量

    mutex 用来控制生产和消费每个时刻只有一个在进行，初始化为 1，用来实现互斥

    empty 表示剩余的可用来存放产品的缓冲区大小，用来实现同步，刚开始有 N 个空位置

    full 表示当前缓冲区中可用的产品数量，用来实现同步，刚开始没有产品可以使用

\*/

semaphore mutex = 1, empty = N, full = 0;

//生产者线程

void \*Producer(void \*args)

{

    int \*x = (int \*)args;

    element temp;//目前待生产的字符

    int sum = 0;

    while (1)

    {

        /\*同步操作：P(empty),判断是否有空位置供存放生产的产品，没有空位则等待并输出提示语句\*/

        while (empty <= 0) //

            printf("%.4lfs| 缓 冲 区 已 满 ! 生 产 者 :%d 等待中......\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x);

        empty--; //等到一个空位置，我们先把他占用

        /\*互斥操作： // P(mutex),判断当前临界区中是否有进程正在生产或者消费,如果有则等待并输出提示语句\*/

        while (mutex <= 0)

            printf("%.4lfs| 缓冲区有进程正在操作 !    生产者 :%d 等待中......\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x);

        mutex--; //等到占用进程出临界区，进入临界区并占用

        /\*

            生产者的生产过程：

            生产了一个产品后，输出生产成功的字样;

            然后将其放入buffer缓冲区中，in代表循环队列队尾的指针移动，以便下一个产品的生产

        \*/

        temp = (rand() % 26) + 65; //生产一个产品，即任意产生一个 A～Z 的字母

        printf("%.4lfs| 生产者: %d 生产一个产品: %c ,并将其放入缓冲区: %d 位置\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x, temp, in);//输出生产成功的字样

        buffer[in] = temp; //将生产的产品存入缓冲区

        sum++;

        in = (in + 1) % N; //缓冲区索引更新

        /\*互斥操作：V(mutex),临界区使用完成释放信号\*/

        mutex++;

        /\*同步操作：V(full),实现同步，释放 full\*/

        full++;

        /\*生产者生产之后让它休息一下，采用sleep实现\*/

        sleep(SleepTime); //休眠时间设为3个clock

    }

}

//消费者线程

void \*Consumer(void \*args)

{

    int \*x = (int \*)args;

    int sum = 0;

    while (1)

    {

        /\*同步操作：P(full),判断缓冲区当中是否有产品可以消费，同步如果没有产品则等待同时输出提示语句\*/

        while (full <= 0)

            printf("%.4lfs| \t\t\t\t\t\t\t 缓冲区为空！消费者: %d 等待中......\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x);

        full--; //等到有一个产品，消费这个产品

        /\*互斥操作：P(mutex),判断临界区是否有进程在处理,如果有则等待并输出提示语句\*/

        while (mutex <= 0)

            printf("%.4lfs| \t\t\t\t\t\t\t 缓冲区有进程正在操作! 消费者 : %d 等待中......\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x);

        mutex--; //等到临界区为空，则进入缓冲区消费

        /\*

            消费者的消费过程：

            发现有能够消费的产品，我们就输出消费成功的字样;

            然后将其从buffer缓冲区中取出，out代表循环队列队首的指针移动，以便下一个产品的消费

        \*/

        printf("%.4lfs| \t\t\t\t\t\t\t 消费者: %d 消费一个产品: %c , 将其从缓冲区取出 : %d 位置 \n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x, buffer[out], out);//输出消费成功的语句

        buffer[out] = 0; //更新缓冲区，把消费掉的产品清空

        sum++;

        out = (out + 1) % N; //更新缓冲区索引

        /\*互斥操作：V(mutex)，消费完成退出缓冲区，释放资源\*/

        mutex++;

        /\*同步操作：V(full)，消费完成产生一个空位，进行同步\*/

        empty++;

        /\*消费者消费之后让它休息一下，采用sleep实现\*/

        sleep(SleepTime); //休眠时间设为3个clock

    }

}

int main()

{

    /\*

        阶段一：一系列的初始化，包括开始时钟的记录，随机种子预备以及输出界面初始化

    \*/

    StartTime = clock(); //记录程序开始的时间，方便记录消费者和生产者活动的时间

    srand((int)time(NULL));//产生随机种子，生产的时候随机生产一个字符

    printf(" 计时");

    printf("    \t\t 生产者动态显示");

    printf("          \t\t\t\t 消费者动态显示\n");

    printf("======================================================================================================================\n");

    /\*

        阶段二：进行线程的创建，在本题中有3个生产者3个消费者，因此需要大小为6的线程数组

        分别创建三个生产者线程以及三个消费者线程

    \*/

    pthread\_t threadPool[ProNum + ConNum];//定义一个线程数组，存储所有的消费者和生产者线程,

    //创建生产者

    int t1=1,t2=2,t3=3;

    pthread\_create(&threadPool[0], NULL, Producer, (void \*)&t1);

    pthread\_create(&threadPool[1], NULL, Producer, (void \*)&t2);

    pthread\_create(&threadPool[2], NULL, Producer, (void \*)&t3);

    //创建消费者

    pthread\_create(&threadPool[3], NULL, Consumer, (void \*)&t1);

    pthread\_create(&threadPool[4], NULL, Consumer, (void \*)&t2);

    pthread\_create(&threadPool[5], NULL, Consumer, (void \*)&t3);

    /\*

        阶段三：线程结束资源的回收

        在这里因为涉及到线程回收因此要添加异常检测

    \*/

    void \*result;

    for (int i = 0; i < ProNum + ConNum; i++) //等待线程结束并回收资源，线程之间同步

    {

        int Return=pthread\_join(threadPool[i], &result);

        try{

            if (Return== -1) throw Return;

        }

        catch(int){

            cerr<<"检测到线程回收失败"<<endl;//回收失败的错误提示

            exit(1);//结束线程

        }

    }

}

2、用“生产者-消费者”模型实现选题1的单词统计功能

/\*

    任务三：用“生产者-消费者”模型实现选题1的单词统计功能

    使用到自己改编重写的P/V操作以及选题1中的重要数据结构

\*/

#include<iostream>//异常输出时需要用io流的cerr

#include<cstdio>//基本格式的输入输出

#include<cstdlib>//使用了malloc申请全局内存，以及调用了信号量互斥访问的函数

#include<pthread.h>//线程库函数

#include<unistd.h>//系统调用API接口用于获取当前目录

#include<ctime>//统计时间以及生成种子以便随机生成生产者待生产的字符

#include<exception>//便于C++的try-catch-throw的异常

#include<cstring>//字符串操作的库函数

#include<map>//用来存储单词出现次数

#include<algorithm>//排序算法必备

#include<vector>//排序的辅助向量

#include<set>//创建了无用词集合

#include<fstream>//文件流读取

#include<sstream>// istringstream

using namespace std;

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2);//排序所用的比较函数

void Print();//打印结果

void InitWord();//词库初始化

bool Ismeaningful(string ch);//检查该单词是否是有意义的

//宏定义一些变量和信号量，方便进行调试

#define N 10             //缓冲区的大小

#define ProNum 3         //生产者的数量

#define ConNum 3         //消费者的数量

#define SleepTime 0      //生产者消费者操作完后的暂停时间

/\*因为任务三注重的是结果，所以中间并不需要去暂停\*/

typedef int semaphore;   //定义信号量的类型均为整数

typedef string element;  //定义生产者生产的元素都是string字符串

element buffer[N]; //定义长度为 N 的缓冲区，内容均是字符型

set<string> st;//存储意义不大的介词、冠词等单词

int in = 0;              //缓冲区下一个存放产品的索引

int out = 0;             //缓冲区下一个可以使用的产品索引

int ProCount = 0;        //生产者序号统计

int ConCount = 0;        //消费者序号统计

double StartTime;        //程序开始时间记录，方便输出执行语句的时间

map<string,int>mp;//创建结果的map

int sum\_num=0;//单词结果的总数

char \*Buffer;//模拟从文件读取的数据块放入的磁盘缓冲区

int pointer=0;//上一行Buffer数组的索引指针

bool flag=false;//判断磁盘缓冲区的字符是否读完，若读完则为true，未读完为false

const int print\_num = 5;//定义每行打印多少单词

/\*

    定义并初始化信号量

    mutex 用来控制生产和消费每个时刻只有一个在进行，初始化为 1，用来实现互斥

    empty 表示剩余的可用来存放产品的缓冲区大小，用来实现同步，刚开始有 N 个空位置

    full 表示当前缓冲区中可用的产品数量，用来实现同步，刚开始没有产品可以使用

\*/

semaphore mutex = 1, empty = N, full = 0;

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2);//排序所用的比较函数

bool Isword(char ch){

    if((ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z'))//只有大小写字母才可以组成单词

        return true;

    return false;

}

//生产者线程

void \*Producer(void \*args)

{

    // printf("1111111111processing\n");

    int \*x = (int \*)args;

    int sum = 0;

    if(Buffer[pointer]=='\0'){//到达了缓冲区尾部

        flag=true;

        return NULL;

    }

    while (Buffer[pointer]!='\0')

    {

        /\*同步操作：P(empty),判断是否有空位置供存放生产的产品，没有空位则等待并输出提示语句\*/

        while (empty <= 0) //

            printf("%.4lfs| 缓 冲 区 已 满 ! 生 产 者 :%d 等待中......\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x);

        empty--; //等到一个空位置，我们先把他占用

        /\*互斥操作： // P(mutex),判断当前临界区中是否有进程正在生产或者消费,如果有则等待并输出提示语句\*/

        while (mutex <= 0)

            printf("%.4lfs| 缓冲区有进程正在操作 !    生产者 :%d 等待中......\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x);

        mutex--; //等到占用进程出临界区，进入临界区并占用

        string word="";//目前待统计的单词

        /\*

            生产者的生产过程：

            生产了一个产品后，输出生产成功的字样;

            然后将其放入buffer缓冲区中，in代表循环队列队尾的指针移动，以便下一个产品的生产

        \*/

        while(1){

            /\*在单词分割的循环里如果已经到达缓冲区尾部，就强行停止

            为什么外部循环有判断缓冲区尾部的判断了，这里还加

            是因为内循环的while(1)循环中可能恰好分割完最后的单词

            如果这里不填加if限制，会导致段错误数组溢出\*/

            if(Buffer[pointer]=='\0'){

                flag=true;

                return NULL;

            }

            if(Isword(Buffer[pointer])){

                word+=Buffer[pointer];

                pointer++;

            }

            else{

                /\*如果当前单词串为空,直接读入字符但是不插入word（因为不是字母）\*/

                if(word==""){

                    pointer++;

                    continue;

                }

                pointer++;

                break;

            }

        }

        printf("%.4lfs| 生产者: %d 生产一个产品: %s ,并将其放入缓冲区: %d 位置\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x, word.c\_str(), in);//输出生产成功的字样

        buffer[in] = word; //将生产的产品存入缓冲区

        word="";

        in = (in + 1) % N; //缓冲区索引更新

        /\*互斥操作：V(mutex),临界区使用完成释放信号\*/

        mutex++;

        /\*同步操作：V(full),实现同步，释放 full\*/

        full++;

        /\*生产者生产之后让它休息一下，采用sleep实现\*/

        sleep(SleepTime); //休眠时间设为0

    }

    return NULL;

}

//消费者线程

void \*Consumer(void \*args)

{

    int \*x = (int \*)args;

    int sum = 0;

    while (!flag)

    {

        /\*同步操作：P(full),判断缓冲区当中是否有产品可以消费，同步如果没有产品则等待同时输出提示语句\*/

        while (full <= 0)

            printf("%.4lfs| \t\t\t\t\t\t\t 缓冲区为空！消费者: %d 等待中......\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x);

        full--; //等到有一个产品，消费这个产品

        /\*互斥操作：P(mutex),判断临界区是否有进程在处理,如果有则等待并输出提示语句\*/

        while (mutex <= 0)

            printf("%.4lfs| \t\t\t\t\t\t\t 缓冲区有进程正在操作! 消费者 : %d 等待中......\n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x);

        mutex--; //等到临界区为空，则进入缓冲区消费

        /\*

            消费者的消费过程：

            发现有能够消费的产品，我们就输出消费成功的字样;

            然后将其从buffer缓冲区中取出，out代表循环队列队首的指针移动，以便下一个产品的消费

        \*/

        printf("%.4lfs| \t\t\t\t\t\t\t 消费者: %d 消费一个产品: %s , 将其从缓冲区取出 : %d 位置 \n", (clock() - StartTime) / CLOCKS\_PER\_SEC, \*x, buffer[out].c\_str(), out);//输出消费成功的语句

        mp[buffer[out]]++;//取出缓冲区内单词

        buffer[out] = ""; //更新缓冲区，把消费掉的产品清空

        sum\_num++;//文章单词总数+

        out = (out + 1) % N; //更新缓冲区索引

        /\*互斥操作：V(mutex)，消费完成退出缓冲区，释放资源\*/

        mutex++;

        /\*同步操作：V(full)，消费完成产生一个空位，进行同步\*/

        empty++;

        /\*消费者消费之后让它休息一下，采用sleep实现\*/

        sleep(SleepTime); //休眠时间设为0

    }

    return NULL;

}

int main(int argc,char\* argv[])

{

    char filename[20];

    strcpy(filename,argv[1]);

    /\*

        阶段一：一系列的初始化，包括开始时钟的记录，随机种子预备，待读取文章以及输出界面初始化

    \*/

    InitWord();//词库初始化

    StartTime = clock(); //记录程序开始的时间，方便记录消费者和生产者活动的时间

    printf(" 计时");

    printf("    \t\t 生产者动态显示");

    printf("          \t\t\t\t 消费者动态显示\n");

    printf("======================================================================================================================\n");

    FILE \*p=fopen(filename,"r");//打开文件

    /\*获取文件的大小\*/

    printf("wocao\n");

    fseek(p,0,SEEK\_END);

    long lSize=ftell(p);

    rewind(p);

    /\*分配能完整读取文件的缓冲区大小\*/

    Buffer=(char\*)malloc(lSize\*sizeof(char));

    /\*开始用fread将字符全部读入缓冲区\*/

    fread(Buffer,sizeof(char),lSize,p);

    fclose(p);//读完这个文件了就关闭文件

    /\*

        阶段二：进行线程的创建，在本题中有3个生产者3个消费者，因此需要大小为6的线程数组

        分别创建三个生产者线程以及三个消费者线程

    \*/

    pthread\_t threadPool[ProNum + ConNum];//定义一个线程数组，存储所有的消费者和生产者线程,

    //创建生产者

    int t1=1,t2=2,t3=3;

    pthread\_create(&threadPool[0], NULL, Producer, (void \*)&t1);

    pthread\_create(&threadPool[1], NULL, Producer, (void \*)&t2);

    pthread\_create(&threadPool[2], NULL, Producer, (void \*)&t3);

    //创建消费者

    pthread\_create(&threadPool[3], NULL, Consumer, (void \*)&t1);

    pthread\_create(&threadPool[4], NULL, Consumer, (void \*)&t2);

    pthread\_create(&threadPool[5], NULL, Consumer, (void \*)&t3);

    /\*

        阶段三：线程结束资源的回收

        在这里因为涉及到线程回收因此要添加异常检测

    \*/

    void \*result;

    for (int i = 0; i < ProNum + ConNum; i++) //等待线程结束并回收资源，线程之间同步

    {

        int Return=pthread\_join(threadPool[i], &result);

        try{

            if (Return== -1) throw Return;

        }

        catch(int){

            cerr<<"检测到线程回收失败"<<endl;//回收失败的错误提示

            exit(1);//结束线程

        }

    }

    Print();

}

/\*自定义cmp用于后期排序使用\*/

bool cmp(const pair<string,int> &p1,const pair<string,int> &p2){

    if(p1.second==p2.second)

        return p1.first<p2.first;

    return p1.second>p2.second;

}

/\*输出统计结果\*/

void Print(){

    printf("单词总数 : %d\n",sum\_num);

    /\*默认的map是按key排序的,输出之前换用value排序\*/

    vector<pair<string,int>> arr;

   for(map<string,int>::iterator it=mp.begin();it!=mp.end();++it)

    arr.push\_back(make\_pair(it->first,it->second));

   sort(arr.begin(),arr.end(),cmp);//对vector排序

    /\*遍历vector等效作为遍历map容器\*/

    printf("现展示出现频率top20的单词如下:\n");

    FILE \*fp;

    fp=fopen("result.txt","w");

    int i,k=1;

    for(vector<pair<string,int>>::iterator it=arr.begin();it!=arr.end();++it){

        i++;

        if(k<=20){

            if(Ismeaningful(it->first)){

                printf("%04d : %s ", it->second, (it->first).c\_str());

                for(int j = 20 - (it->first).size();j > 0;j--) putchar(' ');

                if(k % print\_num == 0) puts("");

                k++;

            }

        }

        fprintf(fp,"%05d : %s ", it->second, (it->first).c\_str());

        for(int j = 20 - (it->first).size();j > 0;j--) fputs(" ",fp);

        if(i % print\_num == 0) fputs("\n",fp);

    }

    if(i % print\_num != 0) fputs("\n",fp);

    printf("具体所有单词统计结果请查看本目录下'result.txt'文件\n");

}

void InitWord(){

    ifstream fp("words.txt");//打开文件

    char s[100];

    while(!fp.eof()){

        fp.getline(s,100);

        st.insert(s);

    }

}

bool Ismeaningful(string ch)

{

    /\*如果单词不在集合里，说明是有意义的\*/

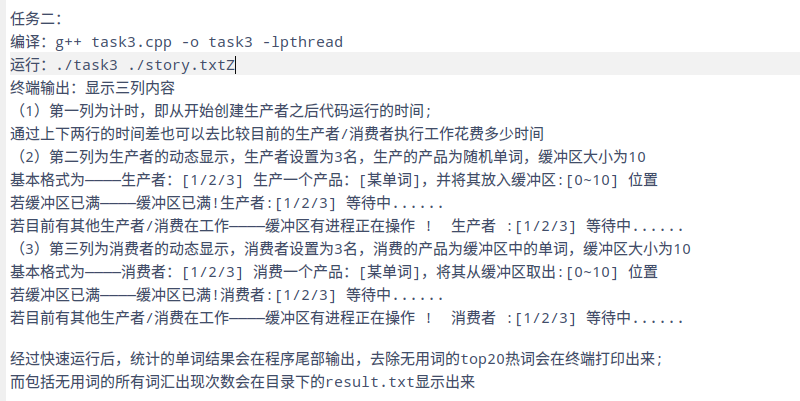
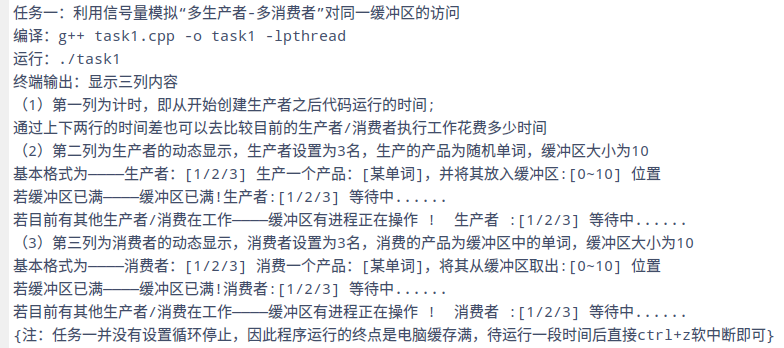
    if(st.count(ch)==0) return true;

    return false;

}

五．调试方案及调试过程问题分析

两个代码文件的运行方式均书写了一份操作手册：



六．运行结果与分析

1.测试用例及各功能运行结果截图

任务一：



图6.1 任务一运行结果

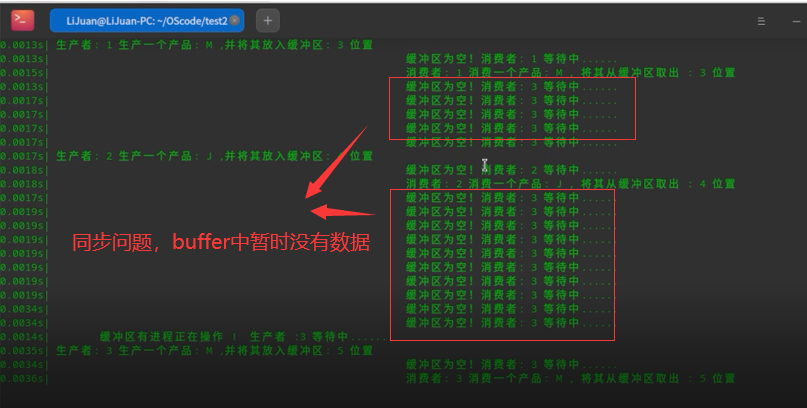


图6.2 任务一运行结果

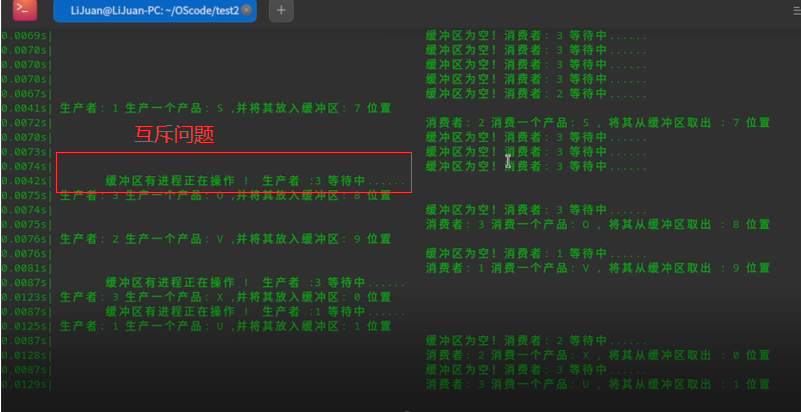


图6.3 任务一运行结果

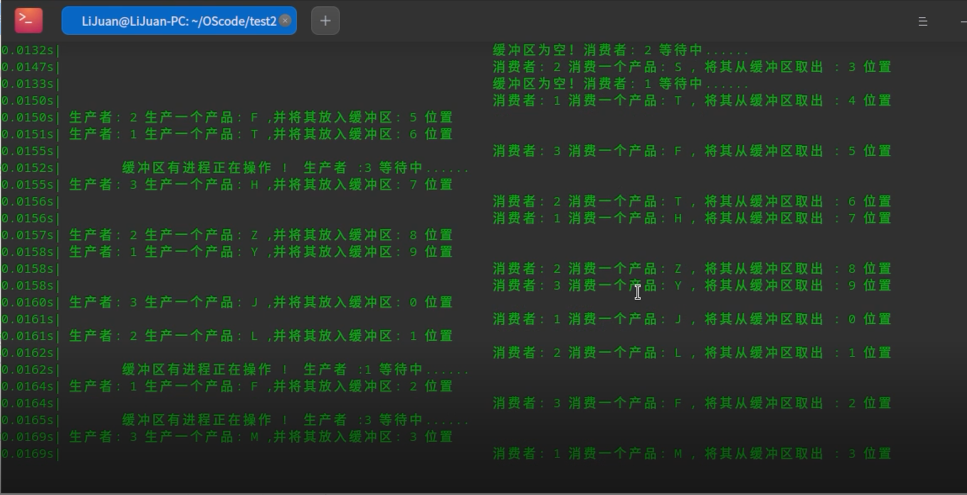


图6.4 任务一运行结果

任务二：



图6.5 任务二运行结果

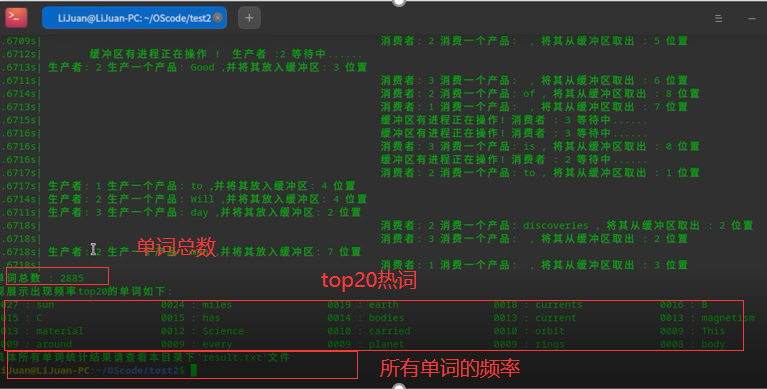


图6.6 任务二运行结果

1. 运行结果分析

任务一：开始时，生产者生产产品数与消费者所需产品数相同，产品消费完后，生产者继续生产，消费者出现长时间等待状态，待生产者将产品重新放入对应消费者所需的buffer时，出现一产一消现象

任务二：开始时读入单词数据，生产者先拥有大量的可以生产的产品就绪，随即让消费者从另一端将单词从buffer中取出，读入我们的map红黑树，待消费者将buffer数据取出，生产者开始生产，后逐渐实现一产一消周期性反复进行。待小说单词读完，输出总体统计结果。

七．总结

1.设计中的经验体会

本次实验是关于生产者与消费者之间同步和互斥问题，实验设一个共享缓存区，采用循环队列存储生产或者消费的数据，可以很好避免了假溢出的问题。

我们小组没有选择linux自带的同步互斥库函数，而是选择自己书写PV操作用以任务的简单实现，因为我们从操作的本质出发，无非就是线程无法完成自己目前的任务时，就将自身放入阻塞队列。待前方任务完成“道路畅通”后才可进行后续操作。

2.后续设计设想

（1）本次实验采用了一篇小说，实质上可以与实验一结合像其一样读一整个文件夹中的所有小说。但涉及到子线程里再嵌套子线程，其并发控制带有一些难度。待知识储备增加后，可以尝试去实现。

（2）自行学习多种语言的接口设计，尝试用Python等方式绘制单词云图

（3）学习引用Java里的线程池，真正将其设计成有实际价值的系统，系统的依赖为生产者消费者模型。

课程设计选题三 动态分区存储管理模拟系统

一．设计要求

1.数据要求

任务一：设计程序来模拟可变分区的分配和回收

**输入数据：**手动选择生成的线程id、大小以及内存分配算法、手动选择分配/回收/显示/退出系统等操作。

**输出数据：**内存分配的成功/失败；内存回收的结果；当前内存占有情况

任务二：用多线程模拟多道作业对内存空间的动态申请和释放

**输入数据：**随机生成的线程大小、随机的内存分配算法

**输出数据：**10个线程并发分配/回收的状态展示；以及实时内存占有情况。

2.功能要求

任务一：

（1）设计一个用来记录内存空间使用情况的数据结构即MAT表。

（2）设计登记空闲区和作业区的数据结构即空闲分区链。

（3）在设计好的该数据结构上面，设计内存分配算法（FF,BF,WF）。

（4）在设计好的该数据结构上面，设计内存回收算法。

（5）提供内存碎片整理功能，即在内存回收时检测上下内存是否空闲。

（6）在控制台添加相关提示操作，引导实现综合模拟系统。

任务二：

（1）设计一个程序创建n个线程，每个线程在执行时调用动态分区管理模块

（2）内存分配成功后在终端打印出实时内存使用情况

（3）随机使用内存分配算法（FF,BF,WF），设计内存回收算法。

（4）模拟出N个线程并发执行的一个更加真实的动态内存分配/回收场景

二．背景知识及设计思路

1.背景知识

**（1）Linux 的线程创建**

使用pthread\_create 函数创建用户级线程

**函数定义：**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr，void \*(\*start\_rtn) (void\*)，void \* arg);

**参数说明：**

thread: 待创建线程的 id 指针

attr: 创建线程时的线程属性

v(\*start\_rtin)(void\*): 返回值是 void\*类型的指针函数

arg: start\_routine 的参数

**返回值：**

成功返回 0

失败返回错误编号

 EAGAIN：表示系统限制创建新的线程，如线程数目过多

 EINVAL：代表线程属性值非法

**（2）线程同步终止**

**函数定义：**

int pthread\_join( pthread\_t thread, void \*\* rval\_ptr); **功能：**

①调用者将挂起并等待新进程终止

②当新线程调用 pthread\_exit()退出或者 return 时，进程中的其他线程可通过pthread\_join()获得进程的退出状态

**返回值：**

成功结束返回值为 0,否则为错误编码

**（3）带有线程程序的编译**

**基于 POSIX 标准的线程编程接口：**

 包括一个 pthread.h 头文件和一个线程库

**编译方法：**

 g++ \*\*.c -o \*\*\* –lpthread

**（4）TOP-N热词统计设计**

C++ 中有一个<map>容器类，其实就是封装了一个红黑树（近似平衡二叉树），可以调用map容器来实现基于平衡二叉树的字典结构。

map<string,int> word\_count

map<string,int>::iterator iter

string word

word\_count.insert(pair<string,int>(word,1))

iter = word\_count.find(word)

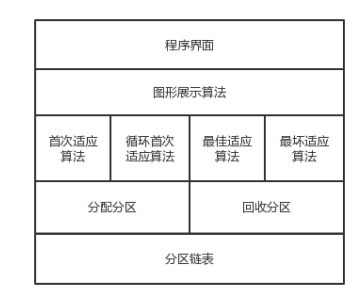
word\_count[iter]+=1

2.设计思路

在 C 语言面向过程编程方法中，常用分层方法进行编程，降低各模块之间的耦合度。在操作系统中，也用到了分层思想进行设计，每一步设计都建立在可靠的基础上，这样易于扩充和维护系统。

在此次的课程设计中，同样也使用分层的方法，可以将各种算法很简单方便的一起实现。最底层是基本的数据结构，用来存储分区的信息。上一层是分配分区的函数 alloc()和回收分区的函数 free()。通过调用 alloc()和 free()函数可以实现四种算法。通过图形展示算法可以直观地显示分配和回收过程。最上一层是程序的界面，允许用户可以自由的对内存进行分配和回收，程序展示动态结果。

总体结构如图所示：



三．设计方案

1.数据结构设计

在实际的操作系统中，分配出去的内存记录在进程控制块中，为了加快内存的分配过程，动态分区最常采用的数据结构是空闲表或空闲链表，数据结构只记录未分配的内存分区。为了降低模拟程序的复杂度，本模拟将分配出去的和未分配出去的内存分区记录在同一个数据结构中，用一个标志表示是否已被分配。各分区按地址递增次序排列。内存的分配和回收只要是对数据结构做插入和删除操作，采用双向链表结构。

为使程序具有扩展性灵活性，分区大小采用宏定义方式 （ #define MEMORY\_SIZE 640），可修改分区的大小，展示在不同分区大小下的分配情况。

#define Max 1048576

typedef struct freeList Flist;

typedef struct node {

  int id;        //装入此内存块进程的id；

  int startAdd;  //内存块的首地址

  int length;    //内存块的长度

  unsigned char status;  //状态位，记录该分区的状态，默认为0，表示未分配，不为0表示已经分配了

} Node;

//记录内存的链表

typedef struct list {

  struct list \*last;  //链表记录的上一个节点

  struct list \*next;  //链表记录的下一个节点

  Flist \*flist;

  Node \*cur;  //链表记录的当前节点

} List;

struct freeList {

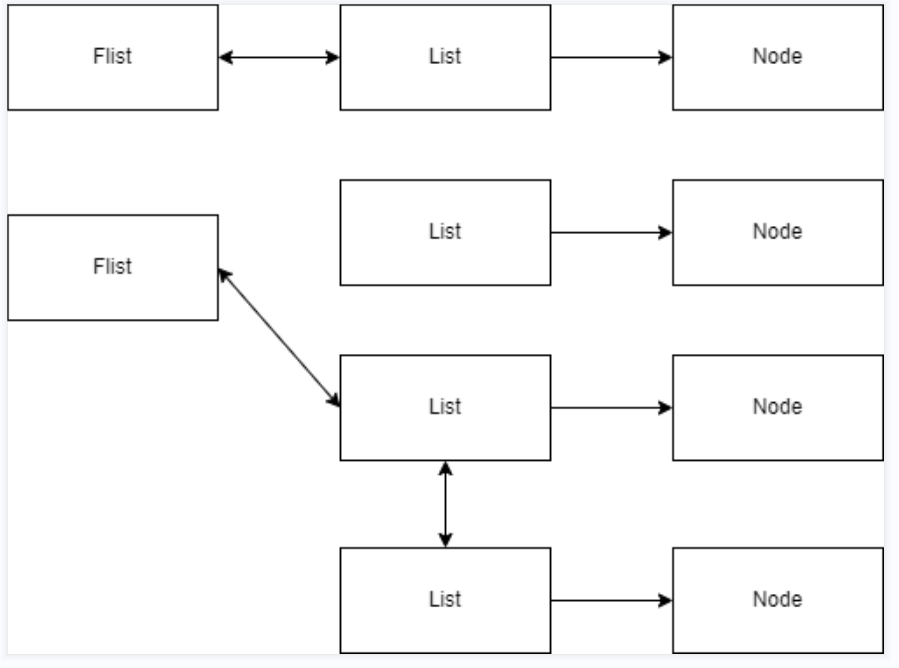
  struct freeList \*last;

  struct freeList \*next;

  List \*list;

};

大致关系图



2.程序结构设计

**内存动态多分区系统总框图：**

可选择进入不同算法进行内存分配和回收的演示，每个算法既支持手动输入

分配回收序列，也支持从文件中输入分配回收序列。每进行一步即展示当前这一

步的分配结果。展示时内存分配表和分配图并排展示，可以更直观清晰的看到内

存分配回收情况，加深对算法的理解程度。

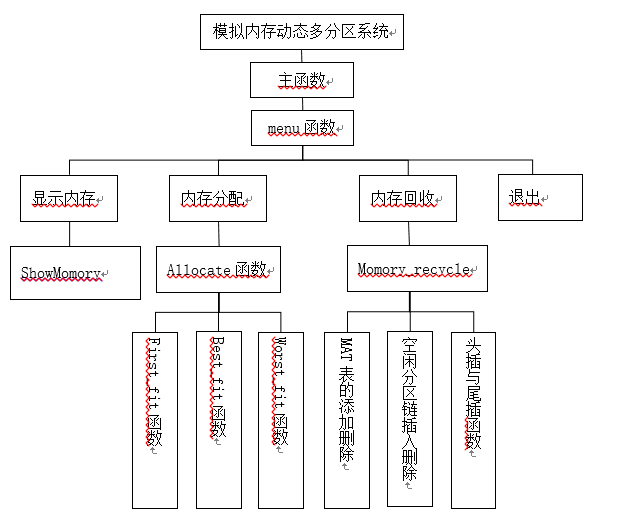


图3.1 内存动态多分区系统总框图

**分区分配算法：**

在链表位置 p 处为编号为 pid，大小为 psize 的作业分配空间，链表 p 通过各种查找空闲分区算法（首次适应、循环首次适应、最佳适应、最坏适应）提供。

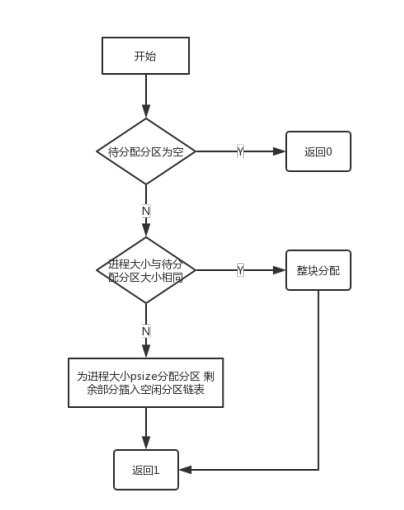


图3.2 分区分配算法

**回收分区算法：**

释放 pid 进程的分区空间，重新插入到空闲分区链表。

返回值为本次回收是否成功。

回收时先遍历链表，找到进程 pid 所在的位置 p，如果存在，可能出现下面四种情况之一：

1） 回收区与前一空闲分区相邻，此时应将回收区与前一分区合并。

2） 回收区与后一空闲分区相邻，此时应将回收区与后一分区合并。

3） 回收区同时与前后两个空闲分区相邻，此时将三个分区合并。

4） 回收区不相邻前后空闲分区，直接改为空闲分区。

算法流程图如下所示：

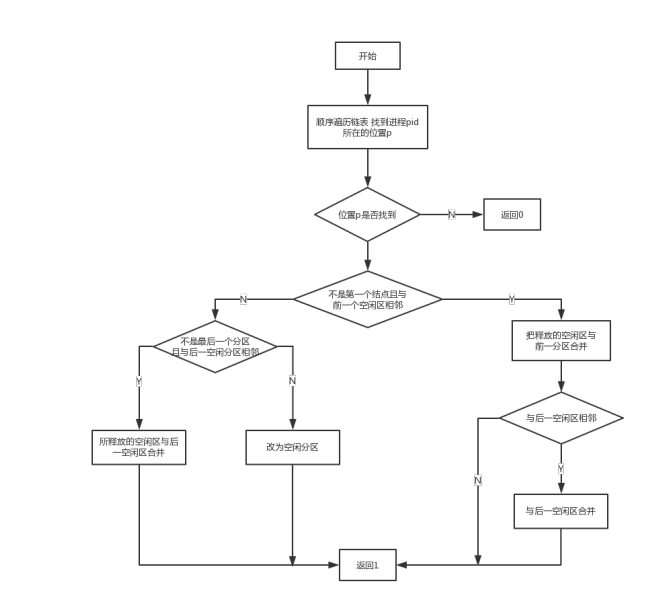


图3.3回收分区算法

**首次适应算法寻找分区位置：**

按照首次适应算法从分区链表中找到第一个不小于 psize 的空闲分区，返回此分区地址。如果找不到可用的空闲分区返回 NULL。

算法流程图如下所示：

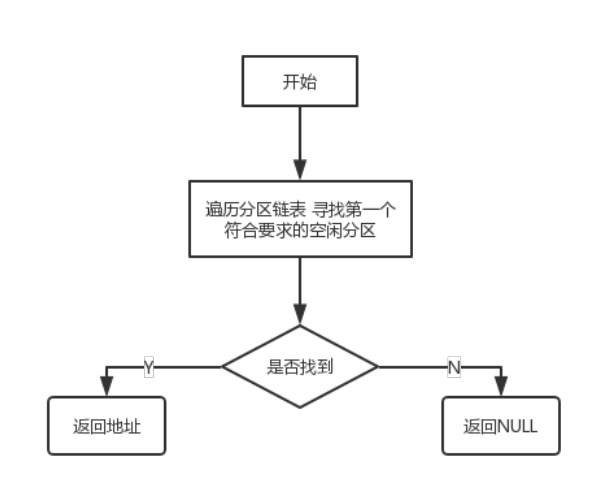


图3.4 首次适应算法

**循环首次适应算法寻找分区位置：**

按照循环首次适应算法从分区链表的当前位置开始寻找第一个不小于psize 的空闲分区，返回此分区地址。寻找结束后更新当前位置。当链表循环一圈后仍没有找到合适的空闲分区则返回 NULL。

算法流程图如下所示：

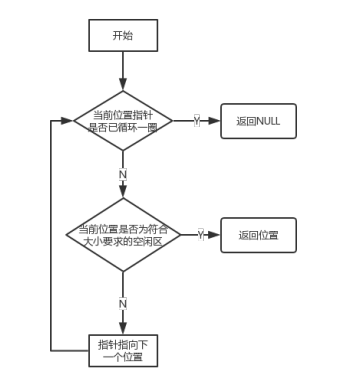


图3.5 循环首次适应算法

**最佳适应算法寻找分区位置：**

可以将所有空闲分区按照大小升序排列，从头寻找第一个符合要求的空间分区进行分配。在这里为简单起见，对链表线性搜索一轮，找到符合空间大小要求的最小的空闲分区。

算法流程图如下所示：

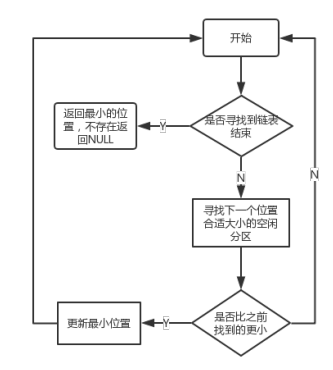


图3.6 最佳适应算法寻找分区位置

**最坏适应算法寻找分区位置：**

可以将所有空闲分区按照大小降序排列，从头寻找第一个符合要求的空间分区进行分配。在这里为简单起见，对链表线性搜索一轮，找到符合空间大小要求的最大的空闲分区。

算法流程图如下所示：

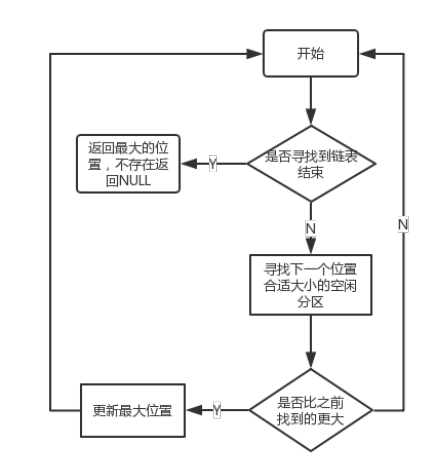


图3.7 最坏适应算法寻找分区位置

**绘制状态图算法：**

显示目前内存的所有分配状态，包括线程id、内存大小、首位置、末位置、空闲状态

3.用到的Linux系统调用函数说明

**（1）pthread\_mutex\_init函数**

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,const pthread\_mutexattr\_t \*restrict attr);

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

该函数用于C函数的多线程编程中，互斥锁的初始化。以动态方式创建互斥锁的，参数attr指定了新建互斥锁的属性。如果参数attr为空(NULL)，则使用默认的互斥锁属性，默认属性为快速互斥锁 。互斥锁的属性在创建锁的时候指定，在LinuxThreads实现中仅有一个锁类型属性。

**（2）pthread\_create函数**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr，void \*(\*start\_rtn) (void\*)，void \* arg);

用于普通用户创建线程，成功返回 0，失败返回错误编号

 EAGAIN：表示系统限制创建新的线程，如线程数目过多

 EINVAL：代表线程属性值非法

**（3）pthread\_join函数**

int pthread\_join( pthread\_t thread, void \*\* rval\_ptr);

①调用者将挂起并等待新进程终止

②当新线程调用 pthread\_exit()退出或者 return 时，进程中的其他线程可通过pthread\_join()获得进程的退出状态，成功结束返回值为 0,否则为错误编码。

**（4）pthread\_mutex\_lock函数**

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

对互斥量加锁，如果该互斥量已经被锁住，函数调用失败，返回EBUSY，否则加锁成功返回0，线程不会被阻塞。

**（5）pthread\_mutex\_unlock函数**

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

对互斥量解锁。

四．设计源代码

1. 设计程序来模拟可变分区的分配和回收。

 #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define Max 1048576

typedef struct freeList Flist;

typedef struct node {

  int id;        //装入此内存块进程的id；

  int startAdd;  //内存块的首地址

  int length;    //内存块的长度

  unsigned char status;  //状态位，记录该分区的状态，默认为0，表示未分配，不为0表示已经分配了

} Node;

//记录内存的链表

typedef struct list {

  struct list \*last;  //链表记录的上一个节点

  struct list \*next;  //链表记录的下一个节点

  Flist \*flist;

  Node \*cur;  //链表记录的当前节点

} List;

struct freeList {

  struct freeList \*last;

  struct freeList \*next;

  List \*list;

};

void Menu();      //界面函数

List \*InitMat();  //初始化mat表的函数，必须在空闲分区表之前

Flist \*InitFreeTable(List \*Mat);  //初始化空闲分区表

int Allocate(Flist \*fl, int id, int psize,Flist \*\*fhead);  //分配方法，具体参数定义见函数体 ,返回值未定义，可自定义

void Momory\_recycle(List \*\*mat, Flist \*\*Free); //回收方法

Flist\* First\_fit(Node\* tempnode,Flist \*fl); //首次适应算法

Flist\* Best\_fit(Node\* tempnode,Flist\*fl);  //最佳适应算法

Flist\* Worst\_fit(Node\* tempnode,Flist \*fl);//最坏适应算法

Flist\* Circl\_fit(Node\* tempnode,Flist \*fl,Flist \*Last\_fl); //循环适应算法

void ShowMomory(List \*head);//输出MAT表所有内存占有情况

void AllChoice(Flist \*fhead);//提供内存分配的选择

int main() {

  List \*Lhead = InitMat();

  Flist \*Fhead = InitFreeTable(Lhead);

  int cid = 0;

  while(1)

  {

    Menu();

    printf("\t请输入操作指令: ");

    scanf("%d",&cid);

    getchar();

    switch(cid)

      {

        case 1:

            AllChoice(Fhead);

            break;

        case 2:

            Momory\_recycle(&Lhead,&Fhead);

            break;

        case 3:

            ShowMomory(Lhead);

            break;

        case 4:

            printf("\t感谢使用！\n");

            return 0;

            break;

        default:

            printf("\t输入指令无效");

            break;

      }

      getchar();

      // system("cls");

  }

//  Allocate(Fhead,100,2000,&Fhead);

//  Allocate(Fhead,80,2000,&Fhead);

//  //Allocate(fl,n.id,n.length,&fhead);

//  ShowMomory(Lhead);

  return 0;

}

//菜单界面

void Menu() {

  printf("\n\n");

  printf("\t┏--------------------------------------------------┓\n");

  printf("\t┃           欢迎使用动态多分区模拟程序             ┃\n");

  printf("\t┃  组员：张杰宁 李坤璘 方凡       小组：15组       ┃\n");

  printf("\t┃  1、进程分配                                     ┃\n");

  printf("\t┃  2、内存回收                                     ┃\n");

  printf("\t┃  3、显示内存                                     ┃\n");

  printf("\t┃  4、退出                                         ┃\n");

  printf("\t┃                                                  ┃\n");

  printf("\t┗--------------------------------------------------┛\n\n");

}

//MAT表初始化

List \*InitMat() {

  List \*head;

  head = (List \*)malloc(sizeof(List) \* 1);

  head->last = NULL;

  head->next = NULL;

  Node \*n = (Node \*)malloc(sizeof(Node) \* 1);

  n->id = -1;

  n->startAdd = 0;

  n->length = Max;

  n->status = 0;

  head->cur = n;

  return head;

}

//空闲分区表初始化

Flist \*InitFreeTable(List \*Mat) {

  Flist \*fhead;

  fhead = (Flist \*)malloc(sizeof(Flist) \* 1);

  fhead->last = NULL;

  fhead->next = NULL;

  fhead->list = Mat;  //指向首元素节点

  (fhead->list)->flist = fhead;

  return fhead;

}

//分配内存，具体参数定义见函数体

int Allocate(Flist \*fl, int id, int psize, Flist \*\*fhead) {

  Node \*ntemp = (fl->list)->cur;

  ntemp->id = id;

  ntemp->status = 1;  // 1表示被占用了

  //进程大小正好与当前分区相同,不用分区,但要移出空闲分区表

  if (ntemp->length == psize) {

    if (\*fhead == fl)  //说明被分配走的是表头

    {

      \*fhead = (\*fhead)->next;

      (fl->list)->flist = NULL;

      free(fl);

    } else  //不是表头

    {

      Flist \*ftemp = fl->last;

      ftemp->next = fl->next;

      if ((fl->next) != NULL) (fl->next)->last = ftemp;

      (fl->list)->flist = NULL;

      free(fl);

    }

    return 0;

  } else  //此处肯定是小于的情况

  {

    //小于情况下，需要新建Mat，修改空闲分区

    Node \*newNode = (Node \*)malloc(sizeof(Node) \* 1);

    List \*newList = (List \*)malloc(sizeof(List) \* 1);

    newNode->length = ntemp->length - psize;

    ntemp->length = psize;  //修改老节点和新节点的长度

    newNode->id = -1;       //初始化id

    //则其起始地址为 老节点起始地址加进程大小

    newNode->startAdd = ntemp->startAdd + psize;

    newNode->status = 0;

    //节点初始化完毕，开始初始化链表

    newList->cur = newNode;  //设置新链表节点的内存节点

    newList->last = fl->list;  //设置新链表节点的上一个节点为  被分割节点

    newList->next = (fl->list)->next;

    (fl->list)->next = newList;  //设置先后关系

    (fl->list)->flist = NULL;

    fl->list = newList;

    newList->flist = fl;

    return 1;

  }

  return 2;

}

//内存的回收

void Momory\_recycle(List \*\*mat, Flist \*\*Free)

{

  List\* now = \*mat;

  /\*输入要撤销的进程id，遍历mat表去查询对应的id号\*/

  while (1) {

    /\*输入要撤销的进程id\*/

    int id;

    printf("\t请输入要撤销的进程号：");

    scanf("%d", &id);

    /\*遍历mat表\*/

    if(id<0){

        printf("\t非法的id\n");

        getchar();

        return ;

    }

    now = \*mat;

    while (now!=NULL&&now->cur->id!=id) {

//      printf("aa\n");

      now = now->next;

    }

    if (now == NULL) {

      printf("\t未找到对应进程\n");

      break;

    } else {

      break;

    }

  }

  /\*检查该结点now前一地址块和后一地址块是否是空闲的\*/

  int flag = 0;  //前后有需要合并的就为0，不需要合并则为1

  if (now->last == NULL)  //链表首部

  {

    printf("a\n");

    List \*Next = now->next;

    if (Next->cur->status == 0)  //后一块地址空闲

    {

      //更改空闲分区链里指向的结点，以及让now指向原Next指向的分区链结点

      Next->flist->list = now;

      now->flist = Next->flist;

      //在mat表里删除后面的结点，将后者长度归并到前者里

      now->cur->length += Next->cur->length;

      now->cur->status = 0;

      now->next = Next->next;  //后指针连起来

      Next->next->last = now;  //前指针连起来

      free(Next);

    } else  //后一块地址不空闲

    {

      flag=1;

    }

  }

  else if (now->next == NULL)  //链表尾部

  {

    printf("b\n");

    List \*Last = now->last;

    if (Last->cur->status == 0)  //前一块地址空闲

    {

      //更改空闲分区链里指向的结点，以及让now指向原Last指向的分区链结点

      Last->flist->list = now;

      now->flist = Last->flist;

      //删除尾结点，更新尾部的结点内存地址长度

      now->cur->length+=Last->cur->length;

      now->cur->status=0;

      now->cur->startAdd=Last->cur->startAdd;

      Last->last->next=now;

      now->last=Last->last;

      free(Last);

    }

    else  //前一块地址不空闲不需要合并

    {

      flag = 1;

    }

  }

  else  //链表中间的常规结点

  {

    printf("c\n");

    List \*Next = now->next;

    List \*Last = now->last;

    if (Last->cur->status == 0)  //前一块地址空闲

    {

      //更改空闲分区链里指向的结点，以及让now指向原Last指向的分区链结点

      Last->flist->list = now;

      now->flist = Last->flist;

      //更新尾部的结点内存首地址

      now->cur->status=0;

      now->cur->startAdd=Last->cur->startAdd;

      Last->last->next=now;

      now->last=Last->last;

      if(Next->cur->status==0)//后一块地址也空闲

      {

        //内存长度为三块内存的和

        now->cur->length+=(Last->cur->length+Next->cur->length);

        //还要额外删除后结点

        Next->next->last=now;

        now->next=Next->next;

        //在空闲分区链里，Next对应的结点也要删除

        Flist \*temp=Next->flist;//Next指向的空闲分区链结点

        temp->last->next=temp->next;

        temp->next->last=temp->last;

        free(temp);

      }

      else//后一块内存不空闲

      {

        //内存长度为两块内存的和

        now->cur->length+=Last->cur->length;

      }

      free(Last);

      free(Next);

    }

    else if(Next->cur->status==0) //只有后一块地址空闲

    {

      //更改空闲分区链里指向的结点，以及让now指向原Next指向的分区链结点

      Next->flist->list = now;

      now->flist = Next->flist;

      //在mat表里删除后面的结点，将后者长度归并到前者里

      now->cur->length += Next->cur->length;

      now->cur->status = 0;

      now->next = Next->next;  //后指针连起来

      Next->next->last = now;  //前指针连起来

      free(Next);

    }

    else//这时候一定是前后内存都不空闲，也就是说不需要合并

    {

      flag=1;

    }

  }

  if(flag)//内存不需要合并，所以需要在空闲分区链里插入一个新结点

  {

      //在空闲分区链和mat表里结点联系

      Flist \*tem = (Flist \*)malloc(sizeof(struct freeList));

      tem->list = now;

      now->flist = tem;

      //头插

      tem->next = \*Free;

      (\*Free)->last = tem;

      (\*Free)=tem;

  }

}

//首次适应算法

Flist \*First\_fit(Node\* tempnode,Flist \*fl){

    Flist \*temp\_fl= fl;

    Flist \*first\_fl = NULL;

    if(temp\_fl==NULL){

        printf("NULL");

    }

    int i=1;

    while(temp\_fl!=NULL){//遍历空白链表，直到NULL

        int temp\_fl\_length =temp\_fl->list->cur->length;

        if(temp\_fl\_length>tempnode->length){

            first\_fl = temp\_fl;

            break;

        }

        else{

            temp\_fl=temp\_fl->next;

        }

        i++;

    }

    return first\_fl;

}

//最佳适应算法

Flist \*Best\_fit(Node\* tempnode,Flist\*fl){

    int max = 0;

    Flist \*temp\_fl = fl;

    Flist \*max\_fl =NULL;

    //链表寻找最大节点

    while(temp\_fl){

        int temp\_fl\_length =temp\_fl->list->cur->length;

        if(temp\_fl\_length>max){

            max = temp\_fl\_length;

            max\_fl = temp\_fl;

        }

        else{

            temp\_fl = temp\_fl->next;

        }

    }

    if(max\_fl->list->cur->length<tempnode->length){

        printf("最大空闲块<进程\n");

        return NULL;//最大空闲块仍无法放入

    }

    else{

        return max\_fl;

    }

}

//最坏适应算法

Flist \*Worst\_fit(Node\* tempnode,Flist \*fl){

    int min=0 ;

    Flist \*temp\_fl = fl;

    Flist \*min\_fl =NULL;

    //链表寻找最大节点

    while(temp\_fl){

        int temp\_fl\_length =temp\_fl->list->cur->length;

        int tempnode\_length = tempnode->length;

        if(temp\_fl\_length<min&&temp\_fl\_length>=tempnode\_length){//双重限制可装入进程前提下的最小空白块

            min = temp\_fl\_length;

            min\_fl = temp\_fl;

            break;

        }

        else{

            temp\_fl = temp\_fl->next;

        }

    }

    return min\_fl;

}

 //循环适应算法

Flist\* Circl\_fit(Node\* tempnode,Flist \*fl,Flist \*Last\_fl){//Last\_fl:上次分配出去的空闲节点

    Flist \*temp\_fl = Last\_fl;

    Flist \*fit\_fl = NULL;

    while(temp\_fl->next!=Last\_fl){//循环至初始位置结束

        int temp\_fl\_length =temp\_fl->list->cur->length;//当前遍历到的空闲块大小

        int tempnode\_length = tempnode->length;//当前进程大小

        if(temp\_fl\_length>=tempnode\_length){//可装入当前进程

            fit\_fl = temp\_fl;

            break;

        }

        else{

            if(temp\_fl->next==NULL){

                temp\_fl = fl;

            }

            else{

                    temp\_fl = temp\_fl->next;

            }

        }

    }

    return fit\_fl;

}

//输出MAT表所有内存占有情况

void ShowMomory(List \*head)

{

    Node \*nt;

    List \*temp = head;

    while(temp!=NULL)

    {

        nt = temp->cur;

        printf("\t内存地址范围：%d - %d，占用状态：%d 进程id: %d\n ",nt->startAdd,nt->startAdd+nt->length,nt->status,nt->id);

        temp = temp->next;

    }

}

//提供内存分配的选择

void AllChoice(Flist \*fhead)

{

    int id;

    printf("\t请选择分配方式:1.FirstFit 2.BestFit 3.WorstFit 4.CircleFit\n\t");

    scanf("%d",&id);

    Node n;

    printf("\t输入进程id:");

    scanf("%d",&n.id);

    getchar();

    printf("\t输入进程大小:");

    scanf("%d",&n.length);

    getchar();

    Flist \*fl = NULL;

    switch(id)

    {

        case 1:

            fl = First\_fit(&n,fhead);

            break;

        case 2:

            fl = Best\_fit(&n,fhead);

            break;

        case 3:

            fl = Worst\_fit(&n,fhead);

            break;

        default:

            printf("\t输入指令无效");

            getchar();

            break;

    }

    if(fl==NULL)

    {

        printf("\t没有可分配内存\n");

    }

    else

    {

        Allocate(fl,n.id,n.length,&fhead);

        printf("\t分配内存成功\n");

    }

}

2. 用多线程模拟多道作业对内存空间的动态申请和释放

#include<iostream>

#include<cstdio>

#include<unistd.h>

#include<pthread.h>

#include<semaphore.h>

#include<cstdlib>

#include<vector>

#include<map>

// #include <cstring>

#include<ctime>

using namespace std;

#define Max 1048576

typedef struct freeList Flist;

typedef struct node {

  int id;        //装入此内存块进程的id；

  int startAdd;  //内存块的首地址

  int length;    //内存块的长度

  unsigned char status;  //状态位，记录该分区的状态，默认为0，表示未分配，不为0表示已经分配了

} Node;

//记录内存的链表

typedef struct list {

  struct list \*last;  //链表记录的上一个节点

  struct list \*next;  //链表记录的下一个节点

  Flist \*flist;

  Node \*cur;  //链表记录的当前节点

} List;

struct freeList {

  struct freeList \*last;

  struct freeList \*next;

  List \*list;

};

pthread\_mutex\_t pmutex; //对2缓冲区的互斥变量

pthread\_t pid[20];

List \*Lhead;//MAT表

Flist \*Fhead;//空闲分区表

List \*InitMat();  //初始化mat表的函数，必须在空闲分区表之前

Flist \*InitFreeTable(List \*Mat);  //初始化空闲分区表

int Allocate(Flist \*fl, int id, int psize,Flist \*\*fhead);  //分配方法，具体参数定义见函数体 ,返回值未定义，可自定义

void Momory\_recycle(List \*\*mat, Flist \*\*Free,int id); //回收方法

Flist\* First\_fit(Node\* tempnode,Flist \*fl); //首次适应算法

Flist\* Best\_fit(Node\* tempnode,Flist\*fl);  //最佳适应算法

Flist\* Worst\_fit(Node\* tempnode,Flist \*fl);//最坏适应算法

Flist\* Circl\_fit(Node\* tempnode,Flist \*fl,Flist \*Last\_fl); //循环适应算法

void ShowMomory(List \*head);//输出MAT表所有内存占有情况

void \*ThreadAllocate(void \*arg)

{

  /\*

    1.确定线程的基本属性

    包括id、线程大小、以及线程占用时间

  \*/

  Node n;

  n.id = \*(int\*)arg;//线程id

  n.length = rand() % 1000 + 100; //线程大小为100-1100

  int taketime = rand() % 10+1; //线程随机先等待1-10个clock

  sleep(taketime);

  /\*

    2.选择内存分配

    随机选择一种内存分配方式(First\_fit Best\_fit Worst\_fit Circl\_fit)

  \*/

  int choice= rand() %3+1;

  Flist \*fl = NULL;

  pthread\_mutex\_lock(&pmutex);//P操作锁住互斥变量

    switch(choice)

    {

        case 1:

            fl = First\_fit(&n,Fhead); //First\_fit

            break;

        case 2:

            fl = Best\_fit(&n,Fhead); //Best\_fit

            break;

        case 3:

            fl = Worst\_fit(&n,Fhead); //Worst\_fit

            break;

    }

  if(fl==NULL)

    printf("\t没有可分配内存\n");

    else

    {

        Allocate(fl,n.id,n.length,&Fhead);

        printf("\t分配内存成功\n");

    }

  printf("\t分配内存后目前所有内存占有情况：\n");

  /\*

    3.分配一次后就输出一次目前内存状况

  \*/

  ShowMomory(Lhead);

  pthread\_mutex\_unlock(&pmutex);//V操作解锁互斥变量

  sleep(10);//分配内存后停留3个clock再模拟回收内存

  /\*

    4.模拟内存的回收

  \*/

  pthread\_mutex\_lock(&pmutex);//P操作锁住互斥变量

  Momory\_recycle(&Lhead,&Fhead,n.id);//内存回收

  printf("\t 回收内存后目前所有内存占有情况：\n");

  /\*

    5.回收一次后就输出一次目前内存状况

  \*/

  ShowMomory(Lhead);

  pthread\_mutex\_unlock(&pmutex);//V操作解锁互斥变量

  sleep(10);//分配内存后停留3个clock再结束此次操作

  return NULL;

}

int Index[20];

int main()

{

    srand(time(0));

    Lhead = InitMat();

    Fhead = InitFreeTable(Lhead);

    pthread\_mutex\_init(&pmutex,NULL);

    for (int i = 0; i < 10; i++)

    {

        Index[i] = i;

        //ThreadAllocate((void\*)&i);

        pthread\_create(&pid[i],NULL,ThreadAllocate,&Index[i]);

    }

    for(int i=0;i<10;i++)

  {

    pthread\_join(pid[i],NULL);

  }

  return 0;

}

//MAT表初始化

List \*InitMat() {

  List \*head;

  head = (List \*)malloc(sizeof(List) \* 1);

  head->last = NULL;

  head->next = NULL;

  Node \*n = (Node \*)malloc(sizeof(Node) \* 1);

  n->id = -1;

  n->startAdd = 0;

  n->length = Max;

  n->status = 0;

  head->cur = n;

  return head;

}

//空闲分区表初始化

Flist \*InitFreeTable(List \*Mat) {

  Flist \*fhead;

  fhead = (Flist \*)malloc(sizeof(Flist) \* 1);

  fhead->last = NULL;

  fhead->next = NULL;

  fhead->list = Mat;  //指向首元素节点

  (fhead->list)->flist = fhead;

  return fhead;

}

//分配内存，具体参数定义见函数体

int Allocate(Flist \*fl, int id, int psize, Flist \*\*fhead) {

  Node \*ntemp = (fl->list)->cur;

  ntemp->id = id;

  ntemp->status = 1;  // 1表示被占用了

  //进程大小正好与当前分区相同,不用分区,但要移出空闲分区表

  if (ntemp->length == psize) {

    if (\*fhead == fl)  //说明被分配走的是表头

    {

      \*fhead = (\*fhead)->next;

      (fl->list)->flist = NULL;

      free(fl);

    } else  //不是表头

    {

      Flist \*ftemp = fl->last;

      ftemp->next = fl->next;

      if ((fl->next) != NULL) (fl->next)->last = ftemp;

      (fl->list)->flist = NULL;

      free(fl);

    }

    return 0;

  } else  //此处肯定是小于的情况

  {

    //小于情况下，需要新建Mat，修改空闲分区

    Node \*newNode = (Node \*)malloc(sizeof(Node) \* 1);

    List \*newList = (List \*)malloc(sizeof(List) \* 1);

    newNode->length = ntemp->length - psize;

    ntemp->length = psize;  //修改老节点和新节点的长度

    newNode->id = -1;       //初始化id

    //则其起始地址为 老节点起始地址加进程大小

    newNode->startAdd = ntemp->startAdd + psize;

    newNode->status = 0;

    //节点初始化完毕，开始初始化链表

    newList->cur = newNode;  //设置新链表节点的内存节点

    newList->last = fl->list;  //设置新链表节点的上一个节点为  被分割节点

    newList->next = (fl->list)->next;

    (fl->list)->next = newList;  //设置先后关系

    (fl->list)->flist = NULL;

    fl->list = newList;

    newList->flist = fl;

    return 1;

  }

  return 2;

}

//内存的回收

void Momory\_recycle(List \*\*mat, Flist \*\*Free,int id)

{

  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

  List\* now = Lhead;

  // while (now!=NULL&&now->cur->id!=id)

  //   now = now->next;

    printf("+++++++++++++++++++++++++++\n");

    while (1) {

    /\*遍历mat表\*/

    if(id<0){

        printf("\t非法的id\n");

        getchar();

        return ;

    }

    now = Lhead;

    while (now!=NULL&&now->cur->id!=id) {

//      printf("aa\n");

      now = now->next;

    }

    if (now == NULL) {

      printf("\t未找到对应进程\n");

      break;

    } else {

      break;

    }

  }

  /\*检查该结点now前一地址块和后一地址块是否是空闲的\*/

  int flag = 0;  //前后有需要合并的就为0，不需要合并则为1

  if (now->last == NULL)  //链表首部

  {

    printf("a\n");

    List \*Next = now->next;

    if (Next->cur->status == 0)  //后一块地址空闲

    {

      //更改空闲分区链里指向的结点，以及让now指向原Next指向的分区链结点

      Next->flist->list = now;

      now->flist = Next->flist;

      //在mat表里删除后面的结点，将后者长度归并到前者里

      now->cur->length += Next->cur->length;

      now->cur->status = 0;

      now->next = Next->next;  //后指针连起来

      Next->next->last = now;  //前指针连起来

      free(Next);

    } else  //后一块地址不空闲

    {

      flag=1;

    }

  }

  else if (now->next == NULL)  //链表尾部

  {

    printf("b\n");

    List \*Last = now->last;

    if (Last->cur->status == 0)  //前一块地址空闲

    {

      //更改空闲分区链里指向的结点，以及让now指向原Last指向的分区链结点

      Last->flist->list = now;

      now->flist = Last->flist;

      //删除尾结点，更新尾部的结点内存地址长度

      now->cur->length+=Last->cur->length;

      now->cur->status=0;

      now->cur->startAdd=Last->cur->startAdd;

      Last->last->next=now;

      now->last=Last->last;

      free(Last);

    }

    else  //前一块地址不空闲不需要合并

    {

      flag = 1;

    }

  }

  else  //链表中间的常规结点

  {

    printf("c\n");

    List \*Next = now->next;

    List \*Last = now->last;

    if (Last->cur->status == 0)  //前一块地址空闲

    {

      //更改空闲分区链里指向的结点，以及让now指向原Last指向的分区链结点

      Last->flist->list = now;

      now->flist = Last->flist;

      //更新尾部的结点内存首地址

      now->cur->status=0;

      now->cur->startAdd=Last->cur->startAdd;

      Last->last->next=now;

      now->last=Last->last;

      if(Next->cur->status==0)//后一块地址也空闲

      {

        //内存长度为三块内存的和

        now->cur->length+=(Last->cur->length+Next->cur->length);

        //还要额外删除后结点

        Next->next->last=now;

        now->next=Next->next;

        //在空闲分区链里，Next对应的结点也要删除

        Flist \*temp=Next->flist;//Next指向的空闲分区链结点

        temp->last->next=temp->next;

        temp->next->last=temp->last;

        free(temp);

      }

      else//后一块内存不空闲

      {

        //内存长度为两块内存的和

        now->cur->length+=Last->cur->length;

      }

      free(Last);

      free(Next);

    }

    else if(Next->cur->status==0) //只有后一块地址空闲

    {

      //更改空闲分区链里指向的结点，以及让now指向原Next指向的分区链结点

      Next->flist->list = now;

      now->flist = Next->flist;

      //在mat表里删除后面的结点，将后者长度归并到前者里

      now->cur->length += Next->cur->length;

      now->cur->status = 0;

      now->next = Next->next;  //后指针连起来

      Next->next->last = now;  //前指针连起来

      free(Next);

    }

    else//这时候一定是前后内存都不空闲，也就是说不需要合并

    {

      flag=1;

    }

  }

  if(flag)//内存不需要合并，所以需要在空闲分区链里插入一个新结点

  {

      //在空闲分区链和mat表里结点联系

      Flist \*tem = (Flist \*)malloc(sizeof(struct freeList));

      tem->list = now;

      now->flist = tem;

      //头插

      tem->next = Fhead;

      Fhead->last = tem;

      Fhead=tem;

  }

}

//首次适应算法

Flist \*First\_fit(Node\* tempnode,Flist \*fl){

    Flist \*temp\_fl= fl;

    Flist \*first\_fl = NULL;

    if(temp\_fl==NULL){

        printf("NULL");

    }

    int i=1;

    while(temp\_fl!=NULL){//遍历空白链表，直到NULL

        int temp\_fl\_length =temp\_fl->list->cur->length;

        if(temp\_fl\_length>tempnode->length){

            first\_fl = temp\_fl;

            break;

        }

        else{

            temp\_fl=temp\_fl->next;

        }

        i++;

    }

    return first\_fl;

}

//最佳适应算法

Flist \*Best\_fit(Node\* tempnode,Flist\*fl){

    int max = 0;

    Flist \*temp\_fl = fl;

    Flist \*max\_fl =NULL;

    //链表寻找最大节点

    while(temp\_fl){

        int temp\_fl\_length =temp\_fl->list->cur->length;

        if(temp\_fl\_length>max){

            max = temp\_fl\_length;

            max\_fl = temp\_fl;

        }

        else{

            temp\_fl = temp\_fl->next;

        }

    }

    if(max\_fl->list->cur->length<tempnode->length){

        printf("最大空闲块<进程\n");

        return NULL;//最大空闲块仍无法放入

    }

    else{

        return max\_fl;

    }

}

//最坏适应算法

Flist \*Worst\_fit(Node\* tempnode,Flist \*fl){

    int min=0 ;

    Flist \*temp\_fl = fl;

    Flist \*min\_fl =NULL;

    //链表寻找最大节点

    while(temp\_fl){

        int temp\_fl\_length =temp\_fl->list->cur->length;

        int tempnode\_length = tempnode->length;

        if(temp\_fl\_length<min&&temp\_fl\_length>=tempnode\_length){//双重限制可装入进程前提下的最小空白块

            min = temp\_fl\_length;

            min\_fl = temp\_fl;

            break;

        }

        else{

            temp\_fl = temp\_fl->next;

        }

    }

    return min\_fl;

}

 //循环适应算法

Flist\* Circl\_fit(Node\* tempnode,Flist \*fl,Flist \*Last\_fl){//Last\_fl:上次分配出去的空闲节点

    Flist \*temp\_fl = Last\_fl;

    Flist \*fit\_fl = NULL;

    while(temp\_fl->next!=Last\_fl){//循环至初始位置结束

        int temp\_fl\_length =temp\_fl->list->cur->length;//当前遍历到的空闲块大小

        int tempnode\_length = tempnode->length;//当前进程大小

        if(temp\_fl\_length>=tempnode\_length){//可装入当前进程

            fit\_fl = temp\_fl;

            break;

        }

        else{

            if(temp\_fl->next==NULL){

                temp\_fl = fl;

            }

            else{

                    temp\_fl = temp\_fl->next;

            }

        }

    }

    return fit\_fl;

}

//输出MAT表所有内存占有情况

void ShowMomory(List \*head)

{

    Node \*nt;

    List \*temp = head;

    while(temp!=NULL)

    {

        nt = temp->cur;

        printf("\t内存地址范围：%d - %d，占用状态：%d 进程id: %d\n ",nt->startAdd,nt->startAdd+nt->length,nt->status,nt->id);

        temp = temp->next;

    }

        printf("\n");

}

五．调试方案及调试过程问题分析

由于任务一是有手动操作界面的，因此在此对其调试过程进行介绍：

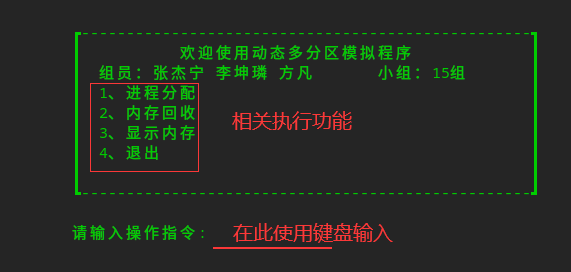
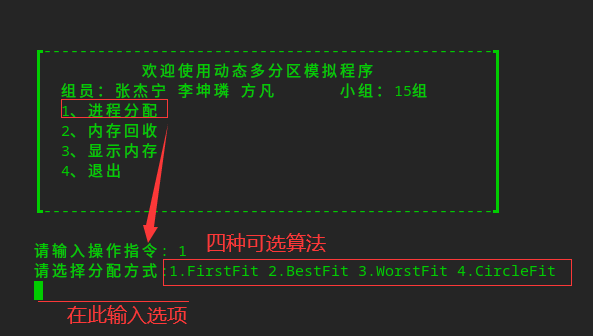
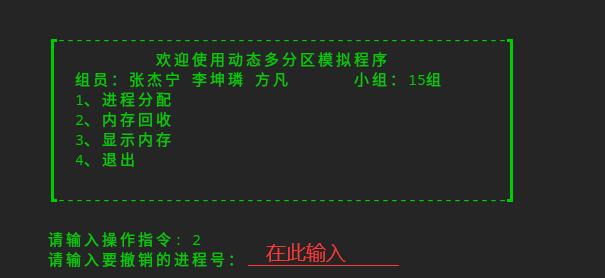


图5.1 开始界面



5.2 进程分配界面



5.3 内存回收界面

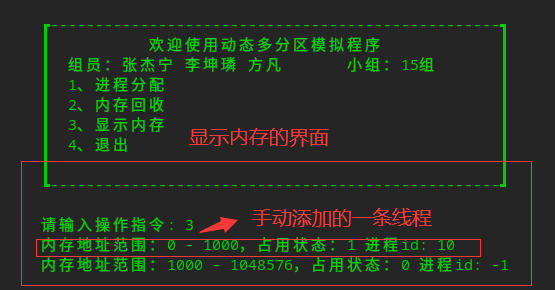


图5.4 内存显示界面

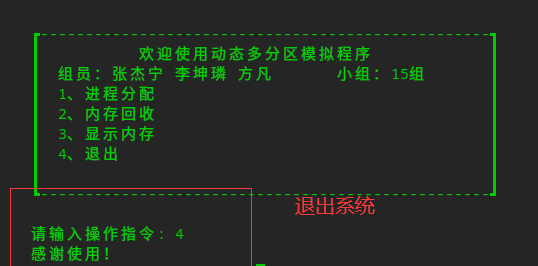


图5.5 退出界面

六．运行结果与分析

1.任务一：

（1）分配初始内存为1048476KB，显示主存分配情况

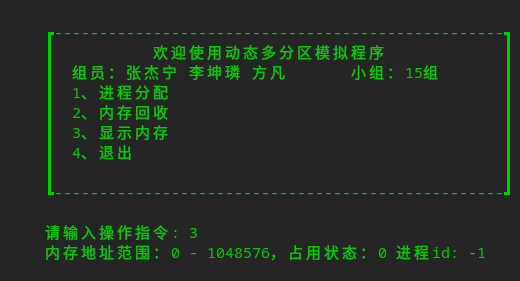


图6.1 显示主存分配情况

（2）进行分配主存，并显示分配结果

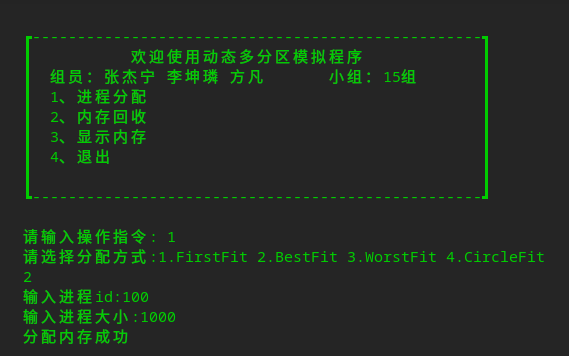


图6.2 进行分配主存

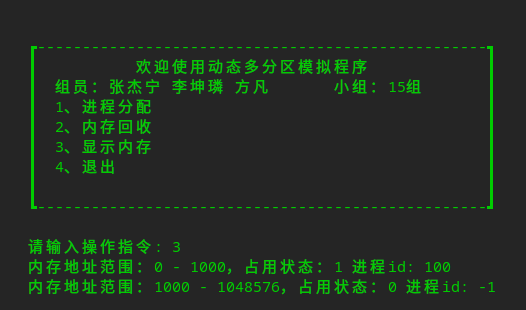


图6.3 显示主存分配情况

（3）继续进行分配并显示分区表分区情况

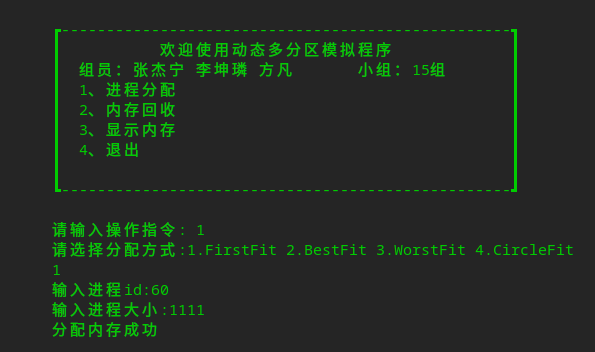
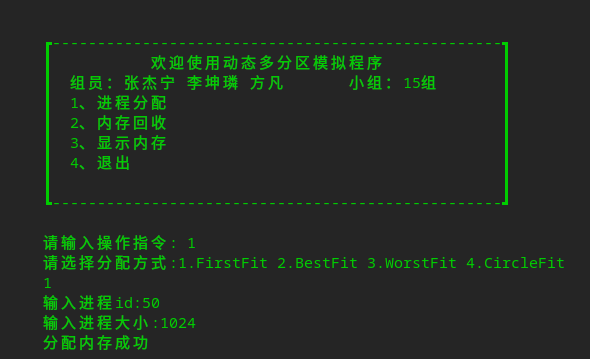
 

图6.4 多组内存分配数据

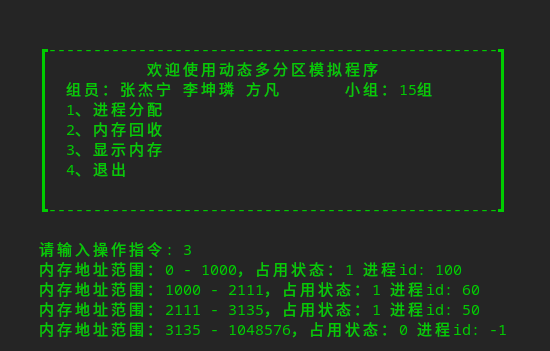


图6.5 显示主存分配状况

（4）继续进行分配，若分配大小大于空闲分区空间，则分配失败，显示分配失败字样，分区表不做变化

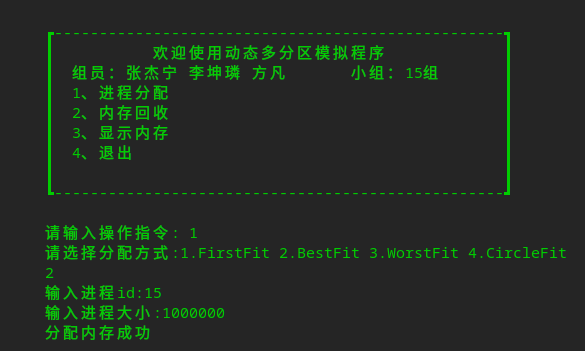
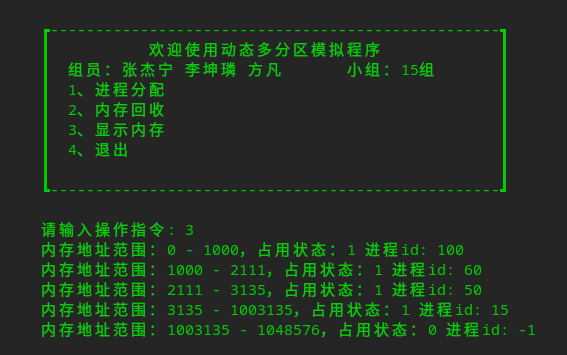
 

图6.6 分配一个超大内存线程占满内存

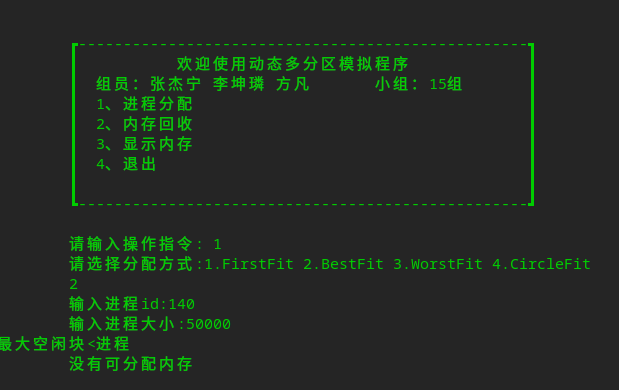


图6.7 内存分配失败字样

（5）测试回收功能，选择回收区号，显示回收后主存分区情况

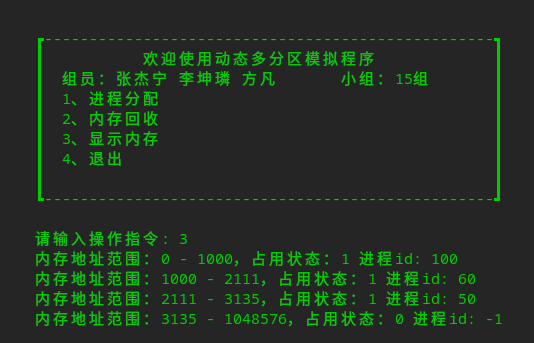


图6.8 当前内存分配情况

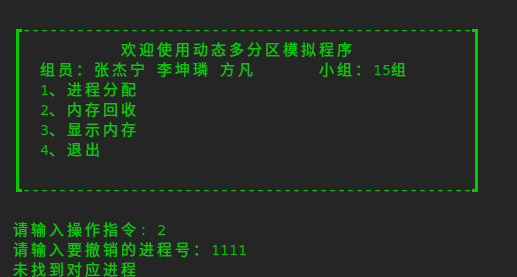


图6.9 当输入一个错误id的情况

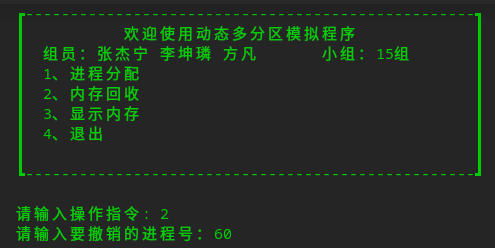


图6.10 撤销id为60的线程



图6.11 内存占用状态变为空闲

（6）退出功能，程序结束



图6.12 结束界面

任务二：

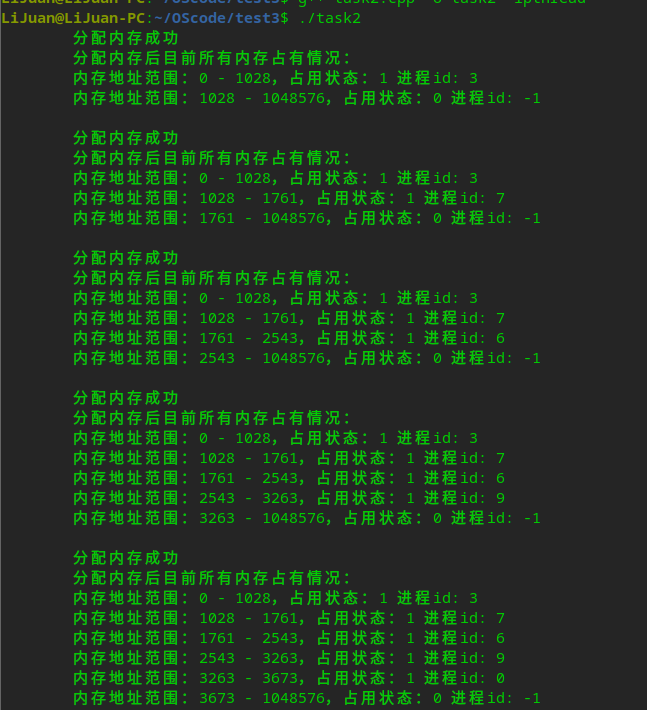


图6.13 多线程同步分配的过程

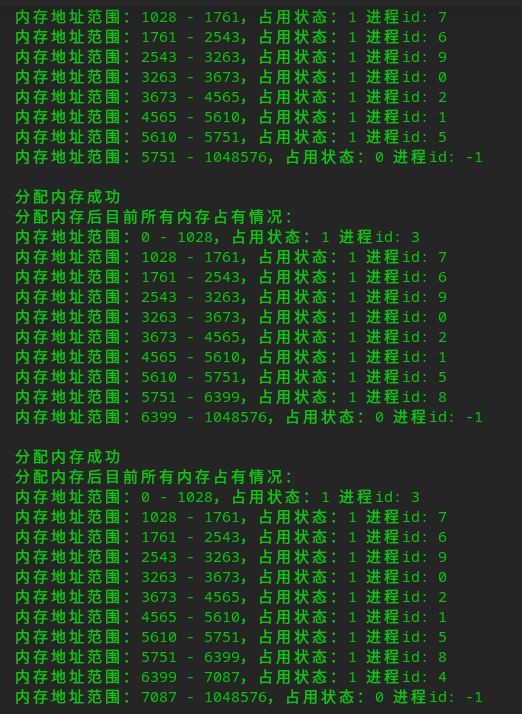


图6.14 多线程同步分配的过程节选

我们可以看见内存的分配过程为随机，进程进入的顺序也为随机

下面显示内存随机回收的过程：id为8的线程被回收



图6.15 第一个线程回收结果



图6.16 第二个线程(id=9)回收结果

以此类推，所有线程回收的结果如下：

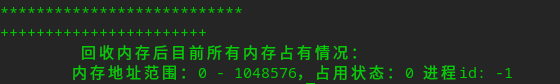


图6.17 所有线程回收结果

因为线程回收过程中，若相邻区域的内存为空闲，会发生内存空间的合并，因此最终仅有一条内存地址范围输出结果。

2.运行结果分析

对比可得，多线程的内存分配回收本质就在于是使用随机数据代替我们的手动输入，其本质总体结果是类似的。我们在内存分配会去检查是否能分配，在回收的时候检查相邻处是否能合并，也能实现自动的内存空间归并。

但是多线程的速度相对而言还是慢了一点，因为创建了10个线程依旧是涉及到CPU的多次调度。

七．总结

1.设计中的经验体会

在此次实验中出现不少问题，主要体现在内存回收通过程序模拟实验上的特殊要求，比如空闲表区与已分分区表的数据结构定义，还有就是回收时程序浮动遇到的问题。

内存分配：对于不同分配函数所遍历的策略不同，对于空闲分区链的选择具有不同的侧重点，最后选择一个相同的分配函数接口，在空闲分区链中遍历，主要是关注头结点和尾结点很特殊。

内存回收：虽然只是一个函数，但是通过代码可以看见其中的逻辑比较复杂，由于头结点和尾结点的存在需要去考虑不同情况下的链表连接，因此在写第一版本的代码时经常出现段错误也就是地址越界指针指向随机地址，在调试的过程中比较费力。

最后对于动态多分区模拟程序，这次实验只是简单的对动态多分区进行实现，由于编程时的疏忽，没有与进程相关联起来，所以用户应该是输入自己的进程的pid来进行分配与回收，如此的话，又对实现增加了一些难度，而且更降低了数组的效率，因为没办法随机读取了，而用动态数组想要降低系统开销，又需要增加一些新的数据结构，所以用链表更优。

通过本次实验，对动态多分区的存储管理过程有了一个更加细致的认知。

2.后续设计设想

（1）增添更多的可视化页面，将动态多分区存储管理系统变成可视化描述过程的教学系统。

（2）尝试设计更加灵活的动态多分区进阶为分页式存储管理、分段式、以及段页式存储管理。

课程设计选题四 请求分页存储管理模拟系统

一．设计要求

1.数据要求

输入：算法（1--OPT，2--FIFO，3--LRU）、内存块数

页面序列（页面1,页面2,页面3,...）

输出：页面变化时内存块装入页面列表1-是否命中/页面变化时内存块装入页面列表2-是否命中/...缺页次数

其中页面变化时内存块装入页面列表：

(1) 内存块1装入页面,内存块2装入页面,内存块3装入页面...，未装入任何页面时由"-”表示

(2) 是否命中：1-命中，0-缺页

2.功能要求

（1）编写并调试完成请求页式存储管理程序。

（2）掌握其中最基本的三种算法，包括FIFO、LRU及OPT。

（3）对于同一个页面引用串，要求能够调用不同的算法对它进行操作。

（4）每个页面访问可用交互式进行，也可照事先随机生成页面访问序列进行。

（5）学会使用linux中文件夹的管理操作

（6）每个页面访问的结果以命中、直接装入和替换来记录。

二．背景知识及设计思路

1.背景知识

**（1）Linux 的线程创建**

使用pthread\_create 函数创建用户级线程

**函数定义：**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr，void \*(\*start\_rtn) (void\*)，void \* arg);

**参数说明：**

thread: 待创建线程的 id 指针

attr: 创建线程时的线程属性

v(\*start\_rtin)(void\*): 返回值是 void\*类型的指针函数

arg: start\_routine 的参数

**返回值：**

成功返回 0

失败返回错误编号

 EAGAIN：表示系统限制创建新的线程，如线程数目过多

 EINVAL：代表线程属性值非法

**（2）线程同步终止**

**函数定义：**

int pthread\_join( pthread\_t thread, void \*\* rval\_ptr); **功能：**

①调用者将挂起并等待新进程终止

②当新线程调用 pthread\_exit()退出或者 return 时，进程中的其他线程可通过pthread\_join()获得进程的退出状态

**返回值：**

成功结束返回值为 0,否则为错误编码

**（3）带有线程程序的编译**

**基于 POSIX 标准的线程编程接口：**

 包括一个 pthread.h 头文件和一个线程库

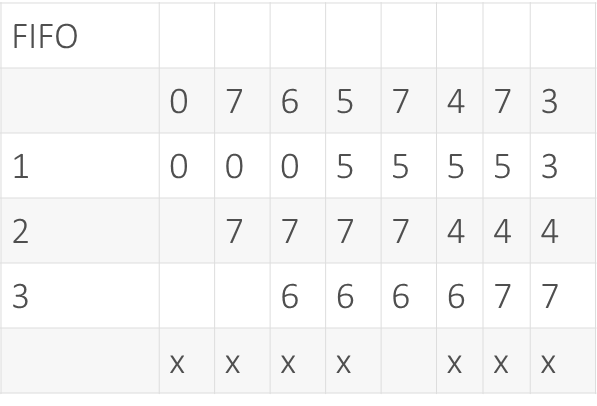
**编译方法：**

 g++ \*\*.c -o \*\*\* –lpthread

.ddir(DIR \* dir);

**（4）FIFO**

总是选择在主存中停留时间最长的一页置换，即**先进入内存的页，先退出内存**。



**（5）OPT**

最优置换算法是在理论上提出的一种算法。其实质是：当调入新的一页而必须预先置换某个旧页时，所选择的旧页应是将来不再被使用，或者是在最远的将来才被访问。



**（6）LRU**

最近最少使用算法LRU算法是与每个页面最后使用的时间有关的。

当必须置换一个页面时，LRU算法选择过去一段时间里最近最少使用的页面。

其总是选择在主存中停留时间最长的一页置换，即先进入内存的页，先退出内存。



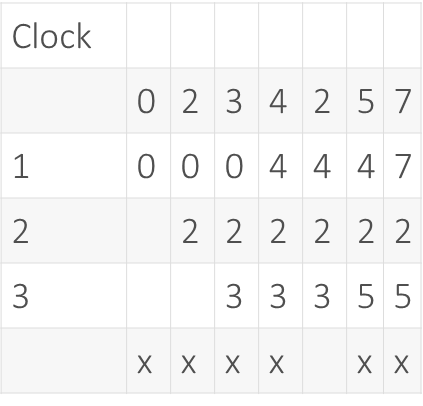
**（7）SCR**

第二次机会算法的基本思想是与FIFO相同的，但是有所改进，避免把经常使用的页面置换出去。当选择置换页面时，检查它的访问位。如果是0，就淘汰这页；如果访问位是1，就给它第二次机会，并选择下一个FIFO页面。当一个页面得到第二次机会时，它的访问位就清为0，它的到达时间就置为当前时间。如果该页在此期间被访问过，则访问位置1。在最坏的情况下，所有的访问位都是1，指针要通过整个队列一周，每个页都给第二次机会。这时就退化成FIFO算法了。



（8）CLOCK

在时钟置换算法里，页表项中的设置一个访问位，当页面被加载到内存中时，该位被置为0，当该页面被访问（读/写）时该位被置为1。然后将各个页面组织成一个环形的链表，定义一个指针指向最先进驻内存的页面，当发生缺页中断时先观察指针指向的当前位置，如果访问位为0就与需要的页面进行置换，如果访问位为1将该位置为0，然后指向指向链表下一个元素，进行同样的操作直到找到目标位置，置换后将指针移动到下一个元素。



2.设计思路

（1）每次输入的页面数据只有页面序号

（2）在算法中需要计算每个页面访问过之后的优先级变化

（3）记录当前页面下一次访问所需要的距离信息

（4）采用结构体数组的形式将需要用到的信息全部存储起来。

（5）临时数组在 LRU 算法中起辅助输出的作用

三．设计方案

1.数据结构设计

任务一：

#define BUSY 1//占用块标志

#define IDLE 0//空闲块标志

#define COLOR\_Exist 0//存在该页面，命中

#define COLOR\_NotExist\_IDLE 1//不存在该页面，但有空闲块

#define COLOR\_NotExist\_NoIDLE 2//不存在该页面，且不存在空闲块

int pages[1000];

//页面数据结构

typedef struct \_Page {

    int pageID;//页号

} Page;

//页面队列

typedef struct \_PageQueue {

    Page page;

    struct \_PageQueue \*next;//指针代表下一页面

} PageQueue;

//块队列中基本元素

typedef struct \_Block {

    Page \*page; //页框

    long time; //最后访问时间

    int state; //页块是否空闲，0代表未使用，1代表已被使用

    int A;// 访问位，如果访问位为1则再给一次机会，如果为0则直接替换

    int visit;

} Block;

//块队列以链表表示

typedef struct \_BlockQueue {

    Block block;

    struct \_BlockQueue \*next;

} BlockQueue;

//描述整体的进程结构

typedef struct process {

    PageQueue pages; //页面

    unsigned int pageLength; // 页面数

} process;//进程

任务二：

**由于本实验的两个任务思路统一，但是代码实现仍有不小差异，因此我们小组两个人分别独立完成该两项任务，因此二人所创建的数据结构也有区别。**

const int frame\_size = 100;

//记录的页表项

struct Page

{

    //页表项

    int page\_id = -1;  //逻辑页面号

    bool state = 0;    //状态位

    bool visit = 0;    //访问位

    bool change = 0;   //修改位

    // struct Page\* next=NULL;

};

//存储的内存帧

struct Page\_frame

{

    int tid = -1; //进程号

    int frame\_id; //占用进程的页号

    Page\* ppage = NULL;

    struct Page\_frame\* map = NULL;

    struct Page\_frame\* next = NULL;

};

//线程（进程）的载体Choose\_occ

struct PCB

{

    int tid = -1;

    int page\_num;

    int bmap\_num;

    struct Page\_frame\* next = NULL;

    int index = 0; //记录当前访问到的位置

    int process[frame\_size];

};

Page\_frame\* frame\_list = new Page\_frame; //内存的物理帧位图

Page \*page; //内存的位示图

int number[10];//待访问页面号序列

int nc\_size = 0;//内存大小

int page\_size = 0;//页面大小

int number\_size = 0; //线程/进程大小

int buffer\_size;//定义的缓冲区的大小

pthread\_mutex\_t list\_mutex;

pthread\_mutex\_t page\_mutex;

2.程序结构设计（模块图，流程图，函数调用关系图）

main（）函数是主程序的入口，控制程序流程，并按照输入的调度信号选择相应的算法模块进行运行

FIFO（）函数是先进先出算法，根据页面的到来顺序进行页面置换

LRU（）函数是最近最久未使用算法，根据页面的使用情况进行页面调度，这里使用了临时数组来辅助信息输出

input（）函数是输入函数，接受程序输入

output（）函数是输出函数，将页面命中与缺页置换的信息进行输出

OPT（）函数是最佳置换算法，根据已知的页面序列和优先级顺序算出最佳的页面调度方案

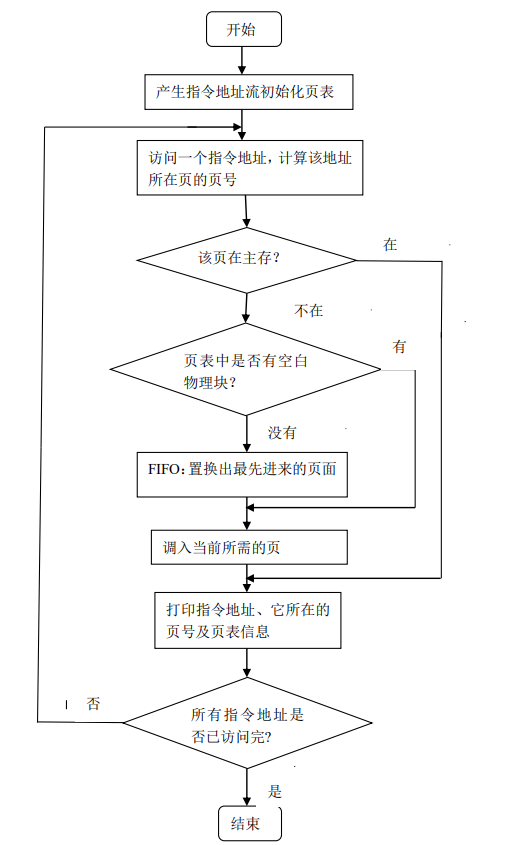


图3.1 FIFO算法流程

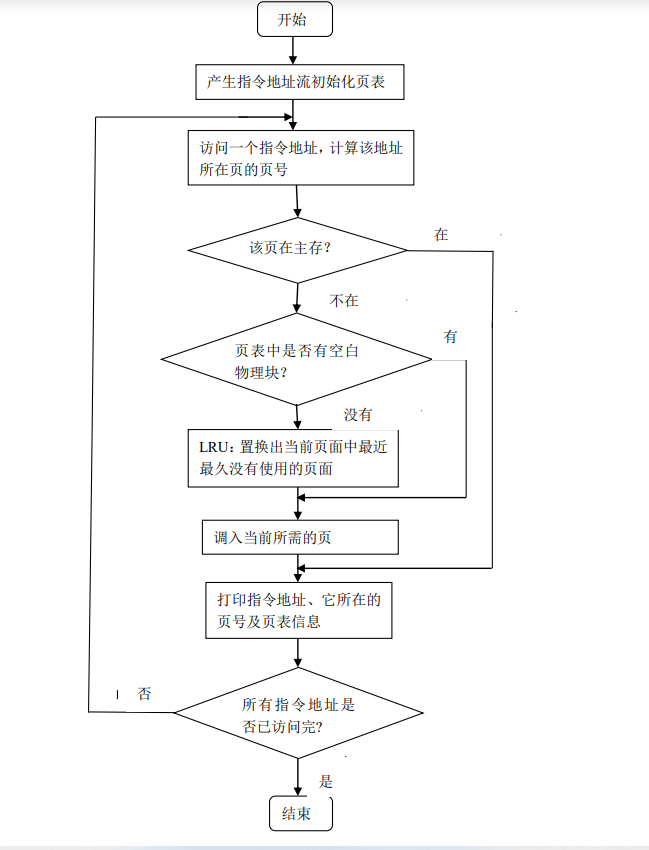


图3.2 LRU算法流程



图3.3 OPT算法流程

3.用到的Linux系统调用函数说明

在多线程实现请求分页存储时，我们涉及到线程的并发控制

**（1）pthread\_mutex\_init函数**

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,const pthread\_mutexattr\_t \*restrict attr);

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

该函数用于C函数的多线程编程中，互斥锁的初始化。以动态方式创建互斥锁的，参数attr指定了新建互斥锁的属性。如果参数attr为空(NULL)，则使用默认的互斥锁属性，默认属性为快速互斥锁 。互斥锁的属性在创建锁的时候指定，在LinuxThreads实现中仅有一个锁类型属性。

**（2）pthread\_create函数**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr，void \*(\*start\_rtn) (void\*)，void \* arg);

用于普通用户创建线程，成功返回 0，失败返回错误编号

 EAGAIN：表示系统限制创建新的线程，如线程数目过多

 EINVAL：代表线程属性值非法

**（3）pthread\_join函数**

int pthread\_join( pthread\_t thread, void \*\* rval\_ptr);

①调用者将挂起并等待新进程终止

②当新线程调用 pthread\_exit()退出或者 return 时，进程中的其他线程可通过pthread\_join()获得进程的退出状态，成功结束返回值为 0,否则为错误编码。

**（4）pthread\_mutex\_lock函数**

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

对互斥量加锁，如果该互斥量已经被锁住，函数调用失败，返回EBUSY，否则加锁成功返回0，线程不会被阻塞。

**（5）pthread\_mutex\_unlock函数**

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

对互斥量解锁。

四．设计源代码

1.模拟一个进程执行过程中请求分页存储管理的请求调页和页面置换功能

#include<iostream>

#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>

#include<sys/time.h>

#include<unistd.h>

#include<stdlib.h>

using namespace std;

#define BUSY 1//占用块标志

#define IDLE 0//空闲块标志

#define COLOR\_Exist 0//存在该页面，命中

#define COLOR\_NotExist\_IDLE 1//不存在该页面，但有空闲块

#define COLOR\_NotExist\_NoIDLE 2//不存在该页面，且不存在空闲块

int pages[1000];

//页面数据结构

typedef struct \_Page {

    int pageID;//页号

} Page;

//页面队列

typedef struct \_PageQueue {

    Page page;

    struct \_PageQueue \*next;//指针代表下一页面

} PageQueue;

//块队列中基本元素

typedef struct \_Block {

    Page \*page; //页框

    long time; //最后访问时间

    int state; //页块是否空闲，0代表未使用，1代表已被使用

    int A;// 访问位，如果访问位为1则再给一次机会，如果为0则直接替换

    int visit;

} Block;

//块队列以链表表示

typedef struct \_BlockQueue {

    Block block;

    struct \_BlockQueue \*next;

} BlockQueue;

//描述整体的进程结构

typedef struct process {

    PageQueue pages; //页面

    unsigned int pageLength; // 页面数

} process;//进程

int t = 0;

BlockQueue \*InitBlockQueue(int size);//初始化内存物理块队列，把首地址返回，如果分配失败返回NULL

int GetBlockQueueSize(BlockQueue \*blockQueue);//获取块长度

void ClearBlockQueue(BlockQueue \*blockQueue);//清空块内容

PageQueue \*InitializePageQueueWithInput(unsigned int pageLength);//初始化页面，把首地址返回。如果分配失败，返回NULL

void InitializeProcessWithInput(process \*proc, unsigned int pageSize);//初始化进程，接收手动输入的页面访问序列

BlockQueue \*SearchPage(BlockQueue \*blockQueue, Page page);//搜索目标指定页面

BlockQueue \*SearchIdleBlock(BlockQueue \*blockQueue);//搜索空闲块

int GetBlockLable(BlockQueue \*blockQueue, BlockQueue \*goalBlock);//返回块号

BlockQueue \*GetOldestBlock(BlockQueue \*blockQueue);//查找停留最久页面

BlockQueue \*GetLongestWithoutAccess(BlockQueue \*blockQueue, PageQueue \*currentPage);//OPT算法

BlockQueue \*GiveSecondChance(BlockQueue \*blockQueue);//SCR算法

BlockQueue \*GetClockBlock(BlockQueue \*blockQueue,PageQueue \*currentPage);//获取clock算法中的块队列元素信息

void PrintBlockList(BlockQueue \*blockQueue, int pageID, int color);//打印物理块块信息

void FIFO(BlockQueue \*blockQueue, process \*proc);//先进先出算法，选择在内存中驻留时间最久的页面予以淘汰，即调用GetOldestBlock函数

void SCR(BlockQueue \*blockQueue, process \*proc);//SCR算法

void LRU(BlockQueue \*blockQueue, process \*proc);//LRU最近最久未使用

void OPT(BlockQueue \*blockQueue, process \*proc);//OPT最佳页面替换算法

void Clock(BlockQueue \*blockQueue, process \*proc);//时钟置换算法

void Menu();//基本展示菜单

int True,False;

int main() {

    while(1){

        int t=0;

        True=False=0;

        unsigned int blockNum;//物理块数

        unsigned int pageNum;//进程页面数

        printf("请输入主存的物理块数(页框个数)：");

        scanf("%u", &blockNum);//3

        printf("请输入程序长度(页面数)：");

        scanf("%u", &pageNum);//8

        BlockQueue \*blocks;

        process proc;

        cout<<"请输入页面序列："<<endl;

         for (int i = 0; i < pageNum; ++i) {

                scanf("%d",&pages[i]);//0 7 6 5 7 4 7 6

            }

        InitializeProcessWithInput(&proc,pageNum);

        blocks = InitBlockQueue(blockNum);

        Menu();

        int oper;

        while (scanf("%d",&oper)){

            int flag = 0;

            printf("\n===========================================================\n");

            for (int i = 1; i<=blockNum; i++) {

                printf("| 物理块%d  | ",i);

            }

            printf("\n");

            switch (oper){

                case 1:

                    OPT(blocks, &proc);

                    break;

                case 2:

                    FIFO(blocks, &proc);

                    break;

                case 3:

                    LRU(blocks, &proc);

                    break;

                case 4:

                    SCR(blocks, &proc);

                    break;

                case 5:

                    Clock(blocks, &proc);

                    break;

                case 0:

                    flag = 1;

                    break;

                default:

                    Menu();

                    printf("非法输入!请重新输入：");

                    break;

            }

            if(flag!=1) printf("缺页率为%.2f\n",(1.0\*False)/(True+False));

            if (flag == 1) break;

            ClearBlockQueue(blocks);

            Menu();

        }

        printf("是否重新输入程序页面序列? y/n\n");

        getchar();

        char c=getchar();

        if(c=='y') continue;

        else break;

    }

}

//初始化内存物理块队列，把首地址返回，如果分配失败返回NULL

BlockQueue \*InitBlockQueue(int size) {

    BlockQueue \*block = NULL, \*p = NULL, \*q = NULL;

    for (int i = 0; i < size; i++) {

        p = (BlockQueue \*) malloc(sizeof(BlockQueue));

        p->block.page = NULL;//页面

        p->block.state = 0;//页块空闲

        p->block.time = 0;//最后访问时间

        p->block.A = 0;

        p->next = NULL;

        if (block == NULL) block = p;//头结点

        else q->next = p;

        q = p;

    }

    return block;//返回首地址

}

//获取块长度

int GetBlockQueueSize(BlockQueue \*blockQueue) {

    BlockQueue \*currentBlock = blockQueue;

    int blockQueueSize = 0;

    while (currentBlock != NULL) {

        blockQueueSize++;

        currentBlock = currentBlock->next;

    }

    return blockQueueSize;

}

//清空块内容

void ClearBlockQueue(BlockQueue \*blockQueue) {

    BlockQueue \*currentBlock = blockQueue;

    while (currentBlock != NULL) {

        currentBlock->block.page = NULL;

        currentBlock->block.state = IDLE;

        currentBlock->block.time = 0;

        currentBlock = currentBlock->next;

    }

}

//初始化页面，把首地址返回。如果分配失败，返回NULL

PageQueue \*InitializePageQueueWithInput(unsigned int pageLength) {

    PageQueue \*head = NULL, \*p = NULL, \*q = NULL;

    for (int i = 0; i < pageLength; i++) {

        p = (PageQueue \*) malloc(sizeof(PageQueue));

        p->page.pageID = pages[i];

        p->next = NULL;

        printf("%d ", p->page.pageID);

        if (head == NULL) head = p;

        else q->next = p;

        q = p;

    }

    printf("\n");

    return head;

}

//初始化进程，接收手动输入的页面访问序列

void InitializeProcessWithInput(process \*proc, unsigned int pageSize) {

    printf("进程初始化：\n");

    proc->pageLength = pageSize;

    proc->pages.next = InitializePageQueueWithInput(pageSize);

}

//搜索目标指定页面

BlockQueue \*SearchPage(BlockQueue \*blockQueue, Page page) {

    BlockQueue \*p = blockQueue;

    while (p != NULL) {

        if (p->block.page != NULL) {

            if (p->block.page->pageID == page.pageID)

                return p;

        }

        p = p->next;

    }

    return NULL;

}

//搜索空闲块

BlockQueue \*SearchIdleBlock(BlockQueue \*blockQueue) {

    BlockQueue \*p = blockQueue;

    while (p != NULL) {

        if (p->block.state == IDLE) return p;//IDLE空闲标志

        else p = p->next;

    }

    return NULL;

}

//返回块号

int GetBlockLable(BlockQueue \*blockQueue, BlockQueue \*goalBlock) {

    BlockQueue \*p = blockQueue;

    int count = 1;

    while (p != goalBlock) {

        p = p->next;

        count++;

    }

    return count;

}

//查找停留最久页面

BlockQueue \*GetOldestBlock(BlockQueue \*blockQueue) {

    BlockQueue \*p = blockQueue, \*oldestAddress;

    if (p == NULL) return p;

    oldestAddress = p;

    long long min = p->block.time;

    while (p != NULL) {

        if (p->block.time < min) {

            min = p->block.time;

            oldestAddress = p;

        }

        p = p->next;

    }

    return oldestAddress;

}

//检索页面序列，找出未来最长时间内不再被访问的页面（OPT算法）

BlockQueue \*GetLongestWithoutAccess(BlockQueue \*blockQueue, PageQueue \*currentPage){

    BlockQueue \*p = blockQueue, \*longestAddress;//p：内存的物理块

    PageQueue \*q = currentPage->next;//q:进程页面

    if (p == NULL) return p;

    longestAddress = p;

    int max= 0;

    while (p != NULL){

        int count = 0;

        while(q != NULL){

            count++;//记录该页面未来不会被访问的时长

            /\*找到未来的进程将访问该物理块存放的页面，记录终止，

            当前计数为未来将不会使用的最长时间 \*/

            if(p->block.page->pageID == q->page.pageID) break;

            q = q->next;

        }

        if(count > max){ //记录未来最长时间不会访问的页面

            max = count;

            longestAddress = p;

        }

        q = currentPage->next;

        p = p->next;

    }

    return longestAddress;

}

//SCR算法

BlockQueue \*GiveSecondChance(BlockQueue \*blockQueue){

    BlockQueue \*p=blockQueue, \*oldestAddress,\*head;

    head=p;

    oldestAddress = p;

    int len=GetBlockQueueSize(blockQueue);

    if (p == NULL) return p;

    int j=1,flag=0;

    long long min= 65535;

    do

    {

        if(flag==1)

        {

            oldestAddress->block.A=0;//得到第二次机会，清0

            p=head;

            j++;//开始找下一个FIFO页面

            min=65535;

        }

        if(j>len)// 所有页面都找过一轮

        {

            j=1;

            p=head;

            for (int i = 0; i < len; i++)

            {

                p->block.visit=0;

                p=p->next;

            }

            p=head;

        }

        while (p != NULL) {

        if (p->block.time < min&&p->block.visit!=1) {

            min = p->block.time;

            oldestAddress = p;

        }

        p = p->next;

    }

        oldestAddress->block.visit=1;

        flag=1;

    }while(oldestAddress->block.A==1);

    return oldestAddress;

}

//获取clock算法中的块队列元素信息

BlockQueue \*GetClockBlock(BlockQueue \*blockQueue,PageQueue \*currentPage){

    BlockQueue \*p = blockQueue,\*clockAddress;

    PageQueue \*q = currentPage;//q

    clockAddress = p;

    while(1){

        if (p == NULL) p = blockQueue;

        if (p->block.A == 1){

            p->block.A = 0;

            p = p->next;

        }

        else{

            clockAddress = p;

            return clockAddress;

        }

    }

}

//打印物理块块信息

void PrintBlockList(BlockQueue \*blockQueue, int pageID, int color) {

    int flag=0;

    BlockQueue \*currentBlock = blockQueue;

    int noteblock;//记录命中、替换及直接加入的块号

    for (int i = 0; i < GetBlockQueueSize(blockQueue); i++) {

        if (currentBlock == NULL) break;

        if (currentBlock->block.state == IDLE) {

            printf("|          | ");

        } else {

            if (currentBlock->block.page->pageID != pageID) {

                printf("|     %d    | ", currentBlock->block.page->pageID);

            } else {

                noteblock= GetBlockLable(blockQueue, currentBlock);

                switch (color) {

                    case COLOR\_Exist:

                        flag=1;

                        printf("|     %d    | ", pageID);

                        break;

                    case COLOR\_NotExist\_IDLE:

                        flag=2;

                        printf("|     %d    | ", pageID);

                        break;

                    case COLOR\_NotExist\_NoIDLE:

                        flag=3;

                        printf("|     %d    | ", pageID);

                        break;

                    default:

                        break;

                }

            }

        }

       // cout<<currentBlock->block.A<<" ";

        currentBlock = currentBlock->next;

    }

    if(flag==1)

    {

        True++;

        printf("\n  说明：%d 号进程页面命中，主存已存在该页面，物理块号为 %d \n",pageID,noteblock);

    }

     if(flag==2)

    {

        False++;

        printf("\n  说明：%d 号进程缺页，主存中不存在该页面，调入 %d 号空闲物理块\n",pageID,noteblock);

    }

    if(flag==3){

        False++;

        printf("\n  说明：%d 号进程缺页，主存中无空闲空间，置换物理块号为 %d 号的页面\n",pageID,noteblock);

    }

    printf("\n===========================================================\n");

}

//先进先出算法，选择在内存中驻留时间最久的页面予以淘汰，即调用GetOldestBlock函数

void FIFO(BlockQueue \*blockQueue, process \*proc) {

    PageQueue \*currentPage = proc->pages.next;//当前程序要使用的页面

    while (currentPage != NULL) {

        if (SearchPage(blockQueue, currentPage->page) ) {//存在该页面，命中！

            PrintBlockList(blockQueue, currentPage->page.pageID, COLOR\_Exist);

        } else {//不存在该页面

            BlockQueue \*idleBlock = SearchIdleBlock(blockQueue);//查找是否有空闲的物理块，并将地址返回给idleBlock

            if (idleBlock != NULL) {//有空闲物理块

                idleBlock->block.page = (Page \*) malloc(sizeof(Page));

                idleBlock->block.state = BUSY;

                idleBlock->block.time = t++;//最后访问时间++

                idleBlock->block.page->pageID = currentPage->page.pageID;//存入当前页面号

                PrintBlockList(blockQueue,

                               currentPage->page.pageID, COLOR\_NotExist\_IDLE);//输出物理块占用情况

            } else {

                idleBlock = GetOldestBlock(blockQueue);//没有空闲物理块，则调用替换在内存中驻留最久的页面的算法（FIFO）

                idleBlock->block.time = t++;//最后访问时间++

                idleBlock->block.page->pageID = currentPage->page.pageID;

                PrintBlockList(blockQueue,currentPage->page.pageID, COLOR\_NotExist\_NoIDLE);

            }

        }

        currentPage = currentPage->next;

    }

}

//SCR算法

void SCR(BlockQueue \*blockQueue, process \*proc) {

    PageQueue \*currentPage = proc->pages.next;

    while (currentPage != NULL) {

        BlockQueue \*searchedBlock = SearchPage(blockQueue, currentPage->page);//查找当前页面是否存在

        if (searchedBlock != NULL) {

            searchedBlock->block.A = 1;//命中该页面，访问位置1（与FIFO区别之处)

            PrintBlockList(blockQueue, currentPage->page.pageID, COLOR\_Exist);

        } else {

            BlockQueue \*idleBlock = SearchIdleBlock(blockQueue);

            if (idleBlock != NULL) {

                idleBlock->block.state = BUSY;

                idleBlock->block.time = t++;

                idleBlock->block.A = 1;

                idleBlock->block.page = (Page \*) malloc(sizeof(Page));

                idleBlock->block.page->pageID = currentPage->page.pageID;

                PrintBlockList(blockQueue,

                               currentPage->page.pageID, COLOR\_NotExist\_IDLE);

            } else {

                idleBlock = GiveSecondChance(blockQueue);

                idleBlock->block.time = t++;

                idleBlock->block.A = 1;

                idleBlock->block.page->pageID = currentPage->page.pageID;

                PrintBlockList(blockQueue,currentPage->page.pageID, COLOR\_NotExist\_NoIDLE);

            }

        }

        currentPage = currentPage->next;

    }

}

//LRU最近最久未使用

void LRU(BlockQueue \*blockQueue, process \*proc) {

    PageQueue \*currentPage = proc->pages.next;

    while (currentPage != NULL) {

        BlockQueue \*searchedBlock = SearchPage(blockQueue, currentPage->page);//查找当前页面是否存在

        if (searchedBlock != NULL) {

            searchedBlock->block.time = t++;//未发生替换，但使用该页面，最后使用时间++（与FIFO区别之处)

            PrintBlockList(blockQueue, currentPage->page.pageID, COLOR\_Exist);

        } else {

            BlockQueue \*idleBlock = SearchIdleBlock(blockQueue);

            if (idleBlock != NULL) {

                idleBlock->block.state = BUSY;

                idleBlock->block.time = t++;

                idleBlock->block.page = (Page \*) malloc(sizeof(Page));

                idleBlock->block.page->pageID = currentPage->page.pageID;

                PrintBlockList(blockQueue,

                               currentPage->page.pageID, COLOR\_NotExist\_IDLE);

            } else {

                idleBlock = GetOldestBlock(blockQueue);

                idleBlock->block.time = t++;

                idleBlock->block.page->pageID = currentPage->page.pageID;

                PrintBlockList(blockQueue,currentPage->page.pageID, COLOR\_NotExist\_NoIDLE);

            }

        }

        currentPage = currentPage->next;

    }

}

//OPT最佳页面替换算法

void OPT(BlockQueue \*blockQueue, process \*proc){

    PageQueue \*currentPage = proc->pages.next;

    while (currentPage != NULL) {

        if (SearchPage(blockQueue, currentPage->page) != NULL) {//命中

            PrintBlockList(blockQueue, currentPage->page.pageID, COLOR\_Exist);

        } else {

            BlockQueue \*idleBlock = SearchIdleBlock(blockQueue);

            if (idleBlock != NULL) {//未命中，但有空闲块

                idleBlock->block.state = BUSY;

                idleBlock->block.time = t++;

                idleBlock->block.page = (Page \*) malloc(sizeof(Page));

                idleBlock->block.page->pageID = currentPage->page.pageID;

                PrintBlockList(blockQueue,

                               currentPage->page.pageID, COLOR\_NotExist\_IDLE);

            } else {//需要使用替换算法

                idleBlock = GetLongestWithoutAccess(blockQueue,currentPage);//找到未来最长时间内不再被访问的页面进行替换

                idleBlock->block.time = t++;

                idleBlock->block.page->pageID = currentPage->page.pageID;

                PrintBlockList(blockQueue,

                               currentPage->page.pageID, COLOR\_NotExist\_NoIDLE);

            }

        }

        currentPage = currentPage->next;

    }

}

//时钟置换算法

void Clock(BlockQueue \*blockQueue, process \*proc){

    PageQueue \*currentPage = proc->pages.next;

    BlockQueue \*currentBlock = blockQueue;

    while (currentPage != NULL) {

        BlockQueue \*temp = SearchPage(blockQueue, currentPage->page);

        if (temp != NULL) {//命中

            temp->block.A = 1;

            if(temp->next == NULL) currentBlock = blockQueue;

            else currentBlock = temp->next;

            PrintBlockList(blockQueue, currentPage->page.pageID, COLOR\_Exist);

        } else {

            BlockQueue \*idleBlock = SearchIdleBlock(blockQueue);

            if (idleBlock != NULL) {//未命中，但有空闲块

                idleBlock->block.state = BUSY;

                idleBlock->block.time = t++;

                idleBlock->block.page = (Page \*) malloc(sizeof(Page));

                idleBlock->block.page->pageID = currentPage->page.pageID;

                idleBlock->block.A = 1;

                PrintBlockList(blockQueue,

                               currentPage->page.pageID, COLOR\_NotExist\_IDLE);

            } else {//需要使用替换算法

                idleBlock = GetClockBlock(currentBlock,currentPage);//找到最近未使用的页面进行替换

                idleBlock->block.time = t++;

                idleBlock->block.page->pageID = currentPage->page.pageID;

                idleBlock->block.A = 1;

                if(idleBlock->next == NULL) currentBlock = blockQueue;

                else currentBlock = idleBlock->next;

                PrintBlockList(blockQueue,

                               currentPage->page.pageID, COLOR\_NotExist\_NoIDLE);

            }

        }

        currentPage = currentPage->next;

    }

}

//基本展示菜单

void Menu(){

    printf("=================欢迎使用请求分页存储管理系统=================\n");

    printf("==================组员：张杰宁 李坤璘 方凡  小组：15组========\n");

    printf("====================请选择页面置换算法========================\n");

    printf("       \t1.OPT\t2.FIFO\t3.LRU\t4.SCR\t5.Clock\t0.exit\n");

    printf("==============================================================\n");

    printf("请输入选项\n");

}

2. 模拟多个进程并发执行时请求分页存储管理的请求调页和页面置换功能。

五．调试方案及调试过程问题分析

#include <iostream>

#include<cstring>

#include<unistd.h>

#include<pthread.h>

using namespace std;

const int frame\_size = 100;

//记录的页表项

struct Page

{

    //页表项

    int page\_id = -1;  //逻辑页面号

    bool state = 0;    //状态位

    bool visit = 0;    //访问位

    bool change = 0;   //修改位

    // struct Page\* next=NULL;

};

//存储的内存帧

struct Page\_frame

{

    int tid = -1; //进程号

    int frame\_id; //占用进程的页号

    Page\* ppage = NULL;

    struct Page\_frame\* map = NULL;

    struct Page\_frame\* next = NULL;

};

//线程（进程）的载体Choose\_occ

struct PCB

{

    int tid = -1;

    int page\_num;

    int bmap\_num;

    struct Page\_frame\* next = NULL;

    int index = 0; //记录当前访问到的位置

    int process[frame\_size];

};

Page\_frame\* frame\_list = new Page\_frame; //内存的物理帧位图

Page \*page; //内存的位示图

int number[10];//待访问页面号序列

int nc\_size = 0;//内存大小

int page\_size = 0;//页面大小

int number\_size = 0; //线程/进程大小

int buffer\_size;//定义的缓冲区的大小

pthread\_mutex\_t list\_mutex;

pthread\_mutex\_t page\_mutex;

void Choose\_occ(PCB\* pcb,int code);//选择置换算法对PCB进行操作

void fifo(PCB\* pcb, Page\_frame\* list);//FIFO算法的实现

void\* CreatPCB(void\* ary); //多线程调用创建新PCB

void InsertList(bool Ishead, Page\_frame\* frame, Page\_frame\* list); //帧页的list插入

void MoToHead(Page\_frame\* frame, Page\_frame\* list); //链表元素的前调

int GetBmapNum(int size); //根据进程大小分配占用多少内存的函数

int GetCurrentNum(PCB\* pcb); //获取当前进程的内存大小

bool IsinList(PCB\* pcb, int frame\_id); //判断某一页是否在内存里

int main()

{

    /\*阶段一：初始化内存大小、页面大小，计算内存帧大小\*/

    cout<<"请输入内存大小：";

    cin>>nc\_size;//40

    cout<<"请输入页面大小：";

    cin>>page\_size;//8

    /\*计算内存帧的大小\*/

    nc\_size /= page\_size;

    buffer\_size = nc\_size;

    pthread\_t pid[3];

    page = new Page[buffer\_size]; //初始化大小

    cout << "内存帧数量：" << buffer\_size << endl;

    cout << "初始化成功" << endl;

    /\*阶段二：创建线程，这里选择创建三个线程，设置对应的id、访问元素个数、具体顺序\*/

    pthread\_mutex\_init(&list\_mutex,NULL);

    pthread\_mutex\_init(&page\_mutex,NULL);

    int pc1[] = { 1,3,4,0,1,2,1 };

    int pc2[] = { 2,3,4,0,1,2,2 };

    // int pc3[] = { 3,5,5,0,1,2,3,3 };

    int pc3[] = { 3,4,5,0,3,1,4,0 };

    pthread\_create(&pid[0], NULL, CreatPCB, pc1);

    pthread\_create(&pid[1], NULL, CreatPCB, pc2);

    pthread\_create(&pid[2], NULL, CreatPCB, pc3);

    /\*阶段三：所有线程执行完毕后释放\*/

    for(int i = 0; i <3;i++)

    {

        pthread\_join(pid[i], NULL);

    }

    return 0;

}

//选择置换算法对PCB进行操作

void Choose\_occ(PCB\* pcb, int code)

{

    if(IsinList(pcb, pcb->process[pcb->index]))

    {

        printf("进程：%d，其%d页已在内存\n", pcb->tid, pcb->process[pcb->index]);

        pcb->index++;

        return;

    }

    //判断自己占用的内存块是否已经大于分配的bmap

    int cnum = GetCurrentNum(pcb);

    if (cnum >= pcb->bmap\_num || cnum >= buffer\_size) //和自己置换

    {

        Page\_frame\* pre\_pcb = NULL;

        Page\_frame\* tmp\_pcb = pcb->next;

        while (tmp\_pcb->next != NULL)

        {

            pre\_pcb = tmp\_pcb;

            tmp\_pcb = tmp\_pcb->next;

        }

        /\*把pcb表里的新访问项添加到表头，移到头部保证兼容\*/

        MoToHead(tmp\_pcb, pcb->next);

        pthread\_mutex\_lock(&list\_mutex);

        /\*list表中的该项记录添加到表头，此时该项记录也需要移动到表头\*/

        MoToHead(tmp\_pcb->map, frame\_list);

        pthread\_mutex\_unlock(&list\_mutex);

        printf("进程：%d，页号：%d号，所占用的物理帧号:%d,替换的页号：%d\n", pcb->tid, pcb->process[pcb->index], (tmp\_pcb->ppage - page), tmp\_pcb->frame\_id);

        tmp\_pcb->frame\_id = pcb->process[pcb->index];

        tmp\_pcb->map->frame\_id = pcb->process[pcb->index];

    }

    else //从内存中

    {

        //pthread\_mutex\_lock(&page\_mutex);

        //找到第一个空闲的帧

        int flag = 0; //用作是否找到空闲帧的标记

        for(int i=0;i<buffer\_size;i++)

        {

            if(page[i].state == 0)

            {

                page[i].page\_id = pcb->tid; //该帧被pcb占用

                page[i].state = 1;

                Page\_frame\*nf\_pcb = new Page\_frame;

                Page\_frame\* nf\_main = new Page\_frame;

               nf\_pcb->tid = pcb->tid;

                nf\_main->tid = pcb->tid; //设置进程id

               nf\_pcb->ppage = &page[i];

                nf\_main->ppage = &page[i]; //设置对应物理帧

               nf\_pcb->frame\_id = pcb->process[pcb->index];

                nf\_main->frame\_id = pcb->process[pcb->index]; //设置对应进程的页号

               nf\_pcb->map = nf\_main;

                nf\_main->map =nf\_pcb; //相互映射

                printf("%d号进程的%d号页占用了物理帧:%d\n", pcb->tid, pcb->process[pcb->index], i);

                InsertList(true,nf\_pcb, pcb->next); //对pcb的访问表修改

                pthread\_mutex\_lock(&list\_mutex);

                InsertList(true, nf\_main, frame\_list); //对总表进行修改

                pthread\_mutex\_unlock(&list\_mutex);

                flag = 1;

                break;

            }

        }

        //pthread\_mutex\_unlock(&page\_mutex);

        if (flag == 0) //内存满了,进行置换,添加顺序是头插法

        {

            if (code == 0) //选择fifo

            {

                fifo(pcb, frame\_list);

            }

        }

    }

    pcb->index++;

}

//判断某一页是否在内存里

bool IsinList(PCB\* pcb, int frame\_id)

{

    pthread\_mutex\_lock(&list\_mutex);

    Page\_frame\* tmp = pcb->next;

    while (tmp->next != NULL)

    {

        if (tmp->frame\_id == frame\_id && tmp->tid == pcb->tid)

        {

            pthread\_mutex\_unlock(&list\_mutex);

            return true;

        }

        tmp = tmp->next;

    }

    pthread\_mutex\_unlock(&list\_mutex);

    return false;

}

//多线程调用创建新PCB

void\* CreatPCB(void\* ary)

{

    /\*

        ary[0]为线程的id

        ary[1]为访问顺序元素的个数

        ary[2]及其以后为访问的具体顺序

    \*/

    PCB\* p = new PCB;

    int\* array = (int\*)ary;

    p->tid = array[0];

    p->page\_num = array[1];

    p->bmap\_num = GetBmapNum(p->page\_num);

    p->next = new Page\_frame;

    int count = array[2];

    for (int i = 0; i < count; i++)

    {

        p->process[i] = array[3 + i];

    }

    while(p->index!=count)

    {

        sleep(3);

        Choose\_occ(p, 0);

    }

    return NULL;

}

//根据进程大小分配占用多少内存的函数

int GetBmapNum(int size)

{

    if(size==1)

         return 1;

    else if(size>=2&&size<=6) return 2;

    else

    {

        return size/3;

    }

}

//获取当前进程的内存大小

int GetCurrentNum(PCB\* pcb)

{

    int count = 0;

    pthread\_mutex\_lock(&list\_mutex);

    Page\_frame\* tmp = pcb->next->next;

    Page\_frame\* pre = pcb->next;

    while (tmp != NULL)

    {

        /\*当进程占用模块时，将不会被舍弃\*/

        if(tmp->tid==pcb->tid) //进程占用

        {

            pre = pre->next;

            tmp = tmp->next;

            count++;

        }

        /\*如果遇到了争夺的行为，此时PCB不是自己的，需要舍弃\*/

        else //被争夺了

        {

            pre->next = tmp->next;

            Page\_frame\* dtmp = tmp; //获取错乱记录表项

            tmp = tmp->next;

            delete dtmp;

        }

    }

    pthread\_mutex\_unlock(&list\_mutex);

    return count;

}

//帧页的list插入

void InsertList(bool Ishead, Page\_frame\* frame, Page\_frame\* list)

{

    if(Ishead)

    {

        frame->next = list->next;

        list->next = frame;

    }

    else

    {

        Page\_frame\* tmp = list;

        while (tmp->next != NULL)

        {

            tmp = tmp->next;

        }

        tmp->next = frame;

    }

}

 //链表元素的前调

void MoToHead(Page\_frame\* frame, Page\_frame\* list)

{

    Page\_frame \*pre = NULL;

    Page\_frame \*tmp = list->next;

    while(tmp->next!=NULL)

    {

        pre = tmp;

        tmp = tmp->next;

        if(tmp==frame)

        {

            pre->next = NULL;

            InsertList(true, tmp, list);

            break;

        }

    }

}

//FIFO算法的实现

void fifo(PCB\* pcb, Page\_frame\* list)

{

    pthread\_mutex\_lock(&list\_mutex);

    pthread\_mutex\_lock(&page\_mutex);

    Page\_frame\* tmp = frame\_list->next; //在总表上找最后一个表项

    while (tmp->next != NULL)

    {

        tmp = tmp->next;

    }

    tmp->frame\_id = pcb->process[pcb->index]; //修改该list表项对应的进程页号

    Page\* opage = tmp->ppage; //获取其占用的物理帧

    printf("进程：%d  抢夺：%d号进程  ，帧号：%d ， 放入页号：%d\n", pcb->tid, opage->page\_id, (opage - page), pcb->process[pcb->index]); //第几号物理帧是计算的偏移

    opage->page\_id = pcb->tid; //修改物理帧的被占用属性

    Page\_frame\* newpcb\_frame = new Page\_frame;

    newpcb\_frame->frame\_id = pcb->process[pcb->index];

    newpcb\_frame->map = tmp;

    newpcb\_frame->ppage = opage;

    newpcb\_frame->tid = pcb->tid;

    MoToHead(tmp, frame\_list);

    pthread\_mutex\_unlock(&list\_mutex);

    pthread\_mutex\_unlock(&page\_mutex);

    InsertList(true, newpcb\_frame, pcb->next); //把抢夺到的帧添加到自己的链表里

}

**任务一：**

任务一为手动输入操作的存储管理系统，因此对其界面功能做简单介绍

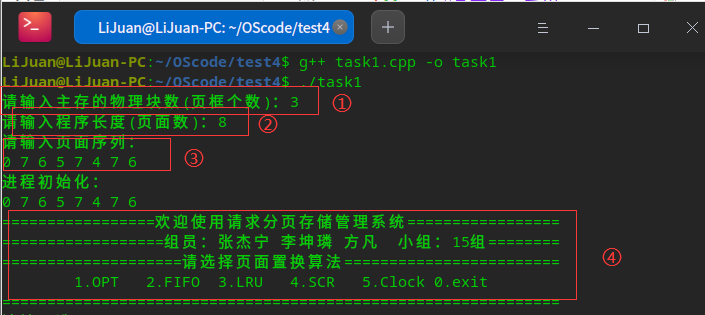


图5.1 管理系统主界面

①为键盘自行输入的主存物理块数，即页框个数

②为我们输入的程序长度，即页面数

③为输入的页面序列，决定了我们进程进入磁盘的页面替换顺序

④为进行上述三步操作初始化成功后的请求分页存储管理系统的主界面。

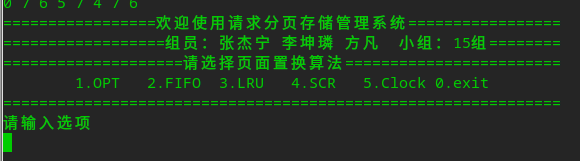


图5.2 选择我们需要的页面置换算法

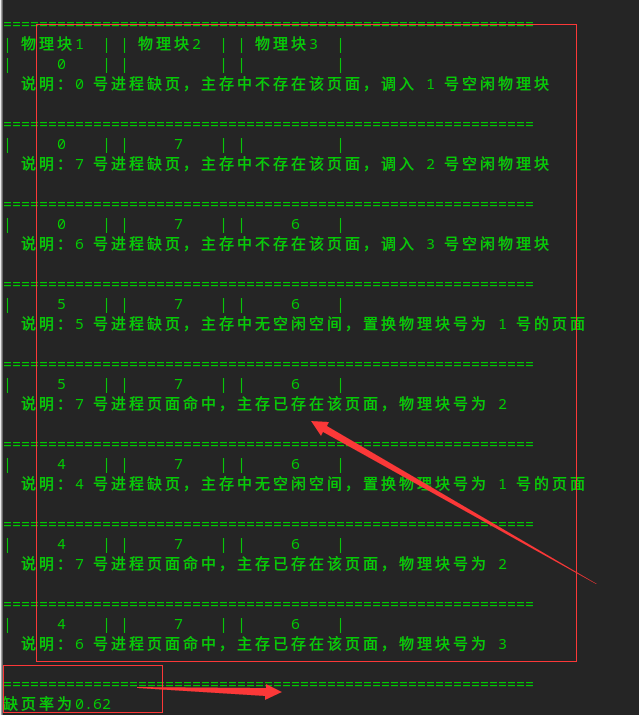


图5.3 结果示例

可以看出上图主要由两部分组成，上半部分为我们采用某种策略后的具体进程活动过程。在进程活动完毕后，最后我们会得到一个数值为缺页率。我们可以通过不同策略的缺页率去比较各个方式的性能。

**任务二：**

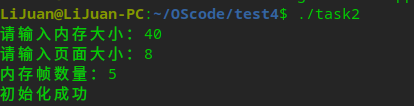


图5.4 任务二程序初始化

当我们将内存大小以及页面大小确定下来后，接下来就交给程序里自定义的进程顺序自动并发执行，并实时显示相关结果以及信息，具体结果见下方的运行结果。

六．运行结果与分析

1.测试用例及各功能运行结果截图

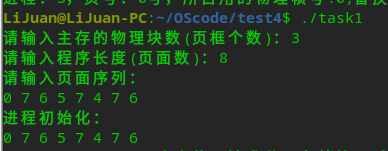


图6.1 本次实验手动输入的初始化进程id

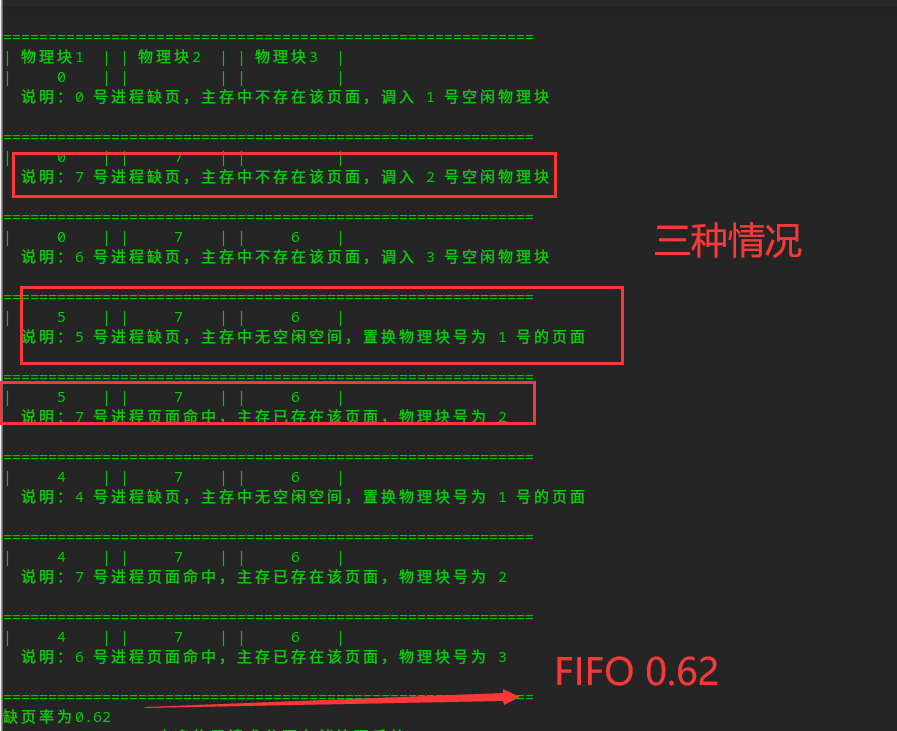


图6.2 FIFO策略的结果

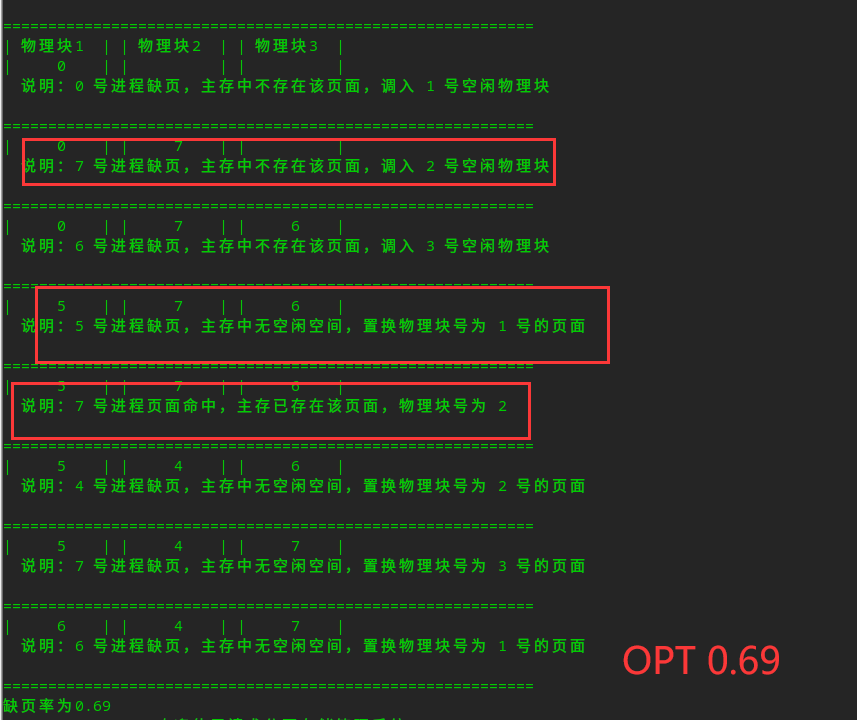


图6.3 OPT策略的结果

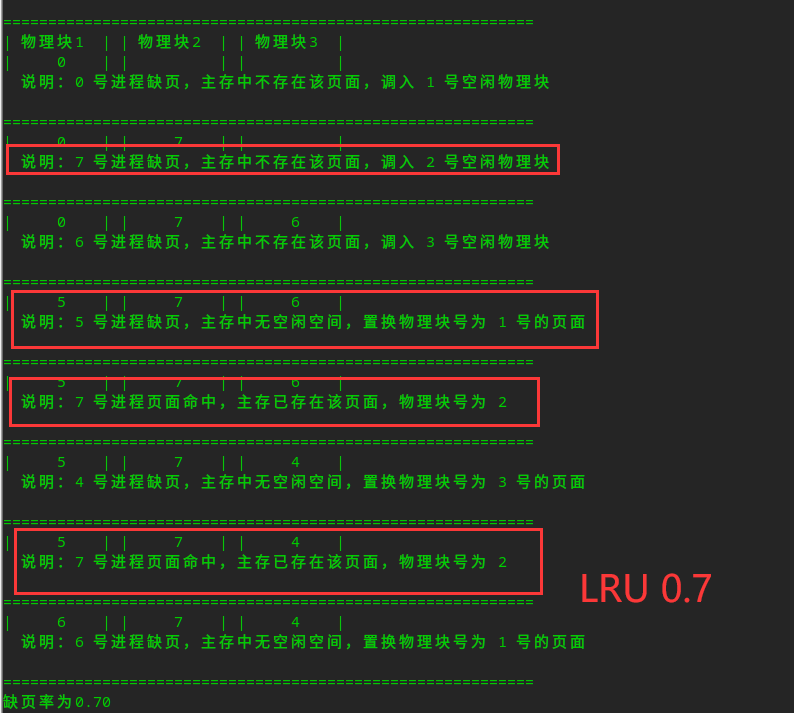


图6.4 LRU策略的结果

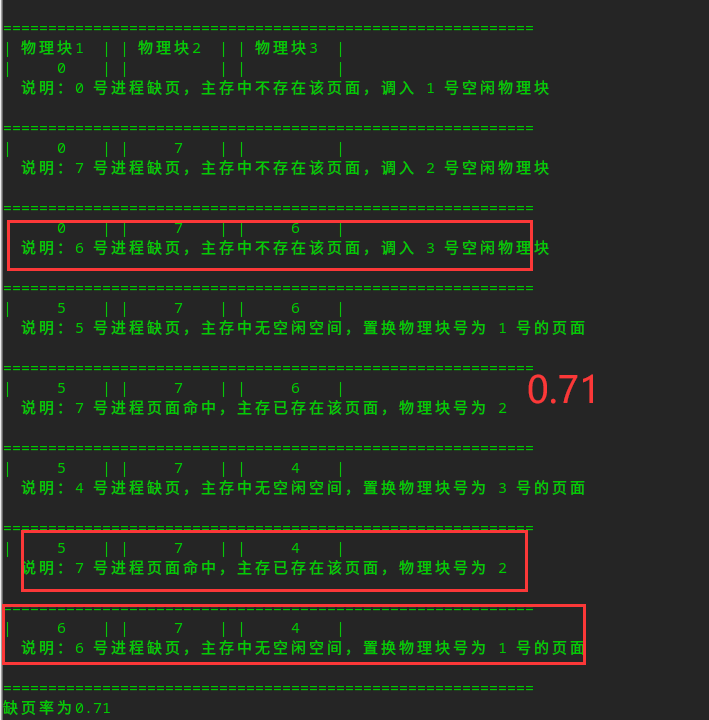


图6.5 SCR策略的结果

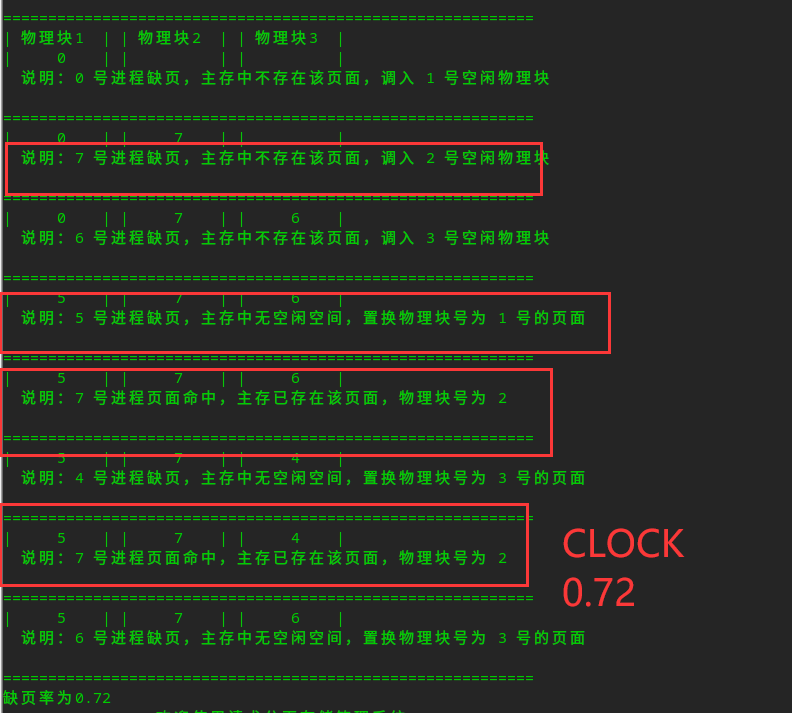


图6.6 CLOCK策略的结果

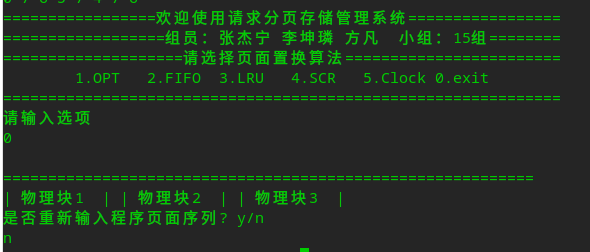


图6.7 退出界面

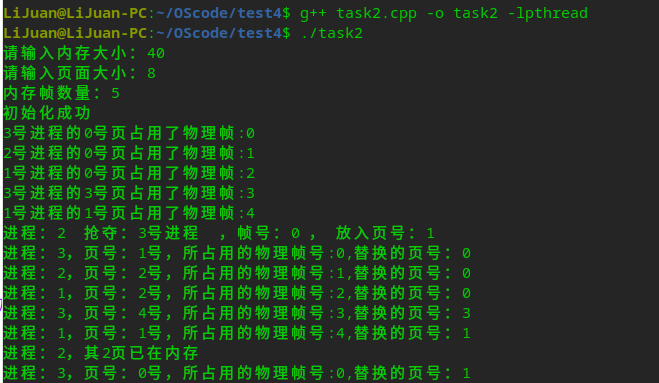


图6.8 任务二多进程实现请求分页存储管理

2.运行结果分析

由于上述实验仅对于一组进程进行缺页率的计算，具有偶然性且不可完全信任，因此我们对课设中的指令序列长度设置为320，每页存放指令个数设置为10.我们生成10次指令序列，并统计和比较各算法的命中率，以此实现性能比较。

数据量过大不适合截取终端的结果，因为包含的信息过多，所以我们仅将每种算法的最终缺页率结果汇总成以下表格：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 次数/策略 | FIFO | OPI | LRU | CLOCK |
| 1 | 0.362500 | 0.509375 | 0.365625 | 0.365625 |
| 2 | 0.321875 | 0.496875 | 0.325000 | 0.321875 |
| 3 | 0.306250 | 0.478125 | 0.306250 | 0.306250 |
| 4 | 0.334375 | 0.509375 | 0.331250 | 0.337500 |
| 5 | 0.365625 | 0.500000 | 0.353125 | 0.362500 |
| 6 | 0.331250 | 0.493750 | 0.328125 | 0.331250 |
| 7 | 0.337500 | 0.496875 | 0.334375 | 0.334375 |
| 8 | 0.337500 | 0.500000 | 0.340625 | 0.337500 |
| 9 | 0.359375 | 0.512500 | 0.359375 | 0.359375 |
| 10 | 0.353125 | 0.515625 | 0.353125 | 0.356250 |
| 平均值 | **0.340938** | **0.50125** | **0.339688** | **0.34125** |

表6.1 各算法指令序列长度为320和页指令个数为10的命中率

我们可以发现OPT策略相对于其他三种策略的命中率更高，其他三种策略的命中率相差不是很大。不过我们在实验中还有一个发现，当指令序列长度和页指令个数增大时，OPT策略相对于其他策略命中率的差距显得更加大了。

七．总结

1.设计中的经验体会

在我们小组进行课程设计时，我们也对我们所学的页替换策略进行了复习。FIFO先进先出页面替换策略替换最早进入的页；OPT最佳页面替换策略淘汰下次访问距当前最远的那些页中序号最小的页；LRU最近最少用页面替换策略通过计数器淘汰上次使用距当前最远的页；CLOCK时钟页面替换策略基于LRU的思想，硬件在页面被访问时设置页表项中的访问位，随着表针的移动，淘汰访问位是0的页面，或清除页面的访问位。

在清楚了页替换策略的原理之后我们构思如何编写算法程序来实现他们，FIFO策略和OPT策略实现主要是通过设计算法后存储需要替换页的下标来实现的，而LRU算法的实现是通过定义一个书上所说的计数器来实现，CLOCK算法是通过定义访问位来实现的。在了解清楚原理之后，设计算法就如鱼得水了。

在最后的统计分析时，OPT策略是理想中的最优策略，故其命中率始终是三个里面最高的。和之前所说的一样，统计这里也有可深入之处，我们在课设报告的性能分析处只比较了不同指令序列长度和页指令个数下命中率的变化

2.后续设计设想

（1）是将驻留集长度设为可变的驻留集长度，为此我将驻留集长度设为了常量，可以随时调节，设置成变量增加程序的灵活性

（2）输出页面排版还可以继续优化

（3）多进程动态进行请求分页存储管理的任务中，可以将我们想要的进程id以及进入的顺序写入文件。这样当我们想要修改顺序的时候就不需要在代码里更改而是直接通过文件在便于修改。

课程设计选题五 模拟实现一个Linux文件系统

一．设计要求

1.数据要求

输入：相关的文件操作指令(如mk,delete,cp,mv，mkdir,rmdir,ls,cd)

输出：linux系统中相关实际操作

2.功能要求

（1）在磁盘空间模拟实现简单的Linux文件系统，并提供基本的文件操作命令

（2）在内存中开辟一个虚拟磁盘空间作为文件存储分区，

（3）在退出该文件系统的使用时，应将虚拟磁盘上的内容以一个文件的方式保存到磁盘上，一遍下次可以将它恢复到内存的虚拟磁盘中。

（4）空闲磁盘空间的管理可选择FAT表、位示图等数据结构

（5）文件目录结构采用多级目录结构，包含文件名、物理地址、长度等信息。

（6）通过目录项实现对文件的读和写的保护。

二．背景知识及设计思路

1.背景知识

**（1）磁盘存储空间的划分与初始化**

磁盘存储空间的划分：将物理磁盘划分为一个个文件卷（逻辑卷、逻辑盘）。

在存储空间初始化时，需要将各个文件卷划分为目录区、文件区。

①目录区：主要存放文件目录信息（FCB）、用于磁盘存储空间管理的信息。

②文件区主要存放文件数据。

**（2）空闲链表法**

①空闲盘块链：以盘块为单位组成一条空闲链。空闲盘块中存储着下一个空闲盘块的指针。

②空闲盘区链：以盘区为单位组成一条空闲链。每个空闲盘区中的第一个盘块记录了盘区的长度、下一个盘区的指针。

**（3）分配算法：**

若某文件申请K个盘块，由于空闲盘区链将连续的盘块组成一个盘区，所以若某个盘区大小满足可以实现一次分配，同样可以采用首次适用、最佳适用等算法，从链头开始检索，按照一定的规则找到一个大小符合要求的空闲盘区分配给文件。若没有合适的连续空闲块，也可以将不同的盘区的盘同时分配给一个文件，同样分配后也需要修改相应的指针链和盘区大小等数据。

**（4）回收算法**

若回收区和某个空闲盘区相邻，则需要将回收区合并到空闲盘区中。若回收区没有和任何空闲区相邻，将回收区作为一个单独的一个空闲盘区挂到链尾。同样也需要修改链表指针和盘区大小等信息。

**（5）成组链接法**

①文件区中的所有空闲盘块分为若干个组，假设所有空闲盘块号是201~9999（实际中空闲盘块是不要求连续的,这里是为了表述方便，认为空闲盘块是连续的），将这些盘块进行分组，每组100个。

②每一组含有的空闲盘块总数N和该组的所有空闲盘号块记入前一组的第一个盘块中。这样各组的第一个盘块可以链成一条链。

③如果是最后一组，那么将前一个分组的第一个盘块的中存放空闲盘块链的结束。

标志符，如“-1”表示下一组是最后一个分组了。

2.设计思路

（1）在该虚拟文件系统启动时，申请一块内存作为磁盘空间。

（2）将这块内存空间进行格式化，本系统仿照的是FAT16文件系统，其结构如下

1块 2块 2块 995块

引导块 FAT1 FAT2 数据区

格式化时，主要包括引导块，FAT1，FAT2，的一些初始化工作。例如设置文件系统的信息，FAT1，FAT2的信息等等。

（3）根据用户输入的命令，调用对应的my\_函数

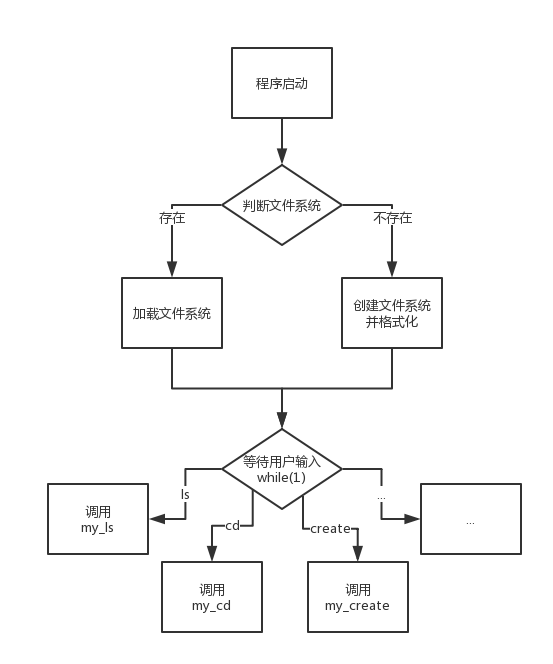
.

图2.1 基本思路流程图

三．设计方案

1.数据结构设计

**文件控制块：**

typedef struct FCB {

char filename[8]; *//文件名*

char exname[3]; *//文件扩展名*

unsigned char attribute; *//文件属性字段(目录or文件)*

unsigned short time; *//创建时间*

unsigned short date; *//创建日期*

unsigned short first; *//起始盘块号*

unsigned short length; *//文件长度*

char free; *//表示目录项是否为空*

} fcb;

**文件分配表：**

typedef struct FAT {

unsigned short id;

} fat;

**用户打开文件表：**

当打开一个文件时，必须将文件的目录项中的所有内容全部复制到内存中，同时还要记录有关文件操作的动态信息，如读写指针的值等。在本实例中实现的是一个用于单用户单任务系统的文件系统，为简单起见，我们把用户文件描述符表和内存FCB表合在一起，称为用户打开文件表，表项数目为10，即一个用户最多可同时打开10个文件。然后用一个数组来描述，则数组下标即某个打开文件的描述符。另外，我们在用户打开文件表中还设置了一个字段“char dir[80]”，用来记录每个打开文件所在的目录名，以方便用户打开不同目录下具有相同文件名的不同文件。

typedef struct USEROPEN {

char filename[8];

char exname[3];

unsigned char attribute;

unsigned short time;

unsigned short date;

unsigned short first;

unsigned short length;

char free;

int dirno; *// 父目录文件起始盘块号*

int diroff; *// 该文件对应的 fcb 在父目录中的逻辑序号*

char dir[MAXOPENFILE][80]; *// 全路径信息*

int count;

char fcbstate; *// 是否修改 1是 0否*

char topenfile; *// 0: 空 openfile*

} useropen;

**引导块BLOCK0：**

在引导块中主要存放逻辑磁盘的相关描述信息，比如磁盘块大小、磁盘块数量、文件分配表、根目录区、数据区在磁盘上的起始位置等。如果是引导盘，还要存放操作系统的引导信息。本实例是在内存的虚拟磁盘中创建一个文件系统，因此所包含的内容比较少，只有磁盘块大小、磁盘块数量、数据区开始位置、根目录文件开始位置等。

typedef struct BLOCK {

char magic\_number[8];

char information[200];

unsigned short root;

unsigned char\* startblock;

} block0;

2.程序结构设计（模块图，流程图，函数调用关系图）

**（1）全局变量定义**

指向虚拟磁盘的起始地址：

unsigned char\* myvhard;

用户打开文件表数组：

useropen openfilelist[MAXOPENFILE];

记录虚拟磁盘上的数据区开始位置：

unsigned char\* startp;

**（2）虚拟磁盘空间布局**

由于真正的磁盘操作需要涉及到设备的驱动程序，所以本实例是在内存中申请一块空间作为虚拟磁盘使用，我们的文件系统就建立在这个虚拟磁盘上。虚拟磁盘一共划分成1000个磁盘块，每个块1024个字节，其布局格式是模仿FAT文件系统设计的，其中引导块占一个盘块，两张FAT各占2个盘块，剩下的空间全部是数据区，在对虚拟磁盘进行格式化的时候，将把数据区第1块（即虚拟磁盘的第6块）分配给根目录文件，如下图所示：



图3.1 虚拟磁盘空间布局

**（3）创建目录流程**

①调用do\_read读入当前目录文件到内存，检查新建文件目录是否重名

②分配一个空闲的打开文件表项

③分配一个空闲的盘块

④在当前目录中问新建目录寻找一个空闲的目录项

⑤设置FCB，文件的属性信息

⑥创建特殊的两个目录项‘.’,‘..’

⑦返回

**（4）删除目录流程**

①Read读入当前目录文件内容到内存，检查要删除的文件目录是否存在

②检查该目录是否为空

③检查是否已经打开，打开用close则关闭

④回收给目录文件的磁盘块

⑤修改该目录文件的目录项

⑥修改用户打开表项的长度信息

⑦返回

**（5）新建文件流程**

①分配一个空闲的打开文件表项

②检查新文件的父目录是否打开

③Read该父目录的文件到内存，并检测新建的文件名是否重名

④检查是否有空闲盘块

⑤寻找空闲的目录项

⑥准备好新文件的FCB

⑦调用close关闭打开的父目录文件

⑧返回

**（6）删除文件流程**

①检查要删除的文件的父目录是否已打开

②Read父目录到内存

③检查文件是否打开

④回收磁盘快

⑤清空该文件的目录项

⑥修改用户打开文件表项中的长度信息

⑦返回

**（7）打开文件流程**

①检查该文件名是否存在

②Read该父目录到内存

③检查用户打开的文件表中是否有空闲表项

④为该文件填写空白用户打开文件表项内容

⑤返回

**（8）写文件流程**

①检查fd的有效性

②提示用户输入写方式

③提示用户输入内容

④调用do\_write（）将键入的内容写入到文件中

⑤返回写入的字节数

3.用到的Linux系统调用函数说明

**（1）opendir函数：**

打开一个目录并建立一个目录流

DIR \*opendir(const char \*name); //参数name 文件夹的名字

如果打开成功的话返回一个DIR结构的指针,该指针用于读取目录数据项。

如果失败的话返回一个空指针

如果文件中的文件过多也可能打开失败

**（2）readdir函数**

struct dirent \*readdir(DIR \*dirp)

函数成功的话返回一个指针，该指针的结构里保存着目录流中下一个目录项的资料，如果错误或者到达目录尾，返回NULL，但不改变error的值，只有发生错误的时候才会设置error;

struct dirent 结构体中有两个元素

ino\_t d\_ino: The inode of the file//文件的inode节点

char d\_name[]: The name of the file//文件名

**（3）chidr函数**

int chdir(const char \*path );

用于改变当前工作目录。调用参数是指向目录的指针，调用进程需要有搜索整个目录的权限。每个进程都具有一个当前工作目录。在解析相对目录引用时，该目录是搜索路径的开始之处。如果调用进程更改了目录，则它只对该进程有效，而不能影响调用它的那个进程。在退出程序时，shell还会返回开始时的那个工作目录。

**（4）lstat函数**

int lstat(const char \*pathname, struct stat \*statbuf);

获取文件属性，比如文件权限，文件属主，文件大小等属性。

第一个参数为传入参数，pathname为绝对路径或相对路径。第二个参数为传出参数，它将文件所具有的属性信息存放在该结构体指针所指的结构体中。接收该属性信息时，需要提前定义一个结构体变量struct stat st，或者结构体指针struct stat\* pst = &st。

四．设计源代码

“千行代码两行泪”

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<time.h>

#define BLOCKSIZE 1024

#define SIZE 1024000

#define END 65535

#define FREE 0

#define ROOTBLOCKNUM 2

#define MAXOPENFILE 10 // 同时打开最大文件数

#define MAX\_TEXT\_SIZE 10000

//文件控制块

typedef struct FCB {

    char filename[8];

    char exname[3];

    unsigned char attribute; // 0: dir file, 1: data file

    unsigned short time;

    unsigned short date;

    unsigned short first;

    unsigned short length;

    char free; // 0: 空 fcb

} fcb;

//文件分配表

typedef struct FAT {

    unsigned short id;

} fat;

//用户的文件表

typedef struct USEROPEN {

    char filename[8];

    char exname[3];

    unsigned char attribute;

    unsigned short time;

    unsigned short date;

    unsigned short first;

    unsigned short length;

    char free;

    int dirno; // 父目录文件起始盘块号

    int diroff; // 该文件对应的 fcb 在父目录中的逻辑序号

    char dir[MAXOPENFILE][80]; // 全路径信息

    int count;

    char fcbstate; // 是否修改 1是 0否

    char topenfile; // 0: 空 openfile

} useropen;

//磁盘引导块

typedef struct BLOCK {

    char magic\_number[8];

    char information[200];

    unsigned short user;

    unsigned char\* startblock;

} block0;

unsigned char\* myvhard;//指向虚拟磁盘的起始地址

useropen openfilelist[MAXOPENFILE];//用户打开文件表数组

int currfd;

unsigned char\* startp;

unsigned char buffer[SIZE];

char\* FILENAME = "zfilesys";

/\*相关的文件操作函数\*/

void startsys();//启动文件系统

void my\_format();//格式化磁盘

void my\_cd(char\* dirname);//进入目录

int do\_read(int fd, int len, char\* text);//读文件具体操作

int do\_write(int fd, char\* text, int len, char wstyle);//写文件具体操作

int my\_read(int fd);//读文件命令

int my\_write(int fd);//写文件命令

void exitsys();//依次关闭、打开文件，写入 FILENAME 文件

int my\_open(char\* filename);//打开文件命令

int my\_close(int fd);//关闭文件命令

void my\_mkdir(char\* dirname);//新建文件夹

void my\_rmdir(char\* dirname);//删除文件夹

int my\_create(char\* filename);//新建文件命令

void my\_rm(char\* filename);//删除文件命令

void my\_ls();//展示目录下所有文件

void help();//帮助指令

int get\_free\_openfilelist();//打开查询FCB中空闲的区域

unsigned short int get\_free\_block();//获取数据块

int main(void)

{

    char cmd[13][10] = {

        "mkdir", "rmdir", "ls", "cd", "create",

        "rm", "open", "close", "write", "read",

        "exit", "help", "test"

    };

    char command[30], \*sp;

    int cmd\_idx, i;

    startsys();

    help();

    while (1) {

        printf("%s$ ", openfilelist[currfd].dir);//dollar符号更有文件系统的感觉

        gets(command);

        cmd\_idx = -1;

        if (strcmp(command, "") == 0) {

            printf("\n");

            continue;

        }

        sp = strtok(command, " ");

        for (i = 0; i < 13; i++) {

            if (strcmp(sp, cmd[i]) == 0) {

                cmd\_idx = i;

                break;

            }

        }

        switch (cmd\_idx) {

        case 0: // mkdir

            sp = strtok(NULL, " ");

            if (sp != NULL)

                my\_mkdir(sp);

            else

                printf("mkdir指令错误！\n");

            break;

        case 1: // rmdir

            sp = strtok(NULL, " ");

            if (sp != NULL)

                my\_rmdir(sp);

            else

                printf("rmdir 指令错误！\n");

            break;

        case 2: // ls

            my\_ls();

            break;

        case 3: // cd

            sp = strtok(NULL, " ");

            if (sp != NULL)

                my\_cd(sp);

            else

                printf("cd 指令错误！\n");

            break;

        case 4: // create

            sp = strtok(NULL, " ");

            if (sp != NULL)

                my\_create(sp);

            else

                printf("create 指令错误！\n");

            break;

        case 5: // rm

            sp = strtok(NULL, " ");

            if (sp != NULL)

                my\_rm(sp);

            else

                printf("rm 指令错误！\n");

            break;

        case 6: // open

            sp = strtok(NULL, " ");

            if (sp != NULL)

                my\_open(sp);

            else

                printf("open 指令错误！\n");

            break;

        case 7: // close

            if (openfilelist[currfd].attribute == 1)

                my\_close(currfd);

            else

                printf("目前没有已打开的文件\n");

            break;

        case 8: // write

            if (openfilelist[currfd].attribute == 1)

                my\_write(currfd);

            else

                printf("在写文件之前请先打开文件\n");

            break;

        case 9: // read

            if (openfilelist[currfd].attribute == 1)

                my\_read(currfd);

            else

                printf("在读文件之前请先打开文件\n");

            break;

        case 10: // exit

            exitsys();

            printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 退出文件系统 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

            return 0;

            break;

        case 11: // help

            help();

            break;

        default:

            printf("错误的指令！: %s\n", command);

            break;

        }

    }

    return 0;

}

//启动文件系统

void startsys()

{

    /\*\*

     \* 如果存在文件系统（存在 FILENAME 这个文件 且 开头为魔数）则

     \* 将 user 目录载入打开文件表。

     \* 否则，调用 my\_format 创建文件系统，再载入。

     \*/

    myvhard = (unsigned char\*)malloc(SIZE);

    FILE\* file;

    if ((file = fopen(FILENAME, "r")) != NULL) {

        fread(buffer, SIZE, 1, file);

        fclose(file);

        if (memcmp(buffer, "10101010", 8) == 0) {

            memcpy(myvhard, buffer, SIZE);

            printf("文件系统已成功读入！\n");

        } else {

            printf("错误的文件系统格式，现重新创建\n");

            my\_format();

            memcpy(buffer, myvhard, SIZE);

        }

    } else {

        printf("未找到存在的文件系统，现新建一个文件系统\n");

        my\_format();

        memcpy(buffer, myvhard, SIZE);

    }

    fcb\* user;

    user = (fcb\*)(myvhard + 5 \* BLOCKSIZE);

    strcpy(openfilelist[0].filename, user->filename);

    strcpy(openfilelist[0].exname, user->exname);

    openfilelist[0].attribute = user->attribute;

    openfilelist[0].time = user->time;

    openfilelist[0].date = user->date;

    openfilelist[0].first = user->first;

    openfilelist[0].length = user->length;

    openfilelist[0].free = user->free;

    openfilelist[0].dirno = 5;

    openfilelist[0].diroff = 0;

    strcpy(openfilelist[0].dir, "/user/");

    openfilelist[0].count = 0;

    openfilelist[0].fcbstate = 0;

    openfilelist[0].topenfile = 1;

    startp = ((block0\*)myvhard)->startblock;

    currfd = 0;

    return;

}

//依次关闭、打开文件，写入 FILENAME 文件

void exitsys()

{

    while (currfd) {

        my\_close(currfd);

    }

    FILE\* fp = fopen(FILENAME, "w");

    fwrite(myvhard, SIZE, 1, fp);

    fclose(fp);

}

//格式化磁盘

void my\_format()

{

    /\*\*

     \* 初始化前五个磁盘块

     \* 设定第六个磁盘块为根目录磁盘块

     \* 初始化 user 目录： 创建 . 和 .. 目录

     \* 写入 FILENAME 文件 （写入磁盘空间）

     \*/

    block0\* boot = (block0\*)myvhard;

    strcpy(boot->magic\_number, "10101010");

    strcpy(boot->information, "fat file system");

    boot->user = 5;

    boot->startblock = myvhard + BLOCKSIZE \* 5;

    fat\* fat1 = (fat\*)(myvhard + BLOCKSIZE);

    fat\* fat2 = (fat\*)(myvhard + BLOCKSIZE \* 3);

    int i;

    for (i = 0; i < 6; i++) {

        fat1[i].id = END;

        fat2[i].id = END;

    }

    for (i = 6; i < 1000; i++) {

        fat1[i].id = FREE;

        fat2[i].id = FREE;

    }

    // 5th block is user

    fcb\* user = (fcb\*)(myvhard + BLOCKSIZE \* 5);

    strcpy(user->filename, ".");

    strcpy(user->exname, "di");

    user->attribute = 0; // dir file

    time\_t rawTime = time(NULL);

    struct tm\* time = localtime(&rawTime);

    // 5 6 5 bits

    user->time = time->tm\_hour \* 2048 + time->tm\_min \* 32 + time->tm\_sec / 2;

    // 7 4 5 bits; year from 2000

    user->date = (time->tm\_year - 100) \* 512 + (time->tm\_mon + 1) \* 32 + (time->tm\_mday);

    user->first = 5;

    user->free = 1;

    user->length = 2 \* sizeof(fcb);

    fcb\* root2 = user + 1;

    memcpy(root2, user, sizeof(fcb));

    strcpy(root2->filename, "..");

    for (i = 2; i < (int)(BLOCKSIZE / sizeof(fcb)); i++) {

        root2++;

        strcpy(root2->filename, "");

        root2->free = 0;

    }

    FILE\* fp = fopen(FILENAME, "w");

    fwrite(myvhard, SIZE, 1, fp);

    fclose(fp);

}

//展示目前路径下的所有文件

void my\_ls()

{

    // 判断是否是目录

    if (openfilelist[currfd].attribute == 1) {

        printf("文件数据：\n");

        return;

    }

    char buf[MAX\_TEXT\_SIZE];

    int i;

    // 读取当前目录文件信息(一个个fcb), 载入内存

    openfilelist[currfd].count = 0;

    do\_read(currfd, openfilelist[currfd].length, buf);

    // 遍历当前目录 fcb

    fcb\* fcbptr = (fcb\*)buf;

    for (i = 0; i < (int)(openfilelist[currfd].length / sizeof(fcb)); i++, fcbptr++) {

        if (fcbptr->free == 1) {

            if (fcbptr->attribute == 0) {

                printf("<DIR> %-8s\t%d/%d/%d %d:%d\n",

                    fcbptr->filename,

                    (fcbptr->date >> 9) + 2000,

                    (fcbptr->date >> 5) & 0x000f,

                    (fcbptr->date) & 0x001f,

                    (fcbptr->time >> 11),

                    (fcbptr->time >> 5) & 0x003f);

            } else {

                printf("<---> %-8s\t%d/%d/%d %d:%d\t%d\n",

                    fcbptr->filename,

                    (fcbptr->date >> 9) + 2000,

                    (fcbptr->date >> 5) & 0x000f,

                    (fcbptr->date) & 0x001f,

                    (fcbptr->time >> 11),

                    (fcbptr->time >> 5) & 0x003f,

                    fcbptr->length);

            }

        }

    }

}

//新建文件夹

void my\_mkdir(char\* dirname)

{

    /\*\*

     \* 当前目录：当前打开目录项表示的目录

     \* 该目录：以下指创建的目录

     \* 父目录：指该目录的父目录

     \* 如:

     \* 我现在在 user 目录下， 输入命令 mkdir a/b/bb

     \* 表示 在 user 目录下的 a 目录下的 b 目录中创建 bb 目录

     \* 这时，父目录指 b，该目录指 bb，当前目录指 user

     \* 以下都用这个表达，简单情况下，当前目录和父目录是一个目录

     \*/

    int i = 0;

    char text[MAX\_TEXT\_SIZE];

    char\* fname = strtok(dirname, ".");

    char\* exname = strtok(NULL, ".");

    if (exname != NULL) {

        printf("无法延伸出文件夹\n");

        return;

    }

    // 读取父目录信息

    openfilelist[currfd].count = 0;

    int filelen = do\_read(currfd, openfilelist[currfd].length, text);

    fcb\* fcbptr = (fcb\*)text;

    // 查找是否重名

    for (i = 0; i < (int)(filelen / sizeof(fcb)); i++) {

        if (strcmp(dirname, fcbptr[i].filename) == 0 && fcbptr->attribute == 0) {

            printf("该文件夹已存在！\n");

            return;

        }

    }

    // 申请一个打开目录表项

    int fd = get\_free\_openfilelist();

    if (fd == -1) {

        printf("打开的目录数量过多！\n");

        return;

    }

    // 申请一个磁盘块

    unsigned short int block\_num = get\_free\_block();

    if (block\_num == END) {

        printf("磁盘块已满！\n");

        openfilelist[fd].topenfile = 0;

        return;

    }

    // 更新 fat 表

    fat\* fat1 = (fat\*)(myvhard + BLOCKSIZE);

    fat\* fat2 = (fat\*)(myvhard + BLOCKSIZE \* 3);

    fat1[block\_num].id = END;

    fat2[block\_num].id = END;

    // 在父目录中找一个空的 fcb，分配给该目录  ??未考虑父目录满的情况??

    for (i = 0; i < (int)(filelen / sizeof(fcb)); i++) {

        if (fcbptr[i].free == 0) {

            break;

        }

    }

    openfilelist[currfd].count = i \* sizeof(fcb);

    openfilelist[currfd].fcbstate = 1;

    // 初始化该 fcb

    fcb\* fcbtmp = (fcb\*)malloc(sizeof(fcb));

    fcbtmp->attribute = 0;

    time\_t rawtime = time(NULL);

    struct tm\* time = localtime(&rawtime);

    fcbtmp->date = (time->tm\_year - 100) \* 512 + (time->tm\_mon + 1) \* 32 + (time->tm\_mday);

    fcbtmp->time = (time->tm\_hour) \* 2048 + (time->tm\_min) \* 32 + (time->tm\_sec) / 2;

    strcpy(fcbtmp->filename, dirname);

    strcpy(fcbtmp->exname, "di");

    fcbtmp->first = block\_num;

    fcbtmp->length = 2 \* sizeof(fcb);

    fcbtmp->free = 1;

    do\_write(currfd, (char\*)fcbtmp, sizeof(fcb), 2);

    // 设置打开文件表项

    openfilelist[fd].attribute = 0;

    openfilelist[fd].count = 0;

    openfilelist[fd].date = fcbtmp->date;

    openfilelist[fd].time = fcbtmp->time;

    openfilelist[fd].dirno = openfilelist[currfd].first;

    openfilelist[fd].diroff = i;

    strcpy(openfilelist[fd].exname, "di");

    strcpy(openfilelist[fd].filename, dirname);

    openfilelist[fd].fcbstate = 0;

    openfilelist[fd].first = fcbtmp->first;

    openfilelist[fd].free = fcbtmp->free;

    openfilelist[fd].length = fcbtmp->length;

    openfilelist[fd].topenfile = 1;

    strcat(strcat(strcpy(openfilelist[fd].dir, (char\*)(openfilelist[currfd].dir)), dirname), "/");

    // 设置 . 和 .. 目录

    fcbtmp->attribute = 0;

    fcbtmp->date = fcbtmp->date;

    fcbtmp->time = fcbtmp->time;

    strcpy(fcbtmp->filename, ".");

    strcpy(fcbtmp->exname, "di");

    fcbtmp->first = block\_num;

    fcbtmp->length = 2 \* sizeof(fcb);

    do\_write(fd, (char\*)fcbtmp, sizeof(fcb), 2);

    fcb\* fcbtmp2 = (fcb\*)malloc(sizeof(fcb));

    memcpy(fcbtmp2, fcbtmp, sizeof(fcb));

    strcpy(fcbtmp2->filename, "..");

    fcbtmp2->first = openfilelist[currfd].first;

    fcbtmp2->length = openfilelist[currfd].length;

    fcbtmp2->date = openfilelist[currfd].date;

    fcbtmp2->time = openfilelist[currfd].time;

    do\_write(fd, (char\*)fcbtmp2, sizeof(fcb), 2);

    // 关闭该目录的打开文件表项，close 会修改父目录中对应该目录的 fcb 信息

    /\*\*

     \* 这里注意，一个目录存在 2 个 fcb 信息，一个为该目录下的 . 目录文件，一个为父目录下的 fcb。

     \* 因此，这俩个fcb均需要修改，前一个 fcb 由各个函数自己完成，后一个 fcb 修改由 close 完成。

     \* 所以这里，需要打开文件表，再关闭文件表，实际上更新了后一个 fcb 信息。

     \*/

    my\_close(fd);

    free(fcbtmp);

    free(fcbtmp2);

    // 修改父目录 fcb

    fcbptr = (fcb\*)text;

    fcbptr->length = openfilelist[currfd].length;

    openfilelist[currfd].count = 0;

    do\_write(currfd, (char\*)fcbptr, sizeof(fcb), 2);

    openfilelist[currfd].fcbstate = 1;

}

//删除目录

void my\_rmdir(char\* dirname)

{

    int i, tag = 0;

    char buf[MAX\_TEXT\_SIZE];

    // 排除 . 和 .. 目录

    if (strcmp(dirname, ".") == 0 || strcmp(dirname, "..") == 0) {

        printf("无法移除，权限不足\n");

        return;

    }

    openfilelist[currfd].count = 0;

    do\_read(currfd, openfilelist[currfd].length, buf);

    // 查找要删除的目录

    fcb\* fcbptr = (fcb\*)buf;

    for (i = 0; i < (int)(openfilelist[currfd].length / sizeof(fcb)); i++, fcbptr++) {

        if (fcbptr->free == 0)

            continue;

        if (strcmp(fcbptr->filename, dirname) == 0 && fcbptr->attribute == 0) {

            tag = 1;

            break;

        }

    }

    if (tag != 1) {

        printf("没有这个文件夹\n");

        return;

    }

    // 无法删除非空目录

    if (fcbptr->length > 2 \* sizeof(fcb)) {

        printf("无法删除非空目录的文件夹\n");

        return;

    }

    // 更新 fat 表

    int block\_num = fcbptr->first;

    fat\* fat1 = (fat\*)(myvhard + BLOCKSIZE);

    int nxt\_num = 0;

    while (1) {

        nxt\_num = fat1[block\_num].id;

        fat1[block\_num].id = FREE;

        if (nxt\_num != END) {

            block\_num = nxt\_num;

        } else {

            break;

        }

    }

    fat1 = (fat\*)(myvhard + BLOCKSIZE);

    fat\* fat2 = (fat\*)(myvhard + BLOCKSIZE \* 3);

    memcpy(fat2, fat1, BLOCKSIZE \* 2);

    // 更新 fcb

    fcbptr->date = 0;

    fcbptr->time = 0;

    fcbptr->exname[0] = '\0';

    fcbptr->filename[0] = '\0';

    fcbptr->first = 0;

    fcbptr->free = 0;

    fcbptr->length = 0;

    openfilelist[currfd].count = i \* sizeof(fcb);

    do\_write(currfd, (char\*)fcbptr, sizeof(fcb), 2);

    // 删除目录需要相应考虑可能删除 fcb，也就是修改父目录 length

    // 这里需要注意：因为删除中间的 fcb，目录有效长度不变，即 length 不变

    // 因此需要考虑特殊情况，即删除最后一个 fcb 时，极有可能之前的 fcb 都是空的，这是需要

    // 循环删除 fcb (以下代码完成)，可能需要回收 block 修改 fat 表等过程(do\_write 完成)

    int lognum = i;

    if ((lognum + 1) \* sizeof(fcb) == openfilelist[currfd].length) {

        openfilelist[currfd].length -= sizeof(fcb);

        lognum--;

        fcbptr = (fcb \*)buf + lognum;

        while (fcbptr->free == 0) {

            fcbptr--;

            openfilelist[currfd].length -= sizeof(fcb);

        }

    }

    // 更新父目录 fcb

    fcbptr = (fcb\*)buf;

    fcbptr->length = openfilelist[currfd].length;

    openfilelist[currfd].count = 0;

    do\_write(currfd, (char\*)fcbptr, sizeof(fcb), 2);

    openfilelist[currfd].fcbstate = 1;

}

//新建文件命令

int my\_create(char\* filename)

{

    // 非法判断

    if (strcmp(filename, "") == 0) {

        printf("请输入文件名\n");

        return -1;

    }

    if (openfilelist[currfd].attribute == 1) {

        printf("目前状态是一个数据文件\n");

        return -1;

    }

    openfilelist[currfd].count = 0;

    char buf[MAX\_TEXT\_SIZE];

    do\_read(currfd, openfilelist[currfd].length, buf);

    int i;

    fcb\* fcbptr = (fcb\*)buf;

    // 检查重名

    for (i = 0; i < (int)(openfilelist[currfd].length / sizeof(fcb)); i++, fcbptr++) {

        if (fcbptr->free == 0) {

            continue;

        }

        if (strcmp(fcbptr->filename, filename) == 0 && fcbptr->attribute == 1) {

            printf("已有同名文件！\n");

            return -1;

        }

    }

    // 申请空 fcb;

    fcbptr = (fcb\*)buf;

    for (i = 0; i < (int)(openfilelist[currfd].length / sizeof(fcb)); i++, fcbptr++) {

        if (fcbptr->free == 0)

            break;

    }

    // 申请磁盘块并更新 fat 表

    int block\_num = get\_free\_block();

    if (block\_num == -1) {

        return -1;

    }

    fat\* fat1 = (fat\*)(myvhard + BLOCKSIZE);

    fat\* fat2 = (fat\*)(myvhard + BLOCKSIZE \* 3);

    fat1[block\_num].id = END;

    memcpy(fat2, fat1, BLOCKSIZE \* 2);

    // 修改 fcb 信息

    strcpy(fcbptr->filename, filename);

    time\_t rawtime = time(NULL);

    struct tm\* time = localtime(&rawtime);

    fcbptr->date = (time->tm\_year - 100) \* 512 + (time->tm\_mon + 1) \* 32 + (time->tm\_mday);

    fcbptr->time = (time->tm\_hour) \* 2048 + (time->tm\_min) \* 32 + (time->tm\_sec) / 2;

    fcbptr->first = block\_num;

    fcbptr->free = 1;

    fcbptr->attribute = 1;

    fcbptr->length = 0;

    openfilelist[currfd].count = i \* sizeof(fcb);

    do\_write(currfd, (char\*)fcbptr, sizeof(fcb), 2);

    // 修改父目录 fcb

    fcbptr = (fcb\*)buf;

    fcbptr->length = openfilelist[currfd].length;

    openfilelist[currfd].count = 0;

    do\_write(currfd, (char\*)fcbptr, sizeof(fcb), 2);

    openfilelist[currfd].fcbstate = 1;

}

//删除文件命令

void my\_rm(char\* filename)

{

    char buf[MAX\_TEXT\_SIZE];

    openfilelist[currfd].count = 0;

    do\_read(currfd, openfilelist[currfd].length, buf);

    int i, flag = 0;

    fcb\* fcbptr = (fcb\*)buf;

    // 查询

    for (i = 0; i < (int)(openfilelist[currfd].length / sizeof(fcb)); i++, fcbptr++) {

        if (strcmp(fcbptr->filename, filename) == 0 && fcbptr->attribute == 1) {

            flag = 1;

            break;

        }

    }

    if (flag != 1) {

        printf("没有这个文件！\n");

        return;

    }

    // 更新 fat 表

    int block\_num = fcbptr->first;

    fat\* fat1 = (fat\*)(myvhard + BLOCKSIZE);

    int nxt\_num = 0;

    while (1) {

        nxt\_num = fat1[block\_num].id;

        fat1[block\_num].id = FREE;

        if (nxt\_num != END)

            block\_num = nxt\_num;

        else

            break;

    }

    fat1 = (fat\*)(myvhard + BLOCKSIZE);

    fat\* fat2 = (fat\*)(myvhard + BLOCKSIZE \* 3);

    memcpy(fat2, fat1, BLOCKSIZE \* 2);

    // 清空 fcb

    fcbptr->date = 0;

    fcbptr->time = 0;

    fcbptr->exname[0] = '\0';

    fcbptr->filename[0] = '\0';

    fcbptr->first = 0;

    fcbptr->free = 0;

    fcbptr->length = 0;

    openfilelist[currfd].count = i \* sizeof(fcb);

    do\_write(currfd, (char\*)fcbptr, sizeof(fcb), 2);

    //

    int lognum = i;

    if ((lognum + 1) \* sizeof(fcb) == openfilelist[currfd].length) {

        openfilelist[currfd].length -= sizeof(fcb);

        lognum--;

        fcbptr = (fcb \*)buf + lognum;

        while (fcbptr->free == 0) {

            fcbptr--;

            openfilelist[currfd].length -= sizeof(fcb);

        }

    }

    // 修改父目录 . 目录文件的 fcb

    fcbptr = (fcb\*)buf;

    fcbptr->length = openfilelist[currfd].length;

    openfilelist[currfd].count = 0;

    do\_write(currfd, (char\*)fcbptr, sizeof(fcb), 2);

    openfilelist[currfd].fcbstate = 1;

}

//打开文件命令

int my\_open(char\* filename)

{

    char buf[MAX\_TEXT\_SIZE];

    openfilelist[currfd].count = 0;

    do\_read(currfd, openfilelist[currfd].length, buf);

    int i, flag = 0;

    fcb\* fcbptr = (fcb\*)buf;

    // 重名检查

    for (i = 0; i < (int)(openfilelist[currfd].length / sizeof(fcb)); i++, fcbptr++) {

        if (strcmp(fcbptr->filename, filename) == 0 && fcbptr->attribute == 1) {

            flag = 1;

            break;

        }

    }

    if (flag != 1) {

        printf("没有这个文件！\n");

        return -1;

    }

    // 申请新的打开目录项并初始化该目录项

    int fd = get\_free\_openfilelist();

    if (fd == -1) {

        printf("my\_open: 目录项已满\n");

        return -1;

    }

    openfilelist[fd].attribute = 1;

    openfilelist[fd].count = 0;

    openfilelist[fd].date = fcbptr->date;

    openfilelist[fd].time = fcbptr->time;

    openfilelist[fd].length = fcbptr->length;

    openfilelist[fd].first = fcbptr->first;

    openfilelist[fd].free = 1;

    strcpy(openfilelist[fd].filename, fcbptr->filename);

    strcat(strcpy(openfilelist[fd].dir, (char\*)(openfilelist[currfd].dir)), filename);

    openfilelist[fd].dirno = openfilelist[currfd].first;

    openfilelist[fd].diroff = i;

    openfilelist[fd].topenfile = 1;

    openfilelist[fd].fcbstate = 0;

    currfd = fd;

    return 1;

}

//进入到路径内

void my\_cd(char\* dirname)

{

    int i = 0;

    int tag = -1;

    int fd;

    if (openfilelist[currfd].attribute == 1) {

        // if not a dir

        printf("你选择的是一个文件而不是文件夹, 可以通过close关闭该文件\n");

        return;

    }

    char\* buf = (char\*)malloc(10000);

    openfilelist[currfd].count = 0;

    do\_read(currfd, openfilelist[currfd].length, buf);

    fcb\* fcbptr = (fcb\*)buf;

    // 查找目标 fcb

    for (i = 0; i < (int)(openfilelist[currfd].length / sizeof(fcb)); i++, fcbptr++) {

        if (strcmp(fcbptr->filename, dirname) == 0 && fcbptr->attribute == 0) {

            tag = 1;

            break;

        }

    }

    if (tag != 1) {

        printf("my\_cd: 没有这个文件夹！\n");

        return;

    } else {

        // . 和 .. 检查

        if (strcmp(fcbptr->filename, ".") == 0) {

            return;

        } else if (strcmp(fcbptr->filename, "..") == 0) {

            if (currfd == 0) {

                // user

                return;

            } else {

                currfd = my\_close(currfd);

                return;

            }

        } else {

            // 其他目录

            fd = get\_free\_openfilelist();

            if (fd == -1) {

                return;

            }

            openfilelist[fd].attribute = fcbptr->attribute;

            openfilelist[fd].count = 0;

            openfilelist[fd].date = fcbptr->date;

            openfilelist[fd].time = fcbptr->time;

            strcpy(openfilelist[fd].filename, fcbptr->filename);

            strcpy(openfilelist[fd].exname, fcbptr->exname);

            openfilelist[fd].first = fcbptr->first;

            openfilelist[fd].free = fcbptr->free;

            openfilelist[fd].fcbstate = 0;

            openfilelist[fd].length = fcbptr->length;

            strcat(strcat(strcpy(openfilelist[fd].dir, (char\*)(openfilelist[currfd].dir)), dirname), "/");

            openfilelist[fd].topenfile = 1;

            openfilelist[fd].dirno = openfilelist[currfd].first;

            openfilelist[fd].diroff = i;

            currfd = fd;

        }

    }

}

//关闭文件命令

int my\_close(int fd)

{

    if (fd > MAXOPENFILE || fd < 0) {

        printf("my\_close: 出错！\n");

        return -1;

    }

    int i;

    char buf[MAX\_TEXT\_SIZE];

    int father\_fd = -1;

    fcb\* fcbptr;

    for (i = 0; i < MAXOPENFILE; i++) {

        if (openfilelist[i].first == openfilelist[fd].dirno) {

            father\_fd = i;

            break;

        }

    }

    if (father\_fd == -1) {

        printf("my\_close: 没有父文件\n");

        return -1;

    }

    if (openfilelist[fd].fcbstate == 1) {

        do\_read(father\_fd, openfilelist[father\_fd].length, buf);

        // update fcb

        fcbptr = (fcb\*)(buf + sizeof(fcb) \* openfilelist[fd].diroff);

        strcpy(fcbptr->exname, openfilelist[fd].exname);

        strcpy(fcbptr->filename, openfilelist[fd].filename);

        fcbptr->first = openfilelist[fd].first;

        fcbptr->free = openfilelist[fd].free;

        fcbptr->length = openfilelist[fd].length;

        fcbptr->time = openfilelist[fd].time;

        fcbptr->date = openfilelist[fd].date;

        fcbptr->attribute = openfilelist[fd].attribute;

        openfilelist[father\_fd].count = openfilelist[fd].diroff \* sizeof(fcb);

        do\_write(father\_fd, (char\*)fcbptr, sizeof(fcb), 2);

    }

    // 释放打开文件表

    memset(&openfilelist[fd], 0, sizeof(useropen));

    currfd = father\_fd;

    return father\_fd;

}

//读文件命令

int my\_read(int fd)

{

    if (fd < 0 || fd >= MAXOPENFILE) {

        printf("没有文件！\n");

        return -1;

    }

    openfilelist[fd].count = 0;

    char text[MAX\_TEXT\_SIZE] = "\0";

    do\_read(fd, openfilelist[fd].length, text);

    printf("%s\n", text);

    return 1;

}

//写文件命令

int my\_write(int fd)

{

    if (fd < 0 || fd >= MAXOPENFILE) {

        printf("my\_write: 没有文件！\n");

        return -1;

    }

    int wstyle;

    while (1) {

        // 1: 截断写，清空全部内容，从头开始写

        // 2. 覆盖写，从文件指针处开始写

        // 3. 追加写，字面意思

        printf("1: 截断写，清空全部内容，从头开始写\n  2:覆盖写，从文件指针处开始写\n  3:追加写\n");

        scanf("%d", &wstyle);

        if (wstyle > 3 || wstyle < 1) {

            printf("input error\n");

        } else {

            break;

        }

    }

    char text[MAX\_TEXT\_SIZE] = "\0";

    char texttmp[MAX\_TEXT\_SIZE] = "\0";

    printf("请输入相关数据, line feed + $$ to end file\n");

    getchar();

    while (gets(texttmp)) {

        if (strcmp(texttmp, "$$") == 0) {

            break;

        }

        texttmp[strlen(texttmp)] = '\n';

        strcat(text, texttmp);

    }

    text[strlen(text)] = '\0';

    do\_write(fd, text, strlen(text) + 1, wstyle);

    openfilelist[fd].fcbstate = 1;

    return 1;

}

//读文件具体操作

int do\_read(int fd, int len, char\* text)

{

    int len\_tmp = len;

    char\* textptr = text;

    unsigned char\* buf = (unsigned char\*)malloc(1024);

    if (buf == NULL) {

        printf("内存出错！\n");

        return -1;

    }

    int off = openfilelist[fd].count;

    int block\_num = openfilelist[fd].first;

    fat\* fatptr = (fat\*)(myvhard + BLOCKSIZE) + block\_num;

    // 定位读取目标磁盘块和块内地址

    while (off >= BLOCKSIZE) {

        off -= BLOCKSIZE;

        block\_num = fatptr->id;

        if (block\_num == END) {

            printf("do\_read: 磁盘块不存在！\n");

            return -1;

        }

        fatptr = (fat\*)(myvhard + BLOCKSIZE) + block\_num;

    }

    unsigned char\* blockptr = myvhard + BLOCKSIZE \* block\_num;

    memcpy(buf, blockptr, BLOCKSIZE);

    // 读取内容

    while (len > 0) {

        if (BLOCKSIZE - off > len) {

            memcpy(textptr, buf + off, len);

            textptr += len;

            off += len;

            openfilelist[fd].count += len;

            len = 0;

        } else {

            memcpy(textptr, buf + off, BLOCKSIZE - off);

            textptr += BLOCKSIZE - off;

            len -= BLOCKSIZE - off;

            block\_num = fatptr->id;

            if (block\_num == END) {

                printf("do\_read: 长度过长！\n");

                break;

            }

            fatptr = (fat\*)(myvhard + BLOCKSIZE) + block\_num;

            blockptr = myvhard + BLOCKSIZE \* block\_num;

            memcpy(buf, blockptr, BLOCKSIZE);

        }

    }

    free(buf);

    return len\_tmp - len;

}

//写文件具体操作

int do\_write(int fd, char\* text, int len, char wstyle)

{

    int block\_num = openfilelist[fd].first;

    int i, tmp\_num;

    int lentmp = 0;

    char\* textptr = text;

    char buf[BLOCKSIZE];

    fat\* fatptr = (fat\*)(myvhard + BLOCKSIZE) + block\_num;

    unsigned char\* blockptr;

    if (wstyle == 1) {

        openfilelist[fd].count = 0;

        openfilelist[fd].length = 0;

    } else if (wstyle == 3) {

        // 追加写，如果是一般文件，则需要先删除末尾 \0，即将指针移到末位减一个字节处

        openfilelist[fd].count = openfilelist[fd].length;

        if (openfilelist[fd].attribute == 1) {

            if (openfilelist[fd].length != 0) {

                // 非空文件

                openfilelist[fd].count = openfilelist[fd].length - 1;

            }

        }

    }

    int off = openfilelist[fd].count;

    // 定位磁盘块和块内偏移量

    while (off >= BLOCKSIZE) {

        block\_num = fatptr->id;

        if (block\_num == END) {

            printf("do\_write: 关闭异常\n");

            return -1;

        }

        fatptr = (fat\*)(myvhard + BLOCKSIZE) + block\_num;

        off -= BLOCKSIZE;

    }

    blockptr = (unsigned char\*)(myvhard + BLOCKSIZE \* block\_num);

    // 写入磁盘

    while (len > lentmp) {

        memcpy(buf, blockptr, BLOCKSIZE);

        for (; off < BLOCKSIZE; off++) {

            \*(buf + off) = \*textptr;

            textptr++;

            lentmp++;

            if (len == lentmp)

                break;

        }

        memcpy(blockptr, buf, BLOCKSIZE);

        // 写入的内容太多，需要写到下一个磁盘块，如果没有磁盘块，就申请一个

        if (off == BLOCKSIZE && len != lentmp) {

            off = 0;

            block\_num = fatptr->id;

            if (block\_num == END) {

                block\_num = get\_free\_block();

                if (block\_num == END) {

                    printf("do\_write: 磁盘块已满\n");

                    return -1;

                }

                blockptr = (unsigned char\*)(myvhard + BLOCKSIZE \* block\_num);

                fatptr->id = block\_num;

                fatptr = (fat\*)(myvhard + BLOCKSIZE) + block\_num;

                fatptr->id = END;

            } else {

                blockptr = (unsigned char\*)(myvhard + BLOCKSIZE \* block\_num);

                fatptr = (fat\*)(myvhard + BLOCKSIZE) + block\_num;

            }

        }

    }

    openfilelist[fd].count += len;

    if (openfilelist[fd].count > openfilelist[fd].length)

        openfilelist[fd].length = openfilelist[fd].count;

    // 删除多余的磁盘块

    if (wstyle == 1 || (wstyle == 2 && openfilelist[fd].attribute == 0)) {

        off = openfilelist[fd].length;

        fatptr = (fat \*)(myvhard + BLOCKSIZE) + openfilelist[fd].first;

        while (off >= BLOCKSIZE) {

            block\_num = fatptr->id;

            off -= BLOCKSIZE;

            fatptr = (fat \*)(myvhard + BLOCKSIZE) + block\_num;

        }

        while (1) {

            if (fatptr->id != END) {

                i = fatptr->id;

                fatptr->id = FREE;

                fatptr = (fat \*)(myvhard + BLOCKSIZE) + i;

            } else {

                fatptr->id = FREE;

                break;

            }

        }

        fatptr = (fat \*)(myvhard + BLOCKSIZE) + block\_num;

        fatptr->id = END;

    }

    memcpy((fat\*)(myvhard + BLOCKSIZE \* 3), (fat\*)(myvhard + BLOCKSIZE), BLOCKSIZE \* 2);

    return len;

}

//打开查询FCB中空闲的区域

int get\_free\_openfilelist()

{

    int i;

    for (i = 0; i < MAXOPENFILE; i++) {

        if (openfilelist[i].topenfile == 0) {

            openfilelist[i].topenfile = 1;

            return i;

        }

    }

    return -1;

}

//获取空闲的数据块

unsigned short int get\_free\_block()

{

    int i;

    fat\* fat1 = (fat\*)(myvhard + BLOCKSIZE);

    for (i = 0; i < (int)(SIZE / BLOCKSIZE); i++) {

        if (fat1[i].id == FREE) {

            return i;

        }

    }

    return END;

}

//帮助界面

void help(){

    printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    printf("\*欢迎来到15组的文件系统！！！                   \n");

    printf("\*主要的功能介绍                                     \n");

    printf("\*    0.创建文件夹 :mkdir                 \n");

    printf("\*    1.删除文件夹    :rmdir                              \n");

    printf("\*    2. 展示路径下文件:ls                          \n");

    printf("\*    3.进入路径内  :cd                              \n");

    printf("\*    4.创建文件  :create                            \n");

    printf("\*    5.删除文件 :rm                          \n");

    printf("\*    6.打开文件  :open                          \n");

    printf("\*    7.关闭文件 :close                            \n");

    printf("\*    8.写入文件  :write                              \n");

    printf("\*    9.阅读文件  :read                                \n");

    printf("\*    10.退出     :exit                              \n");

    printf("\*    11.帮助     :help                             \n");

    printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

五．调试方案及调试过程问题分析

【C++不兼容大量的C实现的内核函数易报错，因此采用了C语言】

编译：gcc file.c -o file

运行：./file



图5.1文件管理系统的界面

最上方为文件系统（磁盘）成功加载的提示栏，实成功与否取决于路径中是否有zlilesys文件



图5.2 文件系统磁盘

若该文件不在路径下，则启动界面如下：

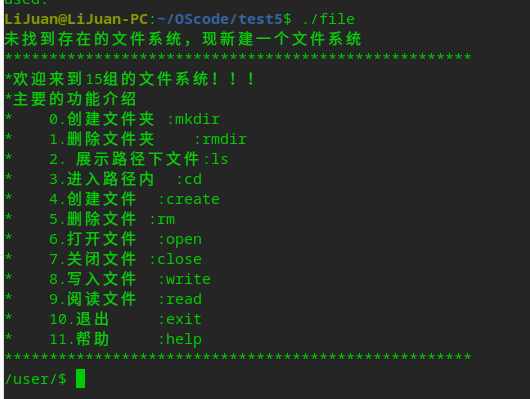


图5.3 初始无磁盘文件的结果

程序会自动为其新建一个文件系统，随后进行接下来的操作。

$dollar符号能使我们代入进Linux系统的感觉

六．运行结果与分析



图5.4 mkdir操作

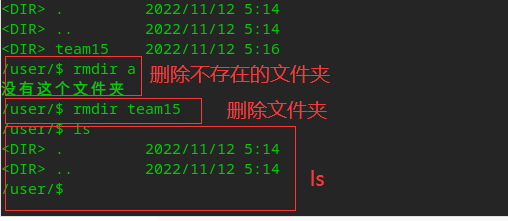


图5.5 rmdir操作

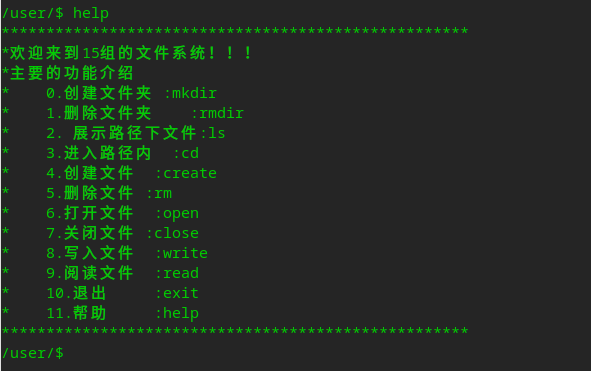


图5.6 帮助界面

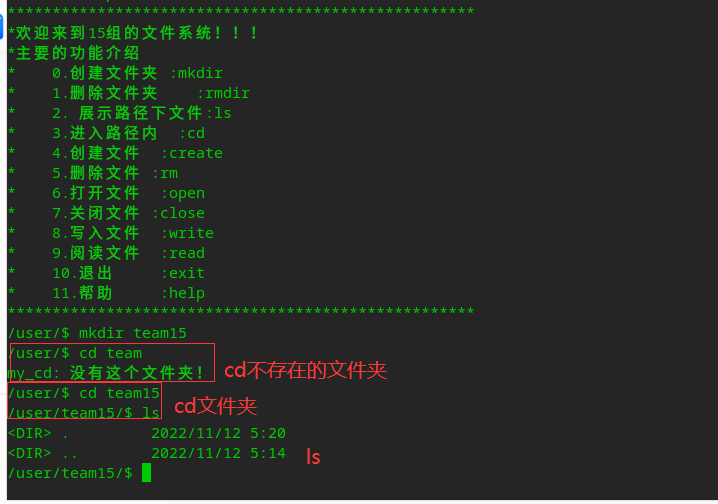


图5.7 cd操作

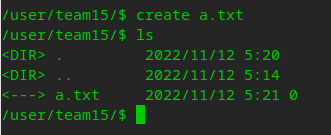


图5.8 create操作

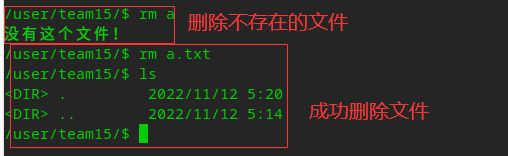


图5.9 rm操作

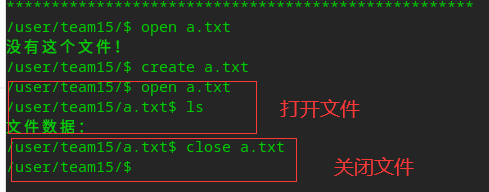


图5.10 open、close操作

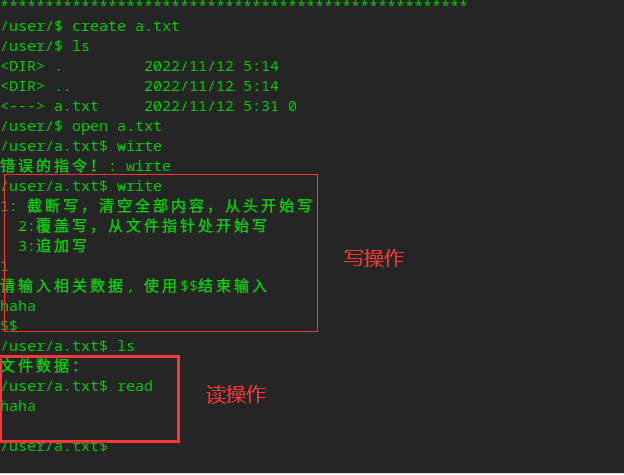


图5.11 write、read操作



图5.12 退出文件系统

2.运行结果分析

所有的自行手写操作对应的关键词都是我们熟悉的指令如mkdir、ls，表面看起来很简单轻松，但实际上文件管理的数据结构比我们做过的任何一个实验都复杂。但是这一定是个实用的文件管理系统。如果将其与其他实验结合在一起，其就是Unix系统的简化版。

七．总结

1.设计中的经验体会

（1）只实现了预想中的部分功能，并且许多功能还有不完善的地方；

（2）代码规范，整洁程度有待提高；

（3）部分功能的实现过程中为了能够尽快完成，使用了一些可能存在可移植性或安全性问题的方法。

（4）磁盘系统函数的建立使用了很笨拙的方法，使得要增添新功能时工作量大。

2.后续设计设想

既然文件系统都已经写出来了，那么可以考虑与前面的实验结合一下，增添多线程方法、生产者消费者问题、优化内存分配还有磁盘管理。将这几项内容结合在一起其实就是一个简单的操作系统了。

该文件系统花费了1000多行，其中有许多冗杂部分值得优化。

以及文件系统看起来简单，但是内部复杂的过程没有用可视化方式呈现出来。

好好学习计算机操作系统，研究更多内核函数去优化个人实验！