**目录**

[**实验一 WINDOWS进程初识** 4](#_Toc211480726)

[**实验二 进程管理** 7](#_Toc211480731)

[**实验三 进程同步的经典算法** 11](#_Toc211480737)

[**实验四 存储管理** 15](#_Toc211480743)

[**试验五 文件系统试验** 19](#_Toc211480754)

**实验有关（参考）代码**

**实验一 WINDOWS进程初识**

1. **实验目的**
   1. 学会使用VC编写基本的Win32 Consol Application（控制台应用程序)。
   2. 掌握WINDOWS API的使用方法。
   3. 编写测试程序，理解用户态运行和核心态运行。
2. **程序清单**

清单1-1 一个简单的Windows控制台应用程序

// hello项目

# include <iostream>

void main()

{

std::cout << “Hello, Win32 Consol Application” << std :: endl ;

}

清单1-2 核心态运行和用户态运行时间比计算

// proclist项目

# include <windows.h>

# include <tlhelp32.h>

# include <iostream.h>

// 当在用户模式机内核模式下都提供所耗时间时，在内核模式下进行所耗时间的64位计算的帮助方法

DWORD GetKernelModePercentage(const FILETIME& ftKernel,

const FILETIME& ftUser)

{

// 将FILETIME结构转化为64位整数

ULONGLONG qwKernel=(((ULONGLONG)ftKernel.dwHighDateTime)

<<32)+ftKernel.dwLowDateTime;

ULONGLONG qwUser=(((ULONGLONG)ftUser.dwHighDateTime)

<<32)+ftUser.dwLowDateTime;

// 将消耗时间相加，然后计算消耗在内核模式下的时间百分比

ULONGLONG qwTotal=qwKernel+qwUser;

DWORD dwPct=(DWORD)(((ULONGLONG)100\*qwKernel)/qwTotal);

return(dwPct);

}

// 以下是将当前运行过程名和消耗在内核模式下的时间百分数都显示出来的应用程序

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

{

if(argc<2)

{

cout<<"请给出你要查询的程序名"<<endl;

exit(0);

}

// 对当前系统中运行的过程拍取“快照”

HANDLE hSnapshot=::CreateToolhelp32Snapshot(

TH32CS\_SNAPPROCESS, // 提取当前过程

0); // 如果是当前过程，就将其忽略

// 初始化过程入口

PROCESSENTRY32 pe;

::ZeroMemory(&pe,sizeof(pe));

pe.dwSize=sizeof(pe);

BOOL bMore=::Process32First(hSnapshot,&pe);

BOOL found = FALSE;

while(bMore)

{

// 打开用于读取的过程

if(!strcmp(pe.szExeFile,argv[1]))

{

found = TRUE;

HANDLE hProcess=::OpenProcess(

PROCESS\_QUERY\_INFORMATION, // 指明要得到信息

FALSE, // 不必继承这一句柄

pe.th32ProcessID); // 要打开的进程

if (hProcess!=NULL)

{

// 找出进程的时间

FILETIME ftCreation,ftKernelMode,ftUserMode,ftExit;

::GetProcessTimes(

hProcess, // 所感兴趣的进程

&ftCreation, // 进程的启动时间

&ftExit, // 结束时间 (如果有的话)

&ftKernelMode, // 在内核模式下消耗的时间

&ftUserMode); // 在用户模式下消耗的时间

// 计算内核模式消耗的时间百分比

DWORD dwPctKernel=::GetKernelModePercentage(

ftKernelMode, // 在内核模式上消耗的时间

ftUserMode); // 在用户模式下消耗的时间

// 向用户显示进程的某些信息

cout<< "process ID: " << pe.th32ProcessID

<< ",EXE file:" << pe.szExeFile

<< ",%d in Kernel mode: " << dwPctKernel << endl;

// 消除句柄

::CloseHandle(hProcess);

}

}

// 转向下一个进程

bMore=::Process32Next(hSnapshot,&pe);

}

if(found==FALSE)

{

cout<<"当前系统没有这个可执行程序正在运行"<<endl;

exit(0);

}

}

清单1-3 核心态运行和用户态运行时间测试程序

#include <stdio.h>

main()

{

int i,j;

while(1){

for(i=0;i<1000;i++);

for(j=1;j<1000;j++) printf(“enter kernel mode running.”);

}

}

**实验二 进程管理**

**1、实验目的**

1) 通过创建进程、观察正在运行的进程和终止进程的程序设计和调试操作，进一步熟悉操作系统的进程概念，理解Windows进程的“一生”。

2) 通过阅读和分析实验程序，学习创建进程、观察进程、终止进程以及父子进程同步的基本程序设计方法。

2. **程序清单**

清单2-1 创建子进程

// proccreate项目

#include <windows.h>

#include <iostream>

#include <stdio.h>

// 创建传递过来的进程的克隆过程并赋于其ID值

void StartClone(int nCloneID) {

// 提取用于当前可执行文件的文件名

TCHAR szFilename[MAX\_PATH] ;

GetModuleFileName(NULL, szFilename, MAX\_PATH) ;

// 格式化用于子进程的命令行并通知其EXE文件名和克隆ID

TCHAR szCmdLine[MAX\_PATH];

sprintf(szCmdLine,"\"%s\" %d",szFilename,nCloneID);

// 用于子进程的STARTUPINFO结构

STARTUPINFO si;

ZeroMemory(&si , sizeof(si) ) ;

si.cb = sizeof(si) ; // 必须是本结构的大小

// 返回的用于子进程的进程信息

PROCESS\_INFORMATION pi;

// 利用同样的可执行文件和命令行创建进程，并赋于其子进程的性质

BOOL bCreateOK=::CreateProcess(

szFilename, // 产生这个EXE的应用程序的名称

szCmdLine, // 告诉其行为像一个子进程的标志

NULL, // 缺省的进程安全性

NULL, // 缺省的线程安全性

FALSE, // 不继承句柄

CREATE\_NEW\_CONSOLE, // 使用新的控制台

NULL, // 新的环境

NULL, // 当前目录

&si, // 启动信息

&pi) ; // 返回的进程信息

// 对子进程释放引用

if (bCreateOK) {

CloseHandle(pi.hProcess) ;

CloseHandle(pi.hThread) ;

}

}

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

// 确定派生出几个进程，及派生进程在进程列表中的位置

int nClone=0;

//修改语句：int nClone;

//第一次修改：nClone=0;

if (argc > 1) {

// 从第二个参数中提取克隆ID

:: sscanf(argv[1] , "%d" , &nClone) ;

}

//第二次修改：nClone=0;

// 显示进程位置

std :: cout << "Process ID:" << :: GetCurrentProcessId()

<< ", Clone ID:" << nClone

<< std :: endl;

// 检查是否有创建子进程的需要

const int c\_nCloneMax=5;

if (nClone < c\_nCloneMax) {

// 发送新进程的命令行和克隆号

StartClone(++nClone) ;

}

// 等待响应键盘输入结束进程

getchar();

return 0;

}

清单2-2 父子进程的简单通信及终止进程的示例程序

// procterm项目

# include <windows.h>

# include <iostream>

# include <stdio.h>

static LPCTSTR g\_szMutexName = "w2kdg.ProcTerm.mutex.Suicide" ;

// 创建当前进程的克隆进程的简单方法

void StartClone() {

// 提取当前可执行文件的文件名

TCHAR szFilename[MAX\_PATH] ;

GetModuleFileName(NULL, szFilename, MAX\_PATH) ;

// 格式化用于子进程的命令行，字符串“child”将作为形参传递给子进程的main函数

TCHAR szCmdLine[MAX\_PATH] ;

//实验2-2步骤3：将下句中的字符串child改为别的字符串，重新编译执行，执行前请先保存已经完成的工作

sprintf(szCmdLine, "\"%s\"child" , szFilename) ;

// 子进程的启动信息结构

STARTUPINFO si;

ZeroMemory(&si,sizeof(si)) ;

si.cb = sizeof(si) ; // 应当是此结构的大小

// 返回的用于子进程的进程信息

PROCESS\_INFORMATION pi;

// 用同样的可执行文件名和命令行创建进程，并指明它是一个子进程

BOOL bCreateOK=CreateProcess(

szFilename, // 产生的应用程序的名称 (本EXE文件)

szCmdLine, // 告诉我们这是一个子进程的标志

NULL, // 用于进程的缺省的安全性

NULL, // 用于线程的缺省安全性

FALSE, // 不继承句柄

CREATE\_NEW\_CONSOLE, //创建新窗口

NULL, // 新环境

NULL, // 当前目录

&si, // 启动信息结构

&pi ) ; // 返回的进程信息

// 释放指向子进程的引用

if (bCreateOK) {

CloseHandle(pi.hProcess) ;

CloseHandle(pi.hThread) ;

}

}

void Parent() {

// 创建“自杀”互斥程序体

HANDLE hMutexSuicide=CreateMutex(

NULL, // 缺省的安全性

TRUE, // 最初拥有的

g\_szMutexName) ; // 互斥体名称

if (hMutexSuicide != NULL) {

// 创建子进程

std :: cout << "Creating the child process." << std :: endl;

StartClone() ;

// 指令子进程“杀”掉自身

std :: cout << "Telling the child process to quit. "<< std :: endl;

//等待父进程的键盘响应

getchar() ;

//释放互斥体的所有权，这个信号会发送给子进程的WaitForSingleObject过程

ReleaseMutex(hMutexSuicide) ;

// 消除句柄

CloseHandle(hMutexSuicide) ;

}

}

void Child() {

// 打开“自杀”互斥体

HANDLE hMutexSuicide = OpenMutex(

SYNCHRONIZE, // 打开用于同步

FALSE, // 不需要向下传递

g\_szMutexName) ; // 名称

if (hMutexSuicide != NULL) {

// 报告我们正在等待指令

std :: cout <<"Child waiting for suicide instructions. " << std :: endl;

//子进程进入阻塞状态，等待父进程通过互斥体发来的信号

WaitForSingleObject(hMutexSuicide, INFINITE) ;

//实验2-2步骤5：将上句改为WaitForSingleObject(hMutexSuicide, 0) ，重新编译执行

// 准备好终止，清除句柄

std :: cout << "Child quiting." << std :: endl;

CloseHandle(hMutexSuicide) ;

}

}

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

{

// 决定其行为是父进程还是子进程

if (argc>1 && :: strcmp(argv[1] , "child" )== 0) {

Child() ;

} else {

Parent() ;

}

return 0;

}

**实验三 进程同步的经典算法**

**1、实验目的**

1) 回顾系统进程、线程的有关概念，加深对Windows 2000线程的理解。

2) 了解互斥体对象，通过对生产者消费者等进程间同步与互斥经典算法的实现，加深对P、V原语以及利用P、V原语进行进程间同步与互斥操作的理解。

**2、程序清单**

清单3-1 生产者消费者问题

#include <windows.h>

#include <iostream>

const unsigned short SIZE\_OF\_BUFFER = 2; //缓冲区长度

unsigned short ProductID = 0; //产品号

unsigned short ConsumeID = 0; //将被消耗的产品号

unsigned short in = 0; //产品进缓冲区时的缓冲区下标

unsigned short out = 0; //产品出缓冲区时的缓冲区下标

int buffer[SIZE\_OF\_BUFFER]; //缓冲区是个循环队列

bool p\_ccontinue = true; //控制程序结束

HANDLE Mutex; //用于线程间的互斥

HANDLE FullSemaphore; //当缓冲区满时迫使生产者等待

HANDLE EmptySemaphore; //当缓冲区空时迫使消费者等待

DWORD WINAPI Producer(LPVOID); //生产者线程

DWORD WINAPI Consumer(LPVOID); //消费者线程

int main()

{

//创建各个互斥信号

//注意，互斥信号量和同步信号量的定义方法不同，互斥信号量调用的是CreateMutex函数，同步信号量

//调用的是CreateSemaphore函数，函数的返回值都是句柄。

Mutex = CreateMutex(NULL,FALSE,NULL);

EmptySemaphore = CreateSemaphore(NULL,SIZE\_OF\_BUFFER,SIZE\_OF\_BUFFER,NULL);

//将上句做如下修改，看看结果会怎样

//EmptySemaphore = CreateSemaphore(NULL,0,SIZE\_OF\_BUFFER-1,NULL);

FullSemaphore = CreateSemaphore(NULL,0,SIZE\_OF\_BUFFER,NULL);

//调整下面的数值，可以发现，当生产者个数多于消费者个数时，

//生产速度快，生产者经常等待消费者；反之，消费者经常等待

const unsigned short PRODUCERS\_COUNT = 3; //生产者的个数

const unsigned short CONSUMERS\_COUNT = 1; //消费者的个数

//总的线程数

const unsigned short THREADS\_COUNT = PRODUCERS\_COUNT+CONSUMERS\_COUNT;

HANDLE hThreads[THREADS\_COUNT]; //各线程的handle

DWORD producerID[PRODUCERS\_COUNT]; //生产者线程的标识符

DWORD consumerID[CONSUMERS\_COUNT]; //消费者线程的标识符

//创建生产者线程

for (int i=0;i<PRODUCERS\_COUNT;++i){

hThreads[i]=CreateThread(NULL,0,Producer,NULL,0,&producerID[i]);

if (hThreads[i]==NULL) return -1;

}

//创建消费者线程

for (i=0;i<CONSUMERS\_COUNT;++i){

hThreads[PRODUCERS\_COUNT+i]=CreateThread(NULL,0,Consumer,NULL,0,&consumerID[i]);

if (hThreads[i]==NULL) return -1;

}

while(p\_ccontinue){

if(getchar()){ //按回车后终止程序运行

p\_ccontinue = false;

}

}

return 0;

}

//生产一个产品。简单模拟了一下，仅输出新产品的ID号

void Produce()

{

std::cout << std::endl<< "Producing " << ++ProductID << " ... ";

std::cout << "Succeed" << std::endl;

}

//把新生产的产品放入缓冲区

void Append()

{

std::cerr << "Appending a product ... ";

buffer[in] = ProductID;

in = (in+1)%SIZE\_OF\_BUFFER;

std::cerr << "Succeed" << std::endl;

//输出缓冲区当前的状态

for (int i=0;i<SIZE\_OF\_BUFFER;++i){

std::cout << i <<": " << buffer[i];

if (i==in) std::cout << " <-- 生产";

if (i==out) std::cout << " <-- 消费";

std::cout << std::endl;

}

}

//从缓冲区中取出一个产品

void Take()

{

std::cerr << "Taking a product ... ";

ConsumeID = buffer[out];

buffer[out] = 0;

out = (out+1)%SIZE\_OF\_BUFFER;

std::cerr << "Succeed" << std::endl;

//输出缓冲区当前的状态

for (int i=0;i<SIZE\_OF\_BUFFER;++i){

std::cout << i <<": " << buffer[i];

if (i==in) std::cout << " <-- 生产";

if (i==out) std::cout << " <-- 消费";

std::cout << std::endl;

}

}

//消耗一个产品

void Consume()

{

std::cout << "Consuming " << ConsumeID << " ... ";

std::cout << "Succeed" << std::endl;

}

//生产者

DWORD WINAPI Producer(LPVOID lpPara)

{

while(p\_ccontinue){

WaitForSingleObject(EmptySemaphore,INFINITE); //p(empty);

WaitForSingleObject(Mutex,INFINITE); //p(mutex);

Produce();

Append();

Sleep(1500);

ReleaseMutex(Mutex); //V(mutex);

ReleaseSemaphore(FullSemaphore,1,NULL); //V(full);

}

return 0;

}

//消费者

DWORD WINAPI Consumer(LPVOID lpPara)

{

while(p\_ccontinue){

WaitForSingleObject(FullSemaphore,INFINITE); //P(full);

WaitForSingleObject(Mutex,INFINITE); //P(mutex);

Take();

Consume();

Sleep(1500);

ReleaseMutex(Mutex); //V(mutex);

ReleaseSemaphore(EmptySemaphore,1,NULL); //V(empty);

}

return 0;

}

**实验四 存储管理**

参考程序

#include <stdlib.h>

#include <stdio.h>

//输出内存块页面序号

int OutputBlockofMemory(int \*BlockofMemory, int BlockCount, int ReplacePage, int PageNum) {

int i;

printf("访问页面 %d 后，", PageNum);

printf("内存中的页面号为:\t");

for(i = 0; i < BlockCount; i++) {

if(BlockofMemory[i] < 0)

printf("# ");

else

printf("%d ", BlockofMemory[i]);

}

if(ReplacePage != -1)

printf("\t淘汰页面号为:%d", ReplacePage);

printf("\n");

return -1;

}

//输出页面引用串号

void OutputPageNumofRef(int\* PageNumofRef, int PageNumRefCount) {

int i = 0;

printf("页面引用串为:\t");

for(i = 0; i < PageNumRefCount; i++)

printf("%d ", PageNumofRef[i]);

printf("\n");

}

//内存块页面号清零

void ResetBlockofMemory(int \*BlockofMemory, int BlockCount) {

int i;

for(i = 0; i < BlockCount; i++)

BlockofMemory[i] = -1;

}

//判断页是否在内存中，如果页在内存中，返回1，否则返回0；

int PageInBlockofMemory(int PageNum, int \*BlockofMemory, int BlockCount) {

int i;

for(i = 0; i < BlockCount; i++)

if(PageNum == BlockofMemory[i])

return 1;

return 0;

}

//下次访问次序

//参数j: 页面在内存块中的位置

//参数i： 页面号在页面号引用串中的位置

int DistanceOpt(int \*BlockofMemory, int \*PageNumofRef, int j, int i, int PageNumRefCount) {

int k;

for(k = i + 1; k < PageNumRefCount; k++)

if(BlockofMemory[j] == PageNumofRef[k])

return k;

return PageNumRefCount;

}

//最佳页面置换算法

void Opt(int \*BlockofMemory, int \*PageNumofRef, int BlockCount, int PageNumRefCount) {

int i, j, k;

int MaxIndex1, MaxIndex2;

int MissCount = 0;

int ReplacePage;

int EmptyBlockCount = BlockCount;

printf("\*\*\*\*\*\*\*\*\*\*最佳页面置换算法：\*\*\*\*\*\*\*\*\*\*\n");

//输出页面引用串号

OutputPageNumofRef(PageNumofRef, PageNumRefCount);

for(i = 0; i < PageNumRefCount; i++) {

if(!PageInBlockofMemory(PageNumofRef[i], BlockofMemory, BlockCount)) {//页不在内存中

MissCount++;

if(EmptyBlockCount > 0) {

BlockofMemory[BlockCount - EmptyBlockCount] = PageNumofRef[i];

OutputBlockofMemory(BlockofMemory, BlockCount, -1, PageNumofRef[i]);

EmptyBlockCount--;

} else {

MaxIndex1 = MaxIndex2 = 0;

//求出未来最长时间不被访问的页

for(j = 0; j < BlockCount; j++) {

MaxIndex2 = DistanceOpt(BlockofMemory, PageNumofRef, j, i, PageNumRefCount);

if(MaxIndex1 < MaxIndex2) {

MaxIndex1 = MaxIndex2;

k = j;

}

}

ReplacePage = BlockofMemory[k];

BlockofMemory[k] = PageNumofRef[i];

OutputBlockofMemory(BlockofMemory, BlockCount, ReplacePage, PageNumofRef[i]);

}

} else {

OutputBlockofMemory(BlockofMemory,BlockCount, -1, PageNumofRef[i]);

}

}

printf("缺页次数为: %d\n", MissCount);

printf("OPT缺页率为: %.3f\n", (float)MissCount / PageNumRefCount);

}

//先进先出页面置换算法

void Fifo(int \*BlockofMemory,int \*PageNumofRef,int BlockCount,int PageNumRefCount) {

int i;

int ReplacePage;

int ReplaceIndex = 0;

int MissCount = 0;

int EmptyBlockCount = BlockCount;

printf("\*\*\*\*\*\*\*\*\*\*先进先出页面置换算法：\*\*\*\*\*\*\*\*\*\*\n");

//输出页面引用串号

OutputPageNumofRef(PageNumofRef,PageNumRefCount);

for(i = 0; i < PageNumRefCount; i++) {

if(!PageInBlockofMemory(PageNumofRef[i], BlockofMemory, BlockCount)) {//页不在内存中

MissCount++;

if(EmptyBlockCount > 0) {

BlockofMemory[BlockCount - EmptyBlockCount] = PageNumofRef[i];

OutputBlockofMemory(BlockofMemory, BlockCount, -1, PageNumofRef[i]);

EmptyBlockCount--;

} else {

ReplacePage = BlockofMemory[ReplaceIndex];

BlockofMemory[ReplaceIndex] = PageNumofRef[i];

ReplaceIndex = (ReplaceIndex + 1) % BlockCount;

OutputBlockofMemory(BlockofMemory, BlockCount, ReplacePage, PageNumofRef[i]);

}

} else

OutputBlockofMemory(BlockofMemory, BlockCount, -1, PageNumofRef[i]);

}

printf("缺页次数为：%d\n", MissCount);

printf("FIFO缺页率为：%.3f\n", (float)MissCount / PageNumRefCount);

}

//最近最久未使用页面置换算法

void Lru(int \*BlockofMemory, int \*PageNumofRef, int BlockCount, int PageNumRefCount) {

int i;

int MissCount = 0;

printf("\*\*\*\*\*\*\*\*\*\*\*\*最近最久未使用页面置换算法：\*\*\*\*\*\*\*\*\*\*\*\*\n");

//输出页面引用串号

OutputPageNumofRef(PageNumofRef, PageNumRefCount);

for(i = 0; i < PageNumRefCount; i++) {

if(!PageInBlockofMemory(PageNumofRef[i], BlockofMemory, BlockCount)) {//页不在内存中

MissCount++;

//

// 在此添加代码，实现 LRU 页面置换算法

//

}

else

OutputBlockofMemory(BlockofMemory, BlockCount, -1, PageNumofRef[i]);

}

printf("缺页次数为：%d\n", MissCount);

printf("LRU缺页率为：%.3f\n", (float)MissCount / PageNumRefCount);

}

int main() {

int \*BlockofMemory; //内存物理块

const int BlockCount = 5;

int PageNumofRef[] = {7,0,1,2,0,3,0,4,2,3,0,1,1,7,0,1,0,3}; //页面号引用串

int PageNumRefCount = sizeof(PageNumofRef) / sizeof(PageNumofRef[0]);

BlockofMemory = (int\*)malloc(BlockCount \* sizeof(int));

if(BlockofMemory == (int\*)NULL) {

printf("内存分配出错\n");

exit(1);

}

ResetBlockofMemory(BlockofMemory, BlockCount);

Opt(BlockofMemory, PageNumofRef, BlockCount, PageNumRefCount);

ResetBlockofMemory(BlockofMemory,BlockCount);

Fifo(BlockofMemory, PageNumofRef, BlockCount, PageNumRefCount);

ResetBlockofMemory(BlockofMemory,BlockCount);

Lru(BlockofMemory, PageNumofRef, BlockCount, PageNumRefCount);

free(BlockofMemory);

return 0;

}

**试验五 文件系统试验**

**1、试验目的**

通过设计一个基于索引结构的文件系统，加深对文件系统的基本知识理解。了解文件系统设计的基本概念。

1) 熟悉文件系统的物理结构；

2) 熟悉文件系统的目录管理；

3) 掌握文件系统空闲空间管理的基本方法；

4) 进一步理解现代操作系统文件管理知识。

**参考程序**

#include "stdio.h"

#include "stdlib.h"

//文件索引表的定义

struct index{

int lr[32];

int pr[32];

char st[32];

}\*wq;

#define JOBN 20

//文件目录的定义

struct list{

char names[32];

int size[32];

struct index\*p[32];//文件的索引表地址

}\*HEAD;

//作业序列

struct que{

char name;

int size;

}job[JOBN];

int i,j,ly,li;

char bb;

int NFile=0;//系统的总文件数，模拟开始时为0；

int N=0;//工作变量，标记当前分配文件

int M=32;//系统中空闲磁盘物理块数，开始为32块

int J[4][8];//用位图表示这些磁盘物理块，J[j][i]为0时标记第j\*8+i块空闲，为1标记该块已分配出去

FILE \*e;//记录模拟中的相关数据

int jobs=0;//作业数

void run(){

int x1,y,z;

//如果当前空闲磁盘物理块数能够满足需要，则进行分配

if(job[N].size<=M) {

ly=0;

N=NFile;//标记当前分配文件

NFile++;//标记下次分配处理文件

M-=job[N].size;

HEAD->names[N]=job[N].name;

//将文件名以及大小，索引地址填写到文件目录

HEAD->size[N]=job[N].size;

wq=(struct index \*)malloc(sizeof(struct index));

HEAD->p[N]=wq;

//对分配文件的索引表初始化

for(z=0;z<32;z++){

wq->lr[z]=z;//逻辑块号

wq->pr[z]=0;////物理块地址

wq->st[z]='N';

}

//从位图中分配，分配出去的块其状态为1。一直到分配完成

for(j=0;j<4&&(ly<job[N].size);j++)

for(i=0;i<8&&(ly<job[N].size);i++) {

if(J[j][i]==0){li=j\*8+i;wq->pr[ly]=li;wq->st[ly]='Y';ly++;J[j][i]=1;}//找到一个空闲块，就分配出去

}

}

//如果当前空闲磁盘物理块数不能够满足需要，则出错处理

else{

fprintf(e,"\n There are no free blocks in the memory now! \n");

fprintf(e,"File %c must wait!\n",job[N].name);

}

//报告目前为止的分配情况，首先报告文件目录的部分信息

fprintf(e,"...This time ,the file directory: ----\n");

fprintf(e,"NAME INDEX\_ADDRESS\n");

for(x1=0;x1<N+1;x1++)

fprintf(e," %c %x\n",HEAD->names[x1],HEAD->p[x1]);

//其次报告文件索引表的部分信息

for(x1=0;x1<N+1;x1++){

fprintf(e," //////////The index of FILE%c://///////\n",HEAD->names[x1]);

fprintf(e," LOGIC\_NUMBER PHYSICAL\_NUMBER FLAG\n");

for(y=0;y<HEAD->size[x1];y++)

fprintf(e," %d %d %c\n",HEAD->p[x1]->lr[y],HEAD->p[x1]->pr[y],HEAD->p[x1]->st[y]);

}

//第三，报告位图信息

fprintf(e," This time the bit mapping graph: \n");

for(j=0;j<4;j++){

fprintf(e, " ");

for(i=0;i<8;i++)fprintf(e,"%d",J[j][i]);

fprintf(e,"\n");

}

}

void main(){

int k;

e=fopen("results.txt","w");//打开保存结果的文件

for(j=0;j<4;j++)

for(i=0;i<8;i++)

J[j][i]=0;//初始化位图

HEAD=(struct list\*)malloc(sizeof(struct list));

for(i=0;i<32;i++){

HEAD->names[i]=' ';

HEAD->size[i]=0;

HEAD->p[i]=NULL;

}//初始化文件目录

printf("Please input number of jobs:");

scanf("%d",&jobs);

scanf("%c",&bb);

int kk=0;

//以下输入建立的文件数以及文件名字，并将这些信息保存在job数组之中

while(kk<jobs){

fprintf(e,"FILE %d: \n",kk);

printf("FILE %d: \n",kk);

printf("Name and Size");

scanf("%c,%d",&(job[kk].name),&(job[kk].size));

scanf("%c",&bb);

fprintf(e,"%c,%d",job[kk].name,job[kk].size);

kk++;

}

for(k=1;k<=jobs;k++)

run();//每个文件进行一次分配

//回收资源

fclose(e);

for(i=0;i<32;i++){

free(HEAD->p[i]);

HEAD->p[i]=NULL;

}

free(HEAD);

HEAD=NULL;

}

**附录B：文件系统模拟程序**

1．索引结构和文件目录

1

2

…

…

N

文件名

**等**

块

块

块

2．文件目录模拟

struct list{

char names[32];

int size[32];

struct index\*p[32];//文件的索引表地址

}\*HEAD;

该模拟文件最多只能模拟32个文件,HEAD 指针指向了该模拟目录。

3．文件索引表模拟

文件索引表的定义

struct index{

int lr[32];

int pr[32];

char st[32];}\*wq;

4．位图

int J[4][8]。位图J表示这些磁盘物理块的情况。J[j][i]为0时标记块空闲，为1标记该块已分配出去。各块号按行存储。因此，J[j][i] 表示j\*8+i(块号)块的分配情况

5.run函数



6．Main函数



附录B ：Visual studio 2008中建项目、运行 C++程序的方法

一、建项目

1. 文件——〉新建——〉项目——〉Visual C++——〉Win32控制台应用程序——〉输入项目名称——〉控制台应用程序，空项目——〉完成。

2. 右键单击新建项目名称——〉从右键菜单中选择“添加”——〉新建项——〉C++文件（.cpp)

3. 键入程序并执行即可。

在Visual studio 2008中执行C++程序:

二、运行 C++程序

不管用什么方法,注意C++中数据类型及其转换。注意Visual studio 2008中项目的默认字符编码是否与VC的一致

方法一

第1步与上相同;

第2步键入程序时,用

#include "iostream"

using namespace std;

代替#include <iostream.h>

方法二

1. 新建一CLR控制台应用程序，步骤与上类似；

2. 在键入源程序时，用如下代码替换#include <iostream.h>

#include "stdafx.h"

#include "iostream"

using namespace std;

3. 例如:

#include "stdafx.h"

#include "iostream"

using namespace std;

int main()

{

long l;

double d;

int a;

float b;

short c;

cout<<"size of char:"<<sizeof(char)<<endl;

cout<<"size of long:"<<sizeof(l)<<endl;

cout<<"size of double:"<<sizeof(d)<<endl;

cout<<"size of float:"<<sizeof(b)<<endl;

cout<<"size of int:"<<sizeof(a)<<endl;

cout<<"size of short:"<<sizeof(c)<<endl;

return 0;

}

4. 调试——>开始执行,程序成功运行；

说明：经查看Visual studio的安装目录，发现已无iostream.h，代之以文本文件iostream，如果还有其它库的变化，类似上述处理即可。