**《操作系统原理》**

**实验报告**

班级:

姓名:

学号:

指导老师：

**目录:**

[实验题目：实验一 线程创建与撤销 2](#_Toc453876314)

[实验题目：实验二 线程同步 6](#_Toc453876315)

[实验题目：实验三 线程互斥 11](#_Toc453876316)

[实验题目：实验四 进程通信 17](#_Toc453876317)

[实验题目：实验五 读者-写者问题 21](#_Toc453876318)

[实验题目：实验六 进程调度 37](#_Toc453876319)

[实验题目：实验七 存储管理之动态链接库 52](#_Toc453876320)

[实验题目：实验八 存储管理之内存分配 56](#_Toc453876321)

[实验题目：实验九 存储管理之页面置换算法 69](#_Toc453876322)

[实验题目：实验十 设备管理 84](#_Toc453876323)

[实验题目：实验十一 文件管理之文件读写 98](#_Toc453876324)

# 实验题目：实验一 线程创建与撤销

**完成人：XXX**

**报告日期：**2018年3月31日

1. 实验内容简要描述

（1）熟悉VC++、Visual Studio开发环境。

（2）使用相关函数创建和撤销线程。

（3）在一个进程中创建3个线程，名字分别为threada、threadb、threadc。

threada输出“hello world! ”。

threadb输出“My name is …”。

threadc输出“Please wait…”，然后sleep 5秒钟，接着输出“I wake up”。

1. 程序设计
2. 设计思路

该函数创建一个在调用进程的地址空间中执行的线程。

1. 主要数据结构

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes,

DWORD dwStackSize,

LPTHREAD\_START\_ROUTINE lpStartAddress,

LPVOID lpParameter,

DWORD dwCreationFlags,

LPDWORD lpThreadId

)；

VOID ExitThread(DWORD dwExitCode)；

VOID Sleep(DWORD dwMilliseconds);

VOID Sleep(DWORD dwMilliseconds);

三、实验结果

1、基本数据

lpThreadAttributes：指向一个SECURITY\_ATTRIBUTES结构，该结构决定了返回的句柄是否可被子进程继承。若lpThreadAttributes为NULL，则句柄不能被继承。

在WindowsNT中该结构的lpSecurityDescriptor成员定义了新进程的安全性描述符。若lpThreadAttributes为NULL，则线程获得一个默认的安全性描述符。

dwStackSize：定义原始堆栈提交时的大小(按字节计)。系统将该值舍入为最近的页。若该值为0，或小于默认时提交的大小，默认情况是使用与调用线程同样的大小。更多的信息，请看ThreadStackSize。

lpStartAddress：指向一个LPTHREAD\_START\_ROUTlNE类型的应用定义的函数，该线程执行此函数。该指针还表示溃程进程中线程的起始地址。该函数必须存在于远程进程中。

lpParameter：定义一个传递给该迸程的32位值。

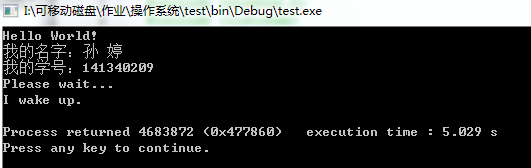
dwCreationFIags：定义控制进程创建的附加标志。若定义了CREATE\_SUSPENDED标志，线程创建时处于挂起状态，并且直到ResumeThread函数调用时d能运行。若该值为0，则该线程在创建后立即执行。

lpThreadId：指向一个32位值，它接收该线程的标识符。

2.

|  |  |  |
| --- | --- | --- |
| 源程序代码行数 | 完成该实验投入的时间（小时数） | 与其他同学讨论次数 |
| 31 | 1 | 1 |

1. 测试结果分析



四、实验体会

1、实验体会和收获

深入理解了线程与进程的概念，熟悉了在Windows环境下何时使用进程，何时使用线程，怎么创建和撤销线程。

五、源代码

#include <windows.h>

#include <iostream>

using namespace std;

DWORD WINAPI ta(LPVOID argv)

{

cout<<"Hello World!\n";

}

DWORD WINAPI tb(LPVOID argv)

{

cout<<"我的名字：孙 婷 \n";

cout<<"我的学号：141340209\n";

}

DWORD WINAPI tc(LPVOID argv)

{

cout<<"Please wait...\n";

Sleep(5000);

cout<<"I wake up.\n";

}

int main()

{

HANDLE threada,threadb,threadc;

DWORD TEST;

threada=CreateThread(NULL,0,ta,NULL,0,0);

threadb=CreateThread(0,0,tb,0,0,0);

threadc=CreateThread(0,0,tc,0,0,0);

ExitThread(TEST);

ExitThread(TEST);

ExitThread(TEST);

return 0;

}

# 实验题目：实验二 线程同步

**完成人：**XXX

**报告日期：**2018年4月7日

1. 实验内容简要描述
2. 在程序中使用CreateSemaphore(NULL,0,1,”SemaphoreName1”)创建一个名为SemaphoreName1的信号量，其初值为0。
3. 使用OpenSemaphore (SYNCHRONIZE| SEMAPHORE\_\_MODIFY\_STA

TE, NULL,” SemaphoreName1”)打开该信号量。

1. 创建一个子线程，主线程创建子线程后调WaitForSingleObject(hHandle,

INFINITE)，这里等待时间设置为INFINITE表示要一直等待下去，直到该信号量被唤醒为止。

1. 子线程sleep 5秒钟，然后输出“I am over.”结束，调用ReleaseSemaphore(hHandle1,1,NULL)释放信号量，使信号量的值加1。
2. 程序设计
3. 设计思路

A）等待一个对象

WaitForSingleObjects函数决定等待条件是否被满足。如果等待条件并没有被满足，调用线程进人一个高效的等待状态，当等待满足条件时占用非常少的处理器时间。

在运行前，一个等待函数修改同步对象类型的状态。修改仅发生在引起函数返回的对象身上。

例如，信号的计数减l。

WaitForSingleObject函数能等待的对象包括：Change notification(改变通告);Console input(控制台输入);Event(事件);Job(作业);Mutex(互斥对象);Process(进程);Semaphore(信号量);Thread(线程);Waitable timer(可等待定时器)。

当使用等待函数或代码直接或间接创建窗口时，一定要小心。如果一个线程创建了任何窗口，它必须处理进程消息。消息广播被发送到系统的所有窗口。一个线程用没有超时的等待函数也许会引起系统死锁。间接创建窗口的两个例子是DDE和COM CoInitialize。因此，如果用户有一个创建窗口的线程，用MsgWaitForMultipleObjects或MsgWaitForMultipleObjectsEx函数，而不要用

SignalObjectAndWait函数。

B）等待多个对象

WaiForMultipleObjects函数当下列条件之一满足时返回：(1)任意一个或全部指定对象处于信号态；(2)超时间隔已过。

C）创建信号量

如果成功就传回一个handle，否则传回NULL。不论哪一种情况，GetLastError都会传回一个合理的结果。如果指定的Semaphore名称已经存在，则函数还是成功的，GetLastError会传回ERROR\_ ALREADY\_EXISTS。

D）打开信号量

为现有的一个已命名信号机对象创建一个新句柄。

E）增加信号量的值

该函数将指定信号对象的计数增加一个指定的值。

1. 主要数据结构

DWORD WaitForSingleObject(HANDLE hHandle，DWORD dwMilliseconds);

DWORD WaitForMultipleObjects(

DWORD nCount，

CONST HANDLE \*lpHandles，

BOOL fWaitAll，

DWORD dwMilliSeconds

)

HANDLE CreateSemaphore(

LPSECURITY\_ATTRIBUTES lpAttributes，

LONG lInitialCount，

LONG lMaximumCount，

LPCTSTR lpName

)；

HANDLE OpenSemaphore(

DWORD dwDesiredAccess， //访问标志

BOOL bInheritHandle, // 继承标志

LPCTSTR lpName // 信号量名

)；

BOOL ReleaseSemaphore(

HANDLE hSemaphore，

LONG lReleaseCount，

LPLONG lpPreviousCount

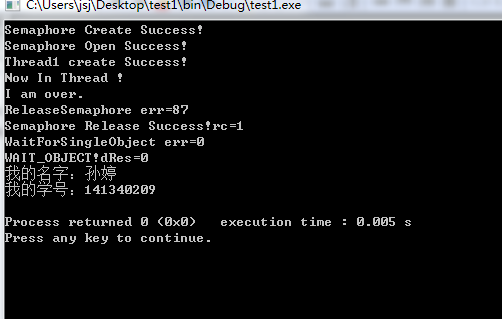
)

三、实验结果

1、基本数据

|  |  |  |
| --- | --- | --- |
| 源程序代码行数 | 完成该实验投入的时间（小时数） | 与其他同学讨论次数 |
| 61 | 2 | 1 |

2、测试结果分析



四、实验体会

1、实验体会和收获

进一步认识了线程同步的实质，学会使用信号量控制线程间的同步。

五、源代码

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

static HANDLE hThread1; //子进程的句柄，作为主线程的局部变量也行

static HANDLE hHandle1=NULL; //信号量的句柄，全局变量

void func(); //子线程的声明

int main(int argc,TCHAR\* argv[],TCHAR\* envp[])

{

int nRetCode=0;

DWORD dwThreadID1;

DWORD dRes,err;

hHandle1=CreateSemaphore(NULL,0,1,"SemaphoreName1"); //创建一个信号量

if(hHandle1==NULL) printf("Semaphore Create Fail!\n");

else printf("Semaphore Create Success!\n");

hHandle1=OpenSemaphore(SYNCHRONIZE|SEMAPHORE\_MODIFY\_STATE,NULL,"SemaphoreName1");

if(hHandle1==NULL)printf("Semaphore Open Fail!\n");

else printf("Semaphore Open Success!\n");

hThread1=CreateThread((LPSECURITY\_ATTRIBUTES)NULL, 0,

(LPTHREAD\_START\_ROUTINE)func,

(LPVOID)NULL,

0,&dwThreadID1); //创建子线程

if (hThread1==NULL) printf("Thread1 create Fail!\n");

else printf("Thread1 create Success!\n");

dRes=WaitForSingleObject(hHandle1,INFINITE); //主线程等待子线程结束

err=GetLastError();

printf("WaitForSingleObject err=%d\n",err);

if(dRes==WAIT\_TIMEOUT)

printf("TIMEOUT!dRes=%d\n",dRes);

else if(dRes==WAIT\_OBJECT\_0)

printf("WAIT\_OBJECT!dRes=%d\n",dRes);

else if(dRes==WAIT\_ABANDONED)

printf("WAIT\_ABANDONED!dRes=%d\n",dRes);

else printf("dRes=%d\n",dRes);

CloseHandle(hThread1);

CloseHandle(hHandle1);

printf("我的名字：孙婷\n");

printf("我的学号：141340209\n");

ExitThread(0);

return nRetCode;

}

//实现子线程

void func()

{

BOOL rc;

DWORD err;

printf("Now In Thread !\n");

printf("I am over.\n");

rc=ReleaseSemaphore(hHandle1,1,NULL); //子线程唤醒主线程

err=GetLastError();

printf("ReleaseSemaphore err=%d\n",err);

if(rc==0) printf("Semaphore Release Fail!\n");

else printf("Semaphore Release Success!rc=%d\n",rc);

}

# 实验题目：实验三 线程互斥

**完成人：**XXX

**报告日期：**2018年4月14日

1. 实验内容简要描述

完成两个子线程之间的互斥。在主线程中使用系统调用CreateThread（）创建两个子线程，并使两个子线程互斥的使用全局变量count。

1. 程序设计
2. 设计思路

a．使用临界区对象（Criticalsection）

Critical Section Object ，A segment of code that is not reentrant and therefore does not support concurrent access by multiple threads. Often, a critical section object is used to protect shared resources。通过定义在数据段中的一个CRITICAL\_SECTION 结构实现。

CRITICAL\_SECTION myCritical；

并且在任何线程使用此临界区对象之前必须对它进行初始化。

void InitializeCriticalSection(LPCRITICAL\_SECTION lpCriticalSection );

之后，任何线程访问临界区中数据的时候，必须首先调用EnterCriticalSection 函数，申请进入临界区（又叫关键代码段，使用共享资源的任何代码都必须封装在此）。在同一时间内，Windows 只允许一个线程进入临界区。所以在申请的时候，如果有另一个线程在临界区的话，EnterCriticalSection 函数会一直等待下去，直到其他线程离开临界区才返回。EnterCriticalSection 函数用法如下：

void EnterCriticalSection( LPCRITICAL\_SECTION lpCriticalSection);

当操作完成的时候，还要将临界区交还给Windows，以便其他线程可以申请使用。这个工作由LeaveCriticalSection 函数来完成。

void LeaveCriticalSection( LPCRITICAL\_SECTION lpCriticalSection);

当程序不再使用临界区对象的时候，必须使用DeleteCriticalSection 函数将它删除。

void DeleteCriticalSection( LPCRITICAL\_SECTION lpCriticalSection);

b．使用互斥锁（Interlocked）

提供一种手段来保证值的递增(减)能够以原子操作方式来进行，也就是不中断地进行。

LONG InterlockedIncrement( LPLONG lpAddend ) ; // 增一操作

LONG InterlockedDecrement( LPLONG lpAddend); // 减一操作

LONG InterlockedExchangeAdd (

PLONG Addend, // pointer to the addend

LONG Increment // increment value

); //增减任意值

1. 主要数据结构

void InitializeCriticalSection(LPCRITICAL\_SECTION lpCriticalSection );

void EnterCriticalSection( LPCRITICAL\_SECTION lpCriticalSection);

void LeaveCriticalSection( LPCRITICAL\_SECTION lpCriticalSection);

void DeleteCriticalSection( LPCRITICAL\_SECTION lpCriticalSection);

LONG InterlockedIncrement( LPLONG lpAddend ) ; // 增一操作

LONG InterlockedDecrement( LPLONG lpAddend); // 减一操作

LONG InterlockedExchangeAdd (

PLONG Addend, // pointer to the addend

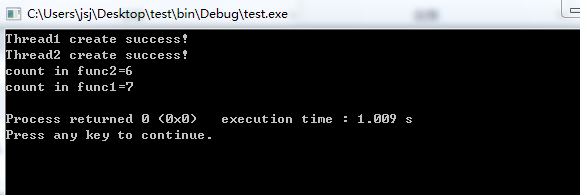
LONG Increment // increment value

三、实验结果

1、基本数据

|  |  |  |
| --- | --- | --- |
| 源程序代码行数 | 完成该实验投入的时间（小时数） | 与其他同学讨论次数 |
| 68 | 2 | 0 |

2.测试结果分析



四、实验体会

1、实验体会和收获

熟练了Windows系统环境下线程的创建与撤销，熟悉了Windows系统提供的线程互斥API，使用Windows系统提供的线程互斥API解决实际问题。

五、源代码

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

static int count=5; //共享变量

static HANDLE h1,h2; //两个子进程的句柄变量

LPCRITICAL\_SECTION hCriticalSection; //定义指向临界区对象的地址指针

CRITICAL\_SECTION Critical; //定义临界区

void func1( ); //线程函数的定义不符合WIN32格式，后面CreateThread函数中

void func2( ); //要附加强制类型转换

//主线程的实现

int main(int argc, TCHAR\* argv[],TCHAR\* envp[])

{

int nRetCode=0;

DWORD dwThreadID1, dwThreadID2;

hCriticalSection=&Critical; //将指向临界区的对象的指针指向临界区

InitializeCriticalSection(hCriticalSection); //初始化临界区

//创建子线程func1

h1=CreateThread((LPSECURITY\_ATTRIBUTES)NULL,

0,

(LPTHREAD\_START\_ROUTINE)func1,

(LPVOID)NULL,

0,&dwThreadID1);

if(h1==NULL)printf("Thread1 create Fail!\n");

else printf("Thread1 create success!\n");

//创建子线程func2

h2=CreateThread((LPSECURITY\_ATTRIBUTES)NULL,

0,

(LPTHREAD\_START\_ROUTINE)func2,

(LPVOID)NULL,

0,&dwThreadID2);

if(h2==NULL) printf("Thread2 create Fail!\n");

else printf("Thread2 create success!\n");

Sleep(1000);

CloseHandle(h1);

CloseHandle(h2);

DeleteCriticalSection(hCriticalSection); //删除临界区

ExitThread(0);

return nRetCode;

}//主线程结束

//子线程fun c2的实现

void func2()

{

int r2;

EnterCriticalSection(hCriticalSection); //进入临界区

r2=count;

Sleep(100);

r2=r2+1;

count=r2;

printf("count in func2=%d\n",count);

LeaveCriticalSection(hCriticalSection); //退出临界区

}

//子线程func1的实现

void func1()

{

int r1;

EnterCriticalSection(hCriticalSection); //进入临界区

r1=count;

Sleep(500);

r1=r1+1;

count=r1;

printf("count in func1=%d\n",count);

LeaveCriticalSection(hCriticalSection); //退出临界区

}

# 实验题目：实验四 进程通信

**完成人：**XXX

**报告日期：**2018年4月21日

1. 实验内容简要描述
2. 利用命名管道的相关知识及函数，分别编写服务器进程和客户端进程程序。
3. 要求服务器进程和客户端进程程序能够通过互相传送数据。
4. 当服务器进程和客户端进程中的任何一端输入“end”时，结束会话。
5. 程序设计
6. 设计思路

（1）建立命名管道

（2）连接命名管道

（3）拆除命名管道的连接

（4）客户端进程连接服务器已建立的命名管道

（5）客户端进程等待命名管道

1. 主要数据结构

Handle CreateNamedPipe(LPCTSTR lpName，……)；

BOOL ConnectNamedPipe(HANDLE hNamePipe，OVERLAPPED lpOverlapped)；

BOOL DisconnectNamedPipe (Long hNamedPipe)；

BOOL CallNamedPipe( String lpNamedPipeName，//欲打开管道的名称

lpInBuffer Any， //要写入管道的数据的内存缓冲区

nInBufferSize Long， //内存缓冲区中的字符数量

lpOutBuffer Any， //指定一个内存缓冲区，用于装载从管道中读出的数据

nOutBufferSize Long，//指定一个长整数变量，用于装载来自管道的数据

lpBytesRead Long， //指定从管道中读出的字节数

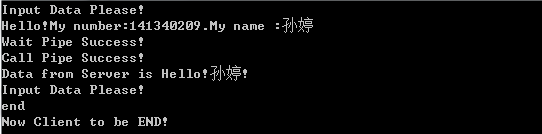
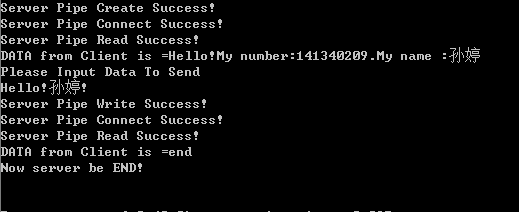
nTimeOut Long， //管道是否可用及超时设置

)；

BOOL WaitNamedPipe(LPCTSTR lpNamedPipeName, DWORD nTimeOut)；

三、实验结果

测试结果



四、实验体会

1、实验体会和收获

了解命名管道的概念，并学会了使用命名管道实现进程间的基本通信。

五、源代码

/服务器进程/

int \_tmain(int argc, TCHAR\* argv[], TCHAR\* envp[])

{

int nRetCode=0;

BOOL rc;

int err;

HANDLE hPipeHandle;

char lpName[]="\\\\.\\pipe\\myPipe";

char InBuffer[50]="";

char OutBuffer[50]="";

DWORD BytesRead,BytesWrite;

hPipeHandle=CreateNamedPipe(

(LPCTSTR)lpName,

PIPE\_ACCESS\_DUPLEX|FILE\_FLAG\_OVERLAPPED|WRITE\_DAC,

PIPE\_TYPE\_MESSAGE|PIPE\_READMODE\_BYTE|PIPE\_WAIT,

1,20,30,NMPWAIT\_USE\_DEFAULT\_WAIT,

(LPSECURITY\_ATTRIBUTES)NULL

) ; //创建命令管道

if((hPipeHandle==INVALID\_HANDLE\_VALUE)||(hPipeHandle==NULL))

{

err= GetLastError();

printf("Server Pipe Create Fail!err=%d\n",err);

exit(1);

}

else printf("Server Pipe Create Success!\n");

while(1) //连接命名管道

{

rc=ConnectNamedPipe(hPipeHandle,(LPOVERLAPPED)NULL);

if(rc==0)

{

err=GetLastError();

printf("Server Pipe Connect Fail!err=%d\n",err);

exit(2);

}

else printf("Server Pipe Connect Success!\n");

strcpy(InBuffer,"");

strcpy(OutBuffer,"");

//命名管道读数据

rc= ReadFile(hPipeHandle,InBuffer,sizeof(InBuffer),

&BytesRead,(LPOVERLAPPED)NULL);

if(rc==0 && BytesRead==0)

{

err=GetLastError();

printf("Server Pipe Read Fail!err=%d\n",err);

exit(3);

}

else printf("Server Pipe Read Success!\n DATA from Client is=%s\n",InBuffer);

rc=strcmp(InBuffer,"end");

if(rc==0) break;

printf("Please Input Data to Send\n");

scanf("%s",OutBuffer);

//向命令管道写数据

rc=WriteFile(hPipeHandle,OutBuffer,sizeof(OutBuffer),

&BytesWrite,(LPOVERLAPPED)NULL);

if(rc==0 )

{err=GetLastError();

printf("Server Pipe Write Fail!err=%d\n",err);}

else printf("Server Pipe Write Success!\n");

DisconnectNamedPipe(hPipeHandle); //拆除与管道命令连接

rc=strcmp(OutBuffer,"end");

if(rc==0) break;

}

printf("Now server be END!\n");

CloseHandle(hPipeHandle);

return nRetCode;

}

/客户端进程/

int \_tmain(int argc, TCHAR\* argv[], TCHAR\* envp[])

{

BOOL rc=0;

char lpName[]="\\\\.\\pipe\\myPipe";

char InBuffer[50]="";

char OutBuffer[50]="";

DWORD BytesRead;

int nRetCode = 0;

int err=0;

while(1)

{

strcpy(InBuffer,"");

strcpy(OutBuffer,"");

printf("Input Data Please!\n");

scanf("%s",InBuffer);

if(!strcmp(InBuffer,"end"))

{CallNamedPipe(lpName,InBuffer,sizeof(InBuffer),OutBuffer,sizeof(OutBuffer),

&BytesRead,NMPWAIT\_USE\_DEFAULT\_WAIT);

break;}

rc=WaitNamedPipe(lpName,NMPWAIT\_WAIT\_FOREVER); //等待命名管道

if(rc==0)

{

err=GetLastError();

printf("Wait Pipe Fali!err=%d\n",err);

exit(1);

}

else printf("Wait Pipe Success!\n");

//使用管道命令读数据

rc=CallNamedPipe(lpName,InBuffer,sizeof(InBuffer),OutBuffer,sizeof(OutBuffer), &BytesRead,NMPWAIT\_USE\_DEFAULT\_WAIT);

rc=strcmp(OutBuffer,"end");

if (rc==0) break;

if (rc==0)

{

err=GetLastError();

printf("Call Pipe Fali!err=%d\n",err);

exit(1);

}

else printf("Call Pipe Success!\n Data from Server is %s\n",OutBuffer);

}

printf("Now Client to be END!\n");

return nRetCode;

}

# 实验题目：实验五 读者-写者问题

**完成人：**XXX

**报告日期：**2018年4月28日

1. 实验内容简要描述
   1. 创建一个控制台进程，此进程包含n个线程。用这n个线程来表示n个读者或写者。每个线程按相应测试数据文件的要求进行读写操作。用信号量机制分别实现读者优先和写者优先的读者-写者问题。
   2. 读者-写者问题的读写操作限制(包括读者优先和写者优先)：

① 写-写互斥，即不能有两个写者同时进行写操作。

② 读-写互斥，即不能同时有一个线程在读，而另一个线程在写。

③ 读-读允许，即可以有一个或多个读者在读。

* 1. 读者优先的附加限制：如果一个读者申请进行读操作时已有另一个读者正在进行读操作，则该读者可直接开始读操作。
  2. 写者优先的附加限制：如果一个读者申请进行读操作时已有另一写者在等待访问共享资源，则该读者必须等到没有写者处于等待状态后才能开始读操作。
  3. 运行结果显示要求：要求在每个线程创建、发出读写操作申请、开始读写操作和结束读写操作时分别显示一行提示信息，以确定所有处理都遵守相应的读写操作限制。

1. 程序设计
2. 设计思路

将所有的读者和所有的写者分别放进两个等待队列中，当读允许时就让读者队列释放一个或多个读者，当写允许时，释放第一个写者操作。

**读者优先：**

如果没有写者正在操作，则读者不需要等待，用一个整型变量readcount记录当前的读者数目，用于确定是否释放写者线程，（当readcout=0 时，说明所有的读者都已经读完，释放一个写者线程），每个 读者开始读之前都要修改readcount,为了互斥的实现对readcount 的修改，需要一个互斥对象Mutex来实现互斥。

另外，为了实现写-写互斥，需要一个临界区对象 write,当写者发出写的请求时，必须先得到临界区对象的所有权。通过这种方法，可以实现读写互斥，当readcount=1 时，（即第一个读者的到来时，），读者线程也必须申请临界区对象的所有权.

当读者拥有临界区的所有权，写者都阻塞在临界区对象write上。当写者拥有临界区对象所有权时，第一个判断完readcount==1 后，其余的读者由于等待对readcount的判断，阻塞在Mutex上！

**写者优先：**

写者优先和读者优先有相同之处，不同的地方在：一旦有一个写者到来时，应该尽快让写者进行写，如果有一个写者在等待，则新到的读者操作不能读操作，为此添加一个整型变量writecount,记录写者的数目，当writecount=0时才可以释放读者进行读操作！

为了实现对全局变量writecount的互斥访问，设置了一个互斥对象Mutex3。

为了实现写者优先，设置一个临界区对象read,当有写者在写或等待时，读者必须阻塞在临界区对象read上。

读者除了要一个全局变量readcount实现操作上的互斥外，还需要一个互斥对象对阻塞在read这一个过程实现互斥，这两个互斥对象分别为mutex1和mutex2。

1. 主要数据结构

void RP\_ReaderThread(void\* p) 读者线程信息

void RP\_WriterThread(void\* p) 写者线程信息

void ReaderPriority(char\* file) 读者优先处理函数

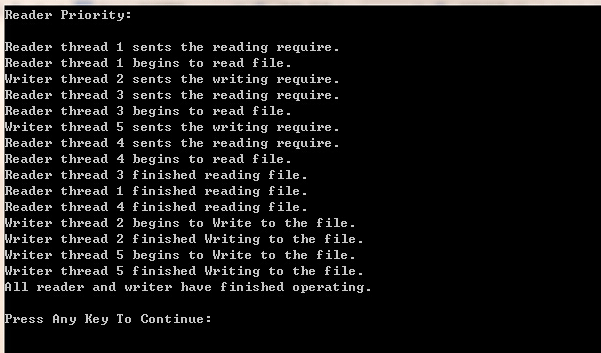
void WP\_ReaderThread(void\* p) 读者线程信息

void WriterPriority(char\* file) 写者优先处理函数

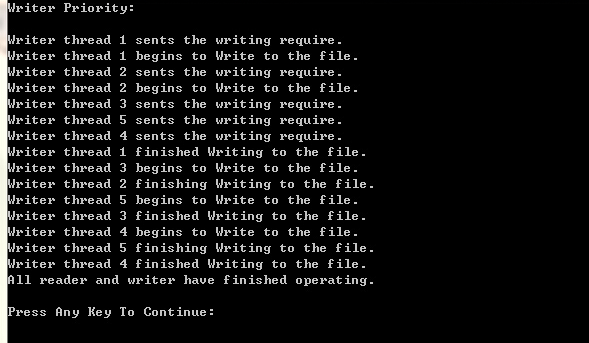
三、实验结果

1、测试结果分析

1. 读者优先运行



1. 写者优先运行



四、实验体会

1、实验体会和收获

Windows2000/XP提供了互斥量(mutex)、信号量(semapore)、事件(event)等三种同步对象和相应的系统调用，用于线程的互斥与同步。通过对读者写者问题的调试，了解Windows2000/XP中的同步机制。

五、源代码

#include "windows.h"

#include <conio.h>

#include <stdlib.h>

#include <fstream.h>

#include <io.h>

#include <string.h>

#include <stdio.h>

#define READER 'R' // 读者

#define WRITER 'W' // 写者

#define INTE\_PER\_SEC 1000 // 每秒时钟中断数目。

#define MAX\_THREAD\_NUM 64 // 最大线程数目

#define MAX\_FILE\_NUM 32 // 最大数据文件数目

#define MAX\_STR\_LEN 32 // 字符串长度

int readcount=0; // 读者数目

int writecount=0; // 写者数目

CRITICAL\_SECTION RP\_Write; //临界区

CRITICAL\_SECTION cs\_Write;

CRITICAL\_SECTION cs\_Read;

struct ThreadInfo

{

int serial; // 线程序号

char entity; //线程类别（判断读者线程还是写者线程）

double delay;

double persist;

};

///////////////////////////////////////////////////////////////////////////

// 读者优先--读者线程

//p：读者线程信息

void RP\_ReaderThread(void\* p)

{

//互斥变量

HANDLE h\_Mutex;

h\_Mutex=OpenMutex(MUTEX\_ALL\_ACCESS,FALSE,"mutex\_for\_readcount");

DWORD wait\_for\_mutex; //等待互斥变量所有权

DWORD m\_delay; // 延迟时间

DWORD m\_persist; // 读文件持续时间

int m\_serial; //线程序号

//从参数中获得信息

m\_serial=((ThreadInfo\*)(p))->serial;

m\_delay=(DWORD)(((ThreadInfo\*)(p))->delay\*INTE\_PER\_SEC);

m\_persist=(DWORD)(((ThreadInfo\*)(p))->persist\*INTE\_PER\_SEC);

Sleep(m\_delay); //延迟等待

printf("Reader thread %d sents the reading require.\n",m\_serial);

// 等待互斥信号，保证对readcount的访问、修改互斥

wait\_for\_mutex=WaitForSingleObject(h\_Mutex,-1);

//读者数目增加

Readcount++;

if(readcount==1)

{

//第一个读者，等待资源

EnterCriticalSection(&RP\_Write);

}

ReleaseMutex(h\_Mutex); //释放互斥信号

//读文件

printf("Reader thread %d begins to read file.\n",m\_serial);

Sleep(m\_persist);

// 退出线程

printf("Reader thread %d finished reading file.\n",m\_serial);

//等待互斥信号，保证对readcount的访问、修改互斥

wait\_for\_mutex=WaitForSingleObject(h\_Mutex,-1);

//读者数目减少

readcount--;

if(readcount==0)

{

//如果所有读者读完，唤醒写者

LeaveCriticalSection(&RP\_Write);

}

ReleaseMutex(h\_Mutex); //释放互斥信号

}

///////////////////////////////////////////////////////////////////////////

// 读者优先--写者线程

//p：写者线程信息

void RP\_WriterThread(void\* p)

{

DWORD m\_delay; // 延迟时间

DWORD m\_persist; // 写文件持续时间

int m\_serial; //线程序号

//从参数中获得信息

m\_serial=((ThreadInfo\*)(p))->serial;

m\_delay=(DWORD)(((ThreadInfo\*)(p))->delay\*INTE\_PER\_SEC);

m\_persist=(DWORD)(((ThreadInfo\*)(p)) ->persist\*INTE\_PER\_SEC);

Sleep(m\_delay); //延迟等待

printf("Writer thread %d sents the writing require.\n",m\_serial);

// 等待资源

EnterCriticalSection(&RP\_Write);

//写文件

printf("Writer thread %d begins to Write to the file.\n",m\_serial);

Sleep(m\_persist);

// 退出线程

printf("Writer thread %d finished Writing to the file.\n",m\_serial);

//释放资源

LeaveCriticalSection(&RP\_Write);

}

///////////////////////////////////////////////////////////////////////////

// 读者优先处理函数

//file：文件名

void ReaderPriority(char\* file)

{

DWORD n\_thread=0; //线程数目

DWORD thread\_ID; //线程ID

DWORD wait\_for\_all; //等待所有线程结束

//互斥对象

HANDLE h\_Mutex;

h\_Mutex=CreateMutex(NULL,FALSE,"mutex\_for\_readcount");

//线程对象的数组

HANDLE h\_Thread[MAX\_THREAD\_NUM];

ThreadInfo thread\_info[MAX\_THREAD\_NUM];

readcount=0; // 初始化 readcount

InitializeCriticalSection(&RP\_Write); //初始化临界区

ifstream inFile;

inFile.open(file); //打开文件

printf("Reader Priority:\n\n");

while(inFile)

{

//读入每一个读者、写者的信息

inFile>>thread\_info[n\_thread].serial;

inFile>>thread\_info[n\_thread].entity;

inFile>>thread\_info[n\_thread].delay;

inFile>>thread\_info[n\_thread++].persist;

inFile.get( );

}

for(int i=0;i< (int)(n\_thread);i++)

{

if(thread\_info[i].entity==READER || thread\_info[i].entity=='R')

{

h\_Thread[i]=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)(RP\_ReaderThread),&thread\_info[i],0,&thread\_ID); //创建读者线程

}

else{

//创建写者线程

h\_Thread[i]=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)(RP\_WriterThread),&thread\_info[i],0,&thread\_ID);

}

}

//等待所有线程结束

wait\_for\_all=WaitForMultipleObjects(n\_thread,h\_Thread,TRUE,-1);

printf("All reader and writer have finished operating.\n");

}

///////////////////////////////////////////////////////////////////////////

// 写者优先--读者线程

//p：读者线程信息

void WP\_ReaderThread(void\* p)

{

//互斥变量

HANDLE h\_Mutex1;

h\_Mutex1=OpenMutex(MUTEX\_ALL\_ACCESS,FALSE,"mutex1");

HANDLE h\_Mutex2;

h\_Mutex2=OpenMutex(MUTEX\_ALL\_ACCESS,FALSE,"mutex2");

DWORD wait\_for\_mutex1; //等待互斥变量所有权

DWORD wait\_for\_mutex2;

DWORD m\_delay; // 延迟时间

DWORD m\_persist; // 读文件持续时间

int m\_serial; //线程序号

//从参数中获得信息

m\_serial=((ThreadInfo\*)(p))->serial;

m\_delay=(DWORD)(((ThreadInfo\*)(p))->delay\*INTE\_PER\_SEC);

m\_persist=(DWORD)(((ThreadInfo\*)(p)) ->persist\*INTE\_PER\_SEC);

Sleep(m\_delay); //延迟等待

printf("Reader thread %d sents the reading require.\n",m\_serial);

wait\_for\_mutex1= WaitForSingleObject(h\_Mutex1,-1);

//进入读者临界区

EnterCriticalSection(&cs\_Read);

// 阻塞互斥对象mutex2，保证对readcount的访问、修改互斥

wait\_for\_mutex2= WaitForSingleObject(h\_Mutex2,-1);

//修改读者数目

readcount++;

if(readcount==1)

{

//如果是第一个读者，等待写者写完

EnterCriticalSection(&cs\_Write);

}

ReleaseMutex(h\_Mutex2); //释放互斥信号mutex2

// 让其他读者进入临界区

LeaveCriticalSection(&cs\_Write);

ReleaseMutex(h\_Mutex1);

//读文件

printf("Reader thread %d begins to read file.\n",m\_serial);

Sleep(m\_persist);

// 退出线程

printf("Reader thread %d finished reading file.\n",m\_serial);

// 阻塞互斥对象mutex2，保证对readcount的访问、修改互斥

wait\_for\_mutex2= WaitForSingleObject(h\_Mutex2,-1);

readcount--;

if(readcount==0)

{

// 最后一个读者，唤醒写者

LeaveCriticalSection(&cs\_Write);

}

ReleaseMutex(h\_Mutex2); //释放互斥信号

}

///////////////////////////////////////////////////////////////////////////

// 写者优先--写者线程

//p：写者线程信息

void WP\_WriterThread(void\* p)

{

DWORD m\_delay; // 延迟时间

DWORD m\_persist; // 写文件持续时间

int m\_serial; //线程序号

DWORD wait\_for\_mutex3;

//互斥对象

HANDLE h\_Mutex3;

h\_Mutex3= OpenMutex(MUTEX\_ALL\_ACCESS,FALSE,"mutex3");

//从参数中获得信息

m\_serial=((ThreadInfo\*)(p))->serial;

m\_delay=(DWORD)(((ThreadInfo\*)(p))->delay\*INTE\_PER\_SEC);

m\_persist=(DWORD)(((ThreadInfo\*)(p))->persist\*INTE\_PER\_SEC);

Sleep(m\_delay); //延迟等待

printf("Writer thread %d sents the writing require.\n",m\_serial);

// 阻塞互斥对象mutex3，保证对writecount的访问、修改互斥

wait\_for\_mutex3= WaitForSingleObject(h\_Mutex3,-1);

writecount++; //修改读者数目

if(writecount==1)

{

//第一个写者，等待读者读完

EnterCriticalSection(&cs\_Read);

}

ReleaseMutex(h\_Mutex3);

//进入写者临界区

EnterCriticalSection(&cs\_Write);

//写文件

printf("Writer thread %d begins to Write to the file.\n",m\_serial);

Sleep(m\_persist);

// 退出线程

printf("Writer thread %d finishing Writing to the file.\n",m\_serial);

//离开临界区

LeaveCriticalSection(&cs\_Write);

// 阻塞互斥对象mutex3，保证对writecount的访问、修改互斥

wait\_for\_mutex3= WaitForSingleObject(h\_Mutex3,-1);

writecount--; //修改读者数目

if(writecount==0)

{

//写者写完，读者可以读

LeaveCriticalSection(&cs\_Read);

}

ReleaseMutex(h\_Mutex3);

}

///////////////////////////////////////////////////////////////////////////

// 写者优先处理函数

//file：文件名

void WriterPriority(char\* file)

{

DWORD n\_thread=0; //线程数目

DWORD thread\_ID; //线程ID

DWORD wait\_for\_all; //等待所有线程结束

//互斥对象

HANDLE h\_Mutex1;

h\_Mutex1=CreateMutex(NULL,FALSE,"mutex1");

HANDLE h\_Mutex2;

h\_Mutex2=CreateMutex(NULL,FALSE,"mutex2");

HANDLE h\_Mutex3;

h\_Mutex3=CreateMutex(NULL,FALSE,"mutex3");

//线程对象

HANDLE h\_Thread[MAX\_THREAD\_NUM];

ThreadInfo thread\_info[MAX\_THREAD\_NUM];

readcount=0; // 初始化 readcount

writecount=0; // 初始化writecount

InitializeCriticalSection(&cs\_Write); //初始化临界区

InitializeCriticalSection(&cs\_Read);

ifstream inFile;

inFile.open(file); //打开文件

printf("Writer Priority:\n\n");

while(inFile)

{

//读入每一个读者、写者的信息

inFile>>thread\_info[n\_thread].serial;

inFile>>thread\_info[n\_thread].entity;

inFile>>thread\_info[n\_thread].delay;

inFile>>thread\_info[n\_thread++].persist;

inFile.get( );

}

for(int i=0;i< (int)(n\_thread);i++)

{

if (thread\_info[i].entity==READER || thread\_info[i].entity=='R')

{

//创建读者线程

h\_Thread[i]=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)(RP\_WriterThread),&thread\_info[i],0,&thread\_ID);

}

else{

//创建写者线程

h\_Thread[i]=CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)(WP\_WriterThread),&thread\_info[i],0,&thread\_ID);

}

}

//等待所有线程结束

wait\_for\_all=WaitForMultipleObjects(n\_thread,h\_Thread,TRUE,-1);

printf("All reader and writer have finished operating.\n");

}

///////////////////////////////////////////////////////////////////////////////

//主函数

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

{

char ch;

while (true)

{

//打印提示信息

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

printf(" 1:Reader Priority\n");

printf(" 2:Writer Priority\n");

printf(" 3:Exit Priority\n");

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

printf("Enter your choice(1,2 or 3): ");

//如果输入信息不正确，继续输入

do{

ch=(char)\_getch( );

}while(ch != '1' &&ch != '2' && ch != '3');

system("cls");

//选择3，返回

if(ch=='3')

return 0;

//选择1，读者优先

else if(ch=='1')

ReaderPriority("thread.dat");

//选择2，写者优先

else

WriterPriority("thread.dat");

//结束

printf("\nPress Any Key To Continue: ");

\_getch( );

system("cls");

}

return 0;

}

# 实验题目：实验六 进程调度

**完成人：**XXX

**报告日期：**2018年5月5日

1. 实验内容简要描述
   1. 设计进程控制块PCB表结构，分别适用于优先数调度算法和循环轮转调度算法。PCB结构通常包括以下信息：进程名，进程优先数（或轮转时间片），进程所占用的CPU时间，进程的状态，当前队列指针等。根据调度算法的不同，PCB结构的内容可以作适当的增删。
   2. 建立进程就绪队列。对两种不同算法编制入链子程序。
   3. 编制两种进程调度算法：①优先数调度；②循环轮转调度。
2. 程序设计
3. 设计思路

（1）设计就绪、执行、完成三个队列；根据输入进程的时间，按照需要的时间越多，优先级越低,（60减去所需时间）建立就绪队列。

（2）对优先数调度算法实现，首先从就绪队列中取出第一个进程。进程执行，优先数减3，CPU时间片数加1，进程还需要时间片数减1，在此设置计数器count，来判断进程是否执行完成。若进程完成，标志flag为0，并插入完成队列；否则插入就绪队列，等待下一次继续执行。重复上述操作，直到进程全部完成。

（3）对轮转法调度算法实现，从就绪队列取进程，在时间片数为2的时间内进程执行，计数器count加2，若进程完成，继续取下一个进程执行，否则，若时间片用完，计数器清零，将该进程排列到就绪队列的尾上。然后取下一个进程，由于计数器已经清零，故相当于又给了一个时间片。重复上述操作，直到进程全部完成。

1. 主要数据结构
2. PCB结构

typedef struct node

{

char name[20]; /\*进程的名字\*/

int prio; /\*进程的优先级\*/

int round; /\*分配CPU的时间片\*/

int cputime; /\*CPU执行时间\*/

int needtime; /\*进程执行所需要的时间\*/

char state; /\*进程的状态，W——就绪态，R——执行态，F——完成态\*/

int count; /\*记录执行的次数\*/

struct node \*next; /\*链表指针\*/

}PCB;

1. 三个应用队列： PCB \*ready=NULL,\*run=NULL,\*finish=NULL; /\*定义三个队列：就绪、执行和完成队列\*/
2. 函数声明部分：

void GetFirst(); /\*从就绪队列取得第一个节点\*/

void Output(); /\*输出队列信息\*/

void InsertPrio(PCB \*in); /\*创建优先级队列，规定优先数越小，优先级越高\*/

void InsertTime(PCB \*in); /\*时间片队列\*/

void InsertFinish(PCB \*in); /\*时间片队列\*/

void PrioCreate(); /\*优先级输入函数\*/

void TimeCreate(); /\*时间片输入函数\*/

void Priority(); /\*按照优先级调度\*/

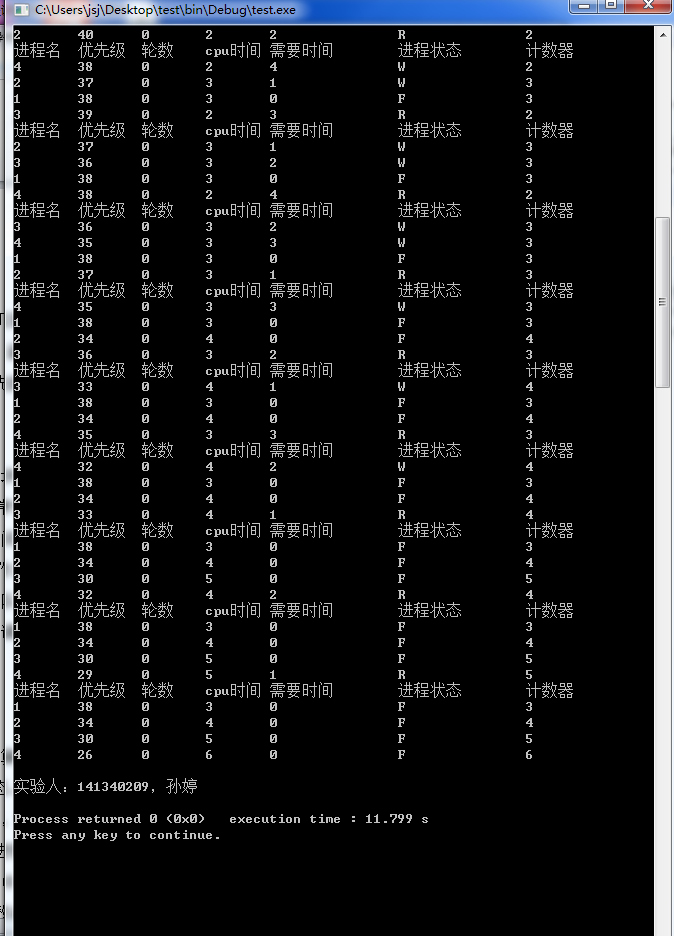
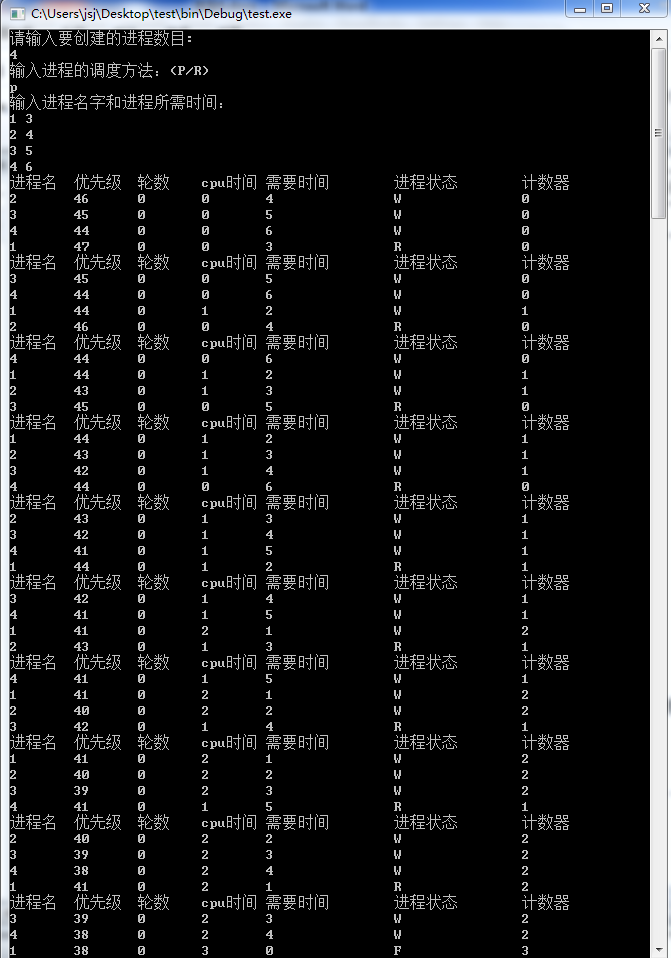
void RoundRun(); /\*时间片轮转调度\*/

三、实验结果

1、基本数据

|  |  |  |
| --- | --- | --- |
| 源程序代码行数 | 完成该实验投入的时间（小时数） | 与其他同学讨论次数 |
| 276 | 3 | 0 |

2.测试结果分析



四、实验体会

1、实验过程中遇到的问题及解决过程

(1)本次试验，思路设计不难，主要还是在利用指针处理时感觉很困难，实验中设计了结构指针用来指向PCB结构，PCB结构中又有链表指针。为此必须时时防止出现野指针，程序崩溃。

(2)在建立优先数就绪队列时主要运用，链表插入模型。但是由于本题是从建立、到完成一个就绪对列，所以必须分多种情况讨论。

2、实验体会和收获

（1）本次试验后对优先数调度算法和时间片轮转调度算法实现的过程，有了很清楚的认识、理解。设计计数器来对进程执行状态的时间分析，使得进程调度这一抽象模型得到具体化。之后，便是对进程的插入（执行完，插入到完成队列，否则插入到就绪）和再次调度（当改进程再次满足条件时，从就绪队列调度到执行队列）重复过程。

（2）通过设计PCB结构，模拟进程调度，加深了对进程的理解。

（3）提高了C语言编程动手能力，在设计就绪队列时，通过优先数将新进程插入就绪队列中的适当位置。要做多重判断，但实际又是“链表插入”模型的运用，无论多复杂的问题，都可以分化成简单的问题在已有的模型上处理。

五、源代码

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct node

{

char name[20]; /\*进程的名字\*/

int prio; /\*进程的优先级\*/

int round; /\*分配CPU的时间片\*/

int cputime; /\*CPU执行时间\*/

int needtime; /\*进程执行所需要的时间\*/

char state; /\*进程的状态，W——就绪态，R——执行态，F——完成态\*/

int count; /\*记录执行的次数\*/

struct node \*next; /\*链表指针\*/

}PCB;

PCB \*ready=NULL,\*run=NULL,\*finish=NULL; /\*定义三个队列，就绪队列，执行队列和完成队列\*/

int num;

void GetFirst(); /\*从就绪队列取得第一个节点\*/

void Output(); /\*输出队列信息\*/

void InsertPrio(PCB \*in); /\*创建优先级队列，规定优先数越小，优先级越高\*/

void InsertTime(PCB \*in); /\*时间片队列\*/

void InsertFinish(PCB \*in); /\*时间片队列\*/

void PrioCreate(); /\*优先级输入函数\*/

void TimeCreate(); /\*时间片输入函数\*/

void Priority(); /\*按照优先级调度\*/

void RoundRun(); /\*时间片轮转调度\*/

int main(void)

{

char chose;

printf("请输入要创建的进程数目:\n");

scanf("%d",&num);

getchar();

printf("输入进程的调度方法：(P/R)\n");

scanf("%c",&chose);

switch(chose)

{

case 'P':

case 'p':

PrioCreate();

Priority();

break;

case 'R':

case 'r':

TimeCreate();

RoundRun();

break;

default:break;

}

Output();

printf("\n");

printf("实验人：141340209，孙婷\n");

return 0;

}

void GetFirst() /\*取得第一个就绪队列节点\*/

{

run = ready;

if(ready!=NULL)

{

run ->state = 'R';

ready = ready ->next;

run ->next = NULL;

}

}

void Output() /\*输出队列信息\*/

{

PCB \*p;

p = ready;

printf("进程名\t优先级\t轮数\tcpu时间\t需要时间\t进程状态\t计数器\n");

while(p!=NULL)

{

printf("%s\t%d\t%d\t%d\t%d\t\t%c\t\t%d\n",p->name,p->prio,p->round,p->cputime,p->needtime,p->state,p->count);

p = p->next;

}

p = finish;

while(p!=NULL)

{

printf("%s\t%d\t%d\t%d\t%d\t\t%c\t\t%d\n",p->name,p->prio,p->round,p->cputime,p->needtime,p->state,p->count);

p = p->next;

}

p = run;

while(p!=NULL)

{

printf("%s\t%d\t%d\t%d\t%d\t\t%c\t\t%d\n",p->name,p->prio,p->round,p->cputime,p->needtime,p->state,p->count);

p = p->next;

}

}

void InsertPrio(PCB \*in) /\*创建优先级队列，规定优先数越小，优先级越低\*/

{

PCB \*fst,\*nxt;

fst = nxt = ready;

if(ready == NULL) /\*如果队列为空，则为第一个元素\*/

{

in->next = ready;

ready = in;

}

else /\*查到合适的位置进行插入\*/

{

if(in ->prio >= fst ->prio) /\*比第一个还要大，则插入到队头\*/

{

in->next = ready;

ready = in;

}

else

{

while(fst->next != NULL) /\*移动指针查找第一个别它小的元素的位置进行插入\*/

{

nxt = fst;

fst = fst->next;

}

if(fst ->next == NULL) /\*已经搜索到队尾，则其优先级数最小，将其插入到队尾即可\*/

{

in ->next = fst ->next;

fst ->next = in;

}

else /\*插入到队列中\*/

{

nxt = in;

in ->next = fst;

}

}

}

}

void InsertTime(PCB \*in) /\*将进程插入到就绪队列尾部\*/

{

PCB \*fst;

fst = ready;

if(ready == NULL)

{

in->next = ready;

ready = in;

}

else

{

while(fst->next != NULL)

{

fst = fst->next;

}

in ->next = fst ->next;

fst ->next = in;

}

}

void InsertFinish(PCB \*in) /\*将进程插入到完成队列尾部\*/

{

PCB \*fst;

fst = finish;

if(finish == NULL)

{

in->next = finish;

finish = in;

}

else

{

while(fst->next != NULL)

{

fst = fst->next;

}

in ->next = fst ->next;

fst ->next = in;

}

}

void PrioCreate() /\*优先级调度输入函数\*/

{

PCB \*tmp;

int i;

printf("输入进程名字和进程所需时间：\n");

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

{

if((tmp = (PCB \*)malloc(sizeof(PCB)))==NULL)

{

perror("malloc");

exit(1);

}

scanf("%s",tmp->name);

getchar(); /\*吸收回车符号\*/

scanf("%d",&(tmp->needtime));

tmp ->cputime = 0;

tmp ->state ='W';

tmp ->prio = 50 - tmp->needtime; /\*设置其优先级，需要的时间越多，优先级越低\*/

tmp ->round = 0;

tmp ->count = 0;

InsertPrio(tmp); /\*按照优先级从高到低，插入到就绪队列\*/

}

}

void TimeCreate() /\*时间片输入函数\*/

{

PCB \*tmp;

int i;

printf("输入进程名字和进程时间片所需时间：/n");

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

{

if((tmp = (PCB \*)malloc(sizeof(PCB)))==NULL)

{

perror("malloc");

exit(1);

}

scanf("%s",tmp->name);

getchar();

scanf("%d",&(tmp->needtime));

tmp ->cputime = 0;

tmp ->state ='W';

tmp ->prio = 0;

tmp ->round = 2; /\*假设每个进程所分配的时间片是2\*/

tmp ->count = 0;

InsertTime(tmp);

}

}

void Priority() /\*按照优先级调度，每次执行一个时间片\*/

{

int flag = 1;

GetFirst();

while(run != NULL) /\*当就绪队列不为空时，则调度进程如执行队列执行\*/

{

Output(); /\*输出每次调度过程中各个节点的状态\*/

while(flag)

{

run->prio -= 3; /\*优先级减去三\*/

run->cputime++; /\*CPU时间片加一\*/

run->needtime--;/\*进程执行完成的剩余时间减一\*/

if(run->needtime == 0)/\*如果进程执行完毕，将进程状态置为F，将其插入到完成队列\*/

{

run ->state = 'F';

run->count++; /\*进程执行的次数加一\*/

InsertFinish(run);

flag = 0;

}

else /\*将进程状态置为W，入就绪队列\*/

{

run->state = 'W';

run->count++; /\*进程执行的次数加一\*/

InsertTime(run);

flag = 0;

}

}

flag = 1;

GetFirst(); /\*继续取就绪队列队头进程进入执行队列\*/

}

}

void RoundRun() /\*时间片轮转调度算法\*/

{

int flag = 1;

GetFirst();

while(run != NULL)

{

Output();

while(flag)

{

run->count++;

run->cputime++;

run->needtime--;

if(run->needtime == 0) /\*进程执行完毕\*/

{

run ->state = 'F';

InsertFinish(run);

flag = 0;

}

else if(run->count == run->round)/\*时间片用完\*/

{

run->state = 'W';

run->count = 0; /\*计数器清零，为下次做准备\*/

InsertTime(run);

flag = 0;

}

}

flag = 1;

GetFirst();

}

}

# 实验题目：实验七 存储管理之动态链接库

**完成人：**XXX

**报告日期：**2018年5月12日

1. 实验内容简要描述

1）编写动态链接库实现如下功能：

①求两个数的最大值和最小值；

②求1+2+3+4+…+n（n作为一个参数）。

2）编写程序，通过隐式链接和显式链接调用动态链接库。

1. 程序设计
2. 设计思路

动态链接库入口函数

DllMain()函数是动态链接库的入口函数，函数原型如下：

BOOL APIENTRY DllMain（HANDLE hModule，

DWORD ul\_reason\_for\_call,

LPVOID lpReserved)

hModule：动态链接库的句柄；

reason\_for\_call：指明系统调用该函数的原因；

lpReserved：动态链接库是否需要动态加载或卸载。

动态链接库的导入与导出

在动态链接库源程序中声明导出函数的代码如下：

\_declspec(dllexport) MyDllFunction(int x,int y);

其中关键字\_declspec(dllexport)表示要导出其后的函数MyDllFunction()。

如果一个动态链接文件中的函数还需要调用其他动态链接库，此时，动态链接库文件除了导出函数外，还需要一个导入函数，声明导入函数的代码如下：

\_declspec(dllimport) DllFunction(int x,int y);

动态链接库的两种链接方式：隐式链接和显式链接。

隐式链接

应用程序的源代码只引用动态链接库中包含的符号，当应用程序运行时，加载程序隐式地将动态链接库装入到进程的地址空间中。

显式链接

应用程序运行时使用LoadLibrary()显式地加载所需要的动态链接库，并显式地链接需要的输出符号表。

1. 主要数据结构

BOOL APIENTRY DllMain( HANDLE hModule,

DWORD ul\_reason\_for\_call,

LPVOID lpReserved

)

{

return TRUE;

}

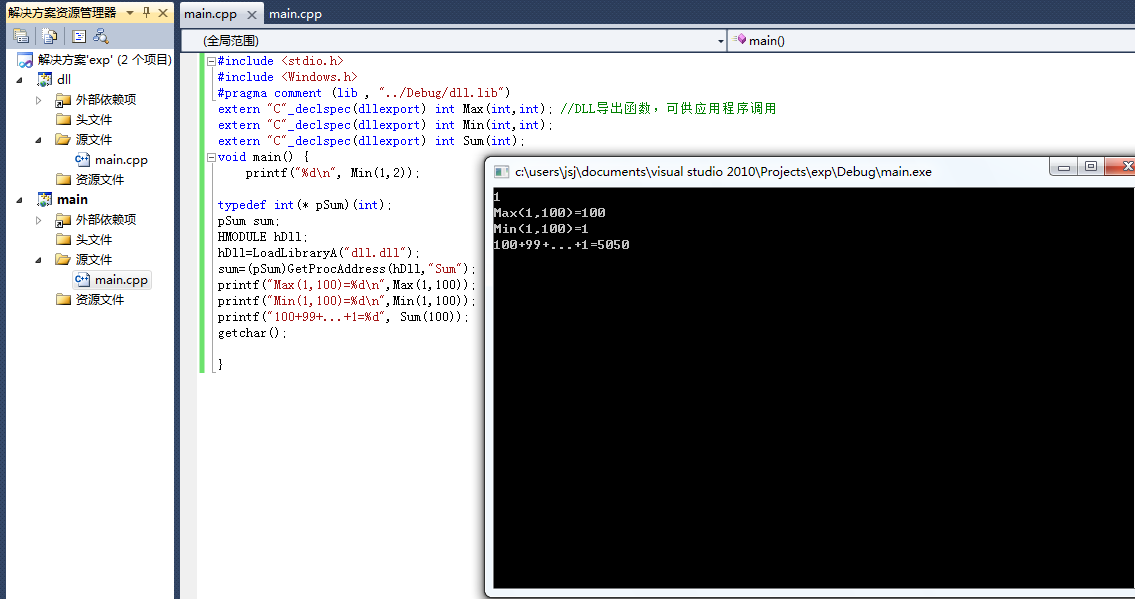
int Max(int x,int y);求最大值

int Min(int x,int y);求最小值

int Sum(int n);求和

1. 实验结果

1.测试结果分析



四、实验体会

1、实验体会和收获

一开始用vc6.0编译时头文件#include”stdafx.h”一直编译不通过所以换成Microsoft Visual Studio 2010编译，实践理解了动态链接库的原理，学会使用动态库编写简单程序，最终完成实验。

五、源代码

Dll：main.cpp

#include "Windows.h"

#include "stdio.h"

extern "C"\_declspec(dllexport) int Max(int x,int y); //DLL导Ì?出?函¡¥数ºy，ê?可¨¦供?应®|用®?程¨¬序¨°调Ì¡Â用®?

extern "C"\_declspec(dllexport) int Min(int x,int y);

extern "C"\_declspec(dllexport) int Sum(int n);

BOOL APIENTRY DllMain( HANDLE hModule,

DWORD ul\_reason\_for\_call,

LPVOID lpReserved

)

{

return TRUE;

}

int Max(int x,int y)

{

if(x>=y)

{

return(x);

}

else

{

return(y);

}

}

int Min(int x,int y)

{

if(x<y)

{

return(x);

}

else

{

return(y);

}

}

int Sum(int n)

{

int s = 0;

for (int i=1;i<=n;i++)

{

s=s+i;

}

return s;

}

Main:main.cpp

#include <stdio.h>

#include <Windows.h>

#pragma comment (lib , "../Debug/dll.lib")

extern "C"\_declspec(dllexport) int Max(int,int); //DLL导Ì?出?函¡¥数ºy，ê?可¨¦供?应®|用®?程¨¬序¨°调Ì¡Â用®?

extern "C"\_declspec(dllexport) int Min(int,int);

extern "C"\_declspec(dllexport) int Sum(int);

void main() {

printf("%d\n", Min(1,2));

typedef int(\* pSum)(int);

pSum sum;

HMODULE hDll;

hDll=LoadLibraryA("dll.dll");

sum=(pSum)GetProcAddress(hDll,"Sum");

printf("Max(1,100)=%d\n",Max(1,100));

printf("Min(1,100)=%d\n",Min(1,100));

printf("100+99+...+1=%d", Sum(100));

getchar();

}

# 实验题目：实验八 存储管理之内存分配

**完成人：**XXX

**报告日期：**2018年5月19日

1. 实验内容简要描述

1) 编写一个程序，创建两个线程，一个用于内存分配，另一个用于跟踪内存的分配情况并打印信息。

2) 将VirtualAlloc函数的参数ftAllocahonType分别改为MEM\_RESET或MEM\_TOP\_DOWN，将nProtect参数分别改为PAGE\_GUARD、PAGE\_NOACCESS或PAGE\_NOCACHE，再进行本实验的各项操作，以及查看内存分配的各个结果，分析原因。

3) 尝试调换分配、回收、内存复位、加锁、解锁、提交、回收的次序，查看结果，并分析原因。

1. 程序设计
2. 设计思路

1）查看系统信息

2）查看系统内存信息

3）虚拟内存分配

4）释放虚拟空间

5）虚拟空间加锁

6）虚拟空间解锁

Windows 进程的虚拟地址空间中也有三种状态的页面：空闲页面、保留页面和提交页面。空闲(Free)页面：空闲页面是指那些可以保留或提交的可用页面。保留(Reserved)页面：保留页面是逻辑页面已分配但没有分配物理存储的页面。设置这种状态的效果是可以保留一部分虚拟地址，这样，如果不预先释放这些地址，就不能被其他应用程序(如 Malloc，LocalAlloc 等)的操作所使用。试图读或写空闲页面或保留页面将导致页面出错异常。保留页面可被释放或提交。提交(Committed)页面：提交页面是物理存储(在内存中或磁盘上)已被分配的页面。可对它加以保护，不许访问或允许只读访问，或允许读写访问。提交也可以被回收以释放存储空间，从而变成保留页面。

在本实验中，首先创建工程makefile生成随机输入文件，其中包含对内存要求作的各种操作；然后创建工程MemoryAllocation，实现输入文件所要求的各项内存管理操作。

1. 主要数据结构

VOID GetSystemInfo(LPSYSTEM\_INFO lpSystemInfo); 该函数返回当前系统的信息

VOID GlobalMemoryStatus(LPMEMORYSTATUS lpBuffer); 该函数可以获得计算机系统中当前使用的物理内存和虚拟内存的信息。

LPVOID VirtualAlloc(LPVOID lpAddress，DWORD dwSize，DWORD flAllocationType，DWORD flProtect); 该函数可以在调用进程的虚拟地址中保留或提交页面。除非设置了MEM\_RESET标志，否则被这个函数分配的内存单元被自动初始化为0。

BOOL VirtualFree(LPVOlD lpAddress，DWORD dwSize，DWORD dwFreeType); 可以释放或注销调用进程虚拟空间中的页面。

BOOL VirtualLocK(LPVOID lpAddress,DWORD dwSize); 该函数可以将进程虚拟空间中的内存加锁。以确保后面的对该区域的存取操作不会产生页面失败。

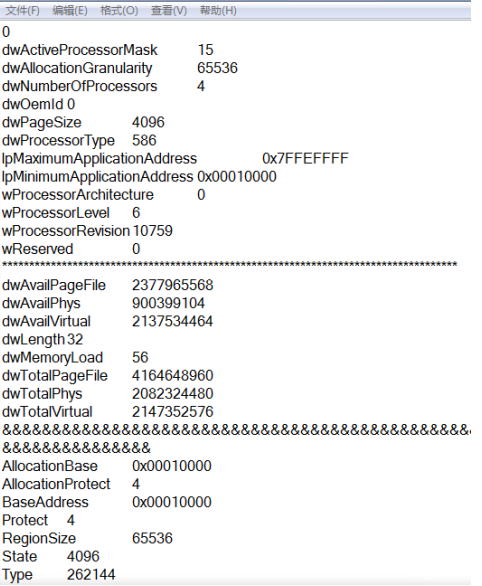
BOOL VirtualUnLock(LPVOID lpAddress,DWORD dwSize); 该函数可以将进程虚拟空间指定范围内的页面解锁，从而系统在必要时可以将这些页面换出到页面文件中。

三、实验结果

1、基本数据

|  |  |  |
| --- | --- | --- |
| 源程序代码行数 | 完成该实验投入的时间（小时数） | 与其他同学讨论次数 |
| 231 | 4 | 0 |

2、实验结果



四、实验体会

1、实验体会和收获

本次试验让我们从不同侧面了解Windows系统对用户进程的虚拟内存空间的管理和分配方法，同时了解跟踪程序的编写方法(与被跟踪程序保持同步，使用Windows提供的信号量)，对Windows分配虚拟内存、改变内存状态，以及对物理内存和页面文件状态查询的API函数的功能、参数限制、使用规则等有了进一步了解。

五、源代码

1)makefile.cpp

//文件生成程序

# include <fstream.h>

# include <stdio.h>

# include <stdlib.h>

# include <time.h>

struct operation

{

int time; //起始时间

int block; //内存页数

int oper; //操作

int protection; //权限

};

int main ()

{

FILE\* file;

file=fopen ("opfile","wb"); //"opfile"为二进制文件用以确定内存操作

operation op;

for (int j=0;j<6;j++) //0--保留，1--提交，2--锁，3--解锁，4--回收，5--释放

for (int i=0;i<5;i++)

//0-PAGE\_READONLY;

//1-PAGE\_READWRITE;

//2-PAGE\_EXECUTE;

//3-PAGE\_EXECUTE\_READ;

//4-PAGE\_EXECUTE\_READWRITE;

{

op.time=rand()%1000; //随机生成等待时间

op.block=rand()%5+1; //随机生成块大小

op.oper=j;

op.protection=i;

fwrite (&op,sizeof(operation),1,file); //将生成的结构写入文件

}

return 0;

}

2) memory\_op.cpp

//从文件读入每次的操作，并将结果输出到out.txt文件中

# include <stdio.h>

# include <stdlib.h>

# include <windows.h>

# include <iostream.h>

# include <fstream.h>

struct operation

{

int time; //起始时间

int block; //内存页数

int oper; //操作

int protection; //权限

};

struct trace //跟踪每一次分配活动的数据结构

{

LPVOID start; //起始地址

long size; //分配的大小

};

HANDLE allo,trac; //信号量句柄

DWORD Tracker (LPDWORD lpdwparm) //跟踪allocator线程的内存行为，并输出必要信息

{ofstream outfile; //输出文件

outfile.open ("out.txt");

for (int i=0;i<=30;i++)

{

WaitForSingleObject (trac,INFINITE); //等待allocator一次内存分配活动结束

//打印内存状况和系统状况

outfile<<i<<endl;

//以下一段显示系统信息，每次执行操作后系统信息不变

SYSTEM\_INFO info; //系统消息

GetSystemInfo (&info);

outfile<<"dwActiveProcessorMask"<<"\t"<<info.dwActiveProcessorMask<<endl;

outfile<<"dwAllocationGranularity"<<"\t"<<info.dwAllocationGranularity<<endl;

outfile<<"dwNumberOfProcessors"<<"\t"<<info.dwNumberOfProcessors<<endl;

outfile<<"dwOemId"<<"\t"<<info.dwOemId<<endl;

outfile<<"dwPageSize"<<"\t"<<info.dwPageSize<<endl;

outfile<<"dwProcessorType"<<"\t"<<info.dwProcessorType<<endl;

outfile<<"lpMaximumApplicationAddress"<<"\t"<<info.lpMaximumApplicationAddress<<endl;

outfile<<"lpMinimumapplicationAddress"<<"\t"<<info.lpMinimumApplicationAddress<<endl;

outfile<<"wProcessorArchitecture"<<"\t"<<info.wProcessorArchitecture<<endl;

outfile<<"wProcessorLevel"<<"\t"<<info.wProcessorLevel<<endl;

outfile<<"wProcessorRevision"<<"\t"<<info.wProcessorRevision<<endl;

outfile<<"wReserved"<<"\t"<<info.wReserved<<endl;

outfile<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

//内存状况

MEMORYSTATUS status; //内存状态

GlobalMemoryStatus (&status);

outfile<<"dwAvailPageFile"<<"\t"<<status.dwAvailPageFile<<endl;

outfile<<"dwAvailPhys"<<"\t"<<status.dwAvailPhys<<endl;

outfile<<"dwAvailVirtual"<<"\t"<<status.dwAvailVirtual<<endl;

outfile<<"dwLength"<<"\t"<<status.dwLength<<endl;

outfile<<"dwMemoryLoad"<<"\t"<<status.dwMemoryLoad<<endl;

outfile<<"dwTotalPageFile"<<"\t"<<status.dwTotalPageFile<<endl;

outfile<<"dwTotalPhy"<<"\t"<<status.dwTotalPhys<<endl;

outfile<<"dwTotalVirtual"<<"\t"<<status.dwTotalVirtual<<endl;

outfile<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

//以下一段显示内存基本信息，每次操作后内存基本信息不变

MEMORY\_BASIC\_INFORMATION mem; //内存基本信息

VirtualQuery (info.lpMinimumApplicationAddress,&mem,sizeof(MEMORY\_BASIC\_INFORMATION) );

outfile<<"AllocationBase"<<"\t"<<mem.AllocationBase<<endl;

outfile<<"AllocationProtect"<<"\t"<<mem.AllocationProtect<<endl;

outfile<<"BaseAddress"<<"\t"<<mem.BaseAddress<<endl;

outfile<<"Protect"<<"\t"<<mem.Protect<<endl;

outfile<<"RegionSize"<<"\t"<<mem.RegionSize<<endl;

outfile<<"State"<<"\t"<<mem.State<<endl;

outfile<<"Type"<<"\t"<<mem.Type<<endl;

outfile<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

//释放信号量通知allocator可以执行下一次内存分配活动

ReleaseSemaphore (allo,1,NULL);

}

return 0;

}

void Allocator () //模拟内存分配活动的线程

{

trace traceArray[5];

int index=0;

FILE\* file;

file=fopen ("opfile","rb");

operation op;

SYSTEM\_INFO info;

DWORD temp;

GetSystemInfo (&info);

for (int i=0;i<30;i++)

{

WaitForSingleObject (allo,INFINITE); //等待tracker打印结束的信号量

cout<<i<<":";

fread (&op,sizeof(operation),1,file);

Sleep (op.time); //执行时间，如果想在指定的时间可以取消注释

GetSystemInfo (&info);

switch (op.protection) //根据文件内容确定权限

{

case 0:

{

index=0;

temp=PAGE\_READONLY;

break;

}

case 1:

temp=PAGE\_READWRITE;

break;

case 2:

temp=PAGE\_EXECUTE;

break;

case 3:

temp=PAGE\_EXECUTE\_READ;

break;

case 4:

temp=PAGE\_EXECUTE\_READWRITE;

break;

default:

temp=PAGE\_READONLY;

}

switch (op.oper)

{

case 0: //保留一个区域

{

cout<<"reserve now "<<endl;

traceArray[index].start=VirtualAlloc (NULL,op.block\*info.dwPageSize,

MEM\_RESERVE,PAGE\_NOACCESS);

traceArray[index++].size=op.block\*info.dwPageSize;

cout<<"starting address:"<<traceArray[index-1].start<<"\t"<<"size:"

<<traceArray[index-1].size<<endl;

break;

}

case 1: //提交一个区域

{

cout<<"commit now "<<endl;

traceArray[index].start=VirtualAlloc (traceArray[index].start,

traceArray[index].size,MEM\_COMMIT,temp);

index++;

cout<<"starting address:"<<traceArray[index-1].start<<"\t"<<"size:"

<<traceArray[index-1].size<<endl;

break;

}

case 2: //锁住一个区域

{

cout<<"lock now"<<endl;

cout<<"starting address:"<<traceArray[index].start<<"size:"<<

traceArray[index].size<<endl;

if (!VirtualLock (traceArray[index].start,traceArray[index++].size) )

cout<<GetLastError ()<<endl; //函数返回错误号

break;

}

case 3: //解锁一个区域

{

cout<<"unlock now"<<endl;

cout<<"starting address:"<<traceArray[index].start<<"\t"<<"size:"

<<traceArray[index].size<<endl;

if (!VirtualUnlock (traceArray[index].start,traceArray[index++].size) )

cout<<GetLastError ()<<endl;

break;

}

case 4: //回收一个区域

{

cout<<"decommit now"<<endl;

cout<<"starting address:"<<traceArray[index].start<<"\t"

<<"size:"<<traceArray[index].size<<endl;

if (! VirtualFree (traceArray[index].start,traceArray[index++].size,

MEM\_DECOMMIT) )

cout<<GetLastError ()<<endl;

break;

}

case 5: //释放一个区域

{

cout<<"release now"<<endl;

cout<<"starting address:"<<traceArray[index].start<<"\t"<<

"size:"<<traceArray[index].size<<endl;

if (!VirtualFree (traceArray[index++].start,0,MEM\_RELEASE) )

cout<<GetLastError ()<<endl;

break;

}

default:

cout<<"error"<<endl;

}

ReleaseSemaphore (trac,1,NULL); //释放信号量通知tracker可以打印信息

}

}

int main ()

{

DWORD dwThread;

HANDLE handle[2]; //生成两个线程

handle[0]=CreateThread (NULL,0,(LPTHREAD\_START\_ROUTINE) Tracker,NULL,0,

&dwThread);

handle[1]=CreateThread (NULL,0,(LPTHREAD\_START\_ROUTINE) Allocator,NULL,0,

&dwThread);

//生成两个信号量

allo=CreateSemaphore (NULL,0,1,"allo");

trac=CreateSemaphore (NULL,1,1,"trac");

//等待线程执行结束后再退出

WaitForMultipleObjects (2,handle,TRUE,INFINITE);

return 0;

}

# 实验题目：实验九 存储管理之页面置换算法

完成人：XXX

报告日期：2018年5月26日

一、 实验内容简要描述

1） 通过随机数产生一个指令序列，共320条指令，指令的地址按下述原则生成：

 50%的指令是顺序执行的；

 25%的指令是均匀分布在前地址部分。

 25%的指令是均匀分布在后地址部分。

具体的实施办法是：

 在[0，319]的指令地址之间随机选取一点m；

 顺序执行一条指令，即执行地址为m+1的指令；

 在前地址[0，m+1]中随机选取一条指令并执行，该指令的地址为m’；

 顺序执行一条指令，其地址为m’+1；

 在后地址[m’+2，319]中随机选取一条指令并执行；

 重复上述步骤——，直到执行320次指令。

2） 将指令序列变换成页地址流，设：

 页面大小为1K；

 用户内存容量为4页到32页；

 用户虚存容量为32K。

在用户虚存中，按每K存放10条指令排列虚存地址，即320条指令在虚存中的存放方式为：

第0条-9条指令为第0页（对应虚存地址为[0，9]）；

第10条-第19条指令为第一页（对应虚存地址为[10，19]）；

……

第310条-第319条指令为第31页（对应虚存地址为[310，319]）。

按以上方式，用户指令可组成32页。

3） 计算并输出下述各种算法在不同内存容量下的命中率：

 先进先出的算法（FIFO）；

 最近最少使用算法（LRR）；

 最佳淘汰算法（OPT）；先淘汰最不常用的页地址；

 最少访问页面算法（LFR）；

 最近不经常使用算法（NUR）；

其中和为选择内容。

命中率=1-页面失效次数/页地址流长度。

在本实验中，页地址流长度为320，页面失效次数为每次访问相应指令时，该指令所对应的页不在内存的次数。

二、 程序设计

1、 设计思路

#define total\_instruction 320 //指令流长

#define total\_vp 32 //虚页长

#define clear\_period 50 //清周期

pfc\_type pfc[total\_vp], //主存区页面控制结构数组

pfc\_type \*freepf\_head, //主存区页面控制结构的空闲页面头指针

pfc\_type \*busypf\_head, //主存区页面控制结构的忙页面头指针

pfc\_type \*busypf\_tail; //主存区页面控制结构的忙页面尾指针

int diseffect; //页错误计数器，初次把页面载入主存时也当做页错误

pl\_type pl[total\_vp]; //页面结构数组

2、 主要数据结构

typedef struct //页面结构

{

int pn, //页面序号

pfn, //页面所在内存区的帧号

counter, //单位时间内访问次数

time; //上次访问的时间

}pl\_type;

struct pfc\_struct{ //页面控制结构，模拟内存中的页集

int pn, //页面号

pfn; //内存区页面的帧号

struct pfc\_struct \*next; //页面指针，用于维护内存缓冲区的链式结构

};

int initialize(int); //初始化页面结构数组和页面控制结构数组

int FIFO(int); //先进先出算法

int LRU(int); //最近最久未使用算法

int OPT(int); //最佳置换算法

int CLOCK(int); //简单时钟(钟表)算法

三、实验结果

1、基本数据

源程序代码行数 完成该实验投入的时间（小时数） 与其他同学讨论次数

301 2 0

2、测试结果截图及分析

理论上，四种替换算法的命中率由高到底排列应该是OPT>LRU>CLOCK>FIFO。实际上，从实验数据观测得到，存在这种由高到低的趋势，由page=4时可以观测到，但是效果不是很明显。

效果不明显的原因：

推测与指令流的产生方式有关系。因为指令流的产生方式要能体现局部性原理，所以该指令流产生设计为：50%的指令是顺序执的，25%的指令是均匀分布在前地址部分，25%的指令是均匀分布在后地址部分。但是这样的指令流设计方式能否最佳地体现局部性原理，这还有待验证。

同时，估计和指令数量有关系。因为320条指令太少了，通常一个稍大点的程序都几千行指令了。

而且由于随即数产生具有一定的波动性，该命中率的计算也有一定的波动性。所以会有局部的实验数据与理论不符。改进方法是多次实验取平均值，这样可以减小波动，让实验数据更加平滑。

唯一显著的是OPT算法的命中率与其他3个调度算法保持了比较大的差距。例如在page=30时，OPT算法就能达到0.9的命中率了。

到后期，由于page越来越大，因此越来越容易命中，因此各替换算法的命中率差距变小了。这由最后几行命中率相似可以看出。

四、实验体会

1、实验体会和收获

存储管理的主要功能之一是合理地分配空间，请求页式管理是一种常用的虚拟存储管理技术。本实验是通过请求页式存储管理中页面置换算法模拟设计，了解虚拟存储技术的特点，掌握请求页式管理的页面置换算法。

一开始做这个实验时，首先是看书，先把书上的替换算法知识点弄明白，要明白各种算法的优缺点和相互之间衍生互补关系。这四个算法中，难以实现的是LRU算法，因为它涉及到访问时间的计算，而且它的开销也比较大。OPT算法次难，它需要计算最近访问时间，并替换最近访问时间最大的页。而FIFO和CLOCK实现起来比较容易，FIFO算法的实现和CLOCK算法的实现很相似，FIFO可视为CLOCK的退化版。我先写了CLOCK算法，再删去一些约束条件就退化为FIFO算法。这就是两者的相同之处。理论上，CLOCK算法需要维持一个循环的主存缓冲区，需要一个循环队列去实现，并且，FIFO算法保持先进先出，因此需要一个先进先出队列。但是，我实现这两个算法只用到了单向链表的数据结构，剩下的由其中的指针去把握了。因此，必须对指针使用有敏锐的感觉。

五、源代码

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h> //在window操作系统下要屏蔽此条指令

#define TRUE 1

#define FALSE 0

#define INVALID -1

#define total\_instruction 320 //指令流长

#define total\_vp 32 //虚页长

#define clear\_period 50 //清周期

typedef struct //页面结构

{

int pn, //页面序号

pfn, //页面所在内存区的帧号

counter, //单位时间内访问次数

time; //上次访问的时间

}pl\_type;

pl\_type pl[total\_vp]; //页面结构数组

struct pfc\_struct{ //页面控制结构

int pn, //页面号

pfn; //内存区页面的帧号

struct pfc\_struct \*next; //页面指针，用于维护内存缓冲区的链式结构

};

typedef struct pfc\_struct pfc\_type; //主存区页面控制结构别名

pfc\_type pfc[total\_vp], //主存区页面控制结构数组

\*freepf\_head, //主存区页面控制结构的空闲页面头指针

\*busypf\_head, //主存区页面控制结构的忙页面头指针

\*busypf\_tail; //主存区页面控制结构的忙页面尾指针

int diseffect; //页错误计数器，初次把页面载入主存时也当做页错误

int a[total\_instruction]; //随即指令流数组

int page[total\_instruction]; //指令对应的页面号

int offset[total\_instruction]; //指令所在页面中的偏移量

int initialize(int); //初始化页面结构数组和页面控制结构数组

int FIFO(int); //先进先出算法

int LRU(int); //最近最久未使用算法

int OPT(int); //最佳置换算法

int CLOCK(int); //简单时钟(钟表)算法

int main( )

{

int s; //随机数

int i;

srand(10\*getpid()); /\*每次运行时进程号不同，用来作为初始化随机数队列的"种子"\*/

s = (int)((float)(total\_instruction-1)\*(rand()/(RAND\_MAX+1.0)));

printf("\n------------随机产生指令流------------\n");

for (i=0; i<total\_instruction; i+=4) //产生指令队列

{

a[i]=s; //任选一指令访问点m

a[i+1]=a[i]+1; //顺序执行一条指令

a[i+2]=(int)((float)a[i]\*(rand()/(RAND\_MAX+1.0))); //执行前地址指令m'

a[i+3]=a[i+2]+1; //顺序执行一条指令

printf("%6d%6d%6d%6d\n", a[i],a[i+1],a[i+2],a[i+3]);

s = (int)((float)((total\_instruction-1)-a[i+2])\*(rand()/(RAND\_MAX+1.0))) + a[i+2];

}

printf("--------------------------------------\n");

for (i=0;i<total\_instruction;i++) //将指令序列变换成页地址流

{

page[i]=a[i]/10;

offset[i]=a[i]%10;

}

printf("\n--不同页面工作区各种替换策略的命中率表--\n");

printf("Page\t FIFO\t LRU\t OPT\t CLOCK\n");

for(i=4;i<=32;i++) //用户内存工作区从个页面到个页面

{

printf(" %2d \t",i);

FIFO(i);

LRU(i);

OPT(i);

CLOCK(i);

printf("\n");

}

return 0;

}

//初始化页面结构数组和页面控制结构数组

//total\_pf; 用户进程的内存页面数

int initialize(int total\_pf)

{

int i;

diseffect=0;

for(i=0;i<total\_vp;i++)

{

pl[i].pn=i;

pl[i].pfn=INVALID; //置页面所在主存区的帧号为-1.表示该页不在主存中

pl[i].counter=0; //置页面结构中的访问次数为

pl[i].time=-1; //置页面结构中的上次访问的时间为-1

}

for(i=0;i<total\_pf-1;i++)

{

pfc[i].next=&pfc[i+1]; //建立pfc[i-1]和pfc[i]之间的链接

pfc[i].pfn=i; //初始化主存区页面的帧号

}

pfc[total\_pf-1].next=NULL;

pfc[total\_pf-1].pfn=total\_pf-1;

freepf\_head=&pfc[0]; //主存区页面控制结构的空闲页面头指针指向pfc[0]

return 0;

}

//最近最久未使用算法

//int total\_pf; 用户进程的内存页面数

int LRU (int total\_pf)

{

int MinT; //最小的访问时间，即很久没被访问过

int MinPn; //拥有最小的访问时间的页的页号

int i,j;

int CurrentTime; //系统当前时间

initialize(total\_pf); //初始化页面结构数组和页面控制结构数组

CurrentTime=0;

diseffect=0;

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) //页面失效

{

diseffect++; //页错误次数加

if(freepf\_head==NULL) //无空闲页面

{

MinT=100000;

for(j=0;j<total\_vp;j++){ //找出time的最小值，表明该页很久没被访问过

if(MinT>pl[j].time&&pl[j].pfn!=INVALID)

{

MinT=pl[j].time;

MinPn=j;

}

}

freepf\_head=&pfc[pl[MinPn].pfn]; //最久没被访问过的页被释放

pl[MinPn].pfn=INVALID; //最久没被访问过的页被换出主存

pl[MinPn].time=-1; //最久没被访问过的页的访问时间置为无效

freepf\_head->next=NULL;

}

pl[page[i]].pfn=freepf\_head->pfn; //有空闲页面,把相应的页面换入主存，并把pfn改为相应的帧号

pl[page[i]].time=CurrentTime; //令访问时间为当前系统时间

freepf\_head=freepf\_head->next; //减少一个空闲页面

}

else

pl[page[i]].time=CurrentTime; //命中则刷新该单元的访问时间

CurrentTime++; //系统当前时间加

}

printf("%6.3f\t",1-(float)diseffect/320);

return 0;

}

//最佳置换算法

//int total\_pf; 用户进程的内存页面数

int OPT(int total\_pf)

{

int i,j;

int MaxD; //将来最近一次访问的距离的最大值（以时间单元度量）

int MaxPn; //将来最近一次访问的距离的最大值的页号

int dis; //距离计数器

int dist[total\_vp]; //距离数组，保存距离上一次访问的时间差距个数

initialize(total\_pf); //初始化页面结构数组和页面控制结构数组

diseffect=0;

for(i=0;i<total\_instruction;i++)

{

if(pl[page[i]].pfn==INVALID) //页面失效

{

diseffect++; //页错误次数加

if(freepf\_head==NULL) //无空闲页面

{

for(j=0;j<total\_vp;j++)

{

if(pl[j].pfn!=INVALID) //如果该页在主存中

dist[j]=100000; // 该页关联的距离值改为最大值

else

dist[j]=0; //如果不在该页主存中，该页关联的距离值改为

}

dis=1; //初始距离值为

for(j=i+1;j<total\_instruction;j++) //从要替换的指令的下一条算起，

{

if(pl[page[j]].pfn!=INVALID &&pl[page[j]].counter==0) //如果该页在主存中,并且是将要最近访问的页

//if(pl[page[j]].pfn!=INVALID && dist[page[j]]==100000) //此条语句原理与上相同

{ dist[page[j]]=dis; //距离值改为dis

pl[page[j]].counter=1; //使访问次数标志加，区别第一次访问和第二次访问

}

dis++;

}

MaxD=-1;

for(j=0;j<total\_vp;j++)

{

pl[j].counter=0; //重置访问次数为

if(MaxD<dist[j]) //查找将来最近一次访问的距离的最大值及其序号

{

MaxD=dist[j];

MaxPn=j;

}

}

freepf\_head=&pfc[pl[MaxPn].pfn]; //替换将来一段时间最久访问的页

freepf\_head->next=NULL;

pl[MaxPn].pfn=INVALID;

}

pl[page[i]].pfn=freepf\_head->pfn; //把当前页换入主存中，并且把当前页的pfn改为换入页的帧号，

freepf\_head=freepf\_head->next; //减少一个空闲页面

}//if

}//for

printf("%6.3f\t",1-(float)diseffect/320);

return 0;

}

//简单时钟算法

//int total\_pf; 用户进程的内存页面数

int CLOCK(int total\_pf)

{

int i;

int use[total\_vp]; //使用位

int swap;

swap=0; //发生替换

initialize(total\_pf);

pfc\_type \*pnext; //时钟指针

pfc\_type \*head; //队列头指针

pnext=freepf\_head;

head=freepf\_head;

for(i=0;i<total\_vp;i++){use[i]=0;} //初始化使用位为

diseffect=0;

for(i=0;i<total\_instruction;i++)

{

if (pl[page[i]].pfn==INVALID) //页面失效,不在主存中

{

diseffect++; //页错误次数加

if(freepf\_head==NULL) //无空闲页面

{

while(use[pnext->pfn]==1) //若时钟指针指向的页的使用位为，则改为并跳过

{

use[pnext->pfn]=0;

pnext=pnext->next;

if(pnext==NULL) pnext=head; //如果时钟指针到达队列尾部，重新返回头部

}

//换出被替换的页

pl[pnext->pn].pfn=INVALID;

swap=1;

}

if(use[pnext->pfn]==0){ //如果使用位为，则换入相应的页

pl[page[i]].pfn=pnext->pfn; //页面结构中要标记帧号

pnext->pn=page[i]; //页面控制结构中要标记页号

use[pnext->pfn]=1; //重置使用位为

pnext=pnext->next; //时钟指针下移

if(pnext==NULL) pnext=head; //如果时钟指针到达队列尾部，重新返回头部

if(swap==0){ freepf\_head=freepf\_head->next; }

}

}else{//页面在主存中

use[pl[page[i]].pfn]=1; //刷新使用位为

}

}

printf("%6.3f\t",1-(float)diseffect/320);

return 0;

}

//先进先出算法版本

//int total\_pf; 用户进程的内存页面数

//实现细节由CLOCK算法退化而来，与FIFO同效果

int FIFO(int total\_pf)

{

int i;

int use[total\_vp];

int swap=0;

initialize(total\_pf);

pfc\_type \*pnext,\*head;

pnext=freepf\_head;

head=freepf\_head;

for(i=0;i<total\_vp;i++){use[i]=0;}

diseffect=0;

for(i=0;i<total\_instruction;i++)

{

if (pl[page[i]].pfn==INVALID) //页面失效,不在主存中

{

diseffect++;

if(freepf\_head==NULL) //无空闲页面

{

while(use[pnext->pfn]==1)

{

use[pnext->pfn]=0;

pnext=pnext->next;

if(pnext==NULL) pnext=head;

}

//换出被替换的页

pl[pnext->pn].pfn=INVALID;

swap=1;

}

if(use[pnext->pfn]==0){ //如果使用位为，则换入相应的页

pl[page[i]].pfn=pnext->pfn; //页面结构中要标记帧号

pnext->pn=page[i]; //页面控制结构中要标记页号

use[pnext->pfn]=1; //重置使用位为

pnext=pnext->next;

if(pnext==NULL) pnext=head;

if(swap==0){ freepf\_head=freepf\_head->next; }

}

}

}

printf("%6.3f\t",1-(float)diseffect/320);

return 0;

}

# 实验题目：实验十 设备管理

**完成人：**XXX

**报告日期：**2018年6月6日

1. 实验内容简要描述

1）实现获取当地磁盘（如C:盘、D:盘、E:盘等）的详细信息。

2）读/写磁盘指定位置信息。

1. 程序设计
2. 设计思路

1）文件创建：函数CreateFile()用于打开磁盘驱动器并返回一个文件句柄，这里驱动器被当做文件处理。

2）获取磁盘的基本信息

3）设置读/写操作的位置

1. 主要数据结构

typedef struct\_DISK\_GEOMETRY{

LARGE\_INTEGER Cylinders; //磁盘柱面数

MEDIA\_TYPE MediaType; //介质类型

DWORD TracksPerCylinder; //每个柱面的磁道数

DWORD SectorsPerTrace; //每个磁道的扇区数

DWORD BytesPerSector; //每个扇区的字节数

}DISK\_GEOMETRY;

HANDLE CreateFile（

LPCTSTR lpFileName， //指向文件名指针

DWORD dwDesiredAccess， //读/写访问模式

DWORD dwShareMode， //共享模式

LPSECURITY\_ATTRIBUTES lpSecurityAttributes，//指向安全属性的指针

DWORD dwCreationDisposition，//文件存在标志

DWORD dwFlagsAndAttributes，//文件属性

HANDLE hTemplateFile //指向访问模板文件的句柄

）

BOOL DeviceIoControl（

HANDLE hDevice， //设备句柄

DWORD dwIoControlCode, //操作控制代码

LPVOID lpInBuffer, //输入数据缓冲区

DWORD nInBufferSize, //输入数据缓冲区大小

LPVOID lpOutBuffer, //输出数据缓冲区

DWORD nOutBufferSize, //输出数据缓冲区大小

LPDWORD lpBytesReturned, //可获取的字节计数

LPOVERLAPPED lpOverlapped //指向OVERLAPPED结构的指针

)

DWORD SetFilePointer(

HANDLE hFile, //文件句柄

LONG lpDistanceToMove, //文件指针要移动的偏移量的低32位

PLONG lpDistanceToMoveHigh, //文件指针要移动的偏移量的高32位

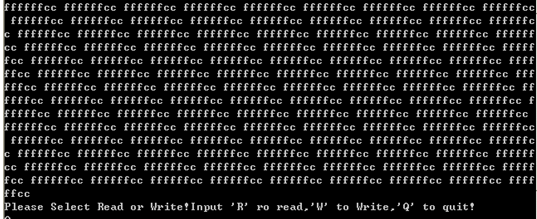
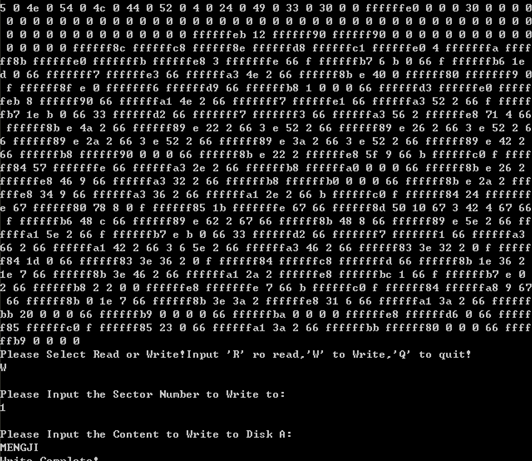
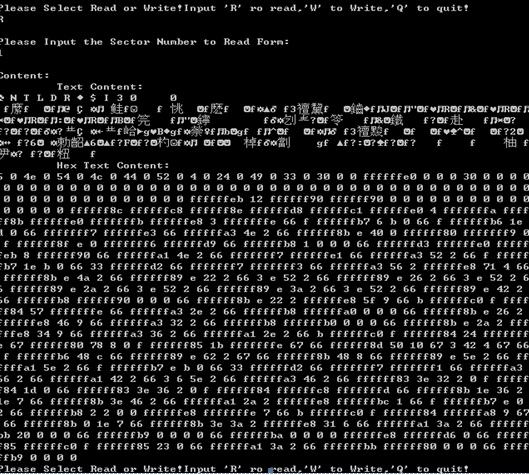
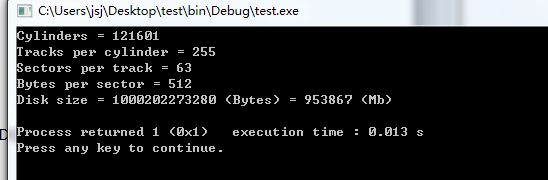
DWORD dwMoveMethod //移动起点

)

dwMoveMethod：文件指针移动的初始位置，其值可取下表中的数值。

三、实验结果

1、测试结果分析



四、实验体会

1、实验体会和收获

通过此次试验，了解磁盘的基本结构，熟悉Windows环境下获取磁盘的详细信息和读写磁盘指定位置的编程。初步了解了操作系统中获取磁盘信息程序代码的基本结构和需要注意的一些细节。

五、源代码

#include <windows.h>

#include <winioctl.h>

BOOL GetDriveGeometry(DISK\_GEOMETRY \*pdg)

{

HANDLE hDevice; // handle to the drive to be examined

BOOL bResult; // results flag

DWORD junk; // discard results

hDevice = CreateFile("\\\\.\\PhysicalDrive0", // drive to open

0, // no access to the drive

FILE\_SHARE\_READ | // share mode

FILE\_SHARE\_WRITE,

NULL, // default security attributes

OPEN\_EXISTING, // disposition

0, // file attributes

NULL); // do not copy file attributes

if (hDevice == INVALID\_HANDLE\_VALUE) // cannot open the drive

{

return (FALSE);

}

bResult = DeviceIoControl(hDevice, // device to be queried

IOCTL\_DISK\_GET\_DRIVE\_GEOMETRY, // operation to perform

NULL, 0, // no input buffer

pdg, sizeof(\*pdg), // output buffer

&junk, // # bytes returned

(LPOVERLAPPED) NULL); // synchronous I/O

CloseHandle(hDevice);

return (bResult);

}

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

{

DISK\_GEOMETRY pdg; // disk drive geometry structure

BOOL bResult; // generic results flag

ULONGLONG DiskSize; // size of the drive, in bytes

bResult = GetDriveGeometry (&pdg);

if (bResult)

{

printf("Cylinders = %I64d\n", pdg.Cylinders);

printf("Tracks per cylinder = %ld\n", (ULONG) pdg.TracksPerCylinder);

printf("Sectors per track = %ld\n", (ULONG) pdg.SectorsPerTrack);

printf("Bytes per sector = %ld\n", (ULONG) pdg.BytesPerSector);

DiskSize = pdg.Cylinders.QuadPart \* (ULONG)pdg.TracksPerCylinder \*

(ULONG)pdg.SectorsPerTrack \* (ULONG)pdg.BytesPerSector;

printf("Disk size = %I64d (Bytes) = %I64d (Mb)\n", DiskSize,

DiskSize / (1024 \* 1024));

}

else

{

printf("GetDriveGeometry failed. Error %ld.\n", GetLastError());

}

return ((int)bResult);

}

#include "stdafx.h"

#include "操作系统实验六.h"

#include "winioctl.h"

#ifdef \_DEBUG

#define new DEBUG\_NEW

#undef THIS\_FILE

static char THIS\_\_FILE[]=\_\_FILE\_\_;

#endif

DISK\_GEOMETRY disk\_info;

HANDLE GetDiskInformation(char drivername);

BOOL SectorRead(HANDLE Handle);

BOOL SectorWrite(HANDLE Handle);

/////////////////////////////////////////////////////

//The one and only application object

CWinApp theApp;

using namespace std;

int \_tmain(int argc,TCHAR\* argv[],TCHAR\* envp[])

{

int nRetCode=0;

HANDLE Handle;

char Choice;

Handle=GetDiskInformation('C');

while(TRUE)

{

printf("Please Select Read or Write!Input 'R' ro read,'W' to Write,'Q' to quit!\n");

Choice=getchar();

printf("\n");

switch(Choice)

{

case 'W':

{

if(!SectorWrite(Handle)) printf("Write Sector Fail!\n");

getchar();

break;

}

case 'R':

{

if(!SectorRead(Handle)) printf("Read Sector Fail!\n");

getchar();

break;

}

case 'Q':

{

exit(0);

break;

}

default:

{

printf("Input Error!,Try again please!\n");

getchar();

}

}

}

return nRetCode;

}

HANDLE GetDiskInformation(char drivername) // GetDiskInformation获取磁盘信息

{

char device[]="\\\\.\\";

device[4]=drivername;

HANDLE FloopyDisk;

DWORD ReturnSize;// DWORD双字节值

DWORD Sector;

double DiskSize;

FloopyDisk=CreateFile(device,

GENERIC\_READ|GENERIC\_WRITE,

FILE\_SHARE\_READ|FILE\_SHARE\_WRITE,

NULL,

OPEN\_EXISTING,

FILE\_FLAG\_RANDOM\_ACCESS|FILE\_FLAG\_NO\_BUFFERING,

NULL);

if(FloopyDisk==INVALID\_HANDLE\_VALUE)

printf("INVALID\_HANDLE\_VALUE!\n");

if(GetLastError()==ERROR\_ALREADY\_EXISTS)

printf("Can not Open Disk!%d\n",GetLastError());

if(!DeviceIoControl(FloopyDisk,

IOCTL\_DISK\_GET\_DRIVE\_GEOMETRY,

NULL,

0,

&disk\_info,

50,

&ReturnSize,

(LPOVERLAPPED)NULL))

printf("Open Disk Error!%d\n",GetLastError());

printf("Disk Information:\n");

printf("\t BytePerSector:%d\n",disk\_info.BytesPerSector);

printf("\t SectorPerTrack:%d\n",disk\_info.SectorsPerTrack);

printf("\t TracksPerCylider:%d\n",disk\_info.TracksPerCylinder);

printf("\t Cylider:%d\n",disk\_info.Cylinders);

Sector=disk\_info.Cylinders.QuadPart\*

disk\_info.TracksPerCylinder\*

disk\_info.SectorsPerTrack;

printf("\t There is %d Sectors!\n",Sector);

DiskSize=Sector\*disk\_info.BytesPerSector;

printf("\t Size of Disk:%4.2f KB\n",(DiskSize)/(1024\*1024));

return FloopyDisk;

}

BOOL SectorRead(HANDLE Handle)

{

char ReadBuffer[1024\*16];

DWORD SectorNumber;

DWORD BytestoRead;

DWORD Sector;

DWORD rc;

int i;

if(Handle==NULL)

{

printf("There is No disk!\n");

return FALSE;

}

printf("Please Input the Sector Number to Read Form:\n");

scanf("%d",&SectorNumber);

printf("\n");

Sector=disk\_info.Cylinders.QuadPart\*

disk\_info.TracksPerCylinder\*

disk\_info.SectorsPerTrack;

if(SectorNumber>Sector) printf("There is not this Sector!\n");

printf("Content:\n");

BytestoRead=SectorNumber\*(disk\_info.BytesPerSector);

rc=SetFilePointer(Handle,BytestoRead,NULL,FILE\_BEGIN);

if(!ReadFile(Handle,ReadBuffer,BytestoRead,&BytestoRead,NULL))

{

printf("Read File Error:%d\n",GetLastError());

return FALSE;

}

printf("\t Text Content:\n");

for(i=0;i<512;i++)

{

printf("%c",ReadBuffer[i]);

}

printf("\n");

printf("\t Hex Text Content:\n");

for(i=0;i<512;i++)

{

printf("%x",ReadBuffer[i]);

printf(" ");

}

printf("\n");

return TRUE;

}

BOOL SectorWrite(HANDLE Handle)

{

char WriteBuffer[1024];

DWORD SectorNumber,SectorMove;

DWORD BytestoWrite;

DWORD Sector;

DWORD rc;

if(Handle==NULL)

{

printf("There is No disk!\n");

return FALSE;

}

printf("Please Input the Sector Number to Write to:\n");

scanf("%d",&SectorNumber);

printf("\n");

Sector=disk\_info.Cylinders.QuadPart\*

disk\_info.TracksPerCylinder\*

disk\_info.SectorsPerTrack;

if(SectorNumber>Sector)printf("There is not this Sector!\n");

printf("Please Input the Content to Write to Disk A:\n");

scanf("%s",&WriteBuffer);// WriteBuffer指的是写缓冲区

SectorMove=SectorNumber\*(disk\_info.BytesPerSector);

rc=SetFilePointer(Handle,SectorMove,NULL,FILE\_BEGIN);

if(!WriteFile(Handle,WriteBuffer,512,&BytestoWrite,NULL))

{

printf("Read File Error:%d\n",GetLastError());

return FALSE;

}

printf("Write Complete!\n");

return TRUE;

}

# 实验题目：实验十一 文件管理之文件读写

**完成人：**XXX

**报告日期：**2018年6月16日

1. 实验内容简要描述

1）创建文件：源文件source1.txt、source2.txt、source3.txt和目标文件dest1.txt、dest2.txt、dest3.txt。文件source1.txt、source2.txt和source3.txt内容完全相同。

2）创建三个线程，每个线程分别实现：

①从文件source1.txt中读取内容并写到文件dest1.txt中，采用无缓冲方法实现文件读/写。

②从文件source2.txt中读取内容并写到文件dest2.txt中，采用高速缓存实现文件读/写。

③从文件source3.txt中读取内容并写到文件dest3.txt中，采用异步方式实现文件读/写。

3）比较三种方式下文件读写速度（消耗时间）。

1. 程序设计
2. 设计思路

（1）创建文件

（2）读文件

（3）写文件

（4）关闭文件句柄

（5）获取系统时间

（6）无缓冲方式

（7）高速缓冲方式

（8）异步方式

（9）几种IO方式的比较

1. 主要数据结构

HANDLE CreateFile（

LPCTSTR lpFileName，//指向文件名指针

DWORD dwDesiredAccess，//读/写访问模式

DWORD dwShareMode，//共享模式

LPSECURITY\_ATTRIBUTES lpSecurityAttributes，//指向安全属性的指针

DWORD dwCreationDisposition，//文件存在标志

DWORD dwFlagsAndAttributes，//文件属性

HANDLE hTemplateFile//指向访问模板文件的句柄

）

BOOL ReadFile（

HANDLE handle， //要读的文件的句柄

LPVOID lpBuffer，//指向文件缓冲区的指针

DWORD nNumberOfBytesToRead，//从文件中要读取的字节数

LPWORD lpNumberOfBytesRead，//指向从文件中要读取的字节数的指针

LPOVERLAPPED lpOverlapped //指向OVERLAPPED结构的指针

）

BOOL WriteFile（

HANDLE handle，//要写的文件的句柄

LPVOID lpBuffer，//指向文件缓冲区的指针

DWORD nNumberOfBytesToWrite， //向文件中要写入的字节数

LPWORD lpNumberOfBytesWritten， //向文件中写入的字节数的指针

LPOVERLAPPED lpOverlapped //指向OVERLAPPED结构的指针

）

BOOL CloseHandle（

HANDLE hObject //要关闭对象的句柄

）

DWORD GetTickCount（VOID）

函数GetTickCount（）的使用方法：

假设在程序中需要测试函数test()的执行时间，那么可以如下测试：

DWORD test\_start\_time= GetTickCount(); //时间单位为ms。

Test();

DWORD test\_end\_time= GetTickCount();

DWORD test\_all\_time=test\_end\_time-test\_start\_time;

typedef struct \_OVERLAPPED {

DWORD Internal; // 系统保留，存放系统设置的状态

DWORD InternalHigh; // 系统保留，存放被传输数据的长度

DWORD Offset;  // 指定文件读或写的开始位置低32位，文件位置是相对文件开始处的字节偏移量。

DWORD OffsetHigh;   // 指定文件读或写的开始位置高32位

HANDLE hEvent;     // 标识事件，数据传送完成时把它设为信号状态

}OVERLAPPED

BOOL GetlappedResult（

HANDLE hFile， //文件、命名管道或通信设备的句柄

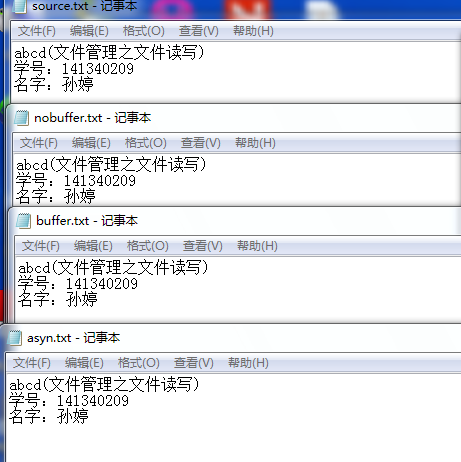
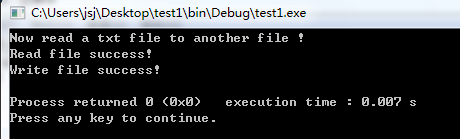
LPOVERLAPPED lpOverlapped， //指向OVERLAPPED结构的句柄

LPDWORD lpNumberOfBytesTransferred，//指向实际传输字节数的指针

BOOL bWait //等待标志

）

三、实验结果



四、实验体会

1、实验体会和收获

通过本次试验，让我加深对Windows的文件管理方法的理解，学会采用不同的方式实现文件读写。通过对无缓冲、有缓冲和异步三种方式实现文件操作的比较，进一步理解操作系统有关文件系统I/O的概念，为选择文件I/O方式提供依据。

五、源代码

1.采用无缓冲方式实现文件读写

#include <windows.h>

#include <iostream.h>

void FileRW\_NoBuffer(const char \* source, const char \* destination);

void main()

{

cout<<"Now read a txt file to another file !"<<endl;

FileRW\_NoBuffer("source.txt", "nobuffer.txt");

}

void FileRW\_NoBuffer(const char \* source, const char \* destination)

{

HANDLE hSource;

HANDLE hDest;

DWORD dwRead;

DWORD dwWrite;

char buf[1024];

hSource = CreateFile(source,

GENERIC\_READ,

0,

NULL,

OPEN\_EXISTING,

FILE\_FLAG\_NO\_BUFFERING, // 指定无缓冲方式进行文件读写

NULL);

if(INVALID\_HANDLE\_VALUE == hSource)

{

cout<<"Could not open the source file!"<<endl;

return;

}

hDest = CreateFile(destination,

GENERIC\_WRITE,

0,

NULL,

CREATE\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

if(INVALID\_HANDLE\_VALUE == hDest)

{

cout<<"Could not create the destination file!"<<endl;

return;

}

if(!ReadFile(hSource, buf, 1024, &dwRead, NULL))

{

cout<<"Read source file error !"<<endl;

return;

}

else

cout<<"Read file success!"<<endl;

if(dwRead == 1024)

{

cout<<"你的文件可能被截断，请增加缓冲区大小！"<<endl;

}

if(!WriteFile(hDest, buf, dwRead, &dwWrite, NULL))

{

cout<<"Read source file error !"<<endl;

return;

}

else

cout<<"Write file success!"<<endl;

CloseHandle(hSource);

CloseHandle(hDest);

}

2. 采用缓冲方式实现文件读写

#include <windows.h>

#include <iostream.h>

void FileRW\_Buffer(const char \* source, const char \* destination);

void main()

{

cout<<"Now read a txt file to another file !"<<endl;

FileRW\_Buffer("source.txt", "buffer.txt");

}

void FileRW\_Buffer(const char \* source, const char \* destination)

{

HANDLE hSource;

HANDLE hDest;

DWORD dwRead;

DWORD dwWrite;

char buf[1024];

hSource = CreateFile(source,

GENERIC\_READ,

0,

NULL,

OPEN\_EXISTING,

FILE\_FLAG\_SEQUENTIAL\_SCAN, // 指定高速缓冲方式进行文件读写

NULL);

if(INVALID\_HANDLE\_VALUE == hSource)

{

cout<<"Could not open the source file!"<<endl;

return;

}

hDest = CreateFile(destination,

GENERIC\_WRITE,

0,

NULL,

CREATE\_ALWAYS,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

if(INVALID\_HANDLE\_VALUE == hDest)

{

cout<<"Could not create the destination file!"<<endl;

return;

}

if(!ReadFile(hSource, buf, 1024, &dwRead, NULL))

{

cout<<"Read source file error !"<<endl;

return;

}

else

cout<<"Read file success!"<<endl;

if(dwRead == 1024)

{

cout<<"你的文件可能被截断，请增加缓冲区大小！"<<endl;

}

if(!WriteFile(hDest, buf, dwRead, &dwWrite, NULL))

{

cout<<"Read source file error !"<<endl;

return;

}

else

cout<<"Write file success!"<<endl;

CloseHandle(hSource);

CloseHandle(hDest);

}

3. 采用异步方式实现文件读写

#include <windows.h>

#include <iostream.h>

void FileRW\_Asyn(const char \* source, const char \* destination);

void main()

{

cout<<"Now read a txt file to another file !"<<endl;

FileRW\_Asyn("source.txt", "asyn.txt");

}

void FileRW\_Asyn(const char \* source, const char \* destination)

{

HANDLE hSource;

HANDLE hDest;

DWORD dwRead;

DWORD dwWrite;

char buf[1024];

hSource = CreateFile(source,

GENERIC\_READ,

0,

NULL,

OPEN\_EXISTING,

FILE\_FLAG\_NO\_BUFFERING | FILE\_FLAG\_OVERLAPPED, // 指定异步方式NULL); //进行文件读写

if(INVALID\_HANDLE\_VALUE == hSource)

{

cout<<"Could not open the source file!"<<endl;

return;

}

hDest = CreateFile(destination,

GENERIC\_WRITE,

0,

NULL,

CREATE\_ALWAYS,

FILE\_FLAG\_NO\_BUFFERING | FILE\_FLAG\_OVERLAPPED,

NULL);

if(INVALID\_HANDLE\_VALUE == hDest)

{

cout<<"Could not create the destination file!"<<endl;

return;

}

OVERLAPPED ovlap;

ovlap.Offset = -1024;

ZeroMemory(&ovlap, sizeof(OVERLAPPED));

if(!ReadFile(hSource, buf, 1024, NULL, &ovlap))

{

if(ERROR\_IO\_PENDING != GetLastError())

{

cout<<"Read source file error !"<<endl;

return;

}

}

cout<<"Read file success!"<<endl;

dwRead = 1024;

if(!WriteFile(hDest, buf, dwRead, &dwWrite, &ovlap))

{

cout<<"Read source file error !"<<endl;

return;

}

else

cout<<"Write file success!"<<endl;

CloseHandle(hSource);

CloseHandle(hDest);

}