# 操作系统实验 进程同步与互斥

## 实验目的

1. 掌握进程同步和互斥原理，理解生产者-消费者模型；
2. 学习Windows2000/xp中的多线程并发执行机制；
3. 学习使用Windows SDK解决读者－写者问题。

## 试验内容1

依据生产者－消费者模型，在Windows环境下创建一个控制台进程，在该进程中创建n个线程模拟生产者和消费者，实现进程(线程)的同步与互斥，分析、熟悉生产者消费者问题仿真的原理和实现技术。

**实验要求**

有两个进程：一组生产者进程和一组消费者进程共享一个初始为空、固定大小为n的缓存（缓冲区）。生产者的工作是制造一段数据，只有缓冲区没满时，生产者才能把消息放入到缓冲区，否则必须等待，如此反复; 同时，只有缓冲区不空时，消费者才能从中取出消息，一次消费一段数据（即将其从缓存中移出），否则必须等待。由于缓冲区是临界资源，它只允许一个生产者放入消息，或者一个消费者从中取出消息。

**实验分析（原理和技术）**

对于生产者，如果缓存是满的就去等待。消费者从缓存中取走数据后就叫醒生产者，让它再次将缓存填满。若消费者发现缓存是空的，就等待。下一轮中生产者将数据写入后就叫醒消费者。  
 只有生产者和消费者两个进程，正好是这两个进程存在着互斥关系和同步关系。那么需要解决的是互斥和同步PV操作的位置。使用“进程间通信”，“信号标”semaphore就可以解决唤醒的问题。我们使用了两个信号标：sem\_full 和 sem\_avail 。信号量sem\_mutex作为互斥信号量，它用于控制互斥访问缓冲池，互斥信号量初值为 1；信号量 sem\_full 用于记录当前缓冲池中“满”缓冲区数，初值为0。信号量 sem\_avail 用于记录当前缓冲池中“空”缓冲区数，初值为3。新的数据添加到缓存中后，sem\_full 在增加，而 sem\_avail 则减少。如果生产者试图在 sem\_avail 为0时减少其值，生产者就会被“催眠”。下一轮中有数据被消费掉时，sem\_avail就会增加，生产者就会被“唤醒”。

**实验结果**

**实验输入**

3

1 P 1 1

2 P 2 1

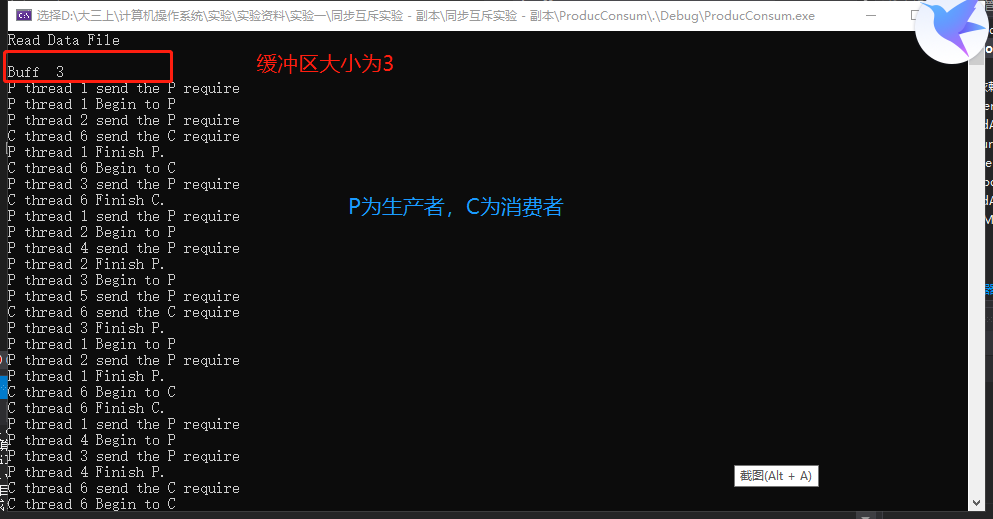
3 P 3 1

4 P 4 1

5 P 5 1

6 C 2 1

**实验输出结果**

****

**实验代码（含详细注释）**

// ProducConsum.cpp : Defines the entry point for the console application.

#include "stdafx.h"

#include "windows.h"

#include <conio.h>

#include <stdlib.h>

//#include <fstream.h>

#include <fstream>

#include <io.h>

#include <string.h>

#include <stdio.h>

#define MSC\_VER

#define INTE\_PER\_SEC 100

#define MAX\_THREAD\_NUM 64 //最大的线程量

#define SEM\_MAX\_FULL 64 //信号量的最大值

#define PRODUC 'P'

#define CONSUM 'C'

using namespace std;

struct ThreadInfo

{

int serial;//线程序号

char entity;//线程类别

double delay;//线程把产品加入到缓冲区/持续时间

double persist;//生产产品/延续时间

};

int buff\_num;

CRITICAL\_SECTION sem\_mutex;

HANDLE sem\_full;

HANDLE sem\_avail;

void ProducConsum(char \*file);

void Thread\_Producter(void \*p);

void Thread\_Consumer(void \*p);

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

{

ProducConsum("pc\_data.txt");

return 0;

}

//3

//1 P 1 1

//2 P 2 1

//3 P 3 1

//4 P 4 1

//5 P 5 1

//6 C 2 1

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

void ProducConsum(char \*file)

{

DWORD n\_thread = 0;

DWORD thread\_ID ;

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

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

ifstream inFile;

inFile.open(file);

puts("Read Data File \n");

inFile >> buff\_num;

while(inFile)

{

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

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

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

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

n\_thread++;

inFile.get();

}

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

{

if(thread\_info[i].entity == PRODUC)

//CreateThread完成线程创建，在调用进程的地址空间上创建一个线程，

//以执行指定的函数；它的返回值为所创建线程的句柄。

h\_Thread[i] = CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)( Thread\_Producter),

&thread\_info[i],0,&thread\_ID);//判断为生产者

else

{

if(thread\_info[i].entity == CONSUM)//判断为消费者

h\_Thread[i] = CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)( Thread\_Consumer),

&thread\_info[i],0,&thread\_ID);

else

{

puts("Bad File\n");

exit(0);

}

}

}

printf("Buff %d\n",buff\_num);

InitializeCriticalSection(&sem\_mutex);

sem\_full = CreateSemaphore(NULL,0 ,SEM\_MAX\_FULL,"sem\_full");//创建一个数值为64的信号量，用于同步

sem\_avail = CreateSemaphore(NULL,buff\_num,SEM\_MAX\_FULL,"sem\_avail");

//HANDLE CreateSemaphore(

//LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes, // SD

// LONG lInitialCount, // initial count

// LONG lMaximumCount, // maximum count

// LPCTSTR lpName // object name

//)

// lpSemaphoreAttributes：为信号量的属性，一般可以设置为NULL

//lInitialCount：信号量初始值，必须大于等于0，而且小于等于 lpMaximumCount，如果lInitialCount 的初始值为0，则该信号量默认为unsignal状态，如果lInitialCount的初始值大于0，则该信号量默认为signal状态，

// lMaximumCount： 此值为设置信号量的最大值，必须大于0

// lpName：信号量的名字，长度不能超出MAX\_PATH ，可设置为NULL，表示无名的信号量。当lpName不为空时，可创建有名的信号量，若当前信号量名与已存在的信号量的名字相同时，则该函数表示打开该信号量，这时参数lInitialCount 和

// lMaximumCount 将被忽略。

WaitForMultipleObjects(n\_thread,h\_Thread,TRUE,-1);//可在指定时间内等待指定对象为可用状态

printf("Task is Finished!\n");

getch();

}

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

void Thread\_Producter(void \*p)

{

DWORD m\_delay;

DWORD m\_persist;

int m\_serial;

//读参数

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

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

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

while (true)

{

Sleep(m\_delay);//把产品放到缓冲区

printf("P thread %d send the P require\n",m\_serial);

WaitForSingleObject(sem\_avail,INFINITE); //等待资源信号量empty

EnterCriticalSection(&sem\_mutex);//等待共享资源

printf("P thread %d Begin to P\n",m\_serial);

Sleep(m\_persist);//生产产品

printf("P thread %d Finish P.\n",m\_serial);

ReleaseSemaphore(sem\_full,1,NULL); //释放资源信号量full

LeaveCriticalSection(&sem\_mutex);//释放共享资源

}

}

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

void Thread\_Consumer(void \*p)

{

DWORD m\_delay;

DWORD m\_persist;

int m\_serial;

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

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

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

while (true)

{

Sleep(m\_delay);//延续时间

printf("C thread %d send the C require\n",m\_serial);

WaitForSingleObject(sem\_full,INFINITE); //等待信号量full

EnterCriticalSection(&sem\_mutex);//等待共享资源

printf("C thread %d Begin to C\n",m\_serial);

Sleep(m\_persist);//持续时间

printf("C thread %d Finish C.\n",m\_serial);

ReleaseSemaphore(sem\_avail,1,NULL); //释放信号量empty

LeaveCriticalSection(&sem\_mutex);//释放共享资源

}

}

## 试验内容2

参考实验内容1和附件2伪码，编程解决读者－写者问题的程序。（具体要求和读写者问题原始伪码内容见附件1）

**实验要求**

有两组并发进程：读者和写者，共享一个文件F，要求：

(1)允许多个读者可同时对文件执行读操作；

(2)只允许一个写者往文件中写信息；

(3)任一写者在完成写操作之前不允许其他读者或写者工作；

(4)写者执行写操作前，应需已有的写者和读者全部退出。

(5) 要求仿真程序产生3个读者进程，两个写者进程，读写者都周期性地产生读写要求，读写操作要持续一定时间。

**实验分析（原理和技术）**

1) 关系分析。由题目分析读者和写者是互斥的，写者和写者也是互斥的，而读者和读者不存在互斥问题。

2) 整理思路。两个进程，即读者和写者。写者是比较简单的，它和任何进程互斥，用互斥信号量的P操作、V操作即可解决。读者的问题比较复杂，它必须实现与写者互斥的同时还要实现与其他读者的同步，因此，仅仅简单的一对P操作、V操作是无法解决的。那么，在这里用到了一个计数器，用它来判断当前是否有读者读文件。当有读者的时候写者是无法写文件的，此时读者会一直占用文件，当没有读者的时候写者才可以写文件。同时这里不同读者对计数器的访问也应该是互斥的。

3) 信号量设置。首先设置信号量readcount为计数器，用来记录当前读者数量，初值为0; 设置w\_mutex为互斥信号量，用于保护更新readcount变量时的互斥；设置互斥信号量rw用于保证读者和写者的互斥访问。

**实验结果**

**实验输入**

1 R 1 1

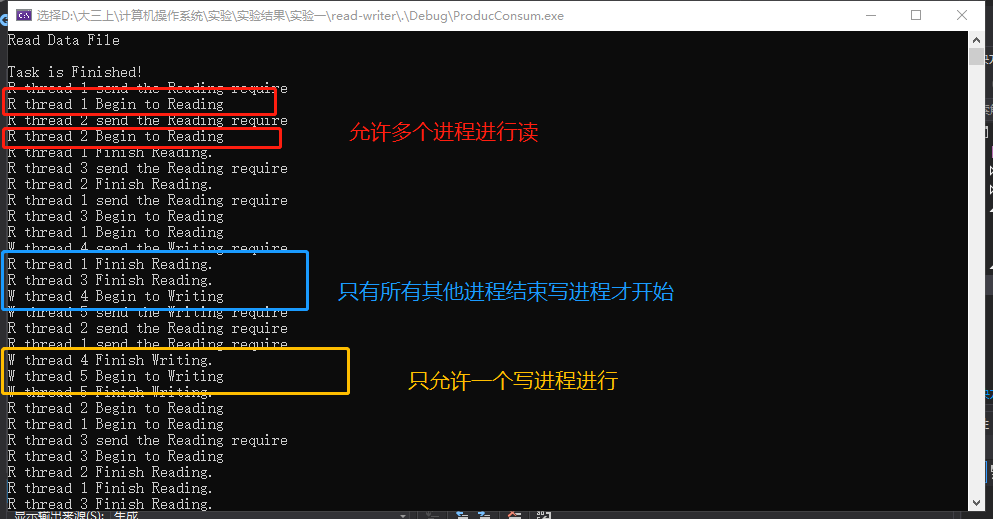
2 R 2 1

3 R 3 1

4 W 4 1

5 W 5 1

**实验输出结果**

****

**实验伪码**

int read\_count ;

semaphore mutex, w;

read\_count= 0; /\* 读进程计数 \*/

W = 1;

mutex = 1;

Reader

{

P(mutex);

read\_count= read\_count + 1;

if (read\_count==1) then P(W)；

V(mutex);

读文件；

P(mutex);

read\_count = read\_count - 1;

if (read\_count == 0) then V(W)；

V(mutex);

}

write()

{

P(W);

写文件;

V(W);

}

**实验原代码（含详细注释）**

// ReadWrite.cpp : Defines the entry point for the console application.

#include "stdafx.h"

#include "windows.h"

#include <conio.h>

#include <stdlib.h>

//#include <fstream.h>

#include <fstream>

#include <io.h>

#include <string.h>

#include <stdio.h>

#define MSC\_VER

#define INTE\_PER\_SEC 100

#define MAX\_THREAD\_NUM 64 //最大的线程量

#define SEM\_MAX\_FULL 64 //信号量的最大值

#define Read 'R'

#define Write 'W'

using namespace std;

struct ThreadInfo

{

int serial;//线程序号

char entity;//线程类别

double delay;//延迟时间

double persist;//读写持续时间

};

//int buff\_num;

int readcount = 0;

//CRITICAL\_SECTION r\_mutex;

CRITICAL\_SECTION w\_mutex;

//HANDLE sem\_full;

//HANDLE sem\_avail;

HANDLE r\_mutex;

void ReadWrite(char \*file);

void Thread\_Read(void \*p);

void Thread\_Write(void \*p);

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

{

ReadWrite("pc\_data.txt");

return 0;

}

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

void ReadWrite(char \*file)

{

DWORD n\_thread = 0;

DWORD thread\_ID ;

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

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

ifstream inFile;

inFile.open(file);

puts("Read Data File \n");

//inFile >> buff\_num;

while(inFile)

{

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

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

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

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

n\_thread++;

inFile.get();

}

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

{

if(thread\_info[i].entity == Read)

//CreateThread完成线程创建，在调用进程的地址空间上创建一个线程，

//以执行指定的函数；它的返回值为所创建线程的句柄。

h\_Thread[i] = CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)( Thread\_Read),

&thread\_info[i],0,&thread\_ID);//判断为读

else

{

if(thread\_info[i].entity == Write)//判断为写

h\_Thread[i] = CreateThread(NULL,0,(LPTHREAD\_START\_ROUTINE)( Thread\_Write),

&thread\_info[i],0,&thread\_ID);

else

{

puts("Bad File\n");

exit(0);

}

}

}

//printf("Buff %d\n",buff\_num);

//InitializeCriticalSection(&r\_mutex);

InitializeCriticalSection(&w\_mutex);

r\_mutex = CreateSemaphore(NULL, 1, SEM\_MAX\_FULL, "r\_mutex");

//sem\_full = CreateSemaphore(NULL,0 ,SEM\_MAX\_FULL,"sem\_full");//创建一个数值为64的信号量，用于同步

//sem\_avail = CreateSemaphore(NULL,buff\_num,SEM\_MAX\_FULL,"sem\_avail");

//HANDLE CreateSemaphore(

//LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes, // SD

// LONG lInitialCount, // initial count

// LONG lMaximumCount, // maximum count

// LPCTSTR lpName // object name

//)

// lpSemaphoreAttributes：为信号量的属性，一般可以设置为NULL

//lInitialCount：信号量初始值，必须大于等于0，而且小于等于 lpMaximumCount，如果lInitialCount 的初始值为0，则该信号量默认为unsignal状态，如果lInitialCount的初始值大于0，则该信号量默认为signal状态，

// lMaximumCount： 此值为设置信号量的最大值，必须大于0

// lpName：信号量的名字，长度不能超出MAX\_PATH ，可设置为NULL，表示无名的信号量。当lpName不为空时，可创建有名的信号量，若当前信号量名与已存在的信号量的名字相同时，则该函数表示打开该信号量，这时参数lInitialCount 和

// lMaximumCount 将被忽略。

//WaitForMultipleObjects(n\_thread,h\_Thread,TRUE,-1);//可在指定时间内等待指定对象为可用状态

printf("Task is Finished!\n");

getch();

}

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

void Thread\_Read(void \*p)

{

DWORD m\_delay;

DWORD m\_persist;

int m\_serial;

//读参数

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

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

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

while (true)

{

Sleep(m\_delay);//延续时间

printf("R thread %d send the Reading require\n", m\_serial);

WaitForSingleObject(r\_mutex,-1); //等待读

//EnterCriticalSection(&r\_mutex);

readcount++;

if (readcount == 1)

{

EnterCriticalSection(&w\_mutex);//互斥信号量

}

ReleaseSemaphore(r\_mutex, 1, NULL);

printf("R thread %d Begin to Reading\n", m\_serial);

Sleep(m\_persist);//读操作

printf("R thread %d Finish Reading.\n",m\_serial);

WaitForSingleObject(r\_mutex, -1);

readcount--;

if (readcount == 0)

{

LeaveCriticalSection(&w\_mutex);

}

ReleaseSemaphore(r\_mutex, 1, NULL);//释放读信号

}

}

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

void Thread\_Write(void \*p)

{

DWORD m\_delay;

DWORD m\_persist;

int m\_serial;

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

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

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

while (true)

{

Sleep(m\_delay);//延续时间

printf("W thread %d send the Writing require\n",m\_serial);

EnterCriticalSection(&w\_mutex);//等待共享资源

printf("W thread %d Begin to Writing\n",m\_serial);

Sleep(m\_persist);//持续时间

printf("W thread %d Finish Writing.\n",m\_serial);

LeaveCriticalSection(&w\_mutex);//释放互斥量

}

}

## 相关知识

### Windows 的线程控制

CreateThread完成线程创建，在调用进程的地址空间上创建一个线程，以执行指定的函数；它的返回值为所创建线程的句柄。

ExitThread用于结束当前线程。

SuspendThread可挂起指定的线程。

ResumeThread可激活指定线程，它的对应操作是递减指定线程的挂起计数，当挂起计数减为0时，线程恢复执行。

### Windows 的进程互斥和同步

在Windows 2000/XP中提供了临界区、互斥对象、信号量对象同步对象和相应的系统调用，用于进程和线程同步。

### 临界区对象(Critical Section)

只能用于在同一进程内使用的临界区，同一进程内各线程对它的访问是互斥进行的。相关API包括：

InitializeCriticalSection对临界区对象进行初始化；

EnterCriticalSection等待占用临界区的使用权，得到使用权时返回；

TryEnterCriticalSection非等待方式申请临界区的使用权；申请失败时，返回0；

LeaveCriticalSection释放临界区的使用权；

DeleteCriticalSection释放与临界区对象相关的所有系统资源。

### 互斥对象(Mutex)

互斥对象相当于互斥信号量，在一个时刻只能被一个线程使用。有关的API：

CreateMutex创建一个互斥对象，返回对象句柄；

OpenMutex返回一个已存在的互斥对象的句柄，用于后续访问；

ReleaseMutex释放对互斥对象的占用，使之成为可用；

### 信号量对象(Semaphore)

信号量对象的取值在0到指定最大值之间，用于限制并发访问的线程数。有关的API：

CreateSemaphore创建一个信号量对象，指定最大值和初值，返回对象句柄；

OpenSemaphore返回一个已存在的信号量对象的句柄，用于后续访问；

ReleaseSemaphore释放对信号量对象的占用；

### 同步对象等待

对于这些同步对象，Windows 2000/XP提供了两个统一的等待操作WaitForSingleObject和WaitForMultipleObjects。

(1) WaitForSingleObject在指定的时间内等待指定对象为可用状态(signaled state)；

DWORD WaitForSingleObject( HANDLE hHandle,

// handle of object to wait for

DWORD dwMilliseconds

// time-out interval in milliseconds

(2) WaitForMultipleObjects在指定的时间内等待多个对象为可用状态；

DWORD WaitForMultipleObjects( DWORD nCount,

//对象句柄数组中的句柄数；

CONST HANDLE \*lpHandles,

// 指向对象句柄数组的指针，数组中可包括多种对象句柄；

BOOL bWaitAll,

// 等待标志：TRUE表示所有对象同时可用，FALSE表示至少一个对象可用；

DWORD dwMilliseconds // 等待超时时限；

);