# 实验一 读者与写者问题

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。

1. 读写者问题基本思路

读写互斥问题是经典的进程同步问题，可以采用记录型信号量来解决读者-写者问题。为实现读写互斥和写写互斥设置一个互斥信号量W\_mutex，设置一个整型变量readcount表示正在读的进程数目。当readcount=1时，执行wait(W\_mutex)操作，阻塞所读进程结束前的写进程请求。当readcount减为0后，执行signal(W\_mutex)操作，以便让写进程开始写操作。另外，readcount是一个可被多个读进程访问的临界资源，也应为其设置互斥信号量（mutex）。

## 三、读写者问题原始算法伪码

引入计数器read\_count对读进程计数；

mutex 是用于对计数器read\_count操作的互斥信号量；

W 表示是否允许写的信号量。

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);

}

1. 实验结果



五、代码实现

#include "stdafx.h"

#include "windows.h"

#include <conio.h>

#include <stdlib.h>

#include <fstream>

#include <io.h>

#include <string.h>

#include <stdio.h>

#include<iostream>

#define INTE\_PER\_SEC 100

#define MAX\_THREAD\_NUM 64

#define MAX\_FILE\_NUM 10

#define READER 'R'

#define WRITER 'W'

#define SEM\_MAX\_FULL 64

using namespace std;

struct ThreadInfo//线程信息

{

int serial;//线程编号

char entity;//线程类型

double delay;//线程转移时间

double persist;//线程执行时间

};

int buff\_num;

CRITICAL\_SECTION mutex; //读进程临界区

HANDLE W\_mutex;//读写互斥，写写互斥。

int read\_count=0;//读者个数

void ReaderAndWriter(char \*file);

void Thread\_Reader(void \*p);

void Thread\_Writer(void \*p);

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

{

ReaderAndWriter("rw\_data.txt");

return 0;

}

void ReaderAndWriter(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");

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)-1; i++)

{

if (thread\_info[i].entity == READER)//创建读进程

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

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

else

{

if (thread\_info[i].entity == WRITER)//创建写进程

h\_Thread[i] = CreateThread(NULL, 0,

(LPTHREAD\_START\_ROUTINE)(Thread\_Writer),

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

else

{

puts("Bad File\n");

exit(0);

}

}

}

//初始化信号量和临界区

InitializeCriticalSection(&mutex);

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

WaitForMultipleObjects(n\_thread, h\_Thread, TRUE, -1);//所有线程终止后再向下执行

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

\_getch();

}

void Thread\_Reader(void \*p)

{

int m\_serial;

DWORD m\_delay;

DWORD m\_persist;

DWORD wait\_for\_mutex;

//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)

{

printf("R thread %d delay %d \n", m\_serial, m\_delay);

Sleep(m\_delay);

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

EnterCriticalSection(&mutex);//进临界区

read\_count++;

if (read\_count == 1)

{

WaitForSingleObject(W\_mutex, INFINITE);//wait(W\_mutex)，获取信号量,此时不可写

}

LeaveCriticalSection(&mutex);//出临界区

////////////////////////////读文件/////////////////////////////////////////

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

printf("R thread %d persist %d \n", m\_serial, m\_persist);

Sleep(m\_persist);

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

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

EnterCriticalSection(&mutex);//进临界区

read\_count--;

if (read\_count == 0)

{

ReleaseSemaphore(W\_mutex, 1, NULL);//signal(mutex),释放信号量，可写

} }

LeaveCriticalSection(&mutex);//出临界区

}

}

void Thread\_Writer(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)

{

printf("W thread %d delay %d \n", m\_serial, m\_delay);

Sleep(m\_delay);

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

WaitForSingleObject(W\_mutex, INFINITE);//获取信号量,此时读进程不可读，其他写进程不可写

/////////////////////////写文件//////////////////////////////////////////

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

printf("W thread %d persist %d \n", m\_serial, m\_persist);

Sleep(m\_persist);

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

////////////////////////////////////////////////////////////////////////// ReleaseSemaphore(W\_mutex, 1, NULL);//释放信号量,读进程可读，其他写进程可写

}

}