

操作系统实验二:生产者——消费者问题

姓名:魏远卓 学号:71113211 提交日期:2015.5.17

【实验内容】

1. 在 Windows 操作系统上,利用 Win32 API 提供的信号量机制,编写应用程序实现生产者——消费者问题。
2. 在 Linux 操作系统上,利用 Pthread API 提供的信号量机制,编写应用程序实现生产者——消费者问题。
3. 两种环境下,生产者和消费者均作为独立线程,并通过 empty、full、mutex 三个信号量实现对缓冲进行插入与删除。
4. 通过打印缓冲区中的内容至屏幕,来验证应用程序的正确性。具体内容可参见“Operating System Concepts (Seventh Edition)” Chapter 6 后的 Project(P236-241)。

【实验目的】

通过实验,掌握 Windows 和 Linux 环境下互斥锁和信号量的实现方法,加深对临界区问题和进程同步机制的理解,同时熟悉利用 Windows API 和 Pthread API 进行多线程编程的方法。

【实验过程】

linux下:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>
#define BUFFER_SIZE 8
char *buffer;
sem_t mutex,empty,full;//mutex为互斥信号量,用于互斥的访问buffer; empty和full
为资源信号量,分别用于记录缓冲区空和满的数量
int producer_count,consumer_count;//生产者和消费者在数组中的下标
```

```

void output()
{
    for(int i=0;i<BUFFER_SIZE;i++)
    {
        printf("%c",buffer[i]);
        printf(" ");
    }
    printf("\n");
}
void *producer(void *ptr)
{
    int j=0;
    do{
        sem_wait(&empty);//buffer有空余，可以生产，并减一
        sem_wait(&mutex);//互斥访问，只能一个线程消费
        printf("%lu%s%d%s\t",pthread_self(),"#####",j,"#####");
        buffer[(producer_count++)%BUFFER_SIZE]='1';//生产者线程赋值为1;
        output();
        j++;
        sem_post(&mutex);
        sem_post(&full);//生产完毕增加一个可消费的量
    }while(j!=4);//每个线程可以做4次
}
void *consumer(void *ptr)
{
    int j=0;
    do{
        sem_wait(&full);//可以消费的量减一
        sem_wait(&mutex);
        printf("%lu%s%d%s\t",pthread_self(),"*****",j,"*****");
        buffer[(consumer_count++)%BUFFER_SIZE]='0';//消费者线程赋值为0;
        output();
        j++;
        sem_post(&mutex);
        sem_post(&empty);
    }while(j!=4);
}
int main()
{
    producer_count=0;
    consumer_count=0;
    buffer=(char*)malloc(BUFFER_SIZE*sizeof(char*));
    for(int i=0;i<BUFFER_SIZE;i++)
        buffer[i]='0';
    //信号量赋初值
    sem_init(&mutex,1,1);
    sem_init(&empty,0,BUFFER_SIZE);
    sem_init(&full,0,0);//初始化信号量
    //多线程
    pthread_t tid[10];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    for(int i=0;i<5;i++)
    {
        pthread_create(&tid[i],&attr,consumer,NULL);
    }
}

```

```

    pthread_create(&tid[i+5], &attr, producer, NULL);
}
for(int j=0; j<10; j++)
{
    pthread_join(tid[j], NULL);
}
return 0;
}

```

```

lebi@lebi-inspiron-5520 ~
300730880-00000000000000000000 1 0 0 0 0 0 0
300730880-00000000000000000000 1 1 0 0 0 0 0
301695264-00000000000000000000 1 1 1 0 0 0 0
301695264-00000000000000000000 1 1 1 1 0 0 0
3050523456-00000000000000000000 1 1 1 1 1 0 0
300730880-00000000000000000000 1 1 1 1 1 0 0
303373808-00000000000000000000 1 1 1 1 1 1 0
3075701568-00000000000000000000 0 1 1 1 1 1 0
3075701568-00000000000000000000 0 0 1 1 1 1 0
3023345344-00000000000000000000 0 0 0 1 1 1 0
3058916160-00000000000000000000 0 0 0 0 1 1 1
303373808-00000000000000000000 0 0 0 0 1 1 1
3008559936-00000000000000000000 0 0 0 0 0 1 1
3000167232-00000000000000000000 1 0 0 0 0 1 1
3075701568-00000000000000000000 1 0 0 0 0 1 1
301695264-00000000000000000000 1 1 0 0 0 0 1
3050523456-00000000000000000000 1 1 1 0 0 0 1
300730880-00000000000000000000 1 1 1 1 0 0 1
3042130752-00000000000000000000 1 1 1 1 0 0 1
3042130752-00000000000000000000 1 1 1 1 0 0 0
303373808-00000000000000000000 1 1 1 1 1 0 0
3042130752-00000000000000000000 0 1 1 1 1 0 0
3000167232-00000000000000000000 0 1 1 1 1 0 0
3042130752-00000000000000000000 0 0 1 1 1 0 0
3075701568-00000000000000000000 0 0 0 1 1 1 0
3023345344-00000000000000000000 0 0 0 0 1 1 0
301695264-00000000000000000000 0 0 0 0 1 1 0
3050523456-00000000000000000000 0 0 0 0 1 1 1
3050523456-00000000000000000000 1 0 0 0 1 1 1
3023345344-00000000000000000000 1 0 0 0 0 1 1
3023345344-00000000000000000000 1 0 0 0 0 1 1
3000167232-00000000000000000000 1 1 0 0 0 0 1
3000167232-00000000000000000000 1 1 1 0 0 0 1
303373808-00000000000000000000 1 1 1 1 0 0 1
3008559936-00000000000000000000 1 1 1 1 0 0 1
3008559936-00000000000000000000 1 1 1 1 0 0 0
3058916160-00000000000000000000 0 0 1 1 0 0 0
3058916160-00000000000000000000 0 0 0 1 0 0 0
3058916160-00000000000000000000 0 0 0 0 0 0 0
lebi@lebi-inspiron-5520 ~$

```

windows下:

```

#include <iostream>
#include <windows.h>
HANDLE mutex, empty, full; 设置信号量
int producercount=0;
int consumercount=0;
#define BUFFER_SIZE 8;
char *buffer;
void output()
{
    for(int i=0; i<BUFFER_SIZE; i++)
        cout<<buffer[i];
    cout<<endl;
}
DWORD WINAPI producer(LPVOID param)
{
    int j=0;
    do{
        WaitForSignalObject(empty, INFINITE);

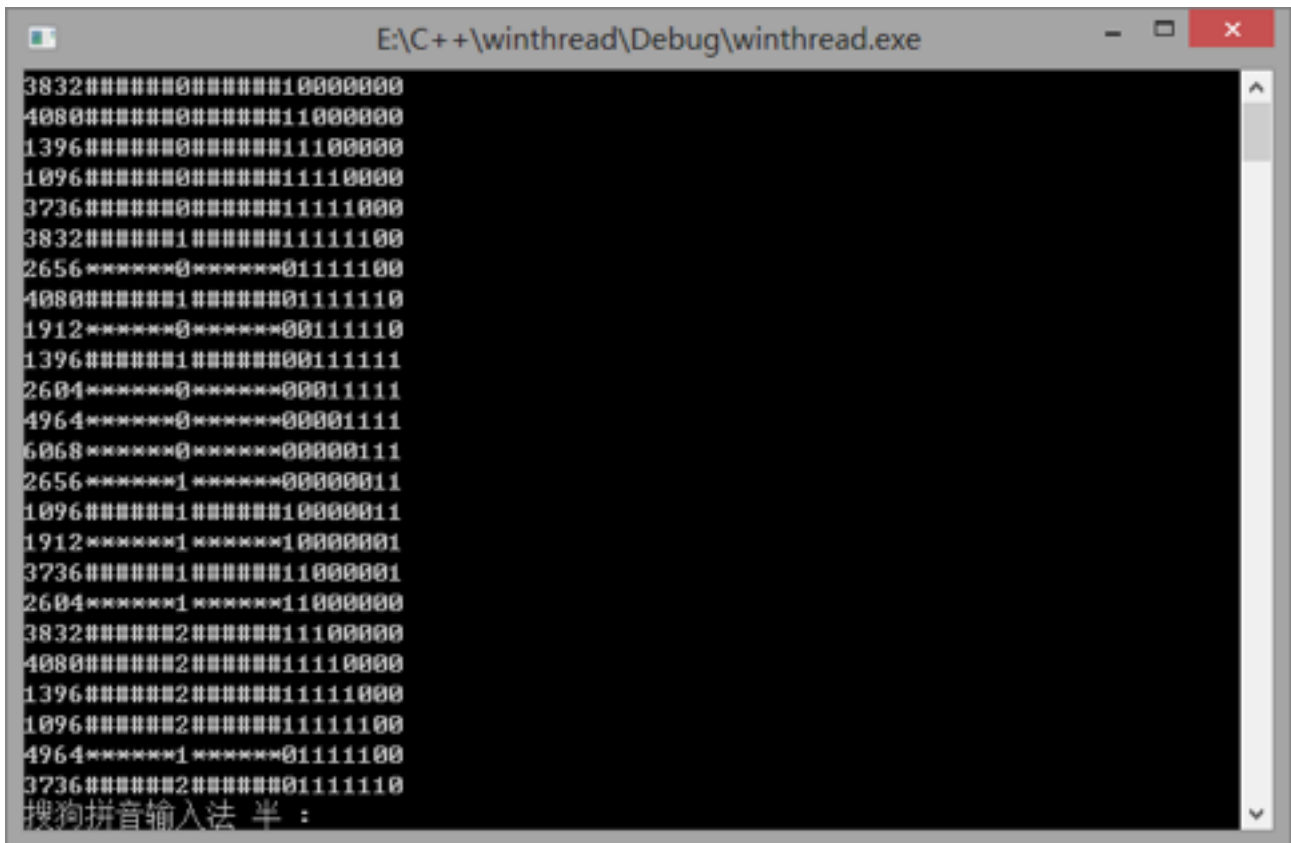
```

```

        WaitForSignalObject(mutex, INFINITE);
        cout<<GetCurrentThreadId()<<"#####"<<j<<"#####";
        buffer[(producercount++)%BUFFER_SIZE]='1';
        output();
        j++;
        ReleaseSemaphore(mutex, 1, NULL);
        ReleaseSemaphore(full, 1, NULL);
    }while(j!=4);
    return 0;
}
DWORD WINAPI consumer(LPVOID param)
{
    int j=0;
    do{
        WaitForSignalObject(full, INFINITE);
        WaitForSignalObject(mutex, INFINITE);
        cout<<GetCurrentThreadId()<<"*****"<<j<<"*****";
        buffer[(consumercount++)%BUFFER_SIZE]='0';
        output();
        j++;
        ReleaseSemaphore(mutex, 1, NULL);
        ReleaseSemaphore(empty, 1, NULL);
    }
    return 0;
}
int main()
{
    empty=CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, NULL);
    full=CreateSemaphore(NULL, 0, BUFFER_SIZE, NULL);
    mutex=CreateSemaphore(NULL, 1, 1, NULL);
    buffer=new char[BUFFER_SIZE];
    for(int i=0; i<BUFFER_SIZE; i++);
        buffer[i]='0';
    DWORD *ThreadId=(DWORD*)malloc(BUFFER_SIZE*sizeof(DWORD));
    HANDLE *ThreadHandle=(HANDLE)malloc(BUFFER_SIZE*sizeof(HANDLE));
    for(int i=0; i<5; i++)
    {
        ThreadHandle[i]=CreateThread(NULL, 0, consumer, NULL, 0, &ThreadId[i]);
        ThreadHandle[i+5]=CreateThread(B=NULL, 0, producer, NULL, 0, &ThreadId[i
+5]);
    }
    return 0;
}

```

}



【心得体会】

用到的函数以及注意事项：

linux：

1.sem_init(sem_t *sem, int pshared, unsigned int value)

pshared=0表示是信号量是由进程内线程共享，pshared=1表示信号量是由进程间线程共享，这里mutex信号量初始化时pshared赋成0或1为看出区别

2.posix（可移植操作系统）线程中的 线程属性pthread_attr_t主要包括scope属性、detach属性、堆栈地址、堆栈大小、优先级。

int pthread_attr_init (pthread_attr_t* attr)

功能：对线程属性变量的初始化。

头文件：#include<pthread.h>

函数传入值：attr:线程属性。

函数返回值：成功：失败： -1

3.pthread_t用来声明线程id

4.int pthread_create ((pthread_t *thread, pthread_attr_t *attr, void * (*start_routine) (void *), void *arg)

头文件：#include <pthread.h>

功能：创建线程（实际上就是确定调用该线程函数的入口点），在线程创建以后，就开始运行相关的线程函数。

说明：thread：线程标识符；

attr：线程属性设置；

start_routine：线程函数的起始地址；

arg：传递给start_routine的参数；

返回值：成功，返回0；出错，返回-1。

5.thread库不是Linux系统默认的库，连接时需要使用静态库libpthread.a，所以编译时要加上 -lpthread

windows:

1.DWORD WINAPI就是一个返回 DWORD（32位数据）的 API 函数

2.LPVOID,是一个没有类型的指针，也就是说你可以将任意类型的指针赋值给LPVOID类型的变量（一般作为参数传递），然后在使用的时候再转换回来

3.DWORD WaitForSingleObject(HANDLE hHandle,DWORD dwMilliseconds)

hHandle[in]对象句柄。可以指定一系列的对象，如Event、Job、Memory resource notification、Mutex、Process、Semaphore、Thread、Waitable timer等。

dwMilliseconds[in]定时时间间隔，单位为milliseconds（毫秒）。如果指定一个非零值，函数处于等待状态直到hHandle标记的对象被触发，或者时间到了。如果dwMilliseconds为0，对象没有被触发信号，函数不会进入一个等待状态，它总是立即返回。如果dwMilliseconds为INFINITE，对象被触发信号后，函数才会返回

4.BOOL ReleaseSemaphore(HANDLE hSemaphore,ULONG lReleaseCount,LPLONG lpPreviousCount);

hSemaphore

[输入参数]所要操作的信号量对象的句柄，这个句柄是CreateSemaphore或者OpenSemaphore函数的返回值。这个句柄必须有SEMAPHORE_MODIFY_STATE 的权限。

lReleaseCount

[输入参数]这个信号量对象在当前基础上所要增加的值，这个值必须大于0，如果信号量加上这个值会导致信号量的当前值大于信号量创建时指定的最大值，那么这个信号量的当前值不变，同时这个函数返回FALSE;

lpPreviousCount

[输出参数]指向返回信号量上次值的变量的指针，如果不需要信号量上次的值，那么这个参数可以设置为NULL;返回值：如果成功返回TRUE,如果失败返回FALSE,可以调用GetLastError函数得到详细出错信息;

5.12.win32API创建线程的函数:

HANDLE WINAPI CreateThread(

LPSECURITY_ATTRIBUTES lpThreadAttributes,

SIZE_T dwStackSize,

LPTHREAD_START_ROUTINE lpStartAddress,

LPVOID lpParameter,

DWORD dwCreationFlags,

LPDWORD lpThreadId

)

第一个参数表示线程内核对象的安全属性，一般传入NULL表示使用默认设置。

第二个参数表示线程栈空间大小。传入0表示使用默认大小（1MB）。

第三个参数表示新线程所执行的线程函数地址，多个线程可以使用同一个函数地址。

第四个参数是传给线程函数的参数。

第五个参数指定额外的标志来控制线程的创建，为0表示线程创建之后立即就可以进行调度，如果为CREATE_SUSPENDED则表示线程创建后暂停运行，这样它就无法调度，直到调用ResumeThread()。

第六个参数将返回线程的ID号，传入NULL表示不需要返回该线程ID号。

函数返回值：

成功返回新线程的句柄，失败返回NULL。

出现的问题及体会：

1. mac osx下sem_init函数调用不成功导致信号量没有被正确初始化所以最后的输出不是预期，但找不到合适的解决方案于是到ubuntu下编译输出

2. 通过打印自定义缓冲区并打印其中内容的形式对多线程加深了理解，发现多线程的运行顺序是不可控的，一个线程要做四次打印，很可能在它还没做完四次时cpu就被另一条线程占用，虽然总行数是固定的四十行，但每次的打印结果都不同（但一定先打印生产者线程，因为信号量的控制）。所以如果要控制多线程，要加各种锁。