**北京科技大学实验报告**

学院： 计通学院 专业： 计算机科学与技术 班级： 计172

姓名： 金玉卿 学号： 41724051 实验日期： 2019 年 12 月 3 日

**实验名称：操作系统实验3 进程同步（5分）**

**实验目的：**以一个教学型操作系统EOS为例，深入理解进程（线程）同步的原理、意义及信号量的含义和实现方法；能对核心源代码进行分析和修改，能运用信号量实现同步问题；训练分析问题、解决问题以及自主学习能力，逐步达到能独立对小型操作系统的功能进行分析、设计和实现。

**实验环境：**EOS操作系统及其实验环境。

**实验内容：**

使用EOS的信号量，实现生产者-消费者问题；跟踪EOS信号量的工作过程，分析EOS信号量实现的源代码，理解并阐述其实现方法；修改EOS信号量的实现代码，使之支持等待超时唤醒和批量释放功能。

**实验步骤：**

**1）使用EOS的信号量实现生产者-消费者问题**

（给出使用EOS的信号量解决生产者-消费者问题的实现方法，包括实现方法的简要描述、源代码、测试及结果等）

* 1. **实现方法简要描述**

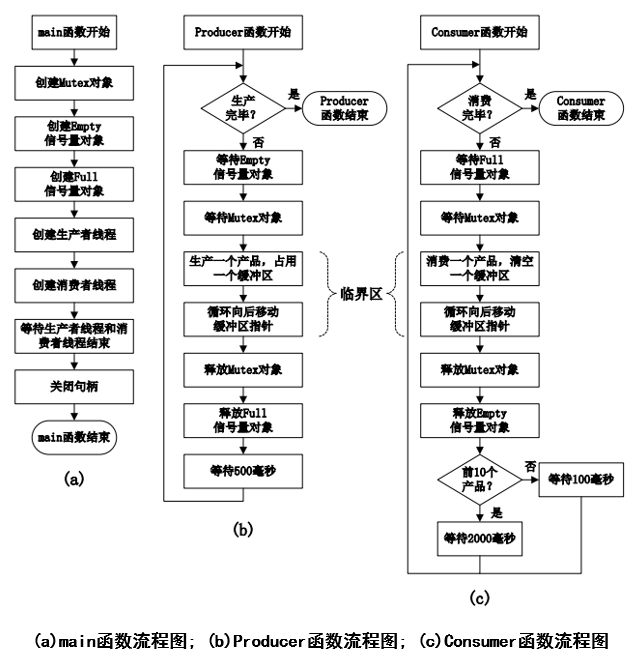
问题描述：有一群生产者进程在生产某种产品，并将此产品提供给一群消费者进程去消费。为使生产者进程和消费者进程能并发执行，在他们之间设置了一个具有 n 个缓冲区的缓冲池，生产者进程可以将它生产的一个产品放入一个缓冲区中，消费者进程可以从一个缓冲区中取得一个产品消费。尽管所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步，即不允许消费者进程到一个空缓冲区去取产品，也不允许生产者进程向一个已经装有产品的缓冲区中放入产品。

简要分析：主函数首先调用CreateMutex函数创建Mutex对象实现信号量的互斥访问，创建成功返回Mutex对象的句柄。同时main函数还创建了Empty和Full信号量，以及生产者和消费者进程。等待生产者线程和消费者线程结束后关闭句柄，main函数执行结束。在

Producer函数：函数首先通过一个for循环，循环变量为想要生产的数量，当未生产完毕时，等待Empty信号量，检查是否为空，是则生产，等待Mutex信号量，实现对临界资源的互斥访问，当满足条件时生产一个产品，占用一个缓冲区，并且将缓冲区指针InIndex循环右移，释放Mutex信号量，释放临界资源。释放Full信号量，休息500ms后再进行循环生产。

Consumer函数：首先通过一个for循环，循环变量为想要消费的产品数量，如果消费完毕则结束函数，否则进入for循环体中，等待Full信号量，如果Full信号量非空，则Consumer消费产品，接着等待Mutex信号量，申请进入临界区，实现对临界资源的互斥访问。进入临界区后，先消费一个产品，并清空一个缓冲区，将OutIndex缓冲区指针指向下一个产品，释放Mutex对象，结束对临界区的访问，最后释放Empty信号量对象。由于想要让前10个数的消费速度比较慢，后面的较快。所以sleep的时间根据if语句分别设置。如果是前10个数则等待2000ms后再进行，如果不是则等待100ms后再循环执行。

各个函数的流程图如下图所示：



* 1. **源代码**

#include "EOSApp.h"

// 缓冲池。

#define BUFFER\_SIZE 10

int Buffer[BUFFER\_SIZE];

// 产品数量。

#define PRODUCT\_COUNT 30

// 用于生产者和消费者同步的对象句柄。

HANDLE MutexHandle;

HANDLE EmptySemaphoreHandle;

HANDLE FullSemaphoreHandle;

// 生产者和消费者的线程函数

ULONG Producer(PVOID Param);

ULONG Consumer(PVOID Param);

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

{

HANDLE ProducerHandle;

HANDLE ConsumerHandle;

// 创建用于互斥访问缓冲池的 Mutex 对象。

MutexHandle = CreateMutex(FALSE, NULL);

if (NULL == MutexHandle) {

return 1;

}

// 创建 Empty 信号量，表示缓冲池中空缓冲区数量。初始计数和最大计数都为 BUFFER\_SIZE。

EmptySemaphoreHandle = CreateSemaphore(BUFFER\_SIZE, BUFFER\_SIZE, NULL);

if (NULL == EmptySemaphoreHandle) {

return 2;

}

// 创建 Full 信号量，表示缓冲池中满缓冲区数量。初始计数为 0，最大计数为 BUFFER\_SIZE。

FullSemaphoreHandle = CreateSemaphore(0, BUFFER\_SIZE, NULL);

if (NULL == FullSemaphoreHandle) {

return 3;

}

// 创建生产者线程。

ProducerHandle = CreateThread( 0, // 默认堆栈大小

Producer, // 线程函数入口地址

NULL, // 线程函数参数

0, // 创建标志

NULL ); // 线程 ID

if (NULL == ProducerHandle) {

return 4;

}

// 创建消费者线程。

ConsumerHandle = CreateThread( 0,

Consumer,

NULL,

0,

NULL );

if (NULL == ConsumerHandle) {

return 5;

}

// 等待生产者线程和消费者线程结束。

WaitForSingleObject(ProducerHandle, INFINITE);

WaitForSingleObject(ConsumerHandle, INFINITE);

// 关闭句柄

CloseHandle(MutexHandle);

CloseHandle(EmptySemaphoreHandle);

CloseHandle(FullSemaphoreHandle);

CloseHandle(ProducerHandle);

CloseHandle(ConsumerHandle);

return 0;

}

// 生产者线程函数。

ULONG Producer(PVOID Param)

{

int i;

int InIndex = 0;

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

WaitForSingleObject(EmptySemaphoreHandle, INFINITE);

WaitForSingleObject(MutexHandle, INFINITE);

printf("Produce a %d\n", i);

Buffer[InIndex] = i;

InIndex = (InIndex + 1) % BUFFER\_SIZE;

ReleaseMutex(MutexHandle);

ReleaseSemaphore(FullSemaphoreHandle, 1, NULL);

// 休息一会。每 500 毫秒生产一个数。

Sleep(500);

}

return 0;

}

// 消费者线程函数。

ULONG Consumer(PVOID Param)

{

int i;

int OutIndex = 0;

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

WaitForSingleObject(FullSemaphoreHandle, INFINITE);

WaitForSingleObject(MutexHandle, INFINITE);

printf("\t\t\tConsume a %d\n", Buffer[OutIndex]);

OutIndex = (OutIndex + 1) % BUFFER\_SIZE;

ReleaseMutex(MutexHandle);

ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL);

// 休息一会儿。让前 10 个数的消费速度比较慢，后面的较快。

if (i < 10) {

Sleep(2000);

} else {

Sleep(100);

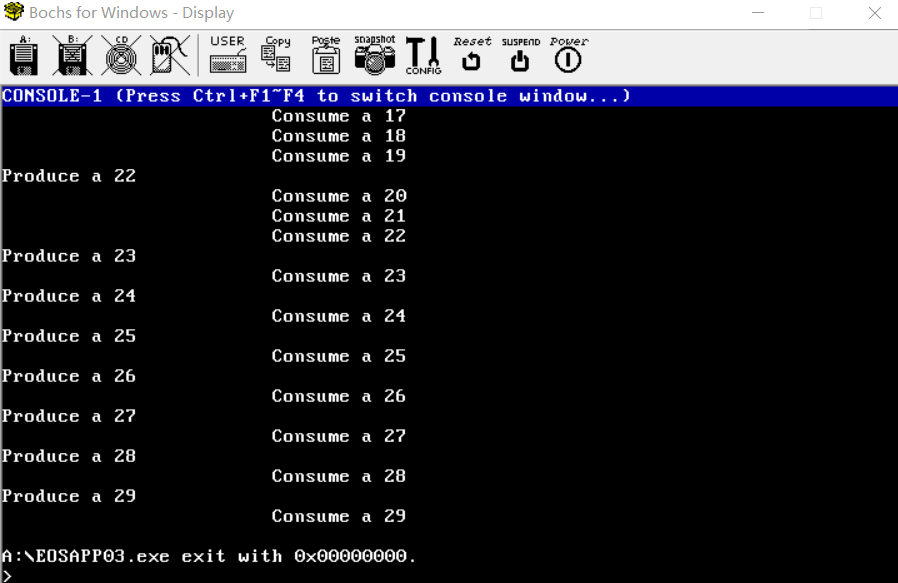
}

}

return 0;

}

* 1. **测试及结果**



* 1. **相关问题及解答**

（1）生产者线程和消费者线程是如何使用 Mutex、Empty 信号量和 Full 信号量来实现同

步的？在两个线程函数中对这三个同步对象的操作能够改变顺序吗？

答：①Mutex、Empty、Full 三个信号量的定义如下：

MutexHandle = CreateMutex(FALSE, NULL);

EmptySemaphoreHandle = CreateSemaphore(BUFFER\_SIZE, BUFFER\_SIZE, NULL);

FullSemaphoreHandle = CreateSemaphore(0, BUFFER\_SIZE, NULL);

BUFFER\_SIZE的宏定义为10，上面三个信号量的初始值分别为 1、10、0。生产者线程中当有生产者线程想要访问缓冲池时，首先对 Empty 减 1，如果大于 0，则说明还有剩余缓冲区可以让生产者放入产品，否则生产者线程进入等待队列；再对 Mutex 减 1，如果等于 0，则说明没有线程占用缓冲池，该生产者线程可以进入临界区，否则生产者线程进入等待队列。生产完产品后，对 Mutex 加 1，使其他线程能够进入临界区；再对 Full加 1，说明生产了一个产品占用了一个缓冲区。消费者线程同理，对信号量的操作顺序与生产者线程相反。

②不能对这三个同步对象的操作改变顺序，否则可能造成死锁。需要先执行对资源信号量的P操作，再执行对互斥信号量的P操作。

（2）生产者在生产了 13 号产品后本来要继续生产 14 号产品，可此时生产者为什么必须等待消费者消费了 4 号产品后，才能生产 14 号产品呢？生产者和消费者是怎样使用同步对象来实现该同步过程的呢？

答：①因为对临界资源的访问限制，程序中限定了缓冲池的大小为 10，只有缓冲池有空余时生产者才能向里边放产品，同时只有缓冲池有产品时消费者才能向外取东西。当生产者生产了13 号产品后，共生产了从 0 到 13 的 14 个产品，但是只消费了从 0 到 3 的 4 个产品，所以缓冲池中的 10 个缓冲区就都被占用了，所以不能继续生产 14 号产品，而要等到消费者消费掉一个产品后，缓冲池有空余位置，才能继续生产。

②当生产者线程生产了 13 号产品后，此时 Full 信号量的值为 10，而 Empty 信号量的值为 0，此时若生产者线程要再生产一个产品，先对 Empty 减 1，此时 Empty 值小于零，生产者线程进入等待队列；而此时若有一个消费者线程要消费一个产品，先对 Full 减 1，此时 Full值为 9，大于 0，如果没有线程占用缓冲池，消费者可以消费一个产品。这样，生产者和消费者就能实现同步过程了。

**2）EOS信号量工作过程的跟踪与源代码分析**

（分析EOS信号量实现的核心源代码，阐述其实现方法，包括数据结构和算法等；简要说明在本部分实验过程中完成的主要工作，包括对EOS信号量工作过程的跟踪等）

**2.1 创建信号量**

创建信号量的步骤可以分为如下几步，以Empty信号量为例：

1. main函数中调用eosapi.c中的CreateSemaphore函数

EmptySemaphoreHandle = CreateSemaphore(BUFFER\_SIZE, BUFFER\_SIZE, NULL);

1. CreateSemaphore函数调用eos内核文件semaphore.c中的PsCreateSemaphoreObject函数

Status = PsCreateSemaphoreObject( InitialCount,

MaximumCount,

Name,

&Handle );

1. 在此函数中，会在EOS内核管理的内存中创建一个信号量对象，初始化信号量对象中的各个成员操作在PsInitializeSemaphore函数中实现。

SEM\_CREATE\_PARAM CreateParam;

CreateParam.InitialCount = InitialCount;

CreateParam.MaximumCount = MaximumCount;

Status = ObCreateObject( PspSemaphoreType,

Name,

sizeof(SEMAPHORE),

(ULONG\_PTR)&CreateParam,

&SemaphoreObject);

// 用于初始化 semaphore 结构体的参数结构体。

typedef struct \_SEM\_CREATE\_PARAM{

LONG InitialCount;

LONG MaximumCount;

}SEM\_CREATE\_PARAM, \*PSEM\_CREATE\_PARAM;

// semaphore 对象的构造函数，在创建新 semaphore 对象时被调用。

VOID

PspOnCreateSemaphoreObject(

IN PVOID SemaphoreObject,

IN ULONG\_PTR CreateParam

)

{

PsInitializeSemaphore( (PSEMAPHORE)SemaphoreObject,

((PSEM\_CREATE\_PARAM)CreateParam)->InitialCount,

((PSEM\_CREATE\_PARAM)CreateParam)->MaximumCount );

}

1. PsInitializeSemaphore函数用于初始化信号量结构体，函数的参数中Semaphore表示要初始化的信号量结构体指针，InitialCount表示信号量的初始值，MaximumCount表示信号量的最大值。在创建mutex 信号量、full 信号量和 empty 信号量的时候，需要预先执行该函数初始化信号量结构体，才能使用其他函数。

VOID

PsInitializeSemaphore(

IN PSEMAPHORE Semaphore,

IN LONG InitialCount,

IN LONG MaximumCount

)

{

ASSERT(InitialCount >= 0 && InitialCount <= MaximumCount && MaximumCount > 0);

Semaphore->Count = InitialCount;

Semaphore->MaximumCount = MaximumCount;

ListInitializeHead(&Semaphore->WaitListHead);

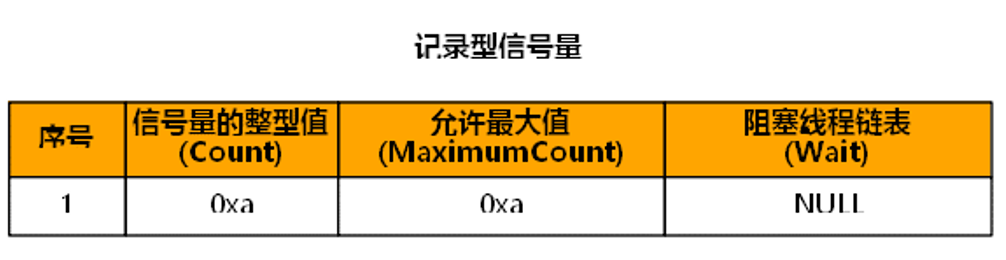
}

经观察此时PsInitializeSemaphore函数中用来初始化信号量结构体成员的值和传入CreateSemaphore函数的参数值是一致的。

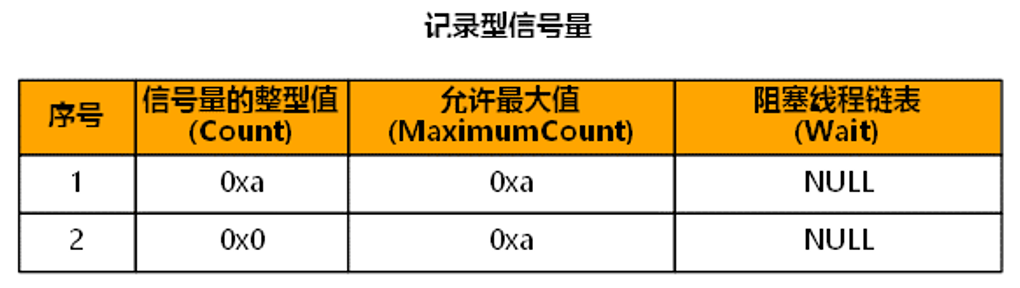
在“调用堆栈”窗口可以查看函数的调用层次，可以看到和上述分析函数调用过程一致。



此时在“记录型信号量”窗口可以看到系统中已经有创建完毕的信号量。



创建Full信号量的过程同上，在创建完成后刷新“记录型信号量”窗口，查看Empty信号量和Full信号量的初始状态。



**2.2 等待信号量（P操作）和释放信号量（V操作）**

**2.2.1 等待信号量（不阻塞）**

生产者和消费者刚开始执行时，用来存放产品的缓冲区都是空的，所以生产者在第一次调用WaitForSingleObject 函数等待 Empty 信号量时，不需要阻塞就可以立即返回。

a）在eosapp.c的Producer函数中，在等待Empty信号量的代码行

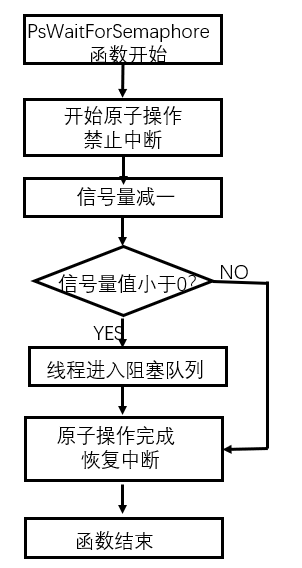
WaitForSingleObject(EmptySemaphoreHandle, INFINITE);

添加一个断点。

b）WaitForSingleObject 函数最终会调用内核中的PsWaitForSemaphore函数（P操作）完成等待信号量操作。

要实现P操作，在关中断后开始原子操作，信号量减1，若信号量小于0则线程进入阻塞队列，若不小于0原子操作完成，恢复中断。

c）函数流程图如下：



d）实现代码如下：

STATUS

PsWaitForSemaphore(

IN PSEMAPHORE Semaphore,

IN ULONG Milliseconds

)

{

BOOL IntState;

STATUS Status; // 定义STATUS类型变量，保存不同情况下的返回值

ASSERT(KeGetIntNesting() == 0); // 中断环境下不能调用此函数。

IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。

// 目前仅实现了标准记录型信号量，不支持超时唤醒功能，所以 PspWait 函数

// 的第二个参数的值只能是 INFINITE。

Semaphore->Count--;

if (Semaphore->Count < 0) {

PspWait(&Semaphore->WaitListHead, INFINITE);

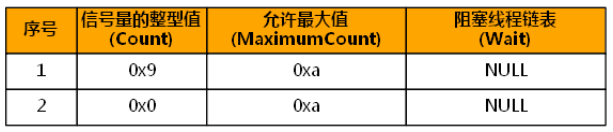
}

KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。

return Status;

}

执行结束后，“记录型信号量”窗口内容如下，可以看到此次执行并没有进行等待，只是将 Empty 信号量的计数减少了 1（由 10 变为了 9）就返回了。



**2.2.2 释放信号量（不唤醒）**

a）在eosapp.c的Producer函数中，释放Full信号量的代码行

ReleaseSemaphore(FullSemaphoreHandle, 1, NULL);

添加一个断点。

b）ReleaseSemaphore 函数最终会调用内核中的PsReleaseSemaphore函数（V操作）。

要实现V操作，在关中断后开始原子操作，判断释放了 Count 个信号量后是否使信号量溢出，若是，记录当前信号量的值并让信号量加1，并判断信号量数是否小于等于 0，若是从阻塞队列唤醒线程，若没溢出，直接返回对应 Status。

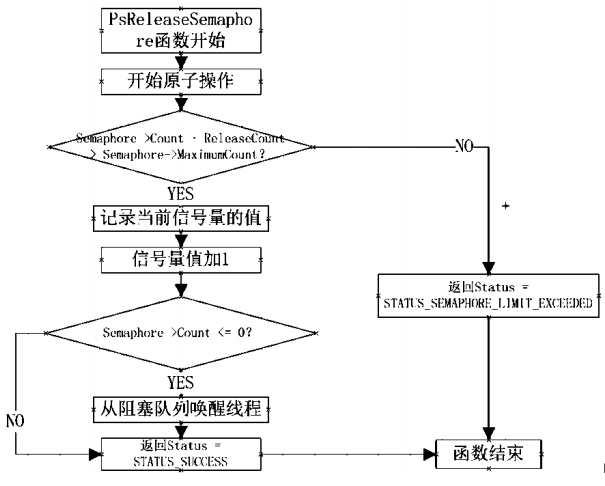
参数：Semaphore -- Wait 操作的信号量对象。

ReleaseCount -- 信号量计数增加的数量。当前只能为 1

PreviousCount -- 返回信号量计数在增加之前的值。

返回值：如果成功释放信号量，返回 STATUS\_SUCCESS。

c）函数流程图如下：



d）实现代码如下：

STATUS

PsReleaseSemaphore(

IN PSEMAPHORE Semaphore,

IN LONG ReleaseCount,

OUT PLONG PreviousCount

)

{

STATUS Status;

BOOL IntState;

IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。

if (Semaphore->Count + ReleaseCount > Semaphore->MaximumCount) {

Status = STATUS\_SEMAPHORE\_LIMIT\_EXCEEDED;

} else {

// 记录当前的信号量的值。

if (NULL != PreviousCount) {

\*PreviousCount = Semaphore->Count;

}

// 目前仅实现了标准记录型信号量，每执行一次信号量的释放操作

// 只能使信号量的值增加 1。

Semaphore->Count++;

if (Semaphore->Count <= 0) {

PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);

}

// 可能有线程被唤醒，执行线程调度。

PspThreadSchedule();

Status = STATUS\_SUCCESS;

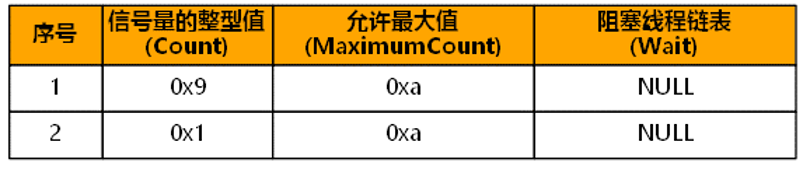
}

KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。

return Status;

}

执行结束后，“记录型信号量”窗口内容如下，可以看到此次执行没有唤醒其它线程（因为此时没有线程在 Full 信号量上被阻塞），只是将 Full 信号量的值增加了 1（由 0 变为了 1）



通过上述则生产者线程通过等待Empty信号量表示空缓冲区数量减少了1，通过释放Full信号量表示满缓冲区数量增加了 1，这样就表示生产者线程生产了一个产品并占用了一个缓冲区。

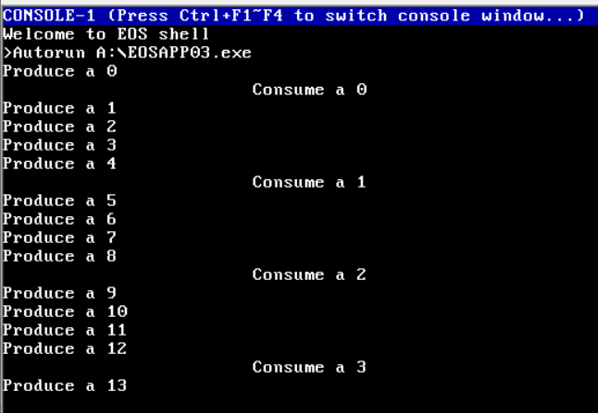
**2.2.3 等待信号量（阻塞）**

由于在生产者和消费者的实现代码中，生产者每生产一个产品休息500ms，而消费者在消费前10个产品时每消费一个产品休息2000ms，消费后面产品时每消费一个产品休息100ms，所以当缓冲池中所有的缓冲区都被产品占用时，生产者再生产新的产品时就会被阻塞。

在这种情况下，会调用semaphore.c文件中的的 PsWaitForSemaphore 函数的

PspWait(&Semaphore->WaitListHead, INFINITE);

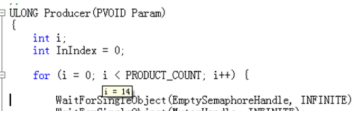
调试刚开始生产者、消费者不会被信号量阻塞，同步执行一段时间后阻塞。此时bochs控制台显示内容如下：



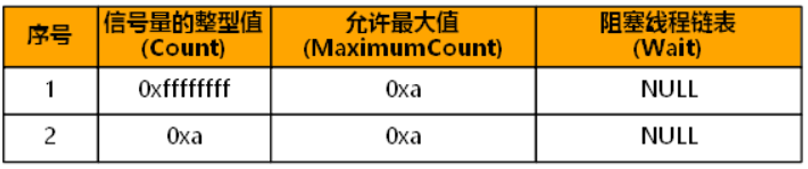
查看“调用堆栈”可知是从生产者线程函数进入的。



同时查看Producer 函数中变量 i 的值为 14，表示生产者线程正在尝试生产 14 号产品，也能说明等待信号量阻塞。



此时“记录型信号量”窗口Empty信号量值为-1，根据PsWaitForSemaphore函数中的设计实现，当信号量的值小于0时，会调用PspWait函数将生产者线程放入Empty信号量的等待队列中进行等待，使之让出处理器。



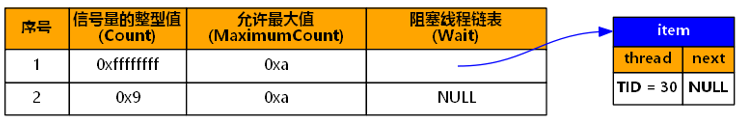
**2.2.4 释放信号量（唤醒）**

在上述情况后，只有当消费者线程从缓冲池中消费了一个产品，从而产生一个空缓冲区后，生产者线程才会被唤醒并继续生产 14 号产品。

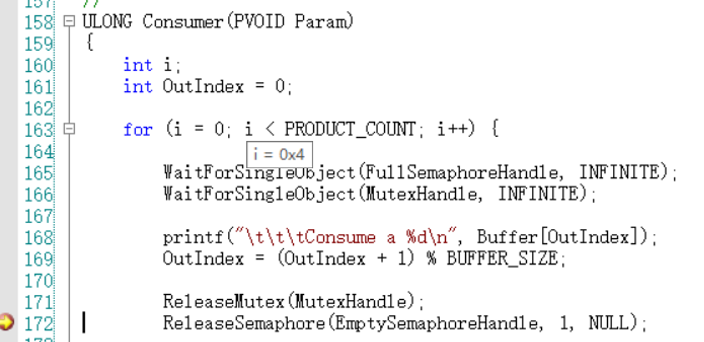
在Consumer函数中执行到释放Empty信号量的代码行时会发生中断。

ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL);

查看“记录型信号量”窗口，显示如下图所示：



此时Consumer 函数中变量i的值为 4，说明已经消费了4号产品。



进入PsReleaseSemaphore函数，执行至PsReleaseSemaphore 函数的第132行

PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);

此时Empty信号量的值增加至0，调用PspWakeThread 函数唤醒阻塞在 Empty 信号量等待队列中的生产者线程（放入就绪队列中），然后调用 PspSchedule 函数执行调度，这样生产者线程就得以继续执行。PspWakeThread 函数实现代码如下：

PTHREAD

PspWakeThread(

IN PLIST\_ENTRY WaitListHead,

IN STATUS WaitStatus

)

{

PTHREAD Thread;

if (!ListIsEmpty(WaitListHead)) {

// 唤醒等待队列的队首线程。

Thread = CONTAINING\_RECORD(WaitListHead->Next, THREAD, StateListEntry);

PspUnwaitThread(Thread);

PspReadyThread(Thread);

// 设置线程从PspWait返回的返回值。

Thread->WaitStatus = WaitStatus;

} else {

Thread = NULL;

}

return Thread;

}

**3）支持等待超时唤醒和批量释放功能的信号量实现**

（给出实现方法的简要描述、源代码、测试及结果等）

**3.1 实现方法简要描述**

* 对于支持等待超时唤醒功能的信号量，其计数值只能是大于等于0。当计数值大于0时，表示信号量为signaled状态；当计数值等于0时，表示信号量为nonsignaled状态。所以，PsWaitForSemaphore函数中原有的代码段

Semaphore->Count--;

if(Semaphore->Count < 0) {

PspWait(&Semaphore->WaitListHead, INFINITE);

}

应被修改为：先用计数值和0比较，当计数值大于0时，将计数值减1后直接返回成功；当计数值等于0时，调用PspWait函数阻塞线程的执行（将参数Milliseconds做为PspWait函数的第二个参数，并使用PspWait函数的返回值做为返回值）。

* 因为不能在原子操作的中途返回，所以在函数开始定义一个STATUS类型的变量，用来保存不同情况下的返回值，并在函数最后返回此变量的值。

修改PsReleaseSemaphore函数时要注意：

* 编写一个使用ReleaseCount做为计数器的循环体，来替换PsReleaseSemaphore函数中原有的代码段

Semaphore->Count++;

if (Semaphore->Count <= 0) {

PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);

}

* 在循环体中完成下面的工作：

1. 如果被阻塞的线程数量大于等于 ReleaseCount，则循环结束后，有 ReleaseCount 个线程会被唤醒，而且信号量计数的值仍然为 0；
2. 如果被阻塞的线程数量（可以为0）小于ReleaseCount，则循环结束后，所有被阻塞的线程都会被唤醒，并且信号量的计数值＝ReleaseCount－之前被阻塞线程的数量＋之前信号量的计数值。

* 在循环的过程中使用宏定义函数ListIsEmpty判断信号量的等待队列是否为空，例如 ListIsEmpty(&Semaphore->WaitListHead)

在EOS Kernel项目inc/rtl.h文件查看此宏定义的源代码。

**3.2 源代码**

对 PsWaitForSemaphore 的修改：

STATUS

PsWaitForSemaphore(

IN PSEMAPHORE Semaphore,

IN ULONG Milliseconds

)

{

BOOL IntState;

STATUS Status; // 定义STATUS类型变量，保存不同情况下的返回值

ASSERT(KeGetIntNesting() == 0); // 中断环境下不能调用此函数。

IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。

//

if (Semaphore->Count > 0){

Semaphore->Count--; // count>0计数值减1返回成功

Status = STATUS\_SUCCESS;

}

else if(Semaphore->Count ==0){

Status = PspWait(&Semaphore->WaitListHead, Milliseconds); // 阻塞执行

}

//

KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。

return Status;

}

**分析**：首先判断信号量的值是否大于 0，如果大于零则消耗一个单位的信号量，并且返回消耗成功，否则进行超时等待，并将此等待的状态输出。原代码未实现超时等待，在计数值减1后判断小于0则插入到等待队列。

对 PsReleaseSemaphore 函数的修改：

STATUS

PsReleaseSemaphore(

IN PSEMAPHORE Semaphore,

IN LONG ReleaseCount,

OUT PLONG PreviousCount

)

{

STATUS Status;

BOOL IntState;

IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。

if (Semaphore->Count + ReleaseCount > Semaphore->MaximumCount) {

Status = STATUS\_SEMAPHORE\_LIMIT\_EXCEEDED;

} else {

// 记录当前的信号量的值。

if (NULL != PreviousCount) {

\*PreviousCount = Semaphore->Count;

}

//

while((ListIsEmpty(&Semaphore->WaitListHead)==0)&&(ReleaseCount!=0)){

PspWakeThread(&Semaphore->WaitListHead, STATUS\_SUCCESS);

ReleaseCount--;

}

Semaphore->Count+=ReleaseCount;

//

// 可能有线程被唤醒，执行线程调度。

PspThreadSchedule();

Status = STATUS\_SUCCESS;

}

KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。

return Status;

}

**分析**：编写一个while循环，当信号量的等待队列不为空或仍有释放数量时，执行循环体。在循环体中唤醒该进程，同时注意将ReleaseCount自减。退出循环体后，此时信号量的计数值即为原来的值加上ReleaseCount的值。

**3.3 测试及结果**

对 Producer 函数的修改，添加以下代码：

while(WAIT\_TIMEOUT == WaitForSingleObject(EmptySemaphoreHandle, 300))

{

printf("Producer wait for empty semaphore timeout\n");

}

分析：该代码功能是当 Empty 信号量超时等待的时间超过 300ms 时，输出“Producer wait for empty semaphore timeout”，通过控制台显示可以判断超时等待是否添加成功。

对 Consumer 函数的修改，添加一下代码：

while(WAIT\_TIMEOUT == WaitForSingleObject(FullSemaphoreHandle, 300))

{

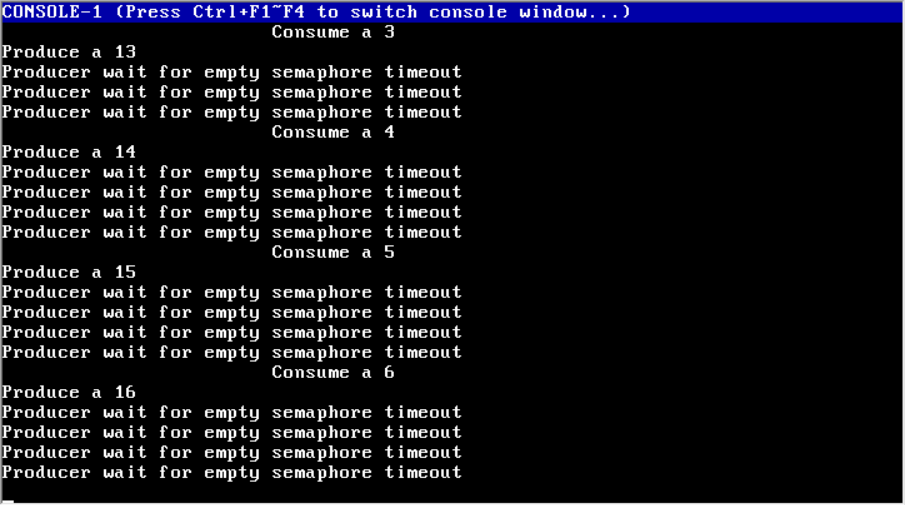
printf("Consumer wait for full semaphore timeout\n");

}

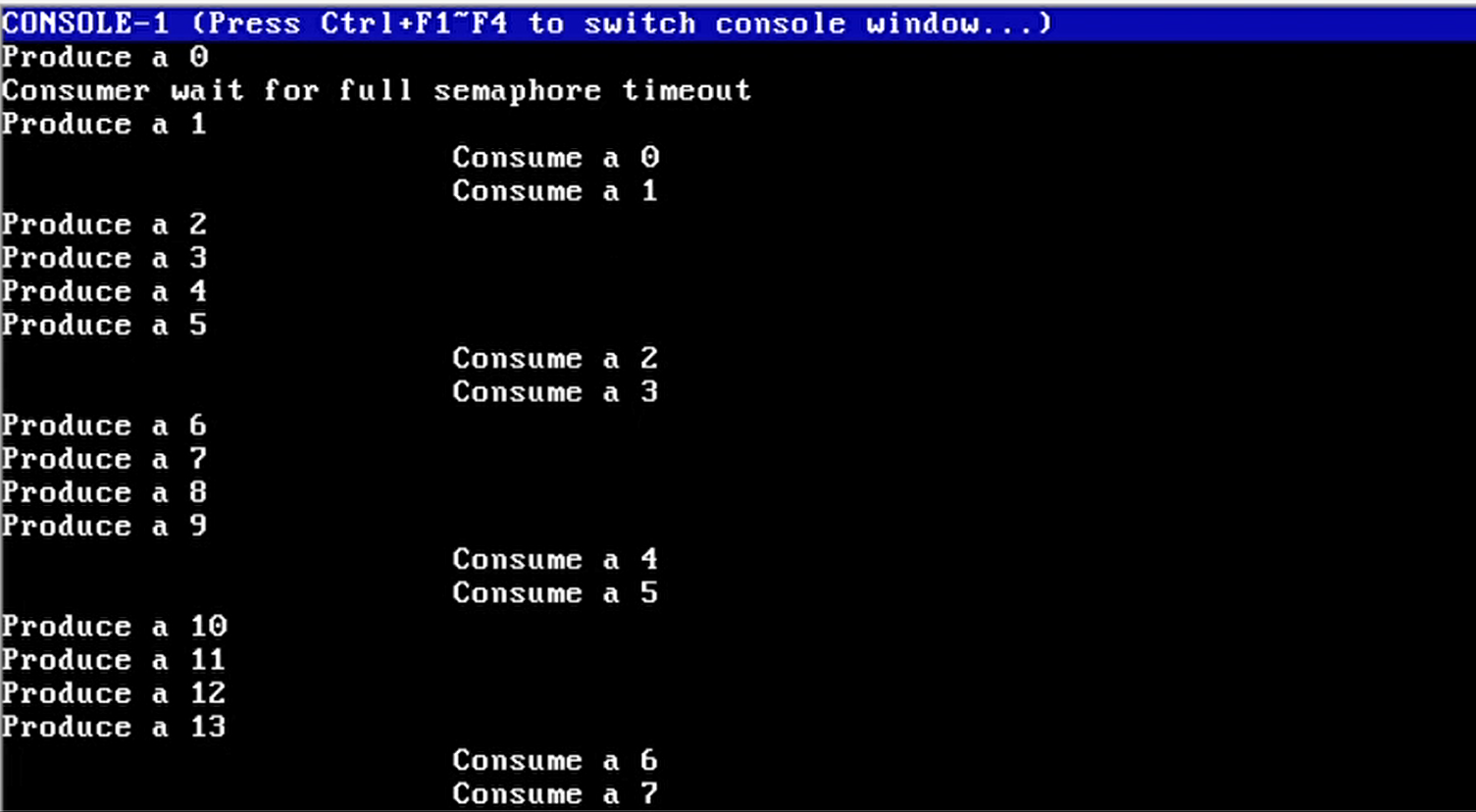
分析：添加的代码是 Full 信号量的超时等待，时间同样是 300ms，如果等待时间查过 300ms，就输出“Consumer wait for full semaphore timeout”，表示没有资源可以消耗。

执行结果如下图：

验证信号量超时等待功能是否能够正常执行：



验证超时等待功能已经能够正常执行：



**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS信号量实现方法的特点、不足及改进意见；结合EOS对信号量实现的相关问题提出自己的思考；分析支持等待超时唤醒和批量释放功能的信号量实现方法的有效性、不足和改进意见，如果同时采用了多种实现方法，则进行对比分析；其他需要说明的问题）

1. EOS信号量实现方法特点：

EOS 内核提供了三种专门用于线程同步的内核对象：互斥（Mutex）对象、信号量（Semaphore）对象和事件（Event）对象。其中 Mutex 对象用于同步多个线程对临界资源的互斥访问，Semaphore 对象是典型的记录型信号量，Event 对象是 EOS 中最具弹性的同步对象，Event 的两种状态 signaled 和nonsignaled 可完全由程序控制，使用灵活。

1. 思考在 ps/semaphore.c 文件内的 PsWaitForSemaphore 和PsReleaseSemaphore 函数中，为什么要使用原子操作？

答：在执行释放信号量和等待信号量时，是不允许 CPU 响应外部中断的，否则，会产生不可预料的结果，可能会产生死锁，无法正常完成函数的功能。在P、V操作中保证原子性是通过开关中断实现的。

IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。

…

KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。

开关中断的设计实现在rtl/i386/hal386.asm汇编文件中可以看到，同时，结合课上所学知识可以知道，开关中断实现原子操作只适合于单CPU的情况下，当多CPU时，可以用机器指令STL或交换函数SWAP设计实现。

1. 总结

在本次实验中，使用EOS信号量解决生产者-消费者问题，对进程同步有了进一步深入的理解。通过跟踪调试EOS信号量的工作过程，以前很模糊抽象的概念的实现在我面前展开，变得不这么神秘。理解了一个真正运行的操作系统是如何通过信号量及P、V操作解决进程同步问题，通过对函数的实现，每个变量的含义对程序代码进行修改，最终使EOS支持等待超时唤醒功能和批量释放，加深理解了对进程同步的原理。