

合肥工业大学

操作系统实验报告

实验题目	实验 5 进程的同步
学生姓名	孙淼
学 号	2018211958
专业班级	计算机科学与技术 18-2 班
指导教师	田卫东
完成日期	12.01

1. 实验目的和任务要求

- 使用 EOS 的信号量，编程解决生产者—消费者问题，理解进程同步的意义
- 调试跟踪 EOS 信号量的工作过程，理解进程同步的原理。
- 修改 EOS 的信号量算法，使之支持等待超时唤醒功能（有限等待），加深理解进程同步的原理。

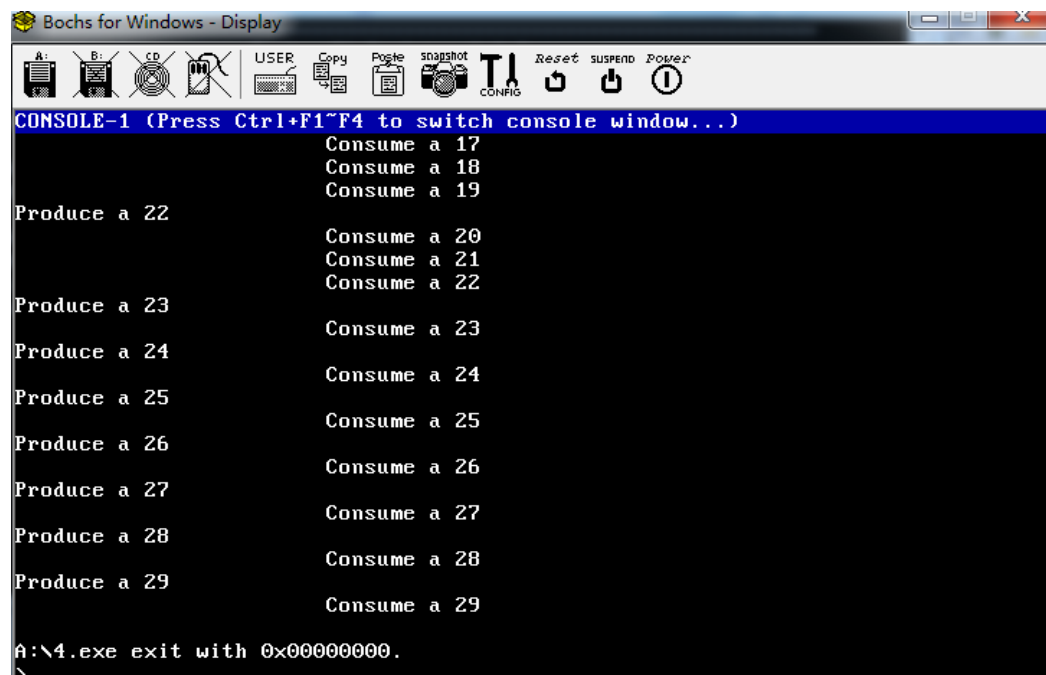
2. 实验原理

阅读本书第 5 章中的 5.3 节，学习 EOS 内核提供的三种同步对象（该实验没有涉及到 Event 同步对象）。重点理解各种同步对象的状态与使用方式。同时学习经典的生产者—消费者问题。

阅读 5.2 节，学习在 EOS 应用程序中调用 EOS API 函数 CreateThread 创建线程的方法。

3. 实验内容

使用 EOS 的信号量解决生产者—消费者问题，在“学生包”本实验对应的文件夹中，提供了使用 EOS 的信号量解决生产者—消费者问题的参考源代码文件 pc.c。使用 OS Lab 打开此文件，启动调试，立即激活虚拟机窗口查看生产者—消费者同步执行的过程。

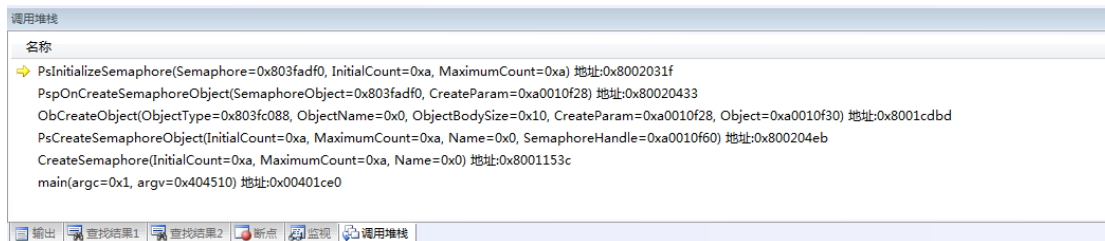


```
Bochs for Windows - Display
A: B: C: D: USER Copy Paste Snapshot T! Reset SUSPEND POWER
CONFIG
CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
Consume a 17
Consume a 18
Consume a 19
Produce a 22
Consume a 20
Consume a 21
Consume a 22
Produce a 23
Consume a 23
Produce a 24
Consume a 24
Produce a 25
Consume a 25
Produce a 26
Consume a 26
Produce a 27
Consume a 27
Produce a 28
Consume a 28
Produce a 29
Consume a 29
A:\4.exe exit with 0x00000000.
```

调试 EOS 信号量的工作过程，创建信号量，在 main 函数中创建 Empty 信号量的代码行，第 69 行 EmptySemaphoreHandle=CreateSemaphore(BUFFER_SIZE, BUFFER_SIZE, NULL); 添加一个断点。

```
61 //
62 MutexHandle = CreateMutex(FALSE, NULL);
63 if (NULL == MutexHandle) {
64     return 1;
65 }
66 //
67 // 创建 Empty 信号量，表示缓冲池中空闲缓冲区数量。初始计数和最大计数都为 BUFFER_SIZE。
68 //
69 EmptySemaphoreHandle = CreateSemaphore(BUFFER_SIZE, BUFFER_SIZE, NULL);
70 if (NULL == EmptySemaphoreHandle) {
71     return 2;
72 }
73 //
74 // 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。
75
```

启动调试 EOS 应用项目，到此断点处中断。进入 CreateSemaphore 函数。可以看到此 API 函数只是调用了 EOS 内核中的 PsCreateSemaphoreObject 函数来创建信号量对象。调试进入 semaphore.c 文件中 PsCreateSemaphoreObject 函数。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（分配一块内存），而初始化信号量对象中各个成员的操作是在 PsInitializeSemaphore 函数中完成的。



在 semaphore.c 文件的顶部查找到 PsInitializeSemaphore 函数的定义（第 19 行），在此函数的 第一行（第 39 行）代码处添加一个断点。按 F5 继续调试，到断点处中断。观察 PsInitializeSemaphore 函数中用来初始化信号量结构体成员的值，应该和传入 CreateSemaphore 函数的参数值是一致的。

```
38 {
39 | ASSERT(InitialCount >= 0 && InitialCount <= MaximumCount && MaximumCount > 0);
40
41 Semaphore->Count = InitialCount;
42 Semaphore->MaximumCount = MaximumCount;
43 ListInitializeHead(&Semaphore->WaitListHead);
44 }
45
46 STATUS
47 PsWaitForSemaphore(
48     IN PSEMAPHORE Semaphore,
```

单步调试 PsInitializeSemaphore 函数到第 44 行。查看信号量结构体被初始化的过程。打开“调用堆栈”窗口，查看函数的调用层次。选择“调试”菜单“窗口”中的“记录型信号量”，打开“记录型信号量”窗口。在该窗口工具栏上点击“刷新”按钮，可以看到当前系统中已经有一个创建完毕的信号量，如图

数据源: SEMAPHORE Semaphore

源文件: ps\semaphore.c

记录型信号量

序号	信号量的整型值 (Count)	允许最大值 (MaximumCount)	阻塞线程链表 (Wait)
1	0xa	0xa	NULL

继续在 EOS 应用程序进行单步调试，查看 Full 信号量的创建过程。也可以在 Full 信号量创建完毕后刷新“记录型信号量”窗口，查看 Empty 信号量和 Full 信号量的初始状态。

序号	信号量的整型值 (Count)	允许最大值 (MaximumCount)	阻塞线程链表 (Wait)
1	0xa	0xa	NULL

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。在 eosapp.c 文件的 Producer 函数中，等待 Empty 信号量的代码行（第 136 行）WaitForSingleObject(EmptySemaphoreHandle, INFINITE)；添加一个断点。

```
133 |
134 | for (i = 0; i < PRODUCT_COUNT; i++) {
135 |
136 |     WaitForSingleObject(EmptySemaphoreHandle, INFINITE);
137 |     WaitForSingleObject(MutexHandle, INFINITE);
138 |
139 |     printf("Produce a %d\n", i);
140 |     Buffer[InIndex] = i;
141 |     InIndex = (InIndex + 1) % BUFFER_SIZE;
142 |
143 |     ReleaseMutex(MutexHandle);
144 |     ReleaseSemaphore(FullSemaphoreHandle, 1, NULL);
145 |
146 |     //
147 |     // 休息一会。每 500 毫秒生产一个数。
```

WaitForSingleObject 函数最终会调用内核中的 PsWaitForSemaphore 函数完成等待信号量操作。所以，在 semaphore.c 文件中 PsWaitForSemaphore 函数的第一行（第 68 行）添加一个断点。

```
60 返回值:
61  STATUS_SUCCESS。
62  当你修改信号量使之支持超时唤醒功能后，如果等待超时，应该返回 STATUS_TIMEOUT。
63
64  --*/
65  {
66      BOOL IntState;
67
68      | ASSERT(KeGetIntNesting() == 0); // 中断环境下不能调用此函数。
69
70      IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。
71
72      //
```

单步调试，直到完成 PsWaitForSemaphore 函数中的所有操作。刷新“记录型信号量”窗口，显示如图 13-4 所示的内容，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。

序号	信号量的整型值 (Count)	允许最大值 (MaximumCount)	阻塞线程链表 (Wait)
1	0x9	0xa	NULL
2	0x0	0xa	NULL

数据源: SEMAPHORE Semaphore

源文件: ps\semaphore.c

记录型信号量

序号	信号量的整型值 (Count)	允许最大值 (MaximumCount)	阻塞线程链表 (Wait)
1	0x9	0xa	NULL
2	0x0	0xa	NULL

释放信号量（不唤醒），在 eosapp.c 文件的 Producer 函数中，释放 Full 信号量的代码行（第 144 行） ReleaseSemaphore(FullSemaphoreHandle, 1, NULL)；添加一个断点。

```
136 WaitForSingleObject(EmptySemaphoreHandle, INFINITE);
137 WaitForSingleObject(MutexHandle, INFINITE);
110
111 | IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。
112
113 | if (Semaphore->Count + ReleaseCount > Semaphore->MaximumCount) {
114
115     Status = STATUS_SEMAPHORE_LIMIT_EXCEEDED;
116
117 | } else {
118
119     //
120     // 记录当前的信号量的值。
121     //
122 | if (NULL != PreviousCount) {
123     Sleep(500);
124
125 | }
```

当黄色箭头指向第 269 行时使用 F11 单步调试，进入 PsReleaseSemaphore

函数。

单步调试，直到完成 PsReleaseSemaphore 函数中的所有操作。刷新“记录型信号量”窗口，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）。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了一个缓冲区。

等待信号量（阻塞），开始时生产者线程生产产品的速度较快，而消费者线程消费产品的速度较慢，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 semaphore.c 文件中的 PsWaitForSemaphore PspWait(&Semaphore->WaitListHead, INFINITE); 代码行（第 78 行）添加一个断点。

启动调试，并立即激活虚拟机窗口查看输出。开始时生产者、消费者都不会被信号量阻塞，同步执行一段时间后才在断点处中断。

数据源: SEMAPHORE Semaphore

源文件: ps\semaphore.c

记录型信号量

序号	信号量的整型值 (Count)	允许最大值 (MaximumCount)	阻塞线程链表 (Wait)
		
76	Semaphore->Count--;		
77	if (Semaphore->Count < 0) {		
78	PspWait(&Semaphore->WaitListHead, INFINITE);		
79	}		
80			
81	KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。		
82			
83	return STATUS_SUCCESS;		
84	}		

```

CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
Welcome to EOS shell
>Autorun A:\4.exe
Produce a 0
Consume a 0
Produce a 1
Produce a 2
Produce a 3
Produce a 4
Consume a 1
Produce a 5
Produce a 6
Produce a 7
Produce a 8
Consume a 2
Produce a 9
Produce a 10
Produce a 11
Produce a 12
Consume a 3
Produce a 13

```

中断后，查看“调用堆栈”窗口，有 Producer 函数对应的堆栈帧，说明此次调用是从生产者线程函数进入的。

ObWaitForObject(Handle=0x5, Milliseconds=0xffffffff) 地址:0x8001d1ab
WaitForSingleObject(Handle=0x5, Milliseconds=0xffffffff) 地址:0x800115c3
Producer(Param=0x0) 地址:0x00401e67
PspThreadStartup() 地址:0x8001f952
??0 地址:0x00000000 (无调试信息)

刷新“记录型信号量”窗口，查看 Empty 信号量计数（Semaphore->Count）的值为-1，所以会调用 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。

数据源: SEMAPHORE Semaphore

源文件: ps\semaphore.c

记录型信号量

序号	信号量的整型值 (Count)	允许最大值 (MaximumCount)	阻塞线程链表 (Wait)
1	0xffffffff	0xa	NULL
2	0xa	0xa	NULL

在“调用堆栈”窗口中双击 Producer 函数所在的堆栈帧，绿色箭头指向等待

Empty 信号量的代码行，查看 Producer 函数中变量 i 的值为 14，表示生产者

```
135 |
136 |     WaitForSingleObject(EmptySemaphoreHandle, INFINITE);
137 |     WaitForSingleObject(MutexHandle, INFINITE);
138 |
139 |     printf("Produce a %d\n", i);
140 |     Buffer[InIndex] = i;
141 |     InIndex = (InIndex + 1) % BUFFER_SIZE;
142 |
143 |     ReleaseMutex(MutexHandle);
144 |     ReleaseSemaphore(FullSemaphoreHandle, 1, NULL);
145 |
146 |     //
147 |     // 休息一会。每 500 毫秒生产一个数。
148 |     //
149 |     Sleep(500);
150 | }
151 |
152 | return 0;
153 | }
```

调用堆栈

名称
ObWaitForObject(Handle=0x5, Milliseconds=0xffffffff) 地址:0x8001d1ab
WaitForSingleObject(Handle=0x5, Milliseconds=0xffffffff) 地址:0x800115c3
➡ Producer(Param=0x0) 地址:0x00401e67
PspThreadStartup() 地址:0x8001f952
??() 地址:0x00000000 (无调试信息)

线程正在尝试生产 14 号产品。

激活虚拟机窗口查看输出的结果。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，这与之前调

```
CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
Welcome to EOS shell
>Autorun A:\4.exe
Produce a 0
Consume a 0

Produce a 1
Produce a 2
Produce a 3
Produce a 4
Consume a 1

Produce a 5
Produce a 6
Produce a 7
Produce a 8
Consume a 2

Produce a 9
Produce a 10
Produce a 11
Produce a 12
Consume a 3

Produce a 13
_
```


试的结果是一致的。

释放信号量（唤醒），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。可以按照下面的步骤调试：在 eosapp.c 文件的 Consumer 函数中，释放 Empty 信号量的代码行（第 172 行）ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL)；添加一个断点。按 F5 继续调试，会在断点处中断。刷新“记录型信号量”窗口，会显示如图 3-8 所示的内容，可以看到此时生产者线程仍然阻塞在 Empty 信号量上。

数据源: SEMAPHORE Semaphore

源文件: ps\semaphore.c

记录型信号量

序号	信号量的整型值 (Count)	允许最大值 (MaximumCount)	阻塞线程链表 (Wait)	item
1	0xffffffff	0xa		thread
2	0x9	0xa	NULL	next
				TID = 30
				NULL

4. 查看 Consumer 函数中变量 i 的值为 4，说明已经消费了 4 号产品。

名称	值	类型
i	0x4	int

验证生产者线程被唤醒后，是从之前被阻塞时的状态继续执行的：在 semaphore.c 文件中 PsWaitForSemaphore 函数的最后一行（第 83 行）代码处添加一个断点。按 F5 继续调试，在断点处中断。查看 PsWaitForSemaphore 函数中 Empty 信号量计数 (Semaphore->Count) 的值为 0，和生产者线程被唤醒时的值是一致的。在“调用堆栈”窗口中可以看到是由 Producer 函数进入的。激活 Producer 函数的堆栈帧，查看 Producer 函数中变量 i 的值为 14，表明之前被阻塞的、正在尝试生产 14 号产品的生产者线程已经从 PspWait 函数返回并继续执行了。

名称	值	类型
i	4	int

修改 EOS 的信号量算法，同时修改 PsWaitForSemaphore 函数 PsReleaseSemaphore 函数中的代码，使这两个参数能够真正起到作用，使信号量对象支持等待超时唤醒功能和批量释放功能。

修改 PsWaitForSemaphore 函数时要注意，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）。  
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。绝不能在原子操作的中途返回！ 在 EOS Kernel 项目 ps/sched.c 文件的第 193 行查看 PspWait 函数的说明和源代码。
```

```

{
    BOOL IntState;
    STATUS Status;

    ASSERT(KernelGetIntNesting() == 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。
    //
    if (Semaphore->Count > 0) {
        Semaphore->Count--;
        Status=STATUS_SUCCESS;
    }
    else if (Semaphore->Count == 0) {
        Status=PspWait(&Semaphore->WaitListHead, Milliseconds);
    }

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

    return Status;
}

```

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 EOS Kernel 项目 ps/sched.c 文件的第 301 行查看 PspWakeThread 函数的说明和源代码 在循环的过程中可以使用宏定义函数 ListIsEmpty 判断信号量的等待队列是否为空，例如 ListIsEmpty(&Semaphore->WaitListHead) 可以在 EOS Kernel 项目 inc/rtl.h 文件的第 113 行查看此宏定义的源代码。

```

        BOOL IntState;

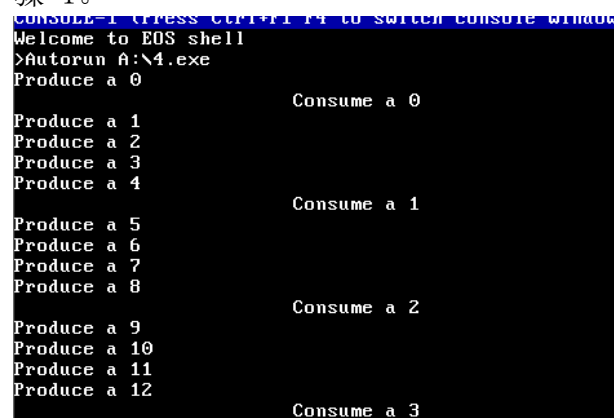
        IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。
3    if (Semaphore->Count + ReleaseCount > Semaphore->MaximumCount) {
            Status = STATUS_SEMAPHORE_LIMIT_EXCEEDED;
3    } else {
            //
            // 记录当前的信号量的值。
            //
3    if (NULL != PreviousCount) {
            *PreviousCount = 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。
            //
3    if (ReleaseCount > 0)
        {
            Semaphore->Count++;
3            while ((!ListIsEmpty(&Semaphore->WaitListHead)) && (ReleaseCount)) {
                PspWakeThread(&Semaphore->WaitListHead, STATUS_SUCCESS);
                PspThreadSchedule();
                ReleaseCount--;
-            }
            Semaphore->Count = ReleaseCount + Semaphore->Count;
            //
            // 可能有线程被唤醒，执行线程调度。
            //
            Status = STATUS_SUCCESS;
-        }
-    }

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

```

按照下面的方法进行测试：使用修改完毕的 EOS Kernel 项目生成完全版本的 SDK 文件夹，并覆盖之前的生产者—消费者应用程序项目的 SDK 文件夹。按 F5 调试执行原有的生产—消费者应用程序项目，结果必须仍然与图 13-2 一致。如果有错误，可以调试内核代码来查找错误，然后在内核项目中修改，并重复步骤 1。



```

CONSOLE-1 (Press Ctrl+F1 F4 to switch console window)
Welcome to EOS shell
>Autorun A:\4.exe
Produce a 0
Consume a 0
Produce a 1
Produce a 2
Produce a 3
Produce a 4
Consume a 1
Produce a 5
Produce a 6
Produce a 7
Produce a 8
Consume a 2
Produce a 9
Produce a 10
Produce a 11
Produce a 12
Consume a 3

```

Producer 函数中等待 Empty 信号量的代码行

```
WaitForSingleObject(EmptySemaphoreHandle, INFINITE);
```

替换为

```
while(WAIT_TIMEOUT == WaitForSingleObject(EmptySemaphoreHandle, 300))  
{  
printf("Producer wait for empty semaphore timeout\n");  
}
```

将 Consumer 函数中等待 Full 信号量的代码行

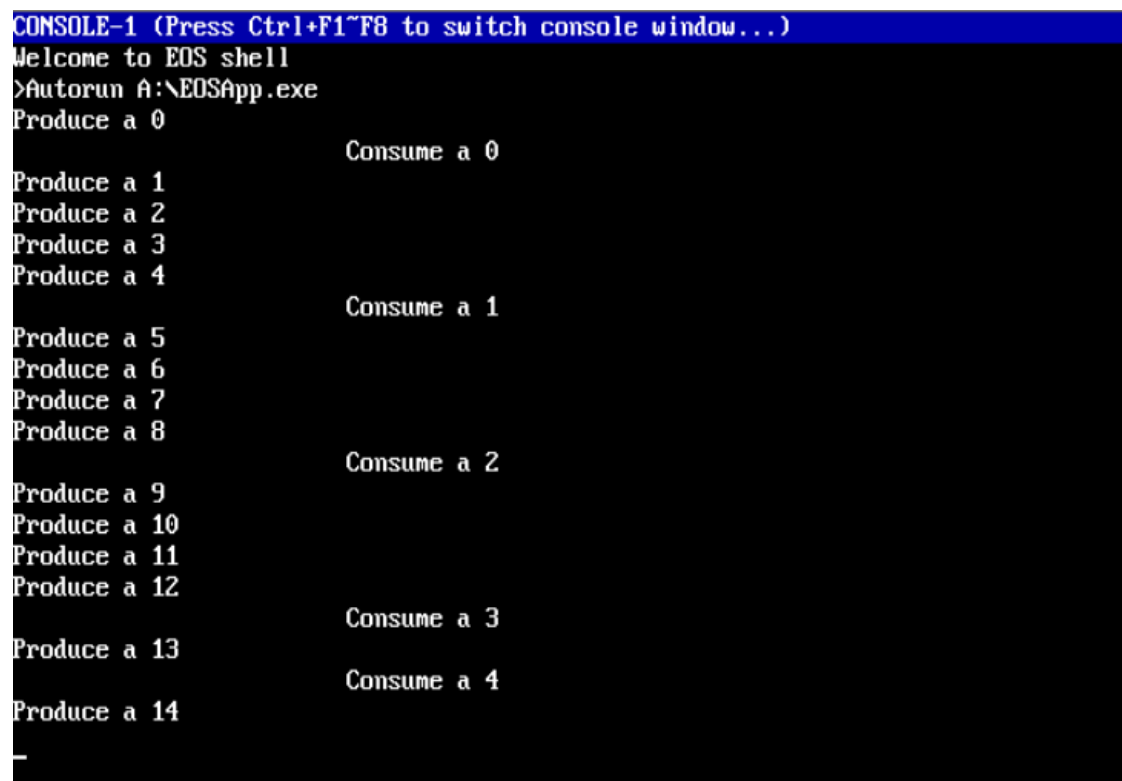
```
WaitForSingleObject(FullSemaphoreHandle, INFINITE);
```

替换为

```
while(WAIT_TIMEOUT == WaitForSingleObject(FullSemaphoreHandle, 300))  
{  
printf("Consumer wait for full semaphore timeout\n");  
}
```

启动调试新的生产者—消费者项目，查看在虚拟机中输出的结果，验证信号量超时等待功能是否能够正常执行，如图 13-6。如果有错误，可以调试内核代码来查找错误，然后在内核项目中修改，并重复步骤 1。

如果超时等待功能已经能够正常执行，可以考虑将消费者线程修改为一次消费两个产品，来测试 ReleaseCount 参数是否能够正常使用，如图 3-12。使用实验文件夹中 NewConsumer.c 文件中的 Consumer 函数替换原有的 Consumer 函数。



```
CONSOLE-1 (Press Ctrl+F1~F8 to switch console window...)  
Welcome to EOS shell  
>Autorun A:\EOSApp.exe  
Produce a 0  
Consume a 0  
Produce a 1  
Produce a 2  
Produce a 3  
Produce a 4  
Consume a 1  
Produce a 5  
Produce a 6  
Produce a 7  
Produce a 8  
Consume a 2  
Produce a 9  
Produce a 10  
Produce a 11  
Produce a 12  
Consume a 3  
Produce a 13  
Consume a 4  
Produce a 14  
_
```

```

Produce a 3
Produce a 4
                Consume a 2
                Consume a 3

Produce a 5
Produce a 6
Produce a 7
Produce a 8
                Consume a 4
                Consume a 5

Produce a 9
Produce a 10
Produce a 11
Produce a 12
                Consume a 6
                Consume a 7

Produce a 13
Produce a 14
Produce a 15
                Consume a 8
                Consume a 9

Produce a 16
Produce a 17

```

4. 实验的思考与问题分析

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？可以参考本书第 2 章中的第 2.6 节。

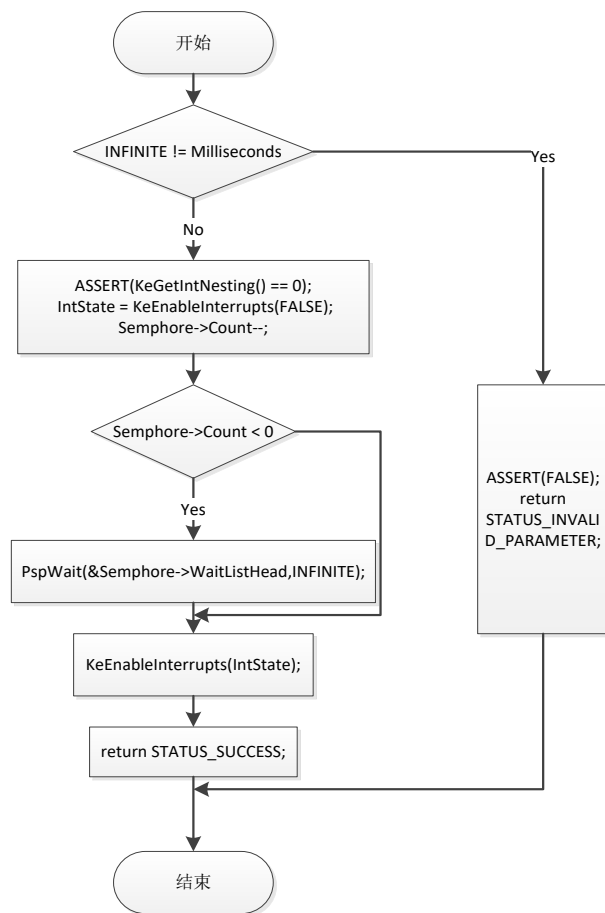
答：

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。EOS 核中维护了大量核数据，正是这些数据描述了 EOS 操作系统的状态如果有组相互关联的核数据共同描述了这个操作系统的某个状态，那么在修改这样一组核数据时就必须保证一致性。这就要求修改这部分数据的代码在执行过程中不能被打断，这种操作叫做“原语操作”。

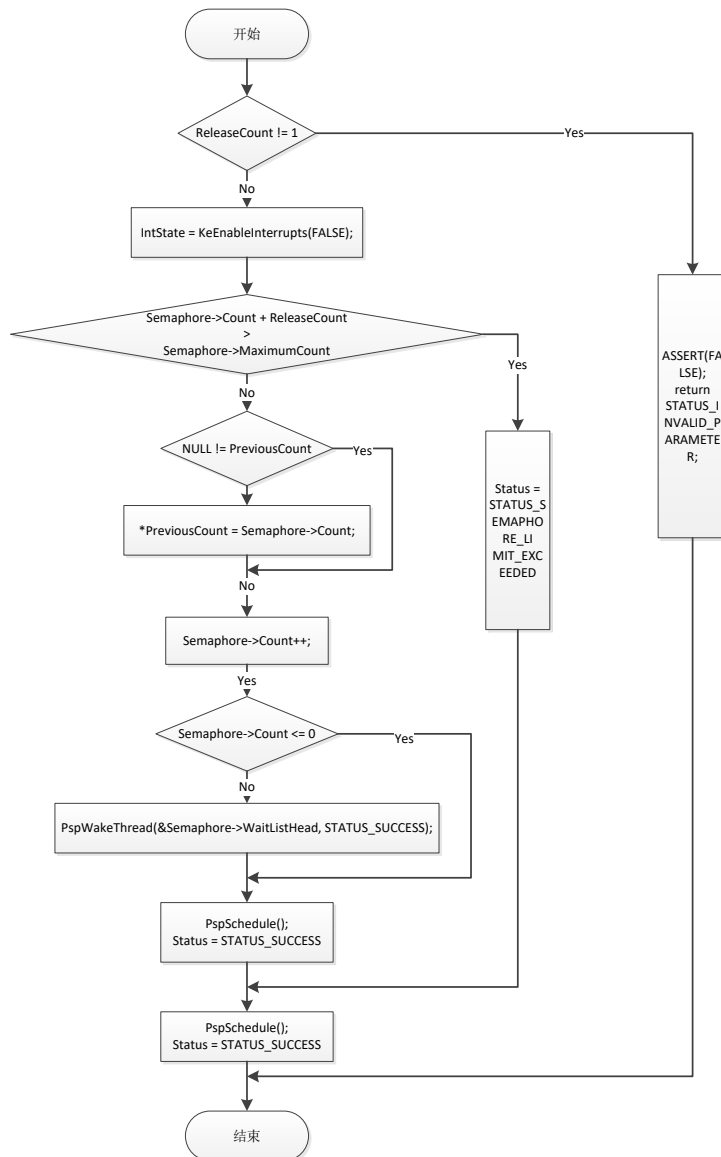
2. 绘制 ps/semaphore.c 文件内 PsWaitForSemaphore 和 PsReleaseSemaphore 函数的流程图。

答：

(1)PsWaitForSemaphore



(2) PsReleaseSemaphore



3. 根据本实验 3.3.2 节中设置断点和调试的方法，练习调试消费者线程在消费第一个产品时，等待 Full 信号量和释放 Empty 信号量的过程。注意信号量计数是如何变化的。

答：

这是因为临界资源的限制。临界资源就像产品仓库,只有“产品仓库”空闲生产者才能生产东西,有权向里面放东西。所以它必须等到消费者，取走产品，“产品空间” (临界资源)空闲时，才继续生产 14 号产品。

4. 根据本实验 3.3.2 节中设置断点和调试的方法，自己设计一个类似的调试方案来验证消费者线程在消费 24 号产品时会被阻塞，直到生产者线程生产了 24 号产品后，消费者线程才被唤醒并继续执行的过程。提示，可以按照下面的步骤进行调试：

(1) 删除所有的断点。

(2) 在 Consumer 函数中等待 Full 信号量的代码行（第 173 行）WaitForSingleObject(FullSemaphoreHandle, INFINITE); 添加一个断点。

- (3) 在“断点”窗口（按 Alt+F9 打开）中此断点的名称上点击右键。
- (4) 在弹出的快捷菜单中选择“条件”。
- (5) 在“断点条件”对话框（按 F1 获得帮助）的表达式编辑框中，输入表达式“ $i == 24$ ”。
- (6) 点击“断点条件”对话框中的“确定”按钮。
- (7) 按 F5 启动调试。只有当消费者线程尝试消费 24 号产品时才会在该条件断点处中断。

答：

删除所有的断点。在 Consumer 函数中等待 Full 信号量的代码行（第 173 行）`WaitForSingleObject(FullSemaphoreHandle, INFINITE)`；添加一个断点。在“断点”窗口（按 Alt+F9 打开）中此断点的名称上点击右键。

在弹出的快捷菜单中选择“条件”。在“断点条件”对话框（按 F1 获得帮助）的表达式编辑框中，输入表达式“ $i == 24$ ”。点击“断点条件”对话框中的“确定”按钮。按 F5 启动调试。只有当消费者线程尝试消费 24 号产品时才会在该条件断点处中断。

5. 创建多个生产者线程和多个消费者线程进行同步，注意临界资源也会发生变化。然后，添加断点，调试程序，在“记录型信号量”窗口中查看信号量的变化情况和阻塞在信号量上的线程信息，同时，在“互斥信号量”窗口中查看互斥信号量的变化情况和阻塞在互斥信号量上的线程信息。

答：

申请两个资源信号量 `empty` 和 `full` 控制生产者线程和消费者线程之间的同步；只有当 `empty > 0` 时，表示缓冲区有空闲，生产者线程可以进入临界区，每次线程结束后 `empty-1`；`full+1`。`empty <= 0` 时缓冲区已满，生产者线程阻塞。只有当 `full > 0` 时，表示缓冲区内有产品，消费者可以进入临界区，每次线程结束后 `empty+1`；`full-1`。`full <= 0` 时缓冲区为空，消费者线程阻塞。

4. 总结和感想体会

此次实验让我对消费者问题加深了理解，消费者问题是一个经典的进程同步问题，该问题最早由 Dijkstra 提出，用以演示他提出的信号量机制。在同一个进程地址空间内执行的两个线程。生产者线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。当生产者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放出一个空缓冲区。当消费者线程消费物品时，如果没有满的缓冲区，那么消费者线程将被阻塞，直到新的物品被生产出来。进程的同步问题是计算机操作系统重要功能，学好进程才能学好其他。

通过本次实验，我巩固了使用多线程编程的方法，和使用信号量同步进/线程的技巧。该实验令我更好的理解了使用信号量实现线程同步的过程，对于信号量的操作和改变有了更深刻的认识。课程上的不懂之处也在本次试验中得到了解决。

参考文献

- [1] 北京英真时代科技有限公司 [DB/CD]. <http://www.engintime.com>.
- [2] 汤子瀛，哲凤屏，汤小丹。计算机操作系统。西安：西安电子科技大学出版社，1996.