

实验 5 进程的同步

1. 实验目的和任务要求

- (1) 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。
- (2) 调试跟踪 EOS 信号量的工作过程，理解进程同步的原理。
- (3) 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

- ① EOS 内核提供的三种同步对象
- ② 各种同步对象的状态与使用方式
- ③ 生产者—消费者问题
- ④ 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

3.1 准备实验

按照下面的步骤准备本次实验：

1. 启动 OS Lab。
2. 新建一个 EOS Kernel 项目。
3. 生成 EOS Kernel 项目，从而在该项目文件夹中生成 SDK 文件夹。
4. 新建一个 EOS 应用程序项目。
5. 使用在第 3 步生成的 SDK 文件夹覆盖 EOS 应用程序项目文件夹中的 SDK 文件夹。

3.2 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（将文件拖动到 OS Lab 窗口中释放即可打开），仔细阅读此文件中的源代码和注释，各个函数的流程图可以参见图 13-1。思考在两个线程函数（Producer 和 Consumer）中，哪些是临界资源？哪些代码是临界区？哪些代码是进入临界区？哪些代码是退出临界区？进入临界区和退出临界区的代码是否成对出现？

按照下面的步骤查看生产者—消费者同步执行的过程：

1. 使用 pc.c 文件中的源代码，替换之前创建的 EOS 应用程序项目中 EOSApp.c 文件内的源代码。
 2. 按 F7 生成修改后的 EOS 应用程序项目。
 3. 按 F5 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，如图 13-2 所示。
 4. 待应用程序执行完毕后，结束此次调试。
- 按下 F5 后弹出的调试对话框，选择“否”，之后开始执行，运行结果如下：

```
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:\eosapp.exe exit with 0x00000000.
>

000000000000i|  l reading configuration from C:\eosapp\eosapp\bochs\bochsrcdb
g.bxrc
000000000000i|  l Enabled gdbstub
000000000000e|  l C:\eosapp\eosapp\bochs\bochsrcdbg.bxrc:6: ataX-master/slave
CHS set to 0/0/0 - autodetection enabled
000000000000i|  l installing win32 module as the Bochs GUI
000000000000i|  l using log file bochsout.txt
Waiting for gdb connection on port 1235
Connected to 127.0.0.1
```

3.3 调试 EOS 信号量的工作过程

3.3.1 创建信号量

信号量结构体（SEMAPHORE）中的各个成员变量是由 API 函数 CreateSemaphore 的对应参数初始化的，查看 main 函数中创建 Empty 和 Full 信号量使用的参数有哪些不同，又有哪些相同，思考其中的原因。

按照下面的步骤调试信号量创建的过程：

1. 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。

添加断点结果如下：



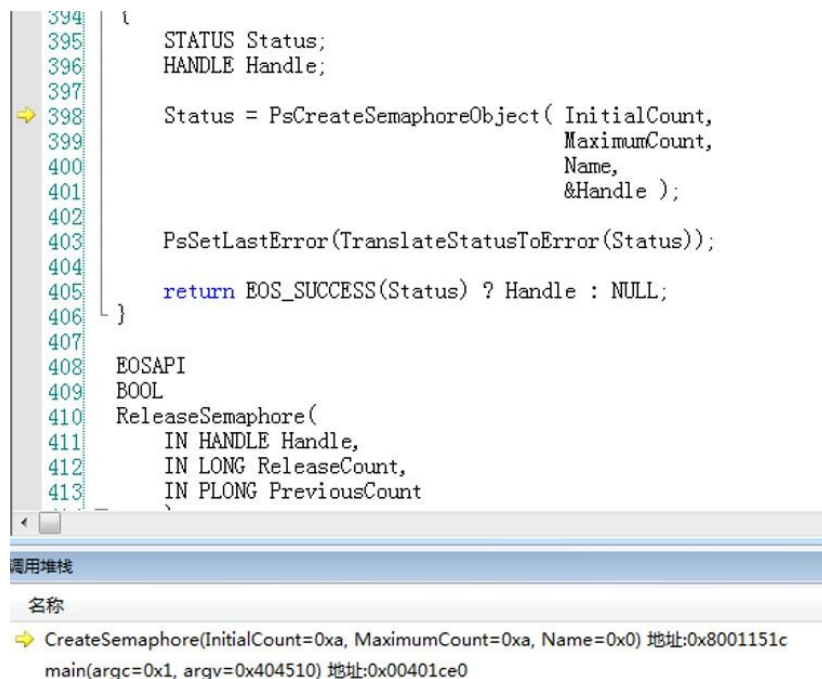
```
63 |         return 1;
64 |     }
65 |
66 |     //
67 |     // 创建 Empty 信号量，表示缓冲池中空缓冲区数量。初始计数和最大计数都为 1
68 |     //
69 |     EmptySemaphoreHandle = CreateSemaphore(BUFFER_SIZE, BUFFER_SIZE, NULL);
70 |     if (NULL == EmptySemaphoreHandle) {
71 |         return 2;
72 |     }
```

2. 按 F5 启动调试 EOS 应用项目，到此断点处中断。

调试结果如下：



3. 按 F11 调试进入 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。
查看结果如下：



4. 按 F11 调试进入 semaphore.c 文件中的 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。
进入 PsCreateSemaphoreObject 函数查看结果如下：

```
218 |
219 | if(InitialCount < 0 || MaximumCount <= 0 || InitialCount > MaximumCo
220 |     return STATUS_INVALID_PARAMETER;
221 | }
222 |
223 | //
224 | // 创建信号量对象。
225 | //
226 | CreateParam.InitialCount = InitialCount;
227 | CreateParam.MaximumCount = MaximumCount;
228 |
229 | Status = ObCreateObject( PspSemaphoreType,
230 |                         Name,
231 |                         sizeof(SEMAPHORE),
232 |                         (ULONG_PTR)&CreateParam,
233 |                         &SemaphoreObject);
234 |
235 | if (!EOS_SUCCESS(Status)) {
236 |     return Status;
237 | }
```

调用堆栈

名称
PsCreateSemaphoreObject(InitialCount=0xa, MaximumCount=0xa, Name=0x0, SemaphoreHandle=0xa0010f...
CreateSemaphore(InitialCount=0xa, MaximumCount=0xa, Name=0x0) 地址:0x8001153c
main(argc=0x1, argv=0x404510) 地址:0x00401ce0

5. 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。添加断点如下：

```
37 | /*
38 | {
39 |     ASSERT(InitialCount >= 0 && InitialCount <= MaximumCount && Maximum
40 |
41 |     Semaphore->Count = InitialCount;
42 |     Semaphore->MaximumCount = MaximumCount;
43 |     ListInitializeHead(&Semaphore->WaitListHead);
44 | }
45 |
46 | STATUS
47 | PsWaitForSemaphore(
48 |     IN PSEMAPHORE Semaphore,
49 |     IN ULONG Milliseconds
50 | )
51 | {
52 | /*
53 | 功能描述:
54 | 信号量的 Wait 操作 (P 操作)。
55 | */
56 | }
```

调用堆栈

名称
PsInitializeSemaphore(Semaphore=0x803fadf0, InitialCount=0xa, MaximumCount=0xa) 地址:0x800202ce
PspOnCreateSemaphoreObject(SemaphoreObject=0x803fadf0, CreateParam=0xa0010f28) 地址:0x80020433
ObCreateObject(ObjectType=0x803fc088, ObjectName=0x0, ObjectBodySize=0x10, CreateParam=0xa0010f28)
PsCreateSemaphoreObject(InitialCount=0xa, MaximumCount=0xa, Name=0x0, SemaphoreHandle=0xa0010f28)
CreateSemaphore(InitialCount=0xa, MaximumCount=0xa, Name=0x0) 地址:0x8001153c
main(argc=0x1, argv=0x404510) 地址:0x00401ce0

6. 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。

7. 按 F10 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。查看结果如下：

记录型信号量			
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

8. 可以继续 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。
Empty 信号量和 Full 信号量的初始状态如下：

记录型信号量			
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
2	0x0	0xa	NULL

3.3.2 等待信号量（P 操作）和释放信号量（V 操作）

3.3.2.1 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。按照下面的步骤进行调试：

1. 为了防止之前添加的断点影响后续的调试过程，首先需要删除所有断点。
2. 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。
3. 按 F5 继续调试，到断点处中断。
4. 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。
5. 按 F5 继续调试，到断点处中断。
6. 按 F10 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。在完成添加断点并调试的步骤后，查看“记录型信号量”窗口：

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

3.3.2.2 释放信号量（不唤醒）

1. 删除所有的断点（防止有些断点影响后面的调试）。
2. 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。
添加断点如下：

```
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。  
148      //  
149      Sleep(500);
```

3. 按 F5 继续调试，到断点处中断。
4. 按 F11 调试进入 ReleaseSemaphore 函数。
进入 ReleaseSemaphore 函数：


```
410 STATUS Status;  
417  
418 Status = PsReleaseSemaphoreObject(Handle, ReleaseCount, PreviousCount);  
419  
420 PsSetLastError(TranslateStatusToError(Status));  
421  
422 return EOS_SUCCESS(Status);  
423 }  
424  
425 EOSAPI  
426 ULONG  
427 WaitForSingleObject(  
428     IN HANDLE Handle,  
429     IN ULONG Milliseconds  
430 )  
431 {  
432     STATUS Status;  
433  
434     Status = ObWaitForObject(Handle, Milliseconds);  
435 }
```

调用堆栈

名称
ReleaseSemaphore(Handle=0x6, ReleaseCount=0x1, PreviousCount=0x0) 地址:0x80011572
Producer(Param=0x0) 地址:0x00401ef9
PspThreadStartup() 地址:0x8001f952
??() 地址:0x00000000 (无调试信息)

5. 继续按 F11 调试进入 PsReleaseSemaphoreObject 函数。
进入 PsReleaseSemaphoreObject 函数:

```
258 STATUS Status;  
259 PSEMAPHORE Semaphore;  
260  
261 if (ReleaseCount < 1) {  
262     return STATUS_INVALID_PARAMETER;  
263 }  
264  
265 // 由 semaphore 句柄得到 semaphore 对象的指针。  
266 Status = ObRefObjectByHandle(Handle, PspSemaphoreType, (PVOID*)&S;  
267  
268 if (EOS_SUCCESS(Status)) {  
269     Status = PsReleaseSemaphore(Semaphore, ReleaseCount, PreviousC  
270     ObDerefObject(Semaphore);  
271 }  
272  
273 return Status;  
274 }  
275  
276
```

调用堆栈

名称
PsReleaseSemaphoreObject(Handle=0x6, ReleaseCount=0x1, PreviousCount=0x0) 地址:0x80020533
ReleaseSemaphore(Handle=0x6, ReleaseCount=0x1, PreviousCount=0x0) 地址:0x8001158b
Producer(Param=0x0) 地址:0x00401ef9
PspThreadStartup() 地址:0x8001f952
??() 地址:0x00000000 (无调试信息)

6. 先使用 F10 单步调试, 当黄色箭头指向第 269 行时使用 F11 单步调试, 进入 PsReleaseSemaphore 函数。
7. 继续按 F10 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用了一个缓冲区。

查看“记录型信号量”窗口：

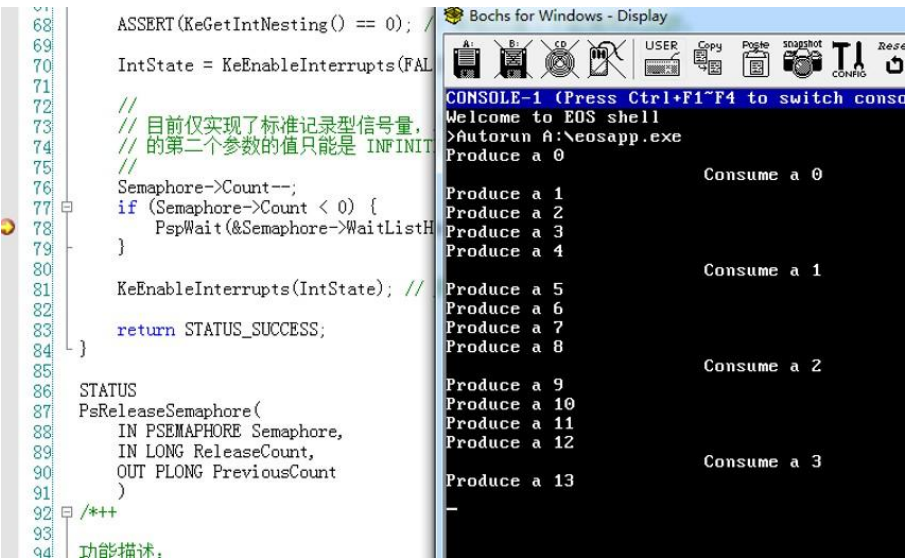
记录型信号量			
数据源: SEMAPHORE Semaphore			
源文件: ps\semaphore.c			
记录型信号量			
序号	信号量的整型值 (Count)	允许最大值 (MaximumCount)	阻塞线程链表 (Wait)
1	0x9	0xa	NULL
2	0x1	0xa	NULL

3.3.2.3 等待信号量（阻塞）

由于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，下面调试这种情况。

- 1. 结束之前的调试。
- 2. 删除所有的断点。
- 3. 在 semaphore.c 文件中的 PsWaitForSemaphore 函数的 PspWait(&Semaphore->WaitListHead, INFINITE); 代码行（第 78 行）添加一个断点。
- 4. 按 F5 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。

中断结果如下：



- 5. 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。

查看“调用堆栈”窗口结果如下：

调用堆栈	
名称	
➤	PsWaitForSemaphore(Semaphore=0x803fadf0, Milliseconds=0xffffffff) 地址:0x80020368
	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)

6. 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。

查看“记录型信号量”窗口结果如下：

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

7. 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，表示生产者线程正在尝试生产 14 号产品。

查看 Producer 函数如下：

131	int i;
132	int InIndex = 0;
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。
148	//
149	Sleep(500);
150	}
151	
152	return 0;

调用堆栈	
名称	
➤	PsWaitForSemaphore(Semaphore=0x803fadf0, Milliseconds=0xffffffff) 地址:0x80020368
	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)

8. 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，这与之前调试的结果是一致的。
查看虚拟机窗口如下：

```
>Autorun A:\test5eos.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
Consume a 5
Produce a 15
```

3.3.2.4 释放信号量（唤醒）

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。可以按照下面的步骤调试：

1. 删除所有断点。
2. 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。
3. 按 F5 继续调试，会在断点处中断。刷新“记录型信号量”窗口，可以看到此时生产者线程仍然阻塞在 Empty 信号量上。

显示结果如下：

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	
2	0x9	0xa	NULL

item

thread	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。
Consumer 函数中变量 i 的值如下：

CreateSemaphore	{HANDLE (LJNG, LJNG, FDIK)} 0x8001151b \CreateSemaphore>	HANDLE (L...
i	0x4	int

5. 按照 3.3.2.2 中的方法使用 F10 和 F11 调试进入 PsReleaseSemaphore 函数。
调试过程如下：

```

105 |
106 | /*
107 | {
108 |     STATUS Status;
109 |     BOOL IntState;
110 |
111 |     IntState = KeEnableInterrupts(FALSE); // 开始原子操作,
112 |
113 |     if (Semaphore->Count + ReleaseCount > Semaphore->MaximumCount)
114 |     {
115 |         Status = STATUS_SEMAPHORE_LIMIT_EXCEEDED;
116 |     } else {
117 |
118 |         //
119 |         // 记录当前的信号量的值。
120 |         //
121 |         if (NULL != PreviousCount) {
122 |             *PreviousCount = Semaphore->Count;
123 |         }
124 |
125 |         //
126 |

```

调用堆栈

名称
PsReleaseSemaphore(Semaphore=0x803fae30, ReleaseCount=0x1, PreviousCount=0x0) 地址:0x8001158b
PsReleaseSemaphoreObject(Handle=0x6, ReleaseCount=0x1, PreviousCount=0x0) 地址:0x8001158b
ReleaseSemaphore(Handle=0x6, ReleaseCount=0x1, PreviousCount=0x0) 地址:0x00401ef9
Producer(Param=0x0) 地址:0x00401ef9
PspThreadStartup() 地址:0x8001f952
??0 地址:0x00000000 (无调试信息)

- 查看 PsReleaseSemaphore 函数中 Empty 信号量计数 (Semaphore->Count) 的值为-1, 和生产者线程被阻塞时的值是一致的。
- 按 F10 单步调试 PsReleaseSemaphore 函数, 直到在代码行 (第 132 行) PspWakeThread(&Semaphore->WaitListHead, STATUS_SUCCESS);处中断。此时, 刷新“记录型信号量”窗口, 可以看到 Empty 信号量计数的值已经由-1 增加为了 0, 需要调用 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。按照下面的步骤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:
查看“记录型信号量”窗口结果如下:

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

- 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。

5. 结束此次调试。
调试虚拟机窗口如下：

```
CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
Welcome to EOS shell
>Autorun A:\neosapp.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
```

查看相关信号量的状态：

记录型信号量



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

记录型信号量

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	0x0	0xa	NULL
2	0xa	0xa	NULL

3.4 修改 EOS 的信号量算法

3.4.1 要求

在目前 EOS Kernel 项目的 ps/semaphore.c 文件中，PsWaitForSemaphore 函数的 Milliseconds 参数只能是 INFINITE，PsReleaseSemaphore 函数的 ReleaseCount 参数只能是 1。现在要求同时修改 PsWaitForSemaphore 函数和 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。

3.4.2 提示

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。

修改代码结果如下：

```
69  
70     IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁  
71  
72     //  
73     // 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  
74     // 的第二个参数的值只能是 INFINITE。  
75     //  
76     STATUS now_status;  
77     if (Semaphore->Count > 0)  
78     {  
79         Semaphore->Count--;  
80         now_status = STATUS_SUCCESS;  
81     }  
82     else  
83     {  
84         if (Semaphore->Count == 0)  
85         {  
86             now_status = PspWait(&Semaphore->WaitListHead, Mi  
87         }  
88     }  
89     KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。  
90     return now_status;  
91     /*  
92     Semaphore->Count--;
```

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。

修改代码结果如下：

```
147         if (Semaphore->Count <= 0) {
148             PspWakeThread(&Semaphore->WaitListHead, STATUS_SU
149         }
150         */
151         if (ReleaseCount > 0)
152         {
153             Semaphore->Count++;
154             while ((!ListIsEmpty(&Semaphore->WaitListHead)) &
155             {
156                 PspWakeThread(&Semaphore->WaitListHead, STATI
157                 PspThreadSchedule();
158                 ReleaseCount--;
159             }
160             Semaphore->Count = Semaphore->Count + ReleaseCour
161             Status = STATUS_SUCCESS;
162         }
```

3.4.3 测试方法

修改完毕后，可以按照下面的方法进行测试：

1. 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。
2. 按 F5 调试执行原有的生产者—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。
3. 将 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");
}
4. 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");
}
5. 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。
6. 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 13-7。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。

查看替换前的情况：

```
CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
Produce a 11
Produce a 12
Consume a 3
Produce a 13
Produce a 14
Produce a 15
Produce a 16
Consume a 14
Produce a 17
Produce a 18
Produce a 19
Consume a 15
Produce a 20
Produce a 21
Consume a 16
Produce a 22
Produce a 23
Consume a 17
Produce a 24
Produce a 25
Consume a 18
Produce a 26
Produce a 27
```

替换后的情况（出现了 timeout）：

```
CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
Producer wait for empty semaphore timeout
Producer wait for empty semaphore timeout
Consume a 16
Produce a 22
Produce a 23
Producer wait for empty semaphore timeout
Producer wait for empty semaphore timeout
Producer wait for empty semaphore timeout
Consume a 17
Produce a 24
Produce a 25
Producer wait for empty semaphore timeout
Producer wait for empty semaphore timeout
Producer wait for empty semaphore timeout
Consume a 18
Produce a 26
Produce a 27
Producer wait for empty semaphore timeout
Producer wait for empty semaphore timeout
Producer wait for empty semaphore timeout
Consume a 19
Produce a 28
Produce a 29
```

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：这两个函数涉及信号量机制，与操作系统的状态有关，为了防止操作系统出现差错，要将它们设定为原子操作，这样可以最大程度地保证系统的平稳运行。

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：调试截图见上文

5. 总结和感想体会

① 通过本次实验，我对于 EOS 信号量机制的了解更加深入，对于课程学习的生产者-消费者问题也有了不同的了解，通过信号量的创建加减等操作，以及编程解决生产者消费者问题，我对于这一块内容的了解得到了升华。

② 在 EOS 信号量算法的调试过程中，我增加了等待唤醒的部分，理解了有限等待原则的实际应用，也对于进程间的同步有了更深的认识。

③ 通过跟踪调试 EOS 信号量，我看到了进程同步过程中对于临界资源的不断调用，以及冲突的避免，通过调节，进一步提升了资源的利用率。