

北京科技大学实验报告

学院：计算机与通信工程学院 专业：计算机科学与技术 班级：计 1703

姓名：张宝丰

学号：41724081

实验日期：2019 年 12 月 1 日

实验名称：操作系统实验 3 线程同步（5 分）

实验目的：以一个教学型操作系统 EOS 为例，深入理解线程（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. 消费者-生产者问题的代码实现

解决消费者-生产者问题需要三个信号量：Empty、Full、Mutex。初始时 Empty=N，Full=0，Mutex=1。生产者每次生产产品时，需要确认有空位置（缓冲区）可以存放产品，因此先等待一个 Empty 信号量，之后占用 Mutex 互斥操作，在完成生产后，释放 Mutex，并释放一个 Full 信号量，允许消费者进行消费。

消费者的代码如下，方框圈出的就是临界区。

```
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);

//
```

生产者的操作与之类似，只不过是 Empty 和 Full 调换了一下位置，代码如下：

```

WaitForSingleObject(FullSemaphoreHandle, INFINITE);
WaitForSingleObject(MutexHandle, INFINITE);

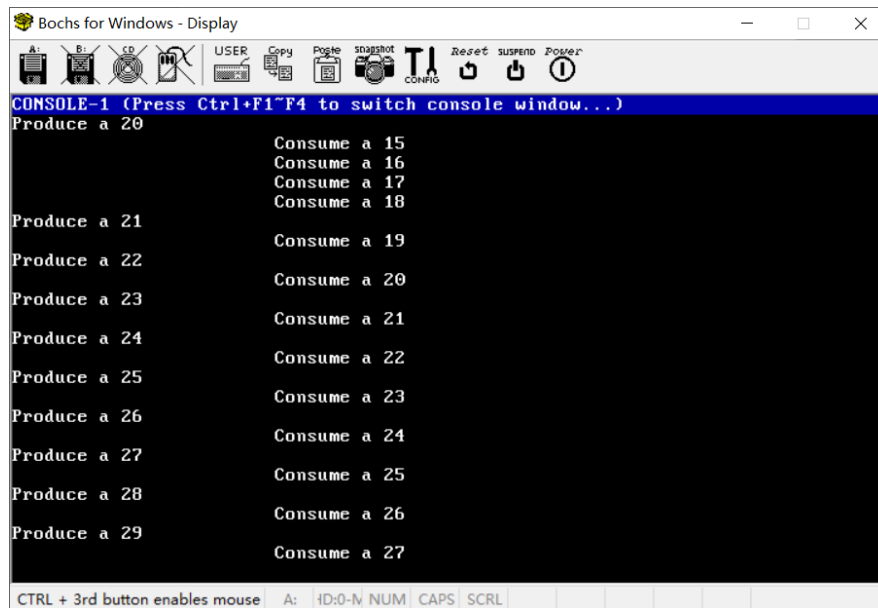
printf("\t\t\tConsume a %d\n", Buffer[OutIndex]);
OutIndex = (OutIndex + 1) % BUFFER_SIZE;

ReleaseMutex(MutexHandle);
ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL);

//

```

下图是程序调试的结果，从中可以看到，生产者生产和消费者消费交替进行。



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)

1. EOS 信号量的实现

EOS 的信号量通过 PSEMAPHORE 结构定义，主要的域为 Count，即信号量的数值，其初始化代码如下：

VOID

PsInitializeSemaphore(

IN PSEMAPHORE Semaphore, // 要初始化的信号量结构指针

IN LONG InitialCount,

IN LONG MaximumCount

)

{

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

Semaphore->Count = InitialCount;

```

Semaphore->MaximumCount = MaximumCount;
ListInitializeHead(&Semaphore->WaitListHead);
}

```

可以从代码中看到，信号量的初始化操作包含初值设定、最大值设定，并在最后为之创建一个等待队列，用于存放因该信号量而阻塞的线程的指针，这在之后的 PV 操作中会用到。

P 操作最终由 PsWaitForSemaphore 实现，该函数在 semaphore.c 中定义，主要操作如下：

```

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

```

```

Semaphore->Count--;

```

```

if (Semaphore->Count < 0) {

```

```

    PspWait(&Semaphore->WaitListHead, INFINITE);

```

```

}

```

```

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

```

可以从代码中看到，P 操作通过关中断的方式实现（V 操作也类似），这只适用于单处理机的情况。P 操作将信号量减一，若减一之后信号量小于 0，说明资源不足，线程进入等待状态。

V 操作最终由 PsReleaseSemaphore 实现，它也是通过关中断来实现，关闭中断后修改信号量的值。考虑到可能有线程被唤醒，故需要在信号量释放之后做线程调度，保证高优先级线程立即得到执行。

```

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;

```

```

    }

```

```

    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。

```

2. 跟踪 EOS 信号量的工作过程

在 main 函数调用 CreateSemaphore 的代码行加断点，可以由此进入信号量的创建过程。

创建信号量的过程是：调用 `PsCreateSemaphoreObject` -> 创建信号量对象 -> 调用 `PsInitializeSemaphthore` 进行初始化。信号量的 P 操作通过 `WaitForSingleObject` 实现，在调用完 `WaitForSingleObject` 之后信号量数值会减 1，若减 1 之后的信号量数值 < 0，则说明资源不足，线程进入等待状态（调用 `PspWait`），并加入等待队列队尾；信号量的 V 操作通过 `ReleaseSemaphore` 实现，若信号量 >= 0，则直接把信号量 +1；若信号量 < 0，说明由线程因为资源不足而阻塞，这时会从阻塞队列的队首开始唤醒线程，并为之分配相应的资源。

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）

超时唤醒的实现：

为了加入超时唤醒功能，规定信号量不能小于 0，这样当资源不足（信号量=0）是，调用 `PspWait` 函数，并设定等待时间（`Milliseconds`），当超过预定等待时间，`Pspwait` 会返回超时状态（`WAIT_TIMEOUT`），这样就可以在外部应用程序中识别超时了。具体的代码如下：

```
if(Semaphore->Count>0){
    Semaphore->Count--;
    myStatus = STATUS_SUCCESS;
}

if(Semaphore->Count==0){
    myStatus = PspWait(&Semaphore->WaitListHead,Milliseconds);
}
```

批量释放的实现：

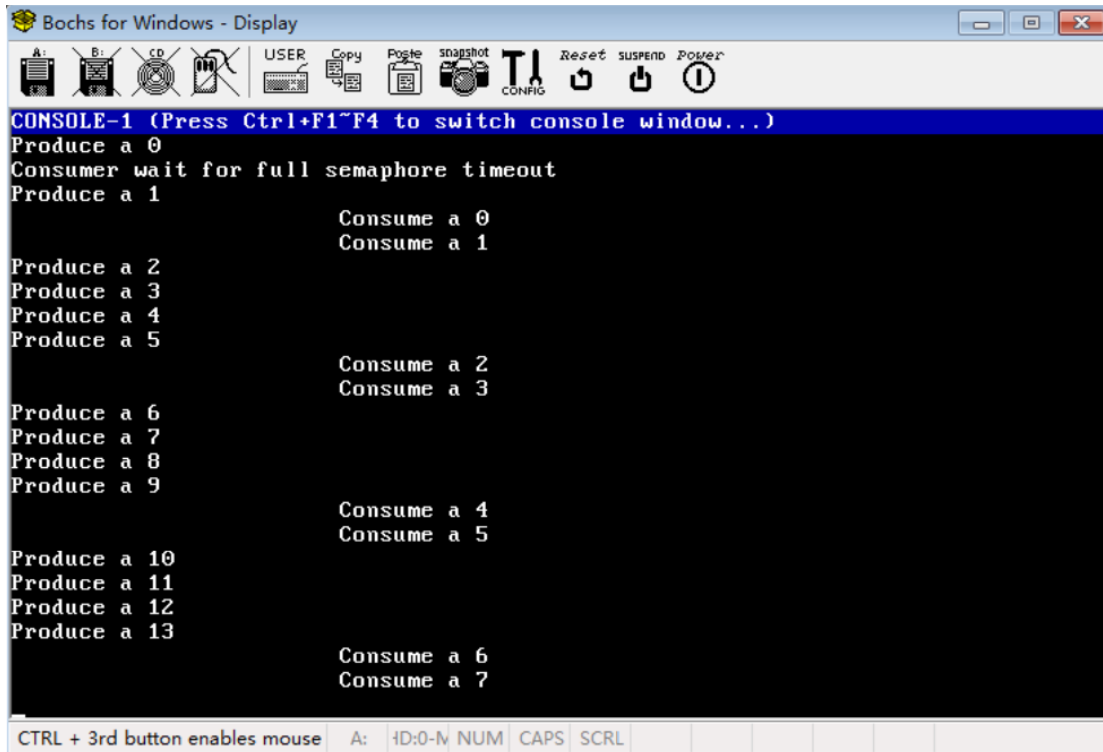
批量释放功能允许一个线程同时占用多个信号量，并可以同时释放多个信号量，亦唤醒多个信号量。实现思路就是在逐个释放信号量的时候，检测该信号量的等待队列，如果等待队列不为空，就唤醒被阻塞的线程，代码如下：

批量释放：

```
int i;
for(i=ReleaseCount;i>0;i--){
    // 检查 waitlist 不为空
    if(!ListIsEmpty(&Semaphore->WaitListHead)){
        PspWakeThread(&Semaphore->WaitListHead, STATUS_SUCCESS);
    }
    else{
        Semaphore->Count += i;
        break;
    }
}
```

}

验证结果如下图，可以看到在第二行，消费线程



```
Bochs for Windows - Display
A: B: CD USER Copy Paste snapshot CONFIG Reset SUSPEND Power
CONSOLE-1 (Press Ctrl+F1~F4 to switch console window...)
Produce a 0
Consumer wait for full semaphore timeout
Produce a 1
Consume a 0
Consume a 1
Produce a 2
Produce a 3
Produce a 4
Produce a 5
Consume a 2
Consume a 3
Produce a 6
Produce a 7
Produce a 8
Produce a 9
Consume a 4
Consume a 5
Produce a 10
Produce a 11
Produce a 12
Produce a 13
Consume a 6
Consume a 7
CTRL + 3rd button enables mouse A: 4D:0-M NUM CAPS SCRL
```

结果分析：

EOS 对于依赖于关中断实现原子操作，这个实现方式只使用与单处理机，同时若关中断时间过长，则有可能导致系统无法响应外部中断，影响系统的稳定性。EOS 的 P 操作将信号量减一，若减一之后信号量小于 0，线程进入等待状态。V 操作将信号量加一，若有线程阻塞，则唤醒线程，并在在信号量释放之后做线程调度，保证高优先级线程立即得到执行。

我在实现超时唤醒功能时用到了系统提供的 PspWait 函数，该函数会检测线程等待时间，若等待时间超过预定时间，则返回 WAIT_TIMEOUT。在实现批量释放功能时，则利用了 EOS 为每个信号量定义的阻塞队列，当有信号量被释放的时候，检测阻塞队列中是否为空，若不为空，则唤醒队首的线程，上面的两个功能经过测试，均运行正常。