进程同步-1.1

教师：姜博
E-Mail: gongbell@gmail.com

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

进程同步

- 间接相互制约。主要原因是资源共享。
- 直接相互制约。主要源于进程合作。
- 进程同步：指多个相关进程在执行次序上的协调，用于保证这种关系的相应机制称为同步机制。

典型的并发错误

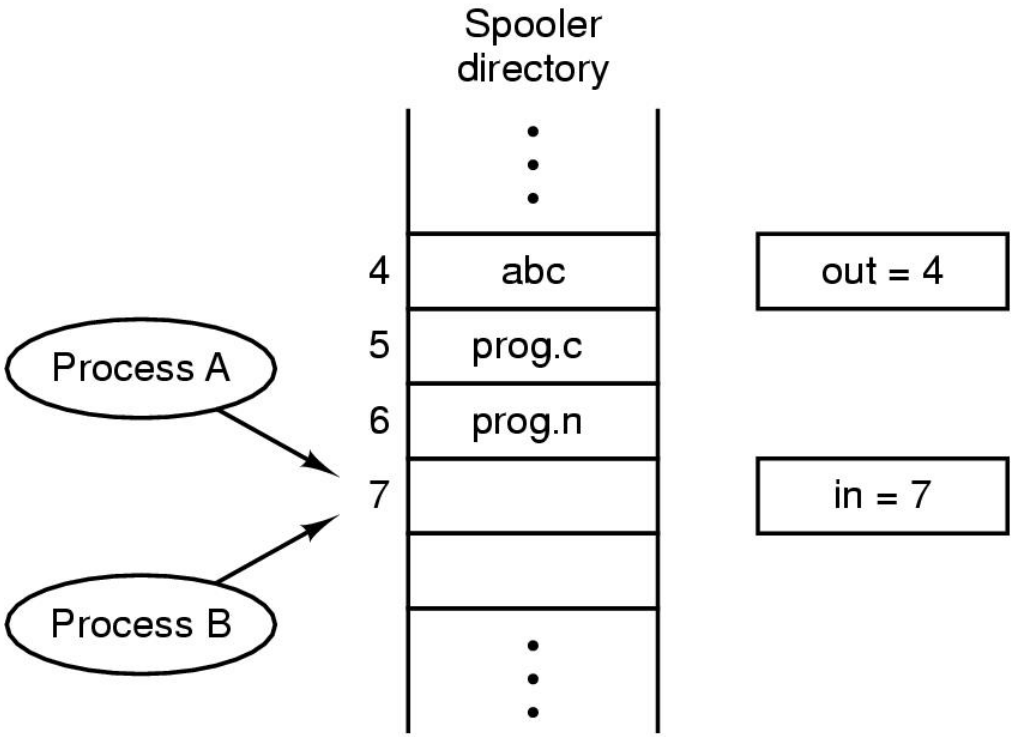
- 数据竞争
  - 多个进程对共享变量的无保护的同时访问
- 原子性违反
  - 原子性：一组事务操作要么都做，要么都不做
  - 例如：转账和账户金额的变更
- 死锁
  - 如果一个进程集合中的每个进程都在等待只能由该进程集合中其他进程才能引发的事件，那么该进程集合就是死锁的。

数据竞争（data race ）

- 数据竞争：多个进程同时访问同一个共享变量，并且至少有一个是写。
  - 没有机制防止进程对共享变量的同时访问
  - 最终结果取决于进程访问的次序。
  - 读读没有问题，读写，写写有问题。
- 在同一时刻，只允许一个进程访问该共享数据，即如果当前已有一个进程正在使用该数据，那么其他进程暂时不能访问。这就是互斥的概念
  - 互斥是避免数据竞争的一个手段。

临界区

- 临界资源
  - 这种一次只允许一个进程使用的资源称为临界资源。例如打印机、共享变量。
  - 假脱机打印（spooling）- 把独占的临界资源共享访问



原子性违反（Atomicity Violation）

```
P1: R1:=count;
    R1:=R1+1;
    count:=R1;

P2: R2:=count;
    R2:=R2+1;
    count:=R2;
```

Count:=2

```
P1: R1:=count;
    R1:=R1+1;
    count:=R1;

P2: R2:=count;
    R2:=R2+1;
    count:=R2;
```

Count:=1

原子性违反

为什么结果无法重现?

- 调度产生了不确定性，导致产生：
  - 数据竞争
  - 原子性违反
- 数据竞争：多个进程在没有同步保护的情况下访问同一个共享变量， 而且至少有一个是写。
- 数据竞争非常难以循环调试：每次执行不确定，调试的时候可能会消失，Heisenbug。

不可再现性的原因

- 不同的环境输入（传感器）
- 不同的API返回（例如系统时间）
- 不同的调度时间
- 中断的时间
- 不同的共享内存的顺序

- 执行重放技术

临界区的定义

- 对临界资源（如共享变量）进行访问的程序片段称为临界区
  - 使两个进程不同时处于临界区-->避免竞争

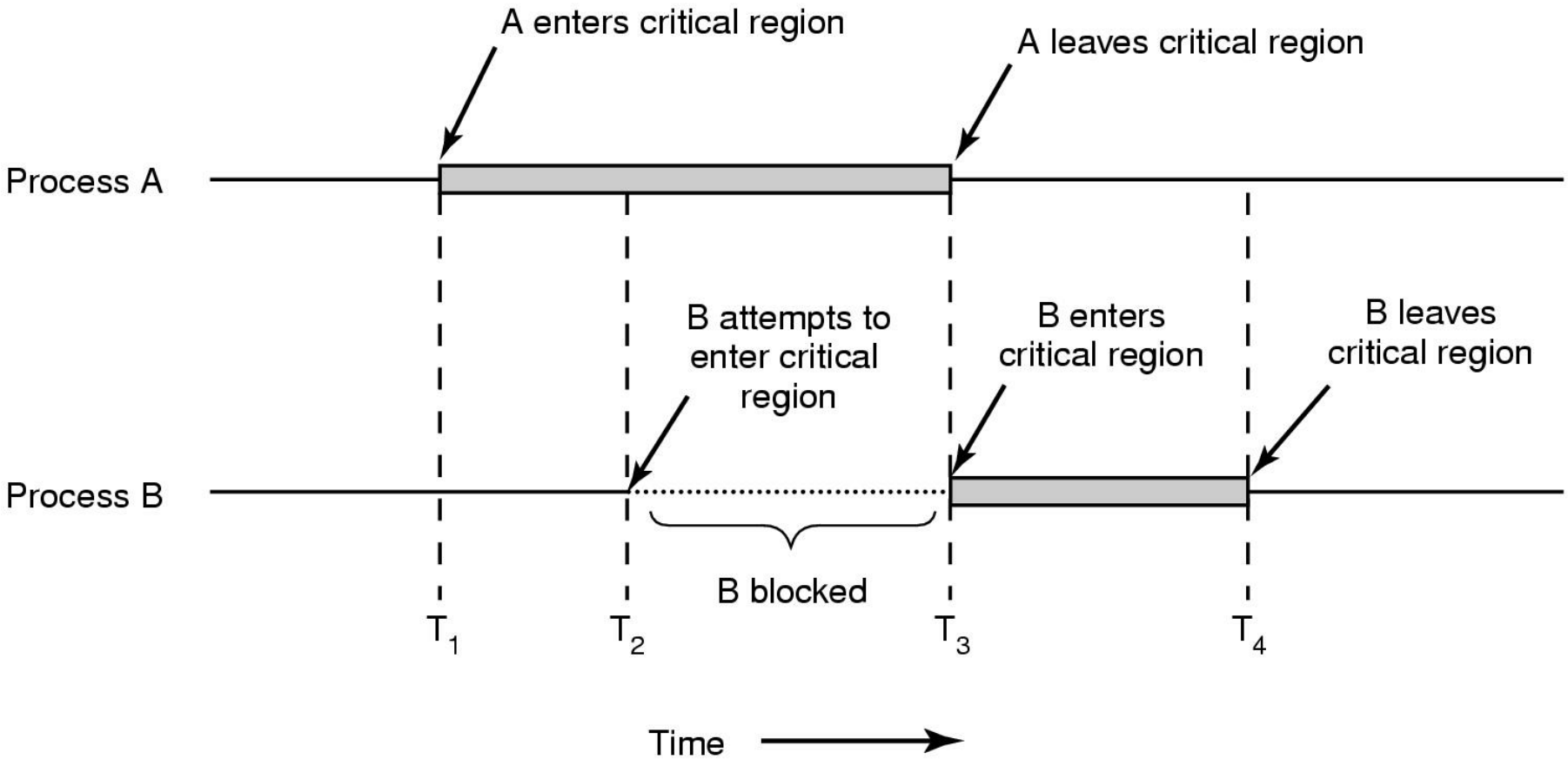
```
repeat
    enter section
    critical section
    exit section
    remainder section
until false
```

临界区解决方案需要满足的条件

- 任何两个进程都不能同时进入临界区；
- 不能事先假定CPU的个数和运行速度；
- 临界区外的进程不能妨碍其他的进程进入临界区；
- 一个进程不能在临界区外无限等待

类比：临界区 v.s. 十字路口

使用临界区互斥的场景


A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
T1
T2
T3
T4
Time

同步机制应遵循的准则

- 空闲让进
  - 临界资源处于空闲状态，允许进程进入临界区
  - 临界区内仅有一个进程执行
- 忙则等待
  - 有进程正在执行临界区代码，所有其他进程则不可以进入临界区
- 有限等待
  - 对要求访问临界区的进程，应在保证在有限时间内进入自己的临界区，避免死等。
- 让权等待
  - 当进程（长时间）不能进入自己的临界区时，应立即释放处理机，尽量避免忙等。

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

简单的尝试

- 屏蔽中断
  - 进入临界区关闭中断，离开临界区打开中断
  - 临界区中可以独自访问共享变量
  - 时钟中断和各种外设的中断会关闭
  - 进程的切换停止，当前进程会独占CPU
- 问题?

简单的尝试 - 屏蔽中断

- 屏蔽中断
  - 进入临界区关闭中断，离开临界区打开中断
  - 时钟中断会关闭，进程切换停止
- 问题?
  - 让用户进程可以关闭中断不安全
    - 如果一个进程关闭中断后再也不打开中断。。。
  - 关闭中断仅仅能关闭一个CPU，多核情况下?
    - disable 指令针对一个核
  - 临界区可能运行时间很长-影响调度
- 对于内核本身非常有用
  - 例如，防止在更新进程控制块链表时被中断而不一致

简单的尝试 - 共享锁变量

- 锁变量：设置共享变量lock
  - 锁：0表示临界区无进程，1表示临界区有进程
  - 锁变量初始值0
  - 进程要进入临界区，测试（read）这把锁，如果锁的值为0，进程就将其设置(write)为1，并进入临界区
  - 如果锁已经为1，则需要等待其变为0
- 问题?

简单的尝试 - 共享锁变量

- 锁变量：设置共享变量lock
  - 锁：0表示临界区无进程，1表示临界区有进程
  - 锁变量初始值0
  - 进程要进入临界区，测试（read）这把锁，如果锁的值为0，进程就将其设置(write)为1，并进入临界区
  - 如果锁已经为1，则需要等待其变为0
- 问题?
  - 第一个进程读入锁变量，发现其为0，但置1前被切换
  - 第二个进程读入锁变量，设置为1，进入临界区
  - 第一个进程再次被调度，再次置1，也进入临界区！

简单的尝试 - 共享锁变量

- 锁变量：设置共享变量lock
  - 锁：0表示临界区无进程，1表示临界区有进程
  - 锁变量初始值0
  - 进程要进入临界区，测试这把锁，如果锁的值为0，进程就将其设置为1，并进入临界区

如何保证锁变量访问本身的原子性？
设置锁需要读取和设置，是一个原子操作

  - 第二个进程读入锁变量，设置为1，进入临界区
  - 第一个进程再次被调度，再次置1，也进入临界区！

进程同步-1.2

教师：姜博
E-Mail: gongbell@gmail.com

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

简单的尝试 - 共享锁变量

- 锁变量：设置共享变量lock
  - 锁：0表示临界区无进程，1表示临界区有进程
  - 锁变量初始值0
  - 进程要进入临界区，测试这把锁，如果锁的值为0，进程就将其设置为1，并进入临界区

如何保证锁变量访问本身的原子性？
设置锁需要读取和设置，是一个原子操作

  - 第二个进程读入锁变量，设置为1，进入临界区
  - 第一个进程再次被调度，再次置1，也进入临界区！

严格轮换法-翻牌子

- 设立一个公用整型变量 turn：描述允许进入临界区的进程标识
  - 在进入临界区循环检查是否允许本进程进入：turn为0时，进程0可进入；turn为1时，进程1可进入；
  - 在退出区修改允许进入进程标识：进程0退出时，改turn为1；进程1退出时，改turn为0

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

严格轮换法

- 设立一个公用整型变量 turn：描述允许进入临界区的进程标识
  - 在进入临界区循环检查是否允许本进程进入：turn为0时，进程0可进入；turn为1时，进程1可进入；
  - 在退出区修改允许进入进程标识：进程0退出时，改turn为1；进程1退出时，改turn为0

忙等:反复测试变量。耗费CPU。

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

严格轮换法

- 严格轮换要求两个进程交替执行
- 如果进程0执行完，turn变成了1
- 这时候如果进程1的临界区之外的工作很多（比如IO），上次循环还没结束，进程0就只能等待了。
- 如果两个进程速度快慢不匹配，轮流进入不合适

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

严格轮换法的问题

- 强制轮流进入临界区，没有考虑进程的实际需要。效率较低
- 违反规则：一个进程可能被一个不在临界区的进程阻塞

Peterson算法

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Peterson算法

初始，如果进程0进入临界区，设置interested 和 turn，
由于1初始不感兴趣，0进入临界区。

```
int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
    int other;                              /* number of the other process */

    other = 1 – process;                    /* the opposite of process */
    interested[process] = TRUE;             /* show that you are interested */
    turn = process;                         /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
    interested[process] = FALSE;            /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Peterson算法

进程0进入临界区后，进程1如果想进入临界区，由于0 的interested为真，所以进程1等待

```
int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
    int other;                              /* number of the other process */

    other = 1 – process;                    /* the opposite of process */
    interested[process] = TRUE;             /* show that you are interested */
    turn = process;                         /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
    interested[process] = FALSE;            /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Peterson算法

如果同一个时刻两个进程都进入临界区，interested和turn是竞争访问，后设置turn的进程会循环，先设置的进入

```
int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
    int other;                              /* number of the other process */

    other = 1 – process;                    /* the opposite of process */
    interested[process] = TRUE;             /* show that you are interested */
    turn = process;                         /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
    interested[process] = FALSE;            /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Peterson算法

问题：Peterson算法在抢占式和非抢占式调度下都可行么？

```
int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
    int other;                              /* number of the other process */

    other = 1 – process;                    /* the opposite of process */
    interested[process] = TRUE;             /* show that you are interested */
    turn = process;                         /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
    interested[process] = FALSE;            /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Peterson算法

问题：Peterson算法在抢占式和非抢占式调度下都可行么？
抢占式OK.非抢占式No:可能先运行进程霸占CPU。

```
int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
    int other;                              /* number of the other process */

    other = 1 – process;                    /* the opposite of process */
    interested[process] = TRUE;             /* show that you are interested */
    turn = process;                         /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
    interested[process] = FALSE;            /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

硬件指令机制

- 考虑锁变量思路的问题
  - 问题：无法保证锁变量的读和设置是原子操作
- Test and Set Lock指令
  - IBM370系列机器中称为TSL；在INTEL8086中成为XCHG指令。
  - TSL：将lock变量读入寄存器，然后置位1
  - 测试（读）并加锁（写）：保证读写是一个原子操作
  - 执行TSL指令的CPU会通过锁住内存总线，禁止其他CPU访问内存
- 利用TSL实现进程互斥

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was not zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

- XCHG：原子性的交换两个位置的内容

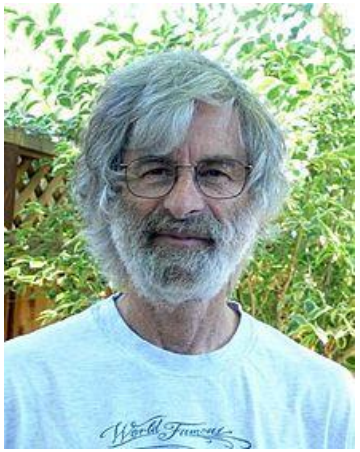
硬件指令机制

- 进入临界区之前调用enter_region，忙等直到获得锁，将锁置为1
- 从临界区返回调用leave_region，将锁置为0

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...



Leslie Lamport
2013 Turing Award

支持任意数量进程，每个进程进入临界区后，分配一个序号，为防止重号，每个进程先跟所有其他进程比较序号，如果自身号码最小则继续，否则等待；如果序号相同，则比较进程号。

http://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm

Bakery Algorithm

- Notation <≡ lexicographical order (ticket #, process id #)
  - (a,b) < (c,d) if a < c or if a = c and b < d
  - max (a_0,..., a_{n-1}) is a number, k, such that k ≥ a_i for i - 0, ..., n – 1
- Shared data

  boolean choosing[n];

  int number[n];

  Data structures are initialized to false and 0 respectively

Bakery Algorithm

```
do {
    choosing[i] = true; //选号
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false; //选号结束
    for (j = 0; j < n; j++) {
            while (choosing[j]) ; //其他进程选号，等待
            while ((number[j] != 0) && ((number[j], j) < (number[i], i))) ; //其他进程离开临界区则可以进入
    } //多个要进入进程，进程i的序号和进程号组合最小才能进入
      critical section
      number[i] = 0;
      remainder section
} while (1);
```

Bakery Algorithm

```
do {
    choosing[i] = true; //选号
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false; //选号结束
    for (j = 0; j < n; j++) {
            while (choosing[j]) ; //其他进程选号，等待
            while ((number[j] != 0) && ((number[j], j) < (number[i], i))) ; //其他进程离开临界区则可以进入
    } //多个要进入进程，进程i的序号和进程号组合最小才能进入
    critical section
} while (1);
```

基本思路：按照需求给进程分配一个先后顺序进入临界区

Peterson，TSL XCHG算法的问题

- 如果不能满足进入临界区条件，都要进行忙等，
- 忙等，不仅仅浪费CPU时间
- 还会导致优先级反转
  - 低优先级进入临界区，高优先级等待的情况
  - 优先级反转的场景
    - H 高优先级 L 低优先级进程
    - 调度规则：H就绪就可以优先运行
    - L在临界区中，H就绪，然后H被优先调度，L被切换
    - H开始在临界区外忙等，但L没机会被调度，无法离开临界区
    - H，L永远等下去

优先级反转

- 解决忙等的一个思路？ - 通知的机制
  - 使用sleep和wakeup
  - sleep阻塞调用进程，进程挂起直到另一个进程唤醒他
  - wakeup唤醒一个进程
- 无法进入临界区，则sleep，使得低优先级进程能够被调度离开临界区，L离开时发送wakeup唤醒H

优先级反转

- 问题:在多线程的场景下，优先级反转是否会发生在用户级线程的实现？为什么？

优先级反转

- 问题:在多线程的场景下，优先级反转是否会发生在用户级线程的实现？为什么？
- 优先级反转发生是由于低优先级进程运行在临界区时由于高优先级进程就绪而被切换。如果高优先级进程使用忙等的策略尝试进入临界区，他就会忙等下去，低优先级进程不会有机会离开临界区。
- 如果使用用户级线程，低优先级线程不会被高优先级线程抢占，因为抢占发生在进程级别。但是对于内核级线程的实现，这个是可能发生的。

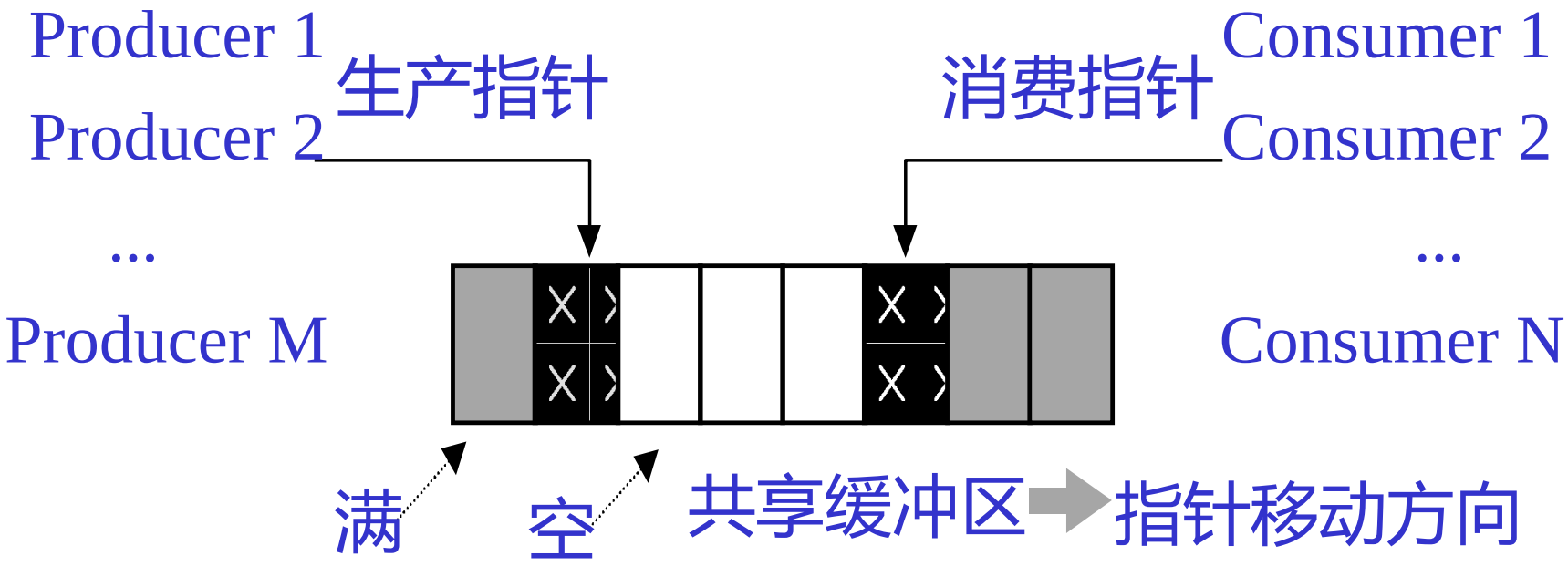
解决之道

- 忙等->休眠（sleep）
- Sleep
- Wakeup

- E.g.银行排队
  - 忙等：看到队很长，坚持排队
  - 阻塞：看到队很长，先回家歇会儿（sleep），有空柜台了，大堂经理（scheduler）电话通知再过来（wakeup）

生产者 - 消费者问题(the producer-consumer problem)

- 问题描述：若干进程通过有限的共享缓冲区交换数据。其中，“生产者”进程不断写入，而“消费者”进程不断读出；共享缓冲区共有N个；任何时刻只能有一个进程可对共享缓冲区进行操作。
- 经典并发程序设计问题的抽象:
  - 在线音乐/视频播放器:网络线程和播放线程
  - OS的设备管理，OS负责读取设备数据到缓冲区，使用设备的进程从缓冲区读取数据。


Producer 1
Producer 2
...
Producer M
生产指针
消费指针
Consumer 1
Consumer 2
...
Consumer N
满
空
共享缓冲区
指针移动方向

生产者和消费者问题（有界缓冲区）

- 简单的（但有问题的）实现：
  - 当缓冲区满，让生产者睡眠，等待消费者取出数据再唤醒生产者
  - 当缓冲区空，让消费者睡眠，等待生产者放入数据再唤醒消费者
  - 使用count变量来追踪缓冲区满或者空的状态

生产者和消费者问题 - sleep&wakeup

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Sleep&wakeup算法的问题

- 对count的读取和睡眠应该是不能中断的原子操作!
- 缓冲区为空---消费者读取count发现为0---消费者被切换---生产者被调度---生产者加入数据，count＝1---推断消费者在睡眠，就wakeup消费者
- 但是消费者只是被切换，还没睡眠---wakeup信号丢失
- 消费者被调度---发现之前读取的count为0---睡眠
- 生产者被调度---不断填满缓冲区----睡眠
- 两个进程全都睡眠!

Sleep&wakeup算法的问题

- 问题：发给未睡眠的线程的wakeup信号丢失了
- 解决：记住唤醒信息---增加唤醒等待位：
  - 如果发送wakeup给一个未睡眠的线程，置位
  - 睡眠前检查唤醒等待位，如果被置位了，就不睡，清除唤醒等待位
- 如果更多进程要合作：
  - 需要更多唤醒等待位---解决方案不够完美

- 更合理的方案:将读取count为0和睡眠放入原子操作，并且记住唤醒操作。

信号量机制

- 如何安全的避免忙等?
- 1965年Dijkstra提出了新的变量类型Semaphore（信号量）
- S >= 0 表示当前可用的资源的数目
- 对信号量P（S）& V（S）操作原语。P、V分别是荷兰语的test(Proberen)和increment(Verhogen))
- P: down V：up
- S的初值
  - (S=1) 实现互斥：二元信号量(等于mutex)
  - (S>1) 实现同步：通用信号量

信号量的使用：

- 必须置一次且只能置一次初值（代表资源的个数）
- 只能由P、V操作来改变
- P/V操作是原语：原子操作不会被打断
  - 原子操作：一组相关的操作要么都执行要么都不执行
  - 例如，A---B的转账操作
  - 数据库的事务

物理意义

- P（down）操作分配资源：检查信号量初值是否大于0，如果大于0，减1，继续执行；如果等于0，进程被直接阻塞（将当前进程从运行队列移动到信号量的队列），减1的操作暂时不做。
  - P是原子操作：一组操作，要么都执行，要么都不执行
  - 当S=1，A和B同时调用P(S),只有一个进程能够完成P(S)
  - 实现时，操作前关闭中断，操作后打开中断。
- 当进程执行P操作阻塞到某个信号量，进程不在运行态，所以在唤醒前都不会占用CPU资源。

物理意义

- V（up）操作释放资源：首先将信号量S增加1 （原子操作)。但是如果有一个或者多个进程在信号量的队列睡眠（这时S=1），就会随机唤醒一个进程（将进程从信号量的队列移入就绪队列），并使得其运行后能完成P操作的减1，所以这时S还是0。
  - V(S)原子操作
  - 实现时，操作前关闭中断，操作后打开中断。
  - 注意“V(S) != S=S+1
  - 如果S=6，进程A，B任意顺序调用V(S), 那么结果一定是8

信号量机制的实现

- 数据结构：一个整数+一个队列
- 对于单CPU系统；
  - 系统在关闭中断的情况下，测试信号量，更新信号量，将进程睡眠或者唤醒。
- 对于多CPU系统：
  - 需要有一个锁变量来保护信号量，并使用TSL指令或者XCHG指令来访问锁变量，从而保证每次只能有一个CPU检查信号量

信号量的取值能小于0么？

- 答案是和不同的系统实现相关:
  - Linux supports kernel semaphores, POSIX semaphores and System V semaphores.
  - System V semaphore API：信号量不小于0
  - POSIX semaphore API: 信号量为负值的时候，绝对值表示等待的进程数
  - Linux kernel 2.6 之前可以为负值，之后演化非负值
  - 信号量大于0意义一致: 表示可同时获取的资源数目。

信号量机制解决生产者消费者问题

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item();                  /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item();                   /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

北京航

Figure 2-28. The producer-consumer problem using semaphores.

信号量机制

Mutex互斥，同一时刻只有一个进程可以读写缓冲区

Full&empty 同步：缓冲区满，生产者停下；缓冲区空，消费者停下。

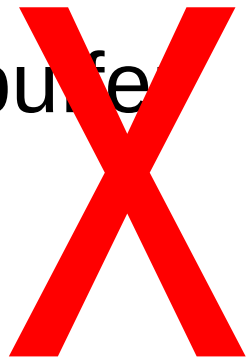
```
#define N 100
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                      /* TRUE is the constant 1 */
        item = produce_item();          /* generate something to put in buffer */
        down(&empty);                   /* decrement empty count */
        down(&mutex);                   /* enter critical region */
        insert_item(item);              /* put new item in buffer */
        up(&mutex);                     /* leave critical region */
        up(&full);                      /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                      /* infinite loop */
        down(&full);                    /* decrement full count */
        down(&mutex);                   /* enter critical region */
        item = remove_item();           /* take item from buffer */
        up(&mutex);                     /* leave critical region */
        up(&empty);                     /* increment count of empty slots */
        consume_item(item);             /* do something with the item */
    }
}
```

北京航

Figure 2-28. The producer-consumer problem using semaphores.

- full是“满”数目，初值为0，empty是“空”数目，初值为N。实际上，full和empty是同一个含义：full + empty == N
- mutex用于访问缓冲区时的互斥，初值是1

生产者	消费者
P(mutex);	P(mutex);
P(empty);	P(full);
one >> buffer	one << buffer
V(full)	V(empty)
V(mutex)	V(mutex)



死锁：如果缓冲区是完全满的，生产者首先将mutex设置为0，
然后生产者会阻塞在empty。
这时候消费者访问缓冲区，会对mutex执行P操作，
就会阻塞在mutex。

生产者	消费者
P(mutex);	P(mutex);
P(empty);	P(full);
one >> buffer	one << buffer
V(full)	V(empty)
V(mutex)	V(mutex)