上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L5-1. 进程同步

宋卓然

上海交通大学计算机系

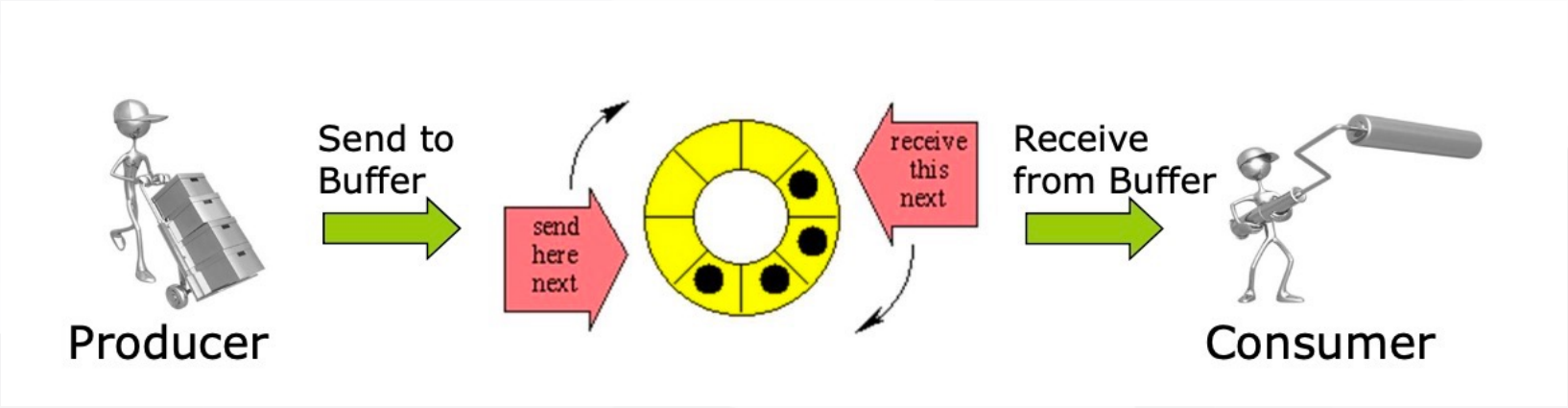
songzhuoran@sjtu.edu.cn

背景

- 进程可以并发访问
- 但并发访问共享数据可能导致数据不一致
- 如何保证数据的一致性、完整性

生产者-消费者问题

- 进程协作的范式，生产者产生由消费者消费的信息
- 一个进程产生的数据，可能被另一个进程所使用



生产者-消费者问题 共享内存

- 共享内存，大小为10

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

生产者-消费者问题 共享内存

- 生产者 Producer；消费者 Consumer

Producer

```
while (true) {
  /* Produce an item */
 while (((in + 1) % BUFFER_SIZE)  == out)
    ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (true) {
  while (in == out)
        ; // do nothing
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
      return item;
}
```

生产者-消费者问题 共享内存

- 生产者 Producer；消费者 Consumer
- 缺陷
  - 最多只有BUFFER_SIZE-1 项
  - 一直处于等待
- 希望允许BUFFER_SIZE

生产者-消费者问题 共享内存

- 生产者 Producer；消费者 Consumer

Producer

```
while (true) {
    /* produce an item */
    while (in % BUFFER_SIZE == out && in != out)
            ;           /* waiting */
    buffer[in % BUFFER_SIZE] = item;
    if ((in + 1) % BUFFER_SIZE == out)
            in = out + BUFFER_SIZE;
    else
            in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (true) {
    while (in == out)
            ; // do nothing

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
return item;
}
```

Consumer代码不变

生产者-消费者问题 共享内存

- 生产者 Producer；消费者 Consumer。限定BUFFER_SIZE项
- 假想为填充所有缓冲区的消费者-生产者问题提供一个更好的解决方案

Producer

```
while (true) {
    /*  produce an item and put in nextProduced  */
    while (counter == BUFFER_SIZE)  ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true)  {
    while (counter == 0) ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /*  consume the item */
}
```

生产者-消费者问题 共享内存

- 允许两个进程并发操作变量counter，导致错误
- 多个进程并发访问和操作同一数据并且执行结果与特定访问顺序有关，称为**竞争条件**
- 为了防止竞争条件，需要确保一次只有一个进程可以操作counter
- 对于抢占式内核，需要确保内核数据结构不会导致竞争条件

T_0:	生产者	执行	$register_1 = counter$	$\{register_1 = 5\}$
T_1:	生产者	执行	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	消费者	执行	$register_2 = counter$	$\{register_2 = 5\}$
T_3:	消费者	执行	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	生产者	执行	$counter = register_1$	$\{counter = 6\}$
T_5:	消费者	执行	$counter = register_2$	$\{counter = 4\}$

临界区问题

- 假设某个系统有n个进程
- 每个进程有一段代码，称为临界区
  - 进程在执行该区时可能修改公共变量、更新一个表、写一个文件
  - 当一个进程在临界区内执行时，其他进程不允许在它们的临界区内执行
- 临界区问题是设计一个协议以便协作进程
- 在进入临界区前， 每个进程应请求许可。实现这一请求的代码区段称为进入区。临界区之后可以有退出区。其他代码为剩余区

```
do {
        entry section          临界区
            critical section
        exit section           剩余区
            remainder section
} while (TRUE);
```

临界区问题

- 临界区问题的解决方案应满足如下三条要求
  - 互斥：如果进程P，在其临界区内执行，那么其他进程都不能在其临界区内执行
  - 进步：如果没有进程在其临界区内执行，并且有进程需要进入临界区，那么只有那些不在剩余区内执行的进程可以参加选择，以便确定谁能进入临界区
  - 有限等待：从一个进程做出进入临界区的请求直到这个请求允许为止，其他进程允许进入其临界区的次数具有上限

临界区问题的解决方案

- Peterson解决方案
- 硬件同步
- 互斥锁
- 信号量

Peterson解决方案

- 适用于两个进程交错执行临界区与剩余区
- 进程共享两个数据项

int turn;

boolean flag[2]

- 变量turn表示哪个进程可以进入临界区。即如果turn==i，那么进程Pi允许在临界区内执行 。数组flag表示哪个进程准备进入临界区。例如，flag[i]为true，那么进程Pi准备进入临界区
- turn的最终值决定了哪个进程允许先进入临界区

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[ j] && turn == j);
              critical section
        flag[i] = FALSE;
              remainder section
} while (TRUE);
```

Peterson解决方案

- 证明
  - 互斥成立
    - turn不可能同时等于i和j
  - 进步要求成立
    - 当Pj退出临界区时，会设置flag[j]=false，允许Pi进入临界区
  - 有限等待成立
    - Pi进程在Pj进程进入临界区后最多一次就能进入临界区

Peterson解决方案

- Peterson无法在现代计算机架构上“正确”运行
- 因为程序员、处理器、编译器通常为了性能，会将没有依赖性的语句调换顺序
- 此时使用Peterson无法解决一致性问题，导致无法预料的结果

Peterson解决方案

- 假定两个线程共享数据：

  boolean flag = false;

  int x = 0;

- 线程1执行：

  while (!flag)

  ;

  print x

- 线程2执行：

  x = 100;

  flag = true

预期的输出结果是?

Peterson解决方案

- 100为预期输出
- 但当将线程2的两条语句顺序置换：

```
flag = true;
x = 100;
```

- 输出可能变为0！

硬件同步

- 所有方案都是基于加锁为前提的，通过锁来保护临界区
- 对于单处理器环境，临界区问题可简单地加以解决：在修改共享变量时禁止中断出现
- 但对于多处理器，中断禁止过于耗时，需要将消息传递到所有处理器
- 硬件同步的三种方法
  - 内存屏障（Memory Barriers）
  - 多处理器提供**特殊硬件指令，**原子地交换两个字（不可中断）
    - TestAndSet()
    - CompareAndSwap()

内存屏障

- 内存屏障是一种机制，用于确保在多核处理器架构下，不同线程或处理器之间的内存访问操作按照程序员的意图进行排序和同步
- 顺序保证
  - 强顺序保证：确保一个处理器对内存的修改会立刻被其他处理器可见
  - 弱顺序保证：无法确保一个处理器对内存的修改会立刻被其他处理器可见
- 在多核处理器中，各个核心可能有自己的本地缓存，内存屏障可以用于强制刷新或更新这些缓存，以确保所有核心都能看到最新的共享数据

内存屏障

- 当增加一个内存屏障以保证线程1输出100:
- 线程1执行:

```
while (!flag)
        memory_barrier();
print x
```

- 线程2执行:

```
x = 100;
memory_barrier();
flag = true
```

硬件指令

- TestAndSet()
  - 定义

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

在执行3条指令中间不允许产生中断

  - 从内存读值
  - 测试该值是否为1（然后返回真假）
  - 内存值设置为1

硬件指令

- 多个进程（可以大于2个）同时进入循环，希望获得锁
  - TestAndSet互斥实现：设置共享布尔变量lock，初始化为false

```
do {
        while ( TestAndSet ( &lock ))
                ;   // do nothing

                //    critical section

        lock = FALSE;

                //    remainder section

} while (TRUE);
```

硬件指令

- CompareAndSwap()

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

- 当value==expected时，将value置为new_value，同时返回原始value

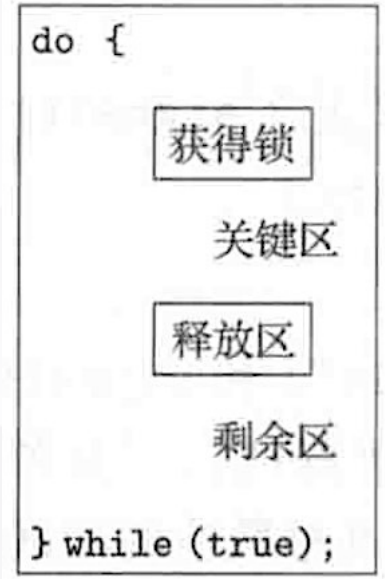
硬件指令

- CompareAndSwap()
  - 将共享变量lock初始化为0

```
while (true){
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

互斥锁

- 采用硬件同步较复杂，不能让程序员直接使用
- 软件工具---互斥锁（mutex lock）
  - acquire()获得锁
  - release()释放锁



互斥锁

- 互斥锁（mutex lock）有一个布尔变量available，表示锁是否可用，若该锁被占用，则进程被阻塞
- available初始化为true

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

```
release() {
    available = true;
}
```

- 当一个进程在临界区，其他进程在进入临界区时需要连续循环调用acquire，也称为“自旋锁”，浪费CPU周期（忙等待）

信号量

- 互斥锁，最简单
- 希望设计一种更鲁棒、提供更高级功能的方法：信号量 S，整型变量
- 两个标准原子操作，用于修改S
  - wait()，最初被称为P()，把信号量减1
  - signal()，最初被称为V()，把信号量加1
- 信号量的修改应不可分割地执行

```
wait (S) {
    while S <= 0
        ; // no-op
    S--;
}
```

```
signal (S) {
    S++;
}
```

信号量

- 信号量
  - 二进制信号量：0、1，类似于互斥锁
  - 计数信号量：不受限制，用于控制访问具有多个实例的资源，初值为资源数量
    - 使用资源，对信号量执行wait()，减少信号量计数
    - 释放资源，对信号量执行signal()，增加信号量计数
    - 当信号量等于0，表示所有资源都在使用中，之后的进程进入阻塞

信号量

- 核心：当执行操作wait()并发现信号量不为正时，不是忙等待，而是阻塞自己
- 定义进程链表

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

- wait()

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                sleep();
        }
}
```

- signal()

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0 && S->list != NULL) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

信号量

- 当一个进程必须等待信号量时，被添加到进程链表，被阻塞
- signal()从等待进程链表取走一个进程，并加以唤醒
- 引入两个新操作
  - sleep()挂起/阻塞调用它的进程
  - wakeup()重新启动阻塞进程的执行
- 信号量可以为负数，为负数时，它的绝对值是等待它的进程数

信号量的使用 实现互斥

- 临界区

wait(mutex);　　P()操作

…

临界区

…

signal(mutex);　V()操作

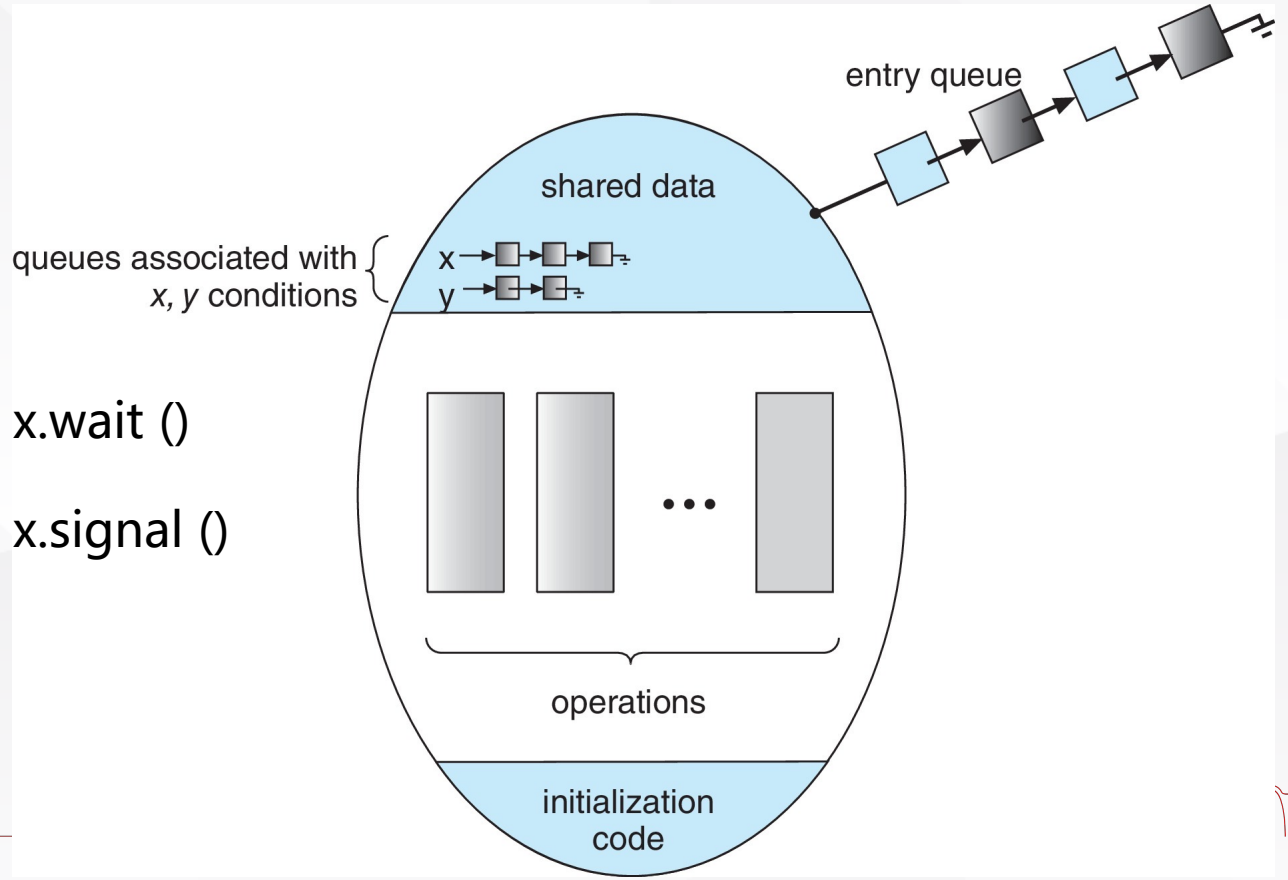
管程

- 管程是一种用于处理进程同步问题的高级抽象概念
- 高级语言工具，封装了数据及对其操作的一组函数
  - 管程提供了一组由程序员定义的、在管程内**互斥的操作(lock)**
  - 管程类型也包含**一组变量(variables)**，用于定义这一类型的实例状态，也包括操作这些变量的函数实现
- 只有管程内定义的函数才能访问管程内局部声明的变量和形式参数

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }

   procedure Pn (…) {……}

   Initialization code (…) { … }
}
```

管程

- 管程结构确保每次只有一个进程在管程内处于活动状态，程序员不需要明确编写同步约束
- 需要用shared data的进程都要排队（entry queue）



管程

- 确保互斥操作
  - lock->Acquire() --- 等待直到锁可用，然后抢占锁
  - lock->Release() --- 释放锁，唤醒等待者，如果有
- 针对条件变量，可以进行两种操作
  - x.wait () 调用这一操作的进程会被挂起，直到另一进程调用x.signal ()，可以看作该进程希望使用资源x，被为之等待
  - x.signal () 重新恢复一个挂起的进程。如果没有挂起进程，那么x.signal () 就没有作用，即x的状态如同没有执行任何操作，这与信号量的操作signal()不同，**信号量永远会被x.signal更新**

条件变量的实现

- 变量

```
int numWaiting=0;
waitQueue q;
```

```
Wait():
{
    numWaiting++;
    Add this thread t to q;
    lock->Release();
    Schedule();
    lock->Acquire();
}
```

为什么release和acquire是反的?

```
Signal():
{
    if(numWaiting>0){
        Remove a thread t from q;
        Wakeup(t);
        numWaiting--;
    }
}
```

使用管程解决生产者-消费者问题

- 变量

```
Lock lock;
int Count=0;
Condition notFull, notEmpty;
```

```
Producer(c):
{
    lock->Acquire();
    while(Count==n)
        notFull.Wait(&lock);
    Add c to the buffer;
    Count++;
    notEmpty.signal();
    lock->Release();
}
```

因为要先释放lock才能睡眠，否则所有进程都无法获得锁

```
Consumer(c):
{
    lock->Acquire();
    while(Count==0)
        notEmpty.Wait(&lock);
    Remove c from the buffer;
    Count--;
    notFull.signal();
    lock->Release();
}
```

注意：lock->Acquire()和lock->Release()是放在头和尾，而不是buffer的前后两句，这是由管程的特点决定的（“管程结构确保每次只有一个进程在管程内处于活动状态”）

具有同步机制的管程

- 假设当操作x.signal () 被进程P调用时，在变量x上有一个挂起进程Q
  - 此时P和Q都可以执行，其中Q被唤醒，那么到底执行哪个线程?
- 两种可能性
  - 唤醒并等待：唤醒Q，让Q先执行，进程P等待直到Q离开管程
    - 更直观，但实现更困难
  - 唤醒并继续：P先执行，再唤醒Q
    - 实现更简单

使用管程解决生产者-消费者问题

Producer(c):唤醒并等待

```
{
    lock->Acquire();
    while(count==n)
        notFull.Wait(&lock);
    Add c to the buffer;
    Count++;
    notEmpty.signal();
    lock->Release();
}
```

Producer(c):唤醒并继续

```
{
    lock->Acquire();
    if(count==n)
        notFull.Wait(&lock);
    Add c to the buffer;
    Count++;
    notEmpty.signal();
    lock->Release();
}
```

Consumer(c):

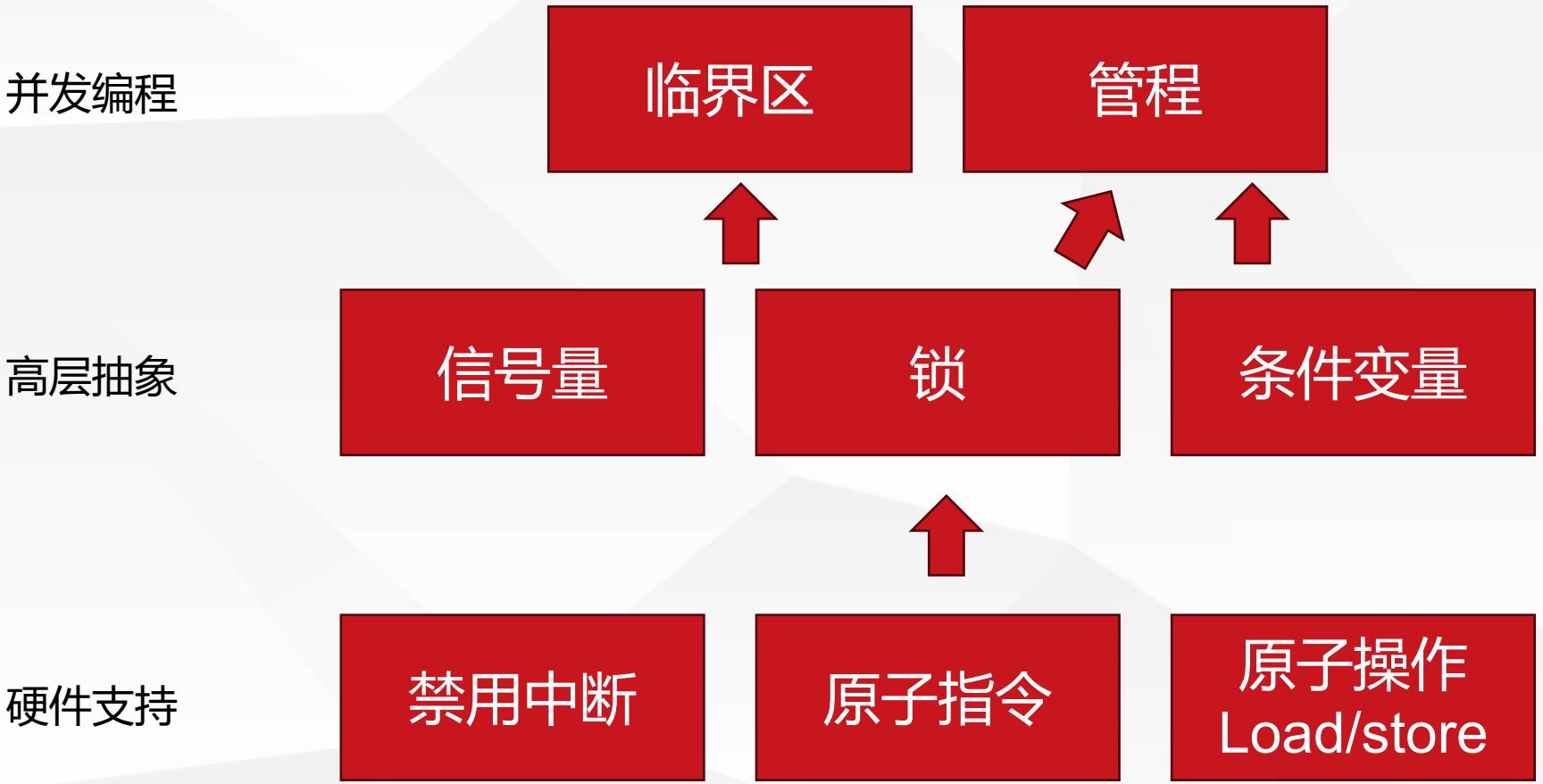
```
{
    lock->Acquire();
    while(Count==0)
        notEmpty.Wait(&lock);
    Remove c from the buffer;
    Count--;
    notFull.signal();
    lock->Release();
}
```

- 唤醒并等待：可能此时有多个进程试图被唤醒，因此需要用while确认count是否等于n
- 唤醒并继续：此时只有一个进程P在运行，不需要用while

具有同步机制的管程

- 唤醒并等待：进程P等待直到Q离开管程，或者等待另一个条件
  - 行为描述： 在调用 signal()后，Q需要重新竞争互斥锁
- 唤醒并继续：进程Q等待直到P离开管程，或者等待另一个条件
  - 行为描述： 在调用 signal() 后，Q不需要重新竞争互斥锁，它继续执行临界区的代码
- 两种各有优劣，由编程语言决定
  - 在某些情况下，唤醒并等待可能更安全，因为它确保被唤醒的进程在继续执行前重新检查条件；在其他情况下，唤醒并继续可能更高效，因为它避免了重新竞争互斥锁的开销

总结



课堂习题

假设两个进程P0和P1，共享变量：boolean flag[2]; int turn;

进程结构如图所示，请证明这个算法满足临界区问题的三个要求

```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```