

# 信号量、管程与屏障

---

中国科学院大学计算机学院  
2024-10-09





# 内容提要

---

- 锁的扩展
- 信号量
- 管程
- 屏障



# 常见的锁类型

- 自旋锁
  - 一个线程试图获取锁时，如果获取不到，则会不断“自旋”，即循环检查锁的状态，直到锁可用
  - 适用于锁定时间非常短的场景，开销小
- 互斥锁
  - 一个线程试图获取锁时，如果获取不到，则会进入等待状态
  - 当线程释放锁时，根据公平性策略决定哪个线程获得锁，例如按锁的请求顺序决定获得锁的线程



# 常见的锁类型

- 读写锁

- 允许多个读者同时访问共享资源，不允许读者和写者同时访问资源；写者在进行写操作时要独占资源
- 读操作
  - 如果没有写者持有锁，且没有任何写者正在等待获取锁（写者优先），那么读者可以获得读锁并访问共享资源
  - 如果没有写者持有锁，那么读者可以获得读锁并访问共享资源（读者优先）
- 写操作：如果没有写者持有锁，且所有持有的读锁都已释放，则可以获得写锁



# 常见的锁类型

- RCU锁

- 读写锁不足

- 读操作需要获取锁
    - 写操作在读操作进行时会被阻塞；读操作在写操作进行时也会被阻塞
    - 读多写少的情况下，读被写阻塞，影响系统效率

- Read-Copy-Update锁（RCU锁）

- 读操作不需要获取锁，可以并发执行
    - 写操作首先复制旧数据，在副本上进行更新
    - 更新完成后，使用RCU函数更新指针（`rcu_assign_pointer`）或替换节点（`list_replace_rcu`）
    - 确认所有当前的读操作已完成，再将旧数据清理（释放内存）

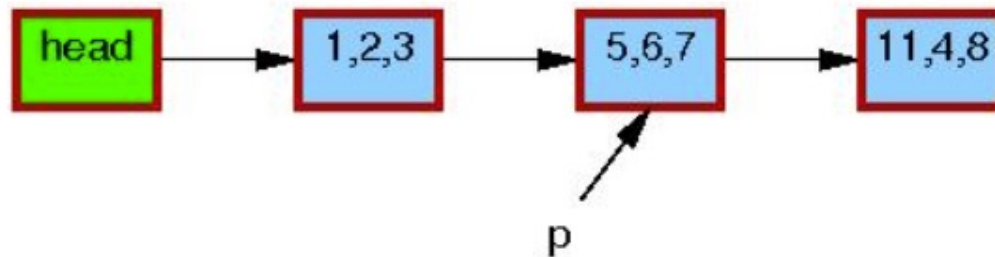
- 适用于高并发读（读多写少）场景



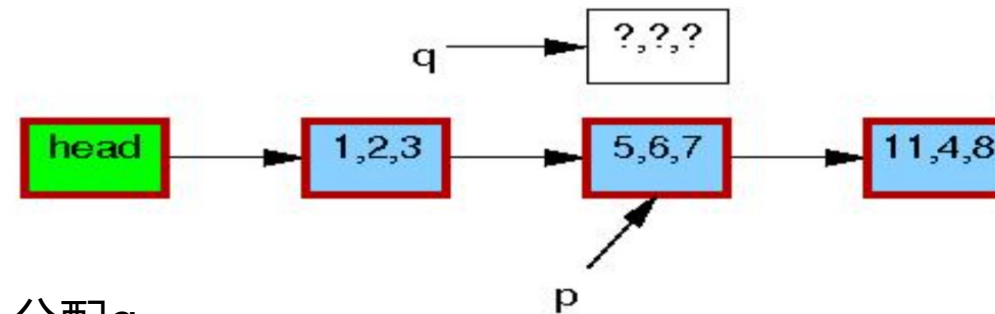
# 常见的锁类型

## • RCU锁示例

```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);  
2 *q = *p;  
3 q->b = 2;  
4 q->c = 3;  
5 list_replace_rcu(&p->list, &q->list);  
6 synchronize_rcu();  
7 kfree(p);
```



初始状态



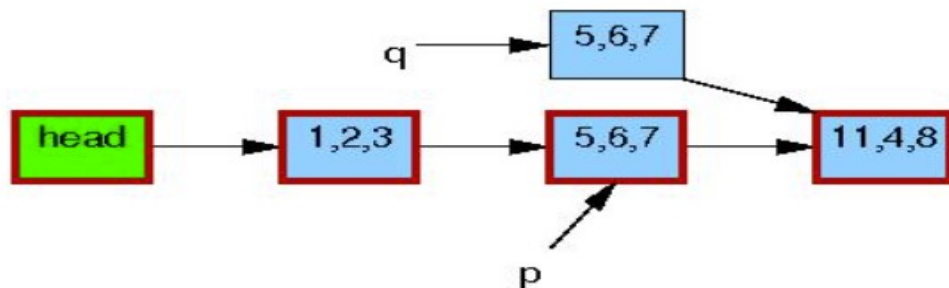
分配q



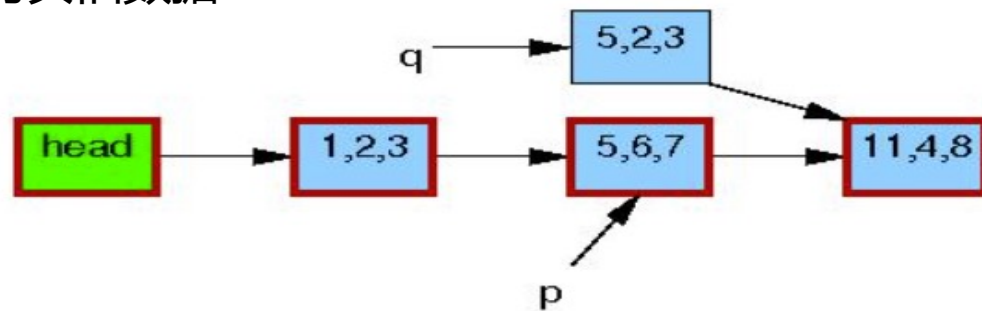
# 常见的锁类型

## • RCU锁示例

```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);  
2 *q = *p;  
3 q->b = 2;  
4 q->c = 3;  
5 list_replace_rcu(&p->list, &q->list);  
6 synchronize_rcu();  
7 kfree(p);
```



拷贝旧数据



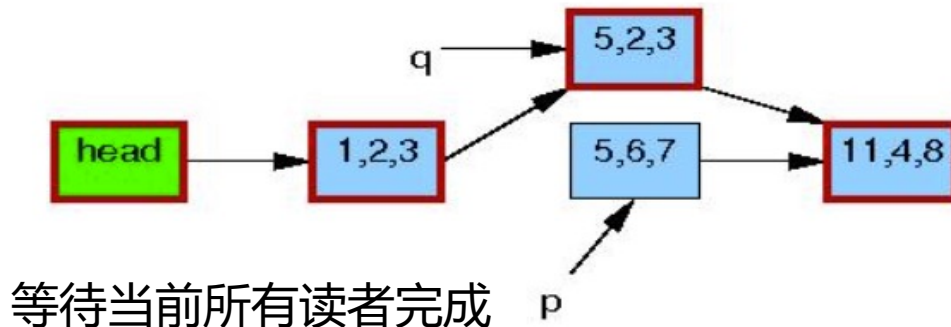
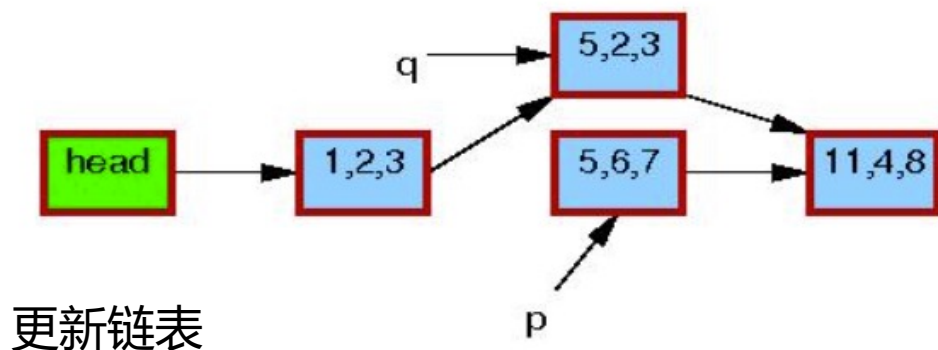
更新数值



# 常见的锁类型

## • RCU锁示例

```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);  
2 *q = *p;  
3 q->b = 2;  
4 q->c = 3;  
5 list_replace_rcu(&p->list, &q->list);  
6 synchronize_rcu();  
7 kfree(p);
```



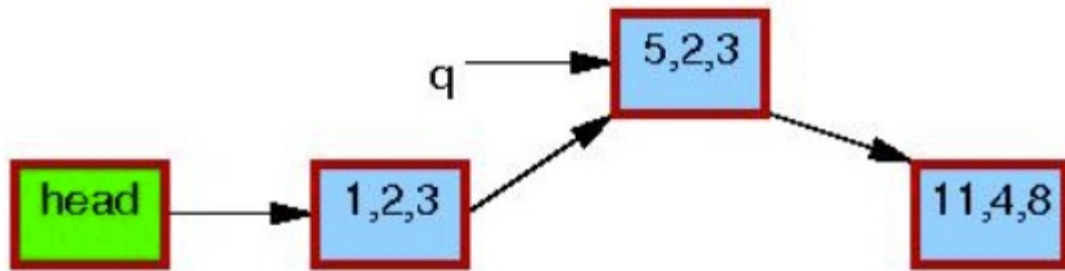




# 常见的锁类型

## • RCU锁示例

```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);  
2 *q = *p;  
3 q->b = 2;  
4 q->c = 3;  
5 list_replace_rcu(&p->list, &q->list);  
6 synchronize_rcu();  
7 kfree(p);
```



链表完成更新，旧数据被释放



# 内容提要

---

- 锁的扩展
- 信号量
- 管程
- 屏障



# 有锁就足够了么？

- 生产者-消费者问题



- 一个生产者在生成数据后，将数据写入一个缓冲区中
  - 一个消费者从缓冲区读出数据进行处理
  - 任何时刻只能有一个生产者或消费者可访问缓冲区
  - 应用场景：驱动程序从设备获取数据后写入缓冲区，内核模块从缓冲区读取数据
- 锁方案
  - 临界区：读写缓冲区
  - 保证只有一个线程访问缓冲区
  - 问题：生产者线程释放锁后，可能仍然是生产者线程获得锁



# 有锁就足够了么？

- 生产者-消费者问题



- 锁方案

- 锁保护共享资源互斥访问
- 无法提供线程按某些条件进行同步

- 所需的同步机制的特征

- 表示资源状态: 缓冲区空 vs. 缓冲区满
- 条件同步：使得多线程根据资源状态执行
  - 缓冲区空时，生产者可以写入数据
  - 缓冲区满时，消费者可以读取数据



# 信号量 Semaphores (Dijkstra, 1965)

- 信号量是操作系统提供的一种协调共享资源访问的方法
- 信号量组成
  - 一个整形变量，表示系统资源的数量
  - 两个原子操作：P操作（Wait操作）和V操作（Signal操作）

## P 操作(又名 Down 或 Wait)

- 等待信号量为正，信号量减1

```
P(s) {  
    while (s <= 0);  
    s--;  
}
```

## V 操作(又名 Up 或 Signal)

- 信号量加1

```
V(s) {  
    s++;  
}
```



# 信号量的实现与使用

```
Class Semaphore {  
    int sem;  
    WaitQueue q;  
}
```

```
Semaphore::P() {  
    sem--;  
    if (sem < 0) {  
        Add this thread t to q;  
        block(t);  
    }  
}
```

```
Semaphore::V() {  
    sem++;  
    if (sem <= 0) {  
        Remove a thread t from q;  
        wakeup(t);  
    }  
}
```

- 信号量的使用

- 互斥访问：保护临界区互斥访问，Semaphore(1)
- 条件同步：多线程之间同步，Semaphore (N>=0)



# 用信号量实现生产者-消费者问题



- 有界缓冲区的生产者-消费者问题
  - ▶ 一个或多个**生产者**在生成数据后放在一个缓冲区里
  - ▶ 单个**消费者**从缓冲区取出数据处理
  - ▶ 任何时刻**只能有一个**生产者或消费者可访问缓冲区
- 要求
  - ▶ 任何时刻只能有一个线程操作缓冲区 (**互斥访问**)
  - ▶ 缓冲区空时, 消费者必须等待生产者 (**条件同步**)
  - ▶ 缓冲区满时, 生产者必须等待消费者 (**条件同步**)



# 用信号量实现生产者-消费者问题

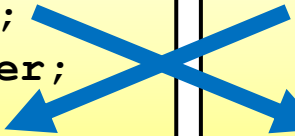
- 设计

- 缓冲区空：信号量emptyBuffers
- 缓冲区满：信号量fullBuffers

```
Class BoundedBuffer {  
    fullBuffers = new Semaphore(0);  
    emptyBuffers = new Semaphore(1);  
}
```

```
BoundedBuffer::Deposit(c) {  
    emptyBuffers->P();  
    Add c to the buffer;  
    fullBuffers->V();  
}
```

```
BoundedBuffer::Remove(c) {  
    fullBuffers->P();  
    Remove c from buffer;  
    emptyBuffers->V();  
}
```







# 用信号量实现生产者-消费者问题

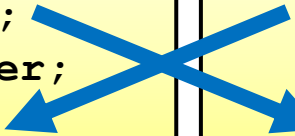
- 设计

- 缓冲区空：信号量emptyBuffers
- 缓冲区满：信号量fullBuffers

```
Class BoundedBuffer {  
    fullBuffers = new Semaphore(0);  
    emptyBuffers = new Semaphore(n);  
}
```

```
BoundedBuffer::Deposit(c) {  
    emptyBuffers->P();  
    Add c to the buffer;  
    fullBuffers->V();  
}
```

```
BoundedBuffer::Remove(c) {  
    fullBuffers->P();  
    Remove c from buffer;  
    emptyBuffers->V();  
}
```





# 用信号量实现生产者-消费者问题

- 设计

- 缓冲区空：信号量emptyBuffers
- 缓冲区满：信号量fullBuffers

```
Class BoundedBuffer {  
    mutex = new Semaphore(1);  
    fullBuffers = new Semaphore(0);  
    emptyBuffers = new Semaphore(n);  
}
```

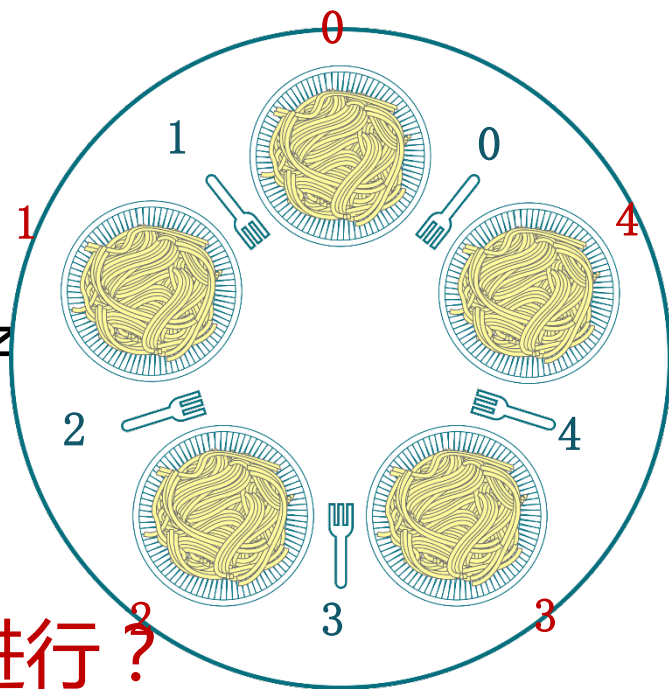
```
BoundedBuffer::Deposit(c) {  
    emptyBuffers->P();  
    mutex->P();  
    Add c to the buffer;  
    mutex->V();  
    fullBuffers->V();  
}
```

```
BoundedBuffer::Remove(c) {  
    fullBuffers->P();  
    mutex->P();  
    Remove c from buffer;  
    mutex->V();  
    emptyBuffers->V();  
}
```



# 示例1：哲学家就餐问题

- 5个哲学家围绕一张圆桌而坐
  - 桌子上放着5支叉子
  - 每两个哲学家之间放一支叉子
- 哲学家的动作包括思考和进餐
  - 进餐时需同时拿到左右两边的叉子
  - 思考时将两支叉子放回原处



- 如何保证哲学家们的动作有序进行？
- 不出现有人永远拿不到叉子



# 方案1

- 二值信号量（锁）

```
#define N 5                                     // 哲学家个数
semaphore fork[5];                             // 信号量初值为1
void philosopher(int i)                        // 哲学家编号：0 - 4
{
    while(TRUE)
    {
        think( );                             // 哲学家在思考
        P(fork[i]);                           // 去拿左边的叉子
        P(fork[(i + 1) % N]);                 // 去拿右边的叉子
        eat( );                               // 吃面条中...
        V(fork[i]);                           // 放下左边的叉子
        V(fork[(i + 1) % N]);                 // 放下右边的叉子
    }
}
```

**不正确，可能导致死锁**



# 方案2

- 一把大锁。。。

```
#define    N    5                                // 哲学家个数
semaphore fork[5];                               // 信号量初值为1
semaphore  mutex;                                // 互斥信号量, 初值1
void  philosopher(int    i)                      // 哲学家编号: 0 - 4
{
    while(TRUE) {
        think( );                                // 哲学家在思考
        P(mutex);                                // 进入临界区
        P(fork[i]);                               // 去拿左边的叉子
        P(fork[(i + 1) % N]);                    // 去拿右边的叉子
        eat( );                                   // 吃面条中....
        V(fork[i]);                               // 放下左边的叉子
        V(fork[(i + 1) % N]);                    // 放下右边的叉子
        V(mutex);                                // 退出临界区
    }
}
```

互斥访问正确, 但每次只允许一人进餐



# 方案3

- 调整执行顺序

```
#define N 5 // 哲学家个数
semaphore fork[5]; // 信号量初值为1
void philosopher(int i) // 哲学家编号: 0 - 4
{
    while(TRUE)
    {
        think( ); // 哲学家在思考
        if (i%2 == 0) {
            P(fork[i]); // 去拿左边的叉子
            P(fork[(i + 1) % N]); // 去拿右边的叉子
        } else {
            P(fork[(i + 1) % N]); // 去拿右边的叉子
            P(fork[i]); // 去拿左边的叉子
        }
        eat( ); // 吃面条中....
        V(fork[i]); // 放下左边的叉子
        V(fork[(i + 1) % N]); // 放下右边的叉子
    }
}
```

没有死锁，可有多人同时就餐



## 示例2：读者-写者问题

- 共享数据的两类使用者
  - 读者：只读取数据，不修改
  - 写者：修改数据
- 读者-写者问题描述：对共享数据的读写
  - “读 - 读”允许
    - 同一时刻，允许有多个读者同时读
  - “读 - 写” 互斥
    - 没有写者时读者才能读
    - 没有读者时写者才能写
  - “写 - 写” 互斥
    - 没有其他写者时写者才能写



# 用信号量解决读者-写者问题

- 用信号量描述每个约束
- 信号量WriteMutex
  - 控制读写操作的互斥
  - 初始化为1
- 读者计数Rcount
  - 正在进行读操作的读者数目
  - 初始化为0
- 信号量CountMutex
  - 控制对读者计数的互斥修改
  - 初始化为1





# 用信号量解决读者-写者问题

- 写互斥
- 读写互斥

Writer

```
P (WriteMutex) ;  
  
write;  
  
V (WriteMutex) ;
```

此实现中，读者优先

Reader

```
P (CountMutex) ;  
if (Rcount == 0)  
    P (WriteMutex) ;  
++Rcount;  
V (CountMutex) ;  
  
read;  
  
P (CountMutex) ;  
--Rcount;  
if (Rcount == 0)  
    V (WriteMutex) ;  
V (CountMutex)
```



# 内容提要

---

- 锁的扩展
- 信号量
- 管程
- 屏障



# 为什么需要管程(Monitors)

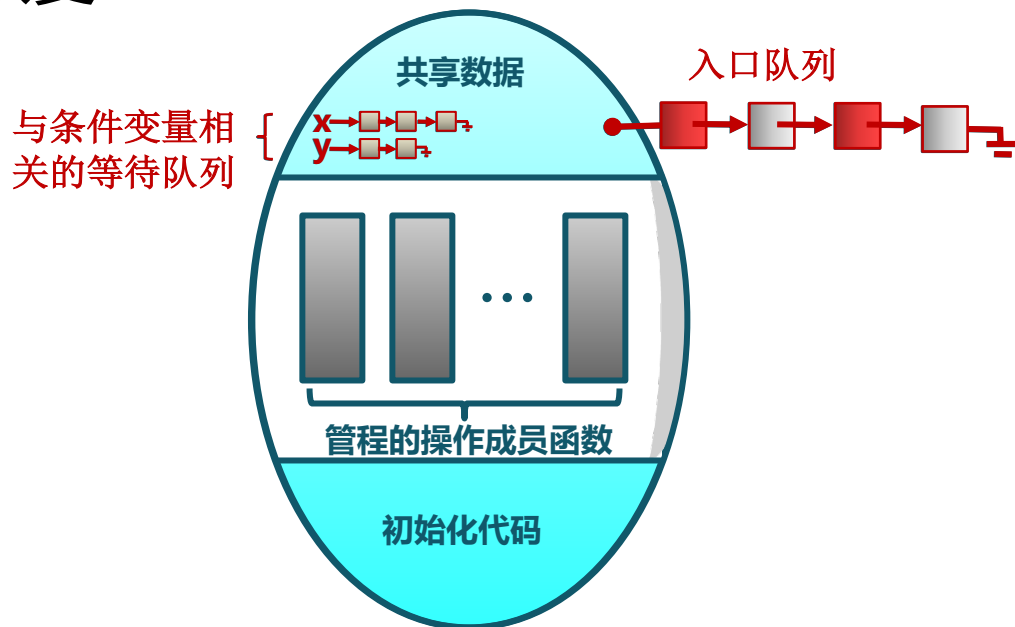
- 信号量定义
  - 资源数目和队列
  - P/V操作
- 潜在问题
  - 调用者手动调用P/V操作做条件同步
  - 可能因调用错误，导致程序出错
- 所需机制的特征
  - 避免调用者错误，将共享变量及对共享变量的操作进行封装
  - 形成一个具有一定接口的功能模块，供线程调用实现并发控制

```
Class Semaphore {  
    int sem;  
    WaitQueue q;  
}
```



# 管程(Monitors)

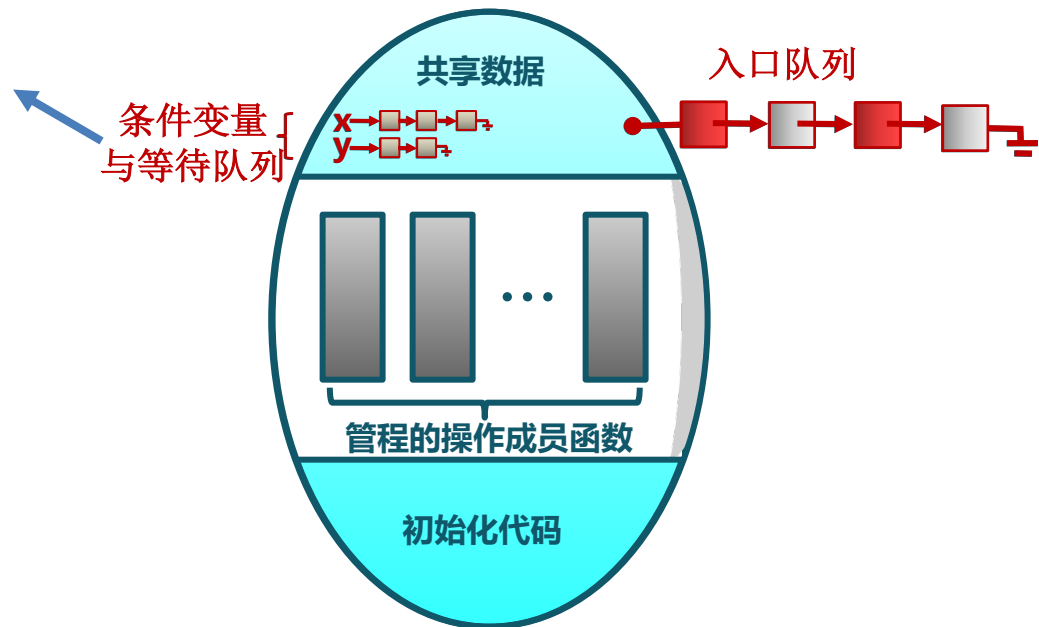
- **管程**是一种用于多线程互斥访问共享资源的程序结构
  - 采用面向对象方法，提供了一个封装共享资源及其操作的机制，使得共享资源的访问模块化、易维护
  - 任一时刻最多只有一个线程执行管程代码
  - 正在管程中的线程可临时放弃管程的互斥访问，等待事件出现时恢复





# 管程组成

- 组成
  - 一个锁
    - 控制管程代码的互斥访问
  - 条件变量(Condition Variables)
    - 管理共享数据的条件同步
  - 等待原因
    - 是否需要等待的条件





# 管程示例

可能需要等待的线程

```
Acquire(mutex);  
while (等待条件满足)  
    cond.wait(mutex);  
...  
(使用资源)  
...  
Release(mutex);
```

改变等待条件的线程

```
Acquire(mutex);  
...  
(使资源可用, 使等待条件不满足)  
...  
Signal(cond);  
/* 或 Broadcast(cond); */  
Release(mutex);
```



# 管程组成

- 面向对象封装

- 成员变量

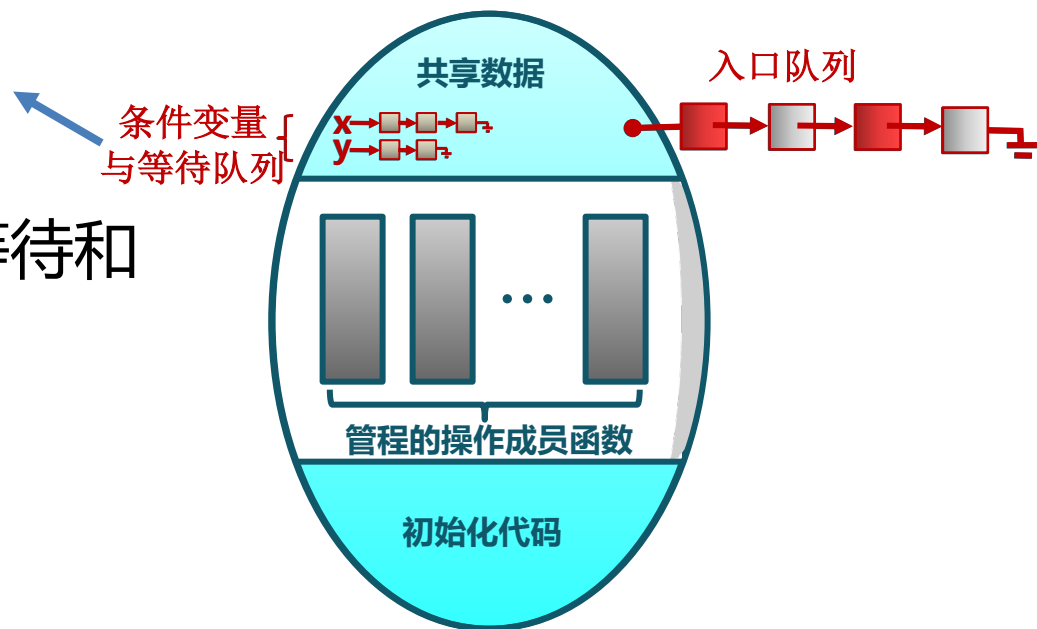
- 等待原因（例如，某个变量值）
    - 条件变量

- 成员函数

- 资源操作代码

- 条件变量

- 提供原语操作，实现等待和通知
  - 包含等待队列





# 管程示例

- 面向对象封装
  - ExampleMonitor.op1()
  - ExampleMonitor.op2()

```
Monitor ExampleMonitor
  condition cv1;
  bool flag;

  procedure op1()
  begin
    acquire(mutex)
    while(flag)
      cv1.wait(mutex)
    use resource
    release(mutex)
  end
```

```
  procedure op2()
  begin
    acquire(mutex)
    use resource
    flag = false
    cv1.signal()
    release(mutex)
  end
```





# 条件变量原语操作

- 条件变量是管程内的等待机制
  - 每个条件变量表示一种等待原因，对应一个等待队列
- Wait()原语操作
  - 将自己阻塞在等待队列中
  - 等待被唤醒或执行线程释放管程的互斥访问
- Signal()原语操作
  - 将等待队列中的一个线程唤醒
  - 如果等待队列为空，则等同空操作
- Broadcast()原语操作
  - 唤醒所有等待的线程



# 条件变量实现

- 实现示例

```
Class Condition {  
    int numWaiting = 0;  
    WaitQueue q;  
}
```

```
Condition::Wait(lock) {  
    numWaiting++;  
    Add this thread t to q;  
    release(lock);  
    schedule();  
    acquire(lock);  
}
```

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



# 用管程实现生产者-消费者问题

- 示例

```
Class BoundedBuffer {  
    ...  
    Lock lock;  
    int count = 0;  
    Condition full, empty;  
}
```

```
BoundedBuffer::Deposit(c) {
```

```
    Add c to the buffer;  
    count++;
```

```
}
```

```
BoundedBuffer::Remove(c) {
```

```
    Remove c from buffer;  
    count--;
```

```
}
```



# 用管程实现生产者-消费者问题

- 示例

```
classBoundedBuffer {  
    ...  
    Lock lock;  
    int count = 0;  
    Condition full, empty;  
}
```

```
BoundedBuffer::Deposit(c) {  
    lock->Acquire();  
  
    Add c to the buffer;  
    count++;  
  
    lock->Release();  
}
```

```
BoundedBuffer::Remove(c) {  
    lock->Acquire();  
  
    Remove c from buffer;  
    count--;  
  
    lock->Release();  
}
```



# 用管程实现生产者-消费者问题

```
classBoundedBuffer {  
    ...  
    Lock lock;  
    int count = 0;  
    Condition full, empty;  
}
```

```
BoundedBuffer::Deposit(c) {  
    lock->Acquire();  
    while (count == n)  
        full.Wait(&lock);  
    Add c to the buffer;  
    count++;  
  
    lock->Release();  
}
```

```
BoundedBuffer::Remove(c) {  
    lock->Acquire();  
  
    Remove c from buffer;  
    count--;  
    full.Signal();  
    lock->Release();  
}
```

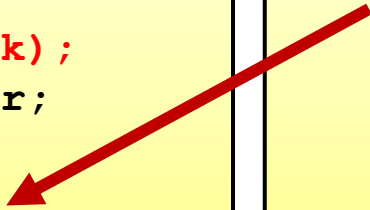


# 用管程实现生产者-消费者问题

```
Class BoundedBuffer {  
    ...  
    Lock lock;  
    int count = 0;  
    Condition full, empty;  
}
```

```
BoundedBuffer::Deposit(c) {  
    lock->Acquire();  
    while (count == n)  
        full.Wait(&lock);  
    Add c to the buffer;  
    count++;  
    empty.Signal();  
    lock->Release();  
}
```

```
BoundedBuffer::Remove(c) {  
    lock->Acquire();  
    while (count == 0)  
        empty.Wait(&lock);  
    Remove c from buffer;  
    count--;  
    full.Signal();  
    lock->Release();  
}
```





# 用管程实现生产者-消费者问题

- 另一种实现
  - 问题：signal后被唤醒线程是否立即运行？

```
Class BoundedBuffer {  
    ...  
    Lock lock;  
    int count = 0;  
    Condition full, empty;  
}
```

```
BoundedBuffer::Deposit(c) {  
    lock->Acquire();  
    if (count == n)  
        full.Wait(&lock);  
    Add c to the buffer;  
    count++;  
    empty.Signal();  
    lock->Release();  
}
```

```
BoundedBuffer::Remove(c) {  
    lock->Acquire();  
    if (count == 0)  
        empty.Wait(&lock);  
    Remove c from buffer;  
    count--;  
    full.Signal();  
    lock->Release();  
}
```



# Signal 之后的三种选择

- 关键：要保证管程代码只有一个线程在执行
- 发送方退出管程 (**Hansen管程**)
  - 规定Signal 必须是管程中的过程的最后一个语句
  - 例如，signal函数后就执行exit退出
- 让被唤醒的线程立即执行，发送方进入signal队列 (**Hoare管程**)
  - 被唤醒的线程执行完后再唤醒发送方
  - 如果发送方发送signal后，仍然有其他工作要做，会因进入signal队列，而无法立即执行
  - 一般实现时，发送方应在发送signal前把工作做完
- 发送方继续执行 (**Mesa管程**)
  - 被唤醒的线程进入管程入口队列中排队
  - 被唤醒的线程实际回到wait()执行时，条件可能不为真，需要重新判断等待原因条件
  - 易于实现





# Mesa管程解决生产者-消费者问题

```
static count = 0;  
static Cond full, empty;  
static Mutex lock;
```

```
Enter(Item item) {  
    Acquire(lock);  
    if (count==1)  
        full.wait(lock);  
    Add c to the buffer  
    count++;  
    Signal(empty);  
    Release(lock);  
}
```

```
Remove(Item item) {  
    Acquire(lock);  
    if (!count)  
        empty.wait(lock);  
    Remove c from buffer  
    count--;  
    Signal(full);  
    Release(lock);  
}
```

不规范的编程模式



# Mesa管程解决生产者-消费者问题

```
static count = 0;  
static Cond full, empty;  
static Mutex lock;
```

```
Enter(Item item) {  
    Acquire(lock);  
    while (count==1)  
        full.wait(lock);  
    Add c to the buffer  
    count++;  
    Signal(empty);  
    Release(lock);  
}
```

```
Remove(Item item) {  
    Acquire(lock);  
    while (!count)  
        empty.wait(lock);  
    Remove c from buffer  
    count--;  
    Signal(full);  
    Release(lock);  
}
```

规范的编程模式



# 编程语言对管程的支持

- Java内置管程
  - 参考MESA模型实现，支持一个条件变量
  - 使用synchronized关键字修饰代码块，编译时自动生成加、解锁的代码
  - 提供wait()，notify()，notifyAll()
- pthread库
  - 提供相关原语，用于实现管程
  - pthread\_mutex\_lock、pthread\_cond\_wait、pthread\_cond\_signal、pthread\_cond\_broadcast



# 信号量与管程对比

- 信号量

- 控制对多个共享资源的访问，用于进程/线程间条件同步
- 可以并发，取决于sem初始值
- sem表示资源数量
- P操作可能导致阻塞，也可能不阻塞
- V操作唤醒其他进程/线程后，当前进程/线程与被唤醒者可以并发执行

- 管程

- 一种程序结构、共享资源访问的封装方法
- 管程内部同一时刻只有一个线程执行
- 自行判断资源可用性（等待原因判断）
- wait操作一定会阻塞
- signal操作后，被唤醒线程是否执行取决于管程风格



# 用管程解决读者-写者问题

- 管程封装

- 两个基本方法

```
Database::Read() {  
    Wait until no writers;  
    read database;  
    check out - wake up waiting writers;  
}
```

```
Database::Write() {  
    Wait until no readers/writers;  
    write database;  
    check out - wake up waiting readers/writers;  
}
```

- 管程的状态变量

```
AR = 0;           // # of active readers  
AW = 0;           // # of active writers  
WR = 0;           // # of waiting readers  
WW = 0;           // # of waiting writers  
Lock lock;  
Condition okToRead;  
Condition okToWrite;
```



# 读者实现

- 对写友好

```
AR = 0;    // # of active readers
AW = 0;    // # of active writers
WR = 0;    // # of waiting readers
WW = 0;    // # of waiting writers
Lock lock;
Condition okToRead;
Condition okToWrite;
```

```
Public Database::Read() {
    //Wait until no writers;
    StartRead();
    read database;
    //check out - wake up waiting writers;
    DoneRead();
}
```

```
Private Database::StartRead() {
    lock.Acquire();
    while ((AW+WW) > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.Release();
}
```

```
Private Database::DoneRead() {
    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0) {
        okToWrite.signal();
    }
    lock.Release();
}
```



# 写者

```
AR = 0;    // # of active readers
AW = 0;    // # of active writers
WR = 0;    // # of waiting readers
WW = 0;    // # of waiting writers
Lock lock;
Condition okToRead;
Condition okToWrite;
```

```
Public Database::Write() {
    //Wait until no readers/writers;
    StartWrite();
    write database;
    //check out-wake up waiting readers/writers;
    DoneWrite();
}
```

```
Private Database::StartWrite() {
    lock.Acquire();
    while ((AW+AR) > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.Release();
}
```

```
Private Database::DoneWrite() {
    lock.Acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    }
    else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.Release();
}
```



# 内容提要

---

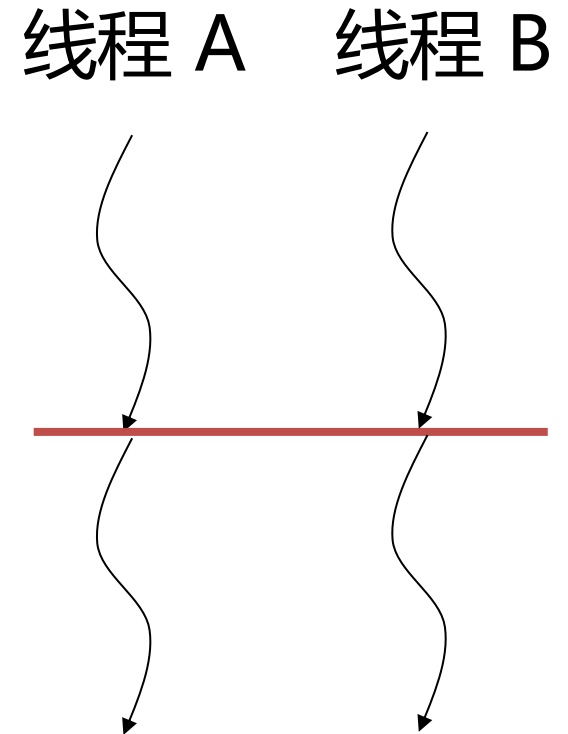
- 锁的扩展
- 信号量
- 管程
- 屏障





# 屏障 (Barrier)

- 线程 A 和线程 B 希望在某个特定的点交会并继续执行
  - 示例1
    - 父线程调用pthread\_join等待
    - 直到子线程调用pthread\_exit退出
  - 示例2
    - 多线程map-reduce操作

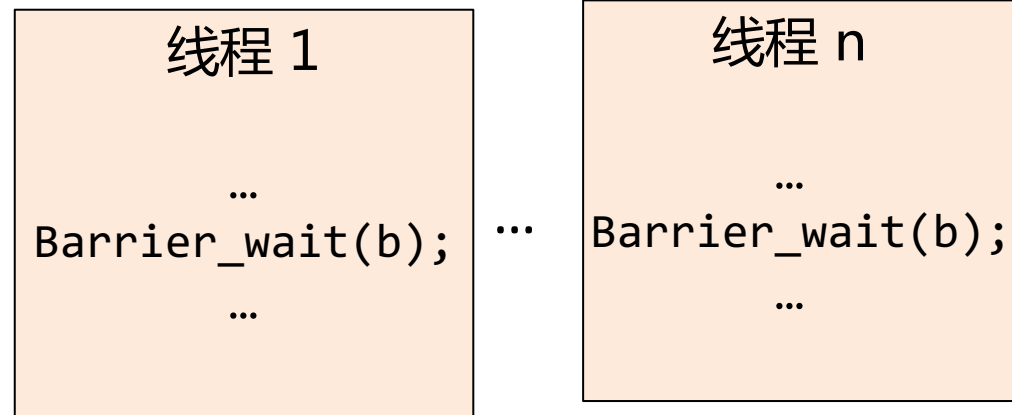




# 屏障原语 ( Barrier )

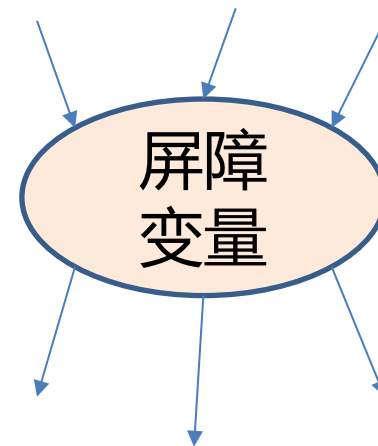
- 功能

- 协调多个线程并行共同完成某项任务
- 设定一个屏障变量b及其初始值n
- 若屏障变量值小于n，则线程等待
- 若屏障变量的值达到n，则唤醒所有线程，所有线程继续工作



- 操作原语

- barrier\_wait





# 屏障实现

---

- 使用管程实现
  - 等待原因：抵达屏障的线程数量没有达到 $n$
- 可以使用信号量来实现么？



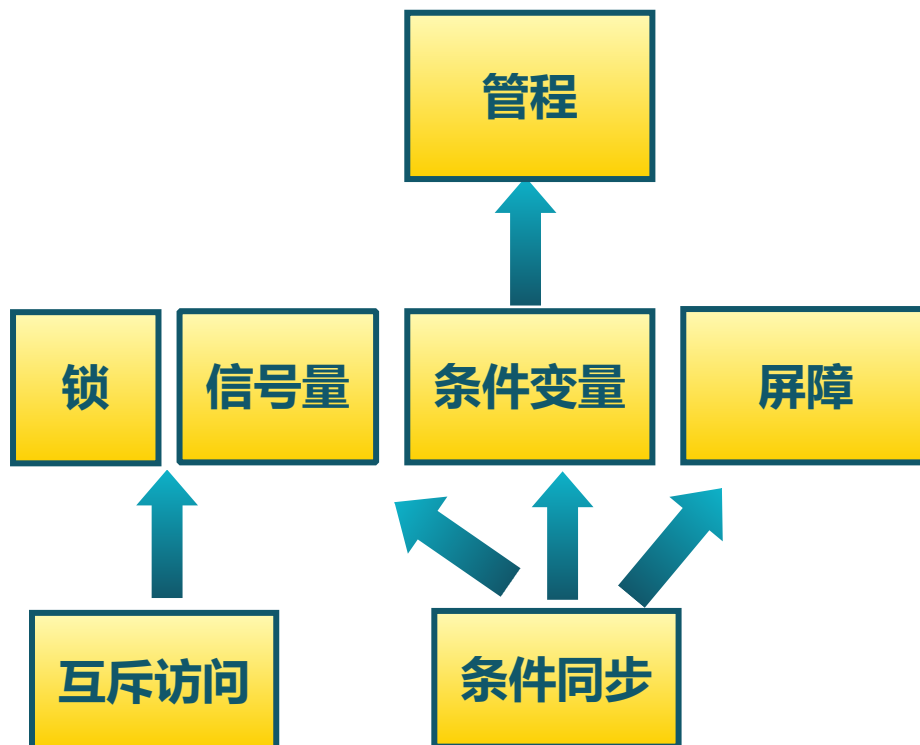
# 同步机制

- 同步机制总结

并发编程

高层抽象

应用需求





# 总结

- 锁的扩展：读写锁、RCU锁
- 锁方案的不足
- 线程间的条件同步
  - 信号量
    - 使用信号量表明可用资源和等待线程数量
    - 提供P/V操作
  - 管程
    - 实现对资源并发访问的封装
  - 屏障
    - 协调多线程在屏障点进行同步