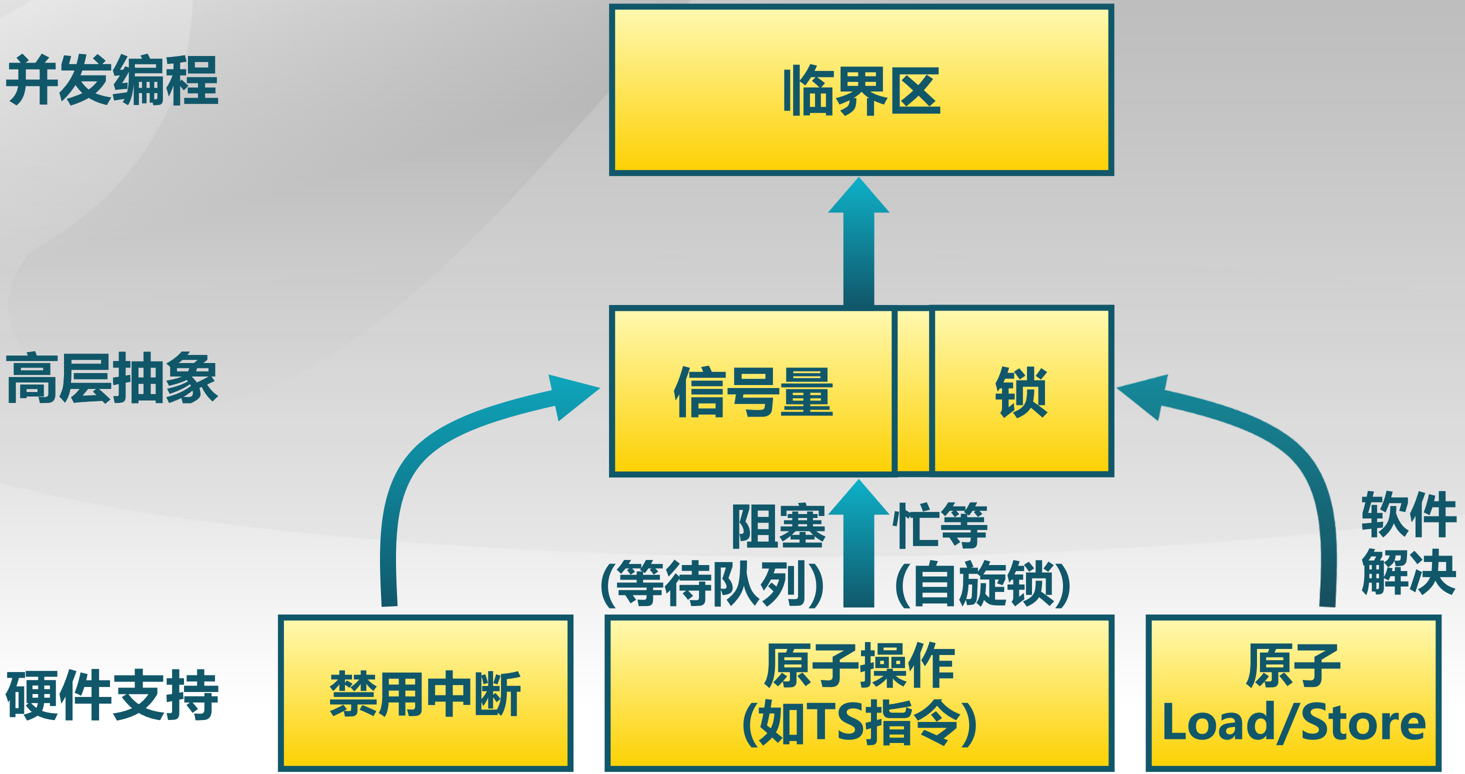
回顾

- 并发问题
  - 多线程并发导致资源竞争
- 同步概念
  - 协调多线程对共享数据的访问
  - 任何时刻只能有一个线程执行临界区代码
- 确保同步正确的方法
  - 底层硬件支持
  - 高层次的编程抽象

基本同步方法



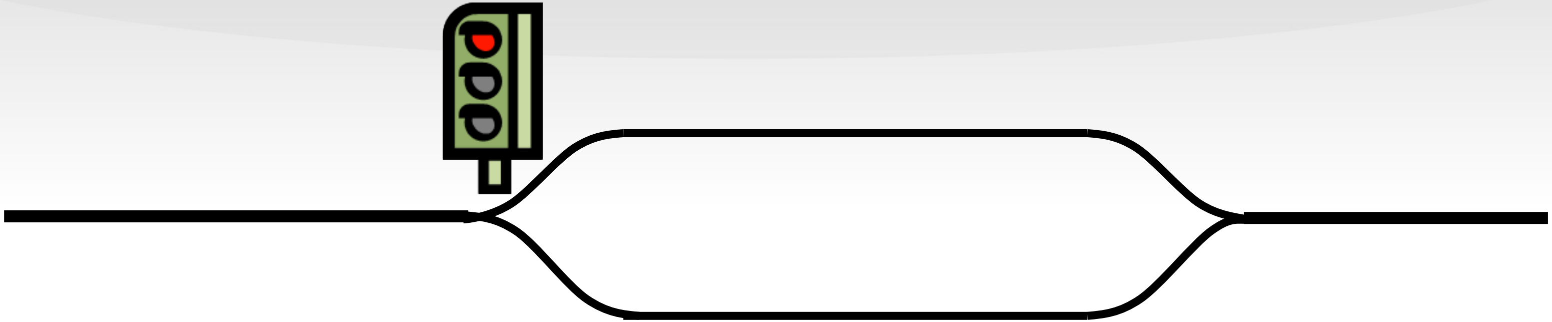
并发编程
高层抽象
硬件支持
临界区
信号量
锁
阻塞
(等待队列)
忙等
(自旋锁)
软件
解决
禁用中断
原子操作
(如TS指令)
原子
Load/Store

信号量(semaphore)

- 信号量是操作系统提供的一种协调共享资源访问的方法
  - 软件同步是平等线程间的一种同步协商机制
  - OS是管理者，地位高于进程
  - 用信号量表示系统资源的数量
- 由Dijkstra在20世纪60年代提出
- 早期的操作系统的主要同步机制
  - 现在很少用（但还是非常重要在计算机科学研究）

信号量(semaphore)

- 信号量是一种抽象数据类型
  - 由一个整形 (sem)变量和两个原子操作组成
  - P() (Prolaag （荷兰语尝试减少））
    - sem减1
    - 如sem<0, 进入等待, 否则继续
  - V() (Verhoog （荷兰语增加））
    - sem加1
    - 如sem≤0,唤醒一个等待进程
- 信号量与铁路的类比
  - 2个站台的车站
  - 2个资源的信号量



信号量的特性

- 信号量是**被保护**的**整数**变量
  - 初始化完成后，只能通过P()和V()操作修改
  - 由操作系统保证，PV操作是原子操作
- **P() 可能阻塞**，V()不会阻塞
- 通常假定信号量是“公平的”
  - 线程不会被无限期阻塞在P()操作
  - 假定信号量等待按先进先出排队

自旋锁能否实现先进先出?

信号量的实现

```
classSemaphore {
int sem;
WaitQueue q;
}
```

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

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

操作系统
Operating Systems

信号量分类

- 可分为两种信号量
  - 二进制信号量：资源数目为0或1
  - 资源信号量:资源数目为任何非负值
  - 两者等价
    - 基于一个可以实现另一个
- 信号量的使用
  - 互斥访问
    - 临界区的互斥访问控制
  - 条件同步
    - 线程间的事件等待

用信号量实现临界区的互斥访问

每个临界区设置一个信号量，其初值为1

```
mutex = new Semaphore(1);
```

```
mutex->P();
Critical Section;
mutex->V();
```

- 必须**成对使用**P()操作和V()操作
  - P()操作保证互斥访问临界资源
  - V()操作在使用后释放临界资源
  - PV操作**不能次序错误、重复或遗漏**

用信号量实现条件同步

每个条件同步设置一个信号量，其初值为0

```
condition = new Semaphore(0);
```

线程A

```
… M …
condition->P();
… N …
```

线程B

```
… X …
condition->V();
… Y …
```

生产者-消费者问题

- 有界缓冲区的生产者-消费者问题描述
  - 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

用信号量解决生产者-消费者问题

- 问题分析
  - 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）
- 用信号量描述每个约束
  - 二进制信号量mutex
  - 资源信号量fullBuffers
  - 资源信号量emptyBuffers

用信号量解决生产者-消费者问题

```
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();
}
```

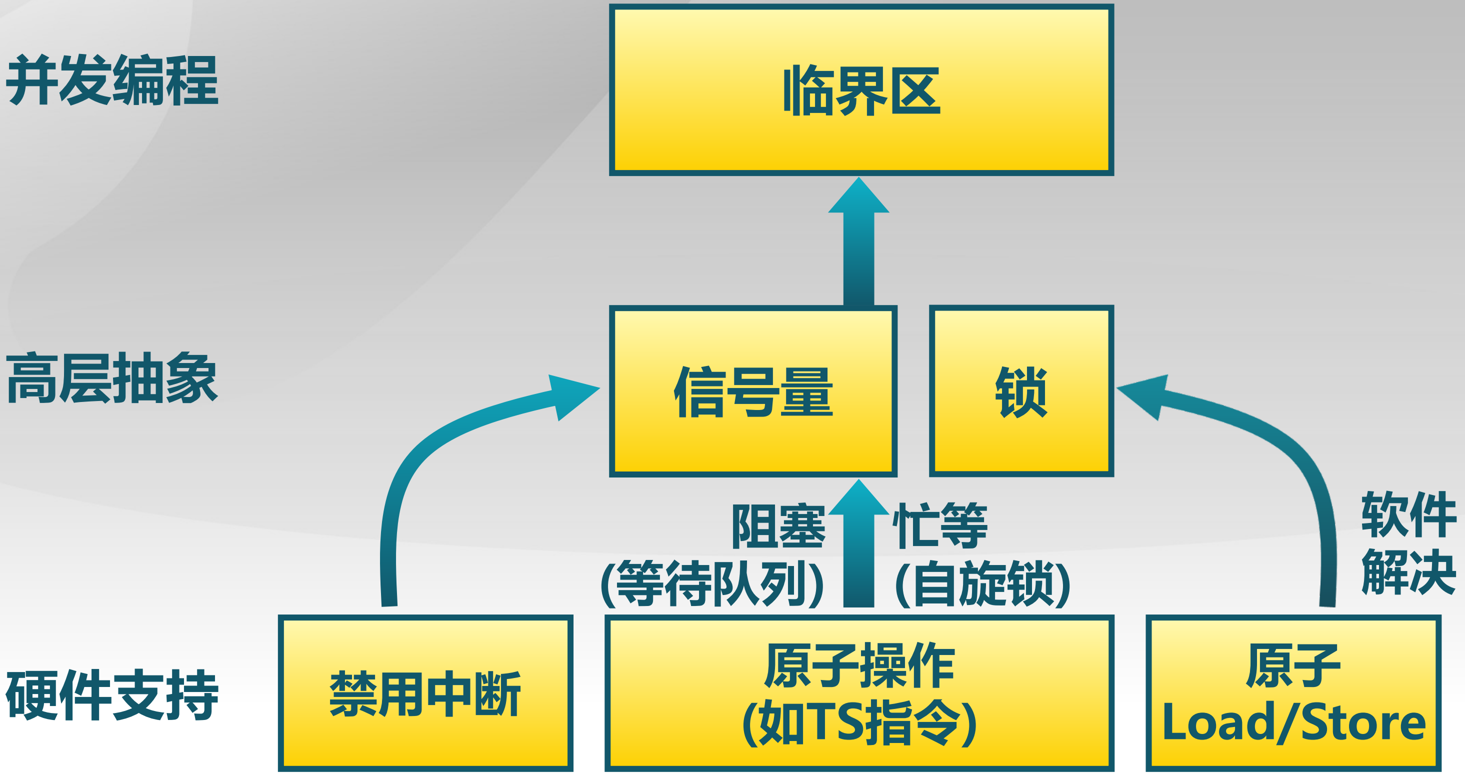
- P、V操作的顺序有影响吗?

使用信号量的困难

- 读/开发代码比较困难
  - 程序员需要能运用信号量机制
- 容易出错
  - 使用的信号量已经被另一个线程占用
  - 忘记释放信号量
- 不能够处理死锁问题

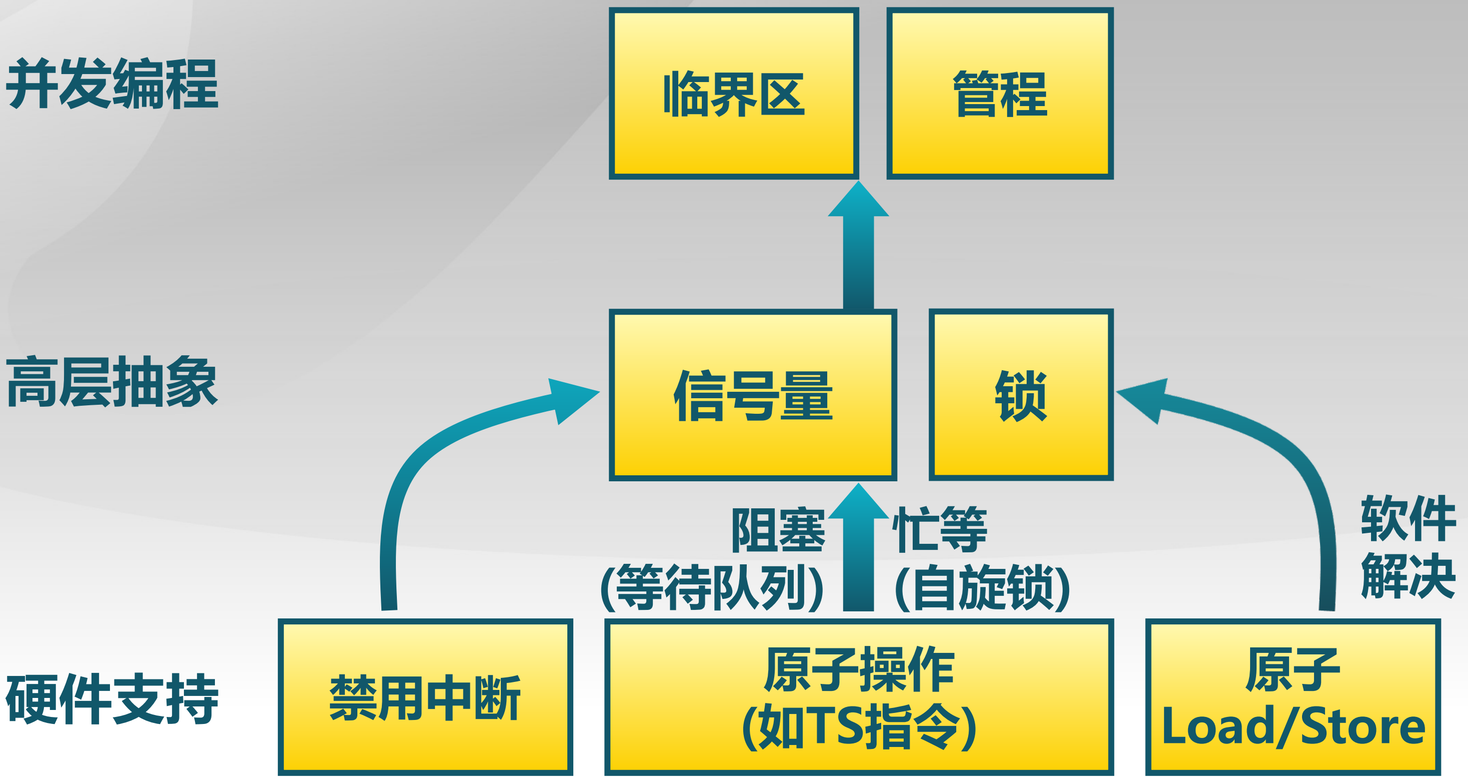
操作系统
Operating Systems

基本同步方法



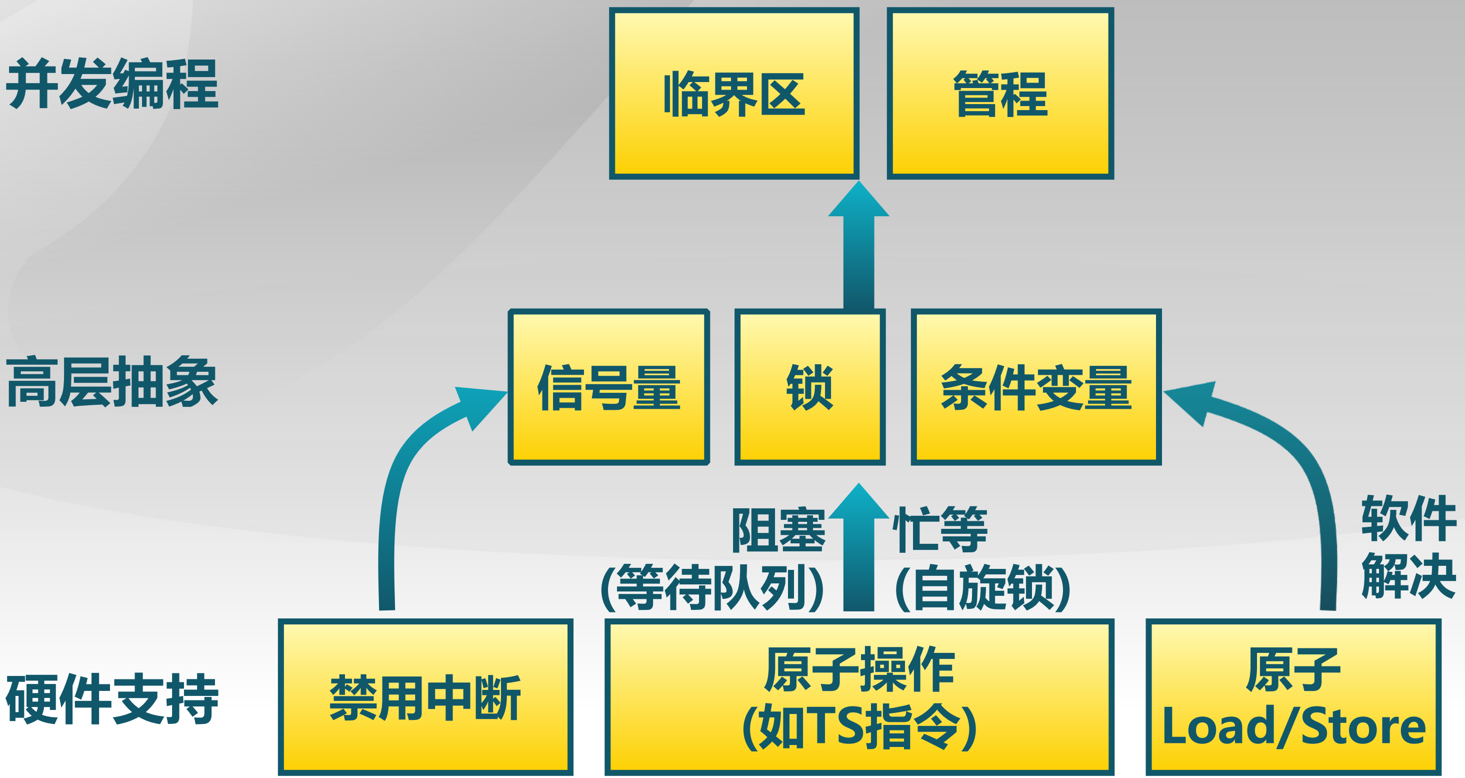
并发编程
高层抽象
硬件支持
临界区
信号量
锁
阻塞
(等待队列)
忙等
(自旋锁)
软件
解决
禁用中断
原子操作
(如TS指令)
原子
Load/Store

基本同步方法



并发编程
临界区
管程
高层抽象
信号量
锁
阻塞
(等待队列)
忙等
(自旋锁)
软件
解决
硬件支持
禁用中断
原子操作
(如TS指令)
原子
Load/Store

基本同步方法

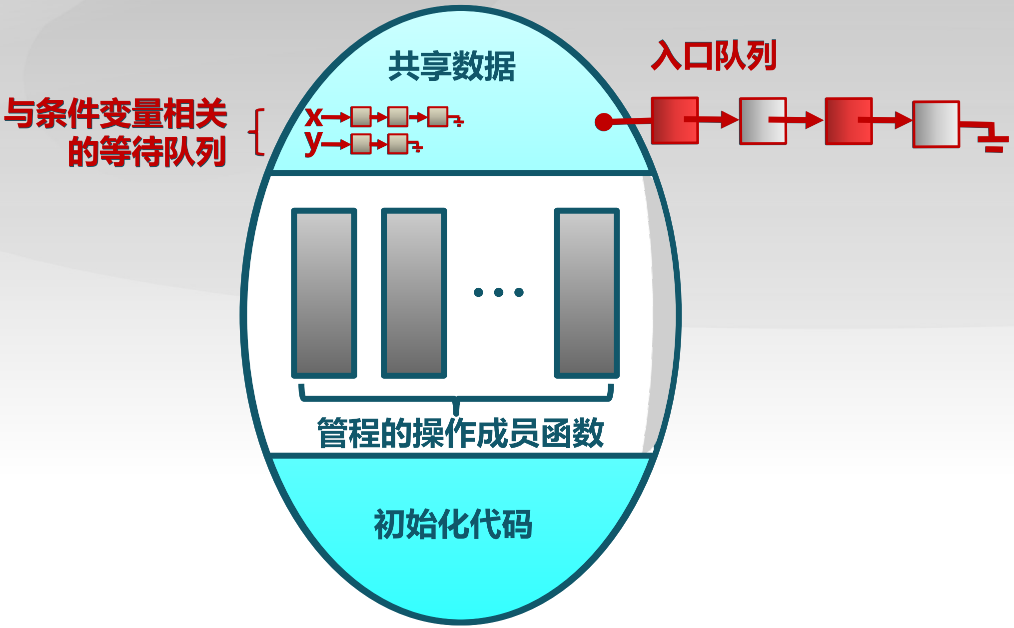

并发编程
临界区
管程
高层抽象
信号量
锁
条件变量
阻塞
(等待队列)
忙等
(自旋锁)
软件
解决
硬件支持
禁用中断
原子操作
(如TS指令)
原子
Load/Store

管程（Moniter）

- 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，简化了线程间的同步控制
  - 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
- 管程的使用
  - 在对象/模块中，收集相关共享数据
  - 定义访问共享数据的方法

管程的组成

- 一个锁
  - 控制管程代码的互斥访问
- 0或者多个条件变量
  - 管理共享数据的并发访问



条件变量（Condition Variable）

- 条件变量是管程内的等待机制
  - 进入管程的线程因资源被占用而进入等待状态
  - 每个条件变量表示一种等待原因，对应一个等待队列
- Wait()操作
  - 将自己阻塞在等待队列中
  - 唤醒一个等待者或释放管程的互斥访问
- Signal()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

条件变量实现

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

```
Condition::Wait(lock){



}
```

```
Condition::Signal(){



}
```

条件变量实现

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

```
Condition::Wait(lock){
    numWaiting++;

}
```

```
Condition::Signal(){

}
```

条件变量实现

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

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


}
```

```
Condition::Signal(){



}
```

条件变量实现

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

```
Condition::Wait(lock){
    numWaiting++;
    Add this thread t  to q;
    release(lock);
    schedule(); //need mutex

}
```

```
Condition::Signal(){




}
```

条件变量实现

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

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

```
Condition::Signal(){


}
```

条件变量实现

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

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

```
Condition::Signal(){
    if (numWaiting > 0) {



    }
}
```

条件变量实现

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

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

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


    }
}
```

条件变量实现

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

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

```
Condition::Signal(){
    if (numWaiting > 0) {
        Remove a thread t from q;
        wakeup(t); //need mutex

    }
}
```

条件变量实现

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

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

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

用管程解决生产者-消费者问题

```
classBoundedBuffer {
    …
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

```
BoundedBuffer::Deposit(c) {



    Add c to the buffer;
    count++;


}
```

```
BoundedBuffer::Remove(c) {



    Remove c from buffer;
    count--;


}
```

用管程解决生产者-消费者问题

```
classBoundedBuffer {
    …
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

```
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();
}
```

用管程解决生产者-消费者问题

```
classBoundedBuffer {
    …
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

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

    lock->Release();
}
```

```
BoundedBuffer::Remove(c) {
    lock->Acquire();


    Remove c from buffer;
    count--;
    notFull.Signal();
    lock->Release();
}
```

用管程解决生产者-消费者问题

```
classBoundedBuffer {
    …
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

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

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

管程条件变量的释放处理方式

- Hansen管程
  - 主要用于真实OS和Java中

```
l.acquire()
…
x.wait()
```

T1进入等待

T2进入管程

```
l.acquire()
…
x.signal()
…
l.release()
```

T2退出管程

```
…
l.release()
```

T1恢复管程执行

- Hoare管程
  - 主要见于教材中

```
l.acquire()
…
x.wait()
```

T1进入等待

T2进入管程

```
l.acquire()
…
x.signal()
```

T2进入等待

```
…
l.release()
```

T1恢复管程执行

T1 结束

T2恢复管程执行

```
…
l.release()
```

Hansen 管程与 Hoare 管程

```
Hansen-style :Deposit(){
   lock->acquire();
   while (count == n) {
        notFull.wait(&lock);
   }
   Add  thing;
   count++;
   notEmpty.signal();
   lock->release();
}
```

```
Hoare-style: Deposit(){
   lock->acquire();
   if (count == n) {
        notFull.wait(&lock);
   }
   Add thing;
   count++;
   notEmpty.signal();
   lock->release();
}
```

- Hansen管程
  - 条件变量释放仅是一个提示
  - 需要重新检查条件
- 特点
  - 高效

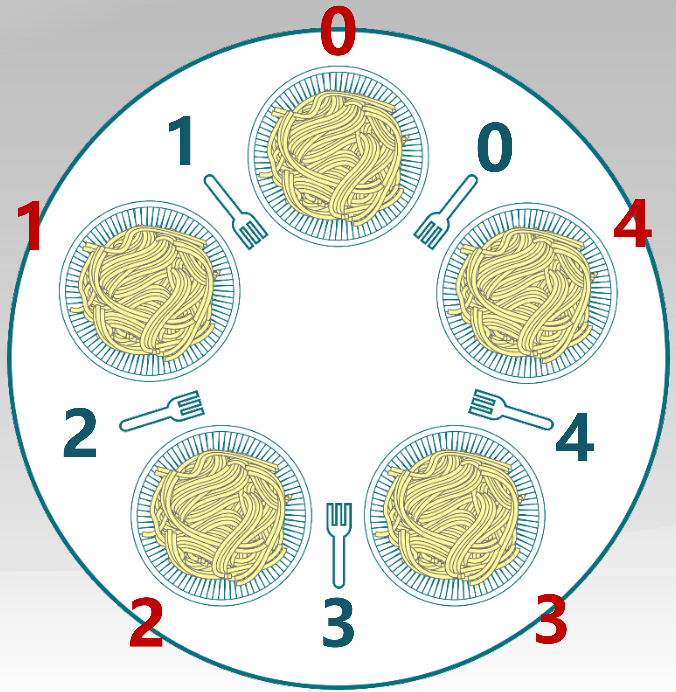
- Hoare管程
  - 条件变量释放同时表示放弃管程访问
  - 释放后条件变量的状态可用
- 特点
  - 低效

操作系统
Operating Systems

哲学家就餐问题

问题描述：

- 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
- 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?
  如：不出现有人永远拿不到叉子



方案1

```
#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

```
#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
```

方案2

```
#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



        eat( );                        // 吃面条中….



    }
```

方案2

```
#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


        eat( );                          // 吃面条中….


        V(mutex);                        // 退出临界区
    }
```

方案2

```
#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(mutex);                            // 退出临界区
    }
```

方案2

```
#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

```
#define    N    5                              // 哲学家个数
semaphore fork[5];                            // 信号量初值为1
```

方案3

```
#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




        eat( );                              // 吃面条中....


    }
```

方案3

```
#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) {


        } else {


        }
        eat( );                        // 吃面条中….


    }
```

方案3

```
#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 {


        }
        eat( );                                  // 吃面条中….


    }
```

方案3

```
#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….


    }
```

方案3

```
#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

操作系统

Operating Systems

读者-写者问题描述

- 共享数据的两类使用者
  - 读者：只读取数据，不修改
  - 写者：读取和修改数据
- 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

Writer

```
write;
```

Reader

```
read;
```

用信号量解决读者-写者问题

Writer

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

Reader

```
    P(WriteMutex);

read;

    V(WriteMutex);
```

用信号量解决读者-写者问题

Writer

```
P(WriteMutex);

  write;

V(WriteMutex);
```

Reader

```
 if (Rcount == 0)
   P(WriteMutex);
 ++Rcount;


read;




   V(WriteMutex);
```

用信号量解决读者-写者问题

Writer

```
P(WriteMutex);

  write;

V(WriteMutex);
```

Reader

```
  if (Rcount == 0)
    P(WriteMutex);
  ++Rcount;


read;


  --Rcount;
  if (Rcount == 0)
   V(WriteMutex);
```

用信号量解决读者-写者问题

Writer

```
P(WriteMutex);

  write;

V(WriteMutex);
```

Reader

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

read;


 --Rcount;
 if (Rcount == 0)
  V(WriteMutex);
```

用信号量解决读者-写者问题

Writer

```
P(WriteMutex);

  write;

V(WriteMutex);
```

Reader

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

read;

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

此实现中，读者优先

读者/写者问题：优先策略

- 读者优先策略
  - 只要有读者正在读状态，后来的读者都能直接进入
  - 如读者持续不断进入，则写者就处于饥饿
- 写者优先策略
  - 只要有写者就绪，写者应尽快执行写操作
  - 如写者持续不断就绪，则读者就处于饥饿

如何实现？

用管程解决读者-写者问题

- 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
```

用管程解决读者-写者问题

- 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;
```

解决方案详情：读者

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

解决方案详情：读者

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

```
Private Database::StartRead() {
    lock.Acquire();




    lock.Release();
}
```

解决方案详情：读者

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

```
Private Database::StartRead() {
    lock.Acquire();




    AR++;
    lock.Release();
}
```

解决方案详情：读者

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

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

解决方案详情：读者

```
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::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();
}
```

解决方案详情：读者

```
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::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--;



    lock.Release();
}
```

解决方案详情：读者

```
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::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 (???) {
        okToWrite.signal();
    }
    lock.Release();
}
```

解决方案详情：读者

```
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::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();
}
```

解决方案详情：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();




    AW++;
    lock.Release();
}
```

解决方案详情：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 (???)  {
          WW++;
          okToWrite.wait(&lock);
          WW--;
     }
     AW++;
     lock.Release();
}
```

解决方案详情：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();
}
```

解决方案详情：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--;

    lock.Release();
}
```

解决方案详情：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();
    }


    lock.Release();
}
```

解决方案详情：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();
}
```

第十四讲：信号量与管程

第 6 节：Rust 语言中的同步机制

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

提纲

1. 第 6 节：Rust 语言中的同步机制
   - 引用计数
   - 原子操作
   - 执行同步
   - 条件变量
   - 互斥信号量
   - 读写锁

Higher-level synchronization objects in Rust

- Arc: A thread-safe atomically Reference-Counted pointer, which can be used in multithreaded environments.
- Barrier: Ensures multiple threads will wait for each other to reach a point in the program, before continuing execution all together.
- Condvar: Condition Variable, providing the ability to block a thread while waiting for an event to occur.
- Mutex: Mutual Exclusion mechanism, which ensures that at most one thread at a time is able to access some data.
- RwLock: Provides a mutual exclusion mechanism which allows multiple readers at the same time, while allowing only one writer at a time.

Rc(Reference Counting)

A single-threaded reference-counting pointer. 'Rc' stands for 'Reference Counted'.

```
pub struct Rc<T: ?Sized> {
    ptr: NonNull<RcBox<T>>,
    phantom: PhantomData<T>,
}
```

Example of Reference Counting

```rust
use std::rc::Rc;

fn main() {
    let rc_examples = "Rc examples".to_string();
    {
        println!("--- rc_a is created ---");

        let rc_a: Rc<String> = Rc::new(rc_examples);
        println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

        {
            println!("--- rc_a is cloned to rc_b ---");

            let rc_b: Rc<String> = Rc::clone(&rc_a);
            println!("Reference Count of rc_b: {}", Rc::strong_count(&rc_b));
            println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

            // Two `Rc`s are equal if their inner values are equal
            println!("rc_a and rc_b are equal: {}", rc_a.eq(&rc_b));

            // We can use methods of a value directly
            println!("Length of the value inside rc_a: {}", rc_a.len());
            println!("Value of rc_b: {}", rc_b);

            println!("--- rc_b is dropped out of scope ---");
        }

        println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

        println!("--- rc_a is dropped out of scope ---");
    }

    // Error! `rc_examples` already moved into `rc_a`
    // And when `rc_a` is dropped, `rc_examples` is dropped together
    // println!("rc_examples: {}", rc_examples);
    // TODO ^ Try uncommenting this line
}
```

```
--- rc_a is created ---
Reference Count of rc_a: 1
--- rc_a is cloned to rc_b ---
Reference Count of rc_b: 2
Reference Count of rc_a: 2
rc_a and rc_b are equal: true
Length of the value inside rc_a: 11
Value of rc_b: Rc examples
--- rc_b is dropped out of scope ---
Reference Count of rc_a: 1
--- rc_a is dropped out of scope ---
```

Arc: Atomically Reference-Counted pointer

A thread-safe reference-counting pointer. 'Arc' stands for 'Atomically Reference Counted'.

```
pub struct Arc<T: ?Sized> {
    ptr: NonNull<ArcInner<T>>,
    phantom: PhantomData<ArcInner<T>>,
}
```

Methods in std::sync::Arc

```rust
pub fn new(data: T) -> Arc<T>
pub fn new_uninit() -> Arc<MaybeUninit<T>>
pub fn new_zeroed() -> Arc<MaybeUninit<T>>
pub fn pin(data: T) -> Pin<Arc<T>>
pub fn try_unwrap(this: Arc<T>) -> Result<T, Arc<T>>
pub fn new_uninit_slice(len: usize) -> Arc<[MaybeUninit<T>]>
pub unsafe fn assume_init(self) -> Arc<[T]>
pub fn into_raw(this: Arc<T>) -> *const T
pub unsafe fn from_raw(ptr: *const T) -> Arc<T>
pub fn into_raw_non_null(this: Arc<T>) -> NonNull<T>
pub fn downgrade(this: &Arc<T>) -> Weak<T>
pub fn weak_count(this: &Arc<T>) -> usize
pub fn strong_count(this: &Arc<T>) -> usize
pub fn ptr_eq(this: &Arc<T>, other: &Arc<T>) -> bool
pub fn make_mut(this: &mut Arc<T>) -> &mut T
pub fn get_mut(this: &mut Arc<T>) -> Option<&mut T>
pub unsafe fn get_mut_unchecked(this: &mut Arc<T>) -> &mut T
pub fn downcast<T>(self) -> Result<Arc<T>, Arc<dyn Any + 'static + Send + Sync>>
```

Atomic

Atomic types provide primitive shared-memory communication between threads, and are the building blocks of other concurrent types.

```
pub struct $atomic_type {
            v: UnsafeCell<$int_type>,
}
```

Methods in Atomic

```rust
impl AtomicUsize
pub const fn new(v: usize) -> Self
pub fn get_mut(&mut self) -> &mut usize
pub fn into_inner(self) -> usize
pub fn load(&self, order: Ordering) -> usize
pub fn store(&self, val: usize, order: Ordering)
pub fn swap(&self, val: usize, order: Ordering) -> usize
pub fn compare_and_swap
pub fn compare_exchange
pub fn compare_exchange_weak
pub fn fetch_add(&self, val: usize, order: Ordering) -> usize
pub fn fetch_sub(&self, val: usize, order: Ordering) -> usize
pub fn fetch_and(&self, val: usize, order: Ordering) -> usize
pub fn fetch_nand(&self, val: usize, order: Ordering) -> usize
pub fn fetch_or(&self, val: usize, order: Ordering) -> usize
pub fn fetch_xor(&self, val: usize, order: Ordering) -> usize
pub fn fetch_update<F>
pub fn fetch_max(&self, val: usize, order: Ordering) -> usize
pub fn fetch_min(&self, val: usize, order: Ordering) -> usize
pub fn as_mut_ptr(&self) -> *mut usize
```

Barrier

A barrier enables multiple threads to synchronize the beginning of some computation.

```
pub struct Barrier {
    lock: Mutex<BarrierState>,
    cvar: Condvar,
    num_threads: usize,
}

impl Barrier
pub fn new(n: usize) -> Barrier
pub fn wait(&self) -> BarrierWaitResult
```

Example of Barrier

```rust
#![allow(unused)]
fn main() {
use std::sync::{Arc, Barrier};
use std::thread;

let mut handles = Vec::with_capacity(10);
let barrier = Arc::new(Barrier::new(10));
for _ in 0..10 {
    let c = barrier.clone();
    // The same messages will be printed together.
    // You will NOT see any interleaving.
    handles.push(thread::spawn(move|| {
        println!("before wait");
        c.wait();
        println!("after wait");
    }));
}
// Wait for other threads to finish.
for handle in handles {
    handle.join().unwrap();
}
}
```

Execution Close

Standard Error

```
Compiling playground v0.0.1 (/playground)
 Finished dev [unoptimized + debuginfo] target(s) in 0.78s
  Running `target/debug/playground`
```

Standard Output

```
before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
```

Condvar

Condition variables represent the ability to block a thread such that it consumes no CPU time while waiting for an event to occur.

```
pub struct Condvar {
    inner: Box<sys::Condvar>,
    mutex: AtomicUsize,
}

impl Condvar
pub fn new() -> Condvar
pub fn wait<'a, T>
pub fn wait_while<'a, T, F>
pub fn wait_timeout_ms<'a, T>
pub fn wait_timeout<'a, T>
pub fn wait_timeout_while<'a, T, F>
pub fn notify_one(&self)
pub fn notify_all(&self)
```

Example of Condvar

```rust
#![allow(unused)]
fn main() {
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = pair.clone();

thread::spawn(move|| {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    // We notify the condvar that the value has changed.
    println!("notify_all");
    cvar.notify_all();
});

// Wait for the thread to start up.
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
// As long as the value inside the `Mutex<bool>` is `false`, we wait.
while !*started {
    println!("before wait");
    started = cvar.wait(started).unwrap();
    println!("after wait");
}
}
```

Execution Close

Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/playground`
```

Standard Output

```
before wait
notify_all
after wait
```

Mutex

A mutual exclusion primitive useful for protecting shared data.

```
pub struct Mutex<T: ?Sized> {
    // Note that this mutex is in a *box*, not inlined into the struct itself.
    // Once a native mutex has been used once, its address can never change (it
    // can't be moved). This mutex type can be safely moved at any time, so to
    // ensure that the native mutex is used correctly we box the inner mutex to
    // give it a constant address.
    inner: Box<sys::Mutex>,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}

impl<T> Mutex<T>
pub fn new(t: T) -> Mutex<T>
pub fn lock(&self) -> LockResult<MutexGuard<T>>
pub fn try_lock(&self) -> TryLockResult<MutexGuard<T>>
pub fn is_poisoned(&self) -> bool
pub fn into_inner(self) -> LockResult<T>
pub fn get_mut(&mut self) -> LockResult<&mut T>
```

RwLock

Rwlock allows a number of readers or at most one writer at any point in time. The write portion of this lock typically allows modification of the underlying data (exclusive access).

```
pub struct RwLock<T: ?Sized> {
    inner: Box<sys::RWLock>,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

Methods in Rwlock

```
impl<T: ?Sized> RwLock<T>
pub fn read(&self) -> LockResult<RwLockReadGuard<T>>
pub fn try_read(&self) -> TryLockResult<RwLockReadGuard<T>>
pub fn write(&self) -> LockResult<RwLockWriteGuard<T>>
pub fn try_write(&self) -> TryLockResult<RwLockWriteGuard<T>>
pub fn is_poisoned(&self) -> bool
pub fn into_inner(self) -> LockResult<T>
pub fn get_mut(&mut self) -> LockResult<&mut T>
```