学习目标

1. 熟练掌握互斥量的使用
2. 说出什么叫死锁以及解决方案
3. 熟练掌握读写锁的使用
4. 熟练掌握条件变量的使用
5. 理解条件变量实现的生产消费者模型
6. 理解信号量实现的生产消费者模型

1 - 互斥量(互斥锁)

1. 互斥锁类型:
   创建一把锁: pthread_mutex_t muext;
2. 互斥锁的特点:
   - 多个线程访问共享数据的时候是串行的
3. 使用互斥锁缺点?
   - 效率低
4. 互斥锁的使用步骤:
   - 创建互斥锁: pthread_mutex_t mutex;
   - 初始化这把锁: pthread_mutex_init(&mutex, NULL); -- mutex = 1
   - 寻找共享资源:
     - 操作共享资源的代码之前加锁
       - pthread_mutex_lock(&mutex); -- mutex = 0
       - 。。。。。。
       - 。。。。。。 临界区
       - pthread_mutex_unlock(&mutex); -- mutex = 1
5. 互斥锁相关函数:
   - 初始化互斥锁

     ```
     pthread_mutex_init(
         pthread_mutex_t *restrict mutex,
         const pthread_mutexattr_t *restrict attr
     );
     ```
   - 销毁互斥锁

     ```
     pthread_mutex_destroy(pthread_mutex_t *mutex);
     ```
   - 加锁

     ```
     pthread_mutex_lock(pthread_mutex_t *mutex);
     ```
     - mutex:
       - 没有被上锁, 当前线程会将这把锁锁上
       - 被锁上了: 当前线程阻塞
         - 锁被打开之后, 线程解除阻塞
   - 尝试加锁, 失败返回, 不阻塞

pthread_mutex_trylock(pthread_mutex_t *mutex);

- 没有锁上：当前线程会给这把锁加锁
- 如果锁上了：不会阻塞，返回

```
if(pthread_mutex_trylock(&mutex)==0)
{
    // 尝试加锁，并且成功了
    // 访问共享资源
}
else
{
    // 错误处理
    // 或者 等一会，再次尝试加锁
}
```

- 解锁

pthread_mutex_unlock(pthread_mutex_t *mutex);

如果我们想使用互斥锁同步线程：

所有的线程都需要加锁

2 - 死锁

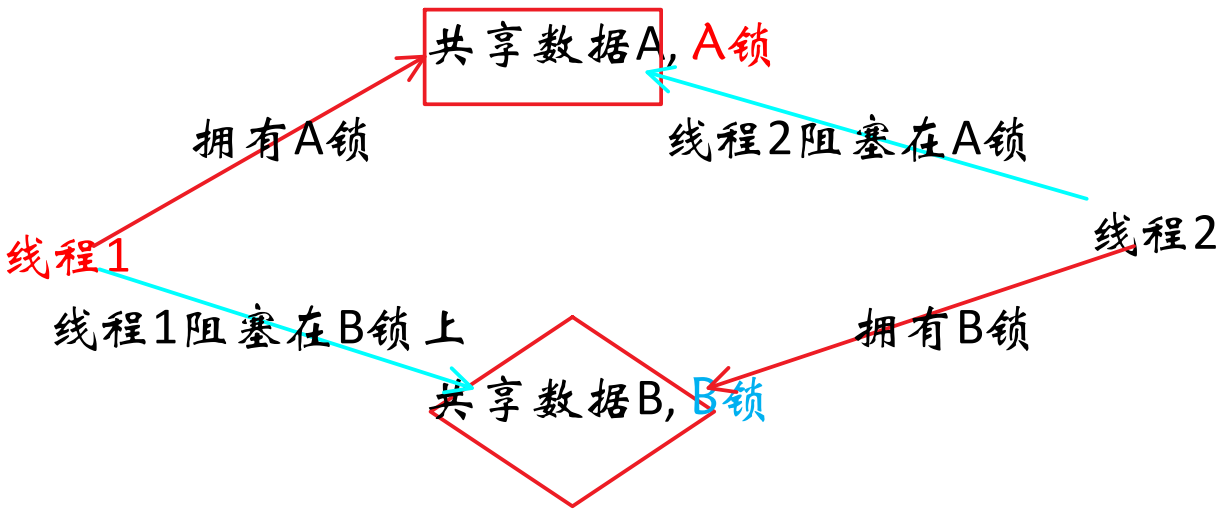
造成死锁的原因:

1. 自己锁自己

```
for(int i=0; i<MAX; ++i)
{
    // 加锁
    pthread_mutex_lock(&mutex);
    pthread_mutex_lock(&mutex);
    int cur = number;
    cur++;
    number = cur;
    printf("Thread A, id = %lu, number = %d\n", pthread_self(), number);
    // 解锁
    pthread_mutex_unlock(&mutex);
    usleep(10);
}
```

操作做完成之后,一定要解锁

2.



线程1对共享资源A加锁成功 - A锁
线程2对共享资源b加锁成功 - B锁

线程1访问共享资源B, 对b锁加锁 - 线程1阻塞在B锁上
线程2访问共享资源A, 对A锁加锁 - 线程2阻塞在A锁上

如何解决：

- 让线程按照一定的顺序去访问共享资源
- 在访问其他锁的时候，需要先将自己的锁解开
- trylock

3 - 读写锁

1. 读写锁是几把锁？
   - 一把锁
   - pthread_rwlock_t lock;
2. 读写锁的类型：
   - 读锁 - 对内存做读操作
   - 写锁 - 对内存做写操作
3. 读写锁的特性：
   - 线程A加读锁成功，又来了三个线程，做读操作，可以加锁成功
     - 读共享 - 并行处理
   - 线程A加写锁成功，又来了三个线程，做读操作，三个线程阻塞
     - 写独占
   - 线程A加读锁成功，又来了B线程加写锁阻塞，又来了C线程加读锁阻塞
     - 读写不能同时
     - 写的优先级高
4. 读写锁场景练习：
   - 线程A加写锁成功，线程B请求读锁
     - 线程B阻塞
   - 线程A持有读锁，线程B请求写锁
     - 线程B阻塞
   - 线程A拥有读锁，线程B请求读锁
     - 线程B加锁成功
   - 线程A持有读锁，然后线程B请求写锁，然后线程C请求读锁
     - B阻塞，c阻塞 - 写的优先级高
     - A解锁，B线程加写锁成功，C继续阻塞
     - B解锁，C加读锁成功
   - 线程A持有写锁，然后线程B请求读锁，然后线程C请求写锁
     - BC阻塞
     - A解锁，C加写锁成功，B继续阻塞
     - C解锁，B加读锁成功
5. 读写锁的适用场景？
   - 互斥锁 - 读写串行
   - 读写锁：
     - 读：并行

        - 写：串行
    - 程序中的读操作》写操作的时候
6. 主要操作函数
    - 初始化读写锁

      pthread_rwlock_init(
          pthread_rwlock_t *restrict rwlock,
          const pthread_rwlockattr_t *restrict attr
      );
    - 销毁读写锁

      pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
    - 加读锁

      pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
        - 阻塞：之前对这把锁加的写锁的操作
    - 尝试加读锁

      pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
        - 加锁成功：0
        - 失败：错误号
    - 加写锁

      pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
        - 上一次加锁写锁，还没有解锁的时候
        - 上一次加读锁，没解锁
    - 尝试加写锁

      pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
    - 解锁

      pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

7. 练习：
    - 3个线程不定时写同一全局资源，5个线程不定时读同一全局资源
        - 先不加锁
8. 读写锁，互斥锁
    - 阻塞线程
    - 不是什么时候都能阻塞线程
        - 链表 Node *head = NULL
        - while（head == NULL）
        - {

```
        // 我们想让代码在这个位置阻塞
        // 等待链表中有了节点之后再继续向下运行
        // 使用了后边要讲的条件变量 - 阻塞线程
    }
    // 链表不为空的处理代码
    。。。。。。
    。。。。。。
```

4 - 条件变量

1. 条件变量是锁吗?
   - 不是锁, 但是条件变量能够阻塞线程
   - 使用条件变量 + 互斥量
     - 互斥量: 保护一块共享数据
     - 条件变量: 引起阻塞
       - 生产者和消费者模型
2. 条件变量的两个动作?
   - 条件不满足, 阻塞线程
   - 当条件满足, 通知阻塞的线程开始工作
3. 条件变量的类型:
   - pthread_cond_t cond;
4. 主要函数:
   - 初始化一个条件变量 - condtion

     pthread_cond_init(
     ```
     pthread_cond_t *restrict cond,
     const pthread_condattr_t *restrict attr
     ```
     );
   - 销毁一个条件变量

     pthread_cond_destroy(pthread_cond_t *cond);
   - 阻塞等待一个条件变量

     pthread_cond_wait(
     ```
     pthread_cond_t *restrict cond,
     pthread_mutex_t *restrict mutex
     ```
     );
     - 阻塞线程
     - 将已经上锁的mutex解锁
     - 该函数解除阻塞, 会对互斥锁加锁

- 限时等待一个条件变量

```
pthread_cond_timedwait(
    pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime
);
```

- 唤醒至少一个阻塞在条件变量上的线程

```
pthread_cond_signal(pthread_cond_t *cond);
```

- 唤醒全部阻塞在条件变量上的线程

```
pthread_cond_broadcast(pthread_cond_t *cond);
```

5. 练习
   - 使用条件变量实现生产者,消费者模型

5 - 信号量(信号灯)

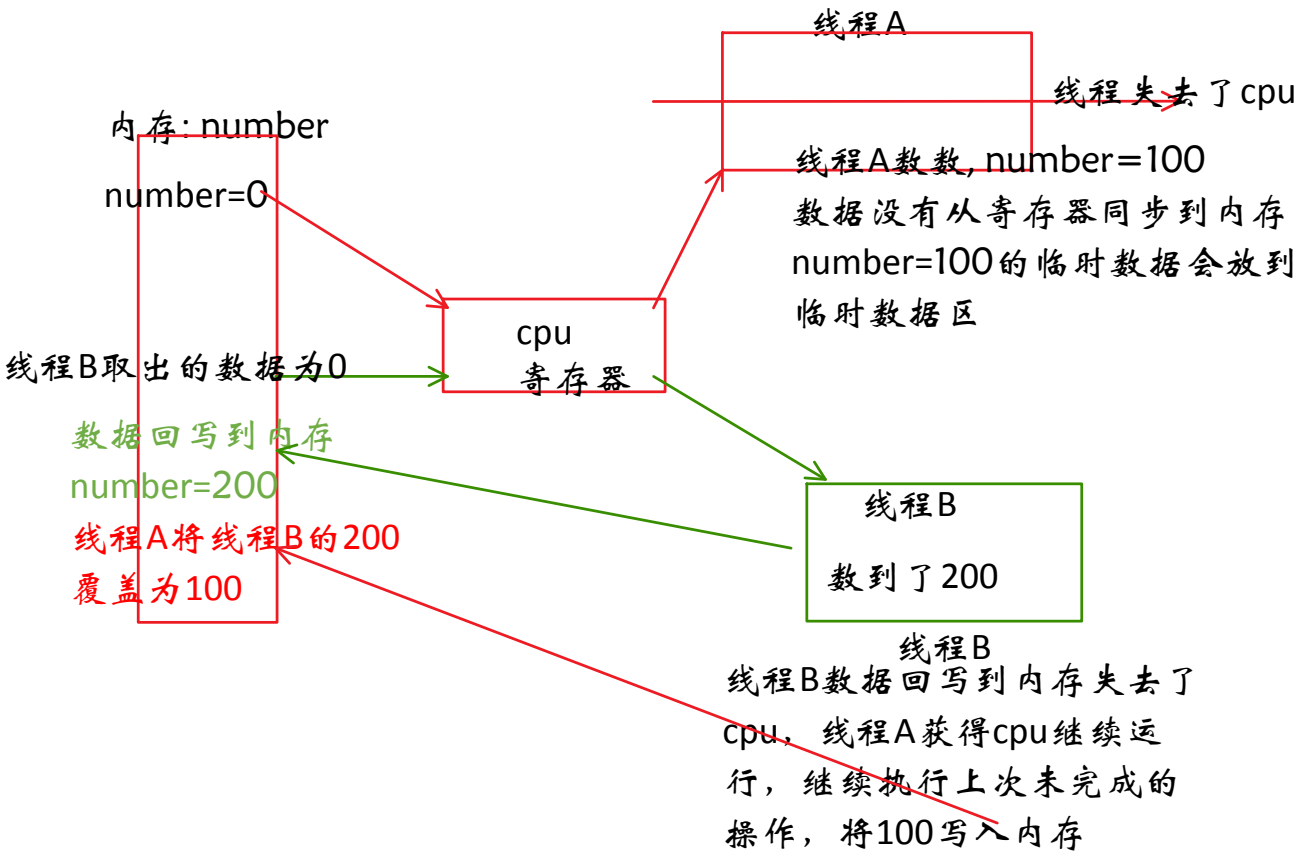
1. 头文件 - semaphore.h
2. 信号量类型
   - sem_t sem;
   - 加强版的互斥锁
3. 主要函数
   - 初始化信号量
     sem_init(sem_t *sem, int pshared, unsigned int value);
     - 0 - 线程同步
     - 1 - 进程同步
     - value - 最多有几个线程操作共享数据 - 5
   - 销毁信号量
     sem_destroy(sem_t *sem);
   - 加锁 --
     sem_wait(sem_t *sem);
     调用一次相当于对sem做了--操作
     如果sem值为0, 线程会阻塞
   - 尝试加锁
     sem_trywait(sem_t *sem);
     - sem == 0, 加锁失败, 不阻塞, 直接返回
   - 限时尝试加锁
     sem_timedwait(sem_t *sem, xxxxx);
   - 解锁 ++
     sem_post(sem_t *sem);
     对sem做了++操作
4. 练习:
   - 使用信号量实现生产者,消费者模型

mutex = 1
lock（） mutex = 0
unlock（） mutex = 1
mutex实现的同步都是串行的

6-哲学家就餐模型

五个哲学家,围着一张桌子吃饭,每个哲学家只有一根筷子,需要使用旁边人的筷子才能把饭吃到嘴里.
抢到筷子的吃饭,没抢到的思考人生.
使用多线程实现多线程实现哲学家交替吃饭的模型

多线程操作共享数据



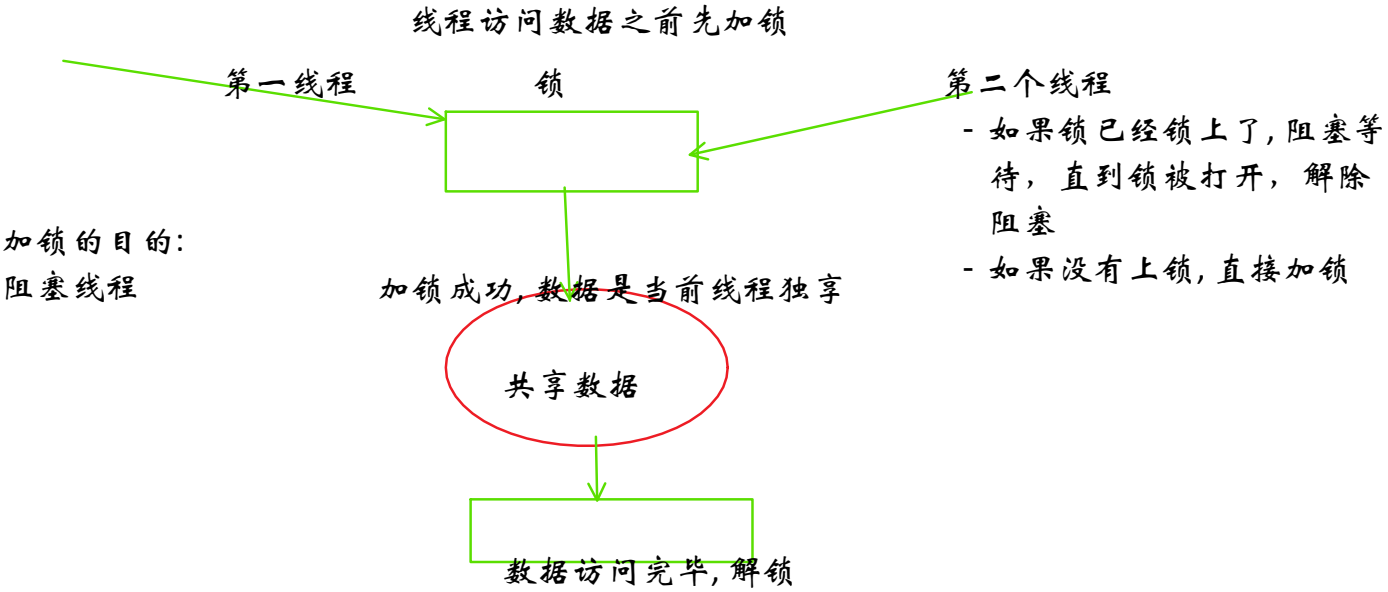
造成数据混乱的原因：

- 多个线程操作了共享数据
- cpu的调度文件
- 提供一套同步机制

什么叫线程同步？

让多个线程协同步调，先后处理某件事情

线程同步基本操作步骤


线程访问数据之前先加锁
第一线程
锁
第二个线程
- 如果锁已经锁上了，阻塞等待，直到锁被打开，解除阻塞
- 如果没有上锁，直接加锁
加锁的目的：
阻塞线程
加锁成功，数据是当前线程独享
共享数据
数据访问完毕，解锁

原子操作

代码1：

```
void* producer(void* arg)
 {
   // 一直生产
   while(1)
   {
     // 创建一个链表的节点
     Node* newNode = (Node*)malloc(sizeof(Node));
     // init
     newNode->data = rand() % 1000;
     newNode->next = head;
     head = newNode;
     printf("+++ producer: %d\n", newNode->data);

     sleep(rand()%3);
   }
   return NULL;
 }
```

代码2：

```
void* producer(void* arg)
 {
   // 一直生产
   while(1)
   {
     // 创建一个链表的节点
     Node* newNode = (Node*)malloc(sizeof(Node));
     // init
     newNode->data = rand() % 1000;
     pthread_mutex_lock(&mutex);
     newNode->next = head;
     head = newNode;
     printf("+++ producer: %d\n", newNode->data);
     pthread_mutex_unlock(&mutex);

     sleep(rand()%3);
   }
   return NULL;
 }
```

原子操作

- cup处理一个指令，线程/进程在处理完这个指令之前是不会失去cpu的
  - printf()
  - int a = b+100;

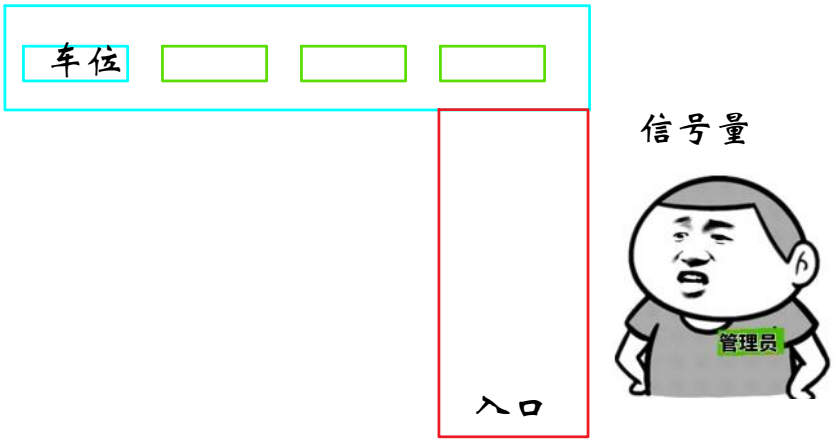
临界区

两个线程：

- 都阻塞了

共享资源，线程，互斥锁

车位：共享资源，管理员：锁 车：线程



如果互斥锁：
管理员每次只能放一辆车进去
信号量：
如果是管理员，每次最多可以放4辆车进去
可以运行多个线程同时访问共享资源

互斥锁：串行
信号量：并行

mutex = 1

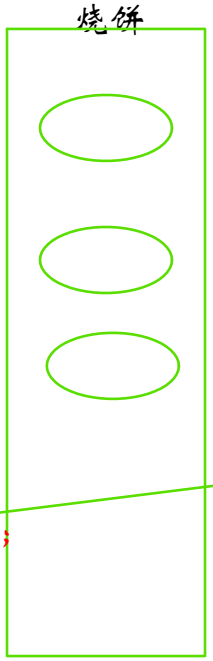
生产者：对应一个信号量：sem_t produce；
消费者：对应以信号量： sem_t customer;
sem_init(&produce, 2); -- 生产者拥有资源，可以工作
sem_init(&customer, 0); -- 消费者没有资源，阻塞

条件变量

2017年3月27日 11:37

head

烧饼



生产者

```
// 一直生产
while（1）
{
    // 创建一个节点
    pthread_mutex_lock(&mutex);
    // 节点插入到链表 - head
    pthead_mutex_unlock(&mutex);
    // 通知阻塞的消费者接触阻塞
    pthread_cond_signal（&cond）;
}
```

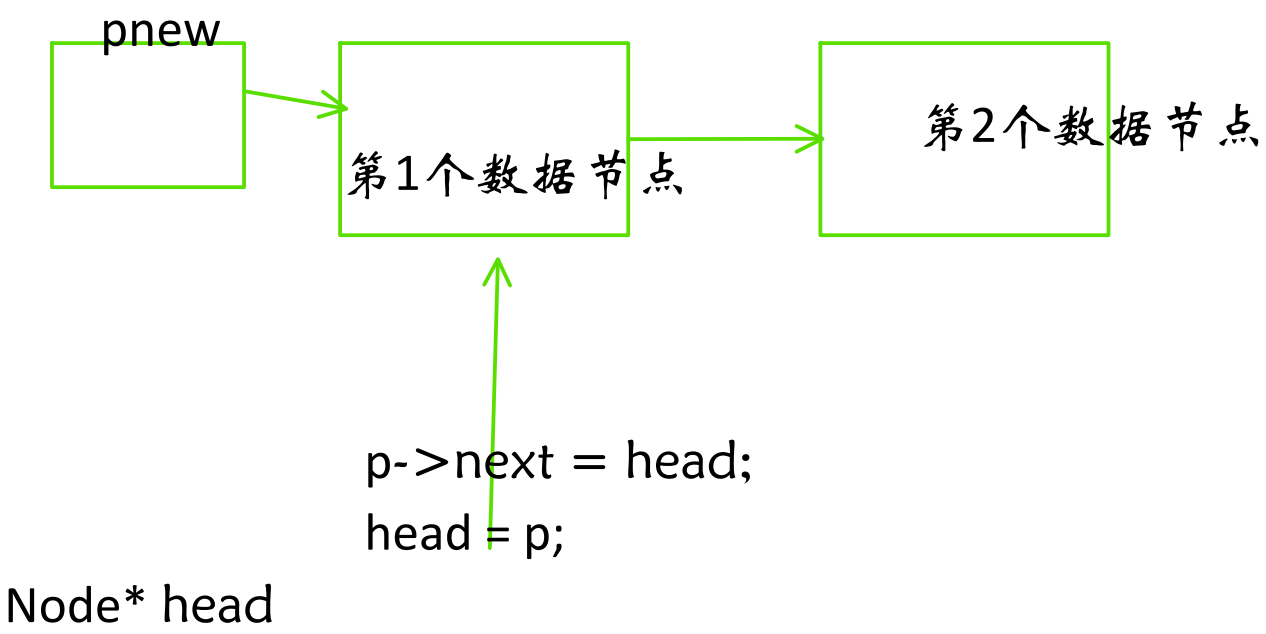
消费者

```
// 一直消费
while(1)
{
    // 判断是不是有烧饼
    pthread_mutex_lock(&mutex);
    if(head == NULL)
    {
        // 阻塞等待
        pthread_cond_wait(&cond,
        &mutex)；
    }
    // 吃烧饼 - 访问共享数据
    pthread_mutex_unlock(&mutex);
    sleep（1）；
}
```

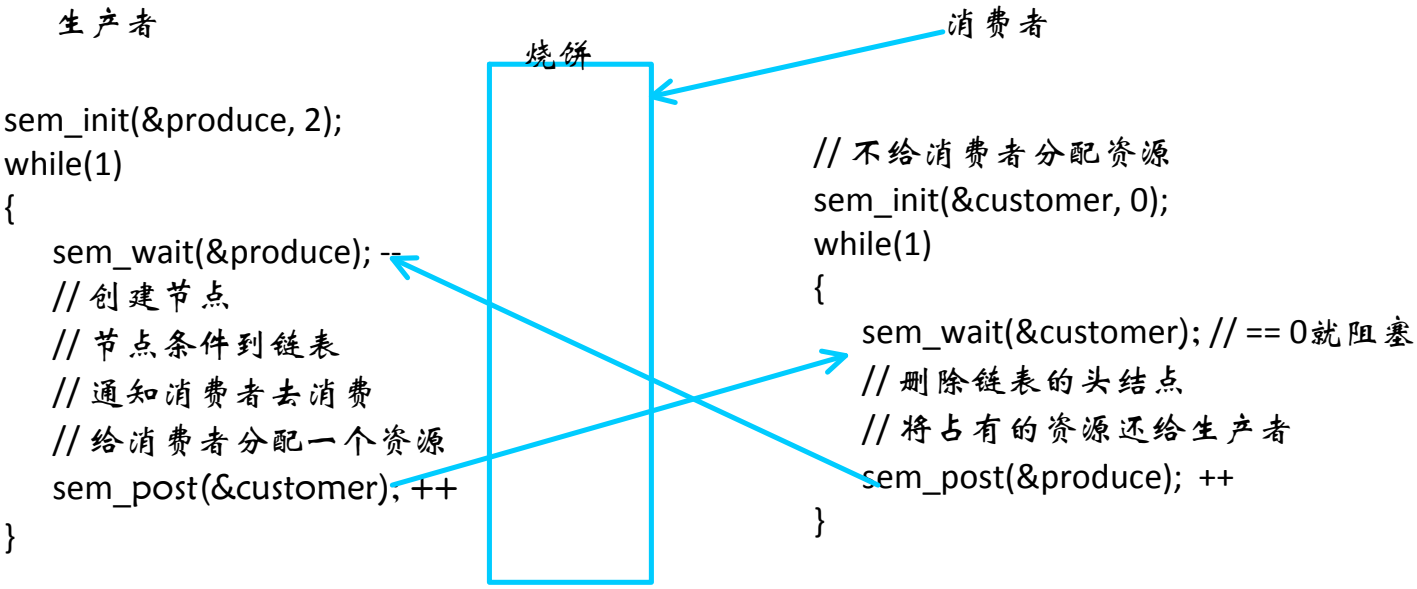
不带头结点的链表

2017年3月27日　　11:40

头插法　头删法



信号量 - 生产者消费者



生产者：对应一个信号量：sem_t produce；
消费者：对应以信号量：　sem_t customer;
sem_init(&produce, 2); -- 生产者拥有资源，可以工作
sem_init(&customer, 0); -- 消费者没有资源，阻塞