

# 8互斥和同步

## 一.指令流间的竞争

参与关系的基本单位——指令流

指令流之间的关系：

竞争关系——最常用的是文件和设备的争用，争用所有权

合作关系——操作的执行按一定的先后次序完成

互补相干——但是所有指令流都有占用量限额的竞争关系

用数学语言描述指令流之间的关系——偏序集（自反，反对称，传递）

严格的偏序集的关系图就是一个有向无环图

### 1.1 临界区

**临界资源**——不能被两个指令流同时使用的资源。只能分别在两个指令流中独占共享。

**共享资源**——将一次允许多个进程同时使用的资源称为共享资源。

**临界区**——访问临界资源的程序段，临界区不能并行的，越短越好

临界区的进入和退出

进入临界区（判断+等待）——>访问临界区（访问临界资源）——>退出临界区

**互斥**——临界资源不能同时使用的现象

### 1.2 自旋锁

单标志法

**算法基本思想：**设置一个公用变量 turn，用于指示被允许进入临界区的进程编号，即若 turn = 0，则允许 P0 进程进入临界区。算法可保证同一时刻只允许一个进程进入临界区，但两个进程必须交替进入临界区。

```
int turn = 0;
```

**指令流S0**

① while (turn == 1) {}

② 访问临界资源;

③ turn = 1;

**指令流S1**

① while (turn == 0) {}

② 访问临界资源;

③ turn = 0;

解决了临界区互斥问题，达到了绝对公平，但是两个指令流对临界区的访问频率不同的话，就会导致饥饿。

## 双标志先检查法

**算法基本思想：**设置一个布尔型数组 want[]，数组中各个元素用来标记各进程想进入临界区的意愿，比如 want[0] = true 意味着 0 号指令流 s0 现在想要进入临界区。每个进程在进入临界区之前先检查当前有没有其他进程想进入临界区，如果没有，则把自身对应的标志 want[i] 设为 true，之后开始访问临界区。

```
bool flag[2];           //表示进入临界区意愿的数组
flag[0] = false;
flag[1] = false;        //刚开始设置为两个进程都不想进入临界区

P0 进程:                P1 进程:
while (flag[1]); ①      while (flag[0]); ⑤
flag[0] = true; ②      flag[1] = true; ⑥
critical section; ③    critical section; ⑦
flag[0] = false; ④    flag[1] = false; ⑧
remainder section;    remainder section;
```

理解背后的含义：“表达意愿”

进入区

//如果此时 P0 想进入临界区，P1 就一直循环等待

//标记为 P1 进程想要进入临界区

//访问临界区

//访问完临界区，修改标记为 P1 不想使用临界区

## 双标志后检查法

**算法基本思想：**双标志先检查法的改版，与前一个算法不同，采用先上锁后检查，来避免上述问题。

```
int want[2] = {0};
```

### 指令流s0

- ① want[0] = 1;
- ② while (want[1] == 1) {}
- ③ 访问临界资源;
- ④ want[0] = 0;

### 指令流s1

- ① want[1] = 1;
- ② while (want[0] == 1) {}
- ③ 访问临界资源;
- ④ want[1] = 0;

解决了临界区互斥的问题，但是两个指令流会循环等待对方，会死锁

解决死锁问题：指令流会短暂放弃进入临界区的愿望，给对方一个机会

```
int want[2] = {0};
```

## 不僵持的实现

### 指令流s0

- ① while (1)
- ① {
- ① want[0] = 1;
- ② if (want[1] == 0)
- ② break;
- ② else
- ② want[0] = 0;
- ② }
- ③ 访问临界资源;
- ④ want[0] = 0;

如果两个指令流推进的速度差不多的话，活锁

### 指令流s1

- ① while (1)
- ① {
- ① want[1] = 1;
- ② if (want[0] == 0)
- ② break;
- ② else
- ② want[1] = 0;
- ② }
- ③ 访问临界资源;
- ④ want[1] = 0;

**算法基本思想：**结合双标志法、单标志法的思想。如果双方都争着想进入临界区，那可以让进程尝试孔融让梨(谦让)，做一个有礼貌的进程。进程在进入区要做的步骤：① 主动争取 ② 主动谦让 ③ 检查对方是否也想使用，且最后一次是不是自己说了客气话

```
int want[2] = {0};  
int turn = 0;
```

### 指令流S0

- ① want[0] = 1;
- ② turn = 1;
- ③ while (want[1] == 1 && turn != 0) {}
- ④ 访问临界资源;
- ⑤ want[0] = 0;

### 指令流S1

- ① want[1] = 1;
- ② turn = 0;
- ③ while (want[0] == 1 && turn != 1) {}
- ④ 访问临界资源;
- ⑤ want[1] = 0;

解决了临界区互斥问题，不会出现死锁，活锁，饥饿问题

为了实现对临界资源的互斥访问，同时保证系统整体性能，需要遵循以下原则：

- ① 空闲让进：临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区；
- ② 忙则等待：当已有进程进入临界区时，其他试图进入临界区的进程必须等待；
- ③ 有限等待：对请求访问的进程，应保证能在有限时间内进入临界区（保证不会饥饿）；
- ④ 让权等待：当进程不能进入临界区时，应立即释放处理机，防止进程忙等待。

**互斥锁**——一种用以控制临界区访问的互斥访问原语，分为加锁和解锁两个原子操作，进入临界区加锁，退出临界区解锁

加锁——检查条件，在满足的时候获得加锁的权力，这个过程中，指令流只要在循环判断检测条件，做忙等待，像旋转的陀螺，称为自旋锁。

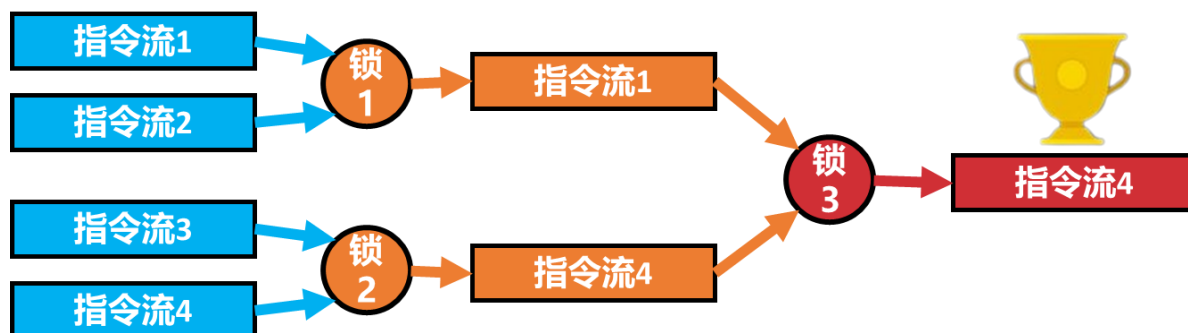
解锁——释放锁的拥有权。

### 多指令流的自旋锁

Peterson算法推广到多指令流。——>锦标赛锁

锦标赛发使一种将二选一机制扩展到多选一机制的方法，将多个对象分成两组，组内再分组，知道每组只剩下一个对象。

我们可以将多个指令流递归分组，每组先竞争自己的锁，然后选出的由盛则再竞争二级锁。



### 多指令流自旋锁：过滤器(Filter)算法

它是Peterson锁算法在多线程上的直接一般化。

### 使用数组

使用数据替换掉标量turn

填充逻辑一定与当前指令流的编号*i*有关，而且要体现出轮番进入。设计等待的指令流有一个队列，那么while循环的外壳应该有一个for循环，经每个等待者依次向前移动直到出队。turn的每个格子就代表队列相应位置上的等待者。

```
int want[n] = {0};
int turn[n] = {0};

① for (int pos = 1; pos < N; pos++)
① {
①     want[i] = pos;
②     turn[pos] = i;
③     while (want[any other] >= pos && turn[pos] == i) {}
① }

④ 访问临界资源;

⑤ want[i] = 0;
```

如果在高等级上还有竞争者

且没有新指令流加进此等级

问题：此算法还能保证像双指令流Peterson算法那样的公平性（FCFS）吗？

不能。同时加入pos等级的指令流，**谁先得到pos提升是不一定的**。如果不被调度，**一直不提升也是可能的**；后面的能超前，前面的能留级。

### 多指令流自旋锁：烘焙坊（Bakery）算法

**思路** 必须要使用某种数据结构保留进入的先后。准备一个永不回头的序列号即可，然后挑选序列号最小的那个进入临界区。

```
int want[n] = {LONG_MAX};
int serial = 0;

① serial++;
② want[i] = serial;
③ while (want[any other] < want[i]) {}

④ 访问临界资源;

⑤ want[i] = INT_MAX;
```

算法非常简单，非常无脑。但是代价是什么呢？

**问题一** serial变量溢出怎么办，后来者反而变成了最先者。

使用64位变量。简单粗暴，比地球上的沙子多。

**问题二** 对serial的++操作包含了读-改-写（RMW）。它可不是原子的

**解决方案** 如果它不是原子的，那就意味着可能存在有两个指令流被分配 同一个serial。此时，这两者之间的优先级依照指令流编号决定，较小的编号较优先。

```
int want[n] = {LLONG_MAX};
long long serial = 1;

① serial++;
② want[i] = serial;
③ while (pair{want[any other], same} < pair{want[i], i}) {}

④ 访问临界资源;

⑤ want[i] = LLONG_MAX;
```

## 字典序 Lexographical Order

当一组有多个属性的对象参加排序时，可以先按照一个属性排序，无法决定顺序的再按照另一个属性排序，依此类推，好像字典中单词的排序方法一样。

### 竞争冒险

(1) 考虑到任何地方可能发生上下文切换，如果一个指令流把 ② 执行了一半，也即取了 serial，但并未写入自己的 want[i]，它就拿着一个过时的小值。

(2) serial 次小的线程是看不到这个尚未写入内存的最小值的，因此它会直接进入临界区。

(3) 此时第一个指令流又被调度。它会直接进入临界区，此时临界区有两个指令流，破坏了互斥条件。

```
int want[n] = {LLONG_MAX};
long long serial = 1;

① serial++;
② want[i] = serial;
③ while (pair{want[any other], same} < pair{want[i], i}) {}

④ 访问临界资源;

⑤ want[i] = LLONG_MAX;
```

**解决方案** 保证等待每一个线程时，要等到它完成叫号过程才行。

```
int want[n] = {LLONG_MAX};
int written[n] = {0};
long long serial = 1;

① written[i] = 1;
① serial++;
① want[i] = serial;
① written[i] = 0;
② for (int j = 0; j < N; j++)
② {
③     while (written[j] == 1) {}
③     while (pair{want[j], j} <= pair{want[i], i}) {}
② }

④ 访问临界资源;

⑤ want[i] = LLONG_MAX;
```

written[i] 不是一个锁，不保证 serial++ 的互斥，它只是一个标志，用来提醒潜在的等待者该线程还在更新 want[i]，请等待它更新完成并以最新值为基准。

到这里算是解决了多指令流互斥问题。可以证明，对于 n 指令流互斥，需要 O(n) 数量的读才可以上锁（一个直观的理解：因为要感知其它指令流的上锁意图）。

那么这些互斥算法的效率如何？有何替代解决办法？

### 回顾：自私的实现

## 回顾：自私的实现

```
int enter = 0;
```

### 指令流S0

- ① while (enter == 1) {}
- ② enter = 1;
- ③ 访问临界资源;
- ④ enter = 0;

### 指令流S1

- ① while (enter == 1) {}
- ② enter = 1;
- ③ 访问临界资源;
- ④ enter = 0;



**问题一** 上述实现解决了临界区的互斥问题吗？为什么？

没有解决这个问题。指令流在执行时，是会发生上下文切换，而这种切换不受指令流自身控制。

考虑执行序列S0①→S1①→S0②→S1②。两个指令流都检查发现对方未进入，于是同时进入临界区，破坏了互斥条件→竞争冒险（Race）。

要让这种写法工作，检查条件与设置标志必须是原子操作。

**问题二** 如何改进这种方法让临界区能够互斥？

如果我们能利用硬件功能，让①和②之间不发生调度，那也可以。

## 关中断（或锁调度器）实现自旋锁：CLI/STI

**解决方案一** 封禁调度器的一个最简单办法就是关中断。一旦关中断，时钟中断就无法发生（此时若时钟中断到来，需要等待开中断后才发生），自然无法进行指令流或线程切换。

```
int enter = 0;
```

### 指令流S0

- ① while (1) {
- ② CLI();
- ③ if (enter == 0) {
- ③ enter = 1;
- ③ break; }
- ④ STI(); }
- ⑤ 访问临界资源;
- ⑥ enter = 0;

### 指令流S1

- ① while (1) {
- ② CLI();
- ③ if (enter == 0) {
- ③ enter = 1;
- ③ break; }
- ④ STI(); }
- ⑤ 访问临界资源;
- ⑥ enter = 0;

**问题** 处理器都有开关中断指令，为何这种方法仅在某些针对无内核模式的CPU的操作系统上（如FreeRTOS、RT-Thread）广泛使用？

该算法要赋予所有指令流直接开关中断的权限。一旦有恶意指令流，它关中断后不再开启，就可以得到100% CPU，可以轻易破坏线程间的时间隔离。出于信息安全原因，我们不能粗暴地这样做。当然，如果操作系统本身就不是信息安全的（也即不支持进程），那就没有这个顾虑了。

## 原子指令实现自旋锁：SWAP/XCHG

**解决方案一** 我们必须确保指令流关中断后一定会开中断，而这是无法使用分立的开关中断指令做到的。因此，需要保证检查和置位是原子操作；相当于将开关中断和检查/置位指令强行封装在一起。

## 交换指令 Exchange (XCHG) /SWAP

操作类别	目的操作数	源操作数	例子
寄存器与寄存器交换	reg/reg		XCHG AL, CH
寄存器与存储器交换	[mem]/reg		XCHG AX, [BX]
指令操作			影响标志位
临时 ← 目的; 目的 ← 源; 源 ← 临时;			无
交换的两个寄存器的字长必须一样。XCHG默认有锁总线的LOCK前缀，无需另加。			

**问题** 如何使用这条原子指令同时完成条件检查和标志置位？

**提示** 我们检查条件时，不希望其它指令也通过检查。

```
int enter = 0;          void swap(int* addr1, int* addr2)
int current[2];
```

### 指令流S0

```
① current[0] = 1;
① do
① {
①     swap(&current[0], &enter);
① }
② while (current[0] == 1);

⑤ 访问临界资源;

⑥ enter = 0;
```

### 指令流S1

```
① current[1] = 1;
① do
① {
①     swap(&current[1], &enter);
① }
② while (current[1] == 1);

⑤ 访问临界资源;

⑥ enter = 0;
```

**问题** (1) 该程序如何能保持互斥？它要如何推广到多指令流的状态？

(2) 它会死锁吗？会活锁吗？效率如何？

(3) 它能保证公平进入吗？

## 原子指令实现自旋锁：TAS/BTS

**解决方案二** SWAP指令总是需要一个额外的变量。这个变量其实可以省去。

## 比较置位指令Test-And-Set, TAS/Bit Test-And-Test, BTS

操作类别	目的操作数	源操作数	例子
寄存器取立即数位置	reg	imm	LOCK BTS AX, 8
寄存器取寄存器位置	reg	reg	LOCK BTS AX, CX
存储器取立即数位置	[mem]	imm	LOCK BTS [DX], 8
存储器取寄存器位置	[mem]	reg	LOCK BTS [DX], CX
指令操作			影响标志位
CF ← 位(目的, 源); 位(目的, 源) ← 1;			CF
操作涉及的一切寄存器的字长必须一样。与XCHG指令不同，BTS不隐含LOCK前缀，需要手动添加。			



```
int enter = 0;
```

### 指令流S0

- ① while (bts(&enter) == 1) {}
- ② 访问临界资源;
- ③ enter = 0;

### 指令流S1

- ① while (bts(&enter) == 1) {}
- ② 访问临界资源;
- ③ enter = 0;

**问题** (1) 该程序如何能保持互斥? 它要如何推广到多指令流的状态?

(2) 它会死锁吗? 会活锁吗? 效率如何?

(3) 它能保证公平进入吗?

### 原子指令实现自旋锁: CASXCHG/CAS

**解决方案三** XCHG和BTS指令总是需要读写锁变量, 而如果锁已被占用, 这个写内存的操作并无必要。如果能先检测锁是否被占用, 并仅在 确认无占用的情况下写入锁变量, 可节约潜在的内存访问。

### 比较交换指令 Compare-And-Exchange, CMPXCHG/Compare-And-Swap, CAS

操作类别	目的操作数	源操作数	例子
寄存器与寄存器交换	reg	reg	LOCK CMPXCHG DL, CH
寄存器与存储器交换	[mem]	reg	LOCK CMPXCHG [BX], DX
指令操作			影响标志位
临时 ← 目的; 如果 (AL/AX/EAX/RAX == 临时) ZF ← 1; 目的 ← 源; 否则 ZF ← 0; AL/AX/EAX/RAX ← 临时; 结束			ZF
操作涉及的一切寄存器的字长必须一样。与XCHG指令不同，CMPXCHG不隐含LOCK前缀，需要手动添加。			

```
int enter = 0;
```

### 指令流S0

- ① while (cmpxchg(0, 1, &enter) == 1) {}
- ② 访问临界资源;
- ③ enter = 0;

### 指令流S1

- ① while (cmpxchg(0, 1, &enter) == 1) {}
- ② 访问临界资源;
- ③ enter = 0;

**问题一** (1) 该程序如何能保持互斥? 它要如何推广到多指令流的状态?

(2) 它会死锁吗? 会活锁吗? 效率如何?

(3) 它能保证公平进入吗?

**问题二** (1) 上述一切基于硬件原子指令的自旋锁和软件算法实现的自旋锁相比有什么缺点?

(2) 上述两种自旋锁 (或者说一切自旋锁) 有一个最大的缺点, 它是什么?



## 1.3 阻塞锁

### 自旋的缺点

对于自旋锁，在不能获得锁时会频繁检查条件。这（1）无谓地浪费了CPU，（2）造成内存总线压力（尤其是多核系统），（3）在指令流对线程多对一模型中是无法使用的（为什么？），（4）此外由硬件原子指令实现的锁还无法确保公平性。

**阻塞锁** 为了克服自旋锁的弱点，我们需要引入阻塞锁。阻塞锁与自旋锁的区别只有一个，就是**当指令流无法获得锁时就停止执行，并等待锁的释放**。阻塞锁中还可以实现**等待队列**，指令流按照先来后到的顺序阻塞在等待队列中。一旦锁的占用者释放，我们就从队列的头部唤醒一个指令流，从而实现公平性。

**阻塞锁的实现** 阻塞锁可以在线程级别（内核空间）实现，也可以在指令流级别（用户空间）实现。如果是前者，线程上的任何指令流阻塞，线程都陷入内核并阻塞；如果是后者，则整个线程上的所有指令流都阻塞，线程才阻塞。作为操作系统的编写者，我们自然关心前者而非后者，因为前者是运行时库提供的。

**好阻塞锁的标准** 在好自旋锁的标准上增加一条“让权等待”，也即无法获得锁时应立即释放处理机。

### 自旋锁

```
int enter = 0;
```

#### 线程T0

- ① while (cmpxchg(0, 1, &enter) == 0);
- ② 访问临界资源;
- ③ enter = 0;

### 阻塞锁

```
pthread_mutex_t mutex;
```

#### 线程T0

- ① pthread\_mutex\_lock(&mutex);
- ② 访问临界资源;
- ③ pthread\_mutex\_unlock(&mutex);

#### 线程T1

- ① while (cmpxchg(0, 1, &enter) == 0);
- ② 访问临界资源;
- ③ enter = 0;

#### 线程T1

- ① pthread\_mutex\_lock(&mutex);
- ② 访问临界资源;
- ③ pthread\_mutex\_unlock(&mutex);

### 阻塞锁的缺点

如果所得并发度（指一齐竞争的线程数）很少，但并发争用行为很频繁，临界区很短，系统调用以及线程阻塞，切换和唤醒的开销就不可以忽视。

pthread:mutex

**pthread\_mutex** pthread线程库提供的阻塞锁实现。在各个OS平台均可使用。

```
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);
```

线程T0

线程T1

- ① pthread\_mutex\_lock(&mutex);
- ② 访问临界资源;
- ③ pthread\_mutex\_unlock(&mutex);

- ① pthread\_mutex\_lock(&mutex);
- ② 访问临界资源;
- ③ pthread\_mutex\_unlock(&mutex);

```
pthread_mutex_destroy(&mutex);
```

类别	pthread调用	描述
创建与 销毁	pthread_mutex_init	初始化阻塞锁，可选择单进程内或进程间共享
	pthread_mutex_destroy	销毁阻塞锁
加锁	pthread_mutex_lock	试图获取锁，如果获取不成功则阻塞（阻塞式接口）
	pthread_mutex_trylock	试图获取锁，如果获取不成功则立即返回（非阻塞式接口）
解锁	pthread_mutex_unlock	放弃阻塞锁

Linux: futex

**问题** 能否设计一种锁，在**高竞争度下接近阻塞锁**从而防止忙等待，在**低竞争度下则接近自旋锁**从而免去系统调用开销呢？

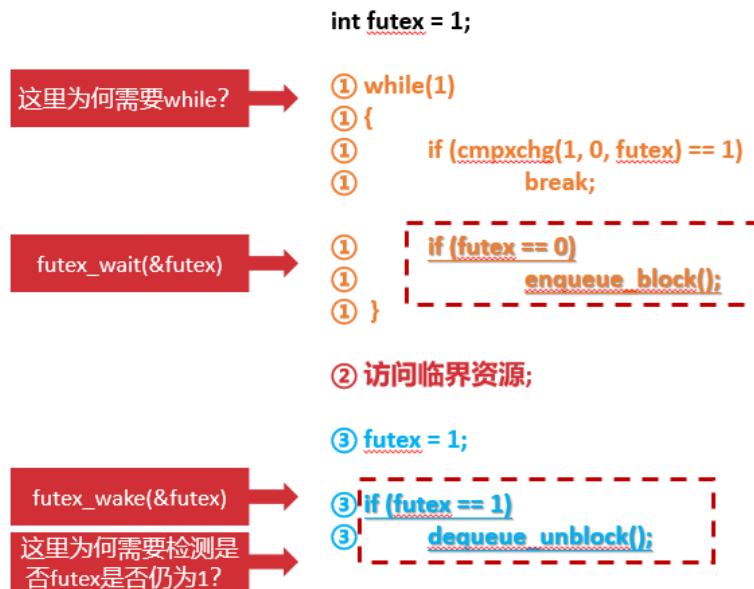
**思考** 这个锁还是要**有阻塞能力**，因此肯定是一个系统级、内核级机制。但要免去系统调用开销，就要让它在**不争用时避免陷入内核**。

用户空间快速阻塞锁 Fast user-level mutex, Futex

Linux提供的一种快速互斥机制。它通过将锁变量（一个整型值）放在用户模式，而阻塞队列放在内核模式，取得了阻塞和自旋 锁间的一个平衡。

pthread\_mutex的实现

实际上，pthread\_mutex在Linux上的实现就是使用futex作为其底层。这样，它比总是陷入内核的阻塞锁性能就好多了。如果两个线程属于不同进程，pthread\_mutex还会提前映射一块共享内存，并把futex变量放在那（后面讲IPC时详讲）。



## Linux: 读写锁

**问题** 对于一个数据结构，往往有读者有写者。如果该锁对读者和写者同等对待，读和写都要取得这把锁，则说明对读和写该区域都是临界区。这样做有什么好处和坏处？

**答案** 对于数据结构，只有读-写与写-写之间不能同时进行。如果是读-读，则该区域并非临界区。完全可以允许多个读者同时读数据构，只要写者在写时候没有读者或其他写者就可以了。

### 读写锁 Read-Write Lock, RW Lock

读者和写者使用的加锁和解锁语义不同：读者可以在没有写者进入临界区时获取锁，而写者只能在没有任何读者或写者时获取锁。多个读者可以同时持有锁，但写者不能和其他读者或写者同时持有锁。

读写锁将读和写分开了，增加了读的并发度，在那些读多写少的场合可提高性能。



```
int num_readers = 0;
mutex_t write = 1, read = 1;
```

## 写者

```
...
lock(&write);

write();

unlock(&write);
...
```

## 读者

```
...
lock(&read);
num_readers++;
if (num_readers == 1)
    lock(&write);
unlock(&read);

read();

lock(&read);
num_readers--;
if (read_count == 0)
    unlock(&write);
unlock(&read);
...
```

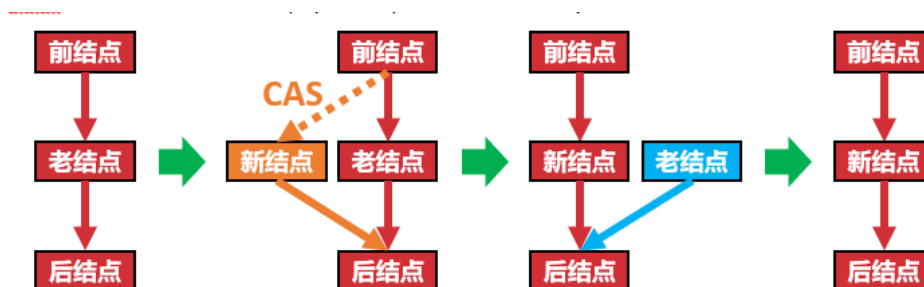
## Linux: “RCU锁”及其它无锁结构

### 读-写-更新 Read-Copy-Update, RCU

在对某个数据结构结点进行修改时，先拷贝一份新结点，然后修改这份结点数据。修改完毕后，使用原子写操作将其插入原位，并替换老结点；老结点则在最后一个潜在读者读完它后才释放。

这意味着，在所有读者看来，新数据是突然出现的，达成了数据更新的原子性。**RCU其实根本就不是锁**，但很多地方为了强调这种方法可以达成锁的效果，称其为RCU锁。

**例子** 以单向链表为例，更改老结点的方法：



**问题一** 如何知道潜在读者什么时候读完老结点？

**提示** 读操作的时间是有限的。

**解决方案** (1) 读者给结点加读锁，当所有读锁被释放时，则知道结点无读者了，可以进行废弃。但这又等于回到加锁的老路上去了。

(2) 读操作的时间是固定的，那就意味着被替换的老结点在这个固定时间之后不可能还有任何读者正在读（为什么？），在这之后老结点就被回收。这个固定时间叫做安定期（Quiescence Period）或宽限期（Grace Period）。

**问题二** 多个写者怎么解决？多个写者之间还是会冲突，每个人都可以添加一个新版本，最后原子写前继结点的指针会冲突。

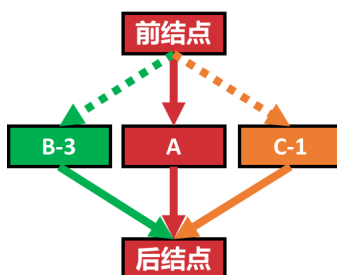
**解决方案** 在更新前继结点指针时使用CMPXCHG（CAS），这样只有一位写者的版本可以提交成功，其他写者的版本会被驳回。因此，其它写者想要再写，就只能重新提取一份最新副本，在上面修改后再次尝试提交了。

**刁钻的问题** 上述问题二的解决方案可能有一个漏洞，是什么？

### ABA问题

无锁数据结构实现中的一个常见问题。老结点被删除后，又被SLAB内存分配器再次分配给其他写者，导致其以新结点的身份被添加到数据结构的同样位置。**因为指针变过去又变回来，正在操作这个位置的其它读者或写者将认为原数据在整个过程中都无变化，但实际却遭到两轮更新。**

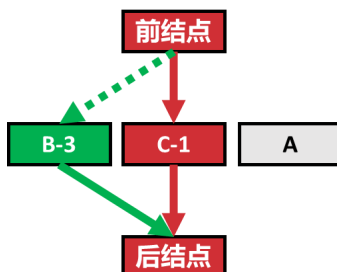
**例子** 考虑如下场景：三个写者竞争更新老结点，每次更新都把自己的线程号追加在原数据后面作为新结点内容。



**步骤一** 3号写者和1号写者同时更新链表的中间结点A。该结点本来是空的，无内容。因此，3号写者为它创建了拷贝B，在B中填写上3；1号写者则为其创建了拷贝C，在C中填写上1。

在这之后，1号写者被调度，3号写者则被搁置一段时间。

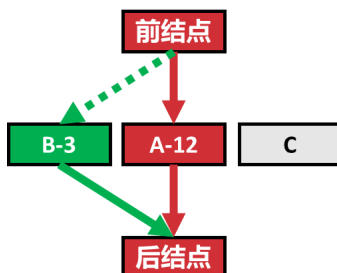
**步骤二** 1号写者完成更新。现在中间结点的地址为C，其内容为1。同时，结点A的内存被释放，回归内存池。



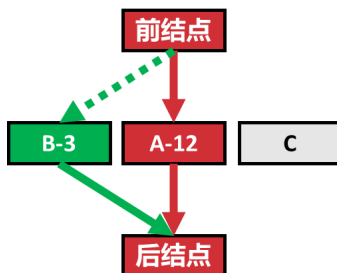
**步骤三** 2号写者开始运行。它从内存池中分配一个结点，内存池恰巧返还了刚刚被释放的A（SLAB等缓存式分配器的分配策略提升了此类问题的概率）。

它将A初始化成C的内容，也即1，然后追加自己的编号。于是现在A的内容被更新成12。

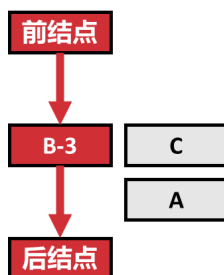
最后，它将A插入链表，并释放节点C



**步骤四** 久被搁置的3号写者终于得到调度。此时，它通过CAS指令检查发现前结点的 指针指向的就是A，于是认为无人更新过A。实际上此A并非彼A，它是被删除掉、释放掉，又被分配到、更新过，重新出现在这个地方的新A。它的内容不再是空，却是12。



**步骤五** 3号写者当然无法得知这一切。于是它的CAS将成功，现在带有错误内容的B结点进入了数据结构。按照之前的约定，此时结点上含的数据应该是123（或者312），现在却直接抹除了1和2这两个数据，相当于发生了意外回滚。



## 如何解决ABA问题

**带时间戳指针** 在每个结点的指针上带上一个时间戳。该时间戳是永恒前进、不会回头的，每次指针被修改，该时间戳便加1。这样，即便发生ABA，前结点的指针也会被更新两次：在第一次指向A时该指针值为  $(A, N)$ ，则在第二次指向A时该指针值为  $(A, N+2)$ 。因为同时更新指针和时间戳，在32位下要使用CMPXCHG8B（64位），在64位下则要使用CMPXCHG16B（128位）。

**问题回顾** 在前面的例子中，3号写者的CAS将失败。因为它假设前结点指针原值为  $(A, N)$ ，而此时该值为  $(A, N+2)$ 。这样，3号写者就明白有人更新过A，将会用更新后的A作为蓝本来追加自己的编号。因此，最终将得到正确内容123。

## 1.4 管程

**问题** 对于不可共享的设备接口，单独暴露其访问接口无意义，因为总是需要加锁。如果有一个指令流忘记加锁（程序员开小差），就乱套了。从编程语言、编译器、运行时库和第三方库设计者的角度，如何避免这个问题？

**管程** 一种语言级或库级别的构造，该构造通过将互斥（或同步）机制与临界资源操作封装在一起并作为对资源的唯一接口导出，保证了任何程序都通过该接口访问临界资源，进而保证了资源的互斥。

管程看起来就像一个对象，该对象的各个成员函数均封装了加锁、访问资源、解锁三个过程。

在某些编程语言中，管程的所有成员函数进入和退出时自动加解锁，所以管程成员函数中lock和unlock也可以不写。比如java，只要写了synchronized关键字，这一切都是编译器搞定。

```

class MyMonitor{
    public synchronized void method(){
        mutex.lock();
        accessResource();
        mutex.unlock();
    }
}

```

## 管程的特点

1. **互斥性** 管程内的临界区被互斥原语（如锁等）保护，只能由一个（或某个有限数量的）指令流进入并执行。
2. **模块化** 管程是可单独编译的程序的基本单元，一般写在一个单独的源文件内。
3. **封装性** 管程封装了对原始临界资源的一切操作，不向外界暴露任何接口。管程内的任何数据只能通过管程的接口访问，同时管程代码也只访问管程内部的数据。
4. **抽象性** 管程抽象掉了原始临界资源的细节属性，仅保留了其高层次的功能接口。

其实只有第一个是管程的本质特征，后三个都是高级编程语言中类的特征。管程还具有其他类的特征，比如构造、析构、继承、多态、组合等。

**问题** 前面讲设备管理时候讲过一个守护进程，它和管程看起来很像。那么它们有什么区别？

## 管程vs守护进程

项目	管程	守护进程
语言级别构造	是	否
独立地址空间	否	是
可被指令流执行	是；指令流亲自执行管程	否；指令流把工作塞给守护进程代为执行
需要缓冲区	不一定	是
可阻塞	是	用于假脱机时一般不阻塞
资源消耗	低	高

**问题一** 从信息安全角度看，谁更胜一筹？

**答案** 守护进程运行在独立的地址空间，垄断了临界资源的I/O。因此，指令流不可能得到机会亲自参与临界资源操作。相比之下，管程只是将敏感操作藏起来，如果是恶意指令流，还是可以直接用函数指针跳转到相应位置强制I/O：因为I/O的代码和这些指令流在同一个地址空间。

当然，恶意指令流还可以选择直接memset毁坏管程数据结构。

**问题二** 信息安全仅仅是空间吗？考虑时间呢？

**时间安全性** 管程虽然在空间上不安全，在时间上却是安全的，因为每个线程执行管程消耗的都是它自己的时间片，发起大量恶意请求只会导致自己的时间预算耗尽。

相比之下，守护进程却不是这样：因为工作都推给了守护进程，因此消耗的是守护进程内的I/O线程的时间片。如果一个恶意线程向守护进程提交大量的垃圾I/O，就可以用掉该I/O线程的所有时间片，使其其他线程的请求得不到执行。

考虑一下在景区，排队排到黄牛，黄牛说给我出1000张票。



## 拒绝服务攻击 Denial-of-Service Attack, DoS

恶意客户端向服务器发送大量数据以饱和其资源，并使其无法将资源分配给正常请求的攻击方法。

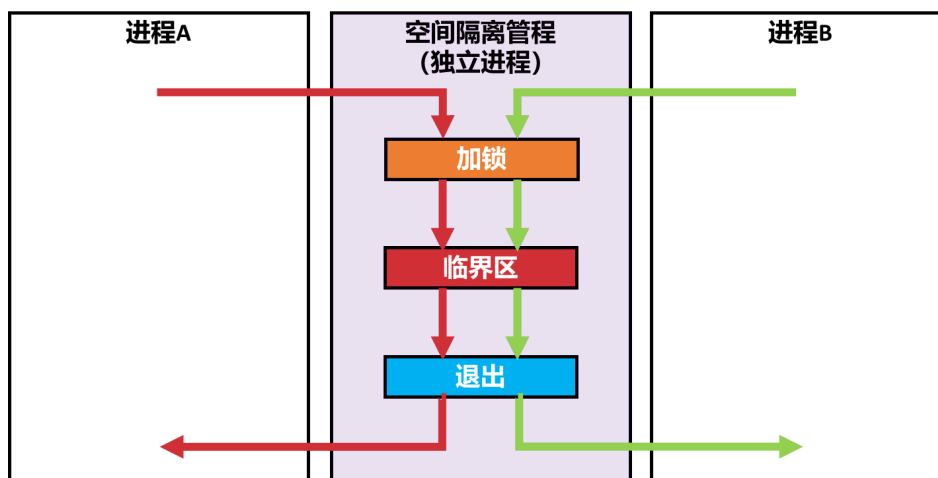
## 分布式拒绝服务攻击 Distributed Denial-of-Service Attack, DDoS

在拒绝服务攻击的基础上，增加恶意客户端的数量，使攻击流量更大、来源更加分散，因此防御也更加困难。

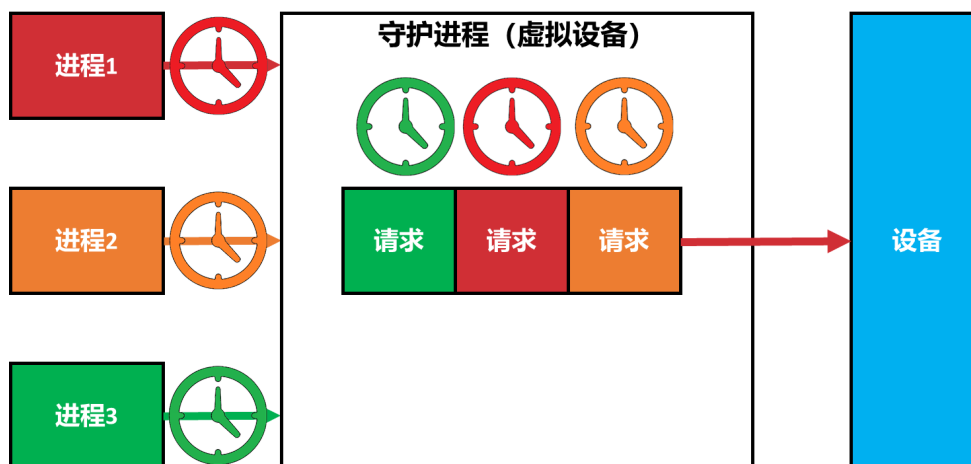
**问题** 如何让管程运行在独立的地址空间，或者让守护进程中的I/O线程使用请求发起者的时间？

### 线程迁移和时间托管：空间隔离管程和时间隔离守护进程

**线程迁移法** 将管程做成一个特殊的进程。该进程内部没有线程，平时也不执行，而是等待其它进程中的线程迁移过来。迁移到管程内部的线程仍是同一个线程、消耗自己的时间片，但此时却执行管程的代码。这是Composite等微内核采取的机制。



**时间授予法 Grant** 要求线程在向守护进程提交I/O请求时，同时转移适量的时间片给守护进程内的I/O线程。I/O线程在执行此请求时，会拿着对应的时间片来执行。如果在这些时间片耗尽之前还没执行完，则放弃该请求并返回错误。这是seL4等微内核采取的机制。



经典问题：哲学家就餐

## 哲学家就餐 Dining Philosophers Problem

一个非常著名的互斥问题。

餐桌上有五个哲学家，仅做两件事：要么吃饭，要么思考。思考需要一定的时间，吃饭也是。思考是无条件的，但吃饭不是。

### 吃饭的条件

- (1) 必须同时拿起两根筷子
- (2) 筷子是互斥共享的
- (3) 米饭是同时共享的

**问题** 如何设计正确的互斥机制，使哲学家都能吃上饭？

### 提示

- (1) 用锁解决
- (2) 用管程解决

**用锁解决** 既然筷子是互斥共享资源，我们就给每个筷子上一把锁。

```
mutex_t chopstick[5] = {0};

第i号哲学家
while(1)
{
    think();

    lock(&chopstick[i%5]);
    lock(&chopstick[(i+1)%5]);

    eat();

    unlock(&chopstick[(i+1)%5]);
    unlock(&chopstick[i%5]);
}
```

**答案** 互斥确实做到了，可惜会死锁。如果所有哲学家都同时拿起左手边的筷子的话... ..

**解决方案** 那就让所有的筷子操作使用同一个锁。这个实现怎么样？

```
mutex_t table = 0;

第i号哲学家
while(1)
{
    think();

    lock(&table);

    pickup(i, (i+1)%5);
    eat();
    putdown(i, (i+1)%5);

    unlock(&table);
}
```

不仅没问题，**甚至还挺公平**，能保证所有的哲学家按照饿肚子的顺序来吃，**不会把某个哲学家饿死**。

**改进** 吃饭的过程和放下筷子的过程可以不包括在临界区里，只有拿起筷子的操作需要在临界区里。这个实现怎么样？

```
mutex_t table = 0;

第i号哲学家
while(1)
{
    think();

    do
    {
        lock(&table);
        result = trypickup(i, (i+1)%5);
        unlock(&table);

    }
    while(result == 0);

    eat();
    putdown(i, (i+1)%5);
}
```

这个实现的并发度拉满了，但（1）现在可能把某个哲学家饿死；而且（2）任何一个哲学家一旦挨饿就会疯狂尝试拿起叉子。如何两全其美？

**管程实现** 把代码中eat之前的操作封装成prepare(), eat之后的操作封装成 cleanup(), 就是一个管程了。然而，封装成管程并不能解决上面代码中的问题，该是什么样还是什么样。管程毕竟只是一张皮。

```
void monitor::prepare(int i)
{
    int result;
    do
    {
        lock(&mutex);
        result = trypickup(i, (i+1)%5);
        unlock(&mutex);
    }
    while(result == 0);
}

void monitor::cleanup(int i)
{
    putdown(i, (i+1)%5);
}

class monitor* table = new monitor();

第i号哲学家
while(1)
{
    think();
    table.prepare(i);
    eat();
    table.cleanup(i);
}
```

## 其它知识

### 无等待数据结构 Wait-free

无锁数据结构只是表示不需要锁，但并不能保证操作不会失败。也即无锁数据结构仅保证全局进展而不保证局部进展，也不一定保证公平性。

在无锁数据结构的基础上，无等待数据结构进一步保障了所有的操作都必然成功，也即保障了局部进展。当然，这需要更精妙的设计和更多的处理器特性，如原子加载自增（Fetch-And-Add, FAA; x86 LOCK XADDL）指令，而且适用的数据结构更少。

从数学上可以证明，一切数据结构都可以做成无锁甚至无等待的。但无锁尤其是无等待算法并未得到推广呢？请参见课后习题。

### 原子加载自增 Exchange-And-Add, XADD/Fetch-And-Add, FAA

操作类别	目的操作数	源操作数	例子
寄存器到寄存器	reg	reg	LOCK XADD AL, CH
存储器到寄存器	[mem]	reg	LOCK XADD [DX], AX
指令操作			影响标志位
临时 ← 源 + 目的; 源 ← 目的; 目的 ← 临时;			CF, PF, AF, SF, ZF, OF
XADD不隐含LOCK前缀，需要手动添加。			

### 链接读和条件写 Load-Linked (LL) /Store-Conditional (SC)

很多处理器如ARM等没有CMPXCHG和XADD，但提供了LL和SC指令。

### 使用LL和SC实现CMPXCHG和XADD

```

int cmpxchg(int old, int new, int* addr)
{
    int temp;
    int result = 0;

    do
    {
        temp = ll(addr);
        if (old == temp)
            result = sc(new, addr);
        else
            break;
    }
    while (result == 1);

    return temp;
}

int faa(int inc, int* addr)
{
    int old;
    int new;
    int result = 0;

    do
    {
        old = ll(addr);
        new = old + inc;
        result = sc(new, addr);
    }
    while (result == 1);

    return old;
}

```

### 内存访问序 Memory Ordering

在现代多核并发处理器设计中，为了最大限度地释放微架构和内存控制器的潜能，不同处理器核心看到的内存访问的先后顺序是不一样的。现代处理器只保证每颗核心看到的自己的内存访问顺序是合乎程序要求的（否则程序的正确性就没有了），但并不保证其它核心看到的内存访问顺序和自己一致，而且每颗核心看到的内存访问顺序都可以不一致。因此，在其它核心看来，本核心的程序执行仿佛出现颠倒：比如一个后写入的变量先被更新到内存，而一个先写入的变量则后出现。

在这堂课中，我们讨论的处理器的一致性模型都是线性一致性模型（Linearizable Model），也即对所有处理器，所有操作的总执行顺序是相同的，且和它们在各个处理器上的发起时间顺序完全一致。它是最强的一致性模型，适合用来学习互斥与同步。

### 内存屏障 Memory Fencing

为克服这一问题，我们需要内存屏障（MFENCE/DMB）：它是一种特殊的指令，能够保证它之前的一切内存访问都提交给内存，而它之后的内存访问在它执行完毕前不会开始。它就像流水线中的一个大气泡，暂时清空流水线，直到前面的指令全部执行完毕（退休，Retire）。

## 二.指令流间的合作

## 2.1条件变量

我们需要一种同步措施：能够

- 让指令流阻塞在临界区，
- 能够在阻塞时即时释放锁
- 能够在满足某些条件时解除阻塞并得到锁

### 条件变量

一种同步原语，可以使**指令流阻塞并等待，直到某个条件发生**。它总是与一个锁配合使用：如果条件不满足，则指令流自动释放锁并加入等待队列；而当条件满足时，等待队列头部的指令流将被唤醒并自动恢复对锁的持有。

### 条件变量的基本操作

1. 阻塞，等待（Wait）：一个线程在条件变量上等待，直到某个条件满足为止。等待操作会自动释放相应的互斥锁，使其他线程能够访问共享资源。线程在等待时处于阻塞状态，直到被唤醒。
2. 唤醒，唤醒（Signal/Notify）：当某个线程改变了条件变量所表示的条件，使得其他线程可能满足等待条件时，它可以通过唤醒操作通知等待的线程。被唤醒的线程将从等待状态转换为就绪状态，然后等待获取互斥锁来访问共享资源。
3. 唤醒全部,wakeup all 唤醒等待队列中所有的指令流，并使其中随机一个获得锁，其他的无法获得锁的指令流继续因为得不到锁而阻塞。此时这些阻塞的指令流并非阻塞在条件变量上，而是阻塞在锁上；一旦那个当前持有锁的指令流释放锁，那些阻塞在锁上的指令流中的其中一个就会立即获取锁而继续执行，直到所有的指令流都获取过一遍锁。

### pthread:cond

pthread线程库提供的条件变量实现。

类别	pthread调用	描述
创建与 销毁	pthread_cond_init	初始化条件变量，可选择单进程内或进程间共享
	pthread_cond_destroy	销毁条件变量
等待	pthread_cond_wait	等待条件变量（阻塞式接口）
	pthread_cond_timedwait	等待条件变量，带超时（阻塞式接口）
唤醒	pthread_cond_signal	唤醒在条件变量上等待的一个线程
	pthread_cond_broadcast	唤醒在条件变量上等待的一切线程

```
pthread_cond_t cond;  
pthread_mutex_t mutex;  
pthread_cond_init(&cond, NULL);  
pthread_mutex_init(&mutex, NULL);
```

```
pthread_cond_t cond;
pthread_mutex_t mutex;
pthread_cond_init(&cond, NULL);
pthread_mutex_init(&mutex, NULL);
```

### 线程T0

```
① pthread_mutex_lock(&mutex);
① while(!condition_met)
    pthread_cond_wait(&cond, &mutex);
② 访问临界资源;
③ pthread_cond_signal(&cond);
③ pthread_mutex_unlock(&mutex);
```

```
pthread_cond_destroy(&cond);
pthread_mutex_destroy(&mutex);
```

### 线程T1

```
① pthread_mutex_lock(&mutex);
① while(!condition_met)
    pthread_cond_wait(&cond, &mutex);
② 访问临界资源;
③ pthread_cond_signal(&cond);
③ pthread_mutex_unlock(&mutex);
```

## 惊群效应

惊群效应（英文：Thundering Herd）是指在并发编程中的一种现象，当多个进程或线程等待同一个事件或资源时，一旦该事件或资源可用，所有等待的进程或线程会同时被唤醒，但只有其中一个能够获得资源或执行任务，其他进程或线程则需要重新等待。这种情况会导致系统资源浪费和性能下降。

为了避免惊群效应，可以采取以下策略：

1. 互斥锁（Mutex）：在条件变量之前使用互斥锁来确保只有一个线程能够获得资源或执行任务。当条件变量满足时，只有获得互斥锁的线程可以进入临界区，其他线程则需要等待互斥锁。
2. 唤醒一个线程：只唤醒一个等待的线程，而不是全部唤醒。通过选择合适的等待队列管理策略，只唤醒一个线程来处理满足条件的情况，其他线程继续等待。

## 条件变量的两个语义

### 1. Hoare语义：

- 在Hoare语义中，等待线程在收到信号并被唤醒后，必须重新获取与条件变量相关联的互斥锁才能继续执行。
- 当等待线程被唤醒后，它会尝试获取互斥锁，只有成功获取互斥锁的线程才能继续执行。如果线程无法获取互斥锁，它将继续等待，直到能够获取到为止。

### 2. Mesa语义：

- 在Mesa语义中，等待线程在收到信号并被唤醒后，不需要重新获取互斥锁就可以继续执行。
- 当等待线程被唤醒后，它会尝试继续执行，而不必等待互斥锁的所有权。如果线程发现条件不再满足，它将重新等待条件。

## 经典问题：生产者-消费者

在生产者和消费者之间，存在一个缓冲区，生产者像缓冲区填充项目，消费者则从缓冲区中拿出项目。缓冲区的长度使给定的N

## 单生产者-单消费者问题SPSC

不考虑队列长度

```
mutex_t queue = 0;
int length = 0;
condition_t touch = COND_INIT;
```



### 生产者

```
lock(&queue);

length++;
enqueue(item);
cond_signal(&touch, &queue);

unlock(&queue);
```

### 消费者

```
lock(&queue);

if (length == 0)
    cond_wait(&touch, &queue);
length--;

item = dequeue();

unlock(&queue);
```

## 单生产者-多消费者问题SPMC

```
mutex_t queue = 0;
int length = 0;
condition_t touch = COND_INIT;
```



### 生产者

```
lock(&queue);

length++;
enqueue(item);
cond_signal(&touch, &queue);

unlock(&queue);
```

### 消费者

```
lock(&queue);

while (length == 0)
    cond_wait(&touch, &queue);
length--;

item = dequeue();

unlock(&queue);
```

## 多生产者-多消费者问题 (MPMC)

代码和SPMC一样

## 单生产者-单消费者SPSC

现在考虑队列的长度为N。修改什么？

### 生产者

```
lock(&queue);

if (length == N)
    cond_wait(&touch, &queue);
length++;

enqueue(item);
cond_signal(&touch, &queue);

unlock(&queue);
```

### 消费者

```
lock(&queue);

length--;

item = dequeue();
cond_signal(&touch, &queue);

unlock(&queue);
```

看上去超简单，者地方加一个判断就可以了。

## 多-单/单-多/多-多问题 (MPSC/SPMC)

现在考虑队列的长度为N。要做什么修改？



### 生产者

```
lock(&queue);

while (length == N)
    cond_wait(&touch, &queue);
length++;

enqueue(item);
cond_signal(&touch, &queue);

unlock(&queue);
```

### 消费者

```
lock(&queue);

while (length == 0)
    cond_wait(&touch, &queue);
length--;

item = dequeue();
cond_signal(&touch, &queue);

unlock(&queue);
```

如何避免惊群效应

### 生产者

```
lock(&queue);

while (length == N)
    cond_wait(&full, &queue);
length++;

enqueue(item);
cond_signal(&empty, &queue);

unlock(&queue);
```

### 消费者

```
lock(&queue);

while (length == 0)
    cond_wait(&empty, &queue);
length--;

item = dequeue();
cond_signal(&full, &queue);

unlock(&queue);
```

**练习** 将上述生产者消费者问题重新封装成管程。

**提示** 管程的类定义、构造、析构、发送和接收。

```
class msgqueue
```

```
{
```

```
    mutex_t queue;
```

```
    int max, length;
```

```
    int* buffer;
```

```
    condition_t full, empty;
```

```
/* void */ msgqueue(int limit);
```

```
/* void */ ~msgqueue(void);
```

```
void send(int item);
```

```
int rcv(void);
```

```
}
```

2.2信号量

信号量（Semaphore）是一种并发编程中用于线程同步和互斥的机制。它是一个计数器对象，用于控制对共享资源的访问。

信号量主要有两个操作：

- 1. P（wait）Acquire操作：当线程需要访问共享资源时，它必须先执行P操作。如果信号量的计数器大于0，则线程可以继续执行并将计数器减1；如果计数器为0，则线程将被阻塞，直到有其他线程释放资源。
- 2. V（signal）Release操作：当线程使用完共享资源时，它必须执行V操作来释放资源。V操作会将信号量的计数器加1，并且如果有线程在等待该信号量，则唤醒其中一个等待线程。

信号量的作用是通过控制对共享资源的访问来实现线程间的同步和互斥。它可以用来解决生产者-消费者问题、读者-写者问题和多线程资源争用等并发编程中的典型场景。

在使用信号量时，需要合理地设置初始值和选择适当的操作顺序，以确保正确的同步和互斥行为。过多或过少地使用信号量可能导致死锁、饥饿或资源浪费等问题。

需要注意的是，信号量并不保证公平性，即它不能保证等待时间最长的线程先获得资源。如果需要公平性，可以使用更高级的同步机制，如条件变量和互斥锁的组合。

sem Linux内核（不属于pthread线程库）提供的信号量实现。

```
sem_t semaphore;  
sem_init(&spin,0, 0);
```

线程T0

- ① 发送数据;
- ② sem\_post(&semaphore);

线程T1

- ① sem\_wait(&semaphore);
- ② 接收数据;

```
sem_destroy(&semaphore);
```

利用信号量完成生产者消费者问题

多-单/单-多/多-多问题（MPSC/SPMC）


```
mutex_t queue = 0;  
semaphore_t capacity = N, count = 0;
```

生产者

```
sem_acquire(&capacity);  
  
lock(&queue);  
enqueue(item);  
unlock(&queue);  
  
sem_release(&count);
```

消费者

```
sem_acquire(&count);  
  
lock(&queue);  
item = dequeue();  
unlock(&queue);  
  
sem_release(&capacity);
```



哪怕考虑队列长度，也比条件变量看起来简洁多了。

这里的锁仅仅是用来保护不能并发的队列本身。  
如果队列是可并发的，那么这把锁也可以省掉。

对比互斥锁，条件变量，信号量

项目	互斥锁	条件变量	信号量
----	-----	------	-----

项目	互斥锁	条件变量	信号量
主要作用	临界区互斥	基于任意条件的同步	基于资源数量的同步
使用方法	同一个指令流内成对的lock()和unlock()	与锁配对使用，但对使用场景无要求	生产者负责release()，消费者负责acquire()
复杂程度	低	中等	高
唤醒丢失	-	可能丢失	基于计数，不会丢失
可替代性	?	?	?

问：信号量能否代替互斥锁

答：能，只要将信号量的数量初始化为1，然后用PV在临界区前后调用即可

问：信号量能否代替条件变量

答：大多情况下可能代替，而且信号量带计数，不会丢失唤醒，很多时候无需额外加锁

问：什么场景信号量不能代替条件变量

答：使用到全部唤醒（cond\_signal\_all）的场合，或者非标准生产者-消费者场景的场合。

## 吸烟者问题

吸烟者问题（Smokers Problem）是经典的并发编程问题，描述了三个吸烟者和一个供应者之间的协作。

问题描述如下：有三个吸烟者，分别需要烟草、纸和火柴三种资源才能吸烟。还有一个供应者，他持续地提供两种资源给吸烟者。每个吸烟者只有在拥有自己所需的两种资源时才能吸烟。

具体的要求如下：

1. 烟草吸烟者拥有纸和火柴资源。
2. 纸吸烟者拥有烟草和火柴资源。
3. 火柴吸烟者拥有烟草和纸资源。
4. 供应者循环提供两种资源中的一种，供应者选择的资源不能与任何一个吸烟者所需的资源相同。

问题的目标是设计一个同步机制，使得吸烟者能够按照自己所需的资源来顺利地吸烟，而供应者在供应资源时能够正确地满足吸烟者的需求。

解决吸烟者问题的一种常见方法是使用信号量和互斥锁的组合。可以使用三个信号量来表示烟草、纸和火柴的可用性，一个信号量用于控制供应者的行为。此外，还可以使用互斥锁来确保每个吸烟者独占所需的两种资源。

以下是一种可能的解决方案：

1. 创建三个信号量：tobacco（烟草）、paper（纸）和match（火柴）。
2. 创建一个互斥锁供应者Mutex。
3. 供应者循环执行以下操作：

- 获取供应者Mutex。
- 随机选择两种资源之一。
- 释放对应的信号量（例如，如果选择了烟草和纸，则释放tobacco和paper信号量）。
- 释放供应者Mutex。

4. 每个吸烟者循环执行以下操作：

- 获取自己所需的两种资源的信号量（例如，烟草吸烟者获取paper和match信号量）。
- 吸烟。
- 释放所需资源的信号量。

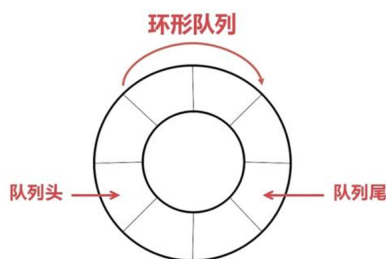
这种解决方案通过信号量和互斥锁的合理运用，确保了吸烟者只有在拥有所需的两种资源时才能吸烟，供应者也能正确地提供所需的资源。

## 无等待数据结构

无锁数据结构能保证全局进展（Global Progress），也即在产生冲突时至少有一个指令流能够完成操作；无等待数据结构更进一步，保证在产生冲突时所有指令流都能轮流完成操作，从而保证了局部进展（Local Progress；公平性）。

## 无等待环形缓冲区

无等待数据结构的典型例子就是高并发环形缓冲区。虽然它很简单，但在网络封包处理等高性能场合用途广泛，因此其高效实现很重要



在这种环形缓冲区的实现中，由于数据流量很高，因此我们不考虑阻塞，只考虑忙等待。

# 三.死锁和活锁

## 3.1 预防和避免

系统资源的使用步骤

申请——>分配——>使用——>释放——>回收

### 死锁

系统中有指令流发生了无法自行解除的循环等待。

只要之中循环等待的发生，则系统发生死锁，

死锁的条件

- 1. 互斥条件，系统中存在有一定互斥性的资源
- 2. 持有条件，指令流已经获得的资源无法释放。**包括保持请求，无法剥夺**
- 3. 循环等待，指令流互相循环等待资源释放不能进展

活锁

指令流并未死锁，但其在多次反复尝试获取资源均失败，无法进展或者进展缓慢。

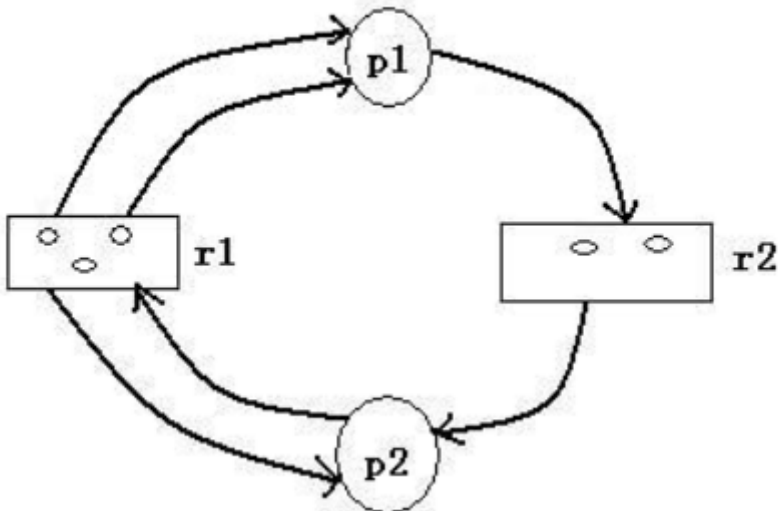
预防和避免

无阻碍数据结构 Obstruction-Free Data Structure

不保证全局进展的无锁数据结构。它是所有非阻塞数据结构中 保障最弱的：系统可以活锁，所有指令流可能反复重试并反复 失败，连一个有进展的都找不出。

项目	无死锁	全局进展	局部进展
无等待数据结构	是	是	是
无锁数据结构	是	是	否
无阻碍数据结构	是	否	否

资源分配图



不安全=死锁？

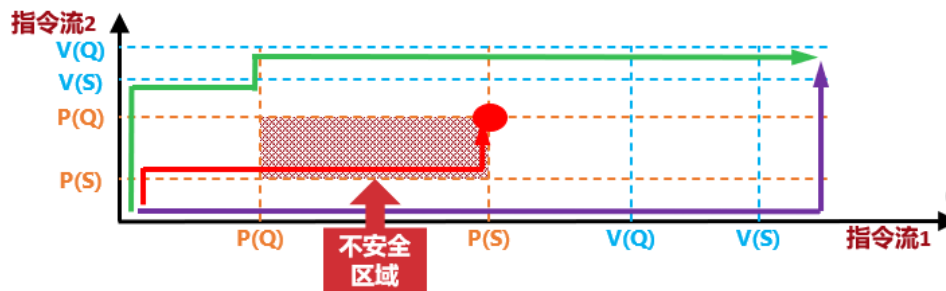
假设所有指令流都是自私的，在得到自己可能请求的全部资源之前绝不会释放已经占有的资源（只进不出），而是要**等到结束后才一并释放**，那么一旦系统执行进入不安全区域，就等于死锁。

如果指令流在执行中可能释放已占有的资源，那么即便**进入不安全区域也不一定死锁**。

**问题四** 我们现在已经能检测死锁是否发生，但这是事后诸葛亮。如何在程序运行避免死锁呢？

**观察** 即便程序本身有可能死锁，最终的结果也不一定是死锁。当执行流进入不安全区域（交叉分配）时，才有可能死锁；

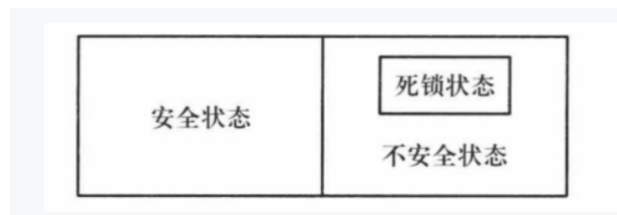
只有在死锁点上（交叉请求），环路才完成，程序才会真正死锁。



**结论** 控制程序的进度，避免进入不安全区域

**问题五** 根据以上的分析，我们知道，只要在程序运行时实时检测资源分配和请求情况，并且控制程序的运行进度就可以了。具体使用什么算法呢？

**提示** 每次分配资源时，查看此分配是否可能让系统进入不安全状态，也即出现潜在的死锁。如果确实可能出现死锁，则拒绝此次分配。



## 银行家算法

### 银行家算法 Banker's Algorithm

一种用于资源分配的算法，由Dijkstra在1965年提出。在已知每个指令流对资源的最大消耗时，它可以通过检查分配请求并仅批准那些不会让系统进入不安全状态的请求，完全避免死锁风险。

在现实生活中，银行业的核心竞争力就是风险控制。这里我们把资源总量看作银行的资金池，将指令流看作是诚实的贷款客户（贷后必还）。

假设系统中有S个指令流和R种资源，则可用如下矩阵表示资源等待图：

#### 可用资源矩阵A Available

1xR矩阵，其A[i]代表了第i种资源的当前可用总量。

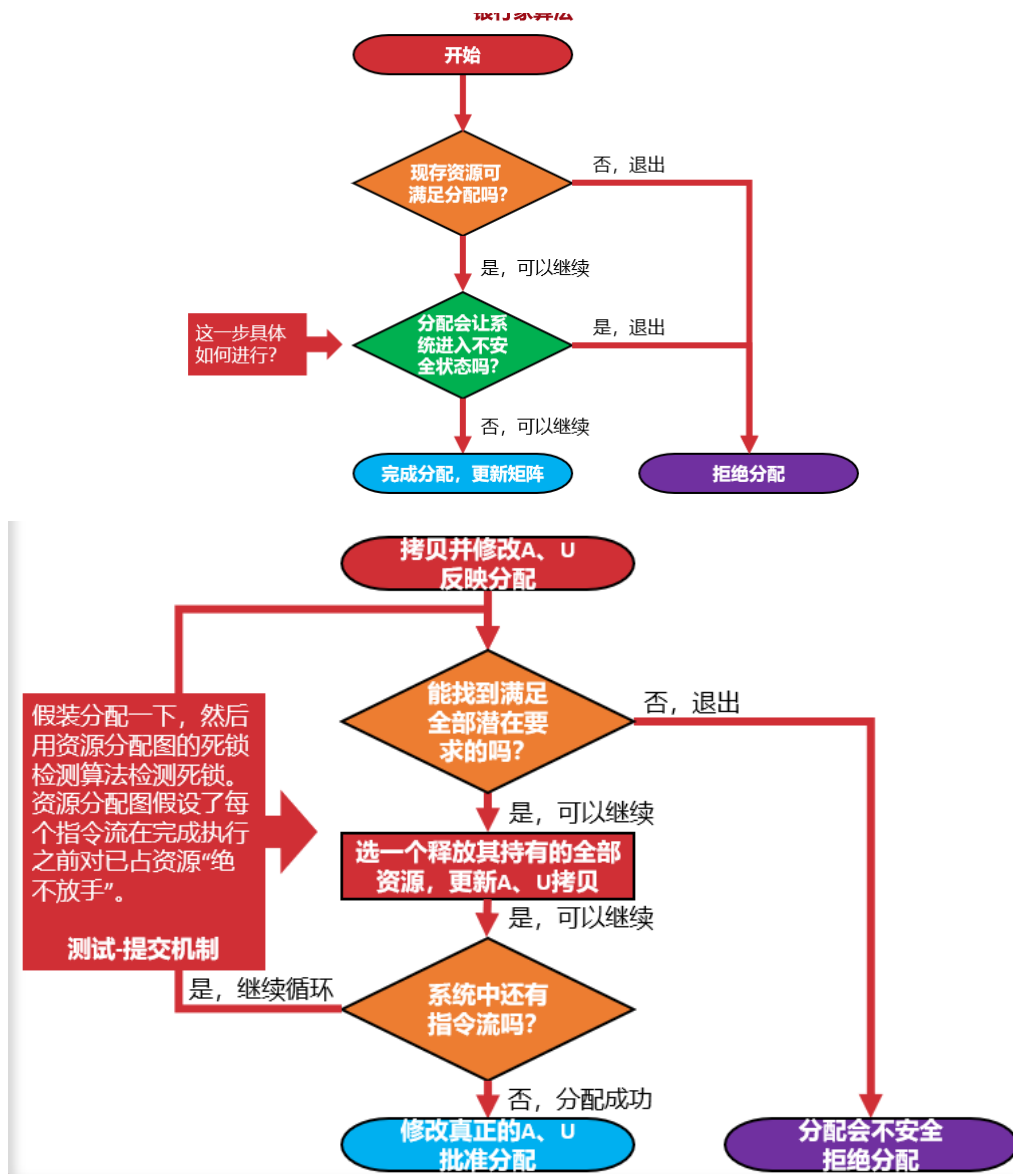
#### 分配资源矩阵U Use

SxR矩阵，其U[i, j]代表了第i个指令流占有的第j种资源的量。

#### 最大资源矩阵M Max

SxR矩阵，其M[i, j]代表了第i个指令流最多需要的第j种资源的量。

**问题** 算法应该是怎样的？回顾资源等待图的分析方法。



### 问题一 银行家算法的开销如何？

银行家算法的开销很大，因为每次进行资源分配都需要对所有的运行该算法。假设系统中有N个进程和M个资源，每次进行指令流试探，都平均要访问N\*M大小的矩阵，而这种试探需要进行N次才能决定全部指令流能否完成执行。如果一共进行A次资源分配，则时间复杂度为 $O(AN^2M)$ 。

### 问题二 银行家算法是离线算法，还是在线算法？

银行家算法对分配请求是在线算法，但对最大资源用量却是离线算法。

### 问题三 避免死锁，是操作系统的责任，还是程序员的责任？

避免死锁，说到底还是程序员的责任。操作系统的责任是提供同步工具。

### 问题四 在满足银行家算法的使用先决条件（各指令流最大资源消耗已知，等等）的现实系统中，在遇到意外故障时，单凭银行家算法能保证系统不死锁吗？

如果有指令流意外终止或者因为BUG进入死循环，资源永远得不到释放的话...



## 结论

银行家算法仅适用于**对程序最大用量知根知底、程序不会出现意外或故障，以及不在乎执行时间开销的场合**。除了非常特别的系统之外，这些条件几乎不可能被满足，尤其是对于通用操作系统；我们并不知道程序的最大资源用量，也不能保证程序没有错误，另外我们对操作系统的资源开销非常敏感。

## 3.2 检测和恢复

**问题** 那么，在现实系统中，遇到死锁怎么处理呢？

**鸵鸟政策** 操作系统仅提供同步工具，不负责探测和解决死锁，而是由人 或其它系统服务在死锁时介入。这么做是有道理的：会死锁的 程序有正确性问题，而保证正确性归根结底是程序员的责任。

### 死锁的检测

#### 问题

(1) 程序员可以用什么方法来探测死锁？

(2) 探测到之后，如何从死锁中恢复？

**答案** (1) 检测程序的进展。如果一个或者一批程序很长时间没有进展，我们就认为它发生了死锁。当然，也有可能是发生了活锁 或者其他BUG，但我们不管那么多：只要程序毫无理由地没有进展，即便不是死锁，也肯定是正确性有问题。

### 死锁的恢复：破坏互斥条件

**破坏互斥条件** 只要想办法让资源本身不互斥、不竞争，就自然不存在等待问题，就不可能死锁。

**问题** 具体可以用哪些方案？

**增加总量** 只要资源够多，就不存在共享，不存在共享就不存在等待。如果打印机不够用，就多买几台。只要有无限多的钱，就有无限多的打印机。

**增加并发度** 将那些不能并发访问的资源升级成能并发访问的资源。如将打印机升级成PDF虚拟文档打印机，采取无纸化办公，自然不存在等待问题。

**假脱机共享** 启动一个守护进程，使其垄断资源的I/O，其他指令流请求它完成任务即可，不再分别申请对该资源的独占。打印者提交文档到队列，由打印服务统筹协调完成打印。

**问题** 这些方案的适用性怎么样

互斥性是一个固有属性，在不增加资源的情况下没那么好改变，况且增加资源是花钱的。假脱机共享又不适用于所有设备。

### 死锁的恢复：破坏保持请求

**破坏保持请求** 要求指令流在无法进展时主动放弃之前获得的资源。

**失败则退出** 在内存等资源分配中是相当常见。

一旦分配失败，直接exit(-1)退出。

**失败则释放** 如果在某个步骤，申请资源失败，就将之前申请的全部资源都释放掉。这是goto机制大放异彩的时刻：

Linux中有大量此种代码。

**不允许占有** 如果程序需要占有资源B，那么在这之前它必须释放资源A。这就从 根本上杜绝了拿着一个资源等待另一个资源的情况。

**一次性获得** 程序必须一次分配一切资源，不允许分步骤逐渐申请。

**问题** 这些方案的适用性怎么样？

需要程序员的介入。如果程序员水平不行，或者团队协作时缺乏编程纪律，这些方法不可能行得通。此外，“一次性获得”方案还会使饥饿的可能性增加。

## 死锁的恢复：破坏无法剥夺

**破坏无法剥夺** 一旦探测到有可能死锁，就强制剥夺死锁参与者的全部资源。

**死锁则挂起** 如果检测到死锁，则暂时剥夺某些程序的资源并将其挂起，待 日后有资源时再回来分配给它们。如果资源的状态很容易恢复 到抢占之前（比如内存的页面交换机制），则可使用此方法。

**死锁则杀死** 在微服务等分布式系统设计中普遍。一旦负责检测系统响 应性的服务检测到系统失去响应，就杀死一些微服务，如果系 统还不响应就继续杀死更多直到系统可以响应。

## 混沌工程学 Chaos Engineering

在“死锁则杀死”的基础上，我们可以走得更远：在系统正常运行时就定期随机挑选一些微服务杀死，这样就连检测死锁都不需要。对于程序员水平参差不齐、代码库体量巨大的大型分布式项目，如果系统在随机杀死任何微服务的情况下都能运行，则说明系统不仅抗死锁，而且还容灾容错，一举解决了所有问题。“既然最后总是一团糟，那就还不如在一开始就让它一团糟。”

对于大型分布式系统，这是主流解决方案，因为它对程序在边界情况下的正确性不做任何假设，也就不要求程序员必须按照某些范式来写程序了，甚至程序有严重的BUG也无所谓。Netflix的微服务系统就是这样搭建的，其中Chaos Monkey负责随机杀死微服务来定期制造一点小小的混乱。

## 死锁的恢复：破坏循环等待

**破坏循环等待** 阻止系统中生成可能的等待环路。

**问题一** 在真实系统中，我们需要维持一个资源分配图，并实时查询它是否有环路吗？

**资源编号法** 将系统中的一切资源排序和编号。在申请资源时，只能按序号 递增的顺序申请资源。

**问题二** （1）如何证明这个方法是正确的？

（2）对资源的释放顺序有要求吗？

**答案** （1）反证法。假设存在死锁，则存在环路。然而资源序号总是递增的，这不可能在环路中成立。

（2）对释放顺序无要求，因为死锁是持有+循环等待导致的，释放本身不是死锁的成因。