Lecture10 Synchronization

1. 多线程问题：竞争条件

操作系统在多处理器多核环境下面临的问题

正确性保证

- 对共享资源的竞争导致错误
- 操作系统提供**同步原语**供开发者使用
- 使用同步原语带来新的问题

性能保证

- 多核多处理器硬件与特性
- 可扩展性问题导致性能断崖
- 系统软件设计如何利用硬件特性

2. 四个场景与对应的同步原语

> *同步原语(Synchronization Primitive)*

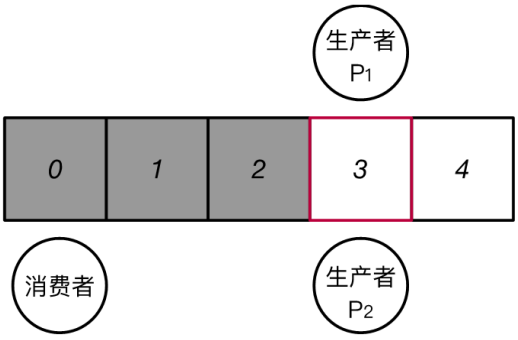
场景一：共享资源互斥访问

> 多个线程需要同时访问同一共享数据，应用程序需要保证**互斥访问**避免数据竞争

```
int shared_var = 0;
void thread_1(void) {
  shared_var = shared_var + 1;
}

void thread_2(void) {
  shared_var = shared_var - 1;
}
```

使用**互斥锁**保证**互斥访问**



回顾：多生产者之间协同

衍生场景一：读写场景并行读取

> 多个线程**只会读取**共享数据，允许读者线程**并发执行**
>
> 区别在于：Reader可以并发执行，Writer必须互斥且执行过程中不能有read操作

```
int shared_var;
void reader(void) {
  local_var = shared_var;
}

void writer(void) {
  shared_var = shared_var++;
}
```

可使用**读写锁**提升读者并行度

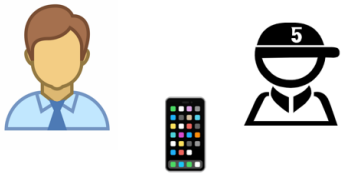
回顾：公告栏问题

场景二：条件等待与唤醒

> 线程等待某条件时**睡眠**，达成该条件之后**唤醒**
>
> 解决了spinlock空转占用计算资源的问题

```
void thread_1(void) {
  doing_something; /* 完成当前线程的工作 */
  notify_thread_2; /* 通知线程2完成 */
}

void thread_2(void) {
  if (thread_1_not_finish)
    wait;        /* 等待线程1完成其工作 */
  doing_something; /* 完成线程2的工作 */
}
```



使用**条件变量**完成线程睡眠/唤醒

回顾：给快递员手机

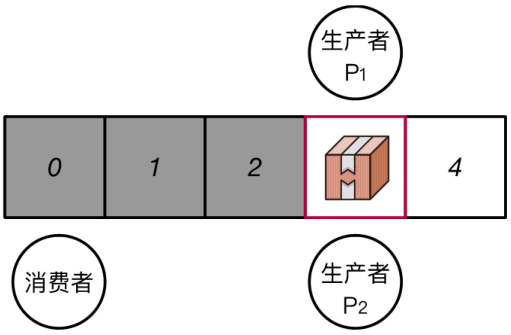
场景三：多资源协调管理

> 多个资源可以被多个线程**消耗或者释放**，正确协同线程获取资源或支持
>
> 实际上就是Producer-Consumer的问题，相比于衍生场景一，差别在于可以有**多个producer**

```
void producer_thread(void) {
      release_resource(shared_resources);
      notify_waiters;
}

void consumer_thread(void) {
 if (not_have_resources)
   wait;
 consume_resource(resource);
}
```

使用**信号量**完成资源管理与线程协同



回顾：生产者消费者之间协同

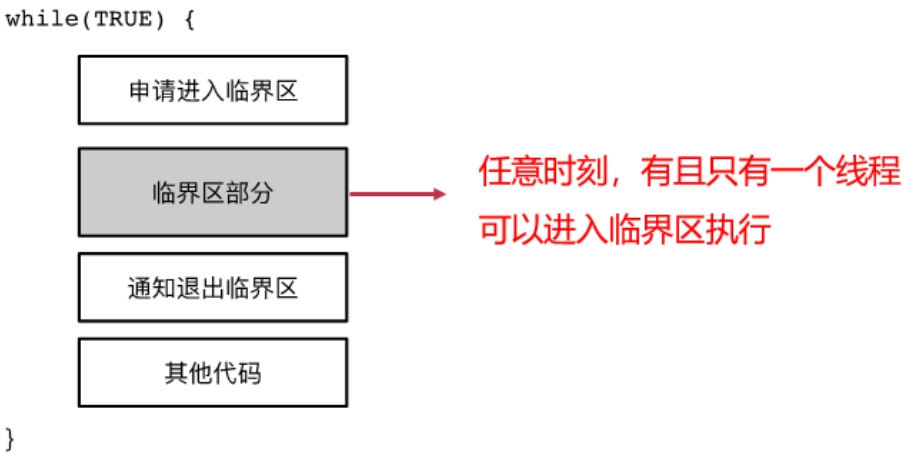
对应的同步原语

同步原语	描述	使用场景
互斥锁	保证对共享资源的**互斥访问**	**场景一** 共享资源互斥访问
读写锁	允许读者线程**并发读取**共享资源	**衍生场景一** 读写场景并发读取
条件变量	提供线程**睡眠**与**唤醒**机制	**场景二** 条件等待与唤醒
信号量	协调**有限数量**资源的消耗与释放	**场景三** 多资源协调管理

3. 同步与临界区

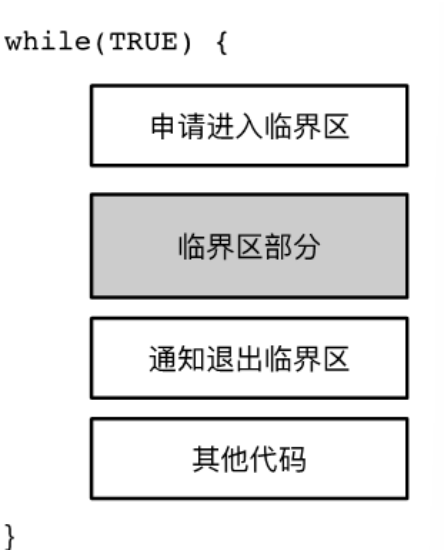
抽象：临界区

> *临界区：Critical Section*



实现临界区抽象的三个要求

- **互斥访问**：在同一时刻，**有且仅有一个线程**可以进入临界区
- **有限等待**：当一个线程申请进入临界区之后，必须在**有限的时间**内获得许可进入临界区而不能无限等待
- **空闲让进**：当没有线程在临界区中时，必须在申请进入临界区的线程中选择一个进入临界区，保证执行临界区的**进展**



什么是同步原语?

- **同步原语**(Synchronization Primitives)是已给平台（如**操作系统**）提供的用于帮助开发者实现线程之间**同步**的软件工具

  > 有限的共享资源上正确的协同工作

- > Q：关闭所有核心的中断能解决临界区问题嘛?
  >
  > A：可以解决单个CPU核上的临界区问题。如果在多个核心中，关闭中断不能阻塞其它进程执行（**并不能阻止多个CPU核同时进入临界区**）

4. 互斥锁

> 就是spinlock(现在感觉又不太一样)

互斥锁(Mutual Exclusive Lock)接口

1. Lock(lock)：尝试拿到锁"lock"
   - 当前没有线程拿锁：拿到锁，并继续往下执行
   - 当前有线程拿锁：不断循环等待放锁(busy loop)
2. Unlock(lock)：释放锁

> 保证同时只有一个线程能够拿到锁

一个例子

> 用pthread库提供的互斥锁实现

创建3个线程，同时执行下面程序：

```
unsigned long a = 0;
void *routine(void *arg) {
    for (int i = 0; i < 1000000000; i++) {
        pthread_mutex_lock(&global_lock);
        a++;
        pthread_mutex_unlock(&global_lock);
    }
    return NULL;
}
```

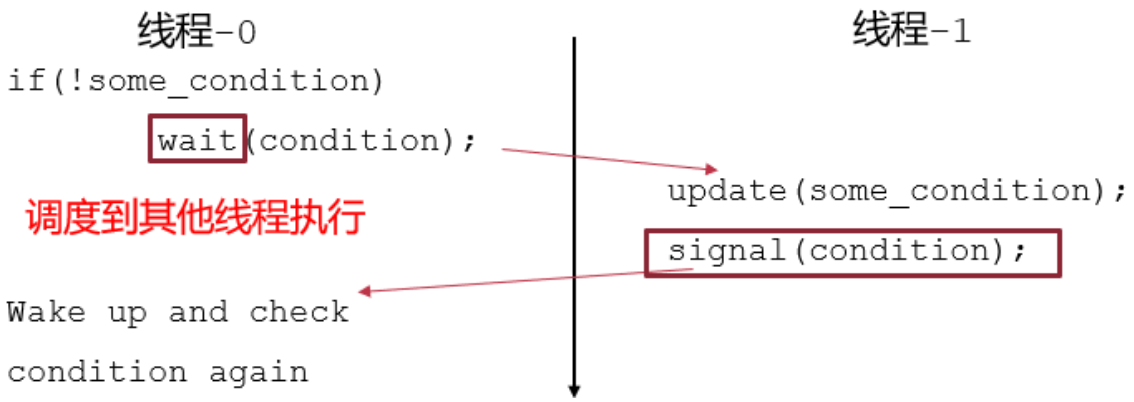
pthread库提供的互斥锁实现

输出结果为： 3000000000

5. 条件变量

> 利用**睡眠/唤醒**机制，避免无意义的等待，**让操作系统的调度器调度其他进程/线程运行**

使用方式



条件变量的接口

1. `void cond_wait(struct cond *cond, struct lock *mutex);`
   - 放入条件变量的**等待队列**
   - 阻塞自己同时**释放锁：即调度器可以调度到其他线程**
   - 被唤醒后重新**获取锁**
2. `void cond_signal(struct cond *cond);`
   - 检查**等待队列**
   - 如果有等待者则**移出等待队列并唤醒**

条件变量的使用示例

等待空位代码

```
1. ...
2. /* Wait empty slot */
3. lock(empty_cnt_lock);
4. while (empty_slot == 0)
5.     cond_wait(empty_cond,
6.               empty_cnt_lock);
7. empty_slot--;
8. unlock(empty_cnt_lock);
9. ...
```

思考：为什么这里要用while?

生产空位代码

```
1. ...
2. /* Add empty slot */
3. lock(empty_cnt_lock);
4. empty_slot++;
5. cond_signal(empty_cond);
6. unlock(empty_cnt_lock);
7. ...
```

> 因为可能有两个线程同时在等待，其中一个线程拿到了锁`empty_cnt_lock`后将empty_slot减1到0然后放锁，另一个线程被唤醒，拿到了锁`empty_cnt_lock`，但此时empty_slot为0，如果用if的话会直接进入critical section，造成错误、

6. 信号量

> ***信号量(Semaphore)：协调（阻塞/放行）多个线程共享有限数量的资源***
>
> 语义上：信号量的值cnt记录了**当前可用资源的数量**
>
> （实现上就是对信号量的一种封装）

生产者消费者问题的另外一种实现

> 即不采用队列序号的方式，采用计数的方式

生产者：　　使用 **互斥锁** 搭配 **条件变量** 完成资源的等待与消耗

```
while(true) {
        new_msg = produce_new();
        lock(&empty_slot_lock);
        while (empty_slot == 0)
                cond_wait(&empty_cond, &empty_slot_lock);
        empty_slot--;
        unlock(&empty_slot_lock);

        buffer_add(new_msg);
        // ...
}
```

PV原语

> 提供了两个原语P和V用于**等待/消耗**资源
>
> 解决了之前实现方式中**需要单独创建互斥锁与条件变量，并手动通过计数器来管理资源数量**的问题

P操作：消耗资源

```
void sem_wait(sem_t *sem) {
        while(sem->cnt <= 0)
                /* Waiting */;
        S--;
}
```

cnt代表剩余资源数量

V操作：增加资源

```
void sem_signal(sem_t *sem) {
        sem->cnt++;
}
```

信号量的使用

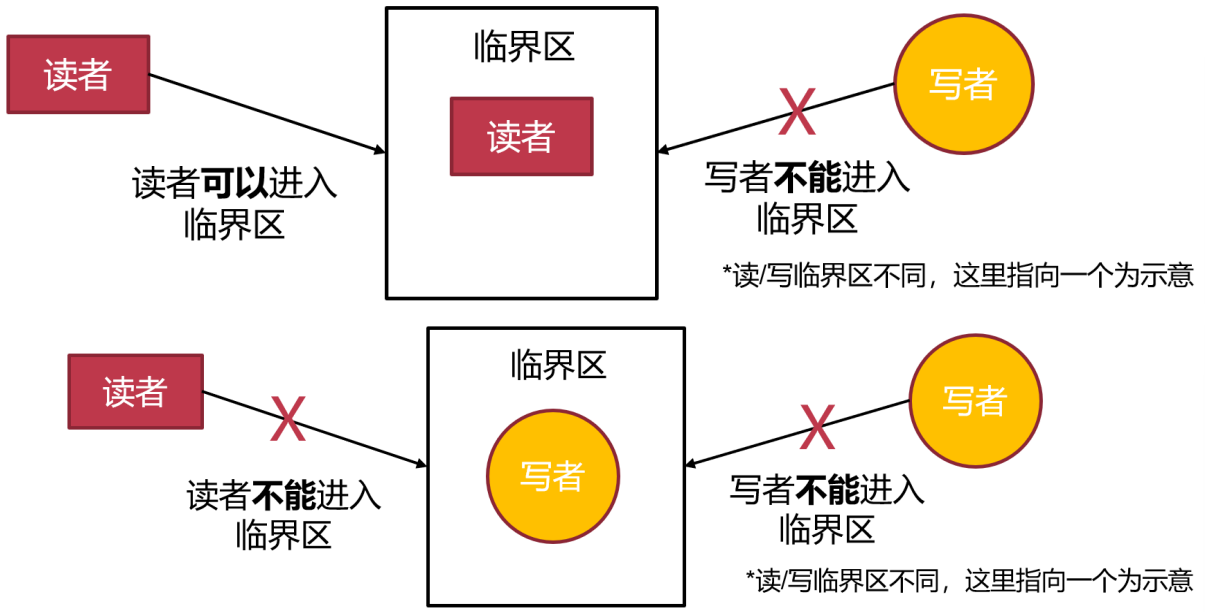
```
void producer(void) {
        new_msg = produce_new();
        sem_wait(&empty_slot_sem);          消耗empty_slot
        buffer_add(new_msg);
        sem_signal(&filled_slot_sem);       增加filled_slot
}

void consumer(void) {
        sem_wait(&filled_slot_sem);         消耗filled_slot
        cur_msg = buffer_remove();
        sem_signal(&empty_slot_sem);        增加empty_slot
        handle_msg(cur_msg);
}
```

> 1. 当初始化的资源数量为1时，为**二元信号量**（同一时刻**只有一个**线程能够拿到资源）
> 2. 当初始化的资源数量大于1时，为**计数信号量**（同一时刻**可能有多个**线程能够拿到资源）

7. 读写锁

> 区分读者与写者，允许读者之间并行，读者与写者之间互斥



- 读者使用`reader_lock`，写者使用`writer_lock`

```
struct rwlock *lock;
char data[SIZE];

void reader(void) {
        lock_reader(lock);
        read_data(data)
        unlock_reader(lock);
}

void writer(void) {
        lock_writer(lock);
        update_data(data);
        unlock_writer(lock);
}
```

8. 同步原语的对比

互斥锁/条件变量/信号量

- **互斥锁**与**二元信号量**功能类似，但**抽象不同**
    1. 互斥锁有拥有者的概念，一般同一个线程拿锁/放锁
    2. 信号量为资源协调，一般一个线程signal，另一个线程wait

- 条件变量用于解决不同问题（睡眠/唤醒），需要搭配**互斥锁**使用

```
lock(&empty_slot_lock);
while (empty_slot == 0)
        cond_wait(&empty_cond,
        &empty_slot_lock);
empty_slot--;
unlock(&empty_slot_lock);
```

搭配互斥锁+计数器可以实现与信号量相同的功能

```
sem_wait(&empty_slot_sem);
```

互斥锁 V.S. 读写锁

- 接口不同：读写锁区分读者与写者
- 针对场景不同：获取更多程序语义，表明只读代码段，达到更好的性能
- 读写锁在读多写少的场景可以显著**提升读者的并行度**
- 只用写者锁，则与互斥锁的语义基本相同

9. 同步原语带来的问题：死锁

死锁产生的原因

- **互斥访问**
- **持有并等待**
- **资源非抢占**
- **循环等待**

A等B，B等A

```
void proc_A(void) {
        lock(A);
        /* Time T1 */
        lock(B);
        /* Critical Section */
        unlock(B);
        unlock(A);
}

void proc_B(void) {
        lock(B);
        /* Time T1 */
        lock(A);
        /* Critical Section */
        unlock(A);
        unlock(B);
}
```

T1时刻的死锁

- **互斥访问**

  > 同一时刻只有一个线程能够访问

- **持有并等待**

  > 一直持有一部分资源并等待另一部分，不会中途释放(比如proc_A不会释放锁A)

- **资源非抢占**

  > 即proc_B不会抢proc_A已经持有的锁A

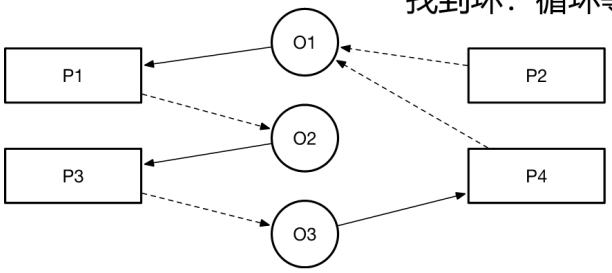
- **循环等待**

  > A等B，B等A

如何解决死锁?

1. **出问题再处理：死锁的检测与恢复**

找到环：循环等待



资源分配表

进程号	资源号
P1	O1
P3	O2
P4	O3

进程等待表

进程号	资源号
P1	O2
P2	O1
P3	O3

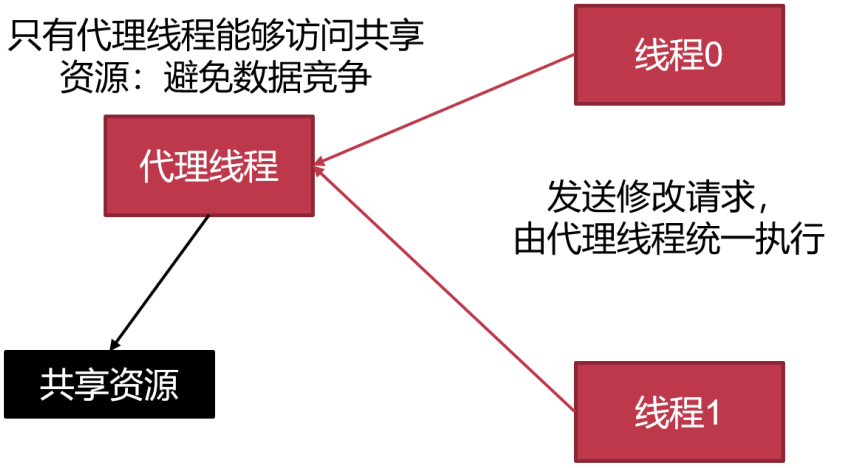
资源分配图

如何恢复？打破循环等待！

- 直接kill所有循环中的线程
- Kill一个，看有没有环，有的话继续kill
- 全部回滚到之前的某一状态

2. **设计时避免：死锁预防**

1. 避免互斥访问：通过其他手段（如代理执行）



*代理锁 (Delegation Lock) 实现了该功能

2. 不允许持有并等待：一次性申请所有资源
3. 资源允许抢占：需要考虑如何恢复

- 1、避免互斥访问：通过其他手段（如代理执行）
- 2、不允许持有并等待：一次性申请所有资源
- 3、资源允许抢占：需要考虑如何恢复

需要让线程A正确回滚到拿锁A之前的状态

```
void proc_A(void) {
        lock(A);
        /* Time T1 */
        lock(B);
        /* Critical Section */
        unlock(B);
        unlock(A);
}
        抢占锁A
void proc_B(void) {
        lock(B);
        /* Time T1 */
        lock(A);
        /* Critical Section */
        unlock(A);
        unlock(B);
}
```

4. 打破循环等待：按照特定的顺序获取资源
    - 对所有资源进行编号
    - 让所有线程递增获取

> 任意时刻：获取最大资源号的线程可以继续执行，然后释放资源

3. **运行时避免死锁：死锁避免**

死锁避免：银行家算法

死锁避免：运行时检查是否会出现死锁

银行家算法的核心：

- 所有线程获取资源需要通过**管理者**同意
- 管理者**预演**会不会造成死锁
    - 如果会造成：阻塞线程，下次再给
    - 如果不会造成：给线程该资源

> 具体信息见PPT