Lecture11 Implementation of Sync

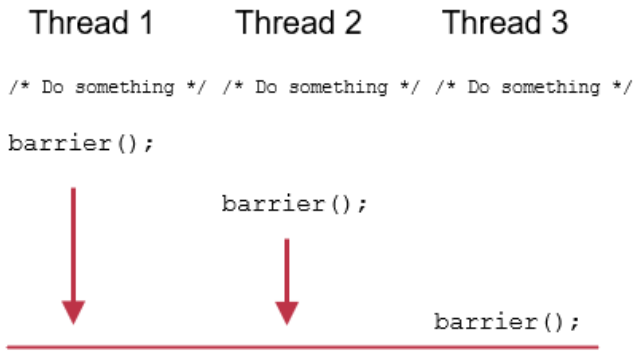
同步的案例分析

> 场景1：共享资源互斥访问->互斥锁
>
> 衍生场景1：读写场景并行读取->读写锁
>
> 场景2：条件等待与唤醒->条件变量 (cond_wait & cond_signal)
>
> 场景3：多资源协调管理->信号量

同步案例-1：多线程执行屏障

- **多线程执行屏障**
- **等待全部执行到屏障后再继续执行**

符合场景2: 线程等待/唤醒

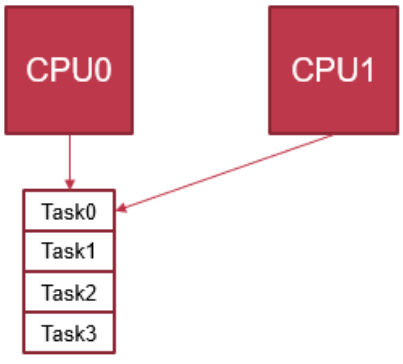



```
lock(&thread_cnt_lock);
thread_cnt--;
if (thread_cnt == 0)
  cond_broadcast(cond);        唤醒所有等待的线程
while(thread_cnt != 0)
  cond_wait(&cond, &thread_cnt_lock);
unlock(&thread cnt lock);
```

同步案例-2：等待队列工作窃取

- **每核心等待队列**
- **在空时允许窃取其他核心的任务**

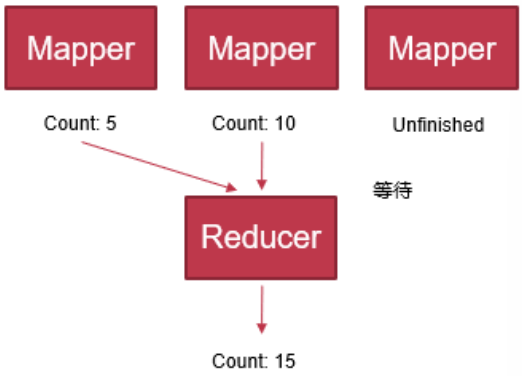
符合场景1: 共享资源互斥访问




```
lock(ready_queue_lock[0]);
```

同步案例-3：map-reduce

- Word-count：大文本拆分字数统计
- Mapper：统计一部分文本自述
- Reducer：一旦其中任意数量的Mapper结束，就累加其结果



符合场景2: 线程等待/唤醒

Reducer

```
lock(&finished_cnt_lock);
while(finished_cnt == 0)
  cond_wait(&cond, &finished_cnt_lock);
/* collect result */
finished_cnt = 0;
unlock(&thread cnt lock);
```

一次性拿走所有的finished的Mapper的结果

```
Reducer:
    lock(&finished_cnt_lock);
    while (finished_cnt == 0)
        cond_wait(&cond, &finished_cnt_lock);
    /* collect result */
    finished_cnt = 0;
    unlock(&finished_cnt_lock);
```

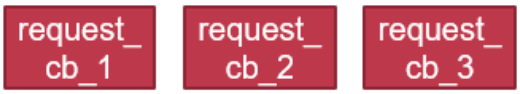
> 同时本场景也符合**场景3**：将Mapper的结果视作资源
>
> 不同之处在于，上一种实现方式可以collect多个结果，本方法依次只能拿走一个
>
> (为什么我感觉这两种实现方式几乎没有差别？)

```
Mapper:
    signal(&finish_sem); /* 这里还应该更新mapper_cnt */
Reducer:
    while (finished_cnt != mapper_cnt) {
        wait(&finish_sem);
        /* collect result */
        finished_cnt++;
    }
```

同步案例-4：网页渲染

> 个人感觉更像是衍生场景1，类似于***偏向读者的读写锁***

- 网页等待所有的请求均完成后再进行渲染



场景2: 等待/唤醒

Request_cb

```
lock(&glock);
finished_cnt ++;
if (finished_cnt == req_cnt)
  cond_signal(&gcond);
unlock(&glock);
```

渲染线程


```
lock(&glock);
while (finished_cnt != req_cnt)
  cond_wait(&gcond, &glock);
unlock(&glock);
```

同步案例-5：线程池并发控制

- 控制同一时刻可以执行的线程数量
- 原因：有的线程阻塞时可以允许新的线程替上
- 例子：允许同时三个线程执行

场景3: 视剩余可并行执行线程数量为有限资源

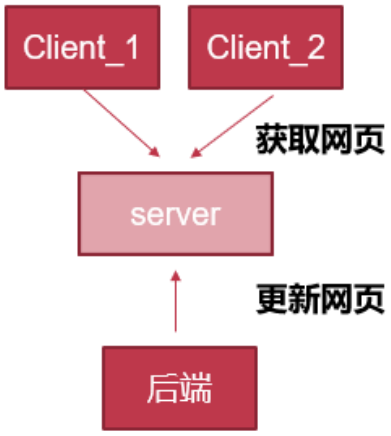
```
thread_routine () {
  wait(&thread_cnt_sem);
  /* doing something */
  signal(&thread_cnt_sem);
}
```

同步案例-6：网页服务器

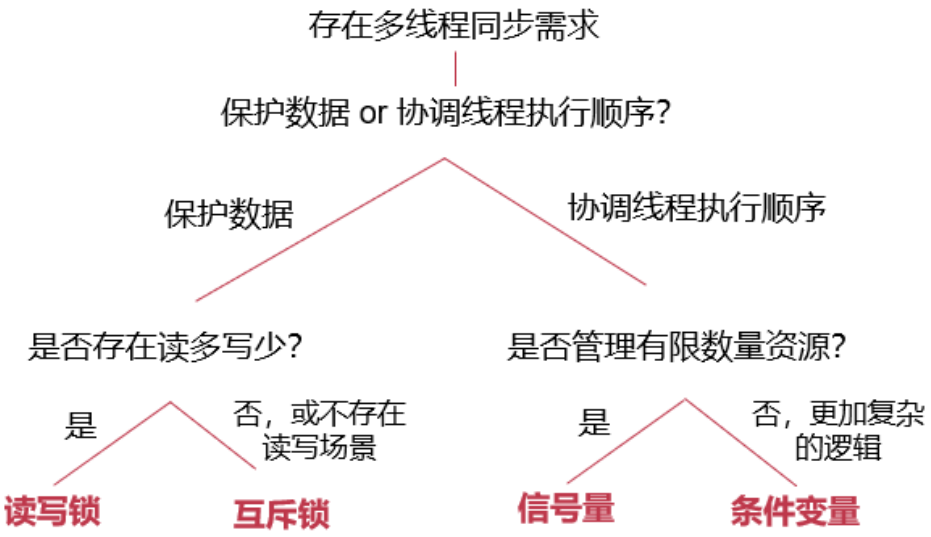
- 处理响应客户端获取静态网页需求
- 处理后端更新静态网页需求
- 不允许读取更新到一半的页面

衍生场景1: 读写场景，可以使用读写锁

- client用读锁
- 后端用写锁



同步原语选择的Guideline



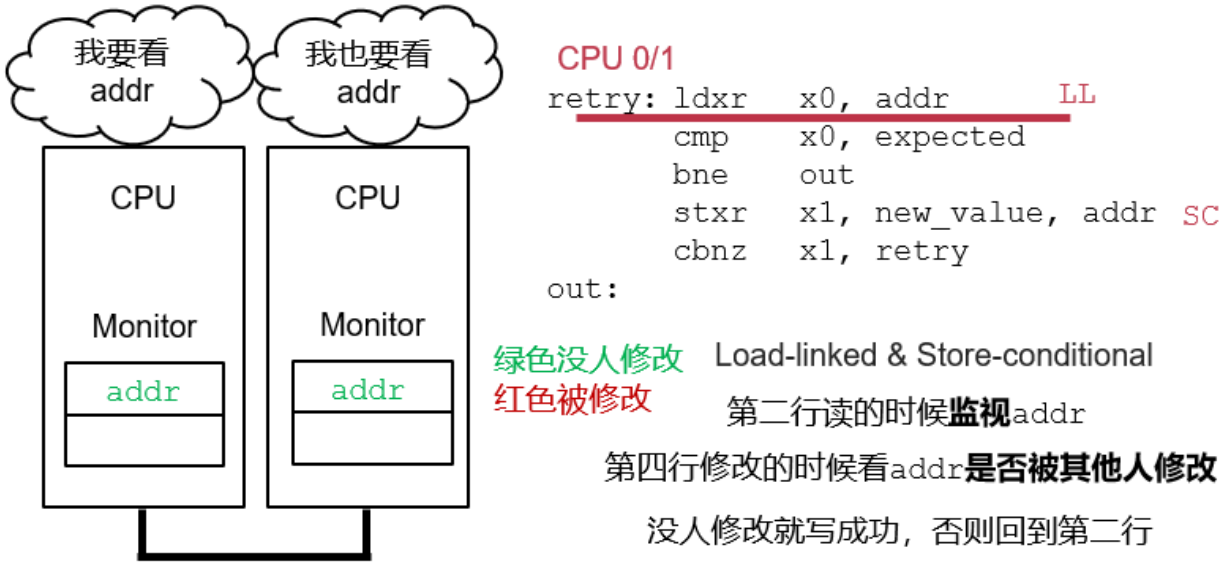
硬件原子指令

常见的原子指令锁实现

1. Test-and-set
2. Compare-and-swap
3. Load-linked & Store-conditional
4. Fetch-and-add

LL/SC

> ARM使用LL/SC实现硬件原子操作
>
> 1. ldxr在addr处放置一个监视器(monitor)
> 2. 比较x0和expected（看看锁是否被放掉了）
> 3. 如果没放掉，跳转到out，将旧值x0 (也就是1) 返回
> 4. 如果放掉了，则用strx指令尝试去将new_value写入到addr，**如果有人抢先写入了，则将x1设置为1，否则x1的值为0，将new_value写入（也就是说拿到锁了）**
> 5. 将旧值x0返回（也就是0）

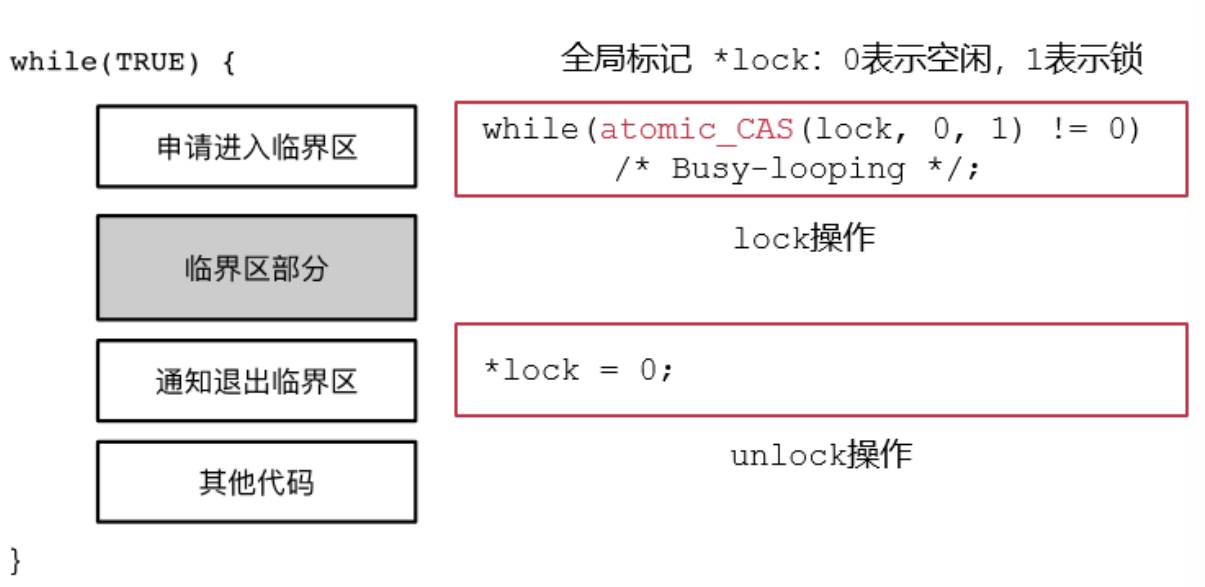


1. The `LDXR` instruction loads a value from a memory address and attempts to silently claim an exclusive lock on the address.
2. The `STXR` instruction then writes a new value to that location only if the lock was successfully obtained and held.

互斥锁的实现

自旋锁(Spinlock)

> int atomic_CAS(int *addr, int expected, int new_value);



1. 互斥访问 -> 可以保证
2. 有限等待 -> 不可以保证，有些运气差的thread可能永远也拿不到锁
3. 空闲让进 -> 不可以保证，这依赖于硬件实现

排号锁(Ticketlock)

> int atomic_FAA(int *addr, int add_value);

思考：我们如何保证竞争者的公平性?

通过遵循竞争者到达的顺序来传递锁。

owner：表示当前的持有者 next：表示目前放号的最新值

lock操作

```
1. my_ticket = atomic_FAA(
        &lock->next, 1);
2. while(lock->owner !=
        my_ticket)
        /* waiting */;
```

拿号

等号

unlock操作

```
1. lock->owner ++;
```

叫号

1. 互斥访问 -> 可以保证
2. 有限等待 -> 按照顺序，在前序竞争者保证有限等待时间释放时，可以达到有限等待
3. 空闲让进 -> 可以保证

读写锁的实现

> 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
>
> 1. 偏向写者的读写锁：**后续读者必须等待写者进入后才能进入临界区**
> 2. 偏向读者的读写锁：**后续读者可以直接进入临界区**

偏向读者的读写锁实现示例

Reader计数器：
表示有多少读者

```
struct rwlock {
        int reader;
        struct lock reader_lock;
        struct lock writer_lock;
};

void lock_reader(struct rwlock *lock) {
        lock(&lock->reader_lock);
        lock->reader += 1;
        if (lock->reader == 1) /* No reader there */
                lock(&lock->writer_lock);
        unlock(&lock->reader_lock);
}
```

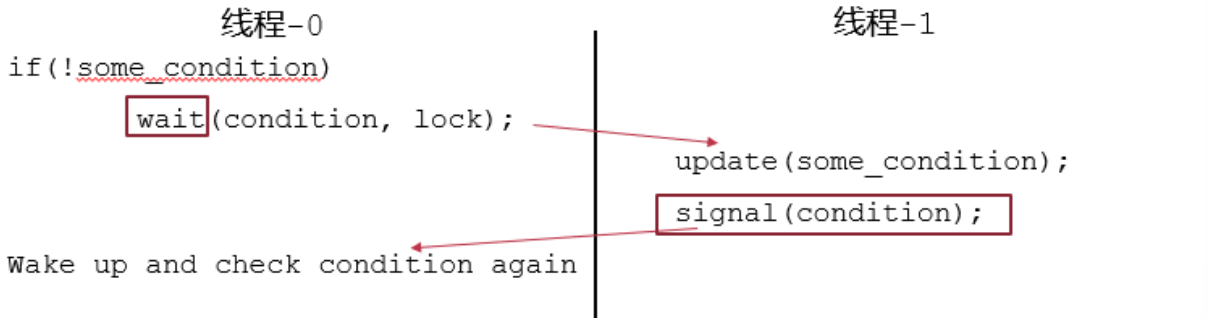
第一个/最后一个reader负责获取/释放写锁

```
void unlock_reader(struct rwlock *lock) {
        lock(&lock->reader_lock);
        lock->reader -= 1;
        if (lock->reader == 0) /* Is the last reader */
                unlock(&lock->writer_lock);
        unlock(&lock->reader_lock);
}

void lock_writer(struct rwlock *lock) {
        lock(&lock->writer_lock);
}

void unlock_writer(struct rwlock *lock) {
        unlock(&lock->writer_lock);
}
```

只有当完全没有读者时写者才能进入临界区

条件变量的实现

> 条件变量：提供睡眠/唤醒机制，避免无意义的等待



> void wait(struct cond *cond, struct lock *mutex);

```
list_append(cond->wait_list, proc_self());
unlock(mutex), yield();     // 注意，这两步是原子性的
lock(mutex);
```

1. 放入条件变量的**等待队列**
2. 阻塞自己同时**释放mutex锁**
3. 被唤醒后重新**获取mutex锁**

> void signal(struct cond *cond);

```
if (!list_empty(cond->wait_list))
    wakeup(list_remove(cond->wait_list));
```

1. 检查**等待队列**
2. 如果有等待者则**移出等待队列并唤醒**

信号量的实现

信号量的实现-1：忙等

```
void wait(sem_t *S) {
        lock(S->sem_lock);
        while(S->value == 0) {
                unlock(S->sem_lock);
                lock(S->sem_lock);
        }
        S->value --;
        unlock(S->sem_lock);
}

void signal(sem_t *S) {
        lock(S->sem_lock);
        S->value ++;
        unlock(S->sem_lock);
}
```

Busy looping，无意义等待

此时已经取得sem_lock，防止同时-1

信号量的实现-2：条件变量

```
void wait(sem_t *S) {
        lock(S->sem_lock );
        while(S->value == 0) {
                cond_wait(S->sem_cond, S->sem_lock);
        }
        S->value --;
        unlock(S->sem_lock);
}

void signal(sem_t *S) {
        lock(S->sem_lock);
        S->value ++;
        cond_signal(s->sem_cond);
        unlock(S->sem_lock);
}
```

使用条件变量避免无意义等待

每次都要signal，很可能无人等待

信号量的实现-3：减少signal的次数

> 这是一种有错误的实现方式，正确的实现方式见 实现-4

```
void wait(sem_t *S) {
        lock(S->sem_lock );
        S->value --;
        while(S->value < 0) {
          cond_wait(S->sem_cond, S->sem_lock);
        }
        unlock(S->sem_lock);
}

void signal(sem_t *S) {
        lock(S->sem_lock);
        S->value ++;
        if (S->value < 0)
          cond_signal(s->sem_cond);
        unlock(S->sem_lock);
}
```

value减到负数代表有人等待

会有什么问题?

比如S->value = -3

signal 后S->value = -2

还是不满足上面while的条件

思考：需要额外的计数器用于单独记录有多少可以唤醒的

改进：加入条件判断是否需要wake

信号量的实现-4：增加wakeup变量

> 信号量 = **条件变量 + 互斥锁 + 计数器**
>
> 新增一个变量**wakeup**：等待时可以唤醒的数量
>
> 某一时刻真实的资源数：`value < 0 ? wakeup : value + wakeup`

```
void wait(sem_t *S) {
    lock(S->sem_lock);
    S->value--;
    if (S->value < 0) {
        do {
            cond_wait(S->sem_cond, S->sem_lock);
        } while (S->wakeup == 0);
        S->wakeup--;
    }
    unlock(S->sem_lock);
}
```

```
void signal(sem_t *S) {
    lock(S->sem_lock);
    S->value++;
    if (S->value <= 0) {
        S->wakeup++;
        cond_signal(S->sem_cond);
    }
    unlock(S->sem_lock);
}
```

> Q：为什么是do while?
>
> A：因为我们需要**有限等待**，确保调用signal后立刻调用wait的线程不会直接拿走资源，而是交给正在等待的线程

RCU：更高效的读写互斥

> Read Copy Update(RCU)

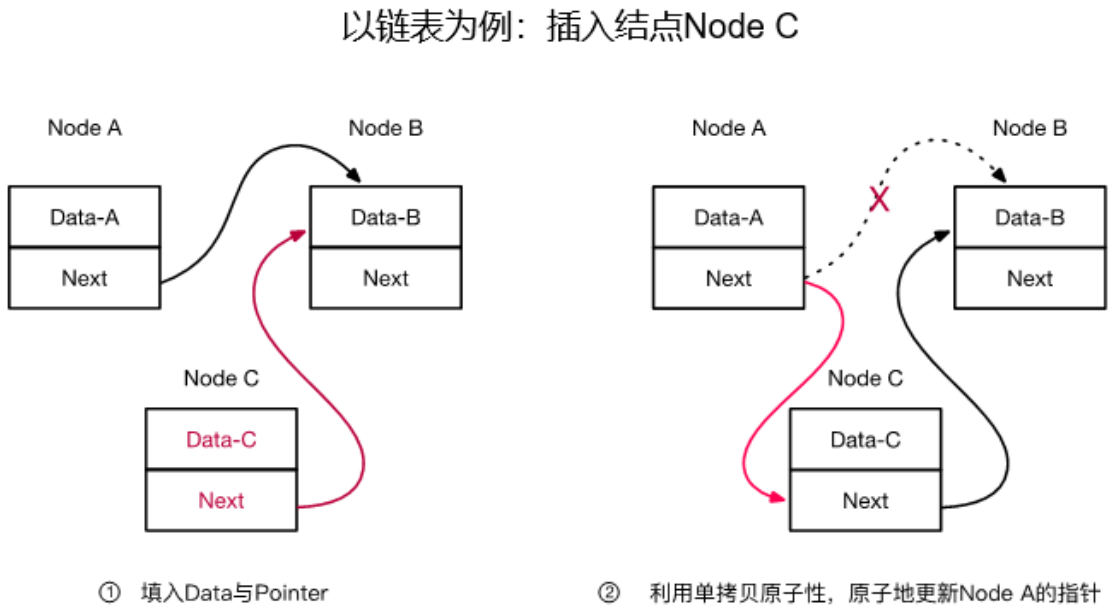
RCU订阅/发布机制

- **需求1**：需要一种能够**类似之前硬件原子操作的方式**，让读者要么看到旧的值，要么看到新的值，不会读到任何中间结果

  > 硬件原子操作：
  >
  > 1. 硬件原子操作有大小限制(最大128bit)
  > 2. 性能瓶颈

- 单拷贝原子性(Single-copy atomicity)：

  > 处理器任意一个操作原子可见 e.g.更新一个指针

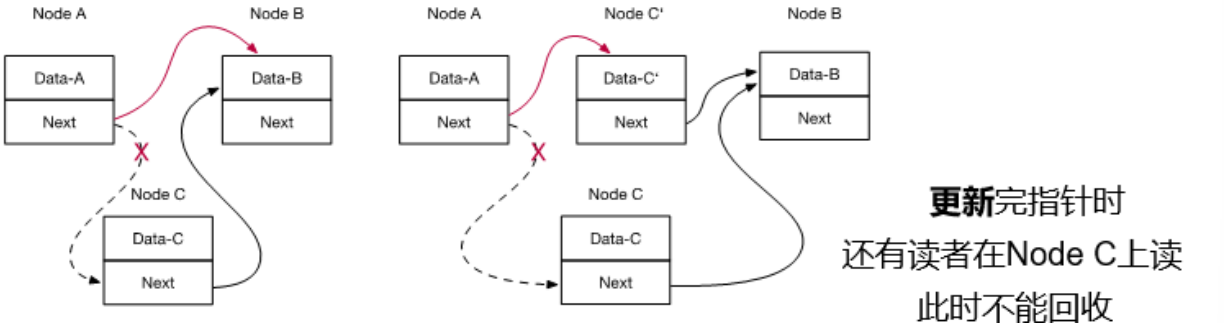


> 局限性1：无法在复杂场景下使用，如双向链表

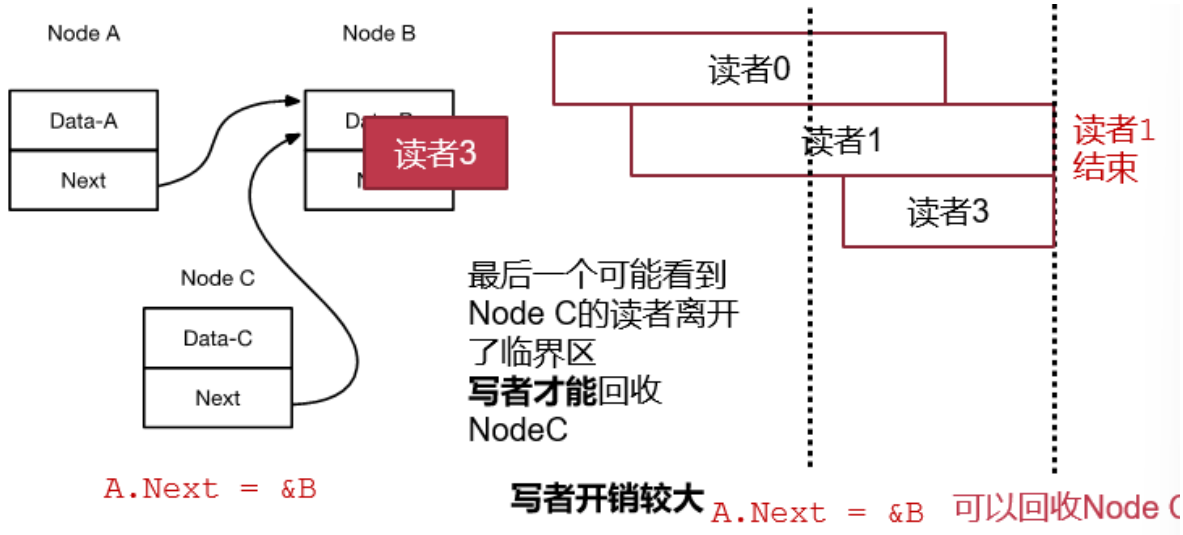
RCU宽限期

- **需求2**：在合适的时间，回收无用的旧拷贝

  > 局限性2：会产生大量的垃圾



> 我们需要知道读临界区什么时候开始，什么时候结束
>
> **最后一个可能读到被删除节点的读者离开了临界区，写者才能回收垃圾节点->写者开销大**



- 可能的实现方式

如何知道读临界区什么时候开始，什么时候结束？

```
void rcu_reader() {
        RCU_READ_START();

        /* Reader Critical Section */

        RCU_READ_STOP();
}
```

通知RCU，读者进临界区了

通知RCU，读者出临界区了

可以使用不同的方式实现：如计数器

同步原语对比：读写锁 vs RCU

读写锁 RCU

相同点：

允许读者并行

不同点：

- 读者也需要上读者锁
- 关键路径上有额外开销
- 方便使用
- 可以选择对写者开销不大的读写锁

- 读者无需上锁
- 使用较繁琐
- 写者开销大