同步：进阶
Synchronization: Advances

钮鑫涛
南京大学
2024春

本门课由蒋炎岩和钮鑫涛共同构建

互斥锁、条件变量、信号量

- 一般来说，信号量可以完成互斥锁的工作（其本身就可以看成一个更加高端的互斥锁）
- 信号量也能很方便的完成某些线程控制
- 但其不能胜任所有事情 （比如，其不太好实现条件变量）
  - 小心泛化(by Lampson)！
- 条件变量可以适用于任何同步条件，其和互斥锁一起可以实现信号量

读者-写者(Readers-writers)问题

读者-写者问题

- 多个线程想要读取某个数据，有一个或多个线程需要写某个数据
- 与之前简单的将所有线程全部互斥不同，读者-写者问题会对读者的同步放宽要求（因为“读”操作不会改变共享数据）：
  - 任何数量的读者都可以同时进行“读”操作（读-读不需要互斥）
  - 但一次只能有一个写者进行“写”操作 （写-写互斥）
  - 如果有一个写者在写，那么此刻没有读者（写-读互斥）

读者-写者问题

- 简单的对所有线程加上互斥锁当然可以解决此问题，但是性能太低，因为这会限制读者之间无需互斥的自由
- 为此，我们需要实现一个新的锁—“读写锁”（readers/writers lock）来保护共享数据
  - 其可以允许多个“读者”同时访问共享数据，只要他们中没有一个修改该数据
  - 一次只能有一个“写者”可以持有读写锁进入临界区，因此可以安全的读和写数据

```
rwlock_t *rw
rwlock_init(rw);
rw->acquire_readlock();
// Read shared data
rw->release_readlock();
```

```
rw->acquire_writelock();
// Read and write shared data
rw->release_writelock();
```

读写锁的实现1

- 读者-写者的问题本质上就是分别给出当前读者和写者是否可以进入临界区的条件
  - 对于读者而言，如果临界区为空或者临界区有其他读者，那么就可进入
  - 对于写者而言，只有临界区为空才可进入
- 要实现上述条件，需要记录当前临界区的“读者”数量

读写锁的实现1

- 基于上述条件的实现如下：

```
typedef struct _rwlock_t{
    sem_t lock; //保护自身锁状态的锁
    sem_t rwlk; //进入临界区获得锁
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw){
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->rwlk, 0, 1);
    rw->readers = 0;
}
```

```
void acquire_readlock(rwlock_t *rw){
    P(&rw->lock);
    if(rw->readers == 0){
        //此时临界区还没有读者
        P(&rw->rwlk); //争抢进入临界区的锁
    }
    //rw->readers > 0 或者 得到了rwlk
    rw->readers++;
    V(rw->lock);
}

void release_readlock(rwlock_t *rw){
    P(&rw->lock);
    rw->readers--;
    if(rw->readers == 0){
        V(&rw->rwlk);
    }
    V(rw->lock);
}
```

```
void acquire_writelock(rwlock_t *rw){
    P(&rw->rwlk);
}

void release_writelock(rwlock_t *rw){
    V(&rw->rwlk);
}
```

上述实现问题

- 上述实现中，如果一直有读者反复进入临界区，那么写者就会进入不了临界区，从而饿死(starvation)
  - 只有等只有最后一个读者退出临界区，其才会释放rwlk
- 这个实现也被称为读者优先(Reader Preference), 显然对写不友好

读写锁的实现2

- 一个改进想法：如果有写者想要进入临界区，那么其应该阻止后来的读者尝试进入临界区

```
typedef struct _rwlock_t{
    sem_t rlock; //读操作时保护自身锁操作的锁
    sem_t wlock; //写操作时保护自身锁操作的锁
    sem_t rwlk; //进入临界区获得锁
    sem_t tryRead; //读者进入之前先看等待写者

    int readers;
    int writers; //想要进入的写者
} rwlock_t;

void rwlock_init(rwlock_t *rw){
    sem_init(&rw->rlock, 0, 1);
    sem_init(&rw->wlock, 0, 1);
    sem_init(&rw->rwlk, 0, 1);
    sem_init(&rw->tryRead, 0, 1);
    rw->readers = 0;
    rw->writers = 0;
}
```

```
void acquire_readlock(rwlock_t *rw){
    P(&rw->tryRead);
    P(&rw->rlock);
    if(rw->readers == 0){
        //此时临界区还没有读者
        P(&rw->rwlk); //争抢进入临界区的锁
    }
    //rw->readers > 0 或者 得到了rwlk
    rw->readers++;
    V(rw->rlock);
    V(&rw->tryRead);
}

void release_readlock(rwlock_t *rw){
    P(&rw->rlock);
    rw->readers--;
    if(rw->readers == 0){
        V(&rw->rwlk);
    }
    V(rw->rlock);
}
```

```
void acquire_writelock(rwlock_t *rw){
    P(&rw->wlock);
    if(rw->writers == 0){//第一个写者
        P(&rw->tryRead);
    }
    rw->writers++;
    V(&rw->wlock);
    P(&rw->rwlk);
}

void release_writelock(rwlock_t *rw){
    V(&rw->rwlk);
    P(&rw->wlock);
    rw->writers--;
    if( rw->writers == 0){
        V(&rw->tryRead);
    }
    V(&rw->wlock);
}
```

上述实现问题

- 上述实现中，如果一直有写者想要进入临界区，那么读者就会进入不了临界区，从而饿死(starvation)
  - 只有等只有最后一个写者退出临界区，其才会释放tryRead，后续读者才能进入
- 这个实现也被称为写者优先(Writer Preference), 显然对读不友好

读写锁的实现3

- 应该给线程排队！排在前面的优先尝试进入临界区
  - 如何排队？ ticket lock!

```
typedef struct  lock_ticket {
    int ticket; //当前发放的最大票号
    int turn;   //当前应该进入的票号
} ticket_t;

void lock_init(ticket_t*  tlk) {
    tlk->ticket = 0;
    tlk->turn   = 0;
}
```

```
void ticket_lock(ticket_t*  tlk) {
    int myturn = __sync_fetch_and_add (&tlk->ticket, 1);
    while (tlk->turn != myturn)
        ; // spin
}
```

```
void ticket_unlock(ticket_t*  tlk) {
    __sync_fetch_and_add (&tlk->turn, 1);
}
```

读写锁的实现3

- 我们就实现了一个较为公平的读写锁

```
typedef struct _rwlock_t{
    sem_t rlock; //读操作时保护自身锁操作的锁
    sem_t rwlk; //进入临界区获得锁

    ticket_t queue; //维持一个先进先出的队列
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw){
    sem_init(&rw->rlock, 0, 1);
    sem_init(&rw->rwlk, 0, 1);

    lock_init(&rw->queue);
    rw->readers = 0;
}
```

```
void acquire_readlock(rwlock_t *rw){
    ticket_lock(&rw->queue);
    P(&rw->rlock);
    if(rw->readers == 0){
        //此时临界区还没有读者
        P(&rw->rwlk); //争抢进入临界区的锁
    }
    rw->readers++;
    V(rw->rlock);
    ticket_unlock(&rw->queue);
}

void release_readlock(rwlock_t *rw){
    P(&rw->rlock);
    rw->readers--;
    if(rw->readers == 0){
        V(&rw->rwlk);
    }
    V(rw->rlock);
}
```

```
void acquire_writelock(rwlock_t *rw){
    ticket_lock(&rw->queue);
    P(&rw->rwlk);
    ticket_unlock(&rw->queue);
}

void release_writelock(rwlock_t *rw){
    V(&rw->rwlk);
}
```

Posix的读写锁

- 读者-写者问题是一个非常普遍的问题，Posix也给出了相应的读写锁

```c
#include <pthread.h>
pthread_rwlock_t rw;
int pthread_rwlock_init(&rw, NULL);
int pthread_rwlock_destroy(&rw);


int pthread_rwlock_rdlock(&rw);
int pthread_rwlock_wrlock(&rw);
int pthread_rwlock_unlock(&rw);
```

linux下pthread的默认实现是读者优先，但是可以通过设置PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP来改变

*Read-Copy-Update

紫薇软剑，三十岁前所用，误伤义士不祥，悔恨无已，乃弃之深谷

重剑无锋，大巧不工，四十岁前持之横行天下

四十岁后，不滞于物，草木竹石均可为剑

自此精修，渐进于无剑胜有剑之境

新境界：无锁（Non-blocking algorithm）

- 锁的获取和释放都是有代价的
  - 一般来说互斥锁上锁/释放的需要的CPU cycles比普通指令多一个数量级
  - 在短临界区的情况下，锁本身占据了最多的时间，成为了并发程序的瓶颈
    - 实际上，在这种情况下，线程越多，反而性能越低
- 虽然在读者-写者情况下“读-读”可以避免互斥，“写-写”天生需要互斥，但“读-写”呢?

一个想法

- 在数据的“一致性”要求不那么严的情况下（强一致性：当一个节点更新数据后，其他节点可以立即获取到最新的数据），读者可以不被写者阻塞
  - 当写者更新数据时，读者可以
    - 要么读到一个旧数据（写操作之前的数据）
    - 要么读到最新的数据（写操作之后的数据）
    - 除此之外，不能获得中间状态！保证“写”的原子性！
  - 当然系统最终是一致的，即当一个节点更新数据后，其他节点可能需要经过一段时间才能获得最新数据，但最终总能获得（最终一致性）

Read-Copy-Update（RCU）

- Linux在2002年（内核2.5）引入的新的同步机制，之后就被广泛的使用（目前2024年linux内核中10%以上的锁都基于RCU），该同步机制比起之前的读写锁更加的highly scalable
- 对于RCU下的读者:
  - 可以并发的和写者一起运行，而不会被阻塞
  - 因此读操作几乎没有overhead
- 相对（简单的读写锁）而言，RCU下的写操作需要一些额外的成本
- 其适用于频繁的读，但不频繁的写场景（read-mostly）

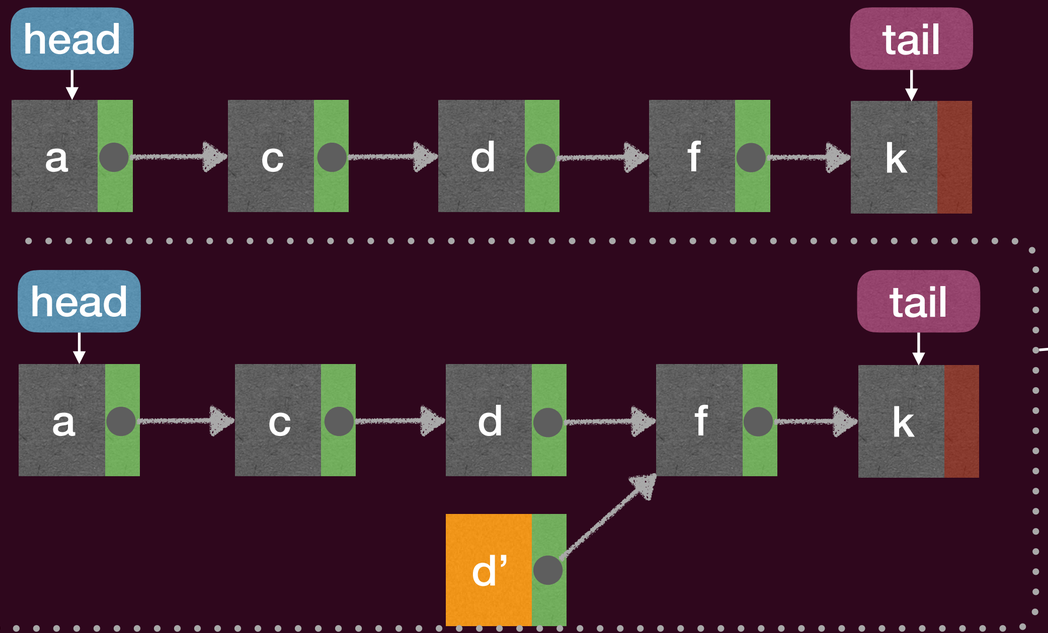
内核的“Read-Mostly”

- 操作系统内核的很多对象都具有“read-mostly”特点
  - 路由表：
    - 网络名称和IP地址的对照表的查询是频繁的，但更新是很少的
  - 用户和组信息
    - 很多操作都需要检查用户和组信息，来判断是否具有一定的权限
    - 但用户和组信息的修改是少的

“Removal”和“Reclamation”两阶段

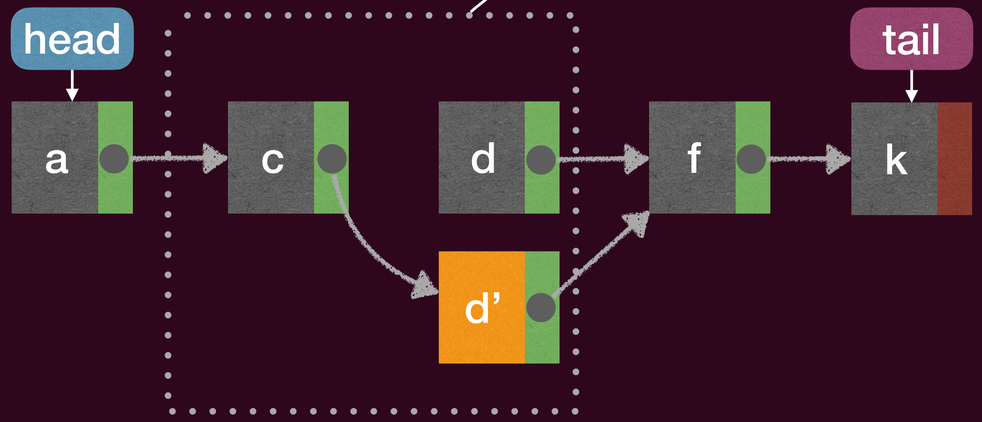
- RCU的核心在于更新数据时，分成“removal”和“reclamation”两个阶段
  - “removal”阶段会删去一个数据结构中某个项的引用（可能伴随将这些引用指向新的项），这个操作可以完全并发的和“读”者的操作进行
  - “reclamation”阶段对旧数据真正意义上清除，这个阶段需要“同步”，不能存在还有读者还在使用这些旧数据
- 更新分为两阶段使得“写”数据不会阻塞读者，对读者非常有利，而写数据只在“reclamation”阶段会增加一些时间成本

例子：更新链表

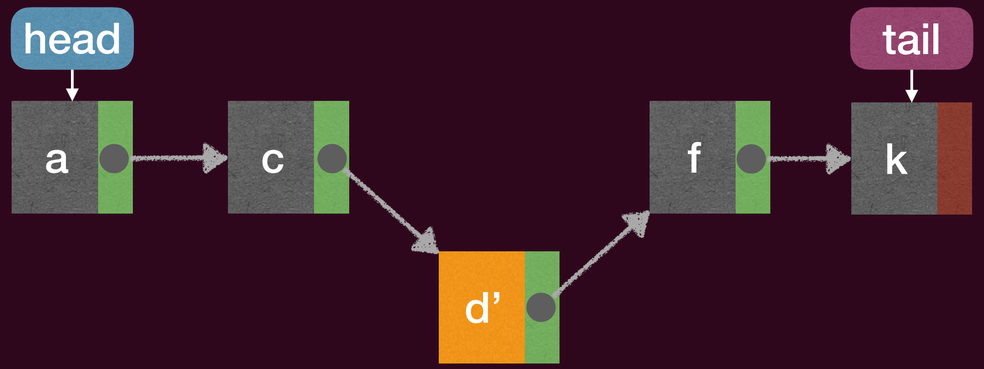


RCU一大特性：需要维护数据的多个版本（有助于不阻塞读者），读者可以读到旧的数据

removal：删去一个数据结构中某个项的引用，并将这些引用指向新的项



reclamation：清除旧数据



RCU的订阅-发布机制：removal的原子性

- 简单的在写者处加锁，而读者处不加锁并不能保证RCU中removal的原子性

```
mutex_t lk;
struct _foo_t {
  int a;
  int b;
  int c;
} foo;

foo *gp = NULL;
```

```
/* 对应写者 */
void foo_update(){
    mutex_lock(&lk);
    new_gp = kmalloc(sizeof(*p),
                     GFP_KERNEL);
    new_gp->a = 1;
    new_gp->b = 2;
    new_gp->c = 3;
    foo *old_gp = gp;
    gp = new_gp;
    mutex_unlock(&lk);
    kfree(old_gp);
}
```

```
/* 对应读者 */
void foo_read(){
    foo *p = gp;
    if(p != NULL)
        do_something(p->a, p->b, p->c);
}
```

在编译器存在优化，以及弱的内存模型下，执行顺序并不可控，读者可以读到非预期的值！
写的原子性就不能保持了！

RCU的订阅-发布机制

- 一个简单的做法就是加入内存屏障，Linux的RCU实现将这个做法直接包裹在一层API中

```
/* 对应写者 */
void foo_update(){
    mutex_lock(&lk);
    new_gp = kmalloc(sizeof(*p),
                     GFP_KERNEL);
    new_gp->a = 1;
    new_gp->b = 2;
    new_gp->c = 3;
    foo *old_gp = gp;
    rcu_assign_pointer(gp, new_fp);
    mutex_unlock(&lk);
    kfree(old_gp);
}
```

```
/* 对应读者 */
void foo_read(){
    foo *p = rcu_dereference(gp);
    if(p != NULL)
        do_something(p->a, p->b, p->c);
}
```

rcu_assign_pointer 发布更新
rcu_dereference 订阅数据
这两个API包裹了内存屏障，保证写和读的顺序！

RCU的宽限期（grace period）

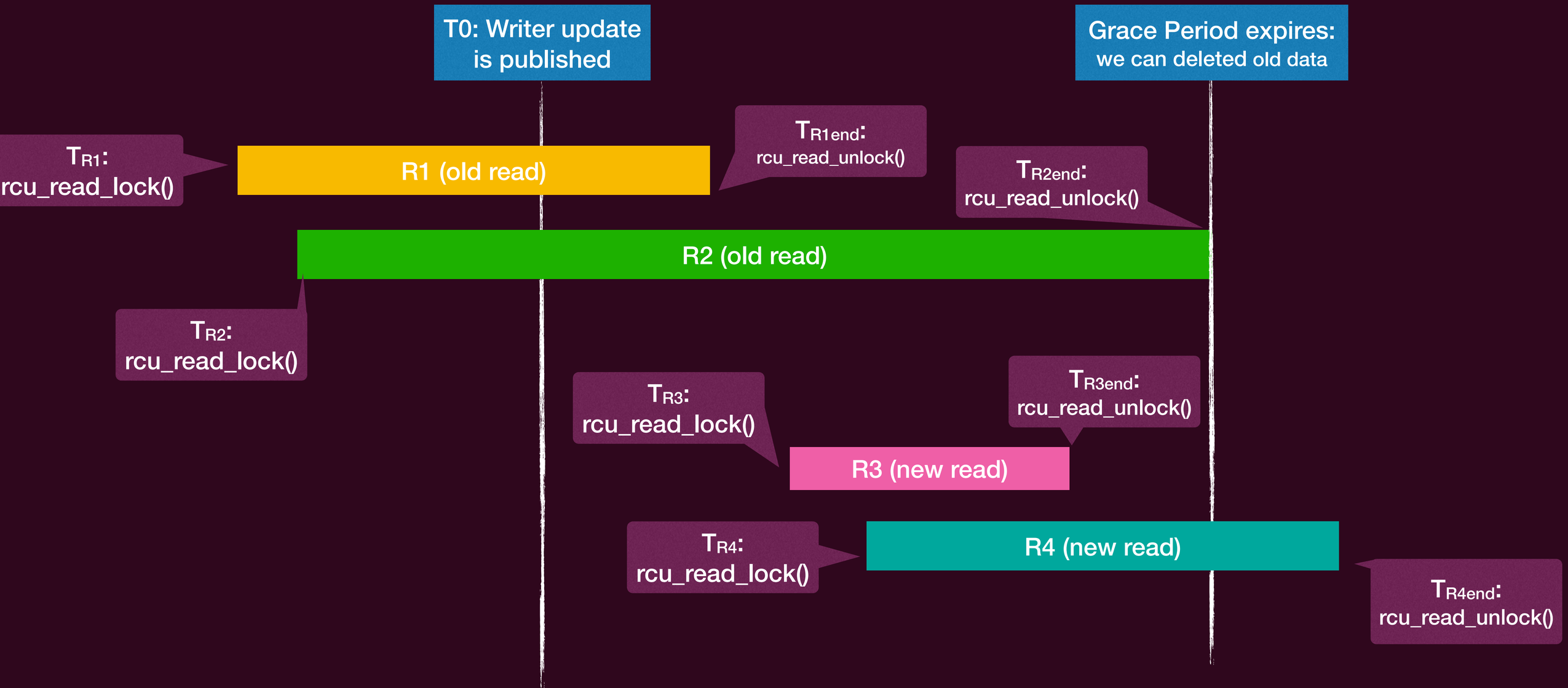
- 第二阶段reclamation什么时候可以真正清除旧数据？所有旧的读者都读完了旧数据！
  - 因此需要“读者”显式地标记读的区间
  - 同时写者需要等到所有的“旧”线程都读结束了，才能释放旧数据，因此需要显式地表达“等”

```
/* 对应写者 */
void foo_update(){
    mutex_lock(&lk);
    new_gp = kmalloc(sizeof(*p), GFP_KERNEL);
    new_gp->a = 1;
    new_gp->b = 2;
    new_gp->c = 3;
    foo *old_gp = gp;
    rcu_assign_pointer(gp, new_fp);
    mutex_unlock(&lk);
    synchronize_rcu();
    kfree(old_gp);
}
```

```
/* 读者 */
void foo_read(){
    rcu_read_lock();
    foo *p = rcu_dereference(gp);
    if(p != NULL)
        do_something(p->a, p->b, p->c);
    rcu_read_unlock();
}
```

synchronize_rcu 等待旧读者结束
rcu_read_lock、rcu_read_unlock标记了读的区间

RCU的宽限期（grace period）



RCU的宽限期（grace period）

- 如何实现宽限期的机制？（注意不能用锁或条件变量，否则RCU就没意义了！）
  - 经典的RCU实现(简化)：
    - 每个读者进入read-sided的临界区前执行关中断 （`rcu_read_lock()`）
    - 读者离开read-side的临界区前执行开中断 （`rcu_read_unlock()`）
  - 这样，写者只要侦测到所有CPU都做完一次context-switch， 就可以安全的释放旧数据了

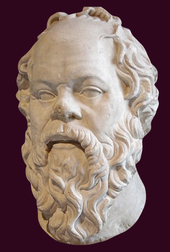
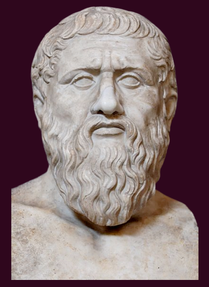
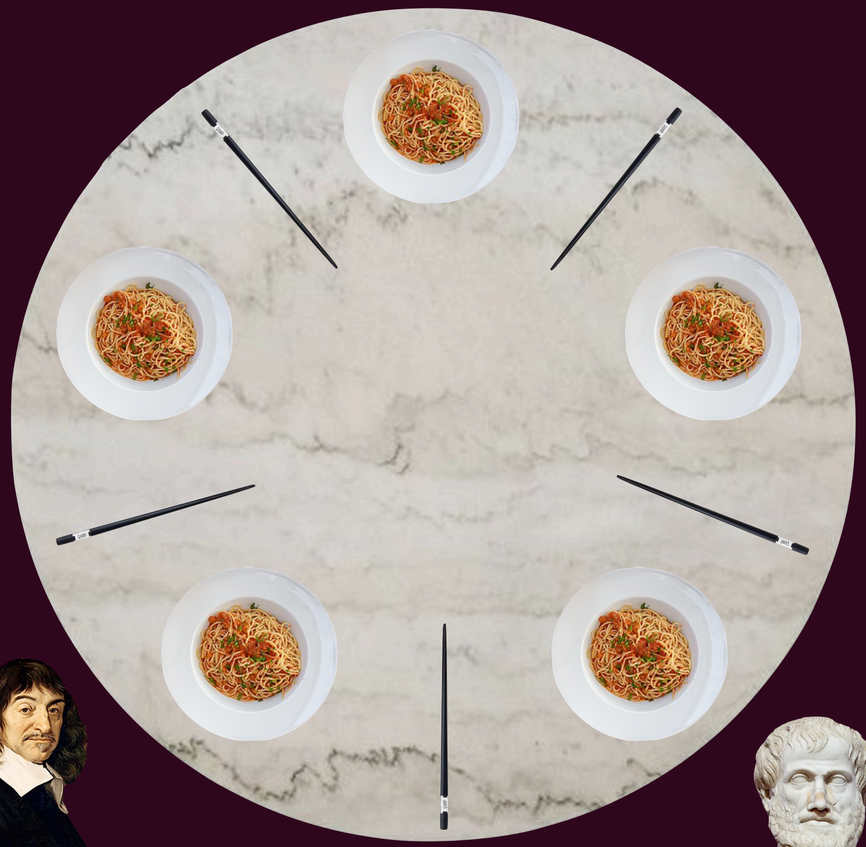
RCU的宽限期（grace period）

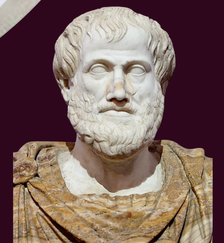
- 如果写者不想阻塞
  - 那么可以通过`call_rcu()`注册一个回调函数，这样可以免除被等待宽限期
  - 所有旧的读者读完了，会自动调用`call_rcu()`所注册的函数
- 用户态能实现RCU吗?
  - User-space RCU

哲学家就餐问题

哲学家就餐问题

- Dijkstra早1965年给出的经典同步问题（Hoare修改了描述得到如今的版本）
- 五位哲学家围坐在一张圆形餐桌旁，做以下两件事情之一：吃饭或者思考。
- 每位哲学家之间各有一只筷子。哲学家必须同时得到左右手的筷子才能吃东西。
- 哲学家们怎么办才能比较合理的吃到饭?
  - 不会死锁、没有饿死、并发度高?

哲学家就餐问题

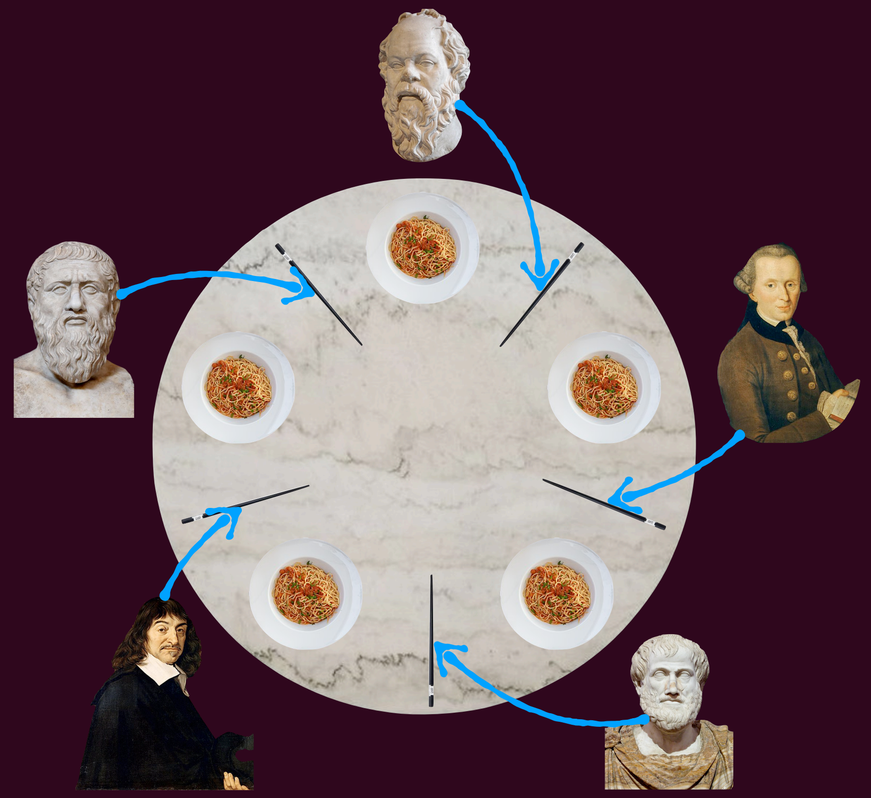
解决方案1（错误方案）

- 每个哲学家都先拿起自己左边的筷子，然后再拿起右边的筷子，然后吃！

```
#define N 5

sem_t forks[N];
int left(int i){ return i;}
int right(int i){ return (i+1)%N;}

void philosopher(int i){
  while(1){
    think();
    P(forks[left(i)]);// Pick up left fork
    P(forks[right(i)]);//Pick up right fork;
    eat();
    V(forks[left(i)]);// Put down left fork
    V(forks[right(i)]);//Put down right fork;
  }
}
```



所有人同时都拿到左边的筷子，然后等待右边的筷子：死锁

解决方案2

- 问题在于自己拿不到筷子还不断持有筷子，尝试拿不到时放下筷子

```
 //尝试P操作
int tryP(sem_t *s){
    //sem_wait的非阻塞版本，如果返回0，正常等到条件
    //否则不会阻塞，而是返回-1
    return sem_trywait(s);
}
void philosopher(int i){
  while(1){
    think();
    while(1){
        P(forks[left(i)]);// Pick up left fork
        int ret = tryP(forks[right(i)]);//Try to pick up right fork;
        if (ret!= 0){
             V(forks[left(i)]);//  Put down left fork
             sleep(sometime);
        }else{
            break;
        }
    }
    eat();
    V(forks[left(i)]);// Put down left fork
    V(forks[right(i)]);//Put down right fork;
  }
 }
```

一个不好的运气会导致所有哲学家同时放下筷子，又同时拿起左手的筷子，再次同时放下... 饿死!

死锁和饿死

- 死锁(Deadlock)和饿死(starvation)都关乎活性(**liveness**) — 死锁和饿死并没有违背安全性!
- 饿死即为一个线程在有限时间内无法行进
- 死锁是一类特殊的“饿死”，其达成的条件是多个线程形成一个等待环，一个线程的行进需要环内的另外一个线程坐某个动作：显然环状意味着这个等待条件永远无法发生
- 死锁一定饿死，饿死并不一定死锁（比如运气不好，一直被其他线程抢占临界区）

解决方案2—改进

- 每个线程睡眠的时间随机！

```
void philosopher(int i){
  while(1){
    think();
    while(1){
        P(forks[left(i)]);// Pick up left fork
        int ret = tryP(forks[right(i)]);//Try to pick up right fork;
        if (ret!= 0){
             V(forks[left(i)]);//  Put down left fork
             sleep(randomtime);
        }else{
            break;
        }
    }
    eat();
    V(forks[left(i)]);// Put down left fork
    V(forks[right(i)]);//Put down right fork;
  }
}
```

已经可以说解决问题了！但是在某些特别安全攸关的场景可能不那么可靠

解决方案3

- 哲学家拿起筷子就已经发生数据竞争了，不如一开始就上锁！

```
ticket_t turn;
lock_init(&turn);

void philosopher(int i){
  while(1){
    think();
    ticket_lock(&turn);
    P(forks[left(i)]);// Pick up left fork
    P(forks[right(i)]);//Pick up right fork;
    eat();
    V(forks[left(i)]);// Put down left fork
    V(forks[right(i)]);//Put down right fork;
    ticket_unlock(&turn);
  }
}
```

一次只有一个哲学家吃饭！并发度不够！
5个哲学家的情况下，最大支持多少个哲学家同时吃饭?

解决方案3—改进

- 每次就一个人并发度不够，每次5个人会死锁，那么每次 5 - 1个人?

```
#define N 5
sem_t capacity = N - 1;

void philosopher(int i){
  while(1){
    think();
    P(capacity);
    P(forks[left(i)]);// Pick up left fork
    P(forks[right(i)]);//Pick up right fork;
    eat();
    V(forks[left(i)]);// Put down left fork
    V(forks[right(i)]);//Put down right fork;
    V(capacity);
  }
}
```

根据鸽巢原理，一定有一个人拿到两个筷子！从而可以吃饭

解决方案4

- 一个更加简单的方案（无需额外互斥锁）：给筷子编号！总是先拿编号小的！

```
void philosopher(int i){
  while(1){
    think();
    if (left(i) < right(i)){
        P(forks[left(i)]);// Pick up left fork
        P(forks[right(i)]);//Pick up right fork;
    } else{
        P(forks[right(i)]);// Pick up right fork
        P(forks[left(i)]);//Pick up left fork;
    }
    eat();
    V(forks[left(i)]);// Put down left fork
    V(forks[right(i)]);//Put down right fork;
  }
}
```

筷子的编号对应着哲学家编号i

阅读材料

- [OSTEP] 第30，31章
- What is RCU, Fundamentally?
- Is Parallel Programming Hard, And, If So, What Can You Do About It?