互斥：进阶
Mutual Exclusion (Mutex): 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

自旋锁的一个改进

- 当一个线程在自旋等待（也叫忙等待，busy waiting）时，它一定能够最终进入临界区吗?
  - 不一定，如果一直有其他线程要进入临界区，并且这些其他线程一直被优先调度进入临界区（我们不能假定调度策略），那这个线程就可能会一直等在那里（违背了有界等待）
  - 解决方法也很简单：排队！

自旋锁的一个改进

- 方法：每次尝试进入临界区就拿一个“号”（下一个尝试的“号”加一），等待“叫号”

```
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
} lock_t;

lock_t flag;

void lock_init() {
    flag.ticket = 0;
    flag.turn   = 0;
}
```

自旋锁的一个改进

- 排号自旋锁（Tick Lock）的加锁和解锁, 通过原子的 “fetch_and_add”实现

```
void lock() {
    int myturn = 1;
    //atomic fetch-and-add
    //equal to myturn = __sync_fetch_and_add (&flag.ticket, 1);
   // 将1加到flag.ticket，  并返回flag.ticket的“旧”值给myturn
    asm volatile (
            "lock xaddl %0, %1"
            : "+r" (myturn),  "+m" (flag.ticket)
            :
            : "memory", "cc"
    );
    while (flag.turn != myturn)
        ; // spin
}
```

```
void unlock() {
    int value = 1;
    asm volatile (
       "lock xaddl %0, %1"
       :”+r"(value),
        "+m" (flag.turn)
       :
       : "memory", "cc"
    );
}
```

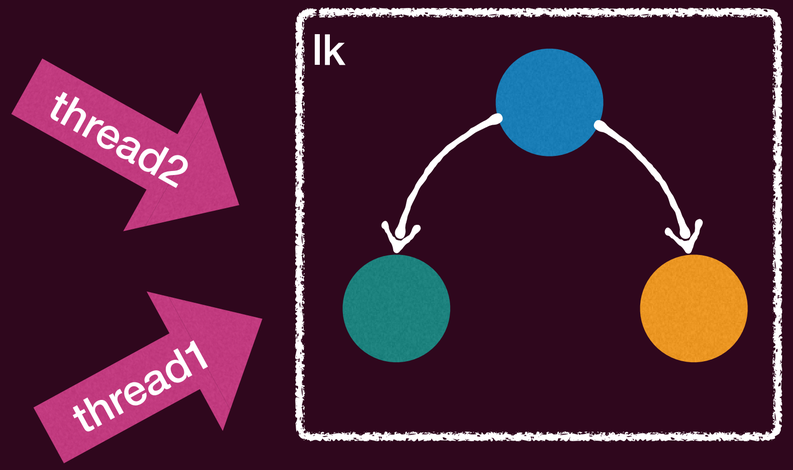
GCC内建了很多这样的原子操作

- type __sync_fetch_and_add (type *ptr, type value)
- bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval)
- type __sync_lock_test_and_set (type *ptr, type value)
- void __sync_lock_release (type *ptr)
- 更多可见:
- https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html
- https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html
- 使用GCC的内建函数可以编写可以跨平台的“原子”操作，而不只是X86

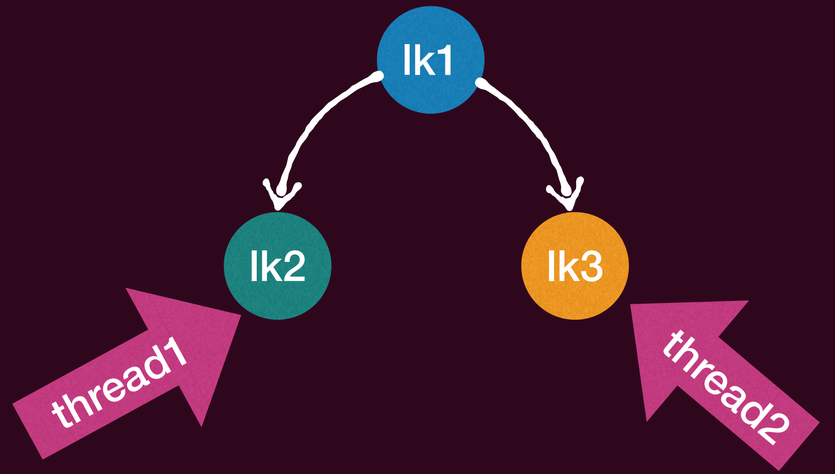
一把大锁会锁住所有？

- 并不是所有线程都“彼此”需要互斥，我们应该给需要彼此互斥的线程集他们独有的“锁”，更加“细粒度”的锁可以提高并发性能！

```
void lock();
void unlock();
```




```
typedef struct {
  ...
} lock_t;
void lock(lock_t *lk);
void unlock(lock_t *lk);
```



更细粒度的锁

- 修改之前的自旋锁实现方法：对参数所指向的锁（而不是全局的一把大锁）进行加锁和解锁操作

```
void lock(lock_t *flag) {
    int myturn = 1;
    //atomic fetch-and-add
    //equal to myturn = __sync_fetch_and_add (&flag.ticket, 1);
   // 将1加到flag.ticket,  并返回flag.ticket的“旧”值给myturn
    asm volatile (
            "lock xaddl %0, %1"
            : "+r" (myturn),  "+m" (flag->ticket)
            :
            : "memory", "cc"
    );
    while (flag->turn != myturn)
        ; // spin
}
```

```
void unlock(lock_t *flag) {
     int value = 1;
     asm volatile (
        "lock xaddl %0, %1"
        :”+r"(value),
         "+m" (flag->turn)
        :
        : "memory", "cc"
    );
}
```

⚠️有了自旋锁就真的互斥了吗?

- lock和unlock没有“真正”意义上的保护临界区的共享资源
  - 互斥锁的正确性依赖于你是“理智”的“合作者”
    - lock和unlock必须要按照正确方式才能形成保护!

```
T1: spin_lock(&lk); sum++; spin_unlock(&lk);
T2: spin_lock(&lk); sum++; spin_unlock(&lk);
T3: sum++;
```

一个不正确的访问整个逻辑就错了

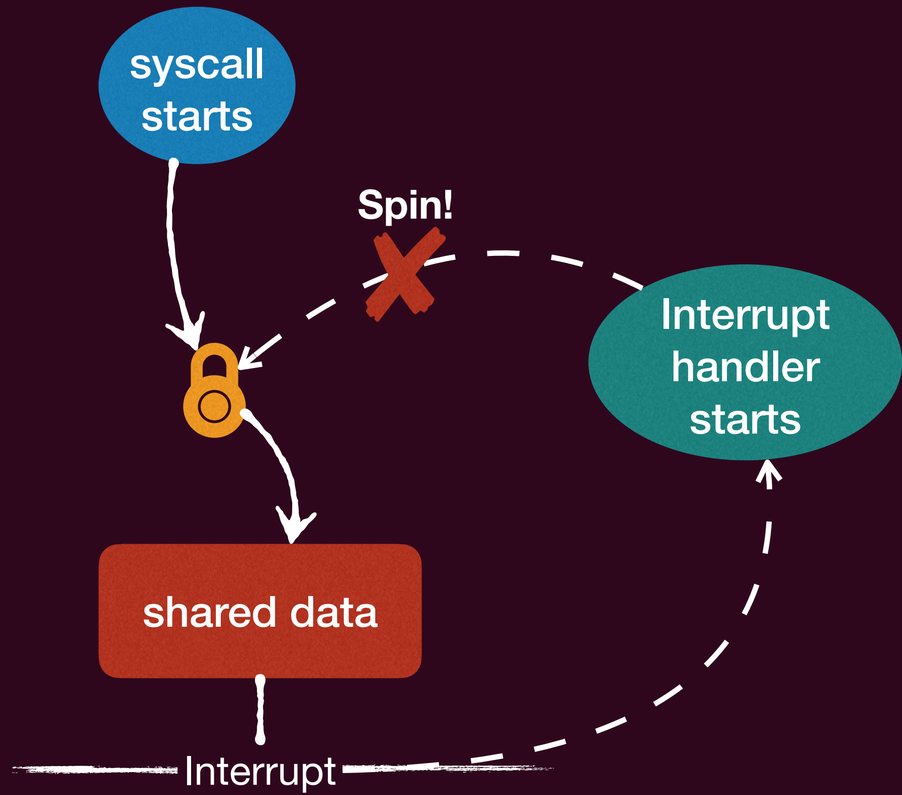
在内核中实现自旋锁的问题

内核中的自旋锁

- 内核的实现中会出现很多需要访问共享资源的情况（毕竟，操作系统就是资源的管理者！）
- 因此内核中使用互斥锁的情况非常普遍
  - 比如线程利用系统调用访问共享的终端输出时(write)，OS会用锁来互斥各个线程
- 内核可能会在各个部分用到自旋锁，除了系统调用程序外，还有中断处理程序中也可能用到(用户程序没有这个部分)
  - 然而，这个部分会造成非常大的麻烦！

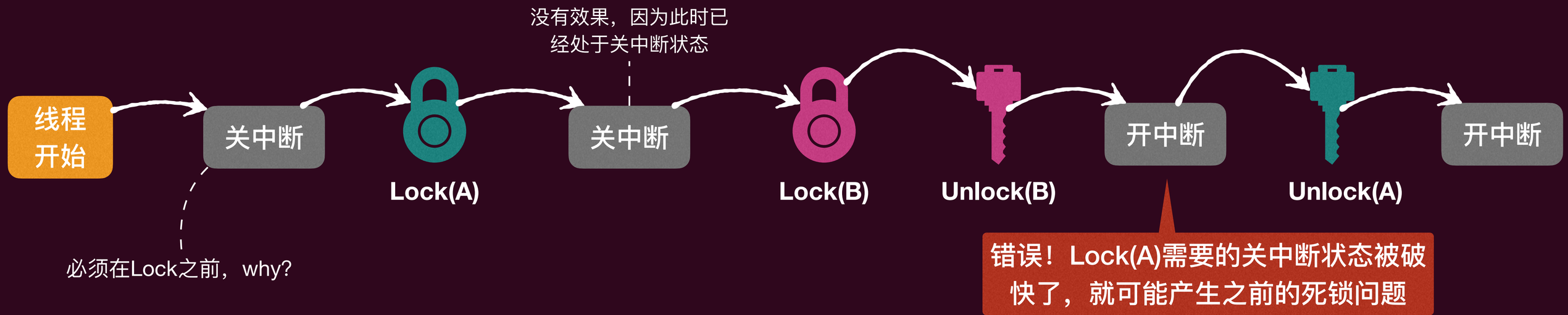
内核中的自旋锁

- 考虑如下情况：
  - 一个线程利用系统调用访问一个共享变量，kernel在访问这个共享变量时上了锁
  - 此时一个中断发生了，CPU强制转向中断处理程序，这个中断处理程序也需要访问这个共享变量，因此尝试获得锁，但是发现这个锁已经被持有了，因此自旋等待
  - 中断处理程序的优先级一般很高，要高过系统调用
  - 因此该中断处理程序就会一直等待一个不再可能发生的事情



解决方法

- 一个尝试：在自旋锁之前关中断
  - 然后释放锁的时候开中断🤔
- 这个尝试是**错误的**！如果在自旋之前就已经关中断了，解锁就打开中断就会破快在这次自旋之前的中断状态！



xv6里自旋锁的实现

- 因此我们需要保存自旋之前的中断状态（打开或关闭），然后在解锁时恢复这个状态.
- xv6(MIT 开发的一个教学用的完整的类Unix 操作系统)给出了很好的实现

```c
void spin_lock(spinlock_t *lk) {
    // Disable interrupts to avoid deadlock.
    push_off();
    // This is a deadlock.
    if (holding(lk)) {
        panic("acquire %s", lk->name);
    }
    // This our main body of spin lock.
    int got;
    do {
        got = atomic_xchg(&lk->status, LOCKED);
    } while (got != UNLOCKED);
    lk->cpu = mycpu;
}
```

```c
void spin_unlock(spinlock_t *lk) {
    if (!holding(lk)) {
        panic("release %s", lk->name);
    }
    lk->cpu = NULL;
    atomic_xchg(&lk->status, UNLOCKED);
    pop_off();
}
```

```c
typedef struct {
    const char *name;
    int status;
    struct cpu *cpu;
} spinlock_t;
```

xv6里自旋锁的实现

- xv6这个实现里，push_off()记录中断关闭的次数，pop_off()记录想要打开中断的次数（只有当该次数等于中断关闭的次数才能真正去打开中断）

```
// it takes two pop_off()s to undo two
//push_off()s.
void push_off(void) {
    //record previous state of interrupt
    int old = ienabled();

    struct cpu *c = mycpu;

    //disable the interrupt
    iset(false);

    if (c->noff == 0) {
        c->intena = old;
    }
    c->noff += 1;
}
```

```
void pop_off(void) {
    struct cpu *c = mycpu;
    // Never enable interrupt when holding a lock.
    if (ienabled()) {
        panic("pop_off - interruptible");
    }
    if (c->noff < 1) {
        panic("pop_off");
    }
    c->noff -= 1;
    if (c->noff == 0 && c->intena) {
        iset(true);
    }
}
```

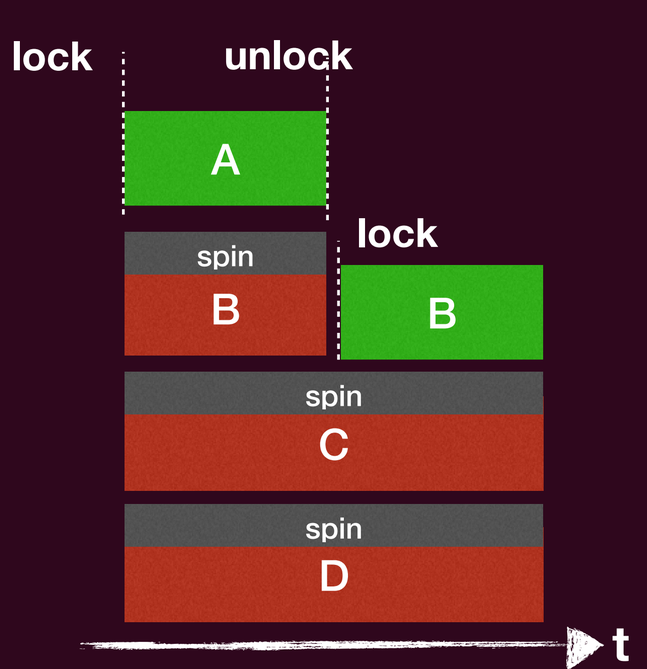
应用程序里的使用互斥锁问题

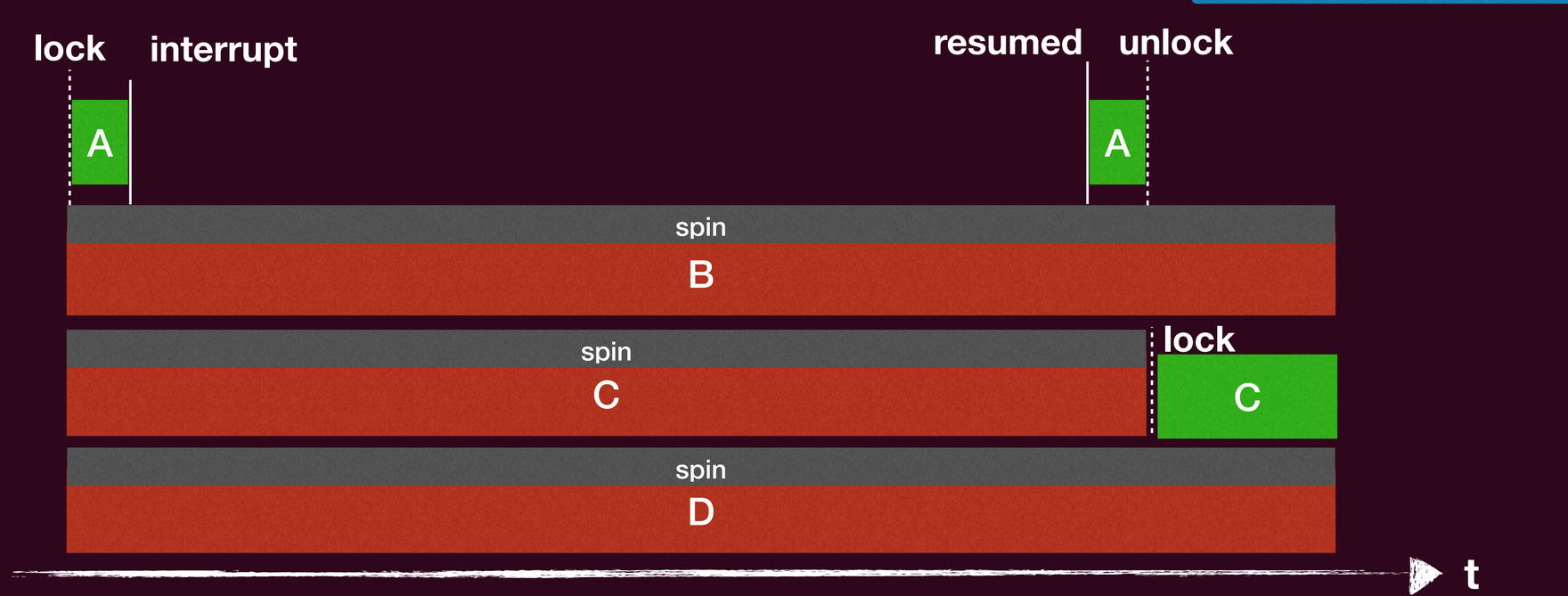
自旋锁本身存在的问题

- 性能问题：
  - 除了进入临界区的线程，其他处理器上的线程都在空转
    - 如果临界区执行时间过长（用户线程的常态），其他线程浪费的CPU越多
    - 此外，如果发生中断将临界区的线程切出去了，计算资源浪费更加严重

内核中的临界区一般都为“短”临界区

然而用户态无法通过关闭中断来解决问题





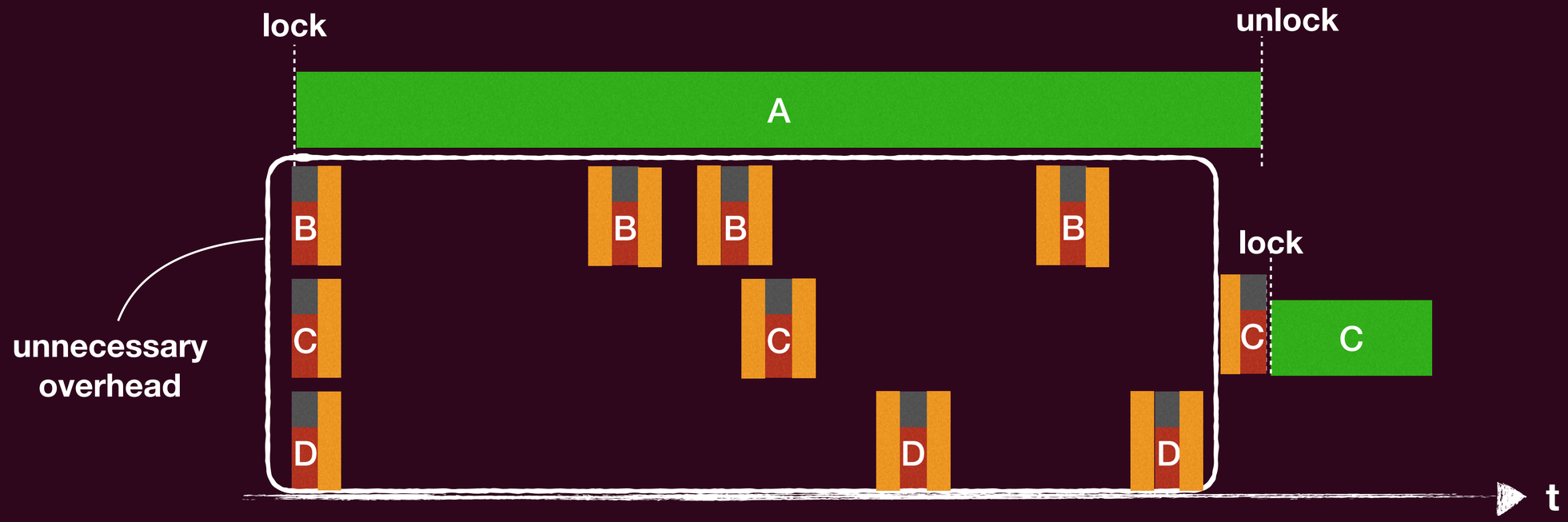
一个简单解决方案：yield！

- 利用系统调用sched_yield()直接让出cpu，让其他线程获得CPU使用

```
void yield_lock(spinlock_t *lk) {
  while (xchg(&lk->locked, 1)) {
    //a wrapper of sched_yield()
    syscall(SYS_yield); // yield() on AbstractMachine
  }
}
void yield_unlock(spinlock_t *lk) {
  xchg(&lk->locked, 0);
}
```

直接yield的问题

- yield只是暂时让出CPU，该线程还处在“ready”的阶段，随时可以被再次调度。
- 在获得锁之前，反复的“被调度—>让出CPU”会带来大量不必要的context-switches



解决方案

- 用户使用和释放锁应该和OS调度程序配合：
  - mutex_lock(&lk): 试图获得lk，如果失败(lk已被持有)，利用系统调用阻塞该线程（此时不是就绪态，无法被调度了），让出CPU并将其加入等待锁的队列之中。否则，成功获得锁进入临界区。
  - mutex_unlock(&lk): 释放锁，如果等待该锁的队列里有线程就利用系统调用选择一个唤醒，使其变成就绪态（ready），从这个等待队列删除，并进入就绪的队列，可以被再次调度。
- 操作系统需要对一个锁维持一个与其相关的队列

当然，队列操作（如增加一个等待线程）需要内核的自旋锁保护

具体细节没那么简单

- 下面的实现有问题吗? 🤔

```
void mutex_lock(spinlock_t* lk) {
    int got;
    do{
        got = atomic_xchg(lk->status, LOCKED);
        if (got != UNLOCK){
            //将当前线程加入等待队列，并标记为阻塞，释放cpu
            wait(lk->wait_list);
        }else{
            break;
        }
    } while (1);
}
```

```
void mutex_unlock(spinlock_t* lk) {
    atomic_xchg(lk->status, UNLOCKED);
    if (!is_empty(lk->wait_list)){
        //等待队列中的一个线程移出队列，标记为就绪
        wakeup(lk->wait_list);
    }
}
```

具体细节没那么简单

- 考虑如下场景：

时间

Thread 1

```
got != UNLOCK
```

```
wait(lk->wait_list);
```

Thread 2

```
atomic_xchg(&lk->status, UNLOCKED);
if (!is_empty(lk->wait_list)){
     wakeup(lk->wait_list);
}
```

此时等待队列没有线程，wakeup丢失！

Linux Futex (“Fast Userspace muTEX”) syscalls

- Linux提供了如下两个系统调用：
  - `futex_wait(int *address, int expected)`
    - 首先原子的test此时address指向的值和期待的值是否相等，相等才会将线程阻塞，否则立即返回给用户线程，使其可以立马再次尝试lock!
  - `futex_wake(int *address)`
    - 唤醒一个等待address指向的锁的线程

Linux Futex (“Fast Userspace muTEX”) syscalls

- 基于这两个系统调用，可以实现如下互斥锁：

```
#define UNLOCK  0
#define ONE_HOLD  1
#define WAITERS  2

void mutex_unlock(spinlock_t* lk){
    //state can only be ONE_HOLD or WAITERS
    if(atomic_dec(lk) != ONE_HOLD){
        //has more than one waiters
        lk = UNLOCK;
        futex_wake(lk);
    }else{
        //No Waiters!
        return; //the fast path unlock!
    }
}
```

```
void mutex_lock(spinlock_t* lk) {
    //return old value of state,
    //and if lk == UNLOCK, set state = ONE_HOLD - uncontested!
    int c = cmpxchg (lk, UNLOCK, ONE_HOLD);

    if (c != UNLOCK){
    //previous state is either ONE_HOLD or WAITERS
        do {
            if (c == WAITERS
                // if previous state is either ONE_HOLD
                // now it has one more waiter!
                // AND it is possible the lock is released now!
                || cmpxchg (lk, ONE_HOLD, WAITERS) != 0)
                futex_wait (lk, WAITERS);
        }
        // repeating check whether the lock is released!
        while ((c = cmpxchg (lk, UNLOCK, WAITERS))!= 0);
    }
    else{
        return; //the fast path lock!
    }
}
```

VERY TRICKY!

Linux Futex (“Fast Userspace muTEX”) syscalls

- 正确性的简单解释：
  - mutex_lock函数只会在成功锁住lk后才会返回
    - 获得lk的线程本身会将lk原子的设为1，而其他等待线程将会将lk原子的设为2，这使得除获得锁外的其他线程不可以在临界区，只有获得锁住线程在退出临界区调用mutex_unlock才会将lk设为0： 这保证了上锁的正确性！
  - 等待的线程能够被唤醒，因为只要有等待的线程在wait，那么此时的lk一定为2！而unlock就一定会将某个等待的线程唤醒！
  - 由于unlock函数在wake_up之前先将lk设为0，因此上述即使发生了上述丢失wakeup的调度，由于futex_wait的特性，那个线程在调用futex_wait时会发现lk的值变了，因此不会休眠！

NOTE：利用系统调用进行加锁的问题

- 虽然避免了自旋浪费CPU，但每次进行Lock/Unlock都要陷入内核（这当然需要额外的开销，比如上下文切换）
  - 只有内核才能让线程“阻塞”、“yield”，线程无法独立完成这样的操作
- 但是其实真实的workload中，多个线程抢一个锁的事件发生的频率并不大，很多时候往往是一个线程加一个锁进行保护
  - 这个时候使用之前的spinlock会更加快速！因为其不用陷入内核，也不会空转

Linux的Futex就是这样的两阶段锁

- 操作系统将两者优点结合起来，实现了一个两阶段锁
  - Fast Path: 自旋一次
    - 一次原子指令，成功直接进入临界区
  - Slow Path: 自旋失败
    - 按照情况利用系统调用 futex_wait，阻塞自己

真实的futex有更多的优化和细节：Futexes are tricky by Ulrich Drepper, LWN: A futex overview and update

Posix的实现方案

- pthread mutex lock实现了上述需求(很多线程的争抢下依然能保持很好的性能)，使用方法和自旋锁一致，大部分情况下（在有操作系统的情况）使用它即可！

```c
#include <pthread.h>

pthread_mutex_t mutex;
int pthread_mutex_init(pthread_mutex_t *restrict mutex, NULL);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

并发数据结构

线程安全(thread-safe)的数据结构

- 线程安全的数据结构指的一个数据结构可以被多个线程并发的访问
  - 也被称为并发数据结构
- 要达成这样的数据结构一般我们需要在访问和更新该数据结构时上锁（一把或多把）
- 最简单的做法：一把大锁（One Big Lock）！所有访问都串行化（实际上，早期的Linux就是这么做的，Big Kernel Lock（BKL），简单，正确！）
  - 但在多处理器时代，可能会造成性能瓶颈

一个例子：并发的Counter

- 下面是一个允许并发线程访问的数据结构counter_t，其支持的操作如increment、decrement和get由于都受到互斥锁的保护，因此都是线程安全的

```c
typedef struct __counter_t {
    int value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    Pthread_mutex_init(&c->lock, NULL);
}
```

```c
void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}
void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}
int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    int rc = c-> value;
    Pthread_mutex_unlock(&c->lock);
    return rc;
}
```

但如果想要更高的性能?

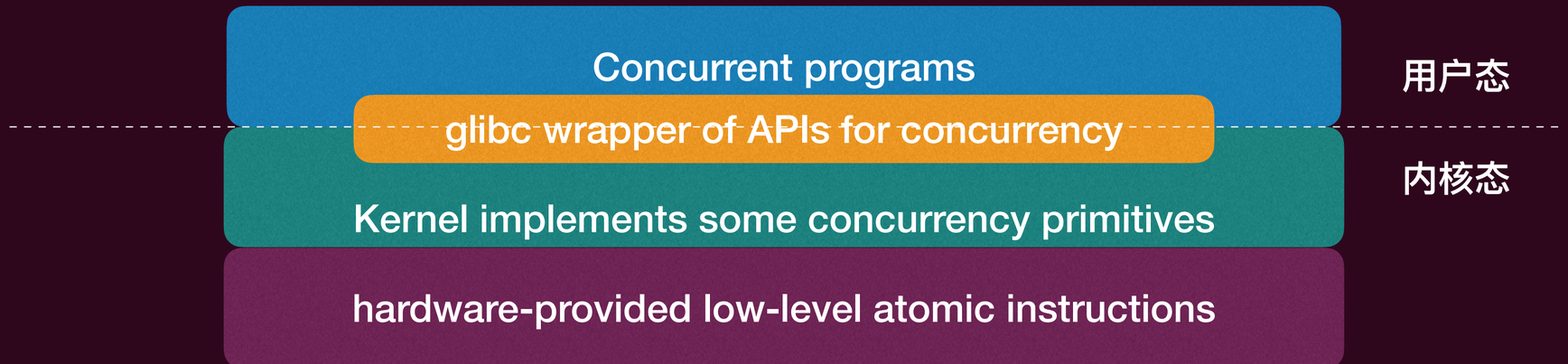
- 一个可行的解决办法：
  - 每个CPU维持一个本地的counter，所有CPU共享一个全局counter
  - 本地的counter只需要本地的一个锁保护（per cpu），全局的counter访问需要一个全局的大锁
  - 本地的counter可以选择每过一段时间（而不是每次）进行更新全局的counter，从而增加并发度

时间

L_1	L_2	L_3	L_4	G
0	0	0	0	0
1	0	1	1	0
2	0	2	1	0
3	0	3	2	0
4	1	3	3	0
5 → 0	1	3	4	5 (from L_1)
0	2	4	5→0	10 (from L_4)

层级化的并发解决方案

- 可以看到，我们的并发解决方案是层级的
  - 并不是所有并发都是硬件提供的，也不是都是软件
  - 而是，硬件提供了基本的原子指令（如cmpxchg），操作系统提供了一些并发(同步)的原语（如futex_wait, futex_wake），库函数包裹一些好用的APIs(如pthread_mutex_lock, pthread_mutex_unlock)，最后用户使用这些原语来构建正确和高效的并发程序



总结

- 软件上实现“互斥”很难，而且在现代计算机系统下也不正确
- 通过硬件支持的原子指令可以实现自旋锁，从而正确实现互斥
- 然而实际的互斥锁有更多的考量：
  - 在内核中的互斥锁要考虑中断带来的麻烦
  - 而在用户态由于临界区过长，自旋出现性能问题，而解决这个问题需要操作系统和用户态共同完成一个wait-wakeup的原语，正确的实现同样不简单！

阅读材料

- [OSTEP] 第25、26、27、28、29章