XV6 线程同步机制 IPC

by 1801210840 姜慧强

在 XV6 中使用自旋锁 SpinLock 来完成线程间同步过程。

自旋锁直观上看就是在循环等待临界区资源，循环等待会阻塞后续程序运行，代价高。

但自旋锁在很多地方都有应用。比如 Linux kernal 、 Nginx 等, 所以说自旋锁的效率和其的应用场景有很大关系。

> *究竟什么时候我们应该使用 SpinLock？*

首先，要注意的是自旋锁只适用于**多核心**状态下（这个多核心指的是当前进程可用的核心数 > 1），比如说你是一个 8 核 Mac，但是你在一个限制 1 核的 Docker 环境中用 SpinLock，卵用也没有！！！

本质上，SpinLock 之所以有效就是假定，当前存在另外一个 CPU 核心正在使用你所需要的资源。

其次，SpinLock 之所以在一些场景下很高效是因为旋等消耗的时钟周期远小于上下文交换产生的时间。

在 Mutex 睡眠等待的过程中首先是尝试上一次锁，如果不行的话就通过调度算法找到一个优先级更高的 Thread，然后才是保存寄存器，写回被修改的数据，然后才是交换上下文。

可以看到这个代价是十分大的，而且交换上下文的代价是要 ✖ 2 的。一般来说，这个代价，在几千~几十万时钟周期。而一个 4GHz 的 8 代处理器，一个时间周期=0.25ns。交换一次的代价还是挺可观的。

所以我们使用 SpinLock 的时候就需要保证我们的临界区代码，能够在这个时钟周期之类完成所有任务。

所以一般 spinLock 等待的代码不会太长，一般几行（具体需要看芯片和编译环境），更不可能是 I/O 等待型的任务。（在 XV6 中，关于文件系统的操作都单独使用基于 SpinLock 的 SleepLock）

然后，其实 SpinLock 更适合系统态，不太适合用户态。因为你用户态没法知道有没有另外一个 CPU 在处理你所需要的资源。而且 SpinLock 并不适合多线程抢占一个资源的场景，比如说开了 60 个线程抢占一个一个内存资源，这个竞争、等待的代价就是超级大的一个数。

SPINLOCK IN XV6

XV6 其实是一个很 Unix 的教学操作系统，通过对 XV6 代码的阅读，我们其实能够以更少代价来了解 Unix 是怎么做的。

SpinLock 在 XV6 中定义在**<spinlock.h>**和**<spinlock.c>**两个文件中，实际上代码量不过 100 行，是很好的分析案例。

```
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

首先, SpinLock 类中用了一个 unsigned int 来表示是否被上锁，然后还有 lock 的名字，被哪个 CPU 占用，还记录了系统调用栈(这个实际上就是完全用来调试用的，当然一个健壮的 OS 需要方方面面考虑到)。

```
void
acquire(struct spinlock *lk)
{
  pushcli();       // 关中断
  if(holding(lk)) // 如果已经被当前CPU获得了, 则返回
    panic("acquire");

  // xchg -> 交换两个变量
  // return &lk->locked
  // and lk->locked = 1
  // 自带锁总线机制,
  // 非阻塞
  while(xchg(&lk->locked, 1) != 0)
    ;

  lk->cpu = cpu;     // 记录上锁的CPU
  getcallerpcs(&lk, lk->pcs);
}
```

当我们需要去获得这个锁的时候，它会先去关中断，再去检查这个锁有没有被当前的 CPU 占用，然后是一个尝试上锁的循环，最后是标记被占用的 CPU，记录系统调用栈。

总的来说思路比较清晰，我们来看一下具体实现细节。

```
void
pushcli(void)
{
  int eflags;
  eflags = readeflags(); // eflags 取出寄存器值
  cli();                 // 禁止中断 (非阻塞, 不管中断是否已经被禁止都会执行cli)
  if(cpu->ncli++ == 0)   // cli 嵌套数是否为0
    cpu->intena = eflags & FL_IF;
  // #define FL_IF           0x00000200      // Interrupt Enable
  // 如果已经禁止中断了, 那么eflags 就会发现 FL_IF 未设置
  // 如果中断又变成可用了, 则标记intena = 0
}
```

pushcli 函数是用来实现关中断过程的一个函数，先会去调用 readeflags 这个函数来读取堆栈 EFLAGS, 然后调用 cli 来实现关中断。 如果是第一个进行关中断的（嵌套关中断数是 0），则还会去再 check 一下 elfags 是不是不等于关中断常量 `FL_IF` 。

而 readeflags()和 cli()这两个函数都是通过内联汇编来实现的。

```
static inline uint
readeflags(void)
{
  uint eflags;
  // asm volatile 内联汇编
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  // pushfl -> 压入EFLAGS堆栈, 然后popl, 并把输出丢给1号表达式
  // 也就是 eflages

  return eflags;
}
```

readeflags, 就是先去把 efalgs 寄存器当前内容保存到 EFLAGS 堆栈中，然后把 EFLAGS 堆栈中的值给到 eflags 变量。

```
static inline void
cli(void)
{
  asm volatile("cli");
}
```

而 cli()就很简单粗暴的调用 cli 汇编指令。

```
// Check whether this cpu is holding the lock.
int
holding(struct spinlock *lock)
{
  return lock->locked && lock->cpu == cpu;
}
```

当我们关中断之后，会去检查当前 CPU 有没有持有这个 SpinLock。

```
void
panic(char *s)
{
  printf(2, "%s\n", s);
  exit();
}
```

如果已经持有这个 SpinLock 则会退出。

当完成这一系列常规操作之后，才是最关键的获取锁的步骤。

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;
  // The + in "+m" denotes a read-modify-write operand.
  // avoid Out of Order -> Pause
  // Transformer to cpu language = 10~100+ NOP
  // X86 USE _sync_synchronize() to do this thing.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

他会去调用 xchg()这个内联汇编函数。会去执行 `lock; xchgl %0, %1` 这个汇编代码。

最关键的实际上是 `xchgl` 这个指令, 从效果上看 xchgl 做的是一个交换两个变量，并返回第一个变量这个事情。

实际上这个指令，首先是一个原子性的操作，当然，这我们可以理解毕竟是多核状态，如果有好几个 CPU 核心来抢占同一个 SpinLock，需要保证互斥性, 需要排他来访问这块内存空间。

其次 xchgl 是一个 Intel CPU 的锁总线操作，对应到汇编上，就是自带 lock 指令前缀，就算前面没有加 `lock;` 这个操作也是原子性的。

其次，既然是锁总线操作，就有可能失败，这个命令式非阻塞的，每次执行只是一次尝试，所以这个 while 就说的通了。通过循环尝试上锁，来实现旋锁机制。

最后，这条命令还用到了一个 read-modify-write 的操作，这个操作，主要是因为在现代 CPU 中基本上都会使用 Out of Order 来对指令执行进行并行优化。

但是我们这个加锁的过程是一个严格的时序依赖过程，我们必须保证，前面一个 CPU 加上了锁，后面 CPU 来查询的时候都显示已经上锁了。即 read-modify-write 顺序执行。

在 XV6 和 后面分析的 Linux 实际上都是用 `__sync_synchronize` 来实现这个过程的。

至此，XV6 SpinLock 最关键的部分就解读完了。

当已经拿到 SpinLock 的时候，就回去更新 cpu，call stack 来给 DeBug 使用。

```
// Record the current call stack in pcs[] by following the %ebp chain.
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  // #define KERNBASE 0x80000000         // First kernel virtual address
  // ---------------  ebp stack bottom
  // |               |
  // |    STACK      |
  // |               | eip -> next to run
  // ----------------  esp stack top
  // |               |
  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){ // 循环去遍历栈指针
    if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
      break;  // 如果栈底 == 0 || < 初始虚拟地址 || == Max of int16
    pcs[i] = ebp[1];     // saved %eip 存储下一个指令执行地址
    ebp = (uint*)ebp[0]; // saved %ebp 栈底
  }
  for(; i < 10; i++) // 其他位置零
    pcs[i] = 0;
}
```

实际上这段代码是依次向前遍历，来获得栈底 EBP，栈顶 ESP，下一个指令地址 EIP 地址。

释放也是相同的套路。

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk)) // 如果 已经释放了
    panic("release");

  lk->pcs[0] = 0; // 释放栈指针
  lk->cpu = 0;

  xchg(&lk->locked, 0); // 同样的xchg 保证了写后读 避免OOD

  popcli();
}
```

先去康康你这个 SpinLock 是不是已经被释放了，然后取清空 call stack， cpu，最后再来修改 locked 位。

这个地方就不用旋等，因为一个 SpinLock 只会被一个 CPU 占用。

最后是关中断

```
void
popcli(void)
{
  if(readeflags()&FL_IF) // 如果取出来的eflag != FL_IF 说明中断已经被打开
    panic("popcli - interruptible");
  if(--cpu->ncli < 0)     // 如果嵌套cli为0 或者 小于0 说明中断被错误的解开
    panic("popcli");
  if(cpu->ncli == 0 && cpu->intena) // 如果cpu的嵌套cli, 和中断标志位都是正确的
    sti();
}
```

还是一样去检查 ELFAGS 堆栈和中断可用常量相不相等。

检查当前 CPU 的嵌套中断数是否大于 0.

然后再来检查 cpu 中断标志是否不为 0，最后再来开中断

SLEEPLOCK IN XV6

前面说到，实际上 SpinLock 不适用于临界区是 I/O 等待的情况，所以在 XV6 中，关于文件系统的锁机制是用 SleepLock 来实现的。

SleepLock 定义在**<proc.c>**文件中

```
// sleep lock
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)   // 如果当前没有正在运行的线程
    panic("sleep");

  if(lk == 0)    // 如果没有锁
    panic("sleep without lk");

  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }

  proc->chan = chan;       // Go to sleep.
  proc->state = SLEEPING;
  sched();
  proc->chan = 0;     // Tidy up.

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

在这里在常规检查之后，并没有之前去请求 SpinLock, 而是先去获取 ptable.lock(这也是一个 SpinLock)。因为逻辑不相干，所以这个 ptable.lock 更容易获得。

然后释放 SpinLock，以便造成堵塞。然后记录下睡眠前状态，把 CPU 交给调度程序来调度。直到被调度回 CPU，先去释放之前占用的 ptable.lock, 然后再来获取真正需要的 SpinLock。

从而实现睡眠锁，可以看到这个这个睡眠锁实际上相关于用另外一个 SpinLock 来做通知的作用，相当于去抢占另外一个不是特别稀缺的资源。

```
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))      // 检查有没有获得ptable.lock
    panic("sched ptable.lock");
  if(cpu->ncli != 1)              // 检查嵌套中断层数是不是1
    panic("sched locks");
  if(proc->state == RUNNING)      // 检查proc状态是不是正在运行
    panic("sched running");
  if(readeflags()&FL_IF)          // 检查eflag 是否与关中断相同
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler); // 交换上下文
  cpu->intena = intena;
}
```

而调度函数 `sched()` 则是依次去检查是否已经释放了 ptable.lock，检查当前 cpu 的嵌套中断数，检查 proc 的状态，检查 EFLAGS 堆栈。最后才是 switch 上下文。

QA

临界区

临界区是指一块对共享数据操作的代码段。通过这些代码段，我们可以对共享内存进行互斥和同步的操作。

同步互斥

同步是指多个线程同时对同一个共享空间进行操作时，保持相互间信息的一致性操作。 互斥是指多个线程对同一个共享空间同时操作时，保持相互间处理操作顺序性的操作。

竞争状况

当多个线程并发的访问一个共享空间，并且执行的结果与线程间执行顺序有关，则称之为竞争状况。

临界区操作时，中断是否应该开启中断

在多核操作系统下，对临界区操作时，必须开启中断。

当进入临界区之后，发生中断，而中断处理程序也请求相同的资源，而因为资源被中断之前的线程占用，则会发生死锁。

xchg

xchg 是一个汇编指令。

从效果上看 xchg 做的是一个交换两个变量，并返回第一个变量这个事情。

实际上这个指令，首先是一个原子性的操作，当然，这我们可以理解毕竟是多核状态，如果有好几个 CPU 核心来抢占同一个 SpinLock，需要保证互斥性, 需要排他来访问这块内存空间。

其次 xchg 是一个 Intel CPU 的锁总线操作，对应到汇编上，就是自带 lock 指令前缀，就算前面没有加 `lock;` 这个操作也是原子性的。

其次，既然是锁总线操作，就有可能失败，这个命令式非阻塞的，每次执行只是一次尝试，所以这个 while 就说的通了。通过循环尝试上锁，来实现旋锁机制。

最后，这条命令还用到了一个 read-modify-write 的操作，这个操作，主要是因为在现代 CPU 中基本上都会使用 Out of Order 来对指令执行进行并行优化。

但是我们这个加锁的过程是一个严格的时序依赖过程，我们必须保证，前面一个 CPU 加上了锁，后面 CPU 来查询的时候都显示已经上锁了。即 read-modify-write 顺序执行。

在 XV6 和 后面分析的 Linux 实际上都是用 `__sync_synchronize` 来实现这个过程的。

利用 XV6 SPINLOCK 实现信号量

```
MAXQUEUE = 0x3fffffff;
struct Semaphore(){
    int num, now;
    SpinLock *sp;
    proc *queue[MAX];
} Sem;

void Init(int num){
    initlock(Sem->sp, 'Semaphore Spin Lock');
    Sem->num = num;
    Sem->now = 0;
}

void P(){
    acquire(Sem->sp);
    if (Sem->now > 0) {
        --Sem->now;
    } else {
        sleep(New_proc(), Sem->sp);
        Sem->queue.append(New_proc());
    }
    release(Sem->sp);
}

void V(){
    acquire(Sem->sp);
    if (Sem->now < Sem->num) {
        ++Sem->now;
    }
    if (Sem->now > 0) {
        wakeup(Sem->queue.end());
    }
    release(Sem->sp);
}
```

利用 XV6 SpinLock 实现读写锁

```
struct RWLock(){
    SpinLock *readLock;
    SpinLock *writeLock;
    int readerNum;
} rwLock;

void LockRead({
    acquire(rwLock->readLock);
    if (rwLock->readerNum){
        acquire(rwLock->writeLock);
    }
    ++readerNum;
    acquire(rwLock->readLock);
}

void UnLockRead({
    acquire(rwLock->readLock);
    --readerNum;
    if (!rwLock->readerNum){
        release(rwLock->writeLock);
    }
    acquire(rwLock->readLock);
}

void LockWrite({
    acquire(rwLock->writeLock);
    acquire(rwLock->readLock);
}

void UnLockWrite({
    release(rwLock->readLock);
    release(rwLock->writeLock);
}
```

SPINLOCK IN LINUX

看完 XV6 的 SpinLock 实现，再来看 Linux 的 SpinLock 实现，就会发现惊人的相似。

本文用 Linux kernal 版本号是 `4.19.30` .

```c
static __always_inline void spin_lock(spinlock_t *lock) // 加锁
{
    raw_spin_lock(&lock->rlock);
    // #define raw_spin_lock(lock)  _raw_spin_lock(lock)
    // SMP
    // #define _raw_spin_lock(lock) __raw_spin_lock(lock)

    // Else
    // #define _raw_spin_lock(lock)         __LOCK(lock)
    // #define __LOCK(lock) \
    // do { preempt_disable(); ___LOCK(lock); } while (0)
    // #define ___LOCK(lock) \
    // do { __acquire(lock); (void)(lock); } while (0)
}


static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable(); // 关中断
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);  // 检查有效情况
    // #define spin_acquire(l, s, t, i)     lock_acquire_exclusive(l, s, t, NULL, i)
    // #define lock_acquire_exclusive(l, s, t, n, i)        lock_acquire(l, s, t, 0, 1, n, i)
    // # define lock_acquire(l, s, t, r, c, n, i)   do { } while (0)
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock); // 上锁
}

static inline int do_raw_spin_trylock(raw_spinlock_t *lock)  // 做一次trylock
{
    return arch_spin_trylock(&(lock)->raw_lock);  // 与环境有关
}
```

其实 Linux 下 SpinLock 的实现有好多种，上次分析了 tryLock，这次来分析最基本的 SpinLock.

Linux 的 SpinLock 的 Locked 是一个叫做 slock 的变量，具体 class 定义就不放了，上锁的函数在linux/spinlock.h>中

spin_lock 会去调用 raw_spin_lock。而 raw_spin_lock 这个函数式指向_raw_spin_lock.

而_raw_spin_lock 则是一个随着运行环境不同的函数。

当处于非 SMP 环境时，实际上就变成了一个简单的禁用内核抢占。

而 SMP 环境中，则会去调用__raw_spin_lock 函数，而这个函数才是真正的实现上锁功能的函数。

大概思路和 XV6 基本一致，先去关中断，然后锁的有效性，最后再去真正的上锁。

而上锁这个函数 LOCK_CONNECT()则是不同环境有不同的实现。

Linux kerenal 中 总共有 15 个实现，（不知道有没有数错）然后以其中几个为例来具体分析。

以<arch/arc/include/asm/spinlock.h>为例

```c
// For arc
static inline int arch_spin_trylock(arch_spinlock_t *lock)
{
    unsigned int val, got_it = 0;

    smp_mb(); // 避免出现OOD
    // #define smp_mb() __sync_synchronize()

    __asm__ __volatile__(
    "1: llock   %[val], [%[slock]]  \n"      // 上锁
    "   breq    %[val], %[LOCKED], 4f   \n" // 已经是上锁状态，跳转至4
    "   scond   %[LOCKED], [%[slock]]   \n" // 比较Locked 与 slosk，然后把slock的值返回
    "   bnz 1b          \n"                 // 上一条指令非0，跳到1
    "   mov %[got_it], 1        \n"        // 当获得Lock时，got it 赋值为1
    "4:                 \n"
    "                   \n"
    : [val]     "=&r"   (val),              //  输出
      [got_it]  "+&r"   (got_it)            //  输出got it
    : [slock]   "r" (&(lock->slock)),       //  输入 slock
      [LOCKED]  "r" (__ARCH_SPIN_LOCK_LOCKED__) // 输入 LOCKED
    : "memory", "cc");

    smp_mb(); // 再次调__sync_synchronize() 保证顺序性

    return got_it;
}
```

这个版本的 arch_spin_trylock 先去声明一个__sync_synchronize()，这个操作和 XV6 中 read-modify-write 中一致。

然后相同是上锁，检查当前上锁状态，比较 Locked slock, 如果不满足条件则继续循环。

当成功上锁，则更改 got_it，并返回。

实际上这个操作流程和 XV6 几乎一样，同样的__sync_synchronize() 同样的判断加锁情况，附带循环比较。

其他版本的 arch_spin_trylock 大概思路也是相同的, 贴一下部分版本解析。

```c
// For alpha
static inline int arch_spin_trylock(arch_spinlock_t *lock)
{
    return !test_and_set_bit(0, &lock->lock);
    // #define test_and_set_bit(nr,p)       ATOMIC_BITOP(test_and_set_bit,nr,p)
}

// For hexagon
static inline unsigned int arch_spin_trylock(arch_spinlock_t *lock)
{
    int temp;
    __asm__ __volatile__(
        "    R6 = memw_locked(%1);\n"      // 上锁
        "    P3 = cmp.eq(R6,#0);\n"        // 判断 是否已经上锁
        "    { if !P3 jump 1f; R6 = #1; %0 = #0; }\n"  // 循环
        "    memw_locked(%1,P3) = R6;\n"   // 获取SpinLock
        "    %0 = P3;\n"                   // 输出1
        "1:\n"
        : "=&r" (temp)
        : "r" (&lock->lock)
        : "memory", "r6", "p3"
    );
    return temp;
}
```

```c
// For arm
static inline int arch_spin_trylock(arch_spinlock_t *lock)
{
    unsigned long contended, res;
    u32 slock;

    prefetchw(&lock->slock);
    do {
        __asm__ __volatile__(
        "    ldrex   %0, [%3]\n"     // 独占寄存器 把lock->slock -> 赋值给slock
        "    mov %2, #0\n"           // contended 赋值0
        "    subs    %1, %0, %0, ror #16\n" // contended = 0, 循环右移16位
        "    addeq   %0, %0, %4\n"           // slock = slock + 1 << TICKET_SHIFT
        "    strexeq %2, %0, [%3]"           // 释放寄存器
        : "=&r" (slock), "=&r" (contended), "=&r" (res)
        : "r" (&lock->slock), "I" (1 << TICKET_SHIFT)
        : "cc");
    } while (res); // 循环尝试加锁

    if (!contended) {
        smp_mb();  // __sync_synchronize();
        return 1;
    } else {
        return 0;
    }
}
```