北京大学 PEKING UNIVERSITY

Multiprocessors SpinLock In XV6 & Linux

discuss

By 韩捷 翁嘉进 陈国强
王圮旭 姜慧强

```
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.
};
```

```
void
acquire(struct spinlock *lk)
{
  pushcli();       // 关中断
  if(holding(lk)) // 如果能获得Lock，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);
}
```

XV6-pushcli

北京大学 PEKING UNIVERSITY

```
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
}
```

XV6-Reafeflag & cli

北京大学 PEKING UNIVERSITY

```c
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;
}
```

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

```
// Per-CPU state
struct cpu {
  uchar id;                    // Local APIC ID; index into cpus[] below
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?

  // Cpu-local storage variables; see below
  struct cpu *cpu;
  struct proc *proc;           // The currently-running process.
};
```

XV6-Holding


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

XV6-xchg

```
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;
}
```

XV6-getcallerpcs

```
// 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;
}
```

XV6-release

```
// 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();
}
```

XV6-popcli

```
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();
}
```

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

XV6-Spinlock Application

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids intermixed output
ftable.lock	Serializes allocation of a struct file in file table
icache.lock	Protects allocation of inode cache entries
idelock	Serializes access to disk hardware and disk queue
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's p->lock	Serializes operations on each pipe
ptable.lock	Serializes context switching, and operations on proc->state and proctable
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

```
// 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);
  }
}
```

```
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;
}
```

1. 什么是临界区？
2. 什么是同步与互斥？
3. 什么是竞争状态？
4. 临界区操作时中断是否应该开启，中断会有什么影响？
5. XV6的锁是如何实现的，有什么操作？
6. xchg指令是什么，该指令有何特性？

XV6- **Semaphore**

北京大学 PEKING UNIVERSITY

```c
struct semaphore {
  int value;
  struct spinlock lock;     // 自旋锁
  struct proc *queue[NPROC];  // 等待队列
  int end;
  int start;
};

void sem_init(struct semaphore *s, int value) {
  s->value = value;
  initlock(&s->lock, "semaphore_lock");
  end = start = 0;
}

void P(struct semaphore *s) {
  acquire(&s->lock);  // 获取锁

  s->value--;
  if (s->value < 0) {
    s->queue[s->end] = myproc();  // 将进程加入到等待队列
    s->end = (s->end + 1) % NPROC;
    sleep(myproc(), &s->lock) // 睡眠
  }

  release(&s->lock);  // 释放锁
}

void V(struct semaphore *s) {
  acquire(&s->lock);  // 获取锁

  s->value++;
  if (s->value <= 0) {
    wakeup(s->queue[s->start]); // 唤醒等待队列队首进程
    s->queue[s->start] = 0;
    s->start = (s->start + 1) % NPROC;
  }

  release(&s->lock);  // 释放锁
}
```

初步方案：

由于XV6操作系统支持多处理器执行命令，因而最初方案本小组参考《多核计算与程序设计》中对读写锁的设计：

1. 将读写操作定义为互斥关系
2. 将不同写操作定义为同步关系，并设置了读者计数器。

```
//定义读写锁的数据结构：
typedef struct RWLOCK_st{
        spinlock ReadLock; // 读锁
        spinlock WriteLock; // 写锁
        UINT uReadcount; // 读者计数器
}RWLOCK;
```

```
RWLock_LockRead() {
ReadLock->acquire(); // 上锁锁住计数器变量的读写，即读锁
uReadcount += 1; // 读者计数器加1
if(uReadcount > 0) // 判断是否触发写锁获取条件——读者非零
WriteLock->acquire(); // 获取写锁
ReadLock->release(); // 解锁计数器变量的读写，即读锁
}
RWLock_UnlockRead() {
ReadLock->acquire(); // 上锁锁住计数器变量的读写，即读锁
uReadcount -= 1; // 读者计数器减1
if(uReadcount == 0) // 判断是否触发写锁释放条件——读者为零
WriteLock->release(); // 释放写锁
ReadLock->release(); // 解锁计数器变量的读写，即读锁
}

RWLock_LockWrite() {
WriteLock->acquire(); // 获取写锁
}
RWLock_UnlockWrite() {
WriteLock->release(); // 释放写锁
}
```

优化方案：为读操作设置副本文件，使得读写操作不存在互斥关系。

优点：保证了多处理器上的读写操作的同步互斥关系 实现简单。
缺点：- 读操作比较频繁时，计数uReadCount可能一直无法归零，会导致写操作饿死现象，消耗较大

RWLock_LockRead():

1. 在获取读锁后，立即判断读者计数器是否为0。
   1.1. 若为0，则建立副本文件并链接至读写锁（副本文件唯一）；
2. 读者计数器加1
3. 释放读锁

RWLock_UnlockRead():

1. 获取读锁
2. 读者计数器减1
3. 判断读者计数器是否为0，若为0，则取消链接并删除副本文件；
4. 释放读锁

优点：解决了写操作可能出现的饿死现象
缺点：副本文件与原文件可能存在滞后的问题。

使用条件变量实现Wait&Signal

- 条件变量是管程内的等待机制
- 进入管程的线程因资源被占用而进入等待状态
- 每一个条件变量均表示一种等待原因，并且对应一个等待队列

```
Class Condition{
    int wait_count = 0; // 等待的线程数目
    Wait_Queue wait_queue; // 等待队列
}
```

Wait的伪代码

```
Condition::Wait(lock){
        wait_count++;
        Add this thread t to wait_queue; // 将当前线程添加到等待队列
        release(lock); // 释放锁
        schedule(); // 需要保证原子性与一致性
        require(lock); // 申请锁，实现忙等
        }
```

Signal的伪代码

```
Condition::Signal(){
        if(wait_count>0){ // 如果等待的线程数目大于0
        Remove a thread t from wait_queue; // 从等待队列移除一个就绪线程
        wakeup(t); // 需要保证原子性与一致性
        wait_count--;
                }
        }
```

Linux-SpinLock

```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
}
```

Linux- arch_spin_trylock

```
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;
}
```

以</arch/arc/include/asm/spinlock.h>为例

[1] Cox, R., Kaashoek, M. F., & Morris, R. (2011). Xv6, a simple Unix-like teaching operating system. *2013-09-05]. http://pdos. csail. mit. edu/6.828/2012/xv6. html*.

[2] Bovet, D. P., & Cesati, M. (2005). *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.".

[3] Stevens, W. R., & Rago, S. A. (2008). *Advanced programming in the UNIX environment*. Addison-Wesley.

[4] Tanenbaum, A. S., & Bos, H. (2015). *Modern operating systems*. Pearson.