

Locks in XV6

Has both locks : spinlock and sleeplock

```
/*
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.
};


*/
```

spinlocks in xv6

spinlocks in xv6 code

```
struct {
    struct spinlock lock;
    struct buf buf[NBUF];
    struct buf head;
} bcache;
struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;
struct {
    struct spinlock lock;
    struct inode inode[NINODE];
} icache;
struct sleeplock {
    uint locked;    // Is the lock held?
    struct spinlock lk;
};

static struct spinlock idelock;
struct {
    struct spinlock lock;
    int use_lock;
    struct run *freelist;
} kmem;
struct log {
    struct spinlock lock;
    ...
}
struct pipe {
    struct spinlock lock;
    ...
}
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
struct spinlock tickslock;
```



```

static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;
    // The + in "+m" denotes a read-modify-
    // write operand.
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}

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.
};

```

Spinlock on xv6

```

void acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to
    avoid deadlock.

```

```

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;
    //extra debugging code
}

```

```

void release(struct spinlock *lk)
{
    //extra debugging code
    asm volatile("movl $0, %0" :
        "+m" (lk->locked) : );
    popcli();
}

```

in acquire (spinlock *lk):

pushcli() : get eflags : readflags() which pushes flag onto stack
 cli : diable interr
 increment ncli in mycpu

if holding(lk) :
 holding : pushcli
 check if locked is set and check if other proc is is holding a lock : then it

panics

while (xchg(&lk->locked,1) : actual locking
 syn_schronise :
 lk->cpu = mycpu() // note down where lock is being used
 getcallerpcs : setting values in pcs array
 pcs gets ebp values .i.e. return values
 which is used in panic

in release

if !holding(lk) :
 panic
 lk->pcs[0] = 0
 lk->cpu = 0
 asm volatile("movl \$0, %0" : "+m" (lk->locked) :); // actual release code

```

Void acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
        panic("acquire");
    .....
void pushcli(void)
{
    int eflags;

    eflags = readeflags();
    cli();
    if(mycpu()->ncli == 0)
        mycpu()->intena = eflags & FL_IF;
    mycpu()->ncli += 1;
}
static inline uint
readeflags(void)
{
    uint eflags;
    asm volatile("pushfl; popl %0" : "=r" (eflags));
    return eflags;
}

```

spinlocks

- Pushcli() - disable interrupts on that processor
- One after another many acquire() can be called on different spinlocks
- Keep a count of them in mycpu()->ncli

```

void
release(struct spinlock *lk)
{
    ...
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );
    popcli();
}
Void popcli(void)
{
    if(readeflags() & FL_IF)
        panic("popcli - interruptible");
    if(--mycpu()->ncli < 0)
        panic("popcli");
    if(mycpu()->ncli == 0 && mycpu()->intena)
        sti();
}

```

spinlocks

- Popcli()
 - Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called

spinlocks

- Always disable interrupts while acquiring spinlock
 - Suppose **iderw** held the **idelock** and then got interrupted to run **ideintr**.
 - **Idintr** would try to lock **idelock**, see it was held, and wait for it to be released.
 - In this situation, **idelock** will never be released
 - Deadlock
- General OS rule: if a spin-lock is used by an **interrupt** handler, a processor must never hold that lock with interrupts enabled
- Xv6 rule: when a **processor** enters a spin-lock critical section, xv6 always ensures interrupts are disabled on that processor.

idelock is never released : because

When chan is not NULL
state has to be WAITING

sleeplocks

- Sleeplocks don't spin. They move a process to a wait-queue if the lock can't be acquired
- Xv6 approach to "wait-queues"
 - Any memory address serves as a "wait channel"
 - The **sleep()** and **wakeup()** functions just use that address as a 'condition'
 - There are no per condition process queues! Just one global queue of processes used for scheduling, sleep, wakeup etc. --> Linear search everytime !
 - **costly, but simple**

void



sleep(void *chan, struct spinlock *lk)

sleep()



```
{
    struct proc *p = myproc();
    ....
    if(lk != &ptable.lock){
        acquire(&ptable.lock);
        release(lk);
    }
    p->chan = chan;
    p->state = SLEEPING;
    sched();
    // Reacquire original lock.
    if(lk != &ptable.lock){
        release(&ptable.lock);
        acquire(lk);
    }
}
```

- At call must hold lock on the resource on which you are going to sleep
- since you are going to change p-> values & call sched(), hold ptable.lock if not held
- p->chan = given address remembers on which condition the process is waiting
- call to sched() blocks the process

actually implemented
so sleep is called on a particular