XK Synchronization + Reader-Writer Locks

Synchronization in XK

- Spinlock: disable interrupts + spin

- sleeplock: give up processor until lock free

- sleep(spinlock, chan): CV wait

- wakeup(chan): CV broadcast

8.15

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

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint64_t pcs[10]; // The call stack (an array of program counters)
                     // that locked the lock.
};

// Acquire the lock. Loops (spins) until the lock is acquired.
void acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock.
  if (holding(lk))
    panic("acquire");

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

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

8.16

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

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m"(lk->locked) :);

  popcli();
}
```

8.17

```
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;
}
```

8.17½

```
// Long-term locks for processes
struct sleeplock {
  uint locked;         // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name; // Name of lock.
  int pid;    // Process holding lock
};

// a sleeping lock relinquishes the processor if the lock is busy
// note mesa semantics: process can wakeup and find the lock still busy.
// NOTE: no spinlocks should be held while calling `acquiresleep`
// (since if `acquiresleep` tries to sleep while a spinlock is held,
// `sched''s contract will be violated).
void acquiresleep(struct sleeplock *lk) {
  acquire(&lk->lk);

  // We should never try to acquire sleeplock while we already hold it
  if (lk->locked && lk->pid == myproc()->pid) {
    panic("Already holding sleeplock!");
  }

  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

8.18

```
// a sleeping lock wakes up a waiting process, if any, on lock release.
void releasesleep(struct sleeplock *lk) {
  acquire(&lk->lk);

  // Assert that the lock is actually locked.
  if (!lk->locked) {
    panic("releasesleep: sleeplock is not locked!");
  }

  // Assert that we are the one holding the lock.
  if (lk->pid != myproc()->pid) {
    panic("releasesleep: sleeplock is not held by current proc!");
  }

  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

8.19

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void sleep(void *chan, struct spinlock *lk) {
  if (myproc() == 0)
    panic("sleep");

  if (lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock to change p->state and then call sched.
  // Once we hold ptable.lock, we are guaranteed not to miss any wakeup
  // (wakeup runs with ptable.lock locked), so it's okay to release lk.
  if (lk != &ptable.lock) { // DOC: sleeplock0
    acquire(&ptable.lock);  // DOC: sleeplock1
    release(lk);
  }

  // Go to sleep.
  myproc()->chan = chan;
  myproc()->state = SLEEPING;
  sched();

  myproc()->chan = 0;

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

8.20

```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void wakeup1(void *chan) {
  struct proc *p;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void wakeup(void *chan) {
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

8.21

Reader-Writer Locks

- allows any number of concurrent readers
- only one writer (and no readers)
- common in DBMSes + OS kernels
  - only worth it if vast majority of ops are reads

API

acquireRead() releaseRead()

acquireWrite() releaseWrite()

Reader-Writer Lock interface

```
interface ReaderWriterLock {

    void acquireRead();

    void acquireWrite();

    void releaseRead();

    void releaseWrite();

}
```

Reader-Writer Lock Design

- RWLock is itself a concurrent data structure!
  - follow design recipe from last time (monitor)
  - ↳ will have a (normal) lock in our RWLock!
- what else?
  - track # of active Readers / active Writers
    - at most one > 0
    - active Writers ≤ 1
  - track # of waiting Readers / waiting Writers

So far

```
class RWLock {
    int activeReaders;
    int activeWriters;
    int waitingReaders;
    int waitingWriters;
```

Reader-Writer Lock Design (continued)

- When do the interface methods need to wait?
  - acquireRead: wait if there are active writers
  - acquireWrite: wait if there are active readers or writers
  - releaseRead / releaseWrite: don't wait

↳ add condition variables for wait conditions

  - noActiveWriters
  - noActiveReadersOrWriters

with CVs

```
class RWLock {
    int activeReaders;
    int activeWriters;
    int waitingReaders;
    int waitingWriters;

    CV noActiveWriters;                    } associated lock: "this"
    CV noActiveReadersOrWriters;           }
```

```
synchronized void acquireRead() {
    waitingReaders ++;
    while (activeWriters > 0)
        noActiveWriters.wait();
    waitingReaders --;
    activeReaders ++;
}

synchronized void acquireWrite() {
    waitingWriters ++;
    while (activeReaders > 0 ||
           activeWriters > 0)
        noActiveReadersOrWriters.wait();
    waitingWriters --;
    activeWriters ++;
}
```

```
synchronized void releaseRead() {
    activeReaders--;
    if (activeReaders == 0)                    // know no active writers
        noActiveReadersOrWriters.signal();
}

synchronized void releaseWrite() {
    activeWriters--;                           // we are only active entity
    if (waitingWriters > 0) {
        noActiveReadersOrWriters.signal();
    } else {
        noActiveWriters.broadcast();
    }
}
```