```
* locking

Two kinds of basic locks:
1. non-blocking
2. blocking

Code:
1. lock(X)
2. code, code, code... (critical section -- CS) [say accessing shared d-s "Y"]
3. unlock(X)

lock(X) requires h/w support:
- requires processor support, special instructions
- instruction: test-and-set (TAS)
        test if register R == 0, if so, set it to 1 -- atomically!
- instruction: compare-and-swap (CAS)
        test if register R1 > R2: if yes, swap values -- atomically!

General kinds of problems:
1. if you access Y w/o a lock -- mess up values of Y, uncoordinated
2. if lock but forget to unlock -- deadlock b/c "lock owner" will never
   release the lock
3. supposed Y is accessed in multiple places in the code, you lock in all of
   them, but one.  Still a problem: must protect d-s Y with lock X
   everywhere Y is accessed.
4. what if CS is very large piece of code.  Results in lower throughput,
   less concurrency.

Two basic locks in Linux (most other OSs): spinlock and a mutex

1. Spinlock (a non-blocking lock)
- pro: fast lock, few instructions
- a busy loop, cycles w/ say a CAS instruction until register value is "unset"
- con: takes up a core, consumes cycles until can get into CS
- useful when CS is really small
- must not use spinlock if can block on I/O!
- most useful when changing values of d-s in memory only
- only one may enter the spinlock CS (only one lock owner)

2. Mutex (a blocking lock)
- pro: can be used when blocking inside CS
- usually used when you have to wait longer than inside a spinlock
- con: a slower lock, takes 100s of instructions to un/lock.
        so inefficient for short CSs
- if you wait on mutex, usually you go into WAIT state by scheduler.
- only one may enter the mutex CS (only one lock owner)

Building other locks types:

3. An integer counter: a reference counter based on mutex, and one based on
   spinlock.
- has API: create counter, add/sub N to counter, inc/dec++ to counter, test
  value, destroy object, etc.

4. read-write-semaphore: rwsem
- a block lock, based on a mutex, a type of counting lock
- supports multiple readers and one writer
- useful when access to shared d-s is mostly for reading, and an occasional write.
- multiple readers may enter the rwsem CS, only one writer may enter.

5. RCU: Read-Copy-Update
- Read: make a quick private copy of a d-s under spinlock
- Copy: take as much time as you want to modify your private copy (no lock needed)
- Update: when done, you have to merge your changes from private copy onto
```

```
    main d-s, under a lock (e.g., spinlock)

  // rwsem: multiple readers, one writer only

  struct rwsem {
    lock l; // to protect 'counter'
    int c; // 0: no owners. >0 #readers, -1 means one writer
    //  bool waiting_writers; // if T, means we have at least one waiting writer.
    // ok, but easier to have 2 queues, one for readers and one for writers
    wait_queue_t readers_wq; // a wait list (for readers)
    wait_queue_t writers_wq; // a wait list (for writers)
  };

  int rwsem_read_lock(rwsem *p)
  {
    lock(l);
    if (c >= 0) { // no owners at all, or only readers
      c++;
    } else if (c == -1) {
      add_to_waitq(p->writers_wq, current_task); // thread goes into wait state
    }
    unlock(l);
  }

  int rwsem_read_unlock(rwsem *p)
  {
    lock(l);
    c--; // possible bugs: what if c==0
    unlock(l);
  }

  int rwsem_write_lock(rwsem *p)
  {
    lock(l);
    if (c == 0) { // no owners at all
      c = -1;
    } else if (c == -1) {
      add_to_waitq(p->writers_wq, current_task); // thread goes into wait state
    } else if (c > 0) { // there are some readers
      add_to_waitq(p->writers_wq, current_task); // thread goes into wait state
      //
    }
    unlock(l);
  }

  int rwsem_write_unlock(rwsem *p)
  {
    lock(l);
    c = 0; // possible bugs: what if c != -1

    // wakeup any waiting writers, assuming writers are "more important"
    if (waitq_not_empty(p->writers_wq))
      wakeup(head_of(p->writers_wq));

    // ???
    if (waitq_not_empty(p->readers_wq))
      wakeup(head_of(p->readers_wq));

    unlock(l);
  }
```