```
See updated rwsem implementation

How many readers to wakeup: depends on system load/capacity, often you
wakeup N based on available free cores/cpus, b/c each can service a thread
concurrency.

In a rwsem, if you find you have a lot of writers all the time, readers may
starve.  If that's the case, you're using the wrong locking primitive.
Better use a plain mutex, where it's FCFS for all lock requesters (whether
they want to read or write the rwsem-protected d-s).

* RCU: Read-Copy-Update

Example: RCU protects a shared linked list of unsorted names.

LIST1 -> "B" -> "D" -> "A" (TAIL)

>1. Read: make a quick private copy of a d-s under spinlock.

So it's a quick spinlock, and you make a copy of the shared d-s.

a. grab RCU spinlock
b. kmalloc LIST2 (may even be OUTSIDE the spinlock)
c. memcpy LIST1 to LIST2
d. release RCU spinlock

>2. Copy: take as much time as you want to modify your private copy (no lock needed)

"Copy" phase means: you can manipulate your private LIST2 copy all you want,
for as long as you want.

If you're a reader only of the shared d-s (LIST1), then you can inspect your
copy (LIST2), and when done, kfree(LIST2) and you're done.

Only writers of the shared d-s need to go into phase 3.

>3. Update: when done, you have to merge your changes from private copy onto
>   main d-s, under a lock (e.g., spinlock)

a. grab RCU spinlock
b. compare LIST1 (orig) to LIST2 (yours)
c. merge changes from LIST2 into LIST1
d. release RCU spinlock

Compare/merge phase:

Ex. 1. added an item

LIST1 (orig) -> "B" -> "D" -> "A"

LIST2 (new)  -> "B" -> "D" -> "A" -> "C"

if I find there's a simple change, like adding a new item "C" at the end of
the list, then all we need to do is copy "C" to end of LIST1, kfree(LIST2),
and we're done.  Even better swap LIST1 and LIST2 heads, and discard the old
LIST1.

Ex. 2. deleted an item

LIST1 (orig) -> "B" -> "D" -> "A"

LIST2 (new)  -> "B" -> "A"

Same as simple base case 1.
```

```
Ex. 3. added item, but LIST1 changed!

LIST1 (orig) -> "B" -> "D" -> "A"

LIST1' (NOW) -> "B" -> "D"

LIST2 (new)  -> "B" -> "D" -> "A" -> "C"

LIST1 changed probably b/c other RCU holders changed it.  And they got to
merge their changes faster than you.

More complicated case: you have to compare changes, figure out how to
preserve past changes, and do a "3-way merge" (may be complex or even
impossible).  So may have to discard your changes, regrab the RCU and try
again.

Incentive is for writers using an RCU to NOT wait too long before merging
changes, else they might have to do a complex 3-way merge.

How to detect changes to LIST1 most effectively?
- a timestamp may be useful to determine changes.
- a counter, each time one merges into the shared d-s.  A "generation ID",
  or version.  Quicker to compare and produce than a timestamp.


// rwsem v2: multiple concurrent readers, one writer only at a time (and no readers)

struct rwsem {
  lock l; // to protect 'counter', may also need to protect the wq's
  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 == -1 || writers_wq != NULL) { // reader go to sleep if there's a
                                       // writer or there are waiting writers
    add_to_waitq(p->readers_wq, current_task); // thread goes into wait state (readers queue)
  } else if (c >= 0) { // no owners at all, or only readers
    // actual "if" here may not be needed
    c++;
  }
  unlock(l);
}

int rwsem_read_unlock(rwsem *p)
{
  lock(l);
  c--; // possible bugs: what if c==0
  // wake up one writer when last reader relased rwsem
  if (c == 0 && writers_wq != NULL)
    wakeup(head_of(p->writers_wq));
  // ensure we wake some more readers, b/c not all may have woken up at write_unlock
  else if (readers_wq != NULL)
    wakeup(head_of(p->readers_wq));

  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 (writes queue)
    } else if (c > 0) { // there are some readers
      // assuming new writer is more important than all readers.
      // so we need to prevent NEW readers (called "throttling" new readers)
      // ... and we need to let existing readers "drain" out of the system.
      add_to_waitq(p->writers_wq, current_task); // thread goes into wait state (writers queue)
    }
    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 no waiting writers, then wakeup a waiting reader if there's one
    // but if there's >1 reader, how many to wakeup? (all, some, etc. --
    // depends on policy).  Wakeup just 1, will serialize all waiting readers,
    // but wakeup all can cause a sudden spike of load.
    if (writers_wq != NULL)
      wakeup(head_of(p->writers_wq));
    else if (reader_wq != NULL)
      wakeup(head_of(p->readers_wq));

    unlock(l);
  }
```