Write Preferring RW Locks

Lock lk;
Condvar reader-cv;
Condvar writer-cv;

read_acquire()
{
  lk.acquire();
  while (waiting-writers > 0)
    { // active writer
      SI reader-cv.wait();
      active-reader ++;
      lk.release();
    }
  assert(!active-write);
}

int active-readers = 0;
int waiting-writers = 0;
bool active-write = false;

write_acquire()
{
  lk.acquire();
  while (active-readers > 0)
    { // active write
      SI writer-cv.wait();
      active-write = false;
      lk.release();
    }
  lk.release();
}

write_release()
{
  lk.acquire();
  active-write = false;
  lk.release();
}

if (waiting-writers > 0)
  writer-cv.signal();
else
  reader-cv.broadcast();
Read Preferring vs. Write Preferring

- new reads allowed as long as there are other reads
- can starve writers

- new reads not allowed when there are writers waiting
- limit amount of concurrent ops (reads)

→ how about this?

- track writers' wait times and/or # of waiting writers & use that as a condition for allowing new reads
Race Conditions

- the correctness of the system is dependent on the ordering of scheduling

```plaintext
thread_a();
print(2);
3

thread_b();
print(3);
3
```

- scheduling order can affect the output, but not a race condition if the output order doesn't affect the correctness.

- some potential causes:
  - unprotected data access, semantically related states not, bad usage of synch. primitives
  - (global_x++) operated on atomically.

- steps to think through for finding races
  - what data is in shared location (statically allocated data, heap if ptr is shared)
  - who accesses the shared data?
  - granularity of shared data

- if an array is shared but each entry is only accessed by one thread, no concurrent access, its safe.
Non-data access race condition

```c
function(src, dst) {
    src->lk->acquire();
    dst->lk->acquire();
    // copy from src to dst
    src->lk->release();
    dst->lk->release();
}
```

$t_1 (A \rightarrow B)$

$t_2 (B \rightarrow A)$

A→lk→acquire();
B→lk→acquire();
A→lk→acquire();
B→lk→acquire();
B→lk→acquire();
B→lk→acquire();

A deadlock given this scheduling order

$\perp$ threads mutually wait on each other (cycle of waits)

→ How can we break this cycle?

→ Lock ordering

→ Try locks, release if can't acquire all locks needed.