10/23/23

Write Preferring RW Locks

```
Lock lk;
Condvar reader_cv;
Condvar writer_cv;
```

```
int active_readers=0;
int waiting_writers=0;
bool active_write = false;
```

```
read_acquire(){
    lk.acquire();
    while (waiting_writers >0
      || active_write) {
        reader_cv.wait();
    }
    active_reader++;
    lk.release();
}  //assert(!active_write)
```

```
write_acquire(){
    lk.acquire();
    waiting_writer++;
    while (active_readers >0
      || active write) {
      writer_cv.wait();
    }
    waiting_writer--;
    active_write = true;
    lk.release();
}  //assert(active_readers ==0);
```

```
read_release(){
    lk.acquire();
    active_readers --;
    if (active_reader ==0
      && waiting_writer >0){
        writer_cv.signal();
    }
    lk.release();
}
```

```
write_release(){
    lk.acquire();
    active_write = false;
    if (waiting_writers >0){
        writer_cv.signal();
    else {
        reader_cv.broadcast();
    }
    lk.release(); }
```

Read Preferring vs. Write Preferring

Read Preferring

- new reads allowed as long as there are other reads
- can starve writers
  - ↳ can we improve on this?
  - ↳ track writers' wait times and/or # of waiting writers & use that as a condition for allowing new reads

Write Preferring

- new reads not allowed when there are writers waiting
- limit amount of concurrent ops (reads)
  - → how about this?

Race Conditions

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

```
thread_a(){
    print(2);
}
```

```
thread_b(){
    print(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, (global_x++)

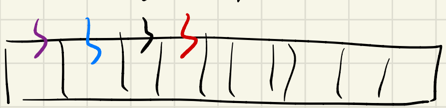
semantically related states not operated on atomically.

bad usage of synch. primitives

→ 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?

maybe only 1 field in a struct is shared



(if an array is shared but each entry is only accessed by one thread, no concurrent access, it's safe)

- granularity of shared data

→ non data access race condition

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

t1 (A→B)	t2 (B→A)
A->lk->acquire();	
	B->lk->acquire();
	A->lk->acquire();
B->lk->acquire();	

☆ deadlock given this scheduling order

↳ 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.