10/20/23

Bounded Buffer Problem

```
char buffer[N];
int consume_ofs = 0;
int produce_ofs = 0;
int count = 0;
```

```
function produce(item) {
    lk.acquire();
    while (count == N) { // buffer is full
        not_full_cv.wait();
    }
    buffer[produce_ofs] = item;
    produce_ofs = (produce_ofs + 1) % N;
    count++;
    not_empty_cv.signal(); // wake up consumers
    lk.release();
}
```

```
condvar. not_full_cv;
condvar. not_empty_cv;
```

```
// returns item consumed.
function consume() {
    lk.acquire();
    while (count == 0) { // buffer is empty
        not_empty_cv.wait();
    }
    // assert count > 0;
    item = buffer[consume_ofs];
    consume_ofs = (consume_ofs + 1) % N;
    count--;
    not_full_cv.signal();
    lk.release();
}
```

→ Synchronization problem: sleeplock (mutex)

```
Lock lk; //spinlock
Condvar lk_cv;
bool free = True; // lock status
```

```
lock_acquire(){
  lk.acquire();
  while(!free){
    lk_cv.wait(lk);
  }
  free = False;
  lk.release();
}
```

```
lock_release(){
  lk.acquire();
  free = True;
  lock_cv.signal();
  lk.release();
}
```

```c
// a sleeping lock relinquishes the processor if the lock is busy
// note mesa semantics: process can wakeup and find the lock still busy
void acquiresleep(struct sleeplock *lk) {
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}

// a sleeping lock wakes up a waiting process, if any, on lock release
void releasesleep(struct sleeplock *lk) {
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

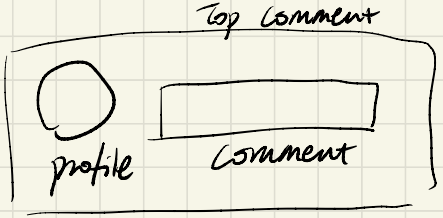
→ Lock: mutual exclusion to shared data.

→ all accesses to shared data are reads?

→ all accesses to shared data are writes?

→ access pattern: mix of read & writes?

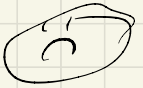
reader writer lock
- → safe to have multiple readers as long as there's no writes
- → writer needs exclusive access (no readers or other writers)



- mostly reads
- occasional writes

☆ 1st approach: lock around all reads & writes.
→ slow perf. :(

☆ 2nd approach: no locks around reads, locks writes.
→ reader could see partial writes:

profile A: Hahn
B: 

Reader Writer Locks

→ APIs: read_acquire, read_release, write_acquire, write_release

↳ can succeed when no reader or writer

↳ can succeed if there are already readers

✱ could lead to starvation

→ Read Preferring vs Write Preferring

- allow new readers to read even if there are writers waiting
- wake up readers when there are waiting readers & writers.
- stop new readers from reads if there are waiting writers
- wake up writer when there are waiting readers & writers.

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_readers++;
  lk.release(); }

read_release() {
  lk.acquire();
  ?
  lk.release(); }

write_acquire() {
  lk.acquire();
  waiting_writer++;
  while(active_readers > 0
  || active_write) {
    writer_cv.wait();
  }
  waiting_writers--;
  active_write = True;
  lk.release(); }

write_release() {
  lk.acquire();
  ?
  lk.release(); }
```