4/19/24

Write Preferring Reader Writer Lock

```
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 (active_write ||
      waiting_writers > 0) {
        reader_cv.wait(lk);
    }
    active_readers++;
    lk.release();
}
// holds the read lock upon success

write_acquire() {
    lk.acquire();
    waiting_writers++;
    while (active_write ||
      active_readers > 0) {
        writer_cv.wait(lk);
    }
    waiting_writers--;
    active_write = True;
    lk.release();
}
// holds the write lock upon success

read_release() {
    lk.acquire();
    active_readers--;
    if (active_readers == 0
      && waiting_writers > 0) {
        writer_cv.signal();
    }
    lk.release();
}
// releases

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

- ↳ can starve writers
- ↳ variations:
  stop new readers from acquiring read lock once a threshold of wait time or number of waiting writers is met.

- ↳ can do a similar hybrid approach to improve throughput

Deadlocks

→ cycle of waiting threads blocked on each other

Deadlock Example 1 :

```
Lock A;        thread_func1() {                    thread_func2() {
Lock B;            A.acquire();                        B.acquire();
                   B.acquire();                        A.acquire();
               }                                   }
```

Deadlock Example 2 :

2 bounded buffer A, B

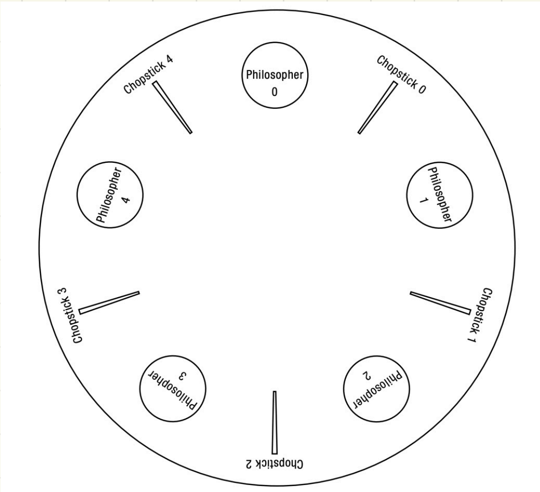
```
thread_func1() {
    A.consume();
    B.produce(item);
}

thread_func2() {
    B.consume();
    A.produce(item);
}
```

Deadlock Example 3:

```
Lock lk;
BoundedBuffer A;
```

```
thread_func1() {
    lk.acquire();
    <do something>
    A.consume();
    <do more things>
    lk.release();
}
```

```
thread_func2() {
    lk.acquire();
    <do something>
    A.produce(item);
    <do more things>
    lk.release();
}
```



Dining Philosophers

Necessary Conditions For Deadlock

Deadlock ⇒ All 4 Conditions are met

1. Bounded Resources : finite instances of resource
2. No Preemption : resource can't be forcibly taken away
3. Hold & Wait : hold on to resource while waiting
4. Circular Wait : cycle of waiting

Are necessary conditions sufficient for deadlock? (guarantees)

→ Single instance resource (e.g. lock) yes!

→ Multiple instance resources (e.g. chopstick, producer consumer) No!

All 4 Conditions are met ⇏ Deadlock

What to do w/ deadlock?

→ break any necessary condition breaks a deadlock

→ 3 types of approach: prevention, avoidance, detection

Deadlock Prevention

→ limit system/program behaviors to break a condition

- Bounded resources: provide sufficient resources (reserve some resources to deal w/ cases before running out of resources)
- No Preemption: let system preempt resources (resource lease)
- Hold & wait: release while wait (lock_try_acquire, acquire_all API)
- Circular wait: lock ordering (total ordering of locks, acquire according to the order)

Deadlock Avoidance (Admission Control)

-> system determines when it's safe to grant resources
-> threads can do whatever they want (acquire lock in any order),
System delay granting request until it's safe to do so.

-> Dining Philosopher Example:

Rules for Chopstick fairy handing out chopstick (want to be maximally permissive)

5 chopsticks,
5 philosophers
request 1 chopstick at a time, each needs 2 max.

-> hand out chopsticks freely until there's 1 left
-> hand out the last chopstick to anyone if a philosopher already have 2 chopstick
-> hand out the last chopstick to someone that already have a chopstick

★ knows max request for each thread

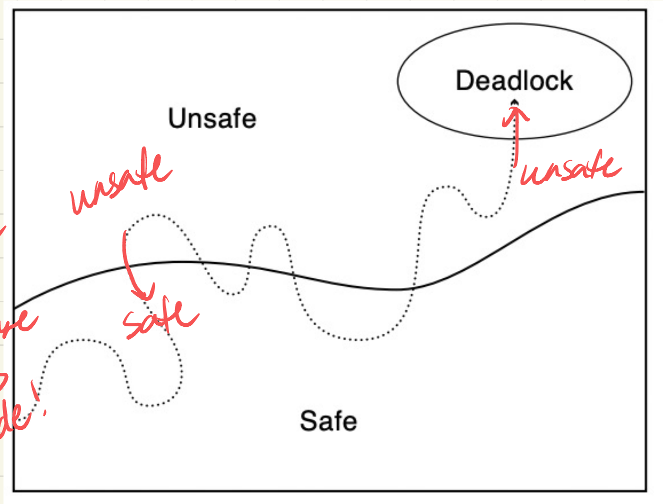
Banker's Algorithm: for deadlock avoidance

→ Safe, unsafe, deadlock

Safe: there's at least 1 ordering to grant requests s.t. everyone can get their max requests eventually

Unsafe: there's at least 1 future request that will cause the system to deadlock regardless of the resource granting order

★ only grant request when doing so will keep the system in a safe state!



goes back to safe if the future request is never made!

happens when the bad future request is made