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();
}

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

read_release()
{
    lk.acquire();
    active_readers--;
    if (active_readers == 0 && waiting_writers > 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();
}

3 3

// holds the read lock upon success

5 5

// holds the write lock upon success

7

// releases
Read Preferring vs. Write Preferring

\( \perp \) can starve writers

\( \perp \) can do a similar hybrid approach to improve throughput

\( \perp \) variations:

- stop new readers from acquiring read lock once
- a threshold of wait time
- or number of waiting writers
- is met
Deadlocks

→ cycle of waiting threads blocked on each other

Deadlock Example 1:

lock A;
lock B;

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

3
3
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;

Bounded Buffer A;

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

3

3

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

3
Necessary Conditions For Deadlock

\[ \text{Deadlock} \Rightarrow \text{All 4 conditions are met} \]

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?

- Single instance resource (e.g., lock) yes!
- Multiple instance resources (e.g., chopstick, producer, consumer) No!

All 4 conditions are met! \( \Rightarrow \) 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 in cases before running out of resources)

No Preemption:

- let system preempt resources (resource lease)

Hold & wait:

- release while wait (lock-try-acquire, acquire all APIs)

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 delays granting request until it's safe to do so.

- Dining Philosopher Example:
  
  Rules for Chopstick fairy handing out chopstick (want to be maximally permissive)
  
  - 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.
Banker's Algorithm: for deadlock avoidance

- Safe, unsafe, deadlock

- There's at least 1 ordering to grant requests s.t. everyone can get their max requests eventually

- Only grant request when doing so will keep the system in a safe state!

- Happens when the bad future request is made