# Concurrency

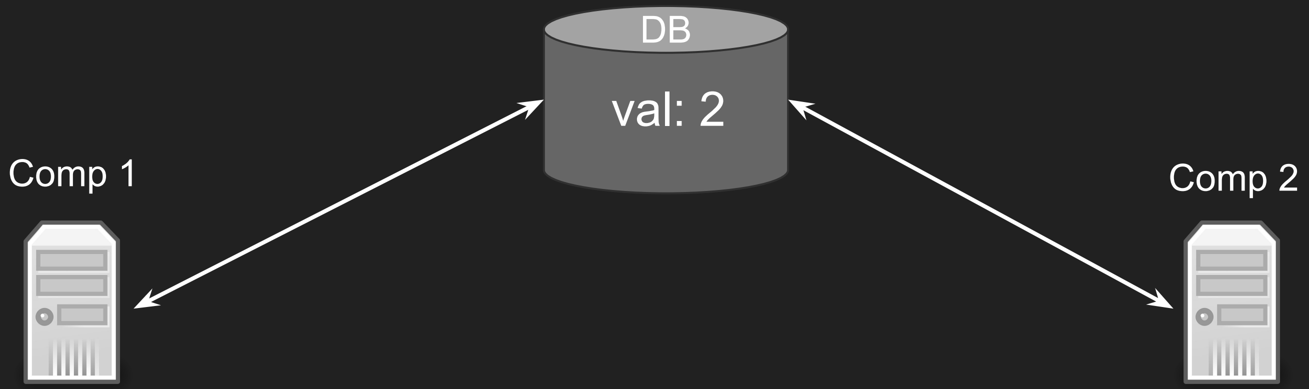
# Prelim

- Grades have been released on Gradescope
  - Median: 79
  - Mean: 76
  - High: 100
- Significant error in the writing of 6.1 so it was not graded
  - We want to go over this question to help reinforce DP; still working on plan for that but will be before the final
- Use Gradescope to request regrades if you believe your submission was graded incorrectly
  - Original grader will re-grade first, can appeal to Ramin if you’re still unsatisfied

# Administrivia

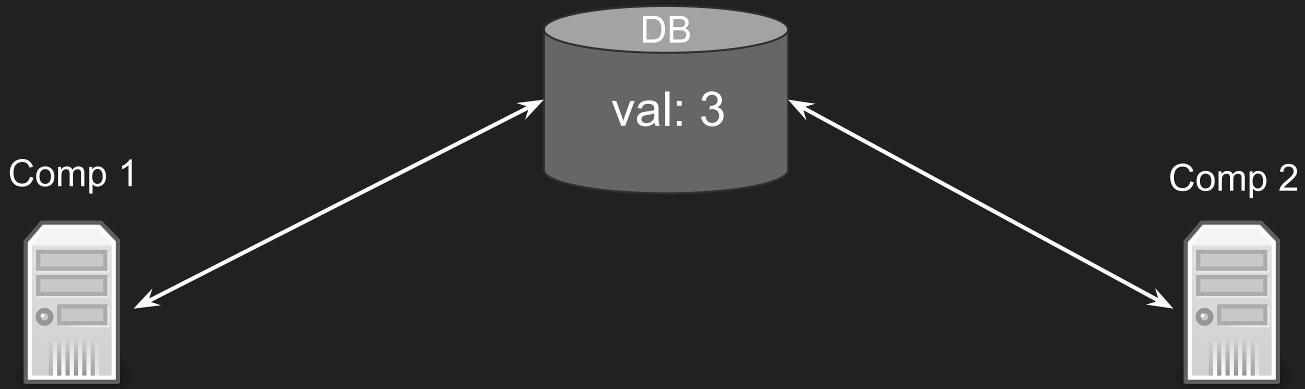
- Drop deadline blanket extended until Nov. 1
  - Additional extension is possible if necessary; please reach out to Ramin/Angy
- Ramin is away at a conference this week
  - Will hold extended office hours next week
  - Available for teleconference for urgent matters
- Deadline for finding HW2 partners is tonight

# Concurrency



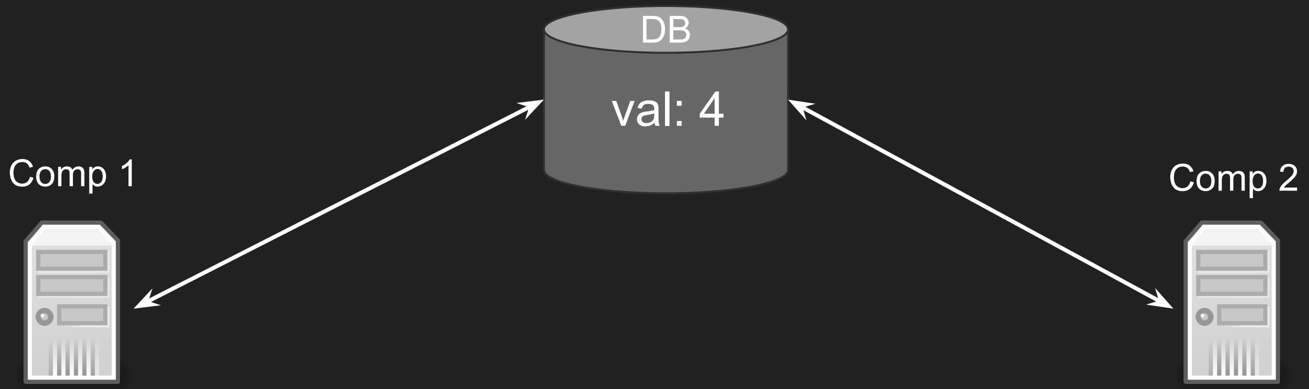
```
➡ def update_val(db):
    ➡ val = db.get_val()
    ➡ val = val + 1
    ➡ db.update_val(val)
```

```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```



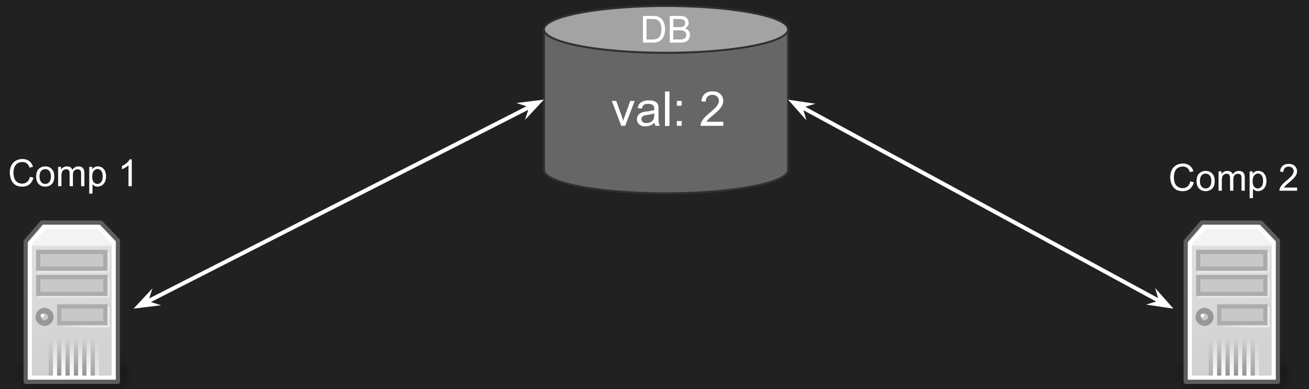
```
def update_val(db):
        val = db.get_val()
        val = val + 1
        db.update_val(val)
```

```
➡ def update_val(db):
    ➡   val = db.get_val()
    ➡   val = val + 1
    ➡   db.update_val(val)
```



```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```

```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```



DB

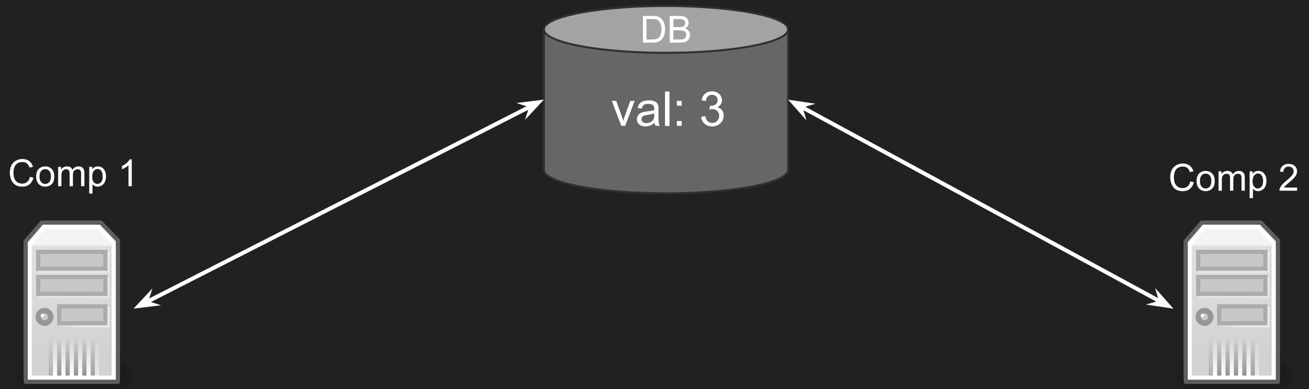
val: 2

Comp 1

Comp 2

```
➡ def update_val(db):
➡     val = db.get_val()
➡     val = val + 1              (val: 3)
➡     db.update_val(val)
```

```
➡ def update_val(db):
➡     val = db.get_val()
➡     val = val + 1              (val: 3)
      db.update_val(val)
```



DB

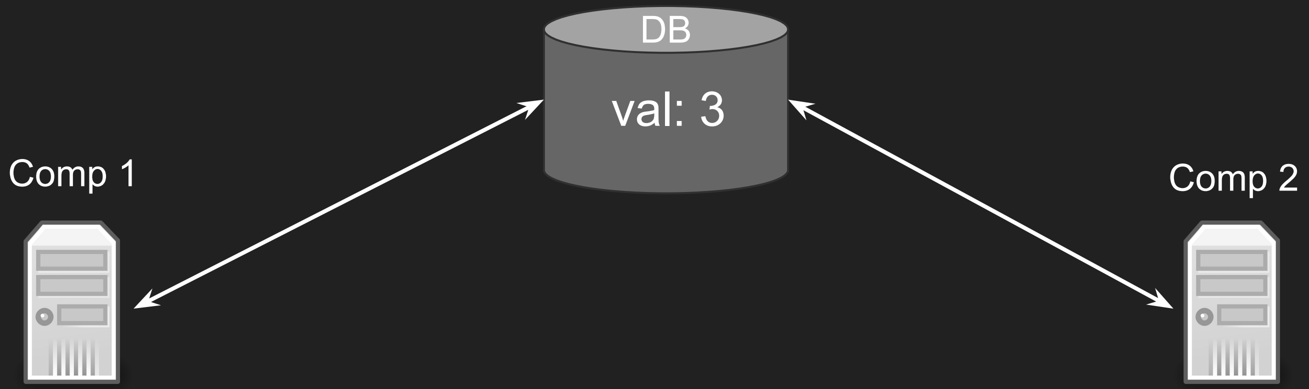
val: 3

Comp 1

Comp 2

```
def update_val(db):
      val = db.get_val()
      val = val + 1
      db.update_val(val)
```

```
  def update_val(db):
        val = db.get_val()
➡      val = val + 1          (val: 3)
➡      db.update_val(val)
```



DB

val: 3

Comp 1

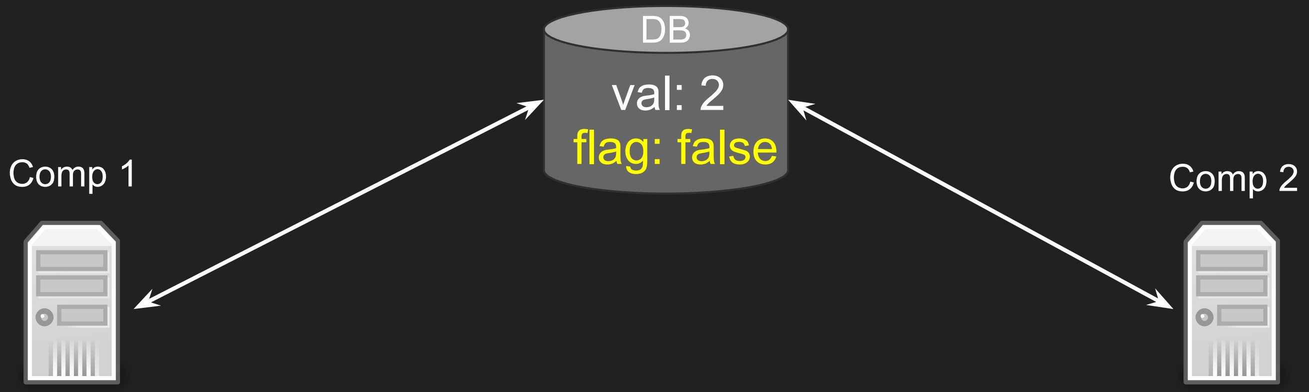
Comp 2

```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```

```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```

# Concurrency

- Race Condition
  - When a system’s behavior depends on timing, sequencing, or other uncontrollable events
  - Bugs occurring from race conditions are often hard to pin down
  - Nondeterministic - “Heisenbug”
  - Multiple threads accessing shared state is a classic place where race conditions happen
  - The “critical section” is the area where the shared state is accessed/updated



DB

val: 2

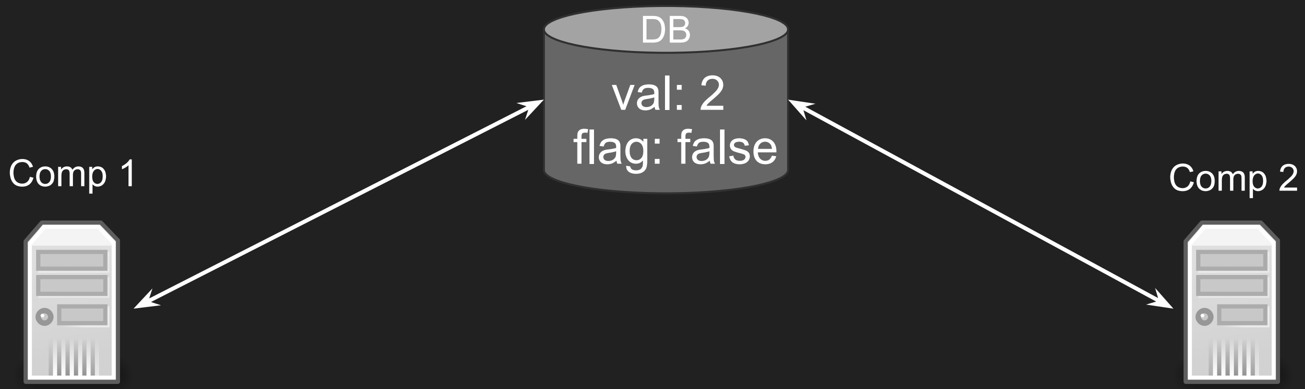
flag: false

Comp 1

Comp 2

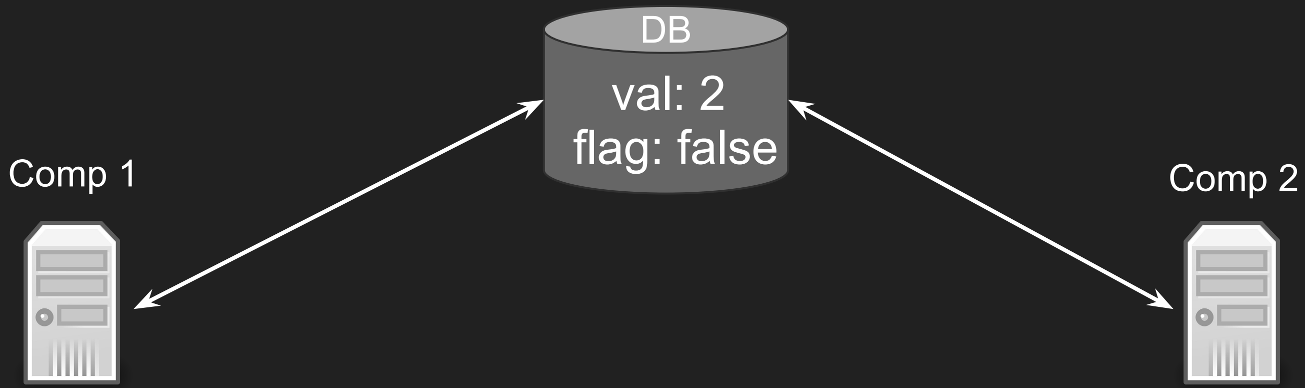
```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```

```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```



```
def update_val(db):
    while(true):
        if not db.get_flag():
            db.set_flag(true)
            break
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.update_flag(false)
```

```
def update_val(db):
    while(true):
        if not db.get_flag():
            db.set_flag(true)
            break
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.update_flag(false)
```



```
def update_val(db):
    while(true):
➡       if not db.get_flag():
            db.set_flag(true)
            break
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.update_flag(false)
```

```
def update_val(db):
    while(true):
➡       if not db.get_flag():
            db.set_flag(true)
            break
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.update_flag(false)
```

# Concurrency

- There always seems to be a way to “interrupt” the flag checking/setting
- Need something “atomic”
  - Atomic == indivisible; guaranteed to all happen as one single operation
- test_and_set
  - Sets a boolean memory value to true and returns what the value WAS as one atomic operation
  - Can be used to create a spin lock

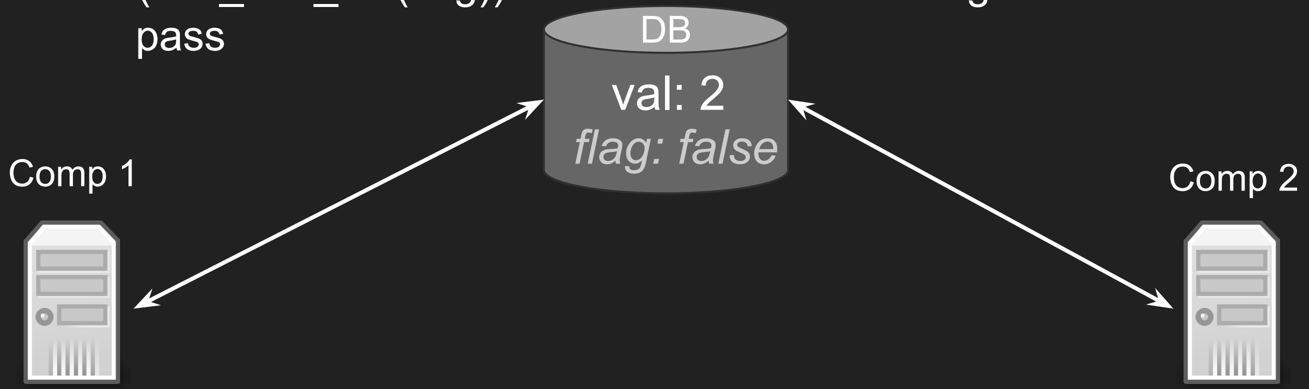
```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

- Generally abstracted into another data structure called a “mutex”
  - Stands for “mutual exclusion”

```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

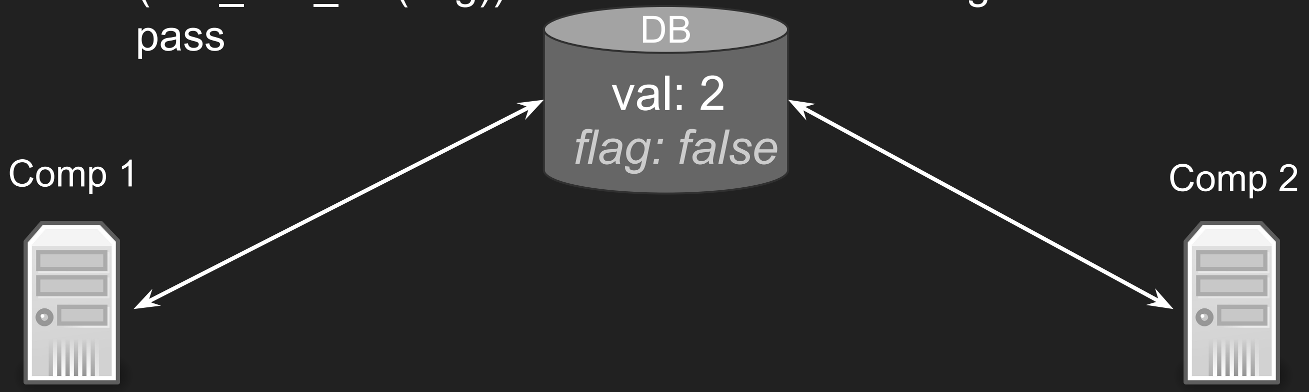


```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```

```
def update_val(db):
    val = db.get_val()
    val = val + 1
    db.update_val(val)
```

```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

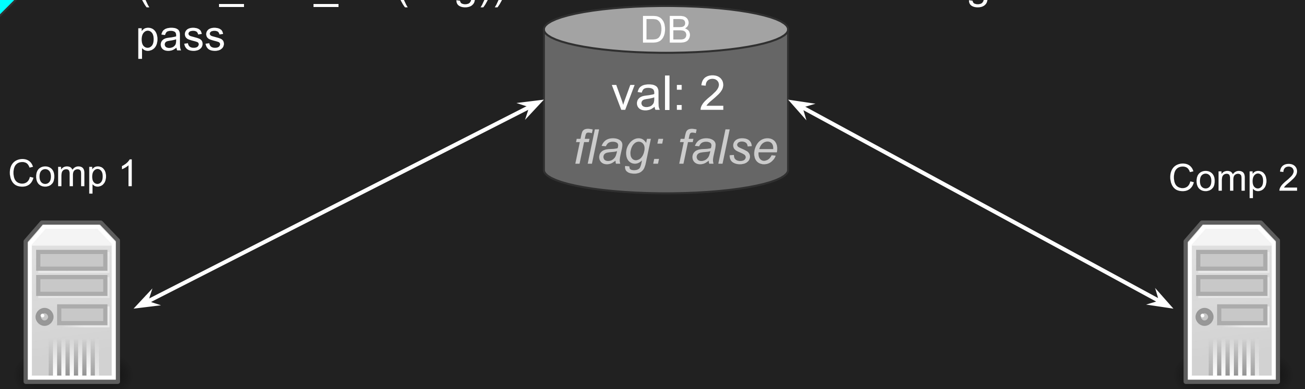


```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

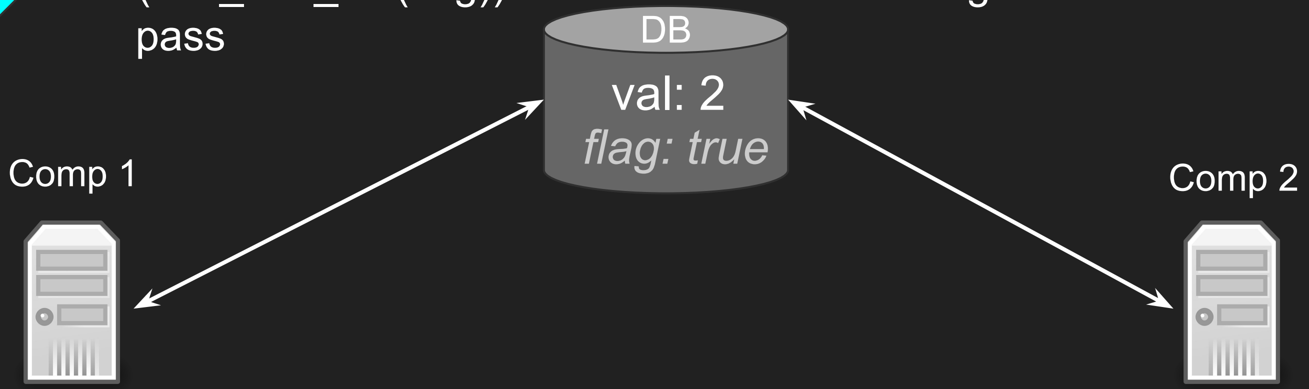


```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

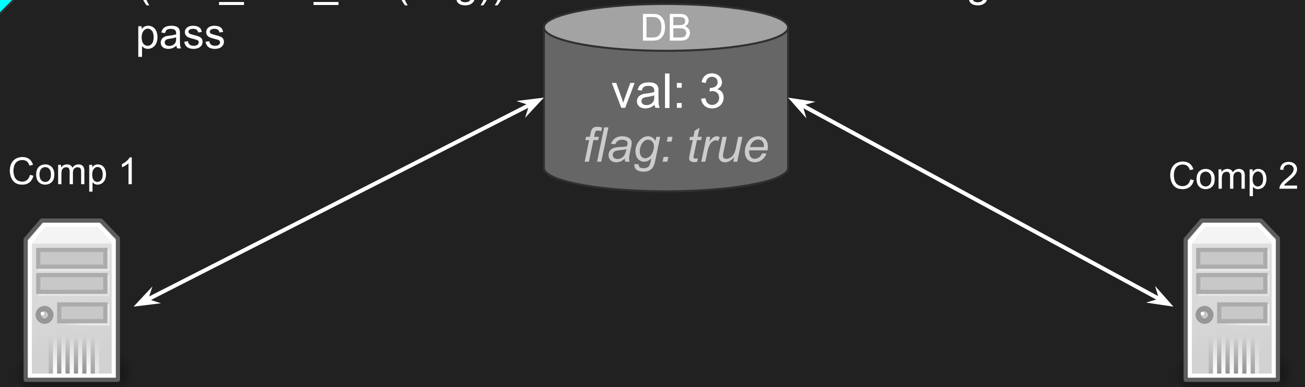


```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

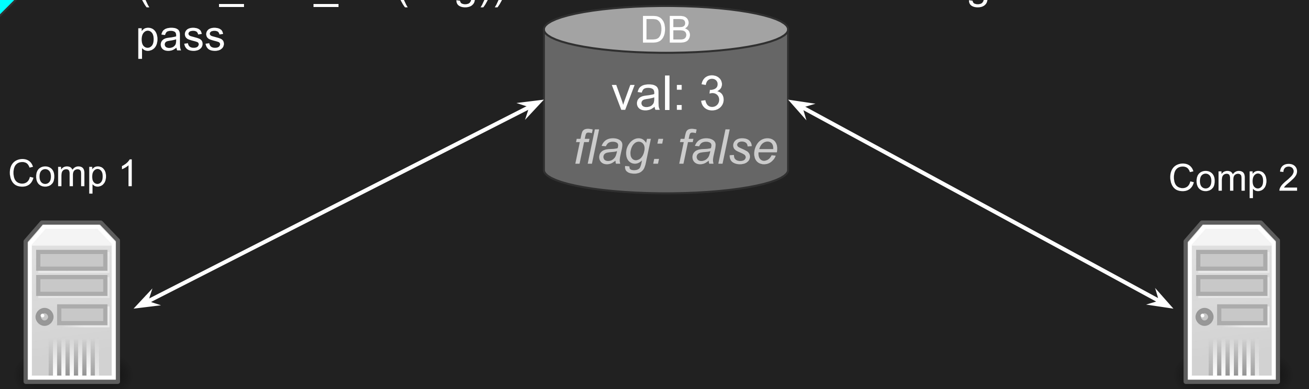


```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

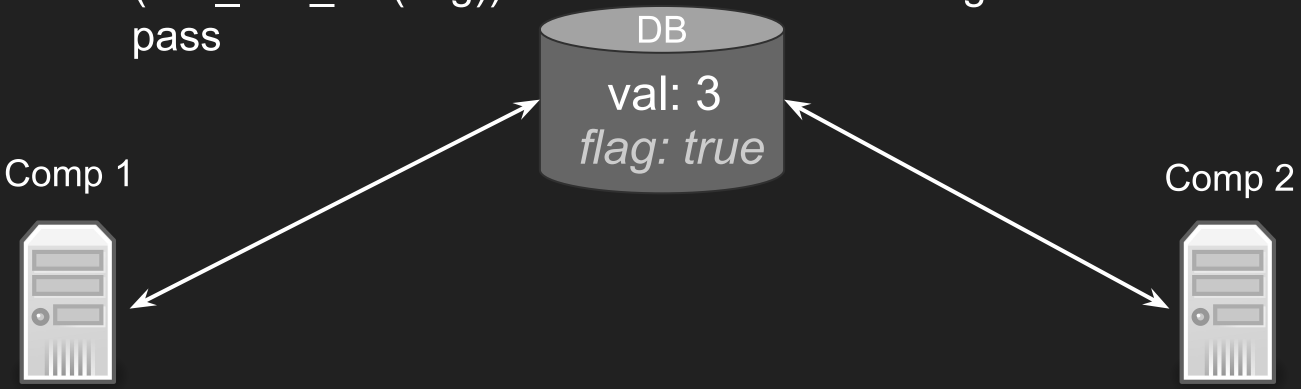


```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

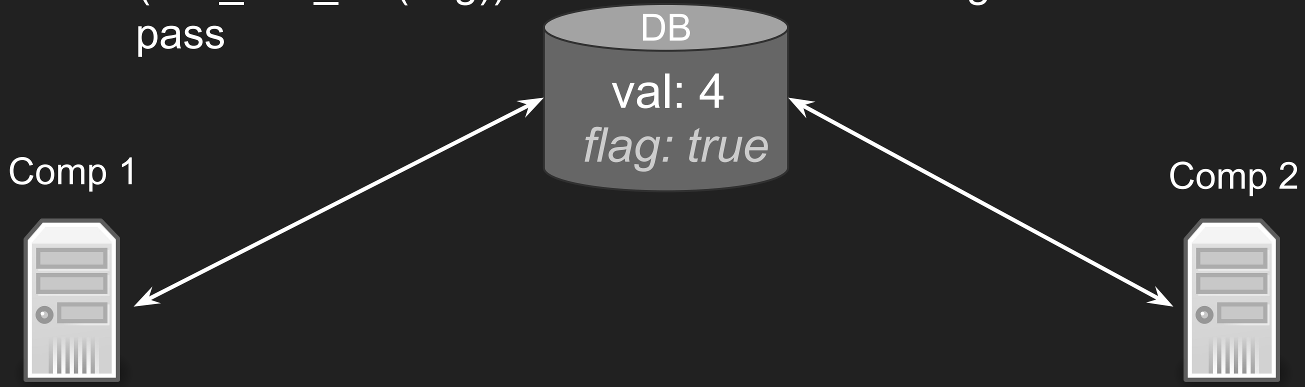


```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```

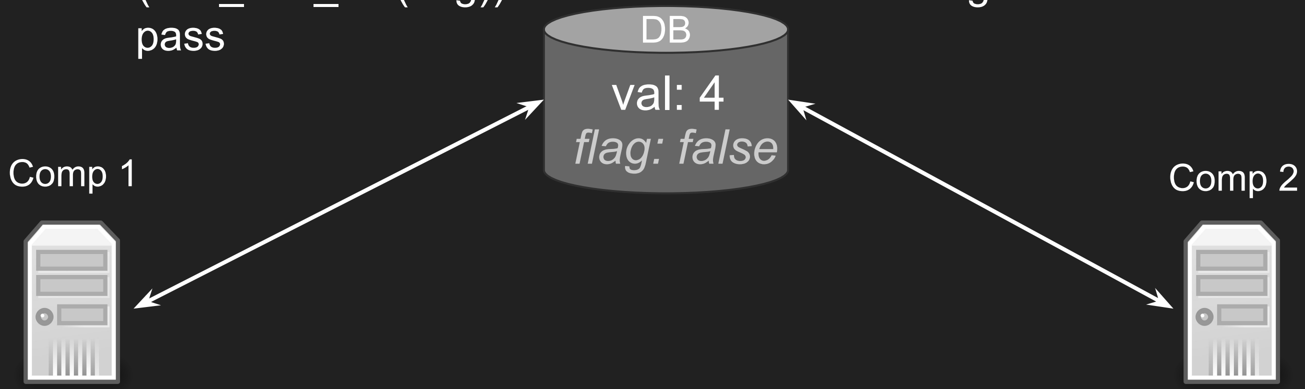


```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

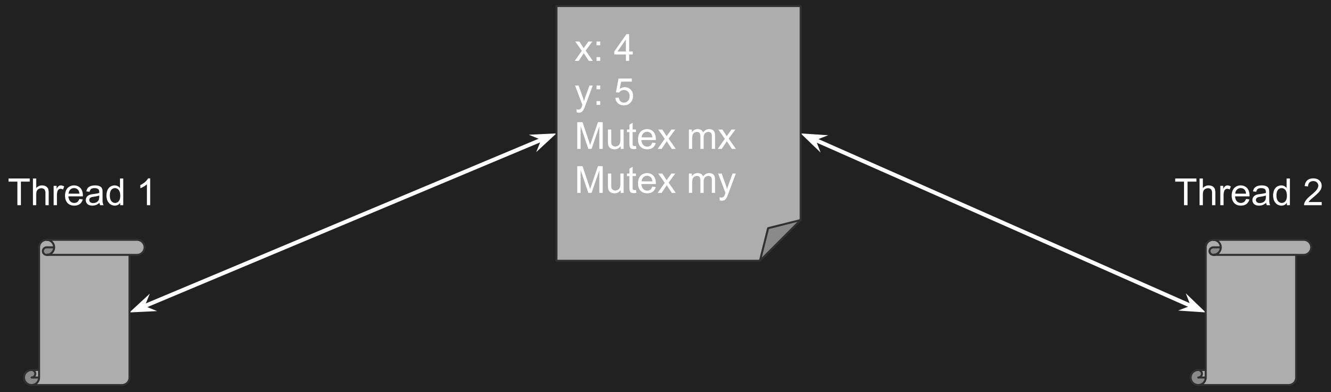
```
def acquire_lock():
    while(test_and_set(flag)):
        pass
```

```
def release_lock():
    flag = false
```



```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```

```
def update_val(db):
    db.acquire_lock()
    val = db.get_val()
    val = val + 1
    db.update_val(val)
    db.release_lock()
```



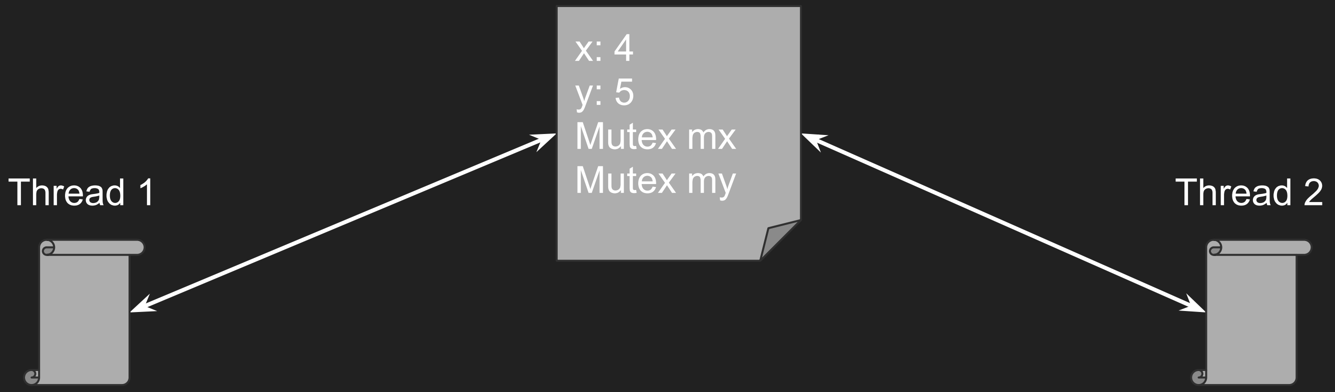
Thread 1

x: 4
y: 5
Mutex mx
Mutex my

Thread 2

```
def update_x():
    mx.acquire_lock()
    my.acquire_lock()
    x = x + y
    my.release_lock()
    mx.release_lock()
```

```
def update_x():
    my.acquire_lock()
    mx.acquire_lock()
    x = x + y
    mx.release_lock()
    my.release_lock()
```



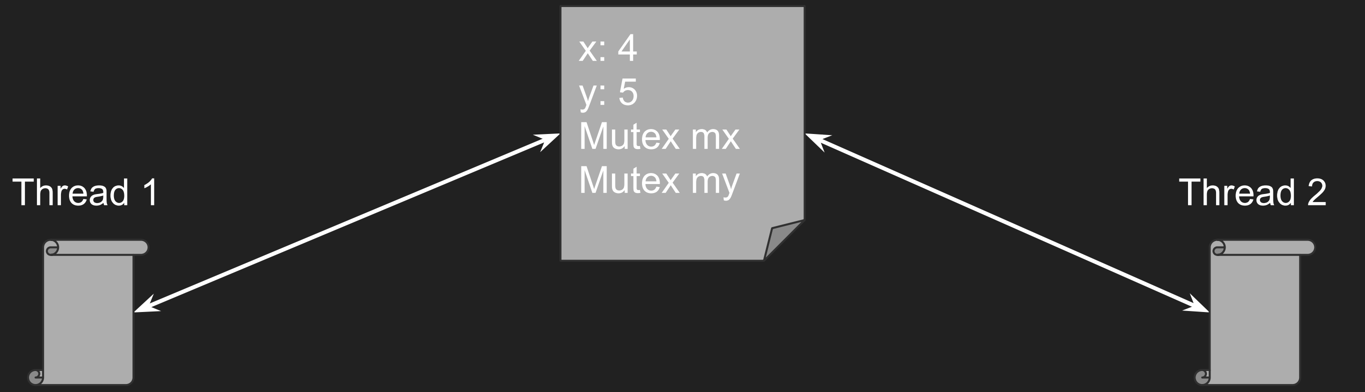
Thread 1

x: 4
y: 5
Mutex mx
Mutex my

Thread 2

```
def update_x():
        mx.acquire_lock()
        my.acquire_lock()
        x = x + y
        my.release_lock()
        mx.release_lock()
```

```
def update_x():
        my.acquire_lock()
        mx.acquire_lock()
        x = x + y
        mx.release_lock()
        my.release_lock()
```

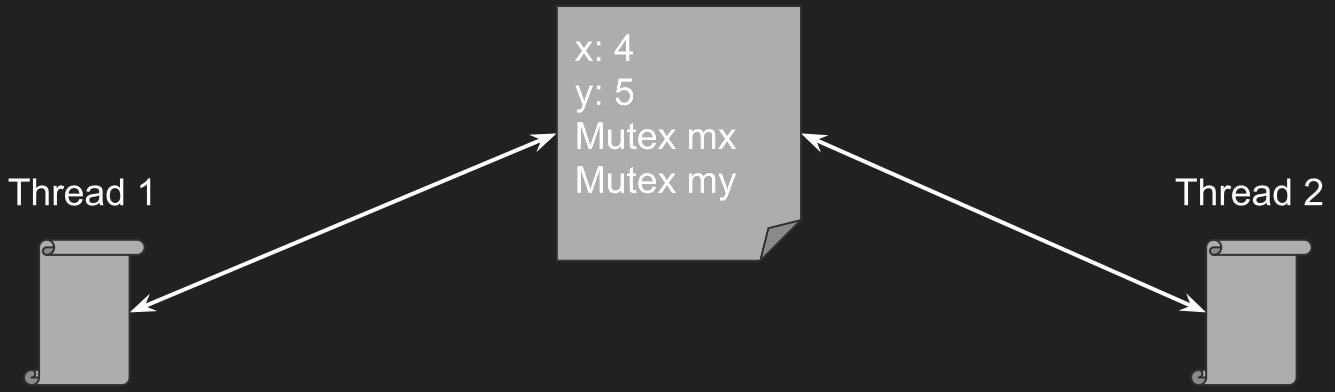


```
def update_x():
➡   mx.acquire_lock()
➡   my.acquire_lock()
    x = x + y
    my.release_lock()
    mx.release_lock()
```

```
def update_x():
➡   my.acquire_lock()
➡   mx.acquire_lock()
    x = x + y
    mx.release_lock()
    my.release_lock()
```

# Concurrency

- Mutual Exclusion can be the cause of *deadlocks*
    - Deadlock is when threads are not progressing because they’re circularly waiting on each other
- Deadlock requires four things:
    - Mutual exclusion (locks around a critical section)
    - Resource holding (already having lock for one resource, and holding it while requesting another)
    - No pre-emption (no way of breaking locks early)
    - Circular wait (thread 1 is waiting on thread 2, 2 on 3, 3 on 4, etc…. n on 1)
- Fixing deadlock requires mitigating one of those four problems
    - In practice, the most straight-forward is avoiding circular wait by ordering the locks
    - “Ostrich Algorithm” - just hope it doesn’t happen



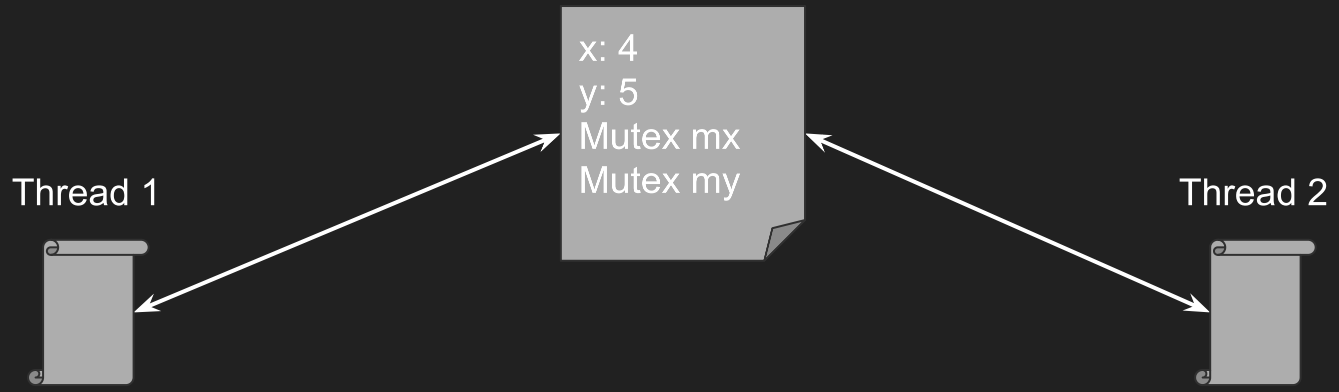
Thread 1

x: 4
y: 5
Mutex mx
Mutex my

Thread 2

```
def update_x():
      mx.acquire_lock()
      my.acquire_lock()
      x = x + y
      my.release_lock()
      mx.release_lock()
```

```
def update_x():
      my.acquire_lock()
      mx.acquire_lock()
      x = x + y
      mx.release_lock()
      my.release_lock()
```



x: 4
y: 5
Mutex mx
Mutex my

Thread 1

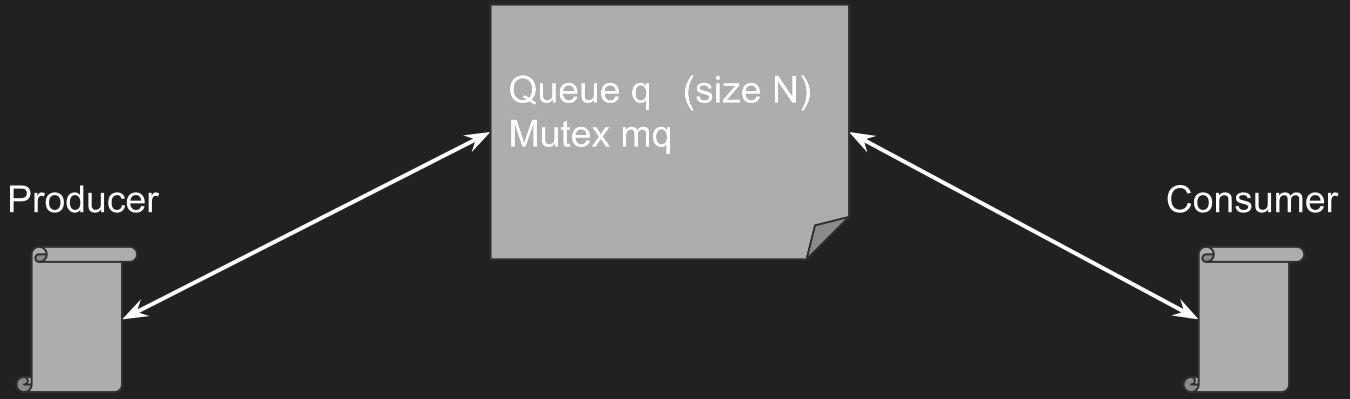
Thread 2

```
def update_x():
        mx.acquire_lock()
        my.acquire_lock()
        x = x + y
        my.release_lock()
        mx.release_lock()
```

```
def update_x():
        mx.acquire_lock()
        my.acquire_lock()
        x = x + y
        my.release_lock()
        mx.release_lock()
```

# Concurrency

- Resource Starvation
  - Many threads may be waiting on a lock, which gets the lock next?
  - If unlucky, some threads may never run at all
  - Scheduling, age prioritization, etc. can help fix this
- Livelock
  - Elements of the system are changing, but none are making any practical progress
  - Can be a risk for deadlock-detection systems



Queue q (size N)
Mutex mq

Producer

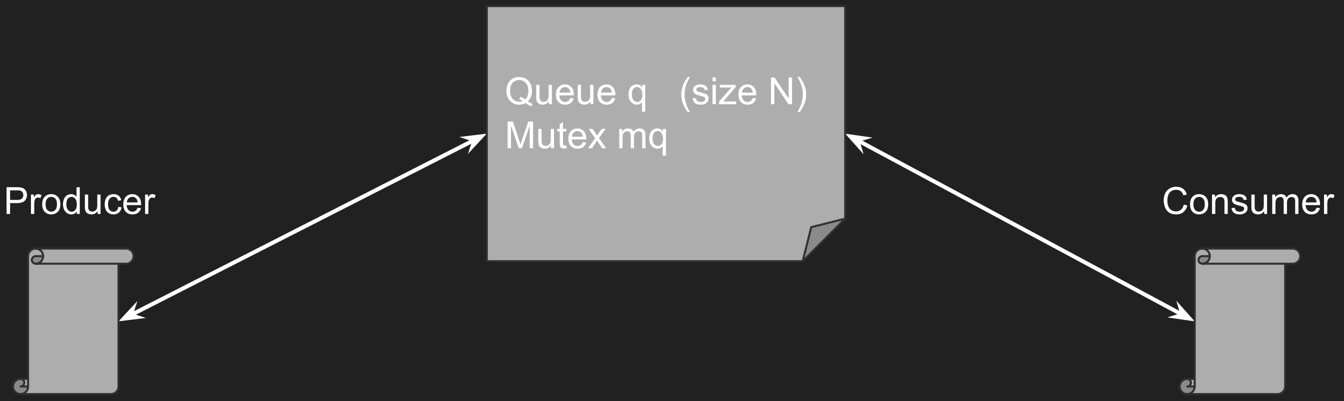
Consumer

```
def produce():
    while(true):
        mq.acquire_lock()
        if q.isFull():
            mq.release_lock()
            continue
        break
    q.add(new Item())
    mq.release_lock()
```

```
def consume():
    while(true):
        mq.acquire_lock()
        if q.isEmpty():
            mq.release_lock()
            continue
        break
    i = q.remove()
    mq.release_lock()
```

# Concurrency

- Threads could spend a lot of time spin-waiting for the queue to be in the right state
    - Waste of CPU resources!
- Solution: semaphores
    - Semaphores are essentially a “count” of how much resource is available
    - Using a resource decrements the count, releasing it increments the count
    - If no resources are available, a decrement just waits
    - A mutex is essentially a semaphore with 1 resource count



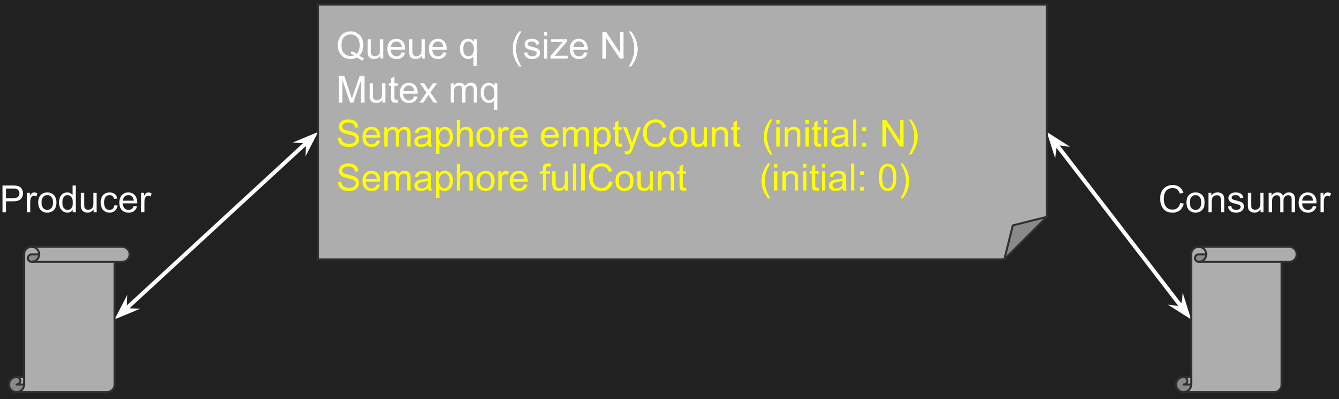
Queue q (size N)
Mutex mq

Producer

Consumer

```
def produce():
    while(true):
        mq.acquire_lock()
        if q.isFull():
            mq.release_lock()
            continue
        break
    q.add(new Item())
    mq.release_lock()
```

```
def consume():
    while(true):
        mq.acquire_lock()
        if q.isEmpty():
            mq.release_lock()
            continue
        break
    i = q.remove()
    mq.release_lock()
```



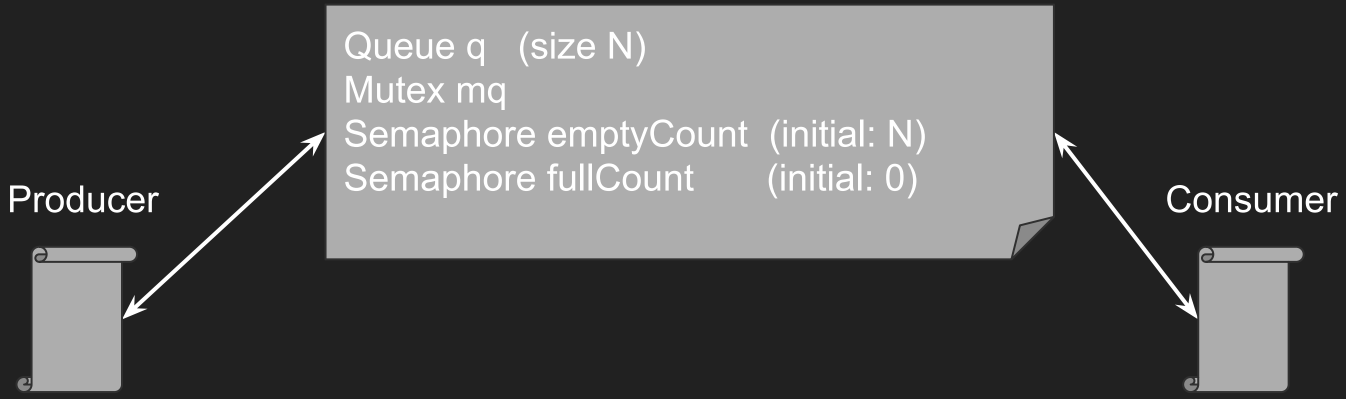
Queue q (size N)
Mutex mq
Semaphore emptyCount (initial: N)
Semaphore fullCount (initial: 0)

Producer

Consumer

```
def produce():
    while(true):
        mq.acquire_lock()
        if q.isFull():
            mq.release_lock()
            continue
        break
    q.add(new Item())
    mq.release_lock()
```

```
def consume():
    while(true):
        mq.acquire_lock()
        if q.isEmpty():
            mq.release_lock()
            continue
        break
    i = q.remove()
    mq.release_lock()
```



Producer

Queue q (size N)
Mutex mq
Semaphore emptyCount (initial: N)
Semaphore fullCount (initial: 0)

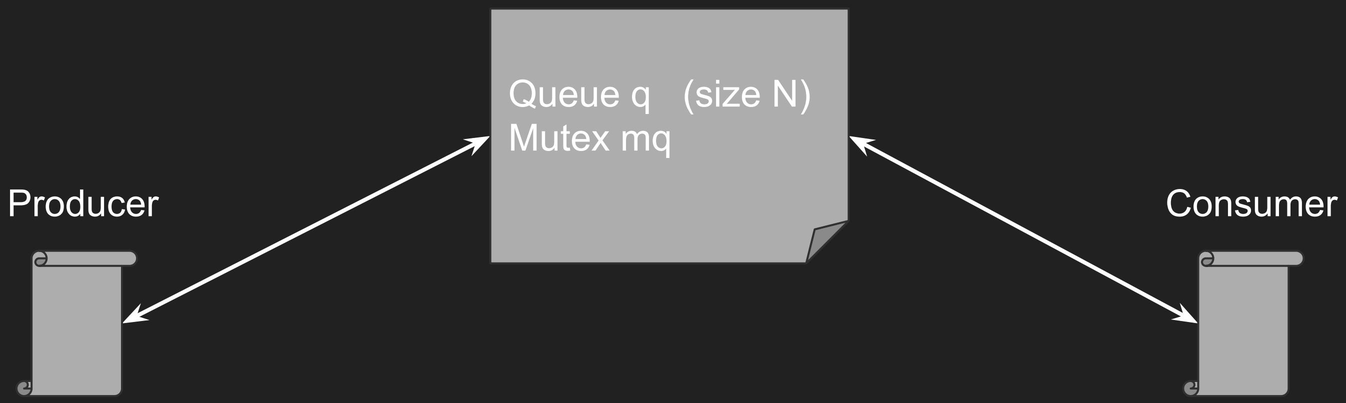
Consumer

```
def produce():
      emptyCount.decrement()
      mq.acquire_lock()
      q.add(new Item())
      mq.release_lock()
      fullCount.increment()
```

```
def consume():
      fullCount.decrement()
      mq.acquire_lock()
      i = q.remove()
      mq.release_lock()
      emptyCount.increment()
```

# Concurrency

- Semaphores are just a counter!
  - If implemented naively, could end up like `val` in the first example
  - Semaphore increment/decrement also needs to happen atomically
- Alternate solution: Condition Variables
  - Condition variables have two methods:
    - wait: release mutex and sleep until awoken
    - notify: wake up a sleeping thread and allow them to automatically re-take the mutex
  - The combination of a condition variable and a mutex is called a *monitor*



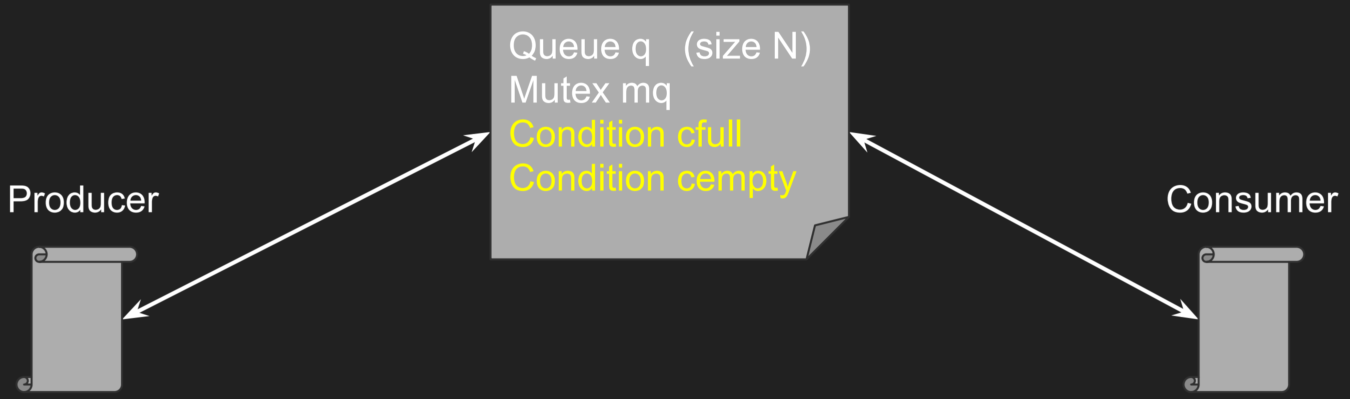
Queue q (size N)
Mutex mq

Producer

Consumer

```
def produce():
    while(true):
        mq.acquire_lock()
        if q.isFull():
            mq.release_lock()
            continue
        break
    q.add(new Item())
    mq.release_lock()
```

```
def consume():
    while(true):
        mq.acquire_lock()
        if q.isEmpty():
            mq.release_lock()
            continue
        break
    i = q.remove()
    mq.release_lock()
```



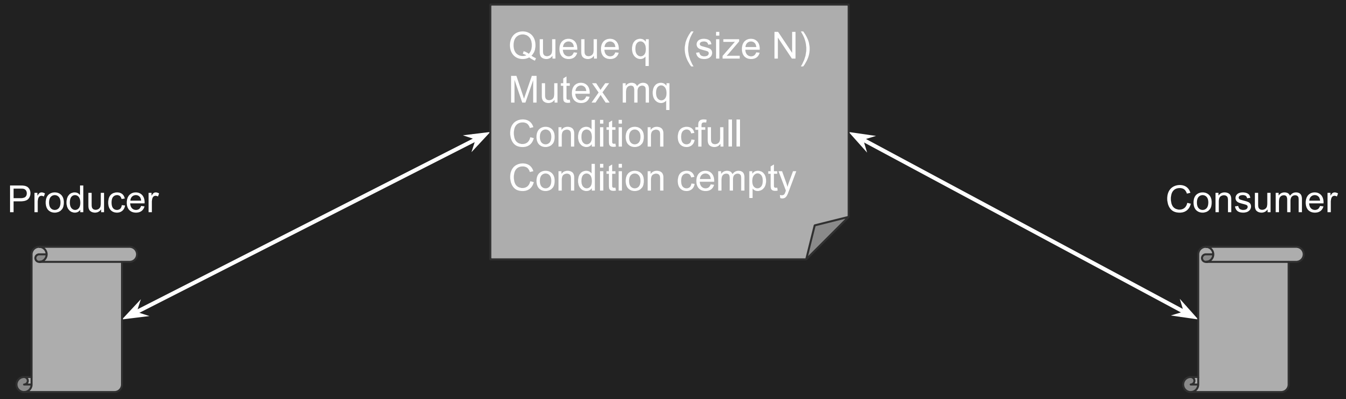
Queue q (size N)
Mutex mq
Condition cfull
Condition cempty

Producer

Consumer

```
def produce():
    while(true):
        mq.acquire_lock()
        if q.isFull():
            mq.release_lock()
            continue
        break
    q.add(new Item())
    mq.release_lock()
```

```
def consume():
    while(true):
        mq.acquire_lock()
        if q.isEmpty():
            mq.release_lock()
            continue
        break
    i = q.remove()
    mq.release_lock()
```



Queue q   (size N)
Mutex mq
Condition cfull
Condition cempty

Producer

Consumer

```
def produce():
    mq.acquire_lock()
    while(q.isFull()):
        cfull.wait(mq)
    q.add(new Item())
    mq.release_lock()
    cempty.notify()
```

```
def consume():
    mq.acquire_lock()
    while(q.isEmpty()):
        cempty.wait(mq)
    I = q.remove()
    mq.release_lock()
    cfull.notify()
```