Synchronization

CSE 306 Operating Systems

Dongyoon Lee

A Simple (Sequential) Queue

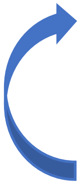
```
void enqueue (node *new_element) {
        node *p;

        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;

}

node *dequeue () {
        node *element = null;

        if (head ->next != null) {
                element = head->next;
                head->next = head->next->next;
        }

        return element;
}
```

Lock-based Concurrent Queue

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        if (head ->next != null) {
                element = head->next;
                head->next = head->next->next;
        }
        UNLOCK(L);
        return element;
}
```

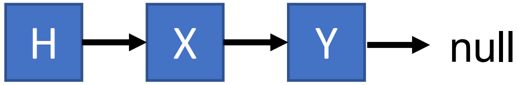
What could go wrong?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        if (head ->next != null) {
                element = head->next;
                head->next = head->next->next;
        }
        UNLOCK(L);
        return element;
}
```



What could go wrong?

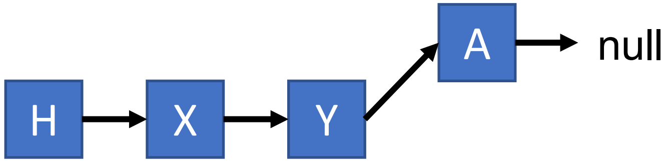
```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        if (head ->next != null) {
                element = head->next;
                head->next = head->next->next;
        }
        UNLOCK(L);
        return element;
}
```



Two enq()s may find the same tail concurrently.

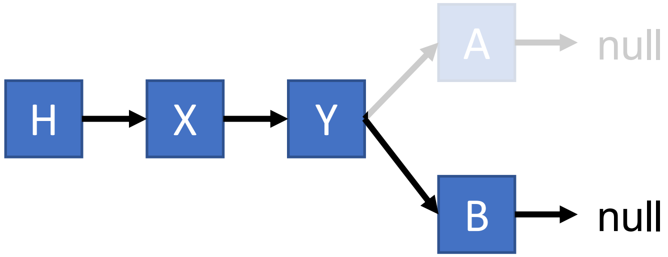
What could go wrong?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        if (head ->next != null) {
                element = head->next;
                head->next = head->next->next;
        }
        UNLOCK(L);
        return element;
}
```



What could go wrong?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        if (head ->next != null) {
                element = head->next;
                head->next = head->next->next;
        }
        UNLOCK(L);
        return element;
}
```



Can we make dequeue wait, if empty?

```
void enqueue (node *new_element) {
    node *p;
    LOCK(L);
    for (p = head; p->next != NULL; p = p->next) ;
    p->next = new_element;
    new_element->next = null;
    UNLOCK(L);
}

node *dequeue () {
    node *element = null;
    LOCK(L);
    if (head ->next != null) {
        element = head->next;
        head->next = head->next->next;
    }
    UNLOCK(L);
    return element;
}
```

Can we make dequeue wait, if empty?

```
void enqueue (node *new_element) {
    node *p;
    LOCK(L);
    for (p = head; p->next != NULL; p = p->next) ;
    p->next = new_element;
    new_element->next = null;
    UNLOCK(L);
}

node *dequeue () {
    node *element = null;
    LOCK(L);
    if (head ->next != null) {
        element = head->next;
        head->next = head->next->next;
    }
    UNLOCK(L);
    return element;
}
```

Add “while”-based waiting

What could go wrong?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        while (head ->next == null) {}

        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

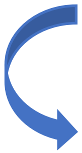
What could go wrong?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        while (head ->next == null) {}

        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

Problem: Deadlock (hold Lock L forever)

What if we acquire Lock after waiting?

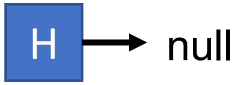
```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        while (head ->next == null) {}

        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```



What if we acquire Lock after waiting?

```
void enqueue (node *new_element) {
    node *p;
    LOCK(L);
    for (p = head; p->next != NULL; p = p->next) ;
    p->next = new_element;
    new_element->next = null;
    UNLOCK(L);
}

node *dequeue () {
    node *element = null;

    while (head ->next == null) {}
    LOCK(L);
    element = head->next;
    head->next = head->next->next;
    UNLOCK(L);
    return element;
}
```

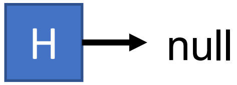
Concurrent deq(), deq(), enq(A)



What if we acquire Lock after waiting?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;

        while (head ->next == null) {}
        LOCK(L);
        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

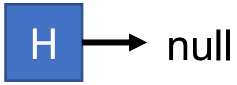
Concurrent deq(), deq(), enq(A)



What if we acquire Lock after waiting?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;

        while (head ->next == null) {}
        LOCK(L);
        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

Concurrent deq(), deq(), enq(A)



Two deq()s may find the queue not-empty concurrently.

A possible solution. Another problem?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        while (head ->next == null) {UNLOCK(L); LOCK(L);}

        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

A possible solution. Another problem?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        UNLOCK(L);
}

node *dequeue () {
        node *element = null;
        LOCK(L);
        while (head ->next == null) {UNLOCK(L); LOCK(L);}

        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

Problem: Busy waiting.

How to avoid busy waiting?

```
node *dequeue () {
        node *element = null;
        LOCK(L);
        while (head ->next == null) {
                UNLOCK(L);


                LOCK(L);
        }
        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

How to avoid busy waiting?

```
node *dequeue () {
        node *element = null;
        LOCK(L);
        while (head ->next == null) {
                UNLOCK(L);
                add myself into a waitlist;
                go to sleep;
                LOCK(L);
        }
        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

WAIT (L, CV)

- **Atomically by OS**

How to avoid busy waiting?

```
node *dequeue () {
    node *element = null;
    LOCK(L);
    while (head ->next == null) {
        UNLOCK(L);
        add myself into a waitlist;
        go to sleep;
        LOCK(L);
    }
    element = head->next;
    head->next = head->next->next;
    UNLOCK(L);
    return element;
}
```

WAIT (L, CV)

- **Atomically by OS**
- **If not,**
  - **T1: put into a waitlist. before go to sleep**
  - **T2: wake up and send signal**
  - **T1: go to sleep**
  - **We “lost the wake-up signal”.**

How to avoid busy waiting?

```
void enqueue (node *new_element) {
    node *p;
    LOCK(L);
    for (p = head; p->next != NULL; p = p->next) ;
    p->next = new_element;
    new_element->next = null;
    take a waiter off the waitlist;
    wake up;
    UNLOCK(L);
}
```

How to avoid busy waiting?

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        take a waiter off the waitlist;
        wake up;
        UNLOCK(L);
}
```

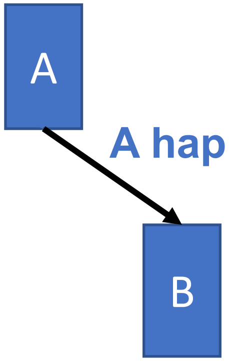
SIGNAL (CV) or BROADCAST (CV)

Locks and Conditional Variables

```
void enqueue (node *new_element) {
        node *p;
        LOCK(L);
        for (p = head; p->next != NULL; p = p->next) ;
        p->next = new_element;
        new_element->next = null;
        SIGNAL(CV);
        UNLOCK(L);
}
node *dequeue () {
        node *element = null;
        LOCK(L);
        while (head ->next == null) {
                WAIT(L, CV);
        }
        element = head->next;
        head->next = head->next->next;
        UNLOCK(L);
        return element;
}
```

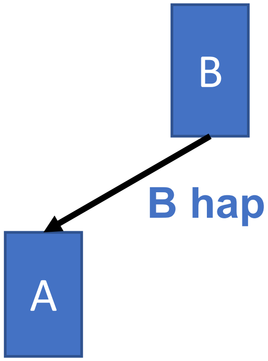
Synchronization: Don’t do simultaneously!

- Locks: Mutual exclusion



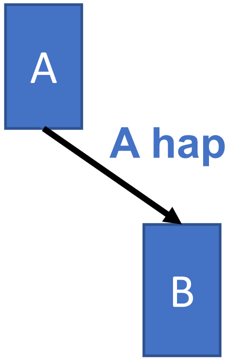
A happens-before B

or



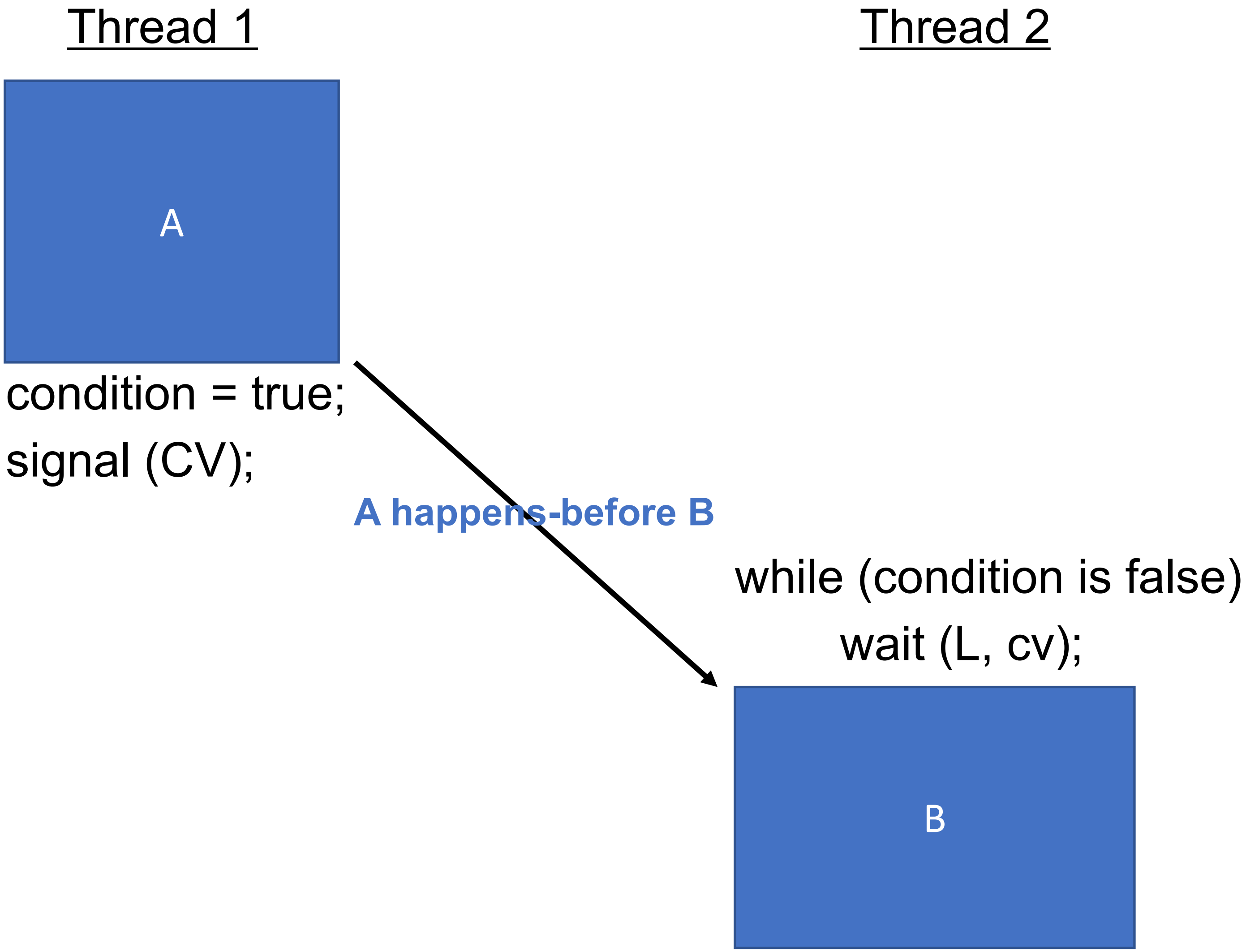
B happens-before A

- Locks + CVs: Ordering constraint



A happens-before B

Monitor: 1 lock and 0~n conditional vars



Thread 1

A

```
condition = true;
signal (CV);
```

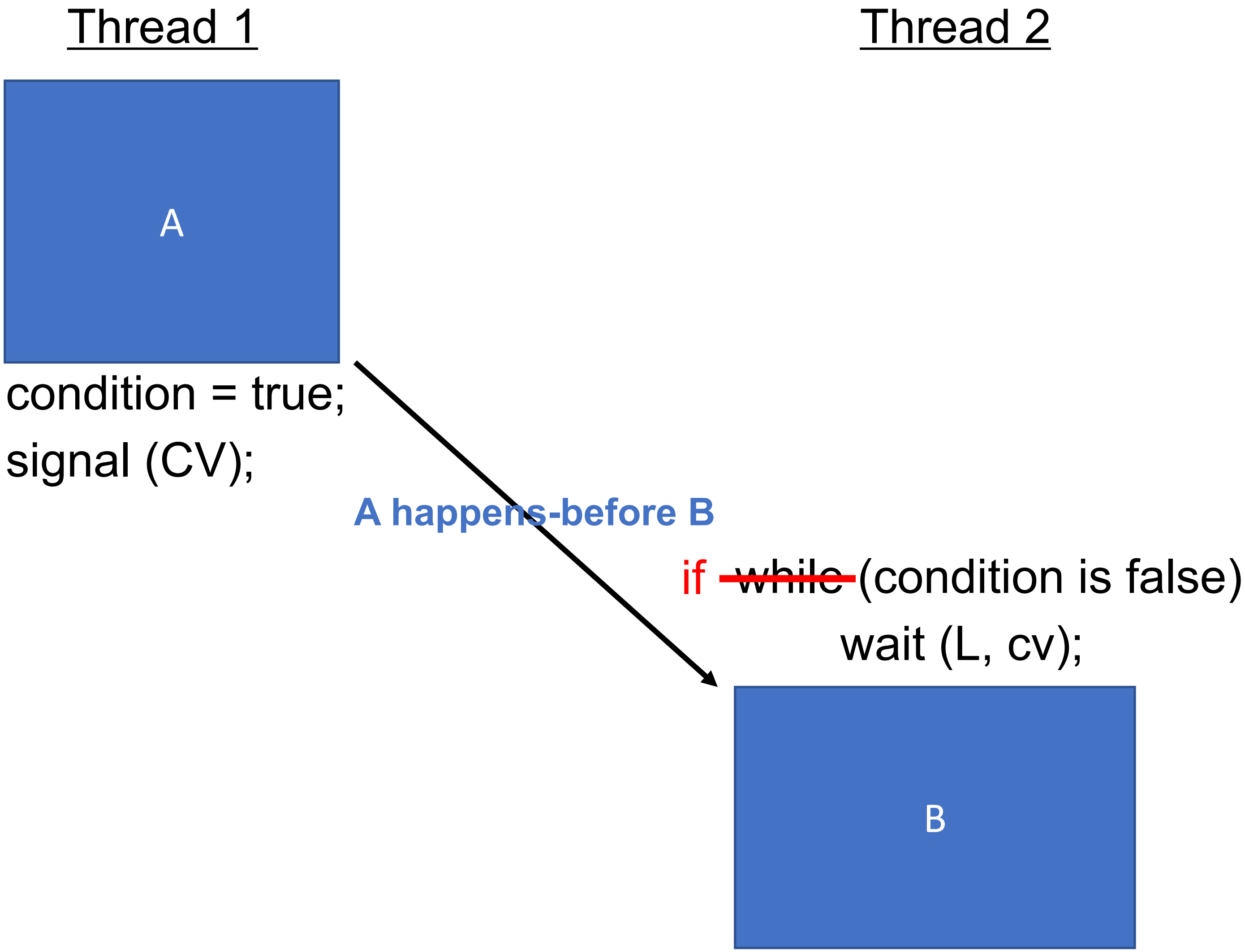
A happens-before B

Thread 2

```
while (condition is false)
     wait (L, cv);
```

B

Monitor: 1 lock and 0~n conditional vars



Thread 1

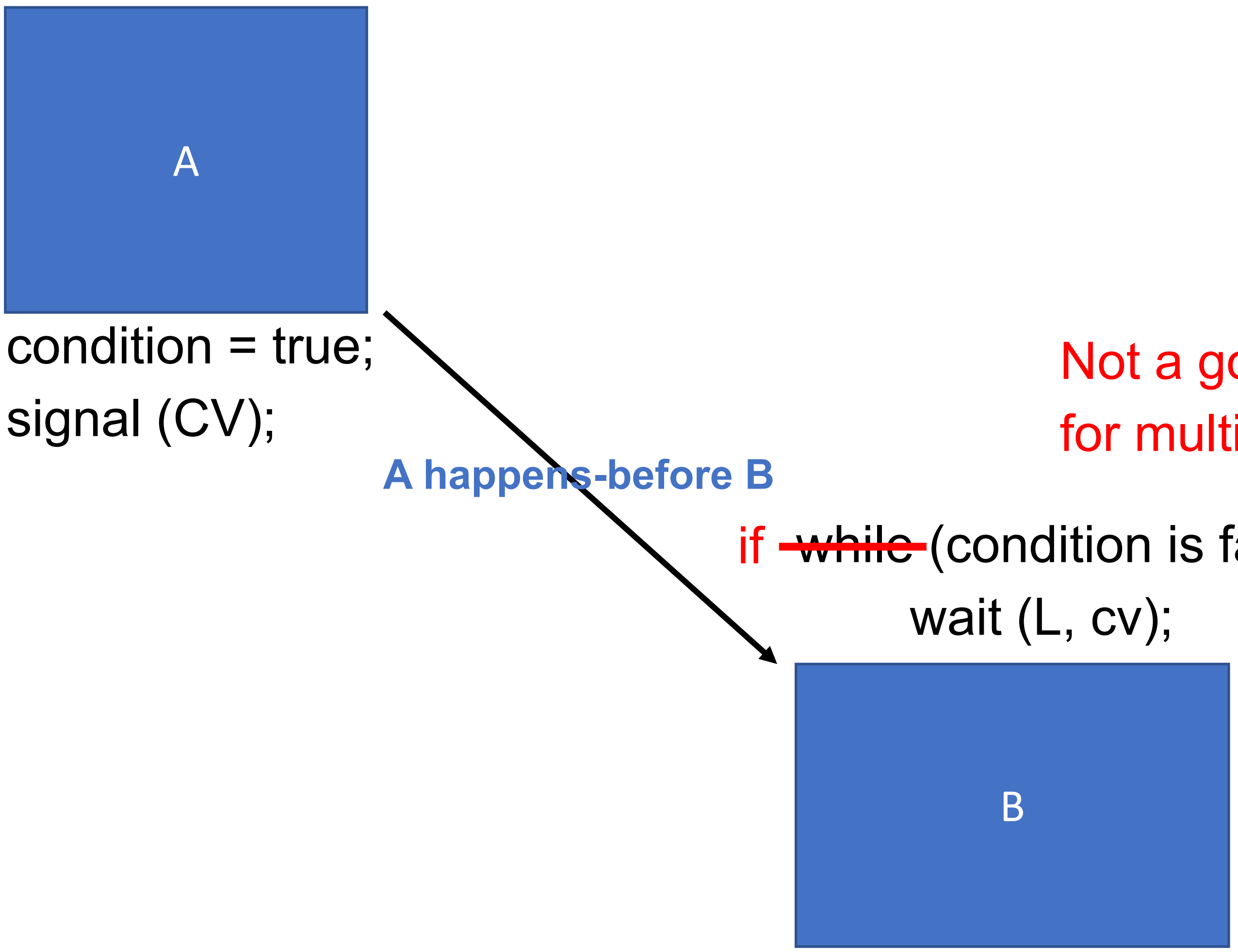
Thread 2

```
condition = true;
signal (CV);
```

A happens-before B

```
if ~~while~~ (condition is false)
    wait (L, cv);
```

Monitor: 1 lock and 0~n conditional vars



Thread 1

condition = true;

signal (CV);

A happens-before B

Thread 2

Not a good practice for multiple waiters

if ~~while~~ (condition is false)

wait (L, cv);

How to build a monitor (step by step)

1. Identify shared data
2. Assign locks to shared data
3. Add lock (before r/w) and unlock (after r/w)
4. List before and after condition (e.g., A has to happen before B) and assign a conditional variable to each one
5. Add wait and while loop
6. Add signal or broadcast (whenever a condition is changed)
7. Make sure reestablish program invariants

Coke Machine Producer and Consumer

```
int numCoke;
Queue
MAX_COKES
```

```
00:  Consumer () {
01:
02:
03:
04:
05:          takeCokeOutofMachine();
06:          numCoke--;
07:
08:
09:  }
```

```
10:  Producer () {
11:
12:
13:
14:
15:          putCokeIntoMachine();
16:          numCoke++;
17:
18:
19:  }
```

Coke Machine Producer and Consumer

1. Identify shared data

```
int numCoke;
Queue
MAX_COKES
```

2. Assign locks: CokeLock

```
00:  Consumer () {
01:
02:
03:
04:
05:          takeCokeOutofMachine();
06:          numCoke--;
07:
08:
09:  }
```

```
10:  Producer () {
11:
12:
13:
14:
15:          putCokeIntoMachine();
16:          numCoke++;
17:
18:
19:  }
```

Coke Machine Producer and Consumer

1. Identify shared data

```
int numCoke;
Queue
MAX_COKES
```

2. Assign locks: CokeLock

3. Add lock/unlock

```
Consumer () {
        LOCK(CokeLock);



        takeCokeOutofMachine();
        numCoke--;

        UNLOCK(CokeLock);
}
```

```
Producer () {
        LOCK(CokeLock);



        putCokeIntoMachine();
        numCoke++;

        UNLOCK(CokeLock);
}
```

Coke Machine Producer and Consumer

4. List before/after conditions
and assign conditional variables

```
int numCoke;
Queue
MAX_COKES
```

A) Must be >0 cokes
before we take a Coke out

B) Must be < MAX_COKES cokes
before we put a Coke in

```
Consumer () {
        LOCK(CokeLock);



        takeCokeOutofMachine();
        numCoke--;

        UNLOCK(CokeLock);
}
```

```
Producer () {
        LOCK(CokeLock);



        putCokeIntoMachine();
        numCoke++;

        UNLOCK(CokeLock);
}
```

Coke Machine Producer and Consumer

4. List before/after conditions
and assign conditional variables

```
int numCoke;
Queue
MAX_COKES
```

noCoke A) Must be >0 cokes
before we take a Coke out

noSpace B) Must be < MAX_COKES cokes
before we put a Coke in

```
Consumer () {
        LOCK(CokeLock);



        takeCokeOutofMachine();
        numCoke--;

        UNLOCK(CokeLock);
}
```

```
Producer () {
        LOCK(CokeLock);



        putCokeIntoMachine();
        numCoke++;

        UNLOCK(CokeLock);
}
```

Coke Machine Producer and Consumer

int numCokes;
Queue
MAX_COKES

A) Must be >0 cokes
before we take a Coke out
B) Must be < MAX_COKES cokes
before we put a Coke in

5. Add wait and while loop

```
Consumer () {
        LOCK (CokeLock);
        while (numCoke == 0) {
                WAIT(CokeLock, noCoke);
        }
        takeCokeOutofMachine();
        numCoke--;

        UNLOCK(CokeLock);
}
```

```
Producer () {
        LOCK (CokeLock);
        while (numCoke == MAX_COKES) {
                WAIT(CokeLock, noSpace);
        }
        putCokeIntoMachine();
        numCoke++;

        UNLOCK(CokeLock);
}
```

Coke Machine Producer and Consumer

int numCokes;
Queue
MAX_COKES

A) Must be >0 cokes
 before we take a Coke out
B) Must be < MAX_COKES cokes
 before we put a Coke in

```
Consumer () {
    LOCK (CokeLock);
    while (numCoke == 0) {
        WAIT(CokeLock, noCoke);
    }
    takeCokeOutofMachine();
    numCoke--;
    SIGNAL(noSpace);
    UNLOCK(CokeLock);
}
```

6. Add signal

```
Producer () {
    LOCK (CokeLock);
    while (numCoke == MAX_COKES) {
        WAIT(CokeLock, noSpace);
    }
    putCokeIntoMachine();
    numCoke++;
    SIGNAL(noCoke);
    UNLOCK(CokeLock);
}
```

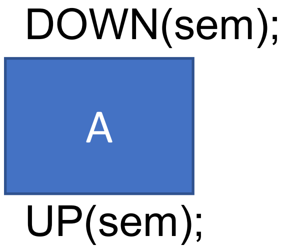
Semaphore

- Combine two abstractions (lock and conditional variables)
- Non-negative integer value
- Two operations

```
DOWN () {
        do{
                if (value > 0){    Atomic
                        value--;
                        break;
                }
        } while (1);
}

UP() {
        value++;    Atomic
}
```

Semaphore

- **Critical Section**
  - sem_init(1): initial value is 1.
  - DOWN: wait if 0, otherwise set 0. ~= LOCK
  - UP: set 1 ~= UNLOCK

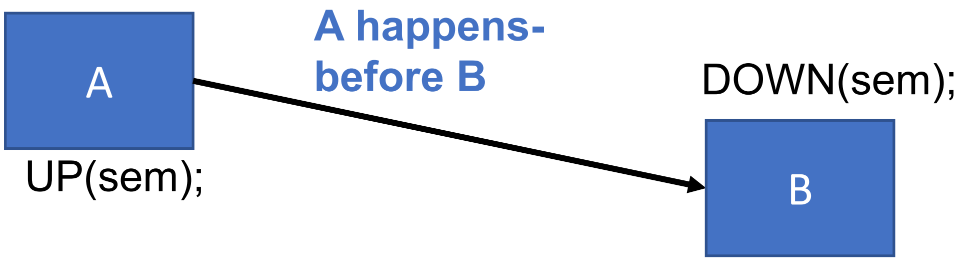


DOWN(sem);

A

UP(sem);

- **Ordering**
  - sem_init(0): initial value is 0.
  - DOWN: wait if 0, otherwise set 0. ~= WAIT
  - UP: set 1 ~= SIGNAL



A

UP(sem);

A happens-before B

DOWN(sem);

B

Coke Machine Semaphore

```
sem Mutex = sem_init (1);
```

```
Consumer () {

        down (Mutex);
        takeCokeOutofMachine();
        numCoke--;
        up (Mutex);

}
```

```
Producer () {

        down (Mutex);
        putCokeIntoMachine();
        numCoke++;
        up (Mutex);

}
```

Coke Machine Semaphore

```
sem Mutex = sem_init (1);
sem Full = sem_init (0);
```

```
Consumer () {
        down (Full);
        down (Mutex);
        takeCokeOutofMachine();
        numCoke--;
        up (Mutex);

}
```

```
Producer () {

        down (Mutex);
        putCokeIntoMachine();
        numCoke++;
        up (Mutex);
        up (Full);
}
```

Coke Machine Semaphore

```
sem Mutex = sem_init (1);
sem Full = sem_init (0);
sem Empty = sem_init (N);
```

```
Consumer () {
        down (Full);
        down (Mutex);
        takeCokeOutofMachine();
        numCoke--;
        up (Mutex);
        up (Empty);
}
```

```
Producer () {
        down (Empty);
        down (Mutex);
        putCokeIntoMachine();
        numCoke++;
        up (Mutex);
        up (Full);
}
```