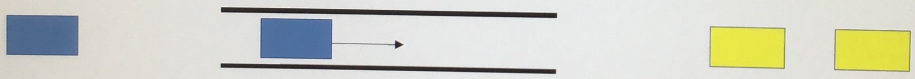
1) The exercise

GENERAL COMMENT: IF I DO NOT ZOOM IN, TELL ME TO ZOOM IN

Narrow Bridge problem

A two way east-west road contains a narrow bridge with only one lane. An eastbound (or westbound) car can pass over the bridge only if there is no oncoming car on the bridge. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. In this system, each car is represented by one thread, which executes the procedure OneVehicle when it arrives at the bridge.



```
OneVehicle(Direction direc) {
    ArriveBridge(direc);
    CrossBridge(direc);
    ExitBridge(direc); }
```

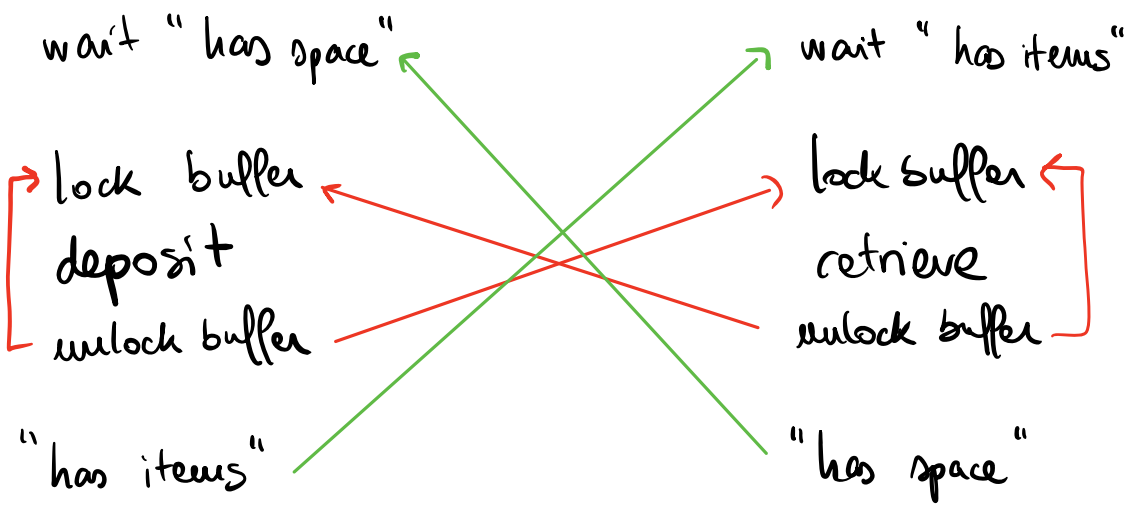
direc gives the direction in which the vehicle will cross the bridge

Write the procedures *ArriveBridge* and *ExitBridge*.

ArriveBridge must not return until it safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions or bridge collapses).

ExitBridge is called to indicate that the caller has finished crossing the bridge; *ExitBridge* should take steps to let additional cars cross the bridge.

2) conditional variables Part 1 - prods/cons with semaphores



Binary semaphore mutex // 1

General semaphore has-items // 0

General semaphore has-space // N

Producer {

// produce

wait(has-space)

wait (mutex)

// add

signal (mutex)

signal (has-items)

} ∞

Consumer {

wait(has-items)

wait(mutex)

// retrieve

signal (mutex)

signal (has-space)

// do something /consume

} ∞

3) conditional variables Part 2 - Prods/Cons with locks

Lock: token that can be held by 0 or 1 thread at a time

acquire: get the lock if the lock free, wait otherwise

release: mark the lock as free

ASSUMPTION: who calls release owns the lock

Attempt 1

```
shared count // 0
shared max_size // N
```

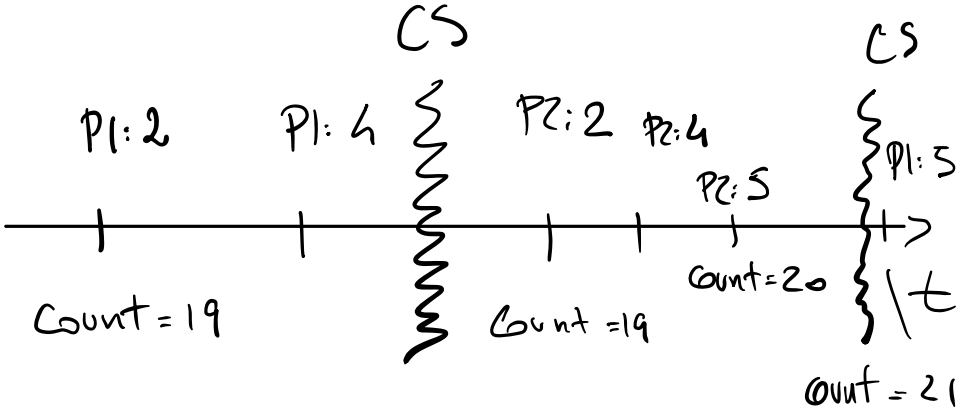
```
Producer {
1     // produce
2     while ( count== max_size){
3     }
4     // add
5     count ++
    } ∞
```

```
Consumer{
 while (count==0){
 }
 // get
 count--;
 //consumes
} ∞
```

Does it work? NO

2 producers P1 P2

N = 20



Attempt 2 : protect accou count using the lock

```
shared  count // 0
shared  max_size // N
shared  lock l
```

```
Producer {
  // produce
  acquire(l)
  while ( count== max_size){
  }
  // add
  Count ++
  release (l)
} ∞
```

```
Consumer{
1  acquire(l)
2  while (count==0){
3  }
4  // get
5  count--;
6  release (l)
7  // consumes
  } ∞
```

Does it work? NO

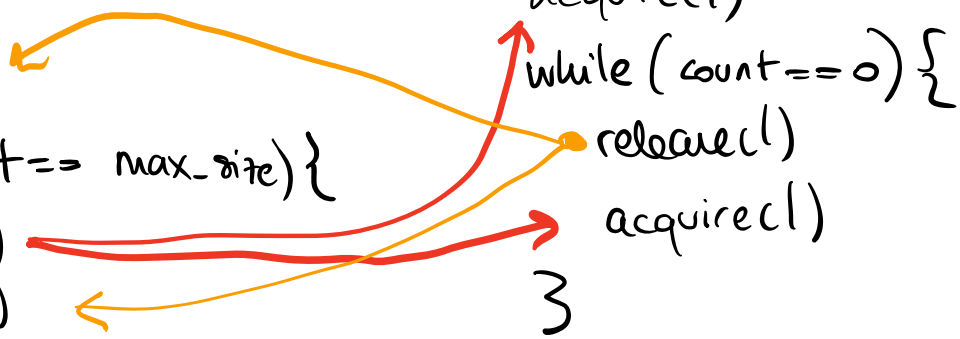
N = 20
Producer P1
Consumer C1

Attempt 3 lock v2

```
shared count // 0
shared max_size // N
shared lock l

Producer {
  // produce
  acquire(l)
  while ( count== max_size){
     release(l)
     acquire(l)
  }
  // add
  count ++
  release (l)
} ∞
```

```
Consumer{
  acquire(l)
  while (count==0) {
     release(l)
     acquire(l)
  }
  // get
  count--;
  release (l)
  // consumes
} ∞
```



Does it work? not really, if you are lucky

How can we do better?

⊕ : $P_S \wedge C_S$ call release

⊖ : $P_S \times C_S$ wake up when it is not necessarily needed

Condition variables : object, you can use it to wait for a condition to become true, associated with a lock

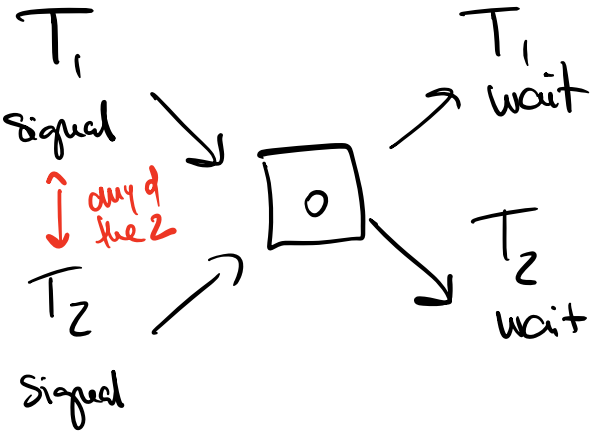
cond_wait(condition, lock) : release the lock
thread goes to sleep/waiting queue
upon waking up, re-acquires the lock

cond_signal(condition, lock) : if a thread is waiting condition, wake up 1 and give the lock

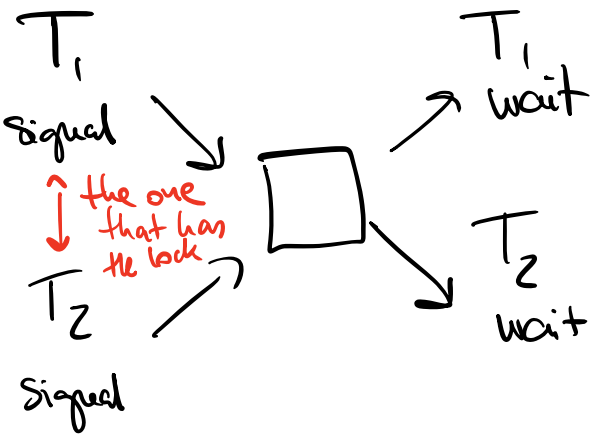
cond_broadcast(condition, lock) : = signal, except all wake up

- WHO CALLS THESE METHODS NEEDS TO HOLD THE LOCK
- FINE-GRAINED CONTROL OVER EXECUTION ORDER

semaphores



cond. vars

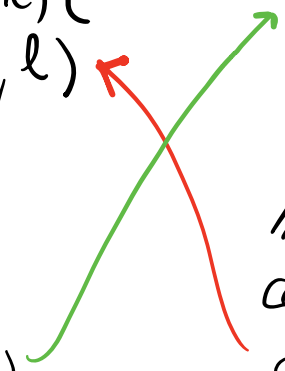


Attempt 4 — Prods & Cons with conditional variables

```
shared count // 0                condition has-space
shared max_size // N             condition has-items
shared lock l
```

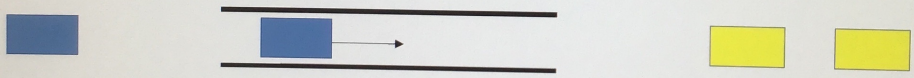
```
Producer {
  // produce
1 acquire(l)
2 while ( count== max_size){
3   cond_wait(has-space, l)
4 }
5 // add
6 count++
7 cond_signal (has_items,l)
8 release (l)
} ∞
```

```
Consumer{
  acquire(l)
  while (count==0){
    cond_wait (has_items, l)
  }
  // get
  count--;
  cond_signal (has_space, l)
  release (l)
  // consumes
} ∞
```



Narrow Bridge problem

A two way east-west road contains a narrow bridge with only one lane. An eastbound (or westbound) car can pass over the bridge only if there is no oncoming car on the bridge. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. In this system, each car is represented by one thread, which executes the procedure OneVehicle when it arrives at the bridge.



```
OneVehicle(Direction direc) {
    ArriveBridge(direc);
    CrossBridge(direc);
    ExitBridge(direc); }
```

direc gives the direction in which the vehicle will cross the bridge

Write the procedures *ArriveBridge* and *ExitBridge*.

ArriveBridge must not return until it safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions or bridge collapses).

ExitBridge is called to indicate that the caller has finished crossing the bridge; *ExitBridge* should take steps to let additional cars cross the bridge.

Constraints

- $0 \leq$ cars ≤ 3
- all cars same direction
- bridge empty $\wedge$ car wants in? $\vee$
  (cars == 0)
- bridge $\neg$(empty $\vee$ full) $\wedge$ car in the same direction? $\vee$
- synchronization shared variables → cars, direction

```
shared var cars // 0
           D    // 0  (direction)
           lock bridge
           waitingToGo[2]

ArriveBridge(d) {
    lock(bridge)
    while(cars==3 OR (cars>0 AND D!=d)) {
        waitingToGo[d]++;
        cond_wait(waitingToGo[d], bridge)
        waitingToGo[d]--;
    }
    cars++;
    D=d;
    release(bride);
}
// traverse bridge    crossBridge();
ExitBridge(d){
    lock(bridge)
    cars--;
    if(waitingToGo[d]>0){
        cond_signal(waitingToGo[d], bridge);
    } else if (cars==0){
        cond_broadcast(waitingToGo[1-d], bridge);
    }
    release(bridge);
}
```