Semaphores

Edited slides from http://cs162.eecs.Berkeley.edu

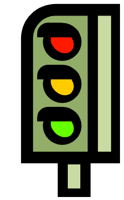
Higher-level Primitives than Locks

- Goal of last couple of lectures:
  - What is right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible

- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs

- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - This lecture and the next presents a some ways of structuring sharing

Semaphores

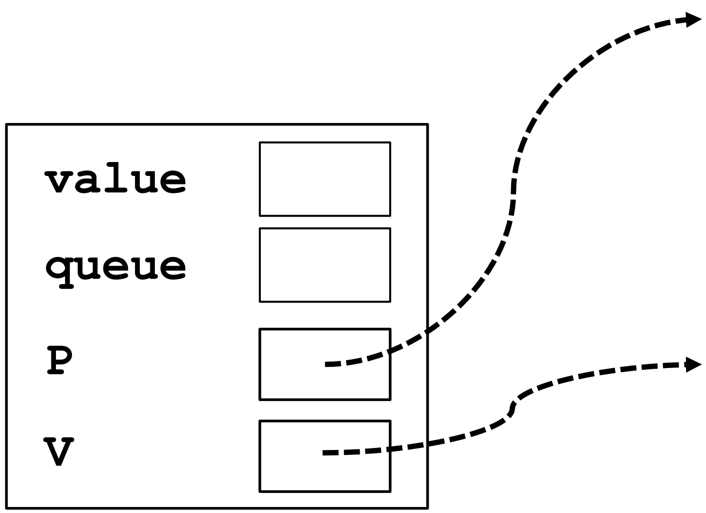


- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » Think of this as the signal() operation
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - » Two P's together can't decrement value below zero
    - » Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time



P() {

an atomic operation that waits for semaphore to become positive, then decrements it by 1

}

V() {

an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

}

Two Uses of Semaphores

Mutual Exclusion (initial value = 1)

- Also called "Binary Semaphore".
- Can be used for mutual exclusion:

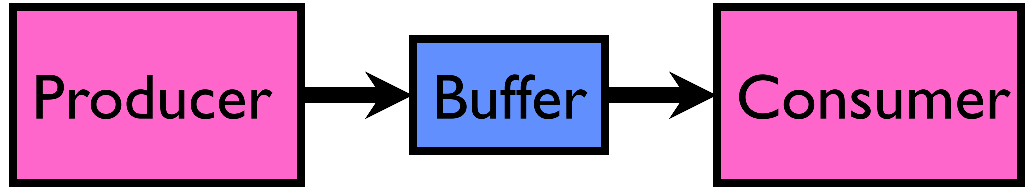
```
semaphore.P();
// Critical section goes here
semaphore.V();
```

Scheduling Constraints (initial value = 0)

- Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 schedules thread 1 when a given event occurs
- Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaphore.P();
}
ThreadFinish {
   semaphore.V();
}
```

Producer-Consumer with a Bounded Buffer



- Problem Definition
  – Producer puts things into a shared buffer
  – Consumer takes them out
  – Need synchronization to coordinate producer/consumer

- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  – Need to synchronize access to this buffer
  – Producer needs to wait if buffer is full
  – Consumer needs to wait if buffer is empty

- Example 1: GCC compiler
  - `cpp | cc1 | cc2 | as | ld`
- Example 2: Coke machine
  – Producer can put limited number of Cokes in machine
  – Consumer can't take Cokes out if machine is empty

Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullSlots; // consumer's constraint`
  - `Semaphore emptySlots;// producer's constraint`
  - `Semaphore mutex;       // mutual exclusion`

Full Solution to Bounded Buffer

```
Semaphore fullSlots = 0;   // Initially, no coke
Semaphore emptySlots = bufSize;
                           // Initially, num empty slots
Semaphore mutex = 1;       // No one using machine

Producer(item) {
    emptySlots.P();        // Wait until space
    mutex.P();             // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V();         // Tell consumers there is
                           // more coke
}

Consumer() {
    fullSlots.P();         // Check if there's a coke
    mutex.P();             // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V();        // tell producer need more
    return item;
}
```


```
Enqueue(item)
item = Dequeue()
```

fullSlots

value	0
queue	
P	
V	

emptySlots

value	n
queue	
P	
V	

mutex

value	1
queue	
P	
V	

자판기의 상태를 밖에서 간접적으로 판단하고 필요한 경우 밖에서 기다린다.

Discussion about Solution

Why asymmetry?

Decrease # of empty slots

Increase # of occupied slots

- Producer does: `emptySlots.P(), fullSlots.V()`
- Consumer does: `fullSlots.P(), emptySlots.V()`

Decrease # of occupied slots

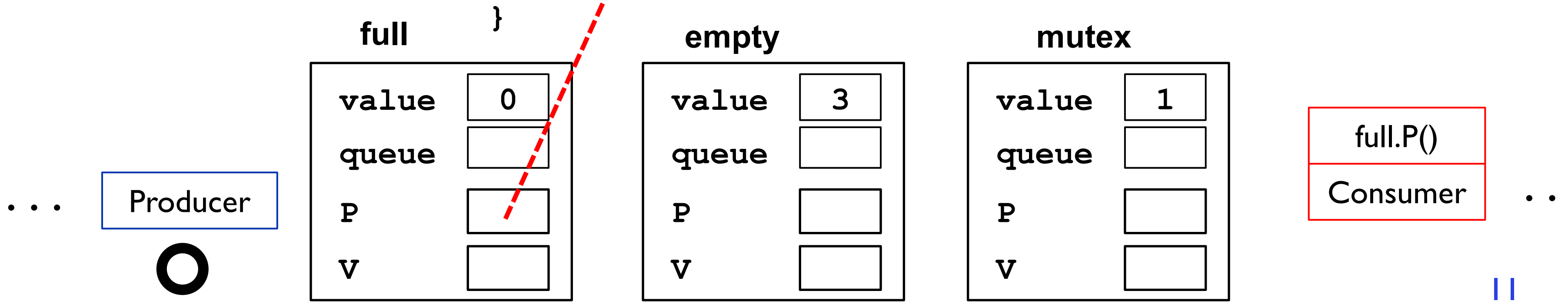
Increase # of empty slots

```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}

Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}

main(){
   for(i=0;i<10;i++)
         ThreadFork(Producer,i);
   for(i=0; i<10; i++)
         ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```



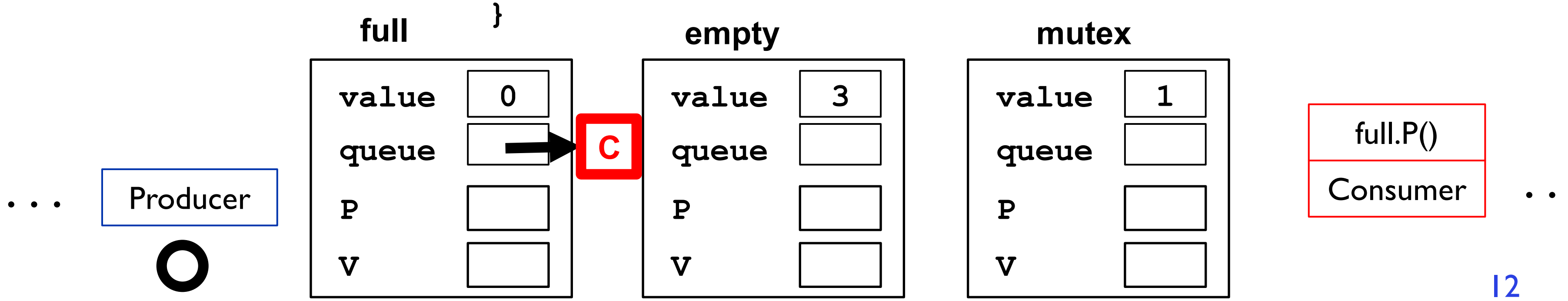
```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

wait

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```



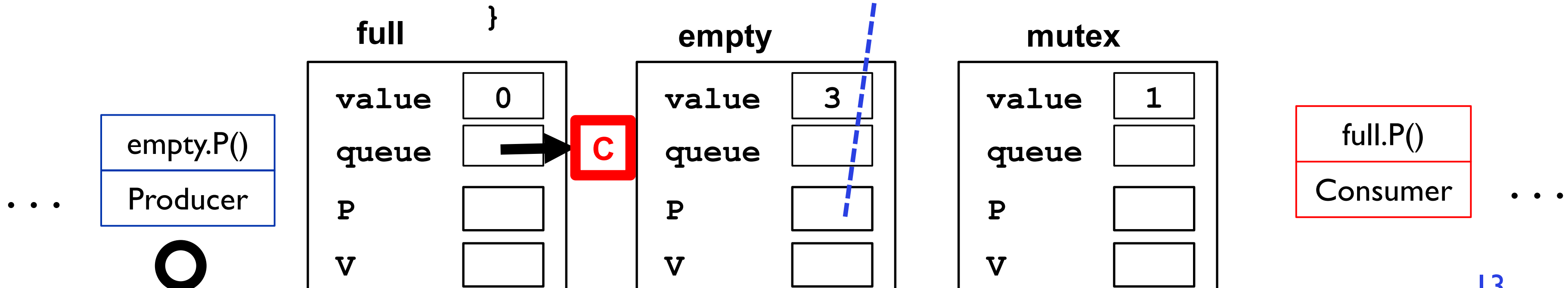
```
Producer(item) {
    empty.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    full.V();
}
```

```
Consumer() {
    full.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    empty.V();
    return item;
}
```

```
main(){
    for(i=0;i<10;i++)
        ThreadFork(Producer,i);
    for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

wait



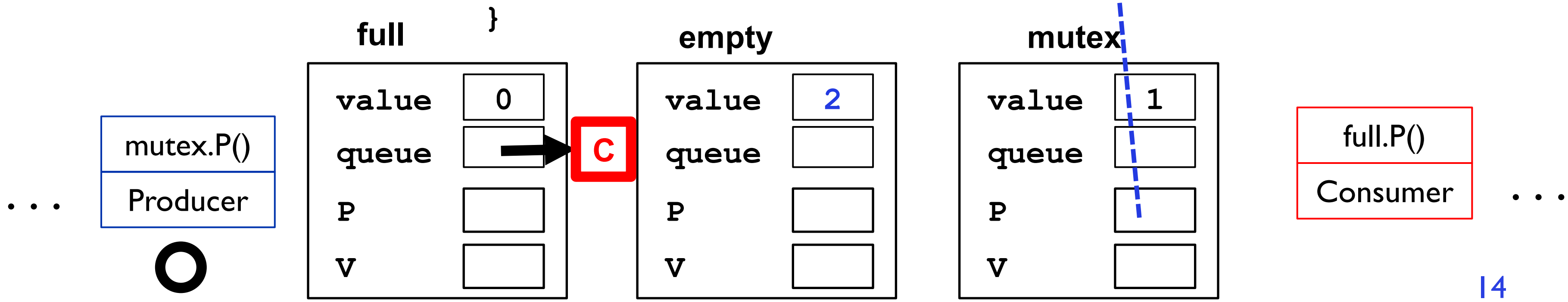
```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

wait



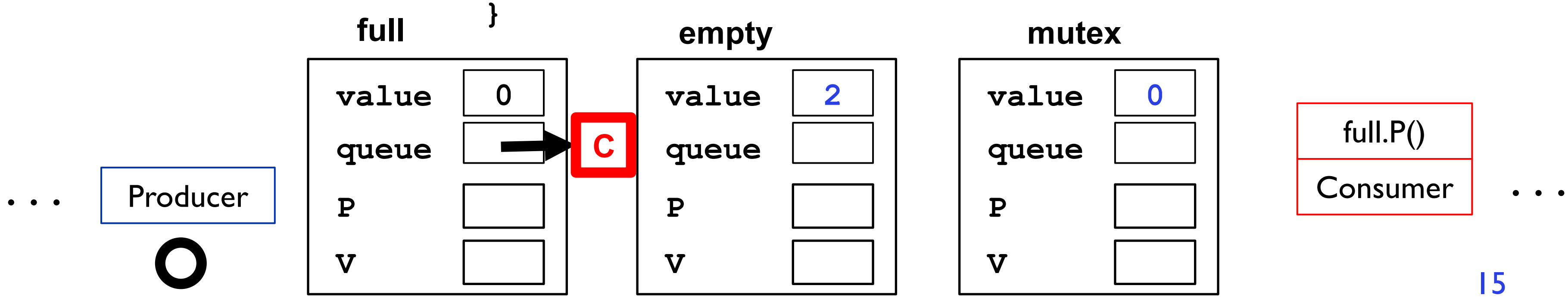
```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

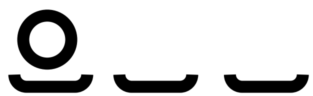
wait

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```



```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```



```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

wait

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

. . . Producer

full

value	0
queue	C
P	
V	

empty

value	2
queue	
P	
V	

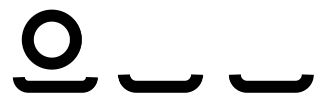
mutex

value	0
queue	
P	
V	

full.P()
Consumer

. . .

```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

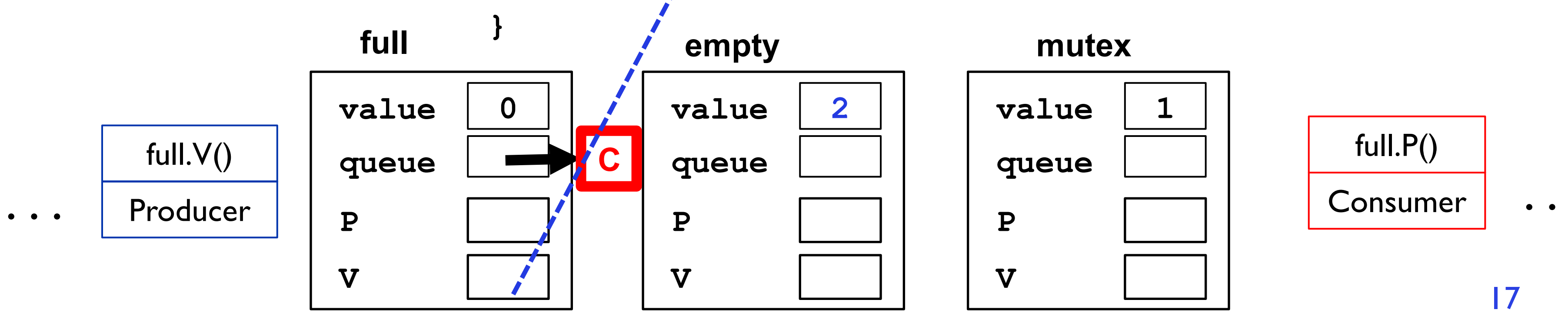


```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

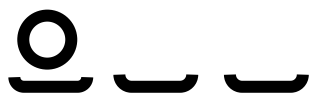
```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

wait



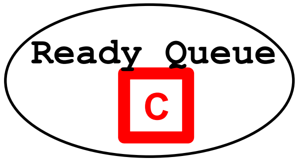
```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```



```
Consumer() {
   full.P();
   mutex.P();
   item = Deque();
   mutex.V();
   empty.V();
   return item;
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

ready



Wake up

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

full

value	1
queue	
P	
V	

empty

value	2
queue	
P	
V	

mutex

value	1
queue	
P	
V	

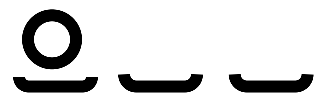
. . . Producer

full.P()
Consumer

. . .

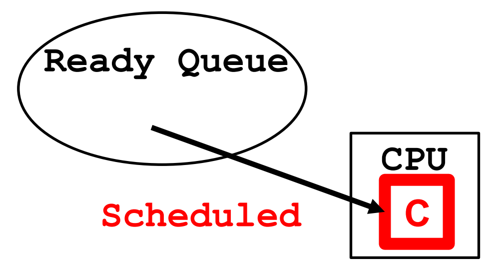
```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

ready →



```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```




```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

full

value	1
queue	
P	
V	

empty

value	2
queue	
P	
V	

mutex

value	1
queue	
P	
V	

. . . Producer | full.P() Consumer . . .

```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

ready

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

	full	empty	mutex
value	0	2	0
queue			
P			
V			

. . . Producer

Consumer . . .

```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

ready

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

ready

```
main(){
   for(i=0;i<10;i++)
         ThreadFork(Producer,i);
   for(i=0; i<10; i++)
         ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

empty.P()
Producer

. . . Producer

full	
value	0
queue	
P	
V	

empty	
value	1
queue	
P	
V	

mutex	
value	0
queue	
P	
V	

Consumer . . .

```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

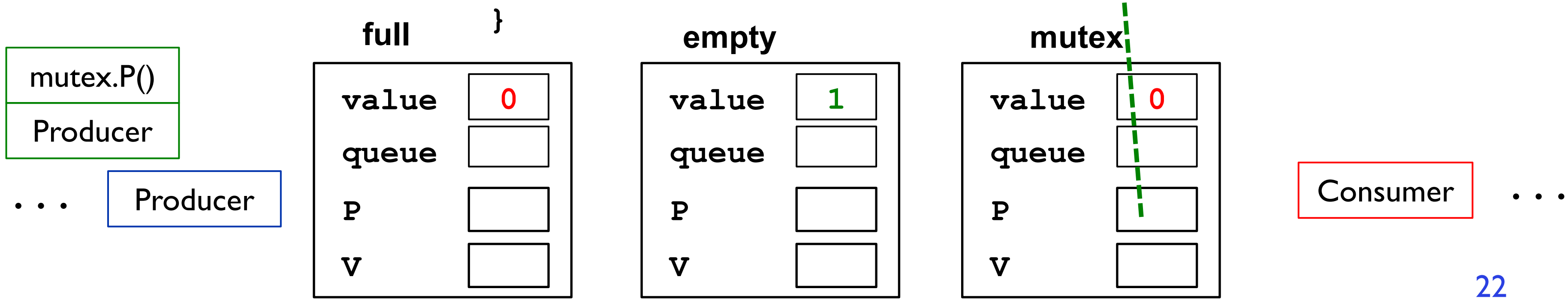
ready

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

ready

```
main(){
   for(i=0;i<10;i++)
         ThreadFork(Producer,i);
   for(i=0; i<10; i++)
         ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```



```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

wait

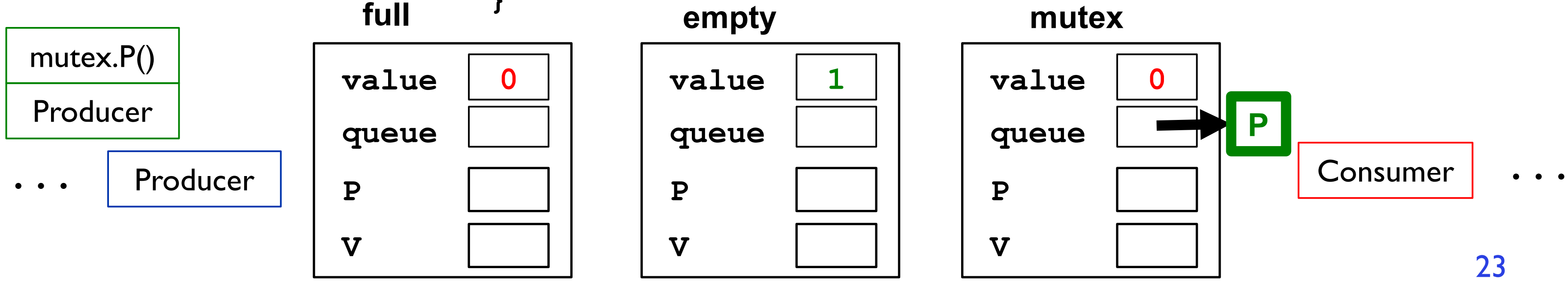
ready

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

ready

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```



wait

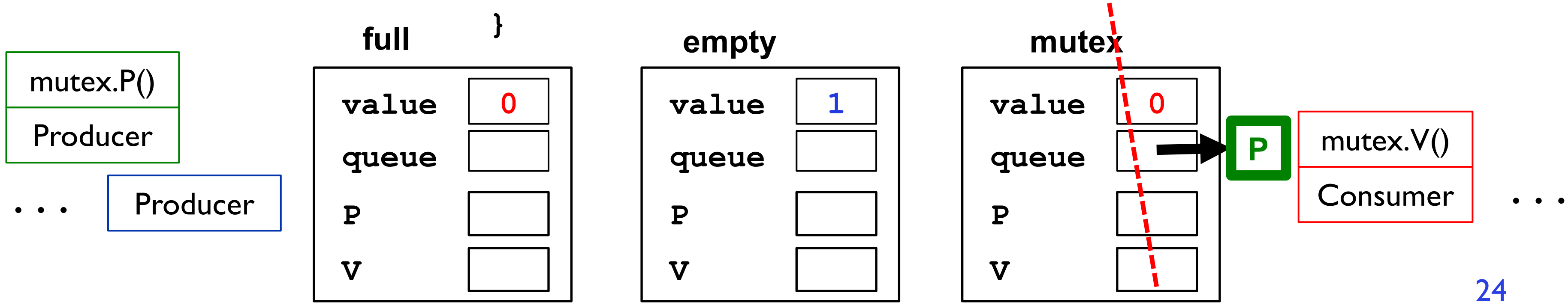
ready

```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
main(){
   for(i=0;i<10;i++)
         ThreadFork(Producer,i);
   for(i=0; i<10; i++)
         ThreadFork(Consumer,null);
}
```



ready

ready

```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

Ready Queue

P

Wake up

full	
value	0
queue	
P	
V	

empty	
value	1
queue	
P	
V	

mutex	
value	1
queue	
P	
V	

mutex.P()

Producer

. . .

Producer

empty.V()

Consumer

. . .

ready

ready

```
Producer(item) {
   empty.P();
   mutex.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

mutex.P()

Producer

. . . Producer

full

value	0
queue	
P	
V	

empty

value	2
queue	
P	
V	

mutex

value	1
queue	
P	
V	

Consumer . . .

Discussion about Solution (cont'd)

Is order of P's important?

- Yes! Can cause deadlock

Is order of V's important?

- No, except that it might affect scheduling efficiency

What if we have 2 producers or 2 consumers?

```
Producer(item) {
    mutex.P();
    emptySlots.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}
Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```

```
Producer(item) {
   mutex.P();
   empty.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

. . .

mutex.P()
Producer

full

value	3
queue	
P	
V	

empty

value	0
queue	
P	
V	

mutex

value	1
queue	
P	
V	

Consumer

. . .

```
Producer(item) {
   mutex.P();
   empty.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

full

value	3
queue	
P	
V	

empty

value	0
queue	
P	
V	

mutex

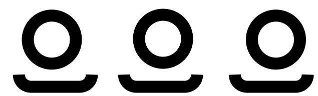
value	0
queue	
P	
V	

. . . empty.P() Producer

Consumer . . .

wait

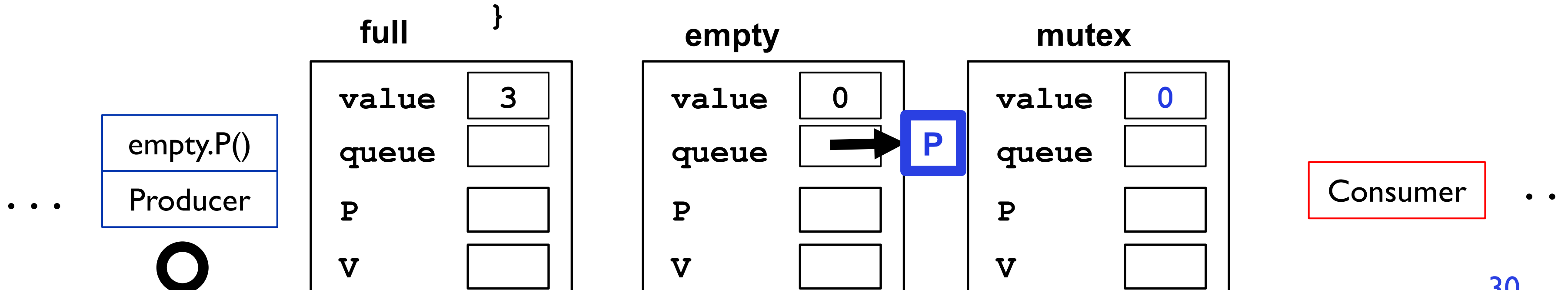
```
Producer(item) {
   mutex.P();
   empty.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```



```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

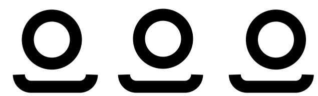
```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```



wait

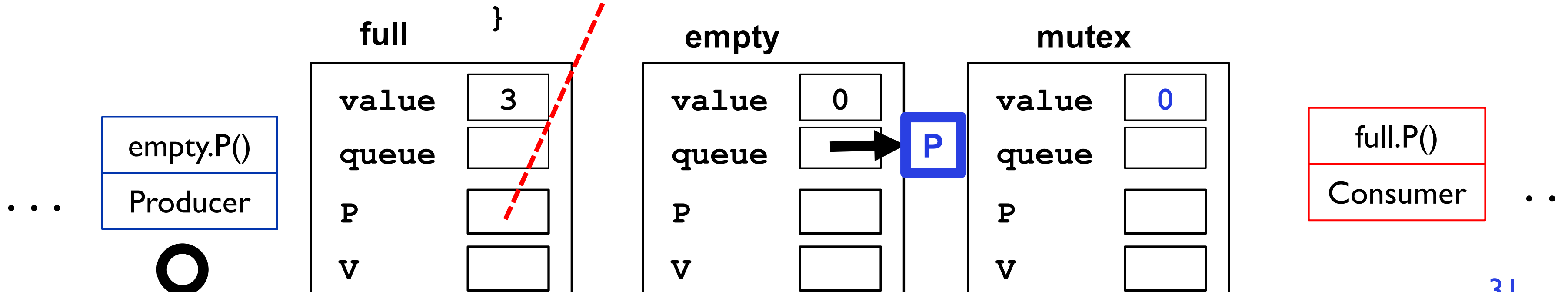
```
Producer(item) {
   mutex.P();
   empty.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```



```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

```
main(){
   for(i=0;i<10;i++)
         ThreadFork(Producer,i);
   for(i=0; i<10; i++)
         ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```



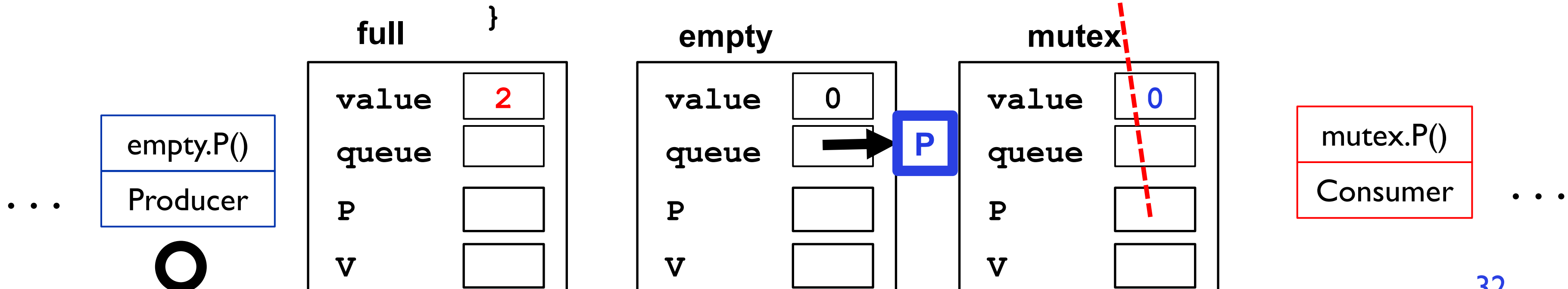
wait

```
Producer(item) {
   mutex.P();
   empty.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

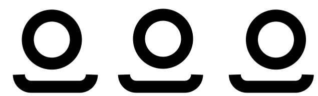
```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```



wait

```
Producer(item) {
   mutex.P();
   empty.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

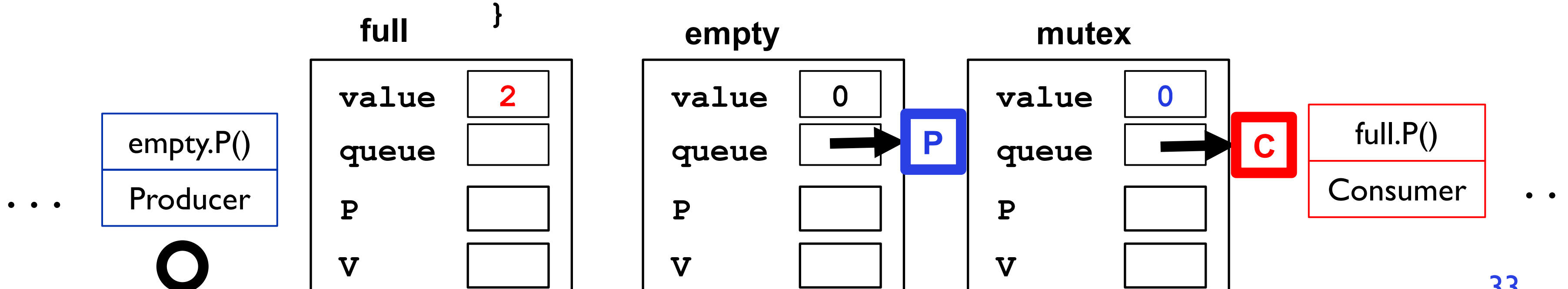


```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

wait

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```



wait

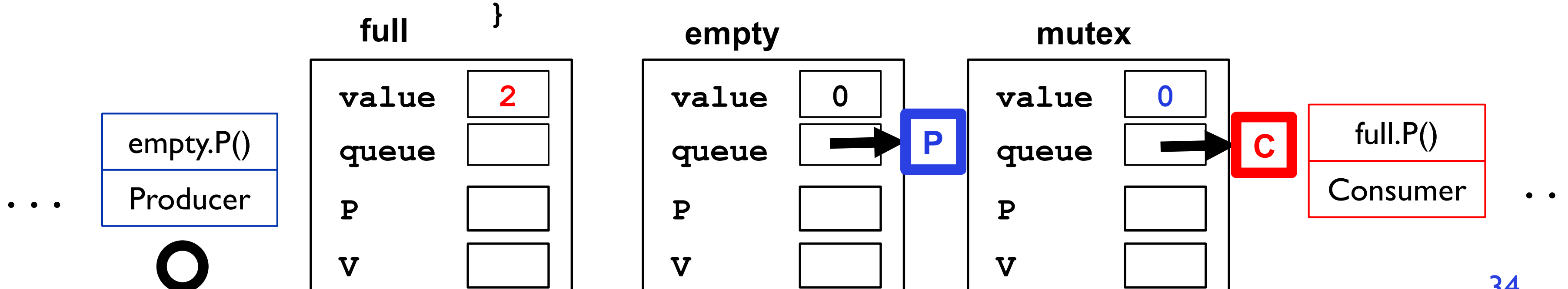
```
Producer(item) {
   mutex.P();
   empty.P();
   Enqueue(item);
   mutex.V();
   full.V();
}
```

```
Consumer() {
   full.P();
   mutex.P();
   item = Dequeue();
   mutex.V();
   empty.V();
   return item;
}
```

wait

```
P() {
an atomic operation that waits
for semaphore to become
positive, then decrements it by
1
}
V() {
an atomic operation that
increments the semaphore by 1,
waking up a waiting P, if any
}
```

```
main(){
   for(i=0;i<10;i++)
        ThreadFork(Producer,i);
   for(i=0; i<10; i++)
        ThreadFork(Consumer,null);
}
```



Summary

- Semaphores: Like integers with restricted interface
  - Two operations:
    - » `P()`: Wait if zero; decrement when becomes non-zero
    - » `V()`: Increment and wake a sleeping task (if exists)
    - » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint