Operating Systems

David Hay

First Data Type: Semaphore

- Record with two fields:
  - value
  - List (L)

First Data Type: Semaphore

- Record with two fields:
  - value
  - List (L)
- Two operations:

Down(S)	Up(S)	Init(S,v)
S.*value*= S.*value* - 1 if S.*value* < 0 then { add this thread to S.*L*; sleep();}	S.*value*= S.*value* + 1 if S.*value*≤0 then {remove a thread T from S.L; Wakeup(T);}	S.*value*= v

First Data Type: Semaphore

- Record with two fields:
  - value
  - List (L)
- Two operations:

Down(S)

S.*value*= S.*value* - 1
if S.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

Up(S)

S.*value*= S.*value* + 1
if S.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

Init(S,v)

S.*value*= v

- The operations are executed atomically
  – How? Implementation is orthogonal to the definition

First Data Type: Semaphore

- Record with two fields:

In literature, the operations are often called P(s) and V(s).
In the book, wait(s), signal(s)

Down(S)	**Up(S)**	**Init(S,v)**
S.*value*= S.*value* - 1 if S.*value* < 0 then { add this thread to S.*L*; sleep();}	S.*value*= S.*value* + 1 if S.*value*≤0 then {remove a thread T from S.L; Wakeup(T);}	S.*value*= v

- The operations are executed atomically
  – How? Implementation is orthogonal to the definition

First Data Type: Semaphore

- Record with two fields:
  - value
  - List (L)
- Two operations:

Some variations bound the maximum and minimum value (e.g., binary semaphores)

Down(S)	**p(S)**	**Init(S,v)**
S.*value*= S.*value* - 1 if S.*value* < 0 then { add this thread to S.*L*; sleep();}	S.*value*= S.*value* + 1 if S.*value*≤0 then {remove a thread T from S.L; Wakeup(T);}	S.*value*= v

- The operations are executed atomically
  – How? Implementation is orthogonal to the definition

Mutual Exclusion w/ Semaphore

Remainder

down(lock)

Critical

up(lock)

Code for Thread i

lock.*value*= lock.*value* - 1
if lock.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

lock.*value*= lock.*value* + 1
If lock.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

Mutual Exclusion w/ Semaphore

- Assume **lock** is a shared semaphore
  - Initially **lock** is 1

Remainder

down(lock)

lock.*value*= lock.*value* - 1
if lock.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

Critical

up(lock)

lock.*value*= lock.*value* + 1
If lock.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

Code for Thread i

Mutual Exclusion w/ Semaphore

Remainder
down(lock)
Critical
up(lock)

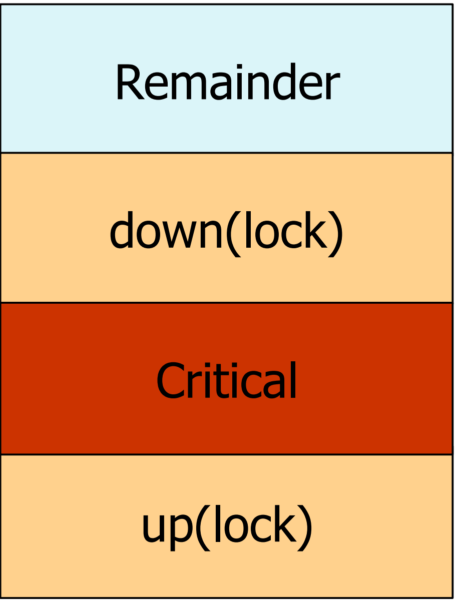
Code for Thread i

lock.*value*= lock.*value* - 1
if lock.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

lock.*value*= lock.*value* + 1
If lock.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

- Assume **lock** is a shared semaphore
  - Initially **lock** is 1
- Value:
  - Cannot be more than 1
  - 0: one process in Critical Section
  - (-x): x processes are waiting

Mutual Exclusion w/ Semaphore



Code for Thread i

```
lock.value= lock.value - 1
if  lock.value < 0 then
 { add this thread to S.L;
   sleep();}
```

```
lock.value= lock.value + 1
If lock.value≤0 then
{remove a thread T from
    S.L;  Wakeup(T);}
```

- Assume **lock** is a shared semaphore
  - Initially **lock** is 1
- Value:
  - Cannot be more than 1
  - 0: one process in Critical Section
  - (-x): x processes are waiting
- All properties hold
  - If L is a FIFO queue

Mutual Exclusion w/ Semaphore

Remainder

down(lock)

Critical

up(lock)

Code for Thread i

lock.*value*= lock.*value* - 1
if lock.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

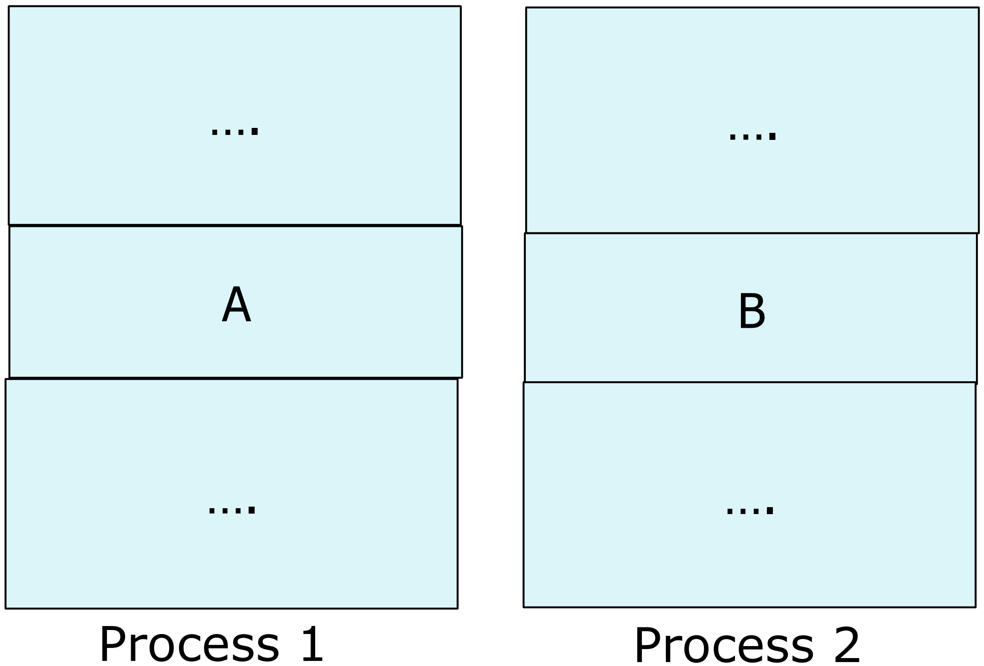
lock.*value*= lock.*value* + 1
If lock.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

- Assume **lock** is a shared semaphore
  - Initially **lock** is 1
- Value:
  - Cannot be more than 1
  - 0: one process in Critical Section
  - (-x): x processes are waiting
- All properties hold
  - If L is a FIFO queue

0/1 (binary) semaphore suffices → Binary semaphores data type are often called Mutex objects

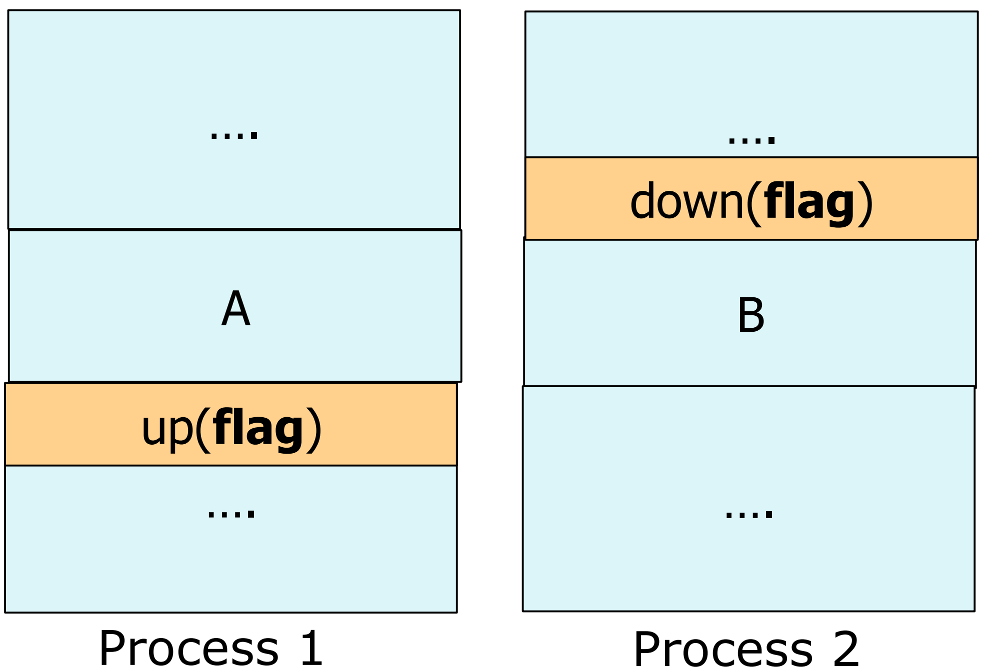
"Execute B after A"

- A different synchronization problem
  - Coordination rather than contention
- One process needs to execute code A, before another process executes code B



"Execute B after A"

- A different synchronization problem
  - Coordination rather than contention
- One process needs to execute code A, before another process executes code B



Semaphore **flag**, initialized to 0

Another Problem: Moving Money Between Accounts

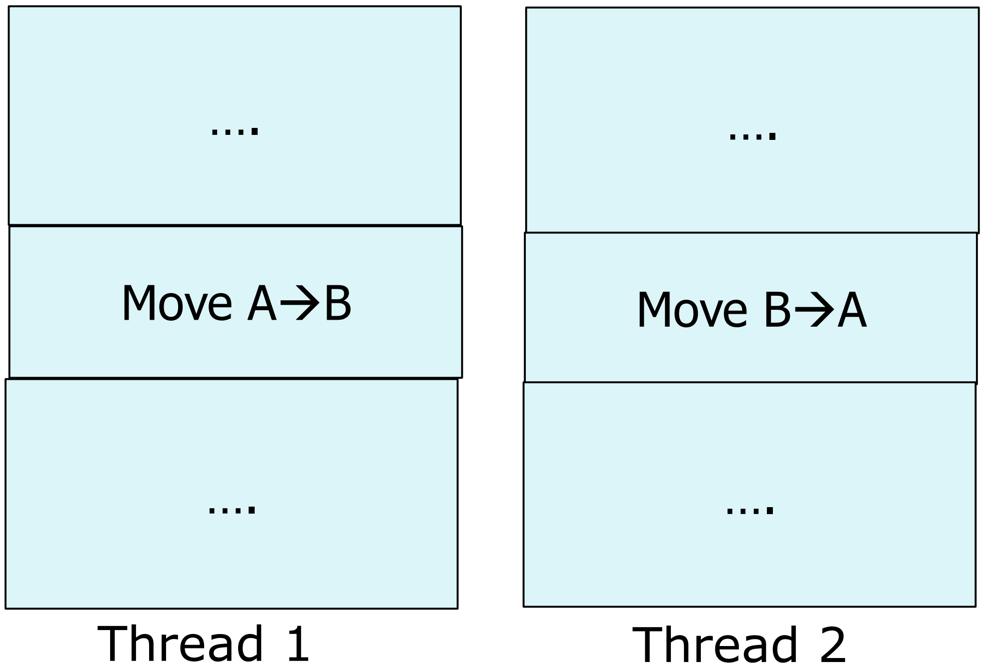
Thread 1 transfers money from account A to B, Thread 2 transfers money from B to A.

If executed simultaneously, some errors might occur (e.g., Thread 1 executes A--, while Thread 2 executes A++)

Another Problem: Moving Money Between Accounts

Thread 1 transfers money from account A to B, Thread 2 transfers money from B to A.

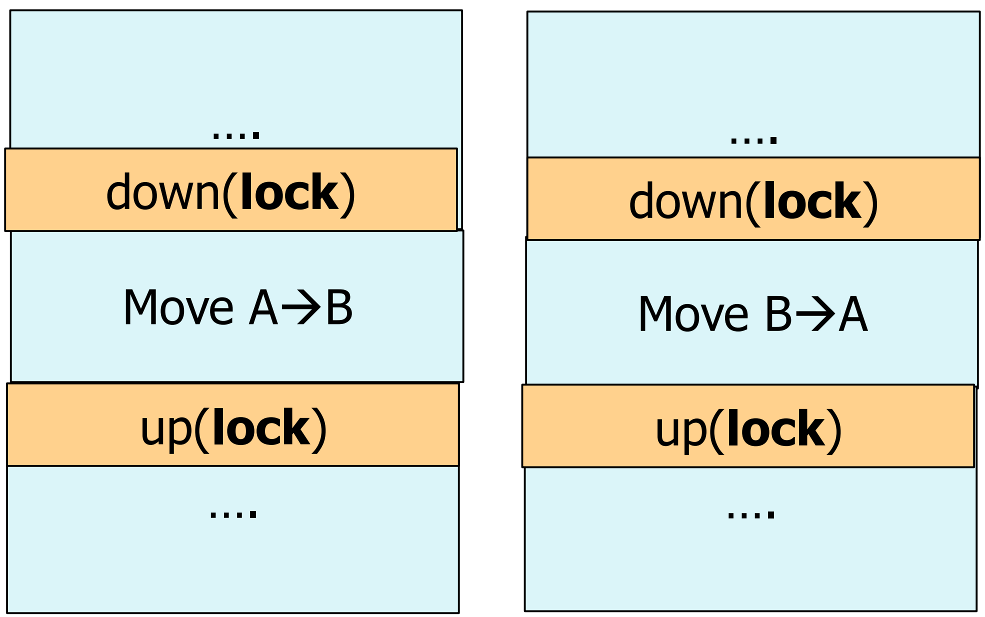
If executed simultaneously, some errors might occur (e.g., Thread 1 executes A--, while Thread 2 executes A++)



Another Problem: Moving Money Between Accounts

Thread 1 transfers money from account A to B, Thread 2 transfers money from B to A.

If executed simultaneously, some errors might occur (e.g., Thread 1 executes A--, while Thread 2 executes A++)



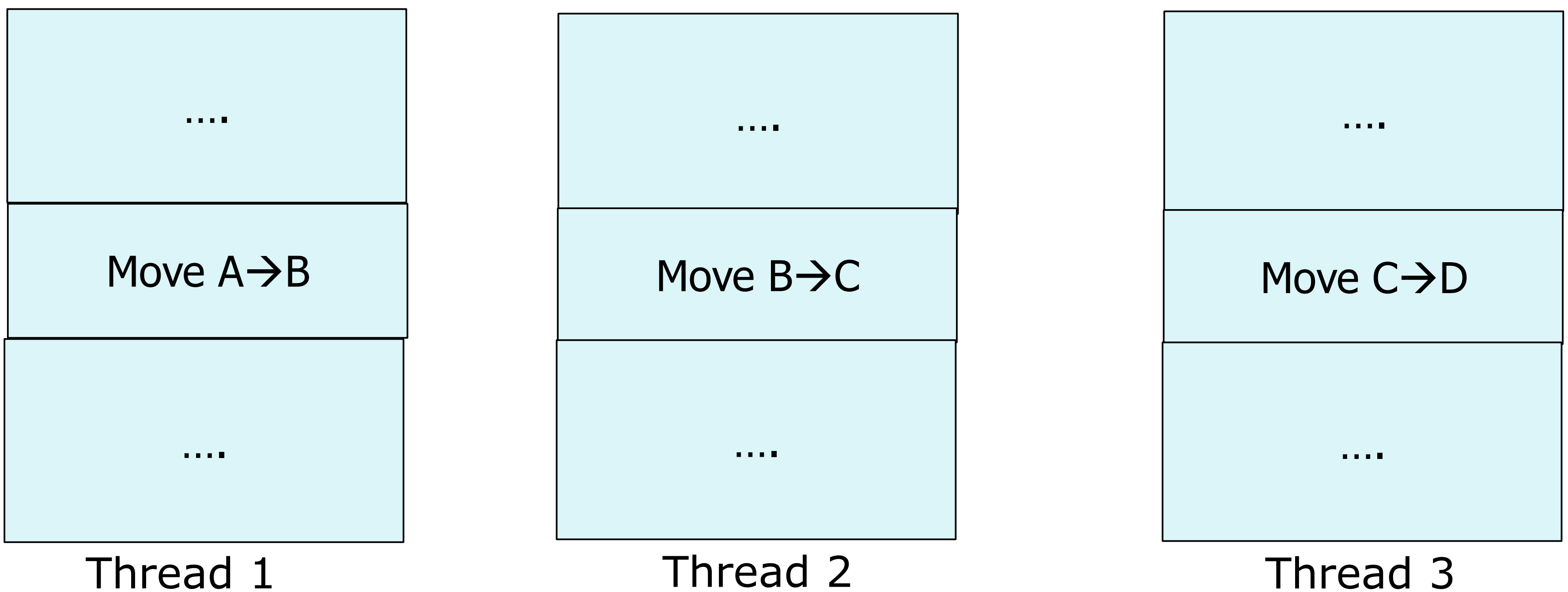
Thread 1 Thread 2

Semaphore **lock**, initialized to 1

Mutual exclusion, where the Move is the critical section

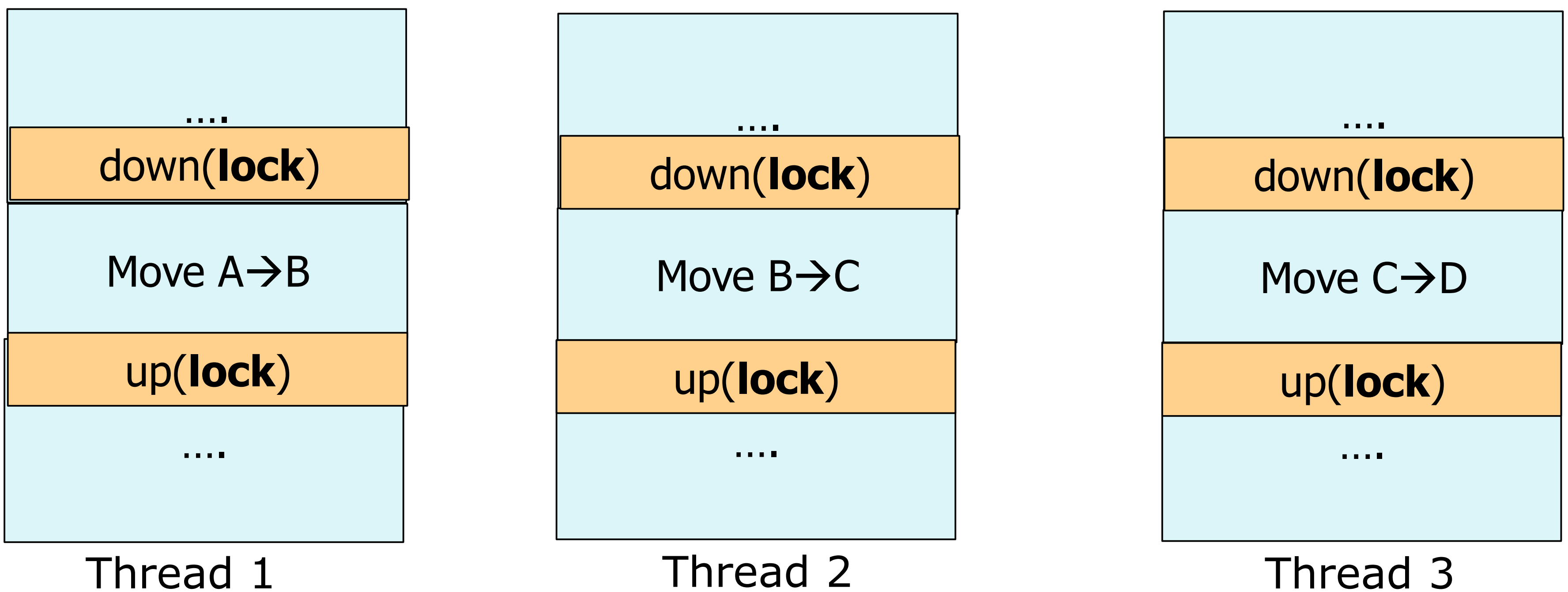
Another Problem: Moving Money Between Accounts

- Three Threads



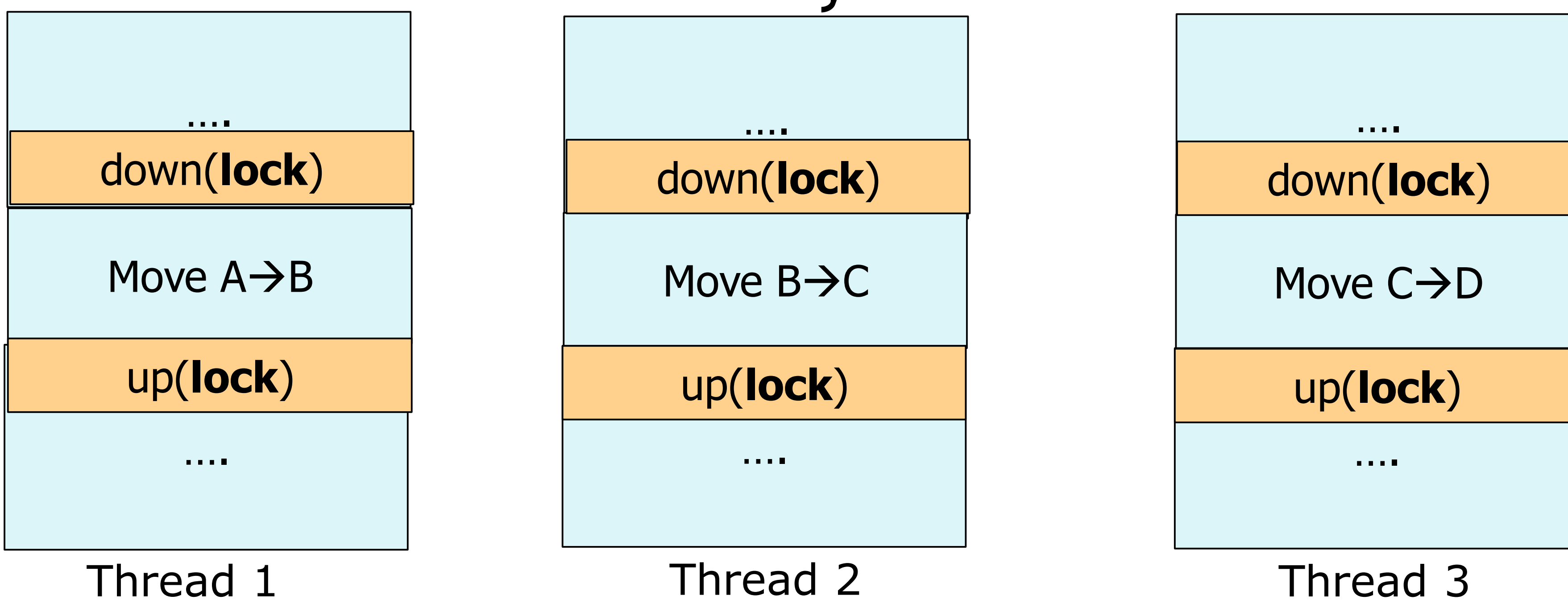
Another Problem: Moving Money Between Accounts

- Three Threads
- Solution 1: mutual exclusion



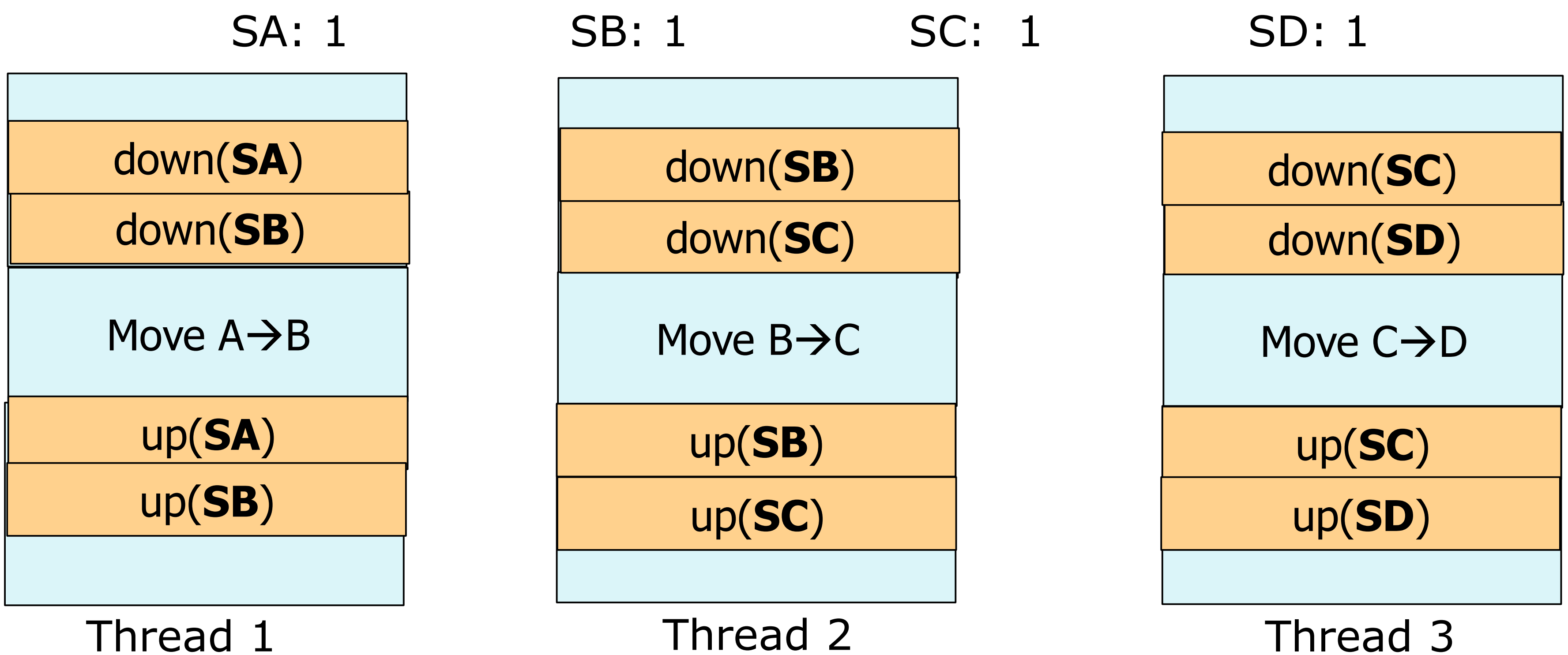
Another Problem: Moving Money Between Accounts

- Three Threads
- Solution 1: mutual exclusion
  – But why Thread 1 and Thread 3 cannot be executed concurrently?



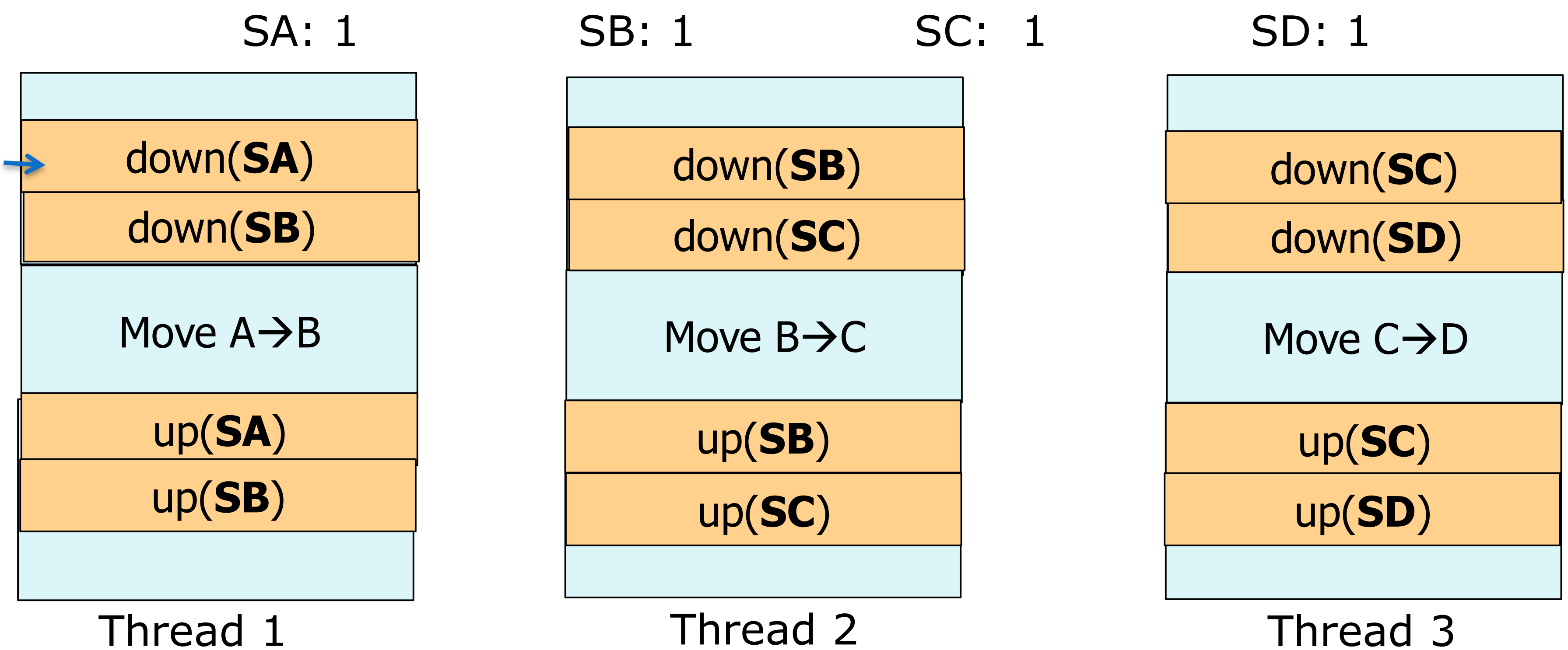
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



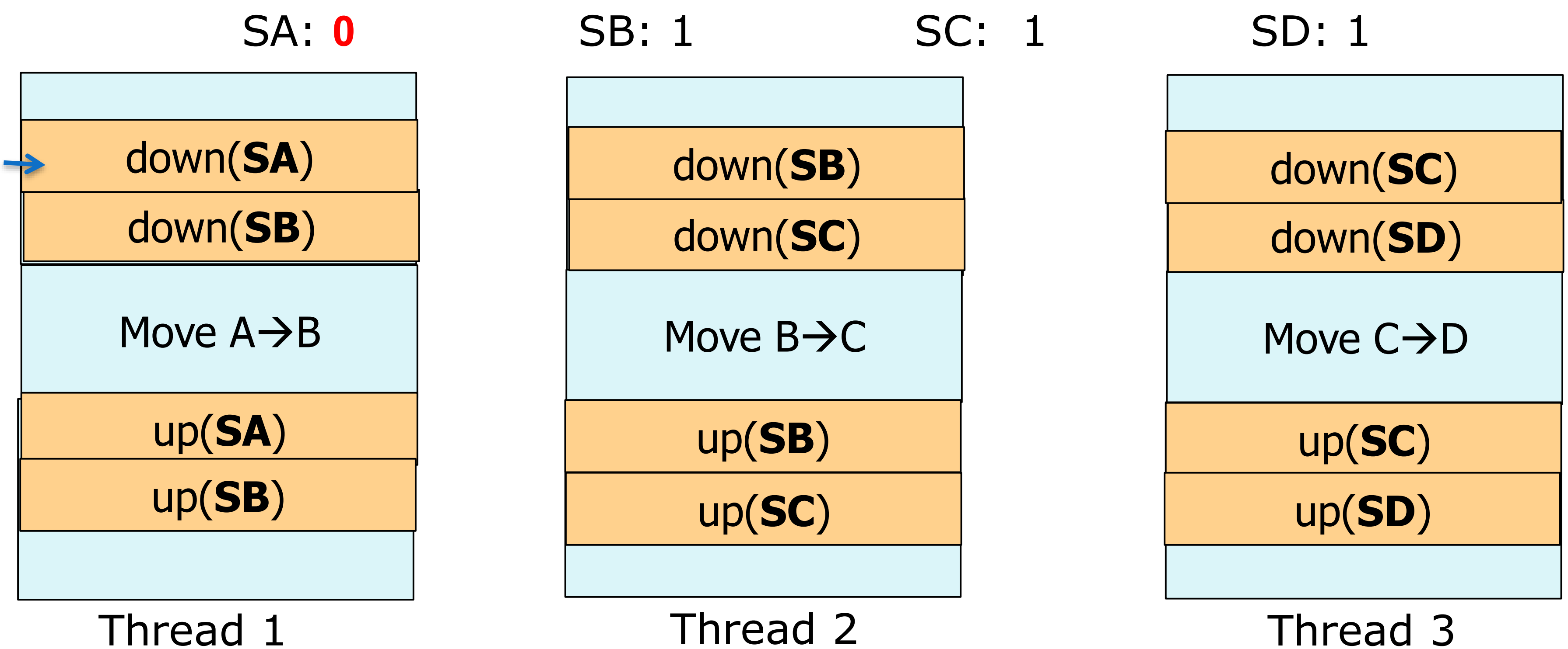
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



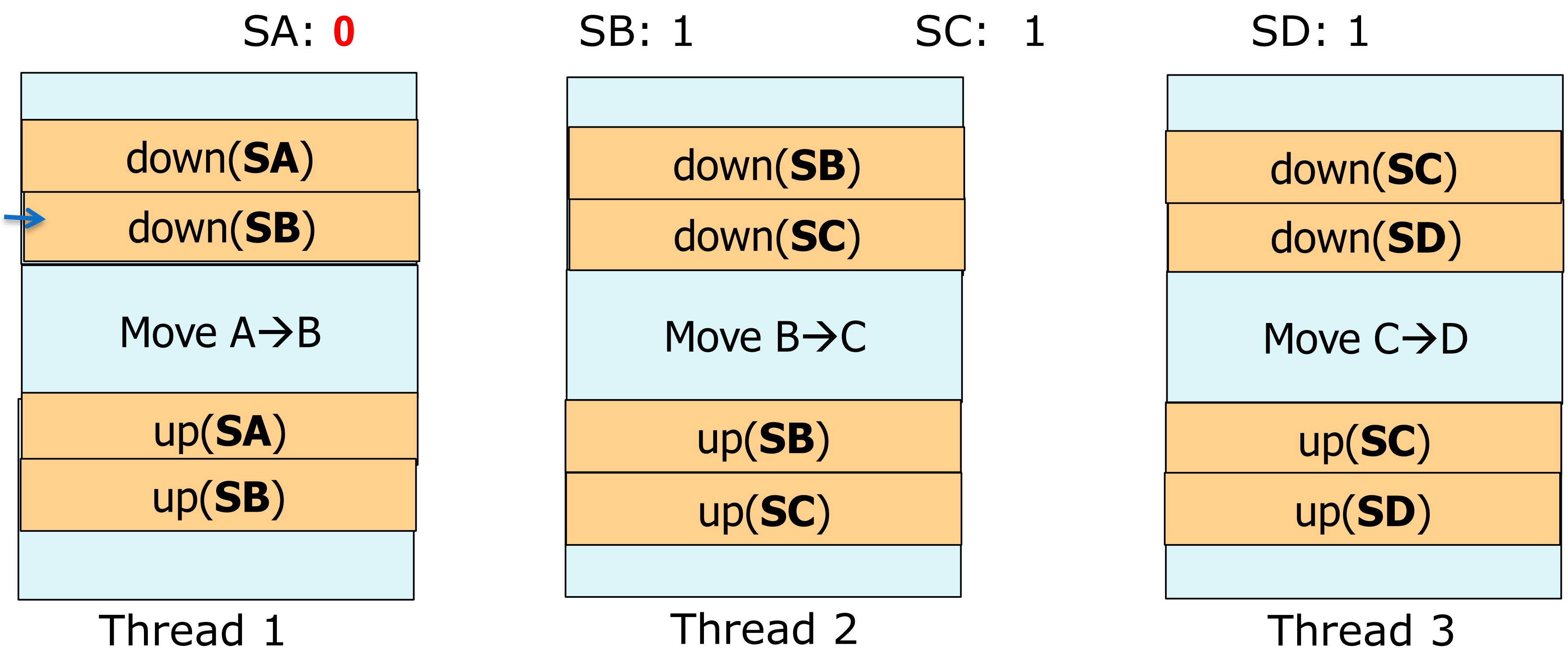
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



Another Problem: Moving Money Between Accounts

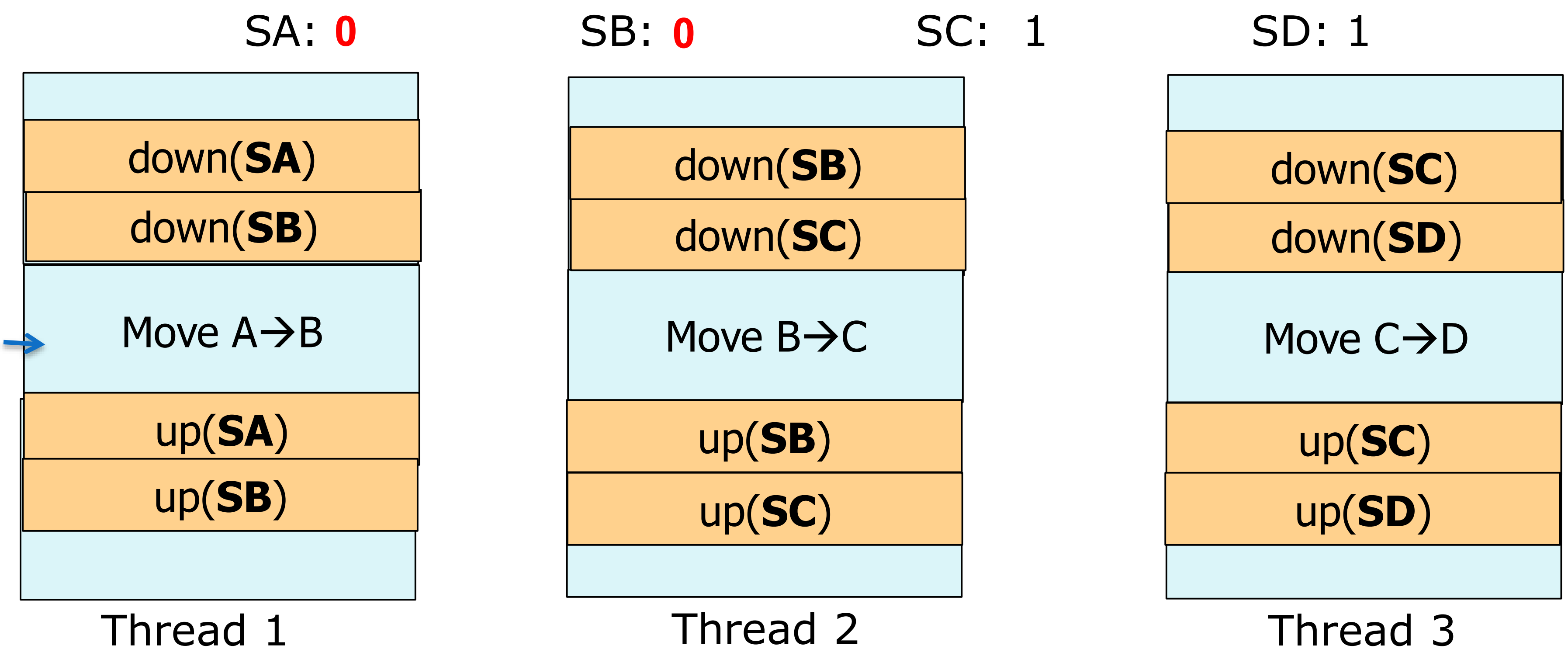
- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.

SA: **0** SB: **0** SC: 1 SD: 1

Thread 1	Thread 2	Thread 3
down(**SA**)	down(**SB**)	down(**SC**)
down(**SB**)	down(**SC**)	down(**SD**)
Move A→B	Move B→C	Move C→D
up(**SA**)	up(**SB**)	up(**SC**)
up(**SB**)	up(**SC**)	up(**SD**)

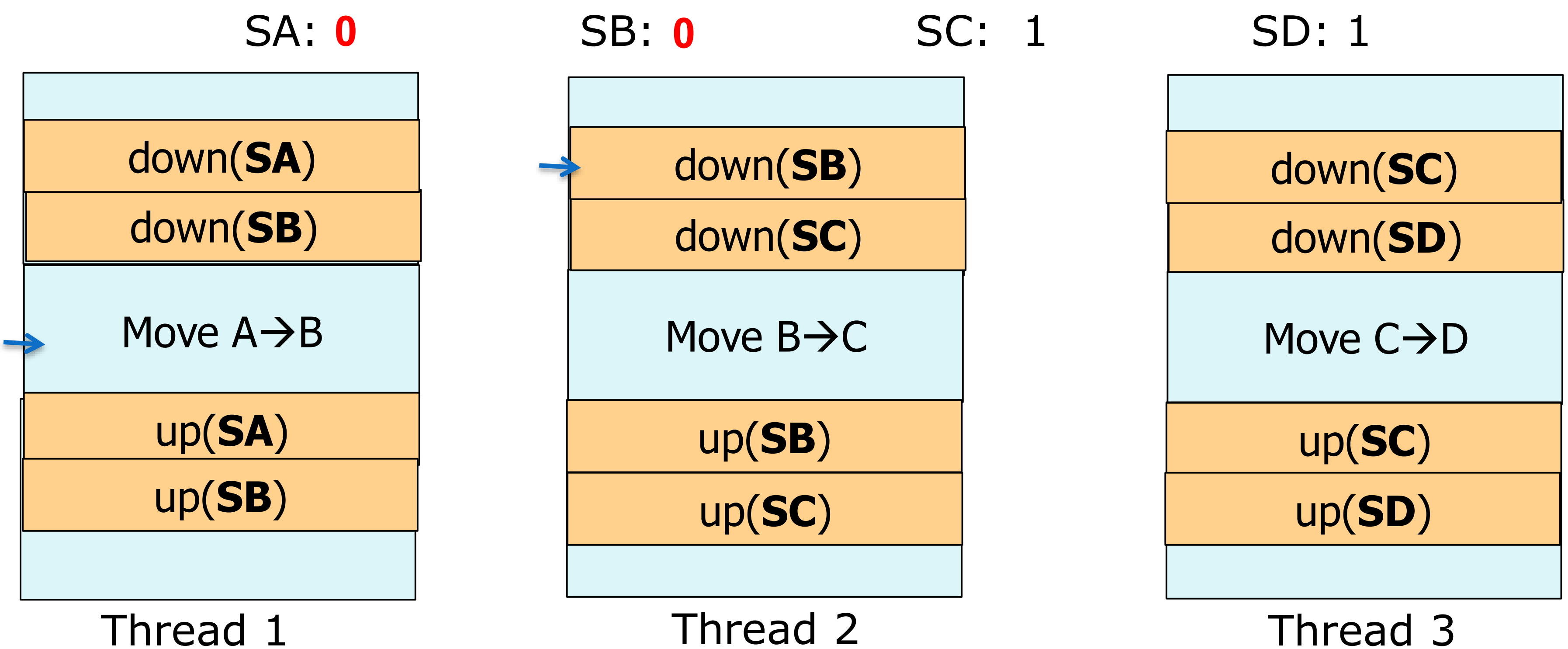
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



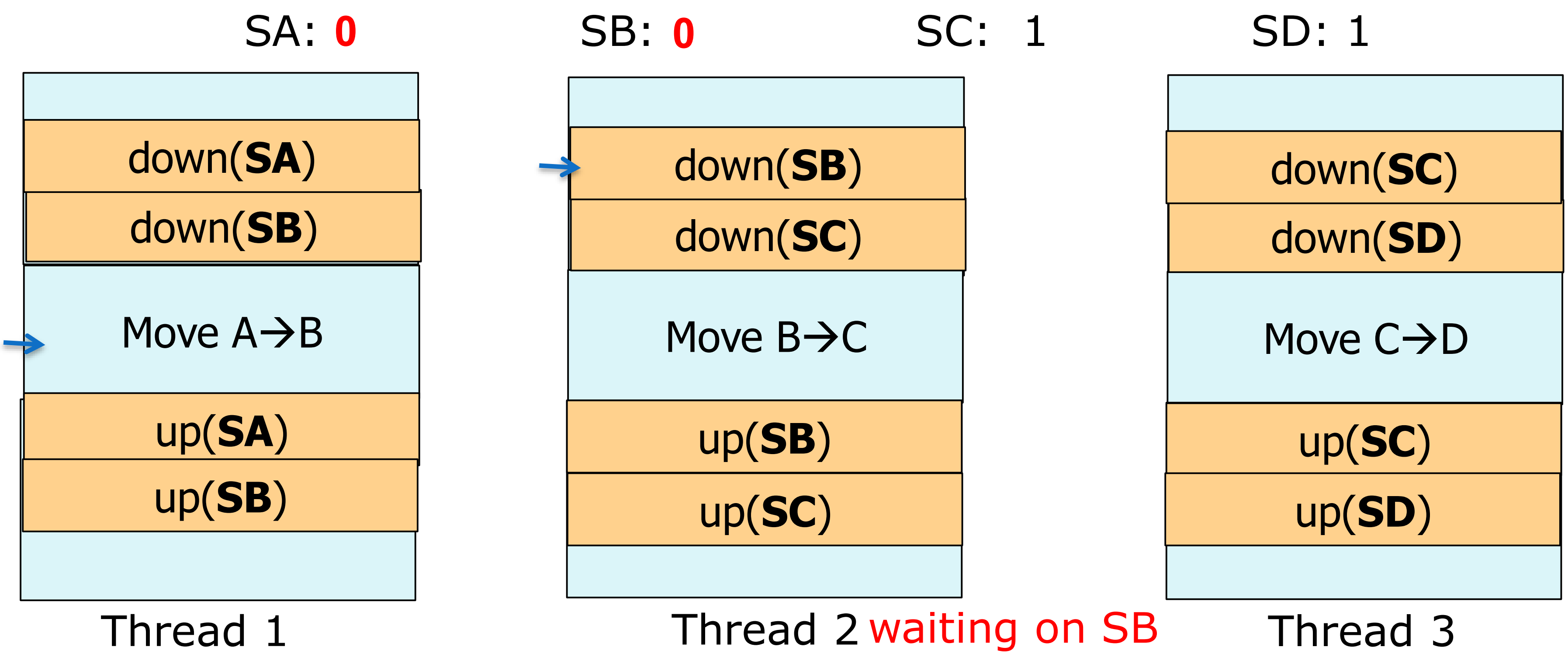
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



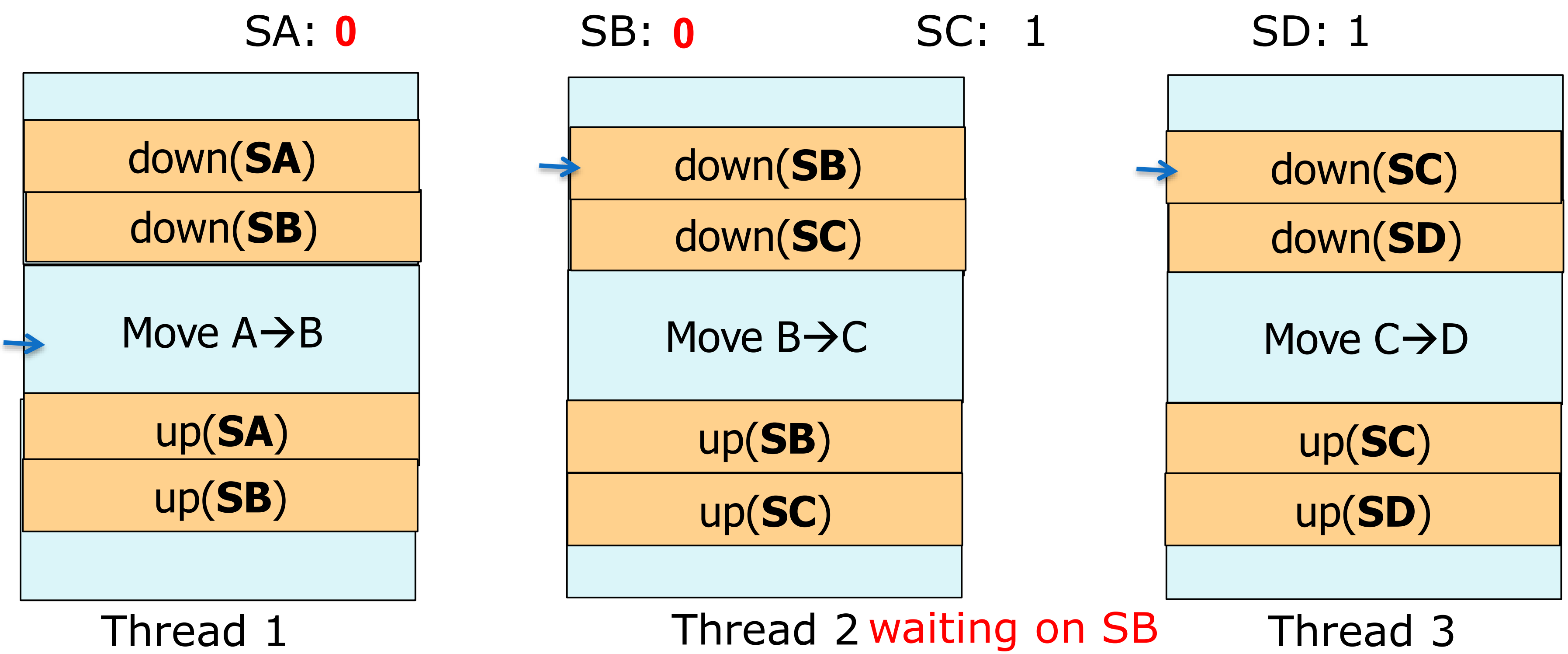
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



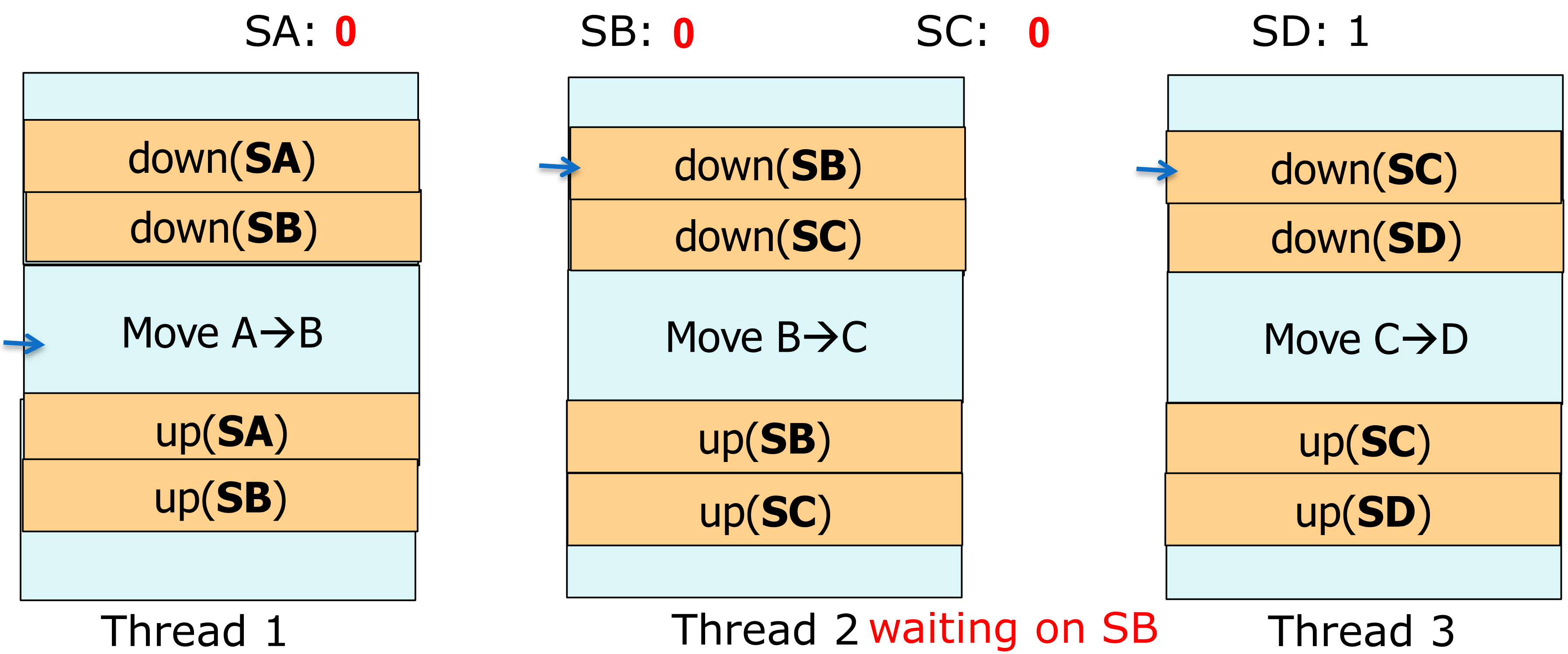
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



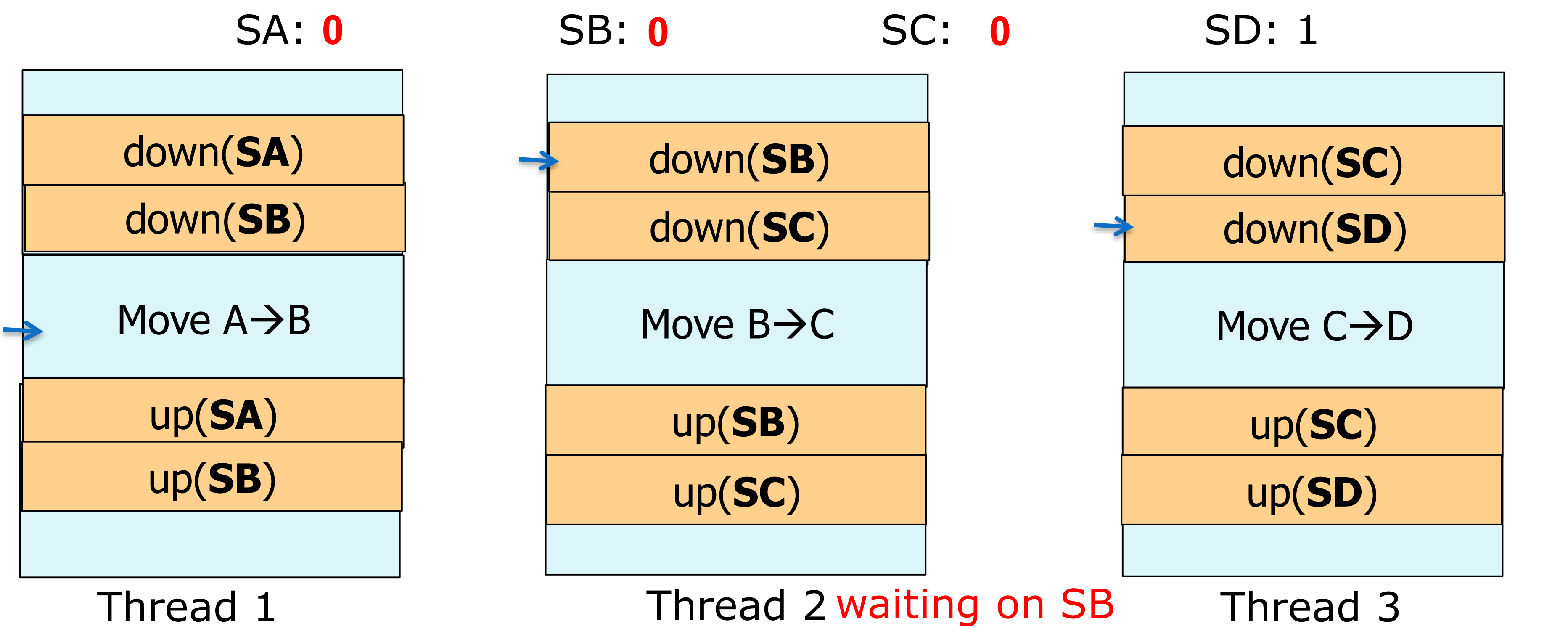
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



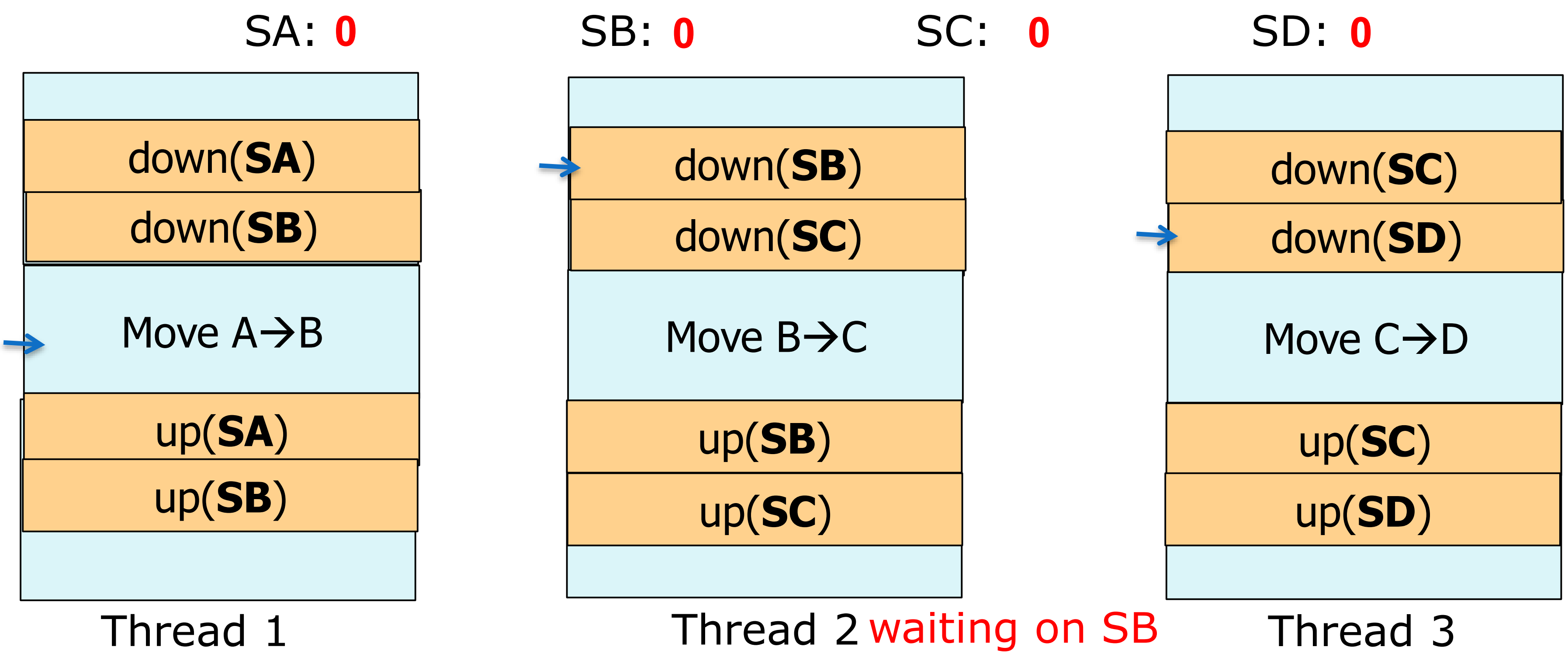
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



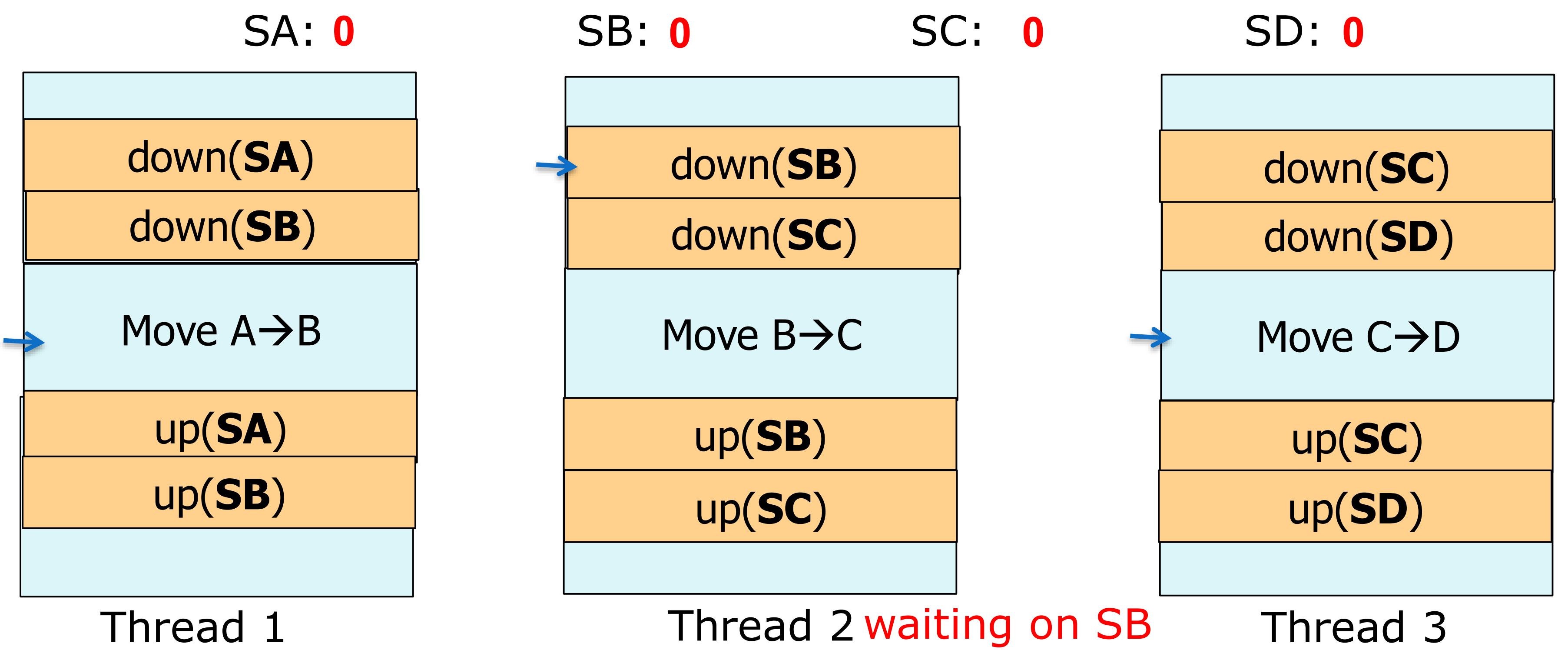
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



Another Problem: Moving Money Between Accounts

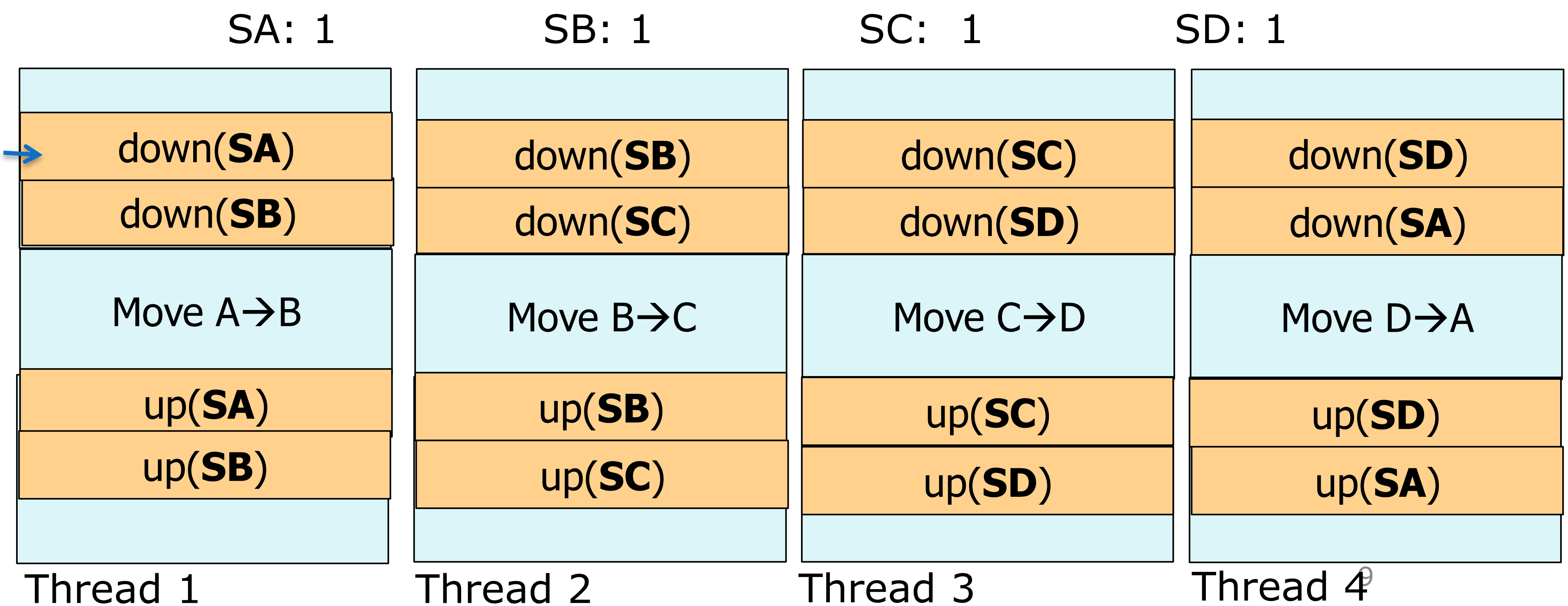
- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



Semaphores are not Silver Bullets

SA: 1	SB: 1	SC: 1	SD: 1
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1	Thread 2	Thread 3	Thread 4

Semaphores are not Silver Bullets



Semaphores are not Silver Bullets

SA: 0	SB: 1	SC: 1	SD: 1
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1	Thread 2	Thread 3	Thread 4

Semaphores are not Silver Bullets

SA: **0**	SB: 1	SC: 1	SD: 1
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1	Thread 2	Thread 3	Thread 4

Semaphores are not Silver Bullets

SA: 0	SB: 0	SC: 1	SD: 1
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1	Thread 2	Thread 3	Thread 4

Semaphores are not Silver Bullets

SA: **0**	SB: **0**	SC: 1	SD: 1
→ down(**SA**)	→ down(**SB**)	→ down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1	Thread 2	Thread 3	Thread 4

Semaphores are not Silver Bullets

SA: 0	SB: 0	SC: 0	SD: 1
→ down(**SA**)	→ down(**SB**)	→ down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1	Thread 2	Thread 3	Thread 4

Semaphores are not Silver Bullets

SA: **0**	SB: **0**	SC: **0**	SD: 1
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1	Thread 2	Thread 3	Thread 4

Semaphores are not Silver Bullets

SA: 0	SB: 0	SC: 0	SD: 0
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1	Thread 2	Thread 3	Thread 4

Semaphores are not Silver Bullets

SA: **0**	SB: **0**	SC: **0**	SD: **0**
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1 waits SB	Thread 2 waits SC	Thread 3 waits SD	Thread 4 waits SA

Semaphores are not Silver Bullets

SA: **0**	SB: **0**	SC: **0**	SD: **0**
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1 waits SB	Thread 2 waits SC	Thread 3 waits SD	Thread 4 waits SA

Semaphores are not Silver Bullets

SA: **0** SB: **0** SC: **0** SD: **0**

Deadlock!

Thread 1	Thread 2	Thread 3	Thread 4
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
→ down(**SB**)	→ down(**SC**)	→ down(**SD**)	→ down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1 waits SB	Thread 2 waits SC	Thread 3 waits SD	Thread 4 waits SA

Semaphores are not Silver Bullets

Although it is much easier to work with semaphores than read/write operations, it is still possible to cause problems (e.g., deadlocks, starvation, incorrectness)

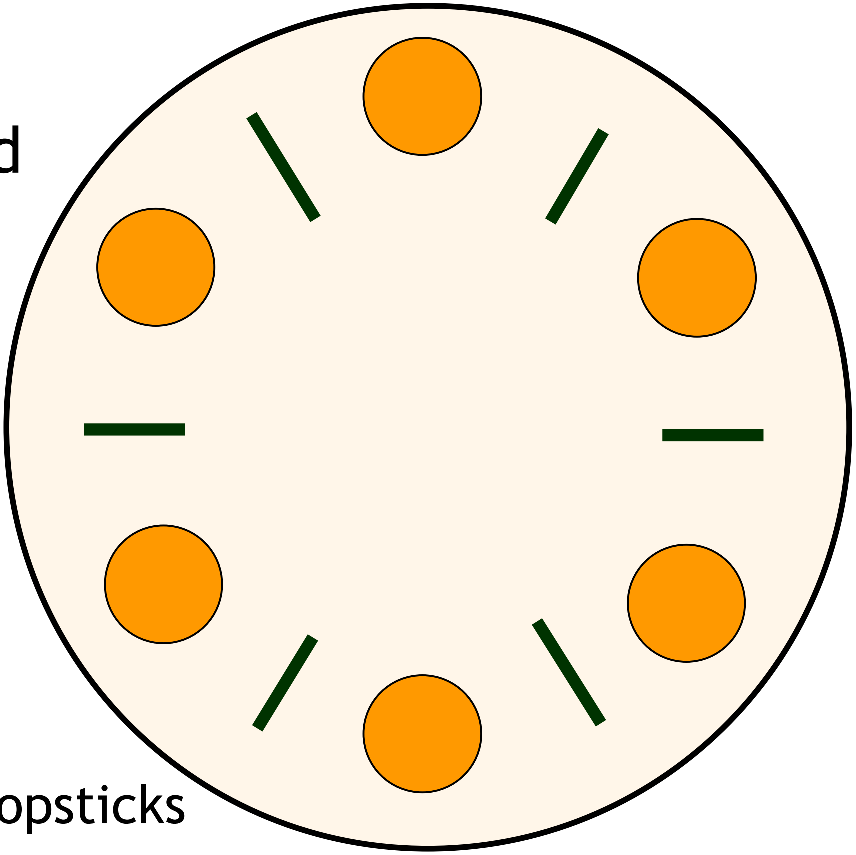
SA: **0**	SB: **0**	SC: **0**	SD: **0**
down(**SA**)	down(**SB**)	down(**SC**)	down(**SD**)
down(**SB**)	down(**SC**)	down(**SD**)	down(**SA**)
Move A→B	Move B→C	Move C→D	Move D→A
up(**SA**)	up(**SB**)	up(**SC**)	up(**SD**)
up(**SB**)	up(**SC**)	up(**SD**)	up(**SA**)
Thread 1 waits SB	Thread 2 waits SC	Thread 3 waits SD	Thread 4 waits SA

Dining Philosophers

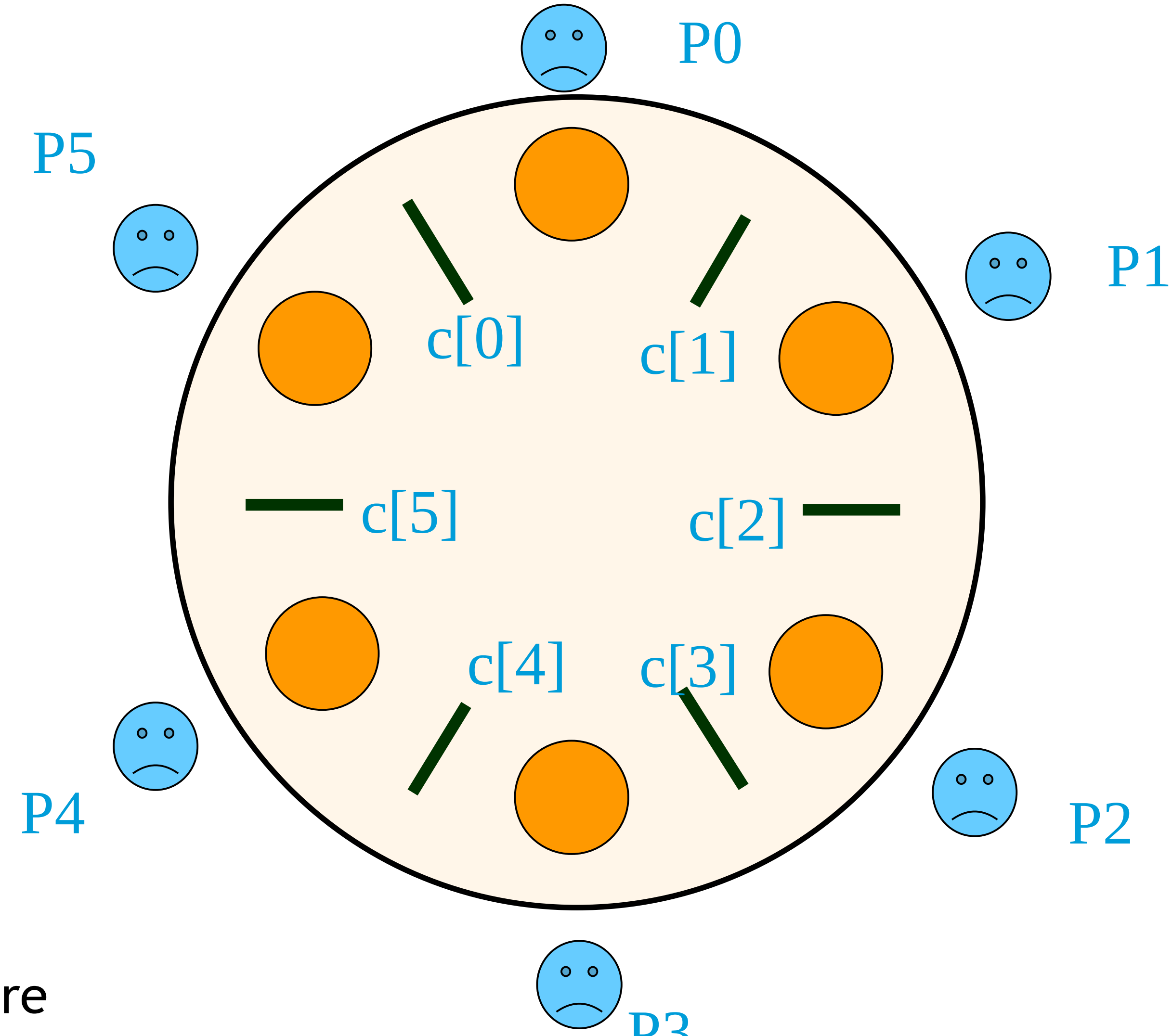
- Philosophers spend their life alternating between **eating** and **thinking**

- Eating requires 2 chopsticks:
  - take chopsticks
  - eat
  - put chopsticks
  - think

- Problem setting:
  1. Philosophers sit in the circle, chopsticks between them
  2. Pick one adjacent chopstick at a time
  3. Two philosophers cannot hold a chopstick together.



Code

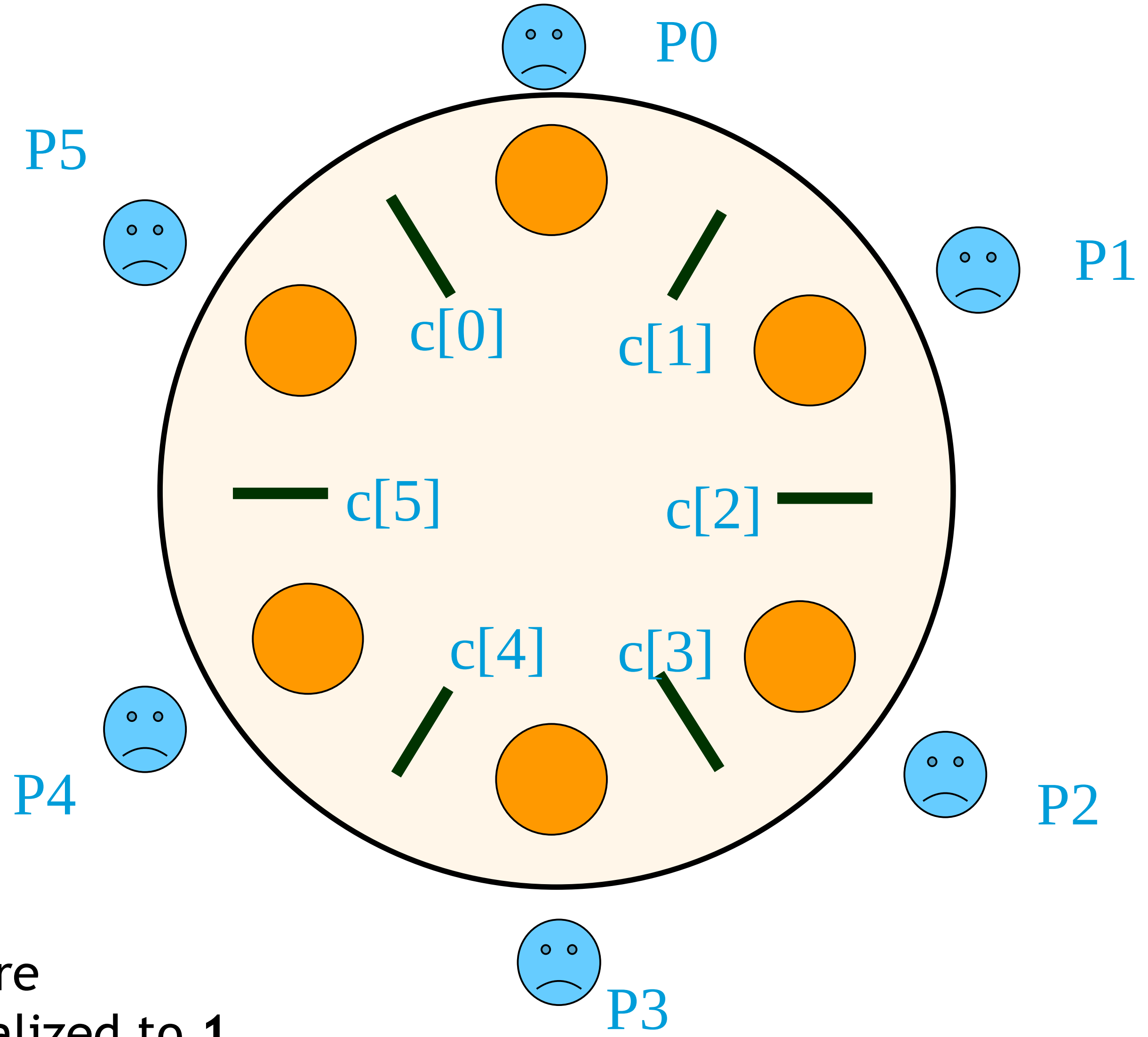


chopstick[0]…chopstick[5] are shared semaphores, all initialized to 1

Code that reaches a deadlock

down(**chopstick[(i+1)mode 6]**)
down(**chopstick[(i)]**)
Eat
up(**chopstick[(i+1)mode 6]**)
up(**chopstick[(i)]**)
Think

Code for philosopher i

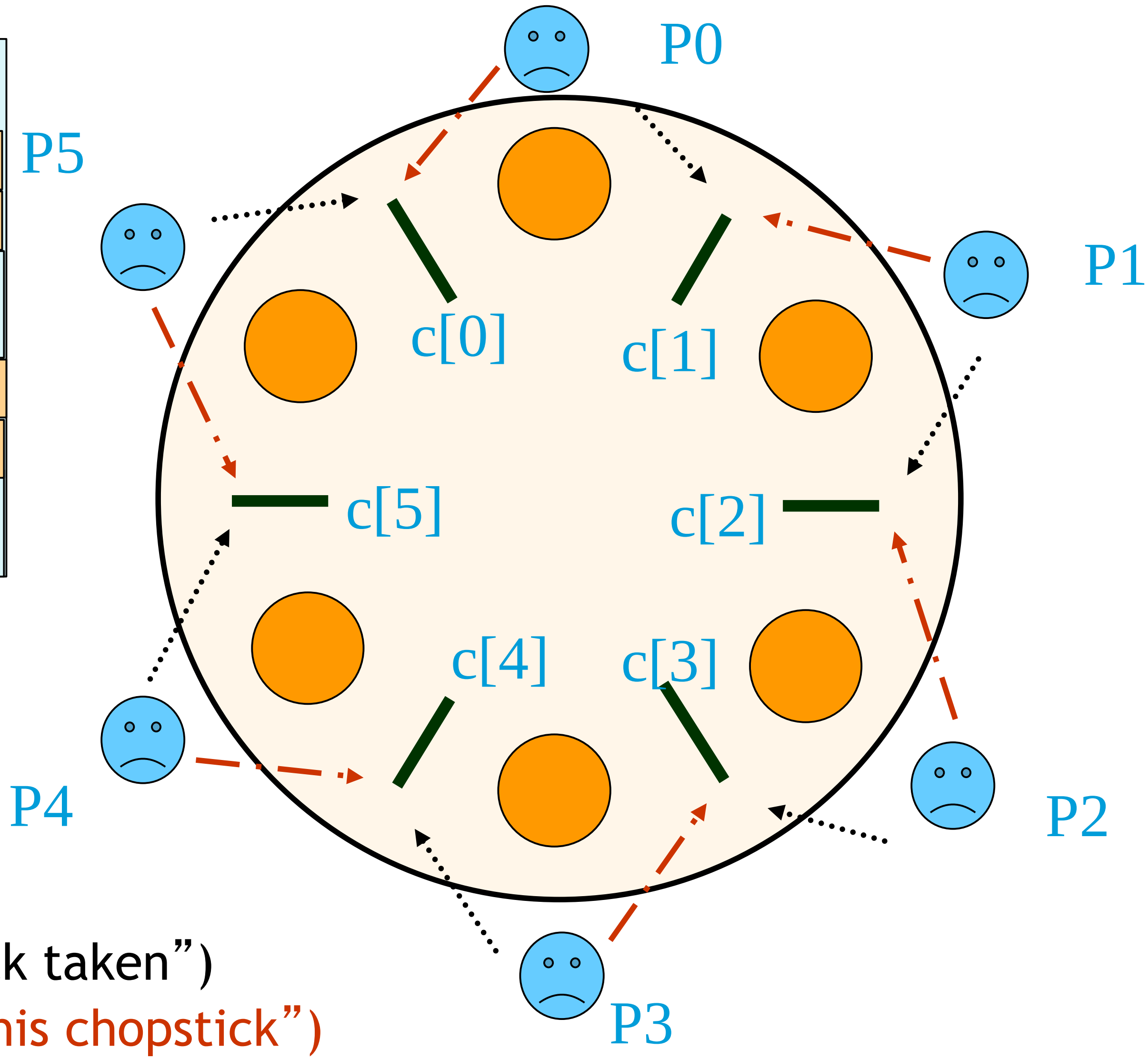


chopstick[0]…chopstick[5] are
shared semaphores, all initialized to 1

Code that reaches a deadlock

down(**chopstick[(i+1)mode 6]**)
down(**chopstick[(i])**)
Eat
up(**chopstick[(i+1)mode 6]**)
up(**chopstick[(i])**)
Think

Code for philosopher i



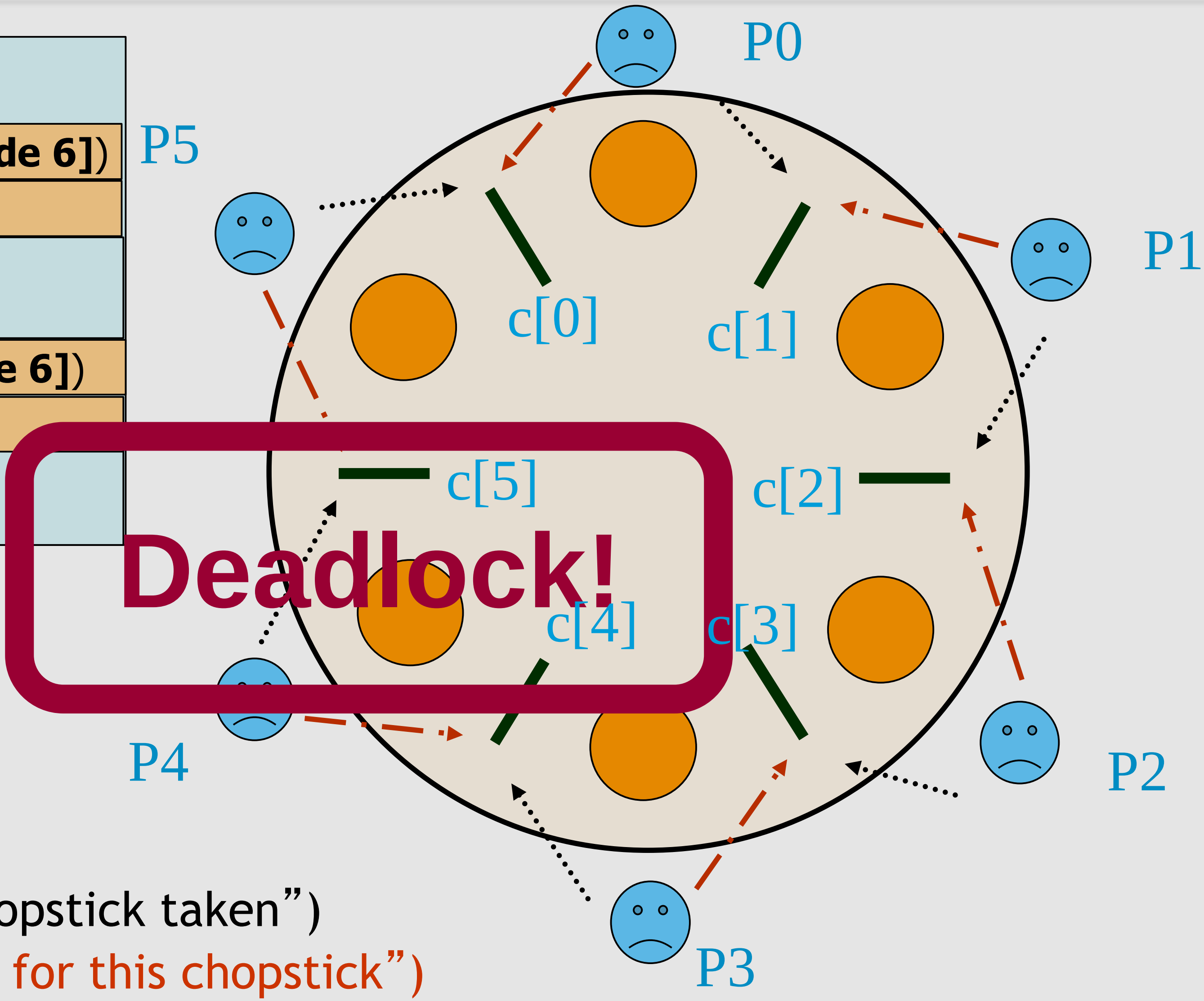
(means "first chopstick taken")

(means "waiting for this chopstick")

Code that reaches a deadlock

down(**chopstick[(i+1)mode 6]**)
down(**chopstick[(i]**)
Eat
up(**chopstick[(i+1)mode 6]**)
up(**chopstick[(i]**)
Think

Code for philosopher i



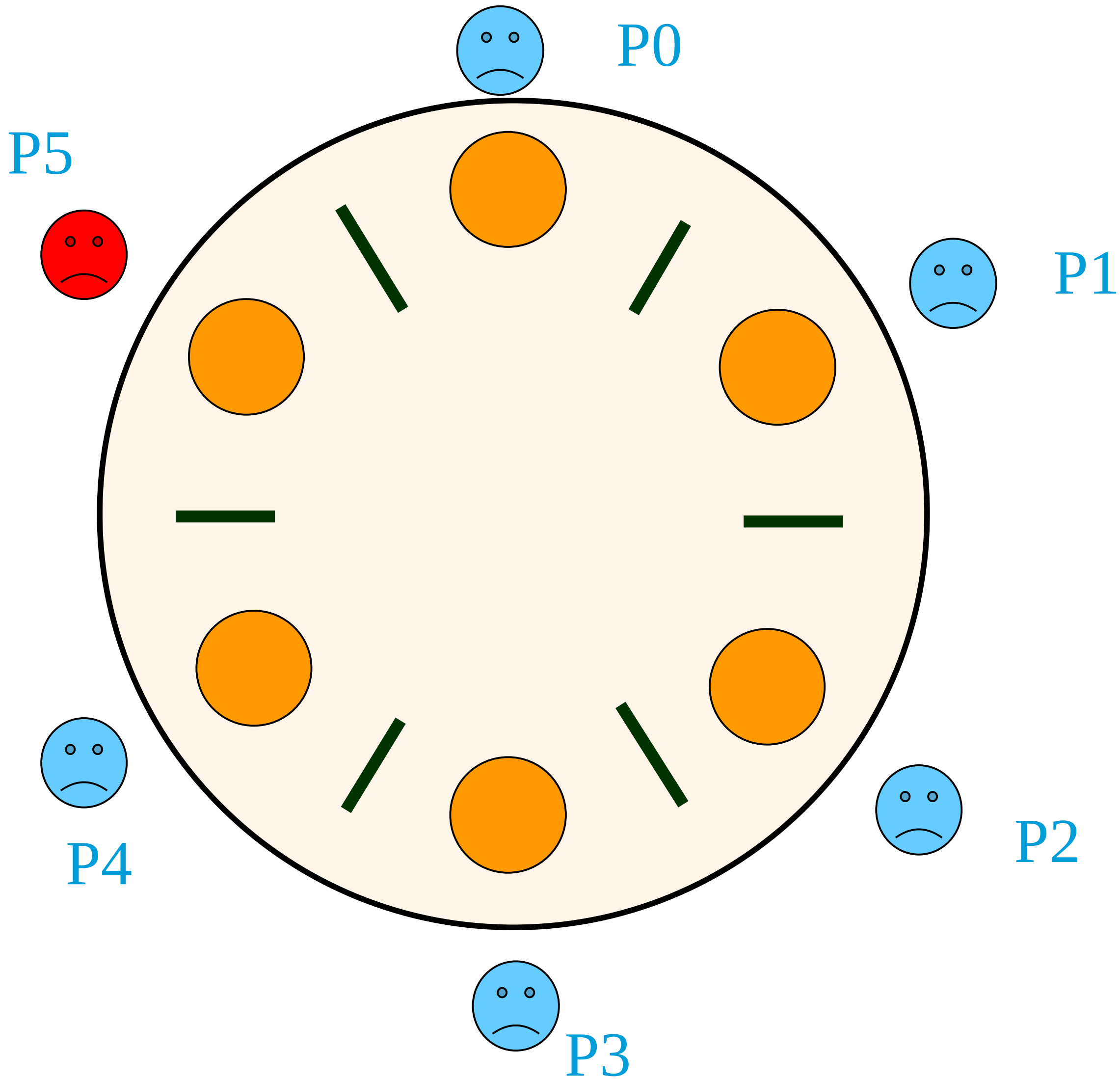
(⋯⋯→ means "first chopstick taken")

(— ·→ means "waiting for this chopstick")

Different code for philosopher 5

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i]**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

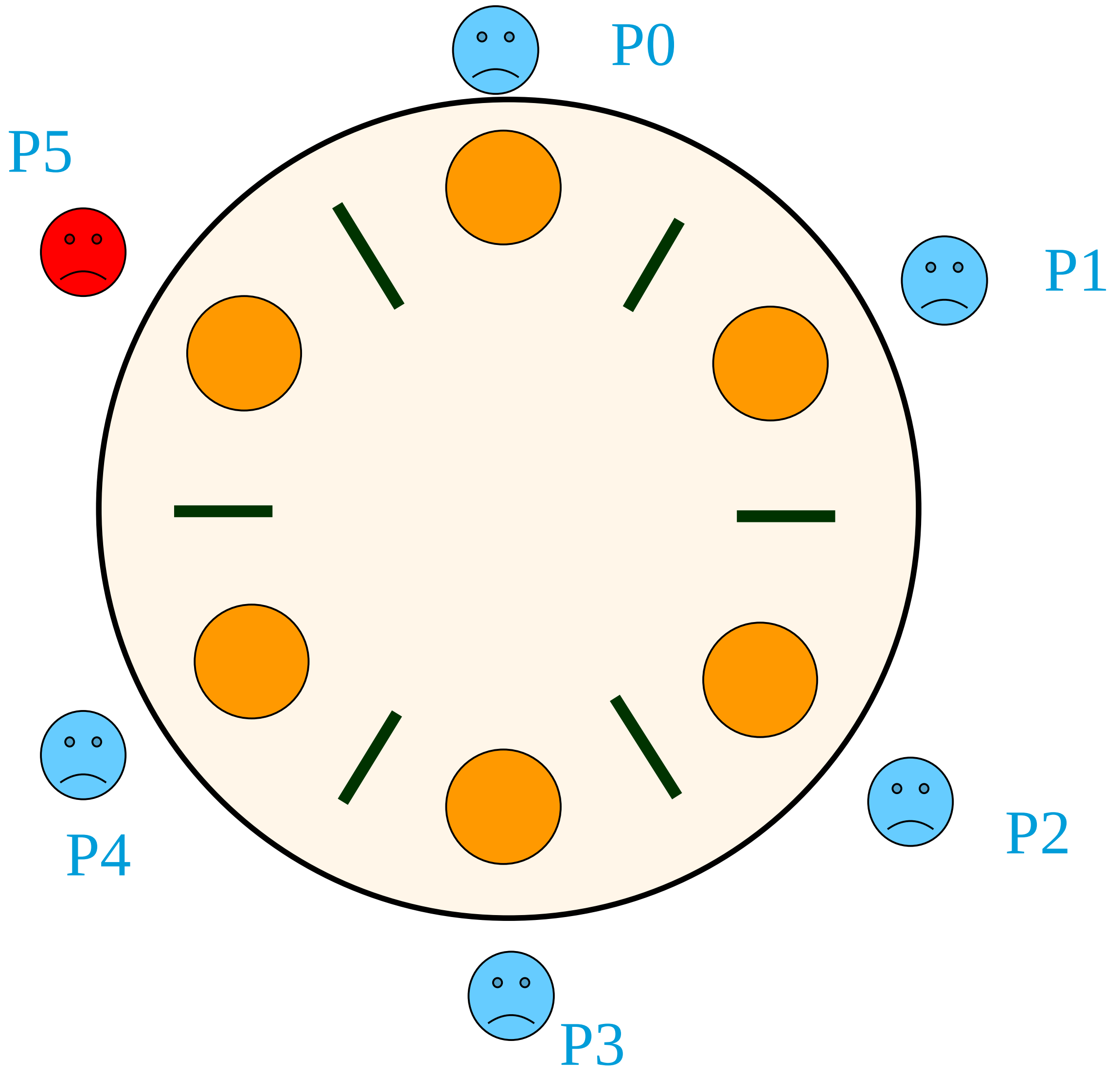
down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think



Different code for philosopher 5

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i]**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think

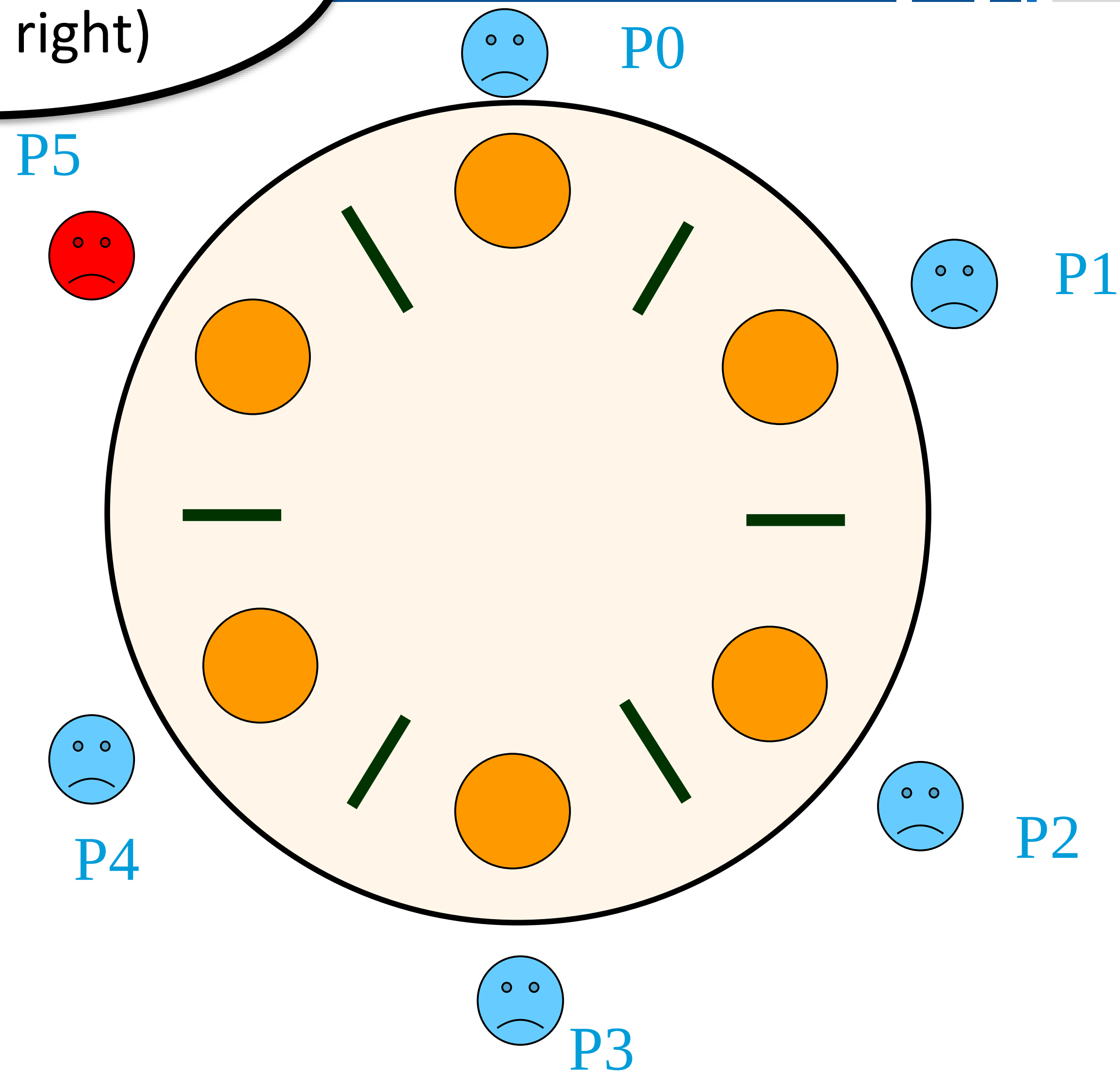


Differen ilosopher 5

Same code for philosophers 0,…,4 (left before right)

```
down(chopstick[(i+1)mod 6])
down(chopstick[(i)])
Eat
up(chopstick[(i+1)mod 6])
up(chopstick[i])
Think
```

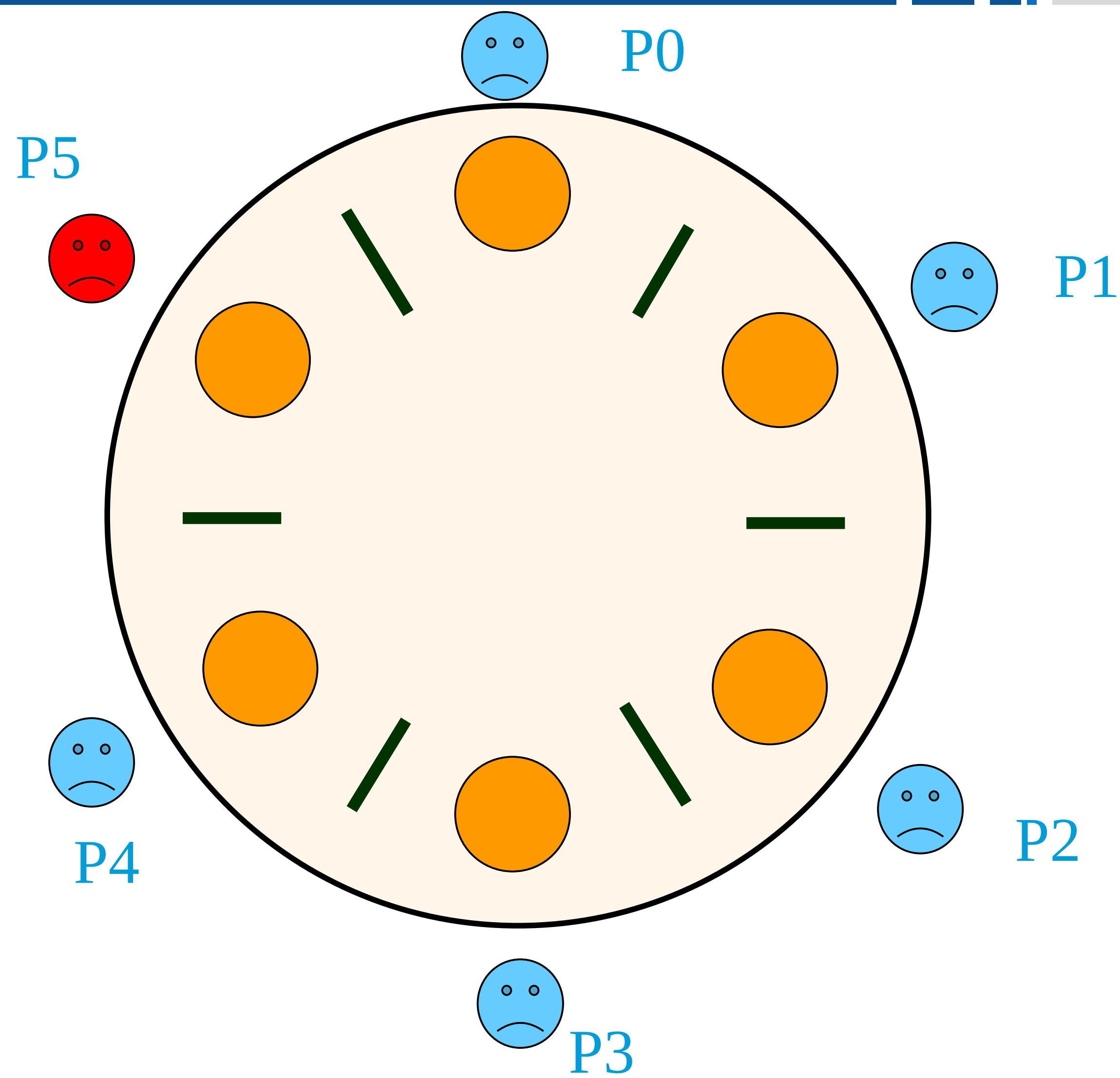
```
down(chopstick[5])
down(chopstick[0])
Eat
up(chopstick[0])
up(chopstick[5])
Think
```



Different code for philosopher 5

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i]**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

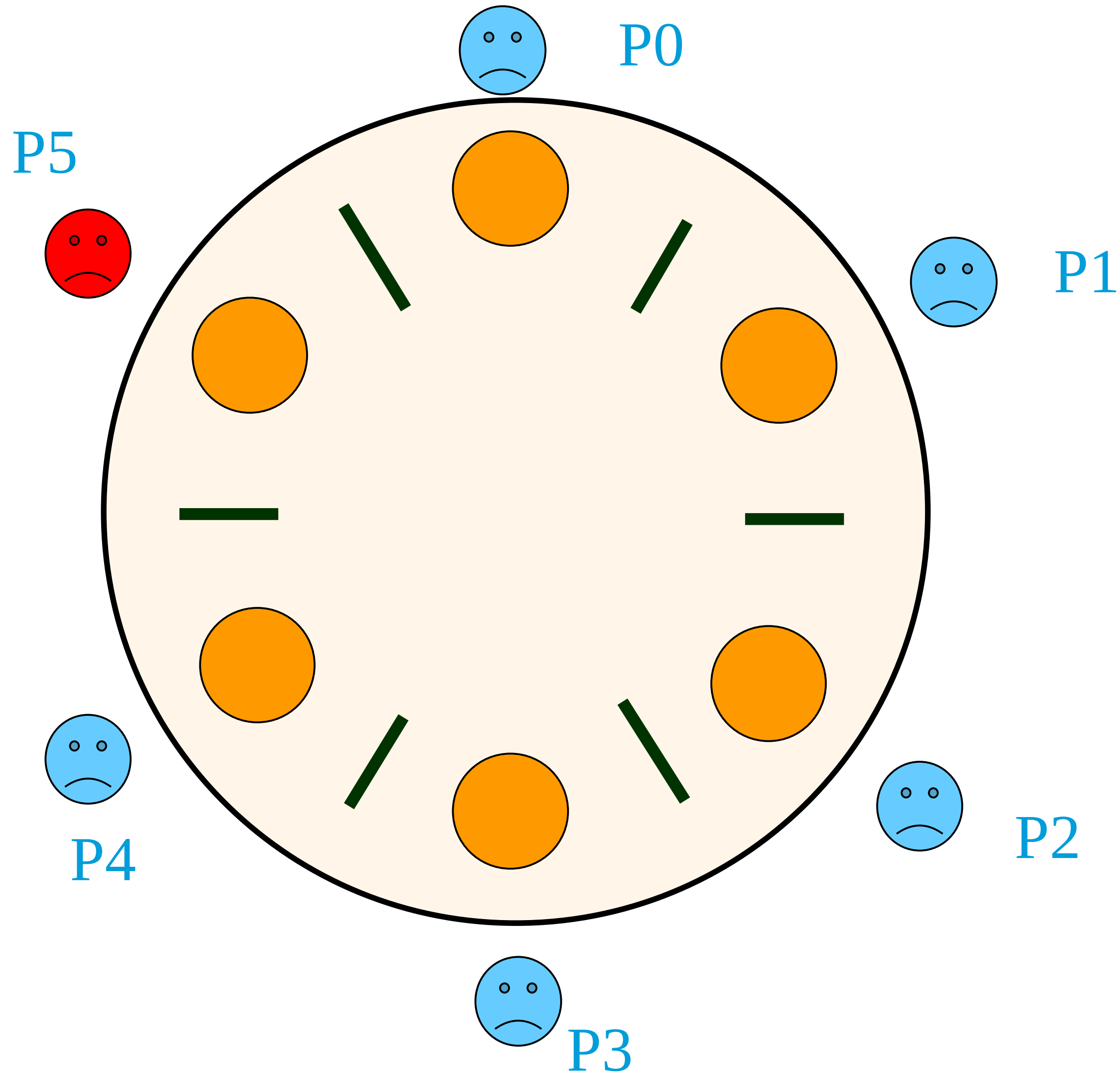
down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think



Different code for philosopher 5

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i]**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

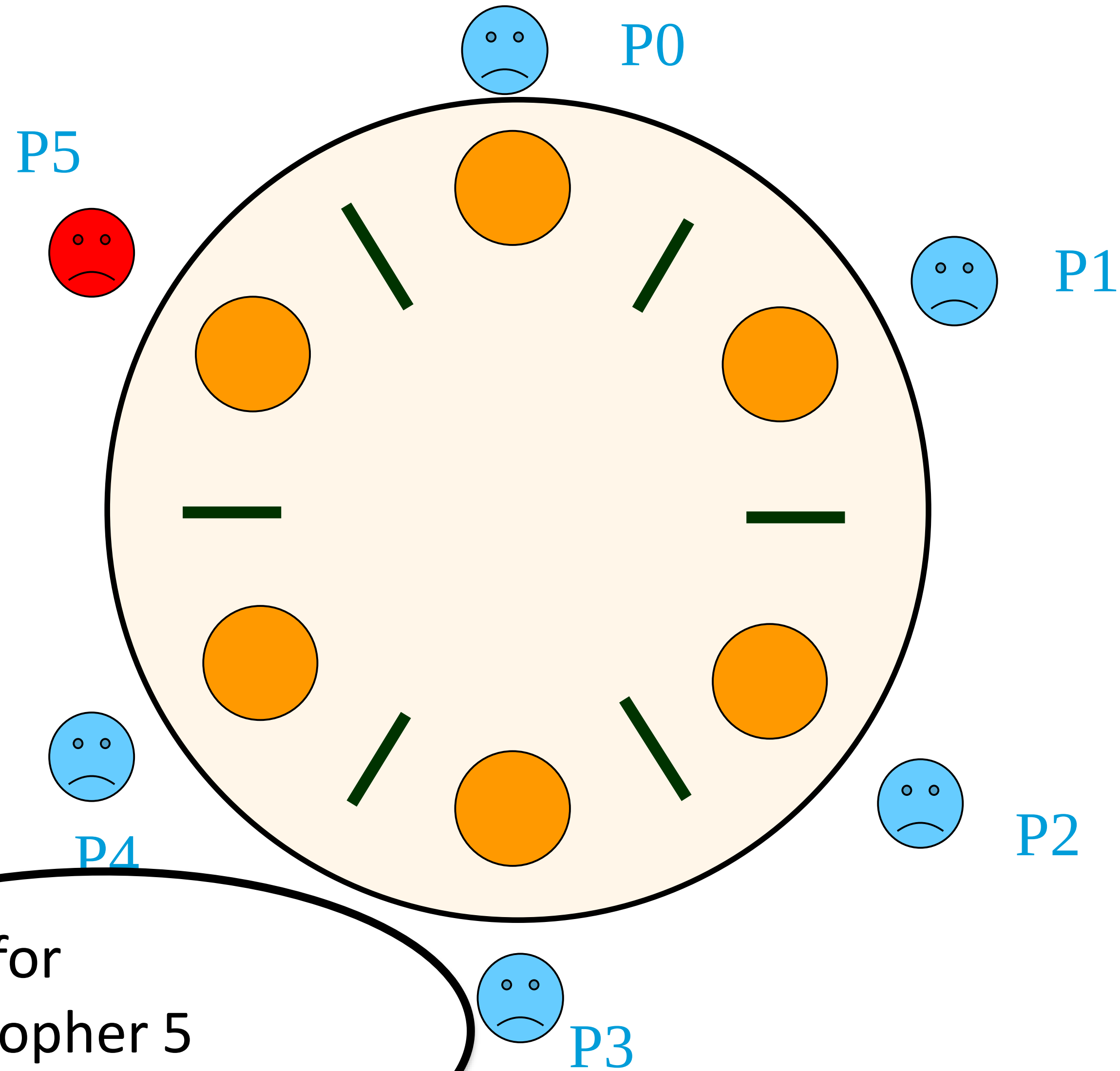
down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think



Different code for philosopher 5

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i]**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chops**
up(**chopstick[5**
Think

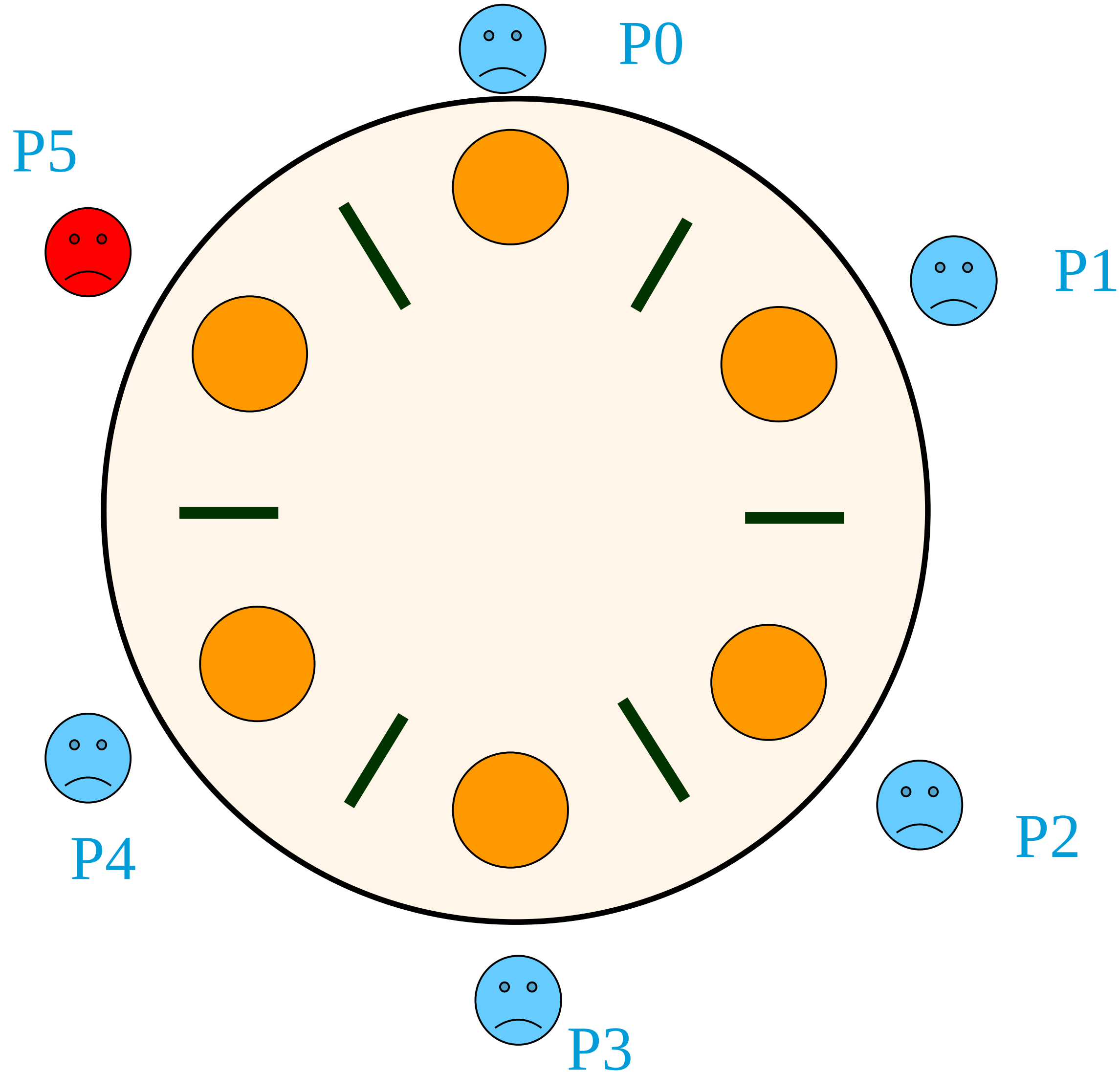


Code for philosopher 5 (right before left)

Different code for philosopher 5

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i]**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think



Progress

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i]**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think

If a philosopher tries to take a chopstick, then eventually some philosopher will eat.

Assume philosopher i tries to take a chopstick but no philosopher will be eating → no philosopher was able to grab the second chopstick.

Progress

down(**chopstick[(i+1)mod 6]**)

down(**chopstick[(i]**)

Eat

up(**chopstick[(i+1)mod 6]**)

up(**chopstick[i]**)

Think

down(**chopstick[5]**)

down(**chopstick[0]**)

Eat

up(**chopstick[0]**)

up(**chopstick[5]**)

Think

i is either 0,1,2, or 3

- i tries to takes the chopstick i+1, but does not succeed → philosopher i+1 took chopstick i+1(its second one) → Contradiction.
- i took chopstick i+1 and wait for chopstick i → philosopher i-1 took chopstick i and wait for i-1 → … → philosopher 0 took 1 and wait for 0 → philosopher 5 took 0 → philosopher 5 will eat

Progress

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i])**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think

i = 4

- Philosopher 4 tries to takes the chopstick 5, but does not succeed → philosopher 5 took chopstick 5 and wait for 0 → philosopher 0 took chopstick 0 (his second one). Contradiction.
- i took chopstick i+1 and wait for chopstick i → philosopher i-1 took chopstick i and wait for i-1 → … → philosopher 0 took 1 and wait for 0 → philosopher 5 took 0 → philosopher 5 will eat

Progress

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i])**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think

i = 4

- Philosopher 4 tries to takes the chopstick 5, but does not succeed → philosopher 5 took chopstick 5 and wait for 0 → philosopher 0 took chopstick 0 (his second one). Contradiction.
- i took chopstick i+1 and wait for chopstick i → philosopher i-1 took chopstick i and wait for i-1 → … → philosopher 0 took 1 and wait for 0 → philosopher 5 took 0 → philosopher 5 will eat

Progress

down(**chopstick[(i+1)mod 6]**)
down(**chopstick[(i])**)
Eat
up(**chopstick[(i+1)mod 6]**)
up(**chopstick[i]**)
Think

down(**chopstick[5]**)
down(**chopstick[0]**)
Eat
up(**chopstick[0]**)
up(**chopstick[5]**)
Think

i = 5

- Philosopher 5 tries to takes the chopstick 0, but does not succeed → philosopher 0 took chopstick 0 (his second one). Contradiction.
- Philosopher 5 tries to takes the chopstick 5, but does not succeed → philosopher 4 took chopstick 5 and wait for 4 → … → philosopher 0 took chopstick 1 and waits for 0, but 0 cannot be taken as philosopher 5 still waits on chopstick 5 → philosopher 0 will grab chopstick 0. Contradiction.

(Necessary) Conditions for Deadlock (Coffman Conditions)

1. Mutual Exclusion

   At least one resource must be held in a non-shareable mode

2. Hold and Wait

   A thread is holding one resource and is requesting a resource that is held by another thread

3. Non-Preemptive Allocation

   A resource can be released only voluntarily by the process holding it

4. Circular Wait

(Necessary) Conditions for Deadlock (Coffman Conditions)

1. Mutual Exclusion

   At least one resource must be held in a non-shareable mode

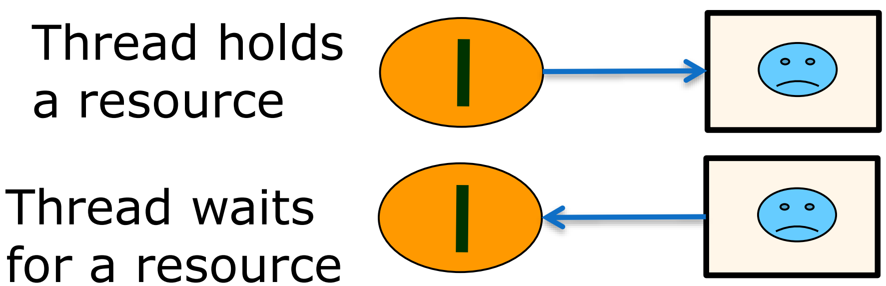
2. Hold and Wait

   A thread is holding one resource and is requesting a resource that is held by another thread

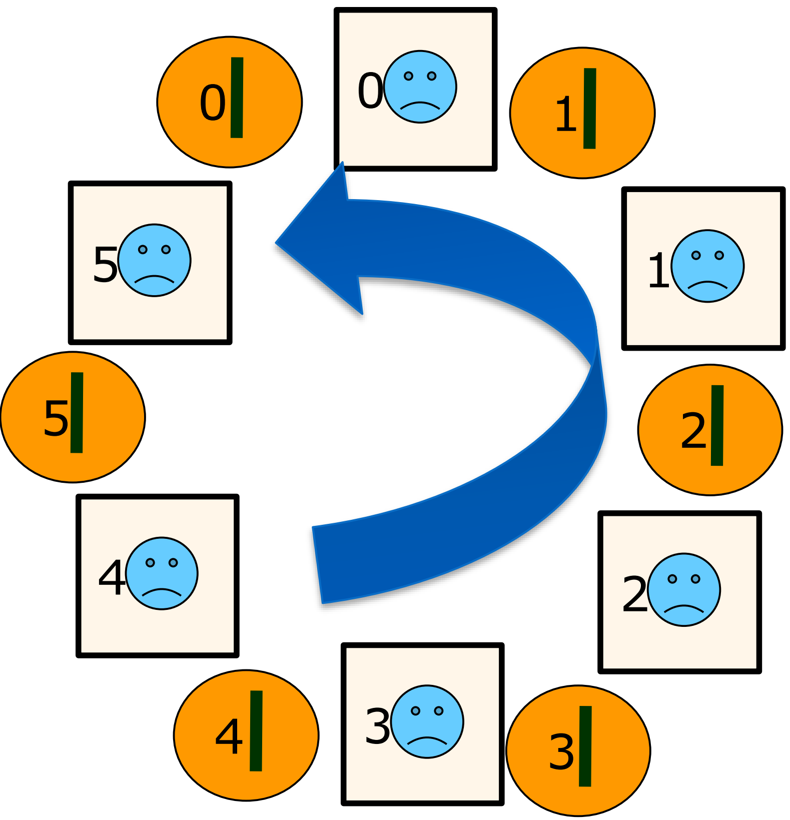
3. Non-Preemptive Allocation

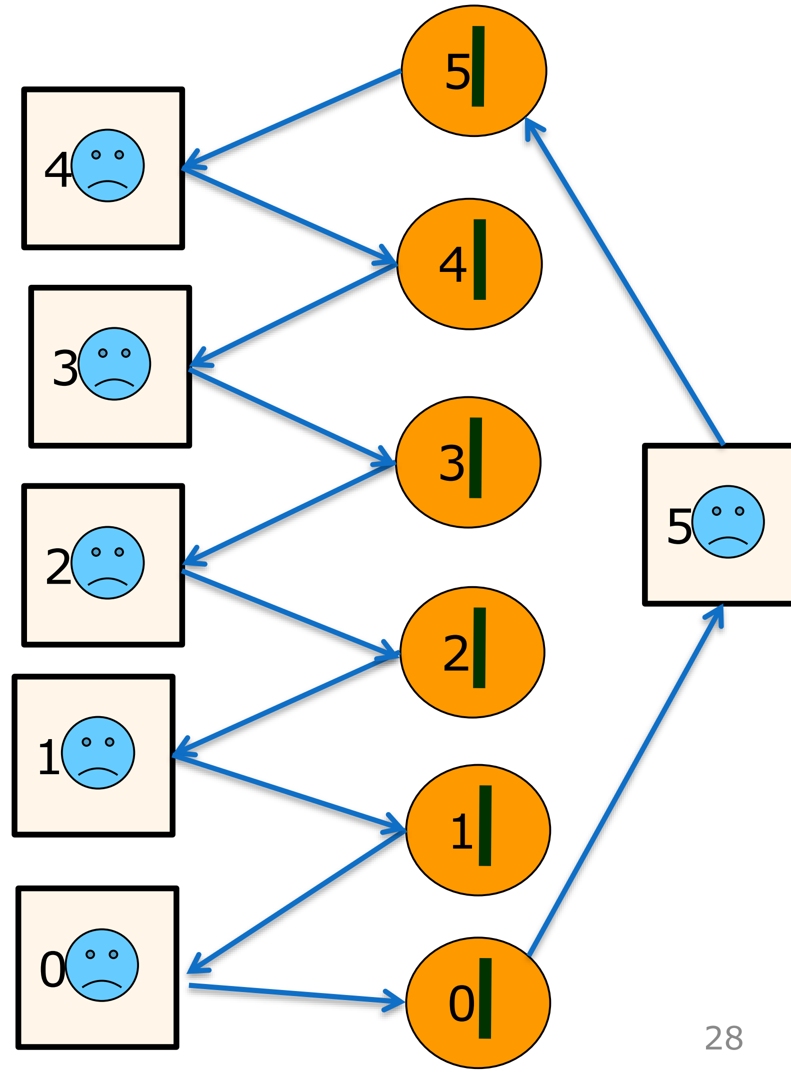
   A resource can be released only voluntarily by the process holding it

4. Circular Wait



Dinning Philosophers' Circular Wait





Better Solution: Breaking Symmetry (a.k.a. the LR solution)

Odd threads	Even threads
down(**chopstick[(i+1)mod 6]**)	down(**chopstick[(i]**)
down(**chopstick[(i]**)	down(**chopstick[(i+1)mod 6]**)
Eat	Eat
up(**chopstick[(i+1)mod 6]**)	up(**chopstick[(i+1)mod 6]**)
up(**chopstick[(i]**)	up(**chopstick[(i]**)
Think	Think

- Provides better concurrency (2 philosophers can always eat concurrently, instead of just one)

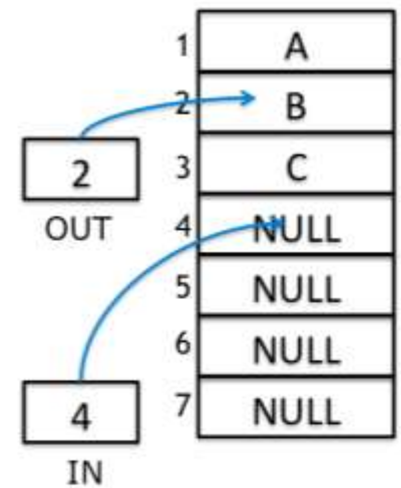
Final word about deadlocks

- Usually, cannot be detected by the OS
- Do not happen in every execution
  - Very tricky bug!
- Good practice:
  1. Order the resources
  2. Always lock the resources in the same order
     - E.g. from high to low
  3. Release resources in reverse order

Next Problem: Producer-Consumer

Motivation: Two processes (or threads) want to print

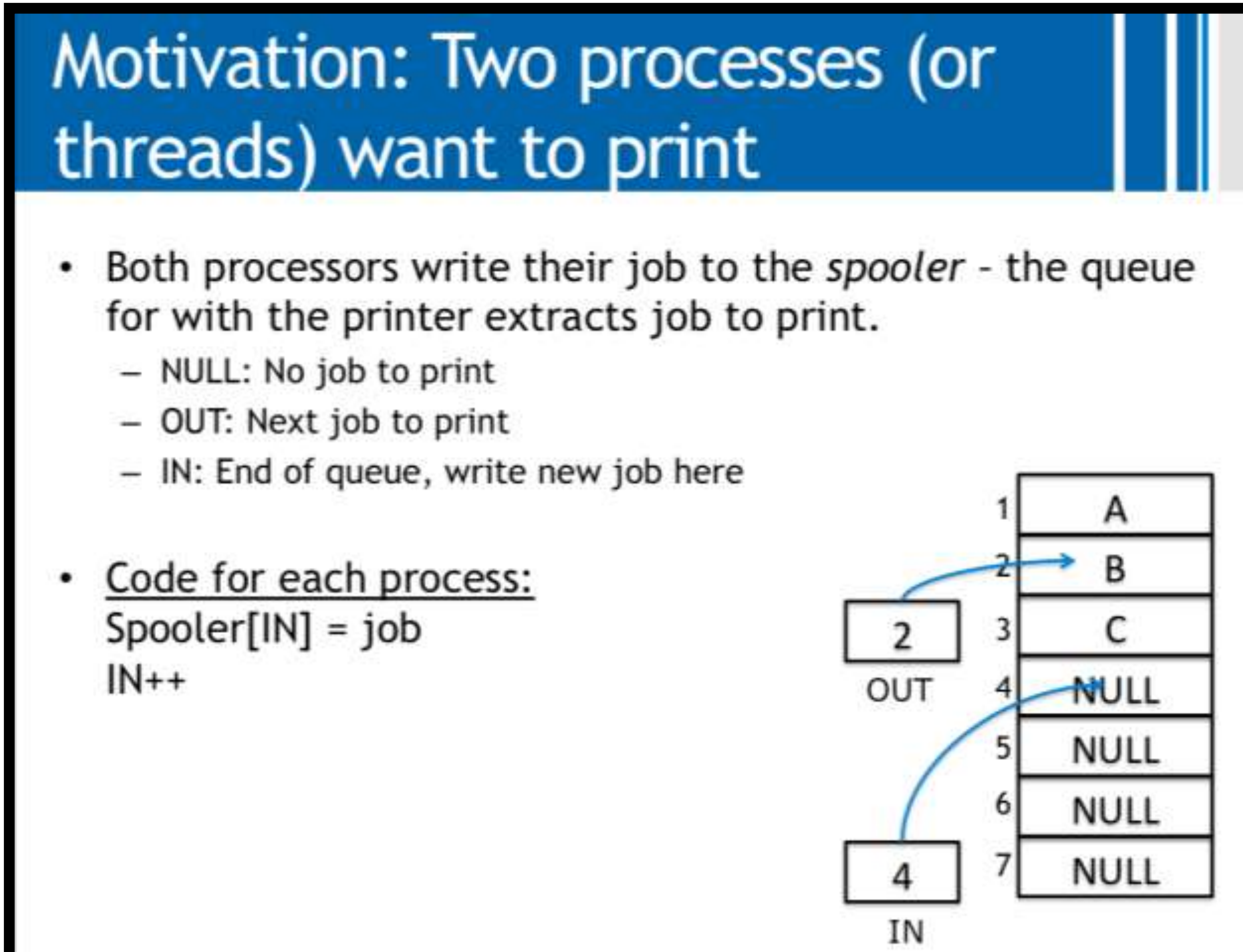
- Both processors write their job to the *spooler* - the queue for with the printer extracts job to print.
  - NULL: No job to print
  - OUT: Next job to print
  - IN: End of queue, write new job here
- <u>Code for each process:</u>
  Spooler[IN] = job
  IN++



Next Problem: Producer-Consumer

We already saw that half of the problem:

A contention between two printing processes (**producers**) on the IN counter can lead to overriding jobs and gaps in the array (**buffer**)

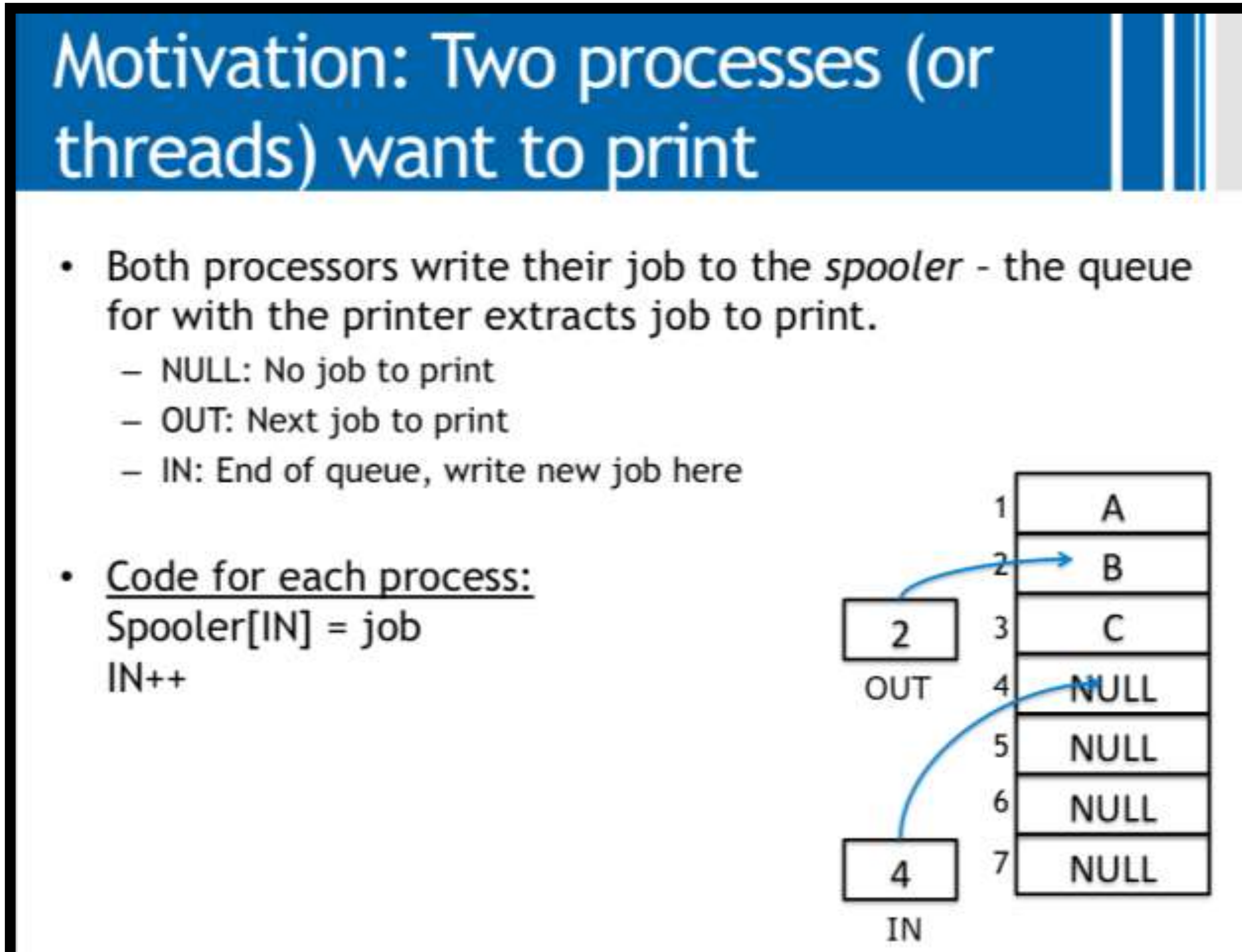


Next Problem: Producer-Consumer

We already saw that half of the problem:

A contention between two printing processes (**producers**) on the IN counter can lead to overriding jobs and gaps in the array (**buffer**)

Similar situations can happen on the OUT variables in case there is more than one printer process (**consumers**)



Motivation: Two processes (or threads) want to print

- Both processors write their job to the *spooler* - the queue for with the printer extracts job to print.
  - NULL: No job to print
  - OUT: Next job to print
  - IN: End of queue, write new job here
- Code for each process:
  Spooler[IN] = job
  IN++

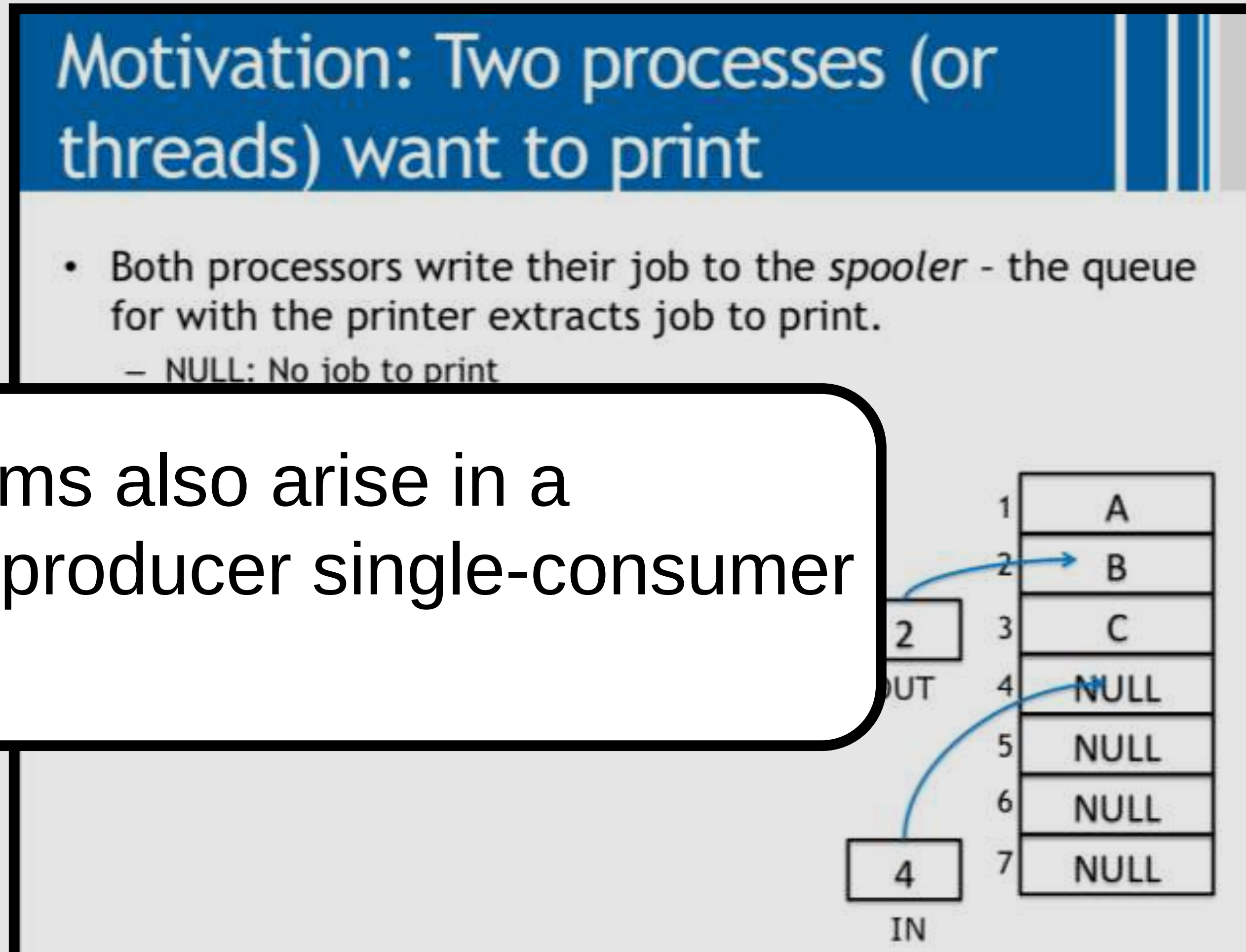
In case the **buffer** is of **bounded** size, how it is managed (cyclic operation) and how empty/full buffers are handled.

Next Problem: Producer-Consumer

We already saw that half of the problem:

A contention between two printing processes (**producers**) on the IN counter can lead to overriding job gaps in the ar (**buffer**)

Similar situat happen on th variables in case there is more than one printer process (**consumers**)

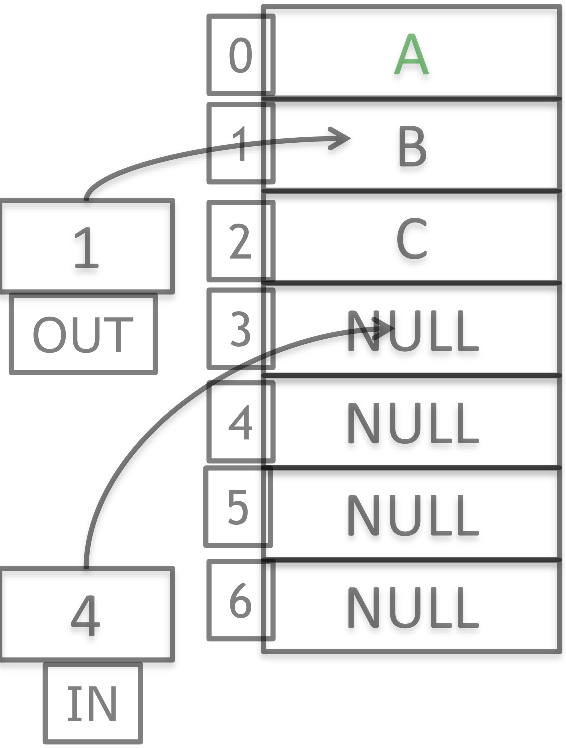


Problems also arise in a single-producer single-consumer setting

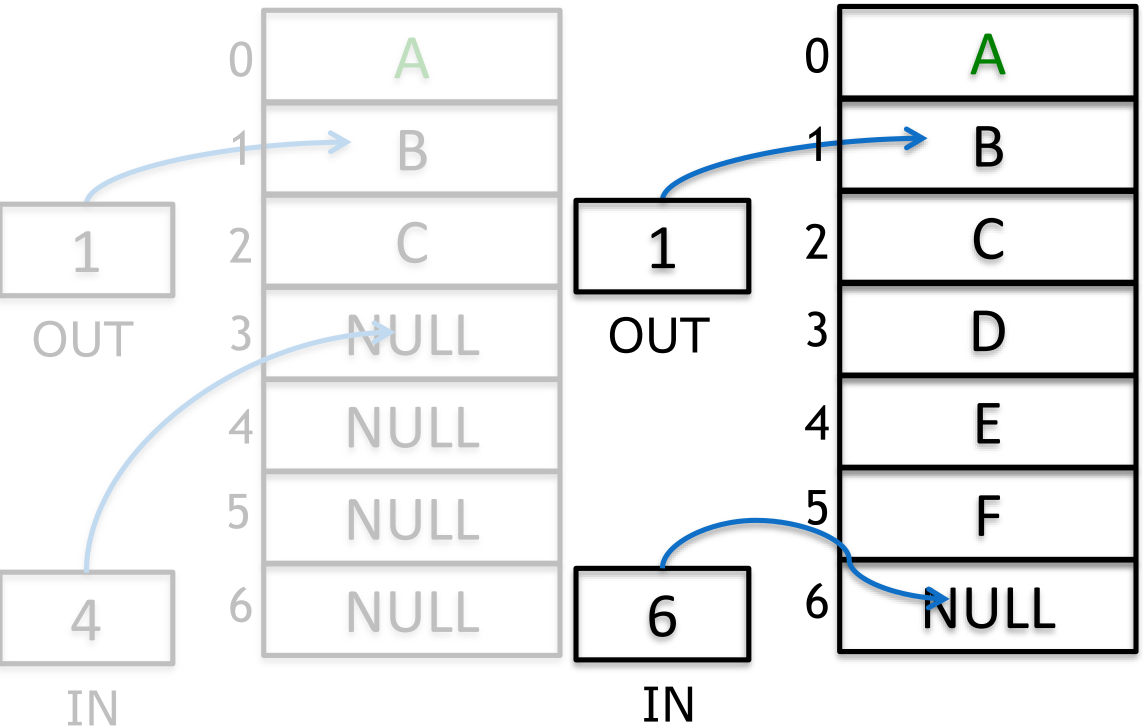
In case the **buffer** is of **bounded** size, how it is managed (cyclic operation) and how empty/full buffers are handled.

Cyclic Buffers

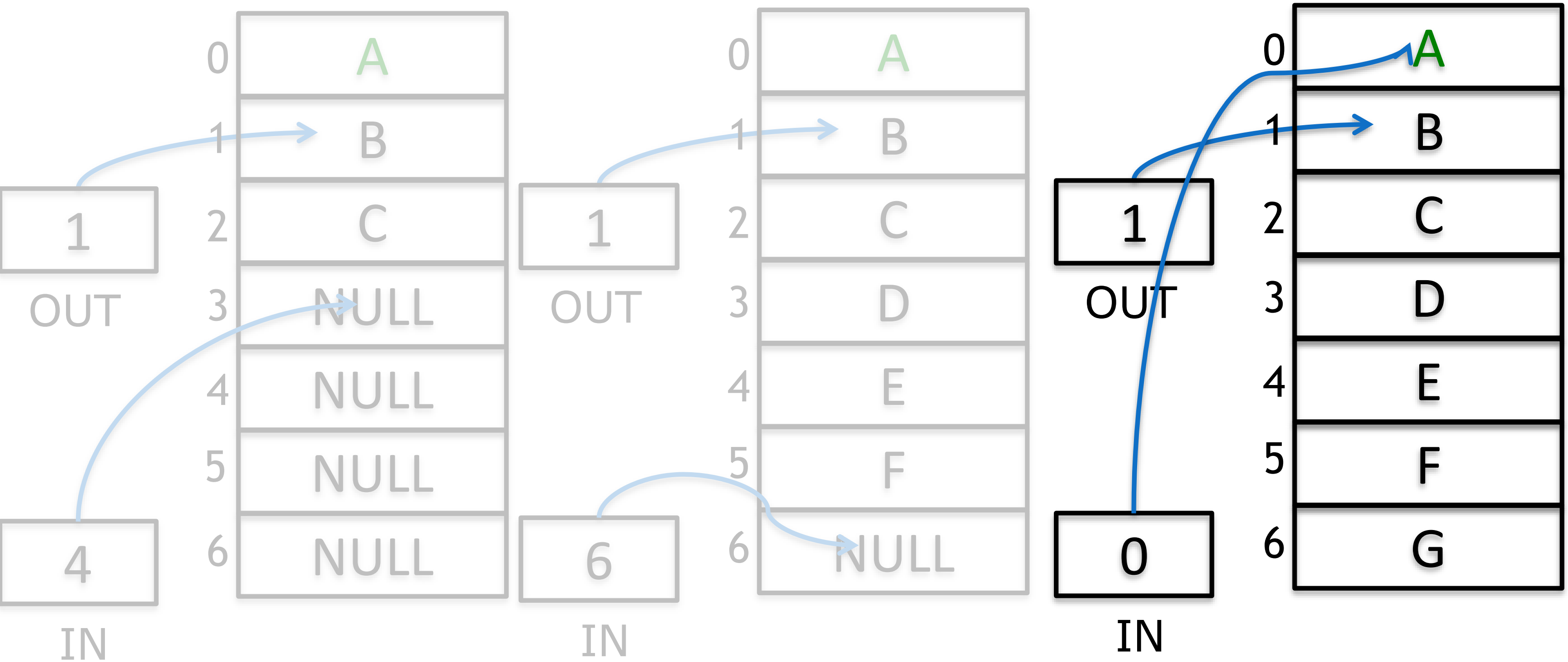
Cyclic Buffers



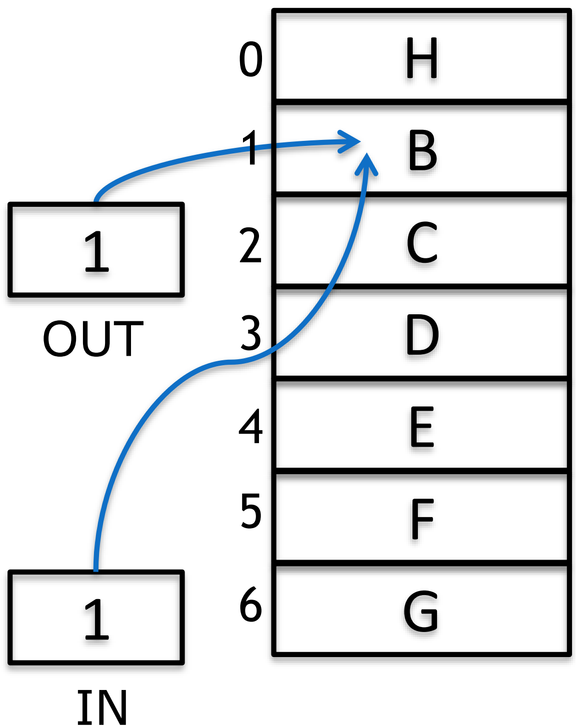
Cyclic Buffers



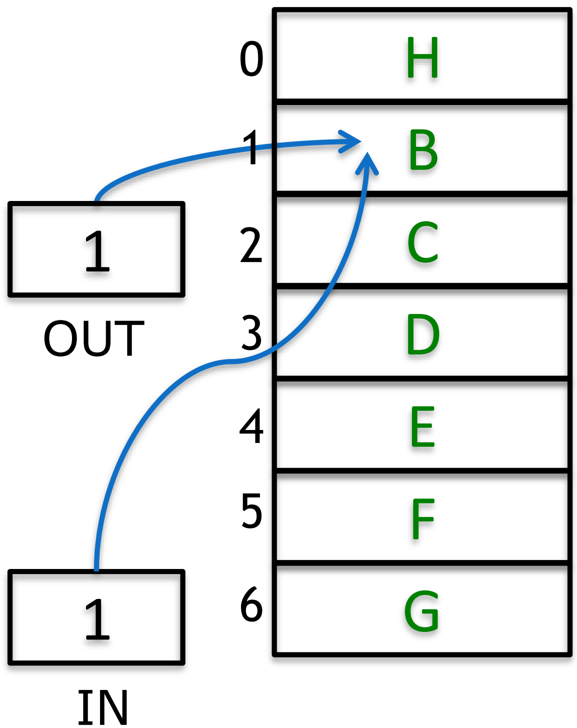
Cyclic Buffers



Cyclic Buffers

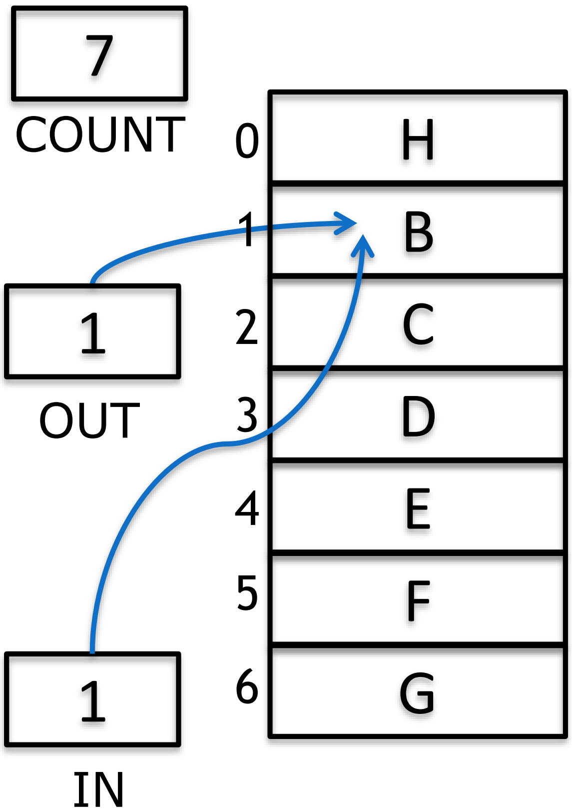


Buffer is full

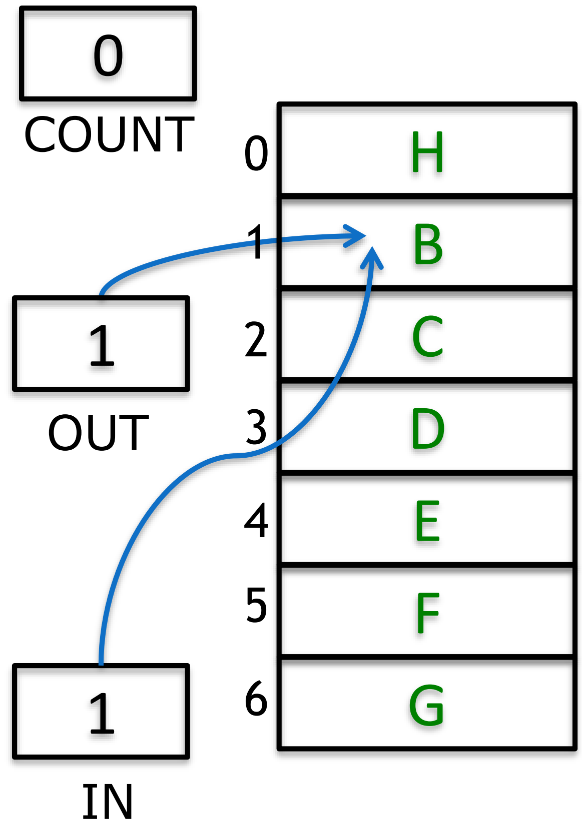


Buffer is empty

Cyclic Buffers



Buffer is full



Buffer is empty

Single Producer- Single Consumer Code

Producer
(e.g., sending printing job)

```
while (COUNT==n);

buffer [IN]=job;
IN=IN+1 mod n;
COUNT++;
```

Consumer
(e.g., printer)

```
while (COUNT==0);

job=buffer [OUT];
OUT=OUT+1 mod n;
COUNT--;
```

Single Producer- Single Consumer Code

Producer
(e.g., sending printing job)

```
while (COUNT==n);

buffer [IN]=job;
IN=IN+1 mod n;
COUNT++;
```

Must be atomic

Consumer
(e.g., printer)

```
while (COUNT==0);

job=buffer [OUT];
OUT=OUT+1 mod n;
COUNT--;
```

Single Producer- Single Consumer Code

Producer
(e.g., sending printing job)

while (COUNT==n);

buffer [IN]=job;
IN=IN+1 mod n;
COUNT++;

Consumer
(e.g., printer)

while (COUNT==0);

job=buffer [OUT];
OUT=OUT+1 mod n;
COUNT--;

Use semaphore
(mutual exclusion)

Must be atomic

Single Producer- Single Consumer Code

Producer
(e.g., sending prin

Busy wait

while (COUNT==n);

buffer [IN]=job;
IN=IN+1 mod n;
COUNT++;

Must be atomic

Consumer
g., printer)

while (COUNT==0);

job=buffer [OUT];
OUT=OUT+1 mod n;
COUNT--;

Use semaphore
(mutual exclusion)

Alternative Solution: Producers-Consumers w/ Semaphores

Producer
(e.g., sending printing job)

down(empty)
down(mutex)

buffer [IN]=job;
IN=IN+1 mod n;

up(mutex)
up(full)

Consumer
(e.g., printer)

down(full)
down(mutex)

job=buffer [OUT];
OUT=OUT+1 mod n;

up(mutex)
up(empty)

Initial values:

mutex	1
empty	n
full	0

Alternative Solution: Producers-Consumers w/ Semaphores

Producer
(e.g., sending printing job)

down(empty)
down(mutex)

buffer [IN]=job;
IN=IN+1 mod n;

up(mutex)
up(full)

Consumer
(e.g., printer)

down(full)
down(mutex)

job=buffer [OUT];
OUT=OUT+1 mod n;

up(mutex)
up(empty)

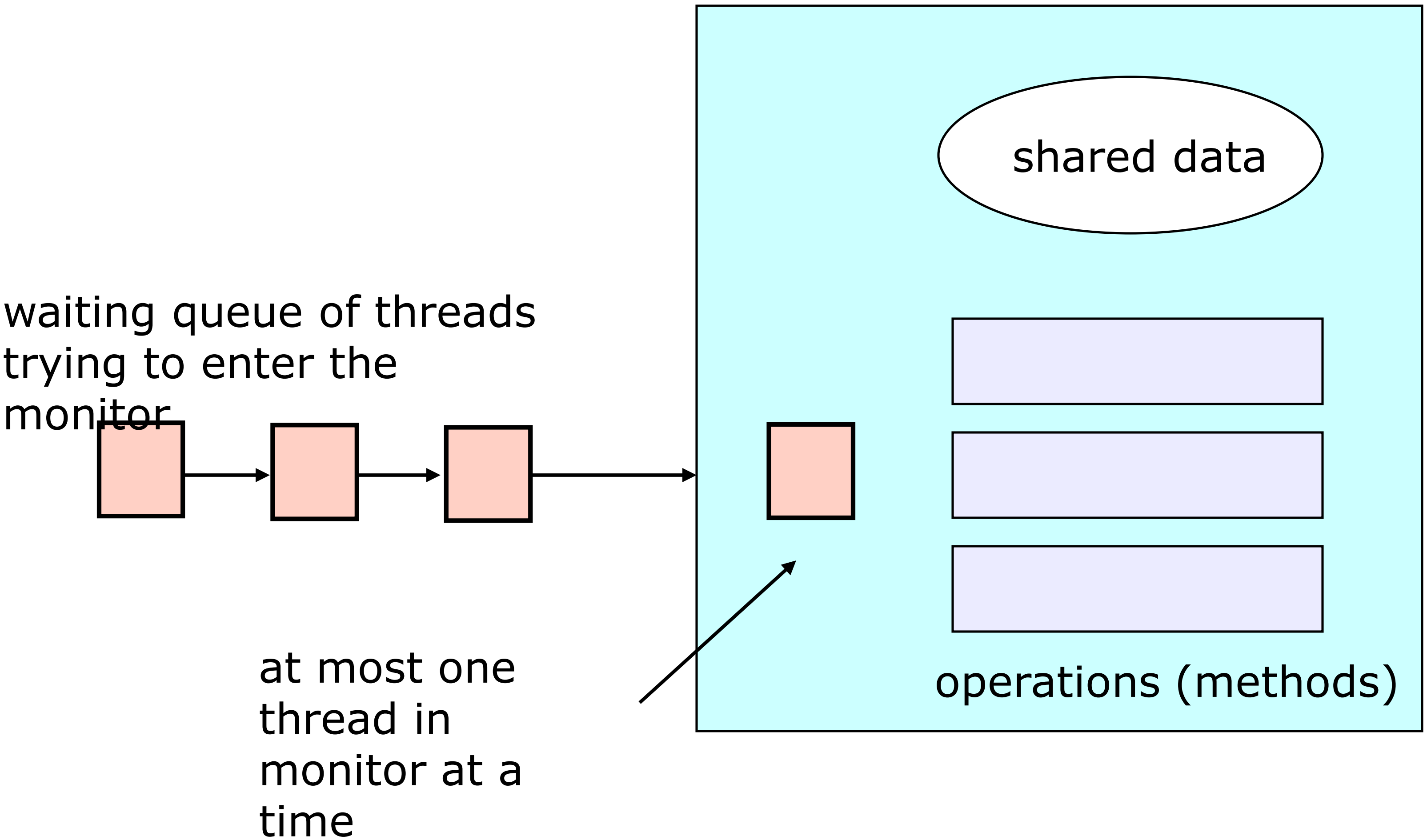
Initial values:

mutex	1
empty	n
full	0

One More Approach: Monitors

- A *monitor* is a <u>programming language</u> construct that supports controlled access to shared data
  - synchronization code is added by the compiler
    - why does this help?
- A monitor encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent threads that invoke those procedures
- Data can only be accessed from within the monitor, using the provided procedures
  - protects the data from unstructured access
- Addresses the key usability issues that arise with semaphores

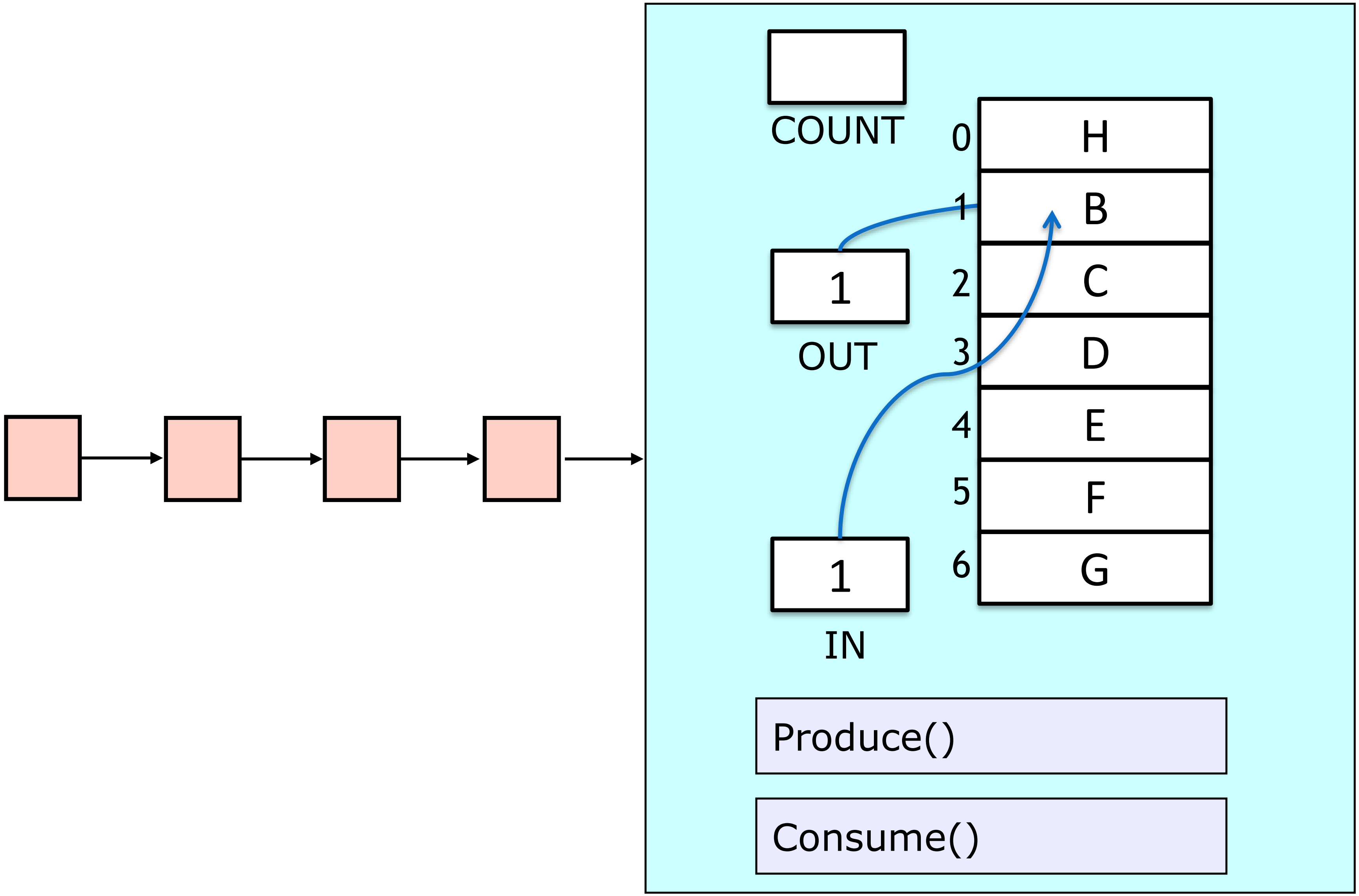
A monitor


shared data
waiting queue of threads trying to enter the monitor
at most one thread in monitor at a time
operations (methods)

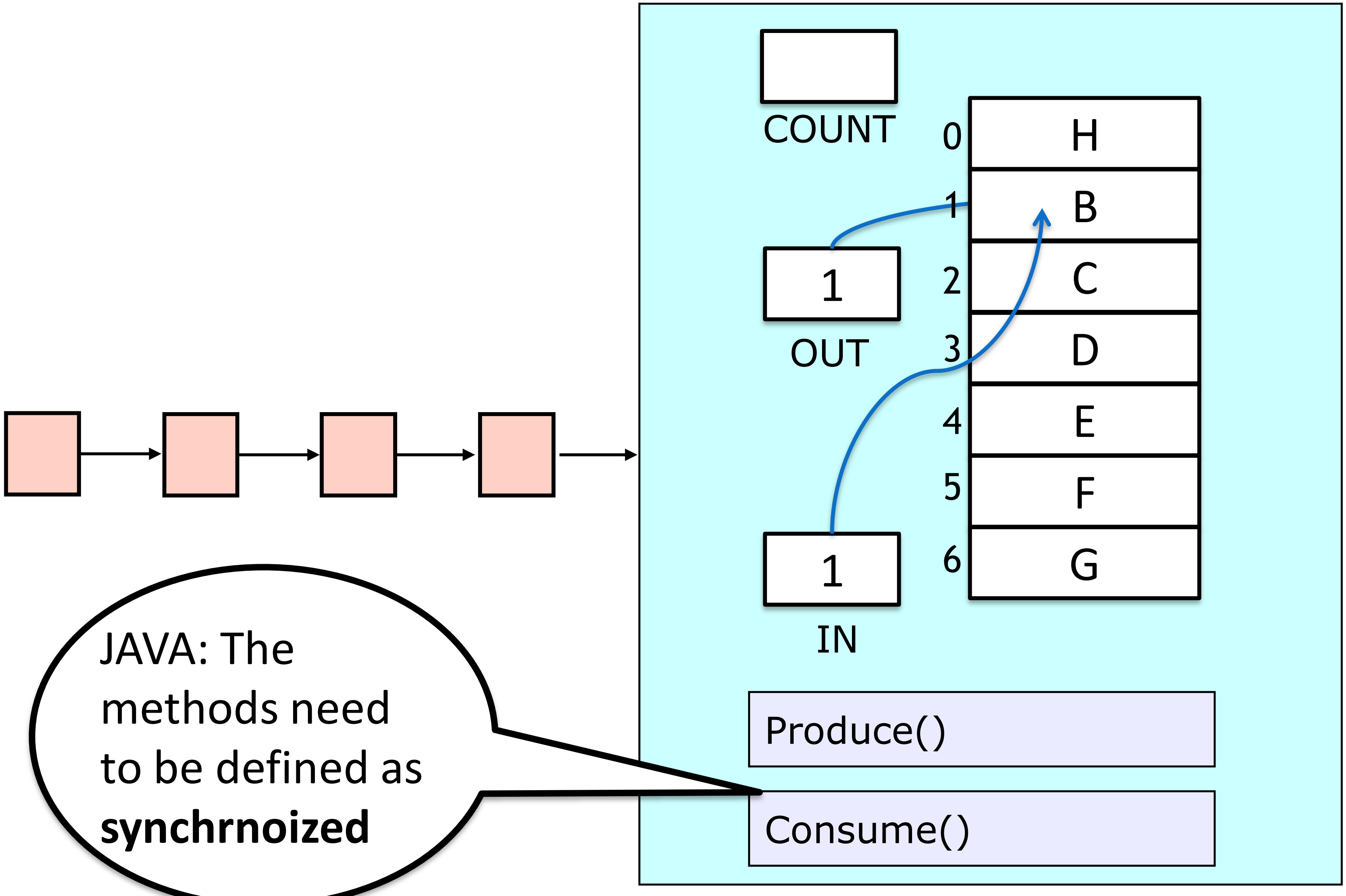
Monitor facilities

- "Automatic" mutual exclusion
  - only one thread can be executing inside at any time
    - thus, synchronization is implicitly associated with the monitor – it "comes for free"
  - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
    - more restrictive than semaphores
    - but easier to use (most of the time)
- But, there is a problem…

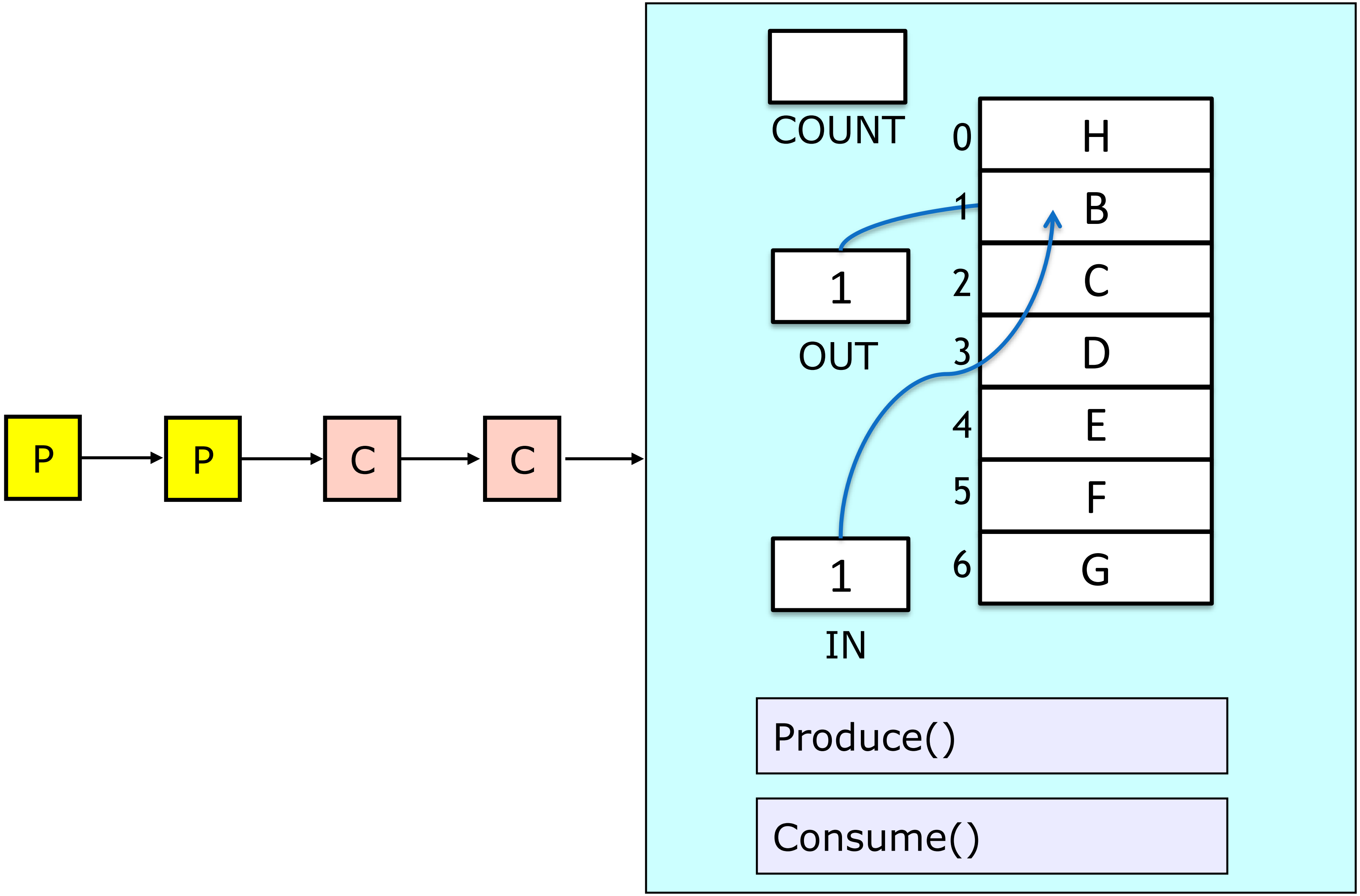
Producers/Consumers Monitor



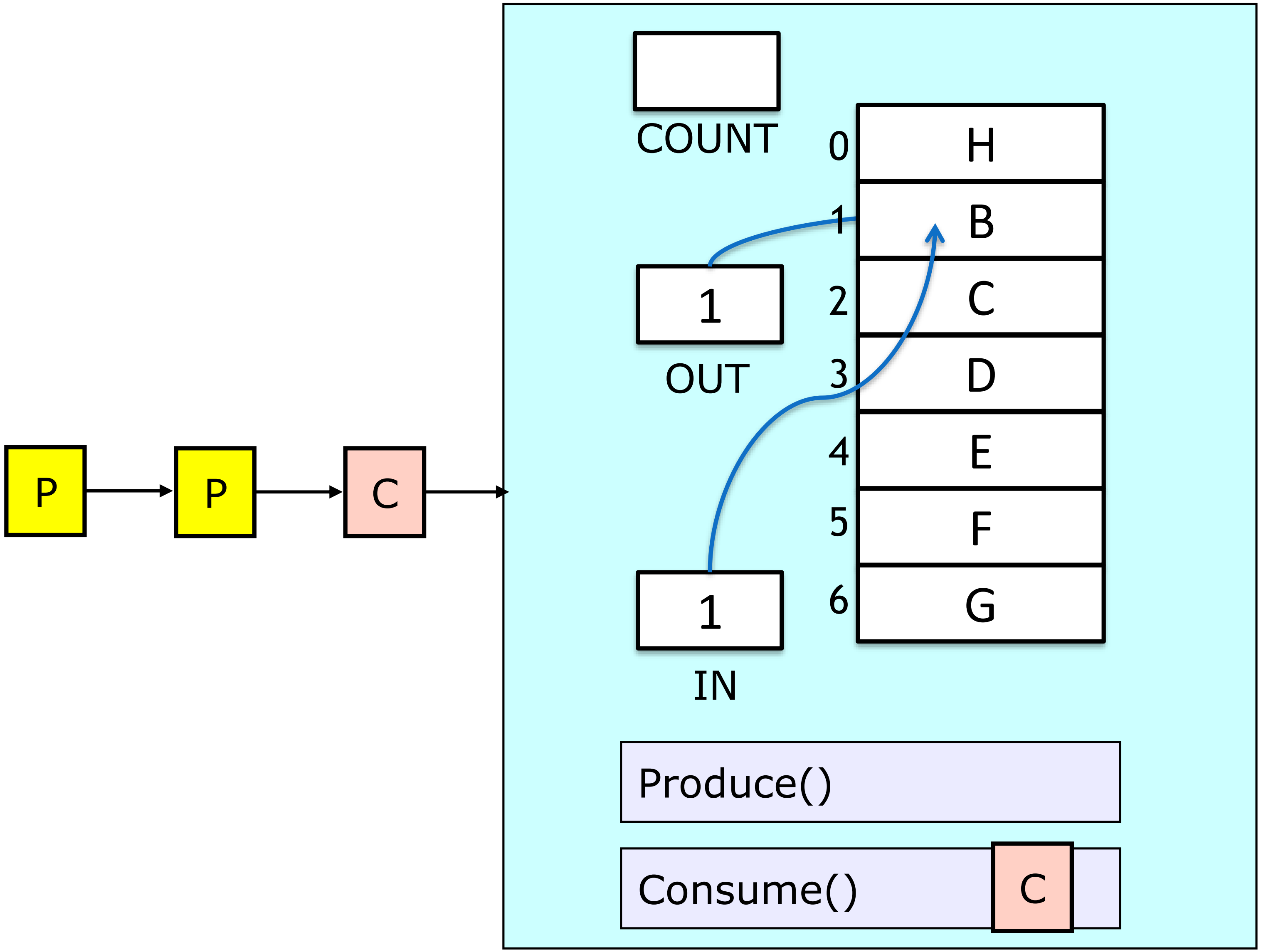
Producers/Consumers Monitor



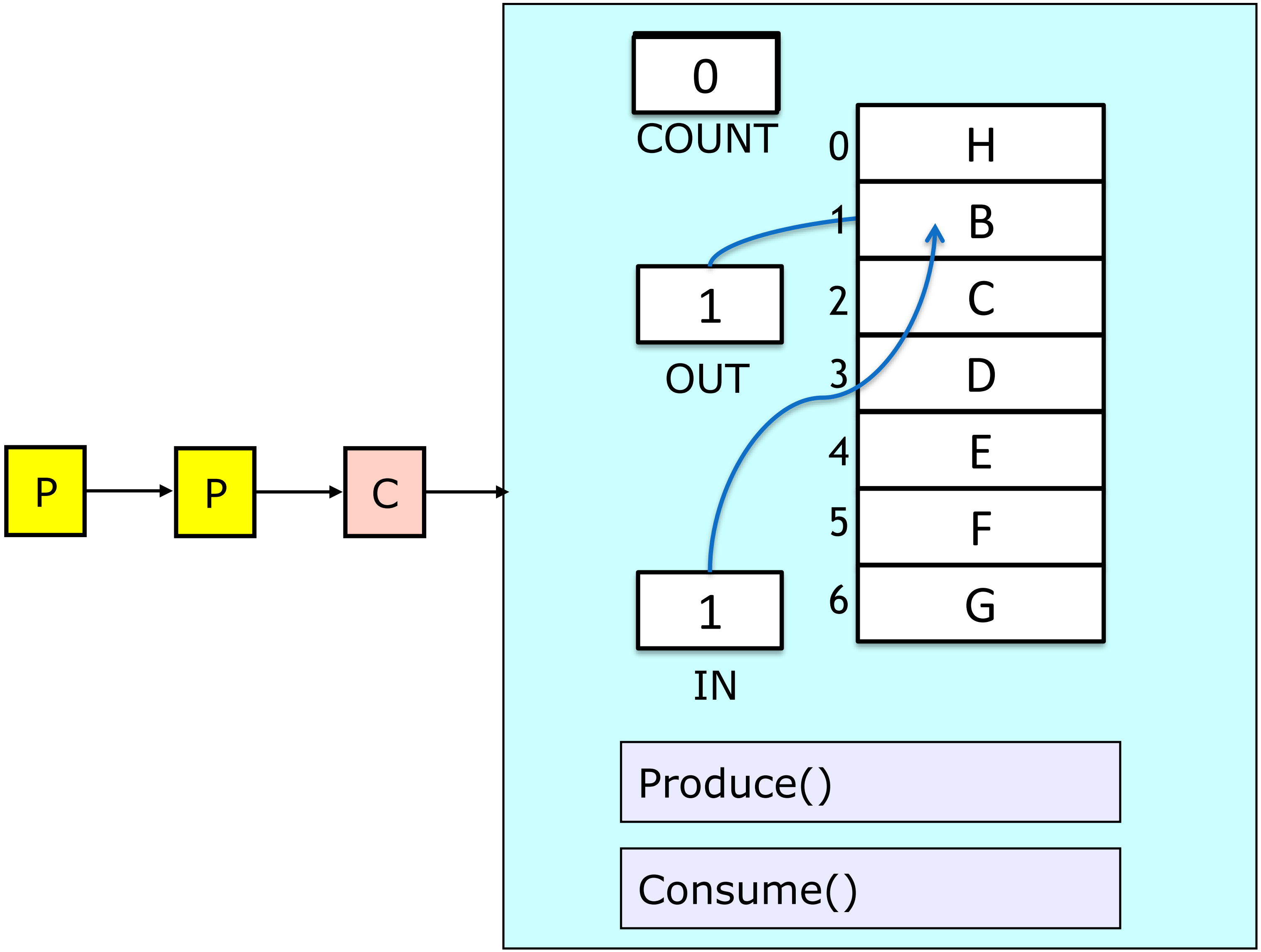
Producers/Consumers Monitor



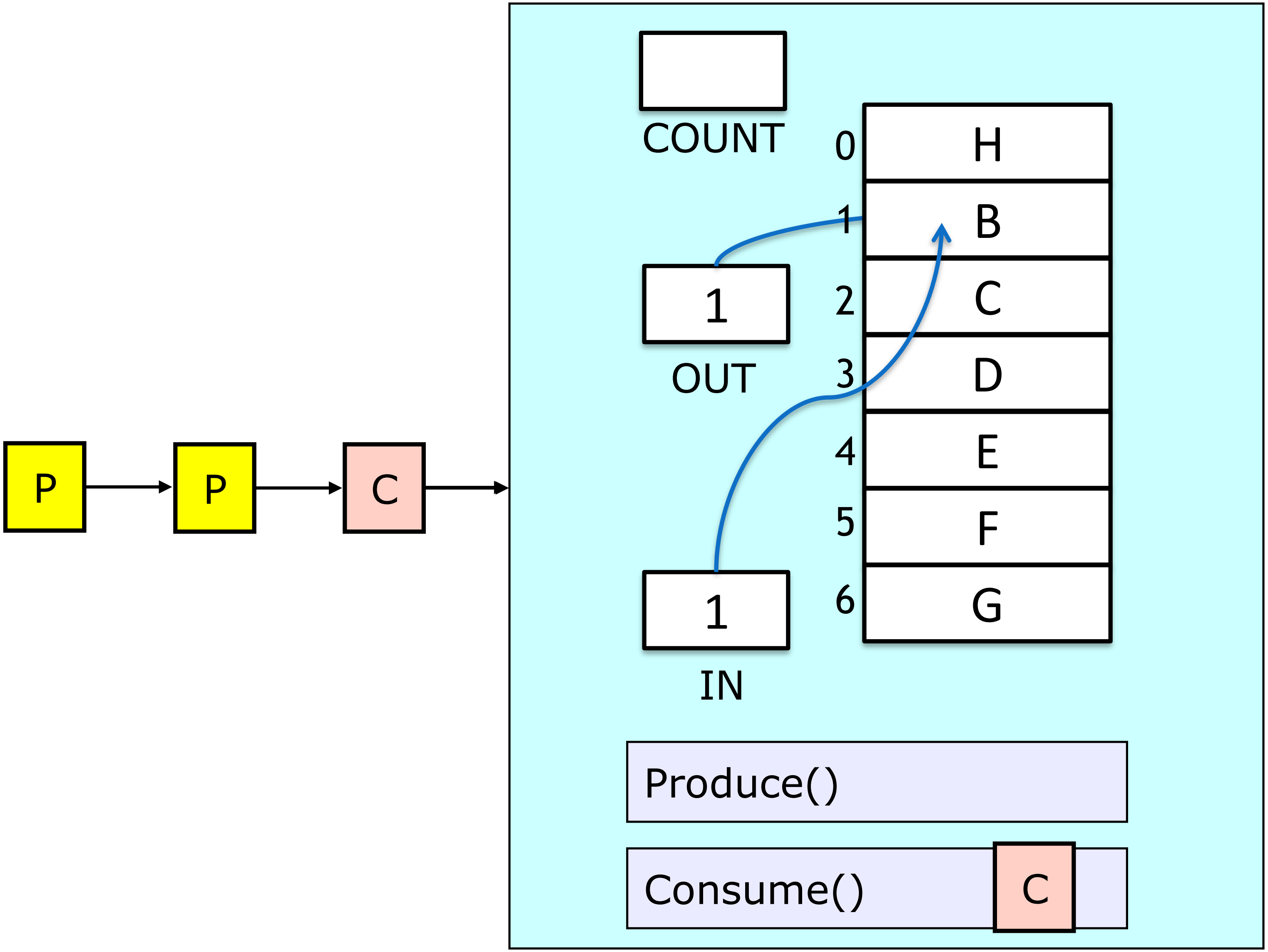
Producers/Consumers Monitor



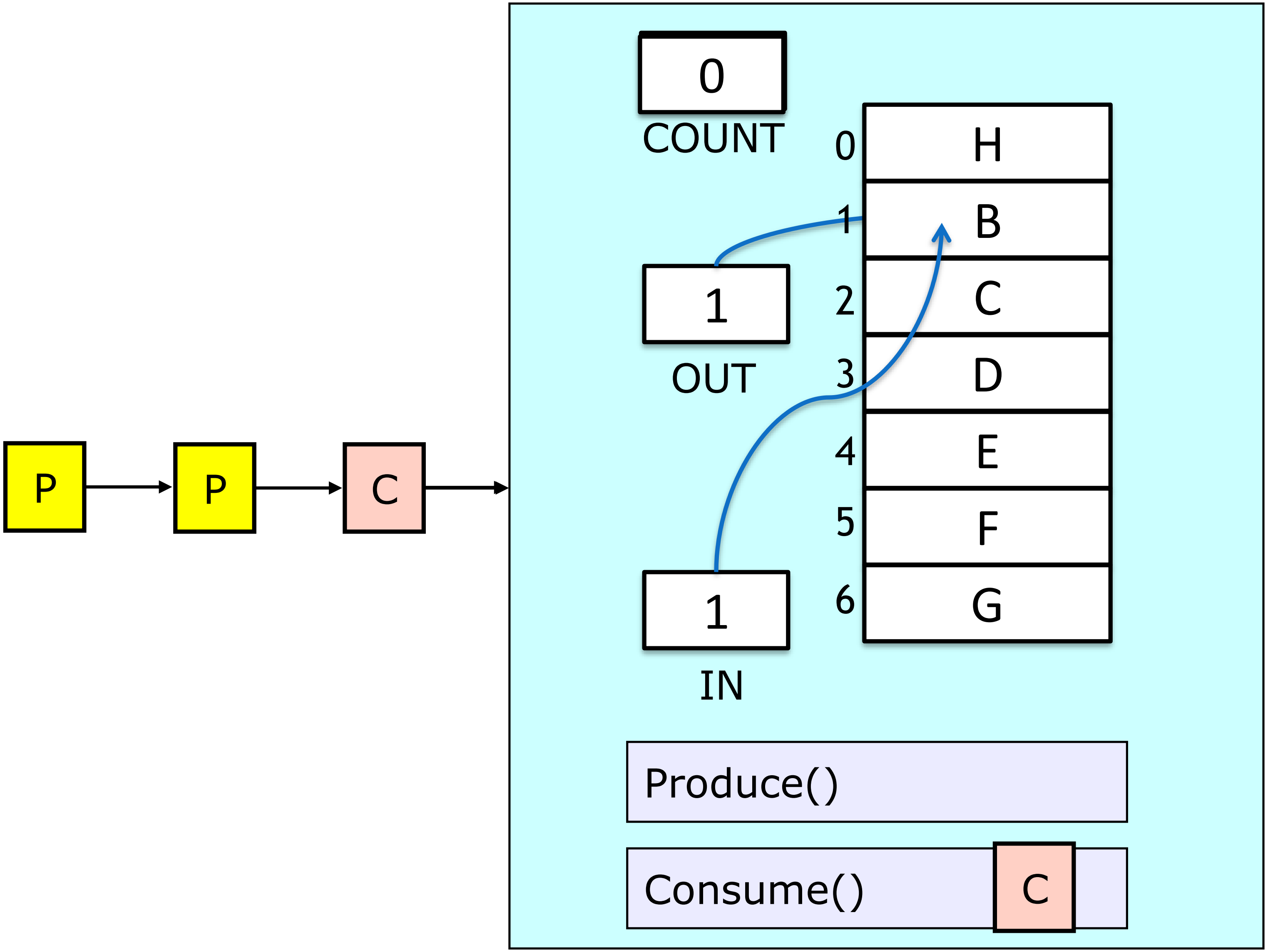
Producers/Consumers Monitor



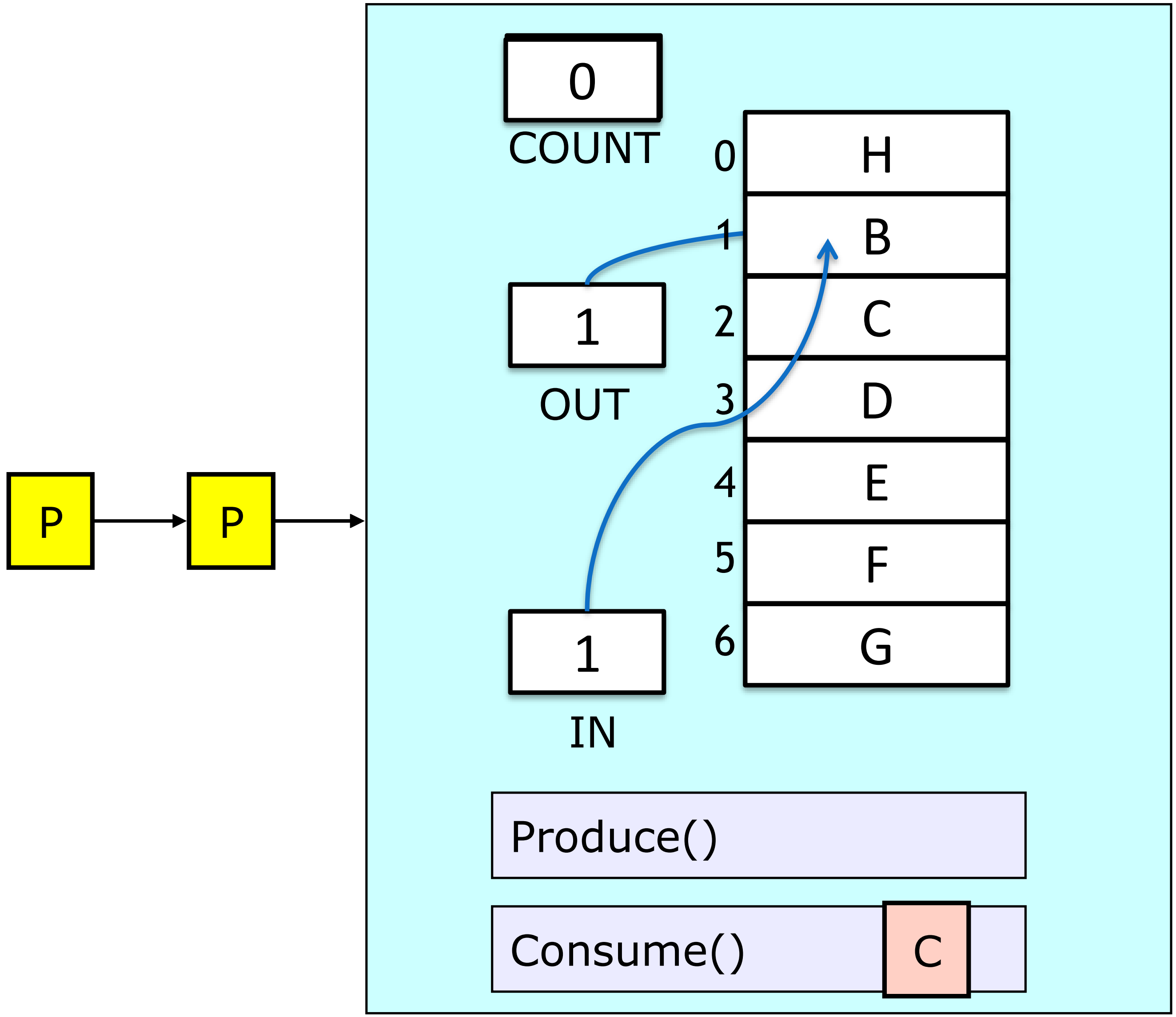
Producers/Consumers Monitor



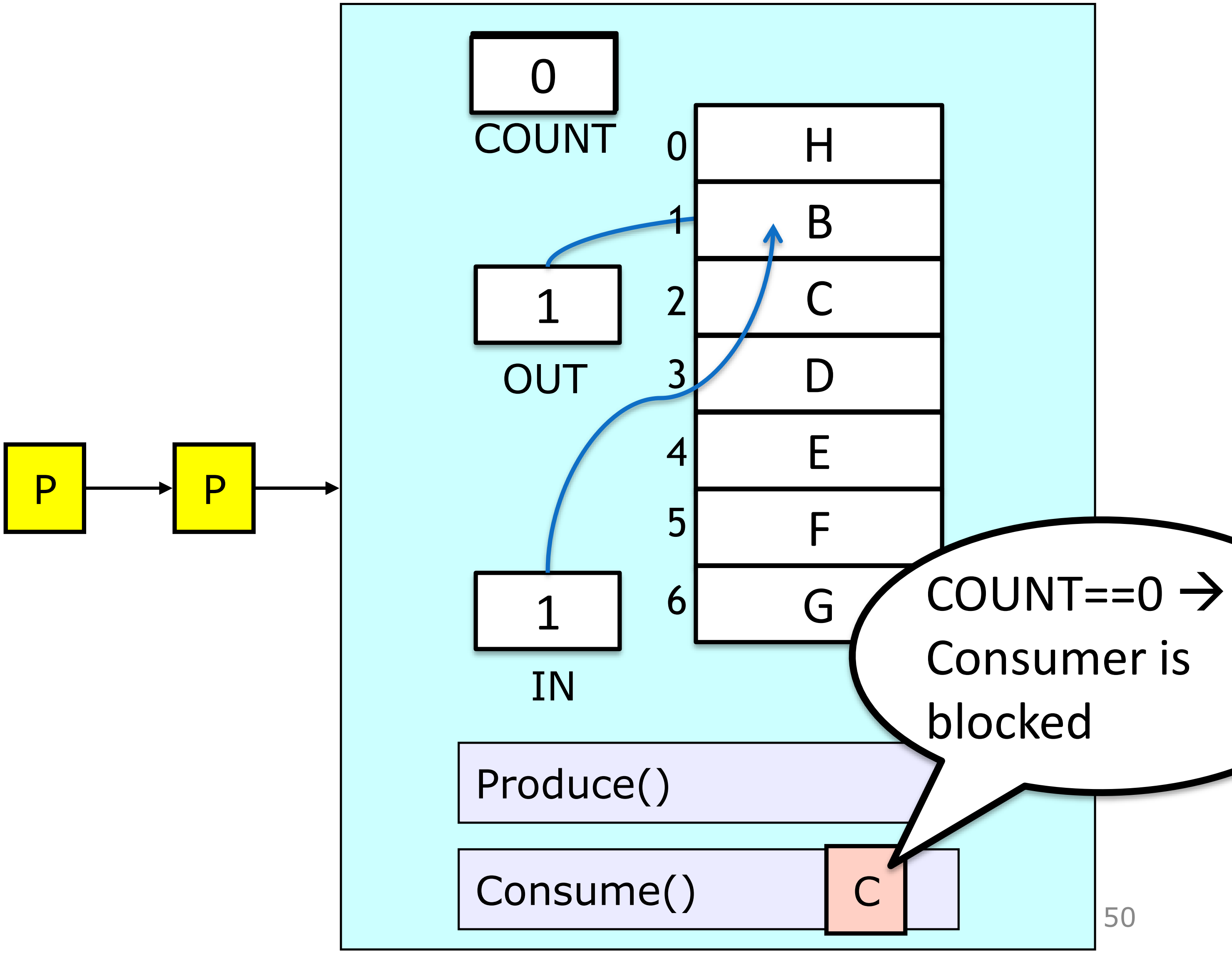
Producers/Consumers Monitor



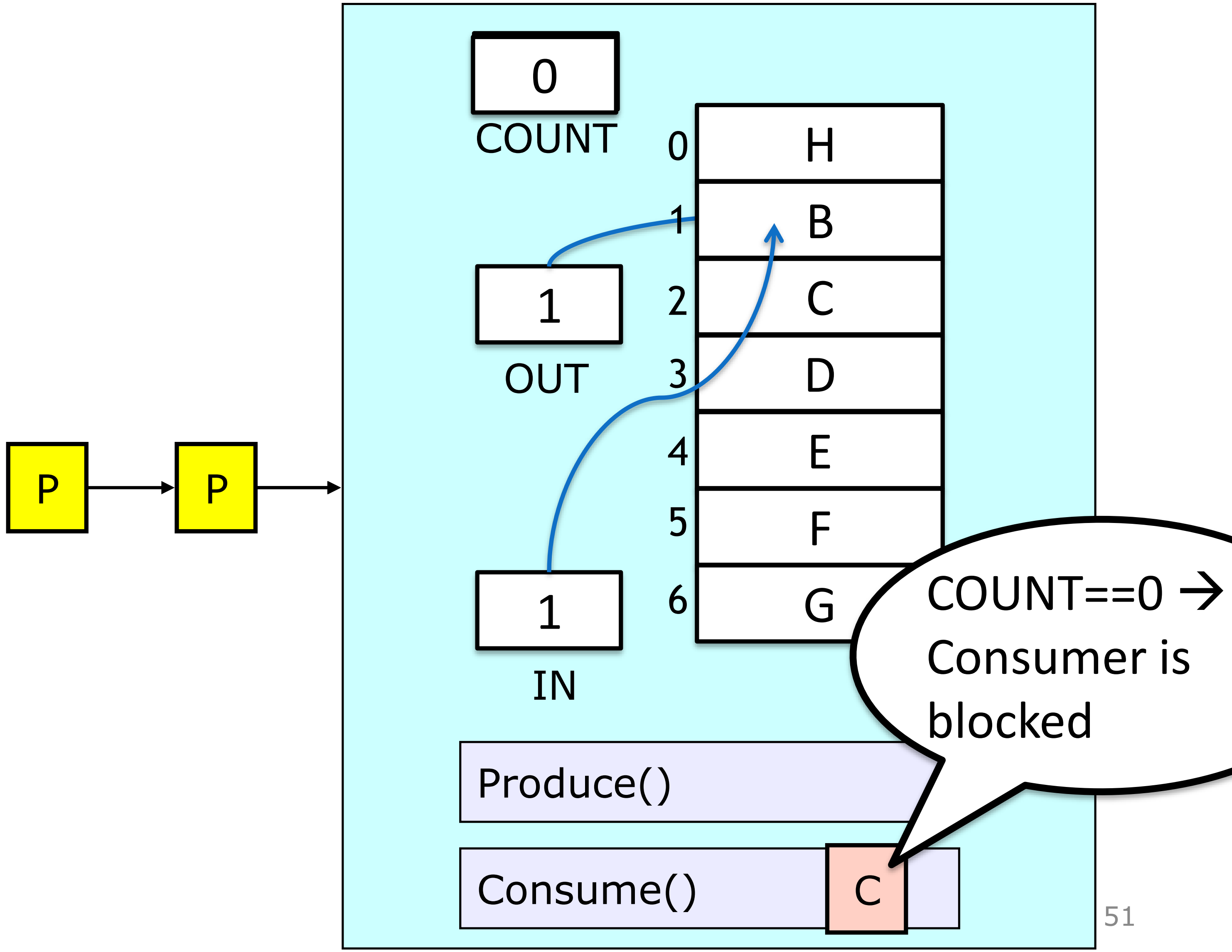
Producers/Consumers Monitor



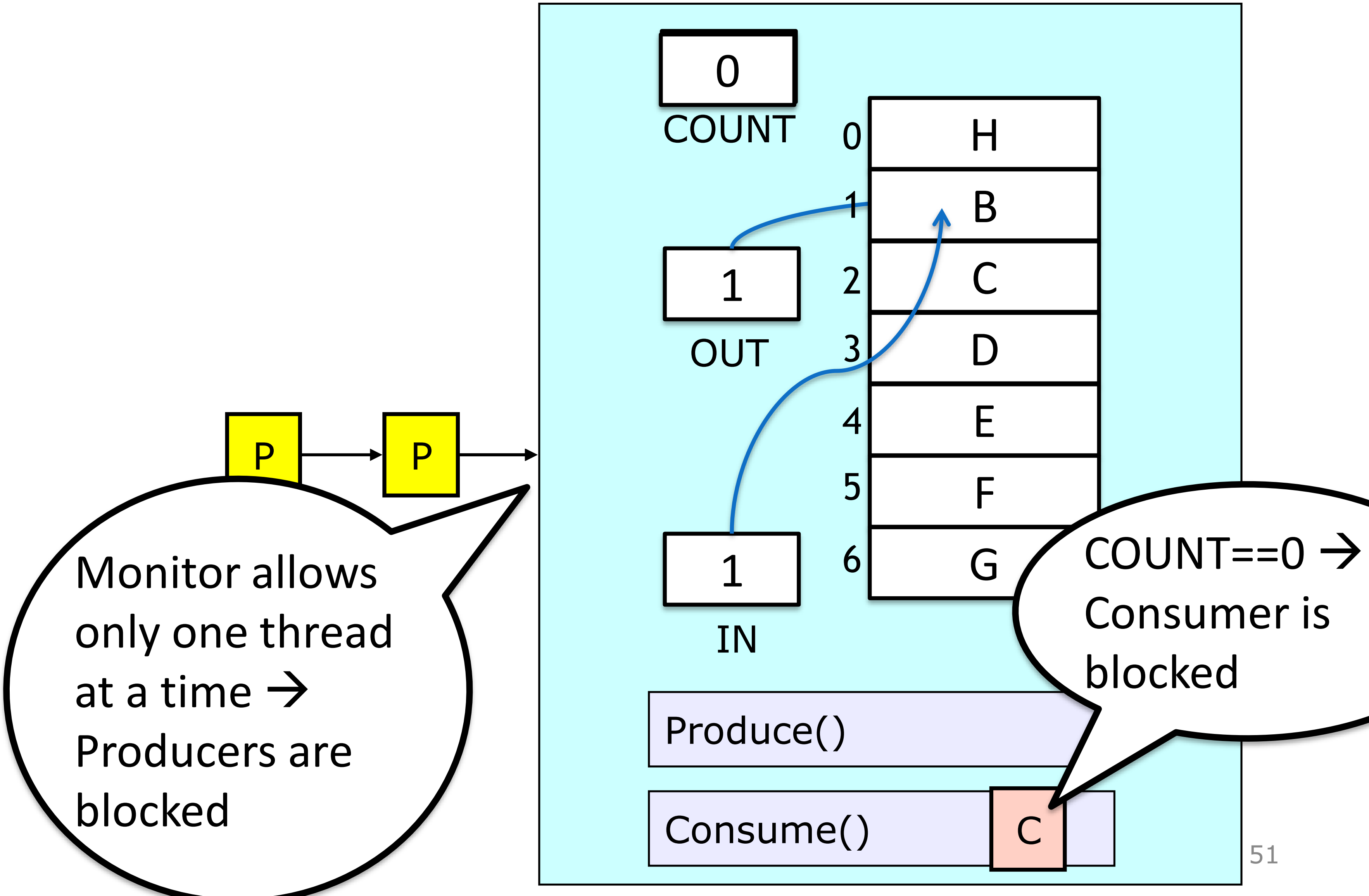
Producers/Consumers Monitor



Producers/Consumers Monitor



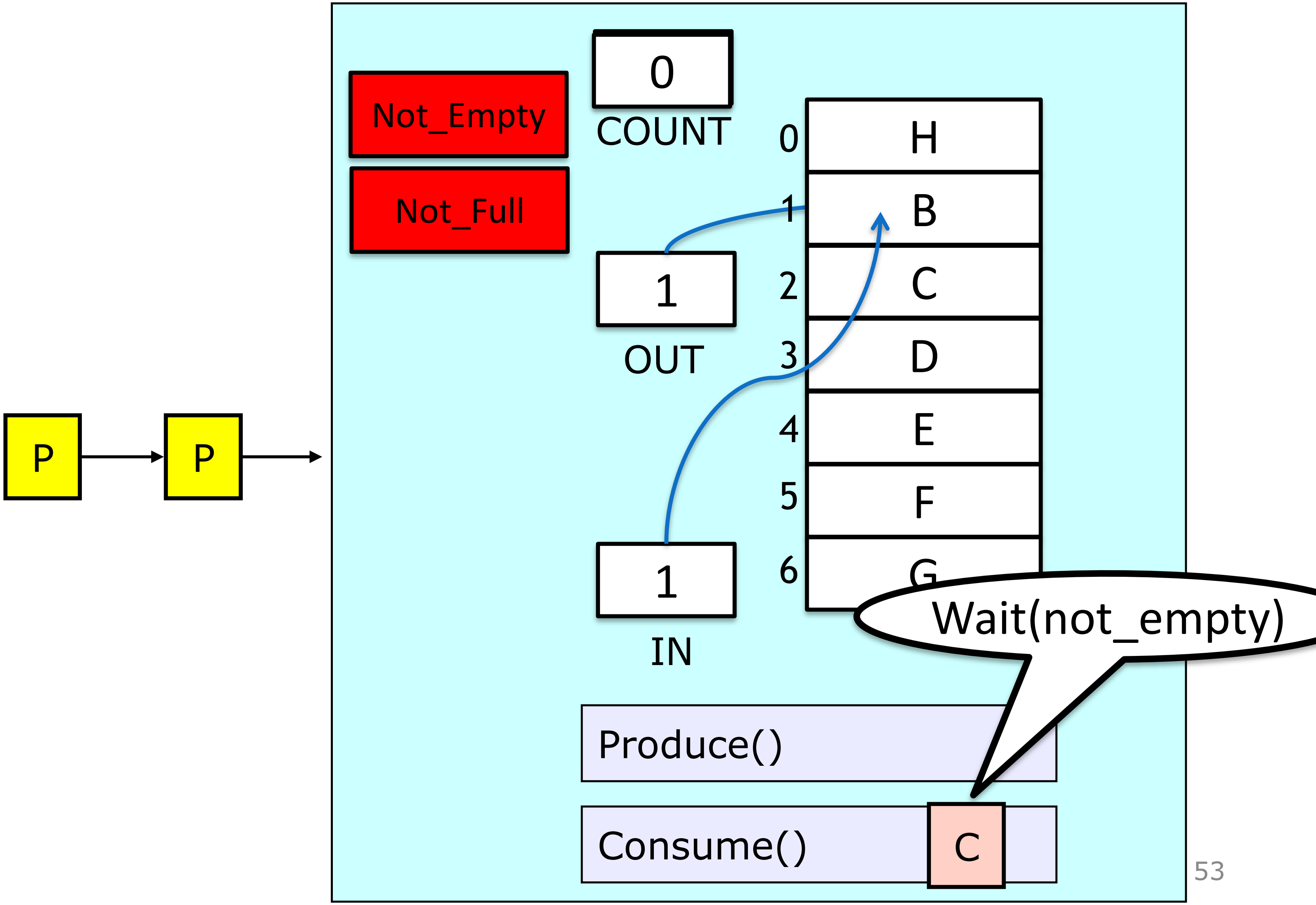
Producers/Consumers Monitor



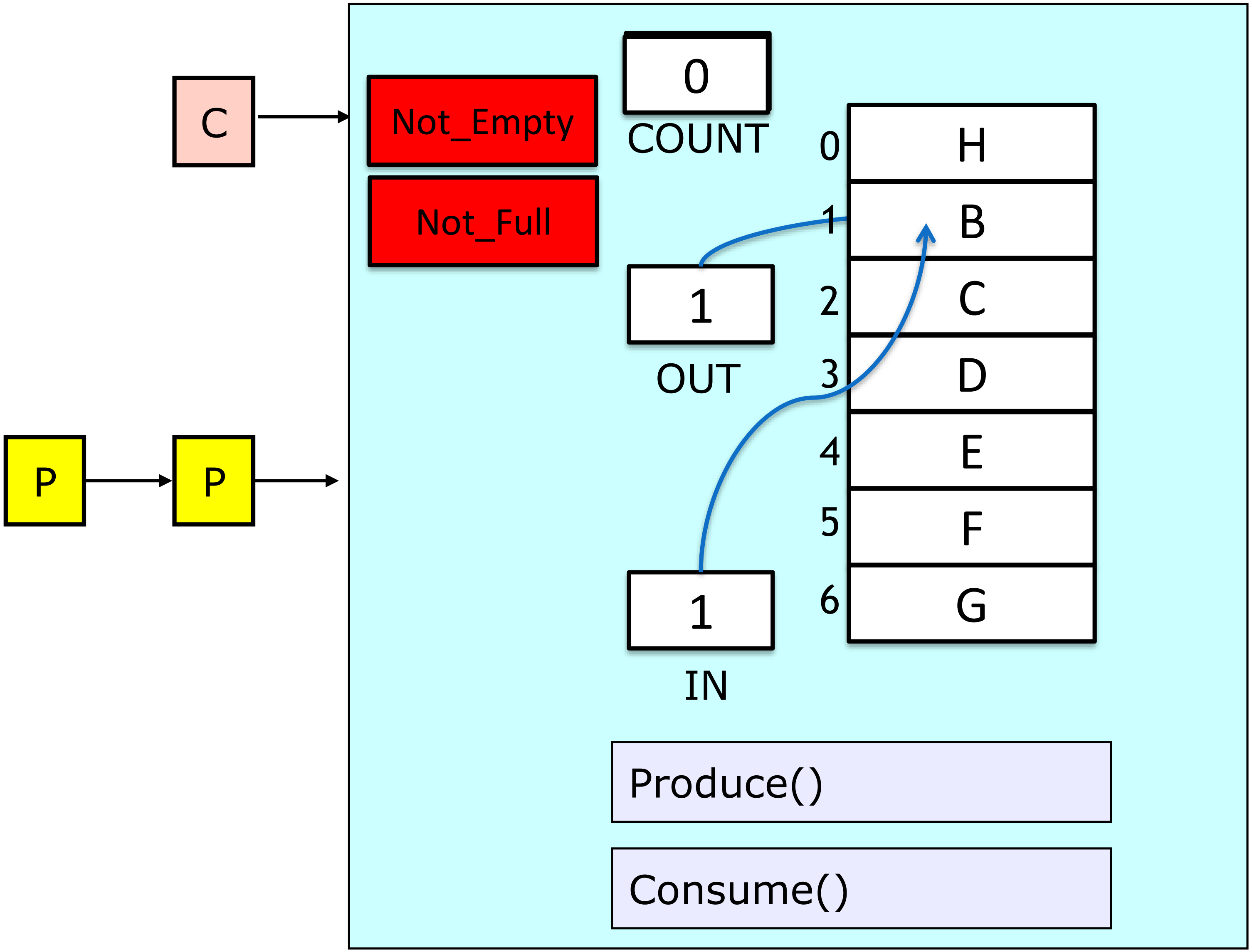
Condition variables (a.k.a. Rendezvous Points)

- “A place to wait”
- “Required” for monitors
  - So useful they’re often provided even when monitors aren’t available
- Three operations on condition variables
  - wait(c) – java: c.wait()
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have associated wait queues
  - signal(c) – java: c.notify()
    - wake up at most one waiting thread
    - if no waiting threads, signal is lost
      - this is different than semaphores: no history!
  - broadcast(c) – java: c.NofityAll()
    - wake up all waiting threads

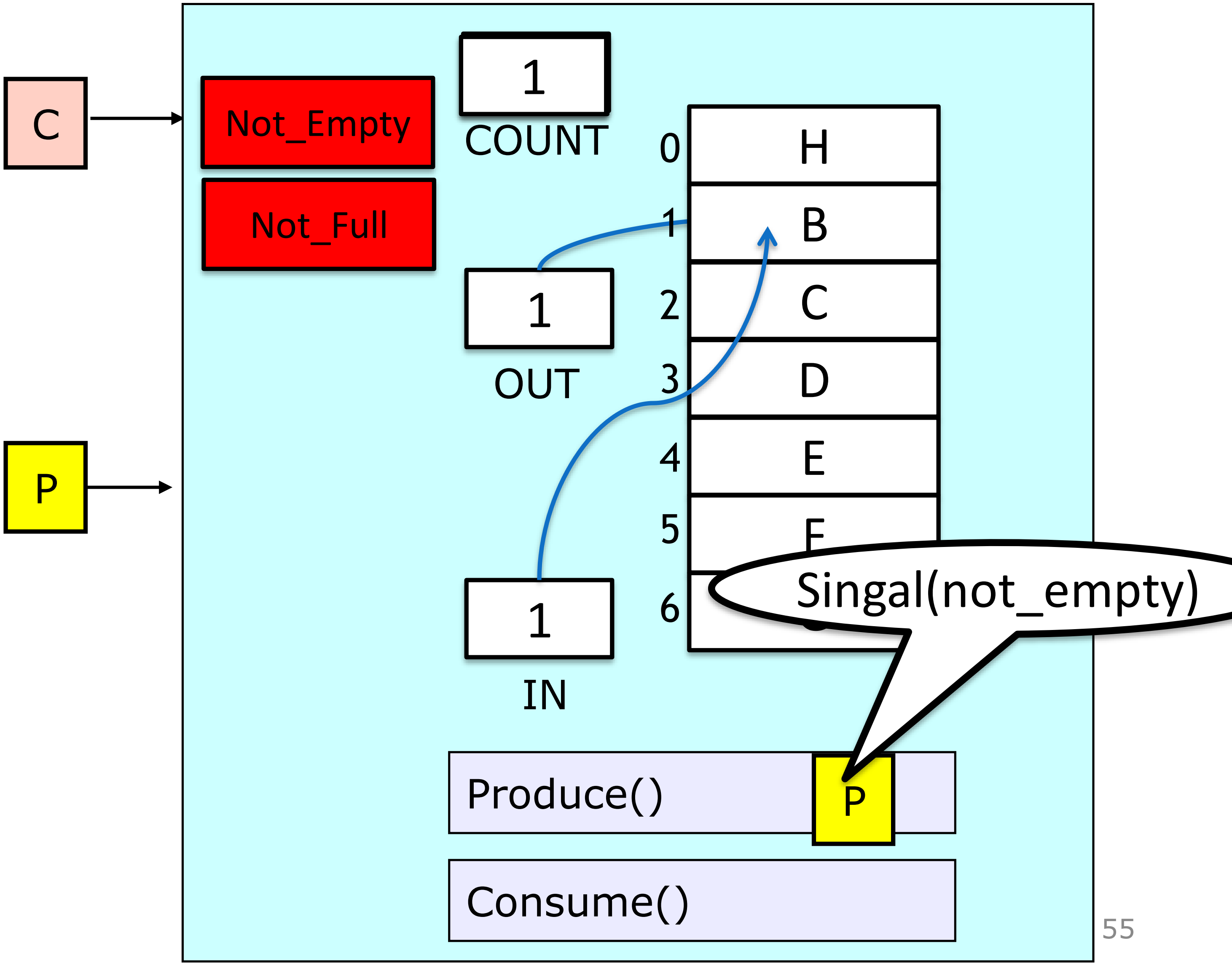
Producers/Consumers Monitor



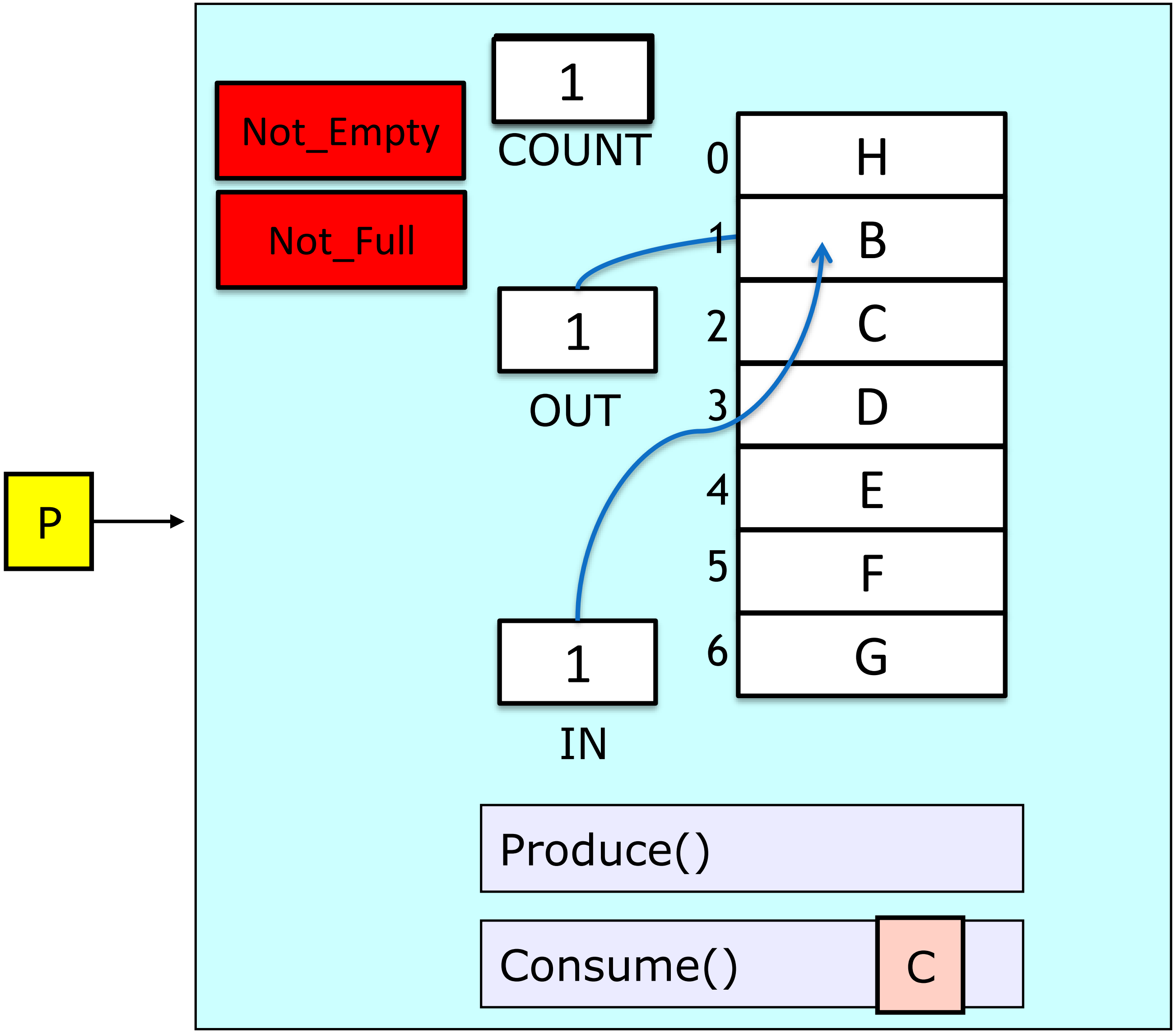
Producers/Consumers Monitor



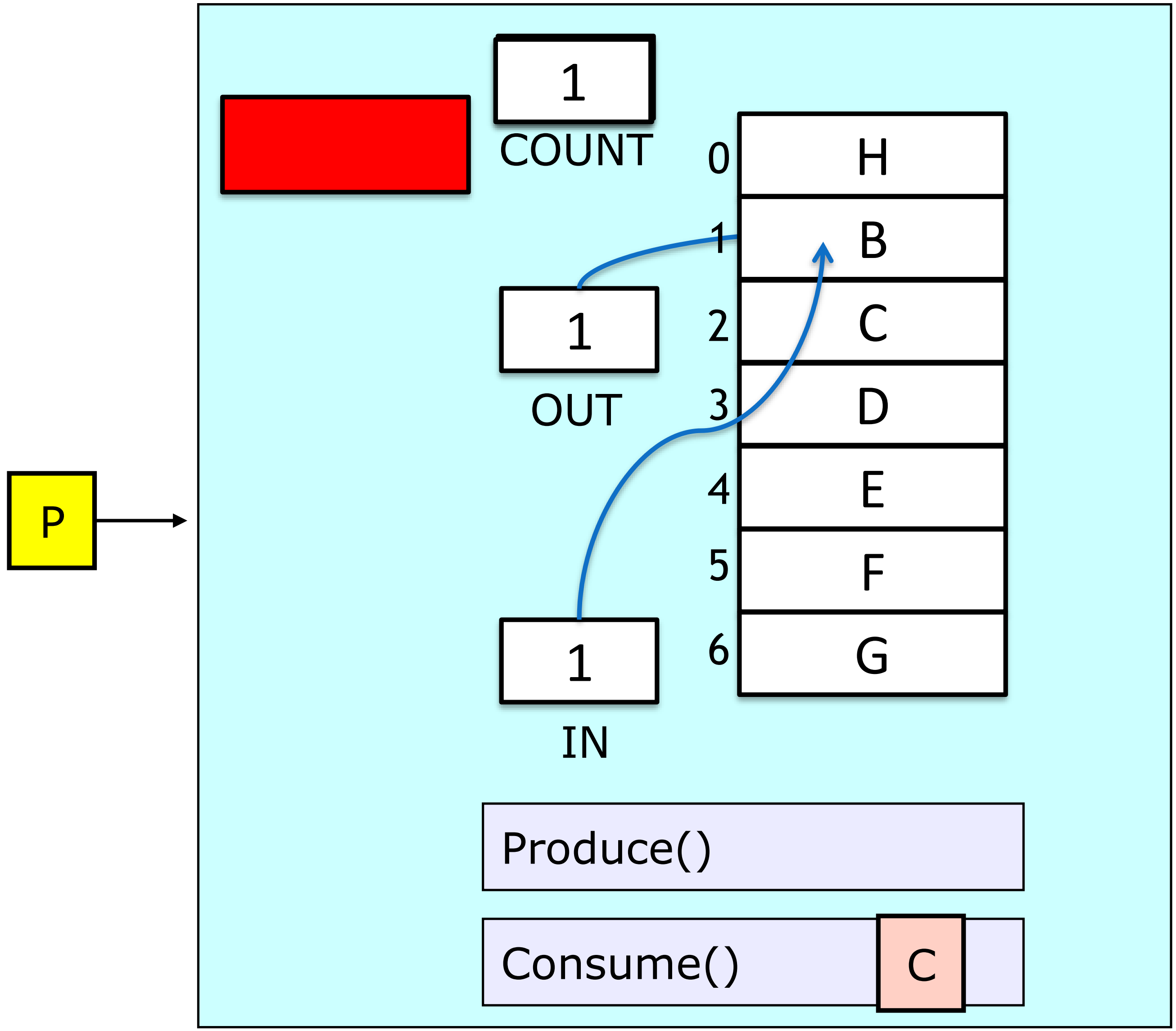
Producers/Consumers Monitor



Producers/Consumers Monitor



Producers/Consumers Monitor



Shared Memory vs. Message Passing Models

- In the course we assume two threads or two processes are communicating through **shared resources** – mostly shared memory
- Another way to communicate: send and receive **messages**
- Similar problems arise, but the solutions are quite different
- More about it in the “Distributed Algorithms” course