University of Pittsburgh

Introduction to Operating Systems
CS 1550

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Some slides are from **Silberschatz, Galvin and Gagne ©2013)**

Announcements

- Upcoming deadlines
  - Homework 2 is due **next Monday** at 11:59 pm
  - Lab 1 is due on Tuesday 2/7 at 11:59 pm
  - Project 1 is due on Friday 2/17 at 11:59 pm
    - Discussed in this week's recitations
- AFS Quota
  - You can check it using the command **fs quota**
  - You can increase it from accounts.pitt.edu.
    - Check README of Lab 1
- VS Code issue
  - Turn off the usage of **flock** to lock files
  - Check my Piazza reply in **Thoth Password** thread

Previous Lecture …

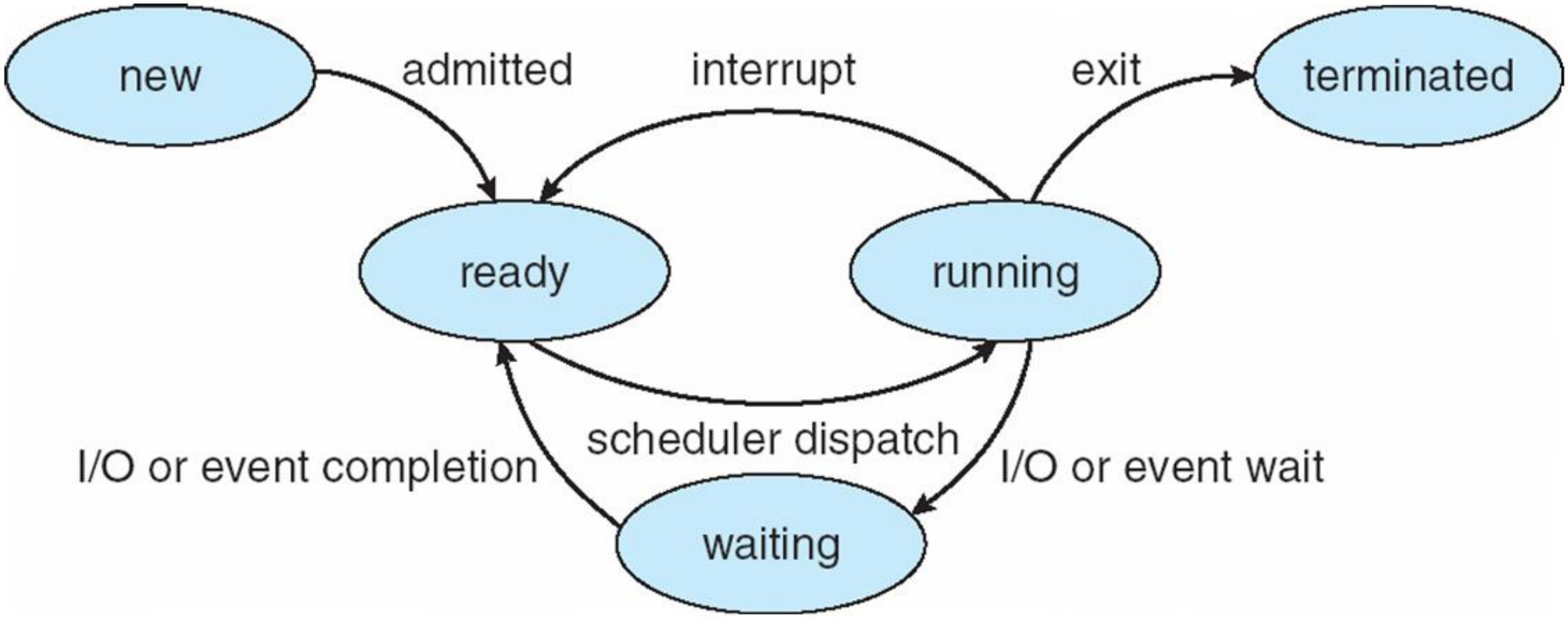
Three usage **problems** of Semaphores

- compromising mutual exclusion
  - **Solution: Mutex**
- deadlock
  - **Solution: Not yet discussed**
- priority inversion
  - **Solution: priority inheritance**

Question

How are processes created, maintained, and terminated?

Process Lifecycle (AKA Process States)



Process Creation

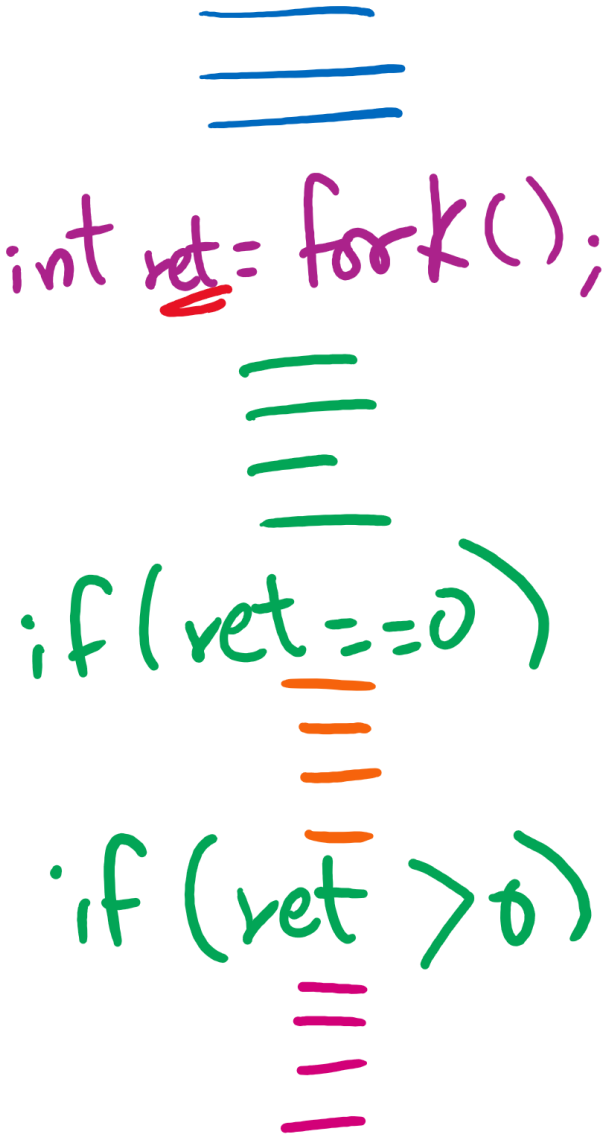
- Via fork() syscall
- Parent process: the process that calls fork()
- Child process: the process that gets created
- Child process has a new context
  - new PCB

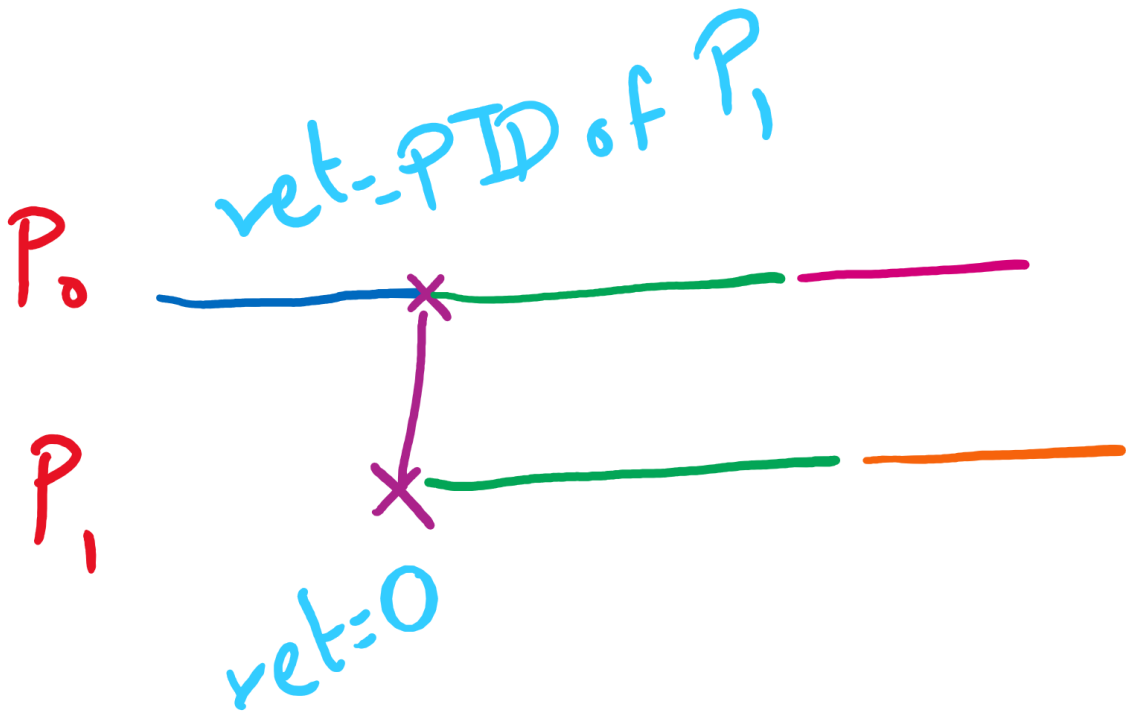
Process Creation

Memory of parent process copied to child process

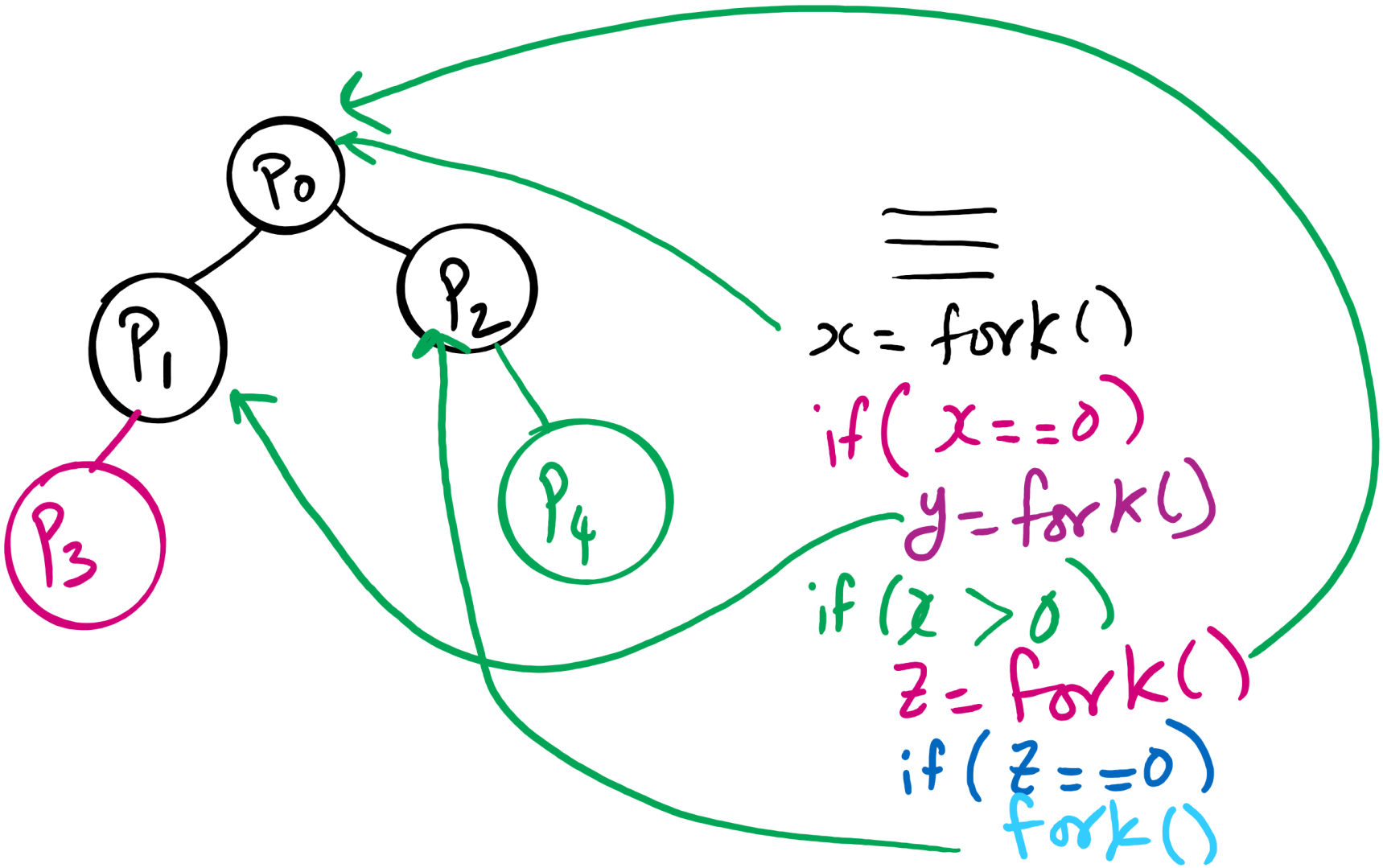
- Too much copying
- Even not necessary sometimes
  - e.g., fork() followed by exec() to run a different program
- Optimization trick:
  - **copy-on-write**
  - copy when any of the two processes writes into its memory
  - copy the affected memory “part” only
  - How would the OS know when a process writes to its memory?

fork() tracing

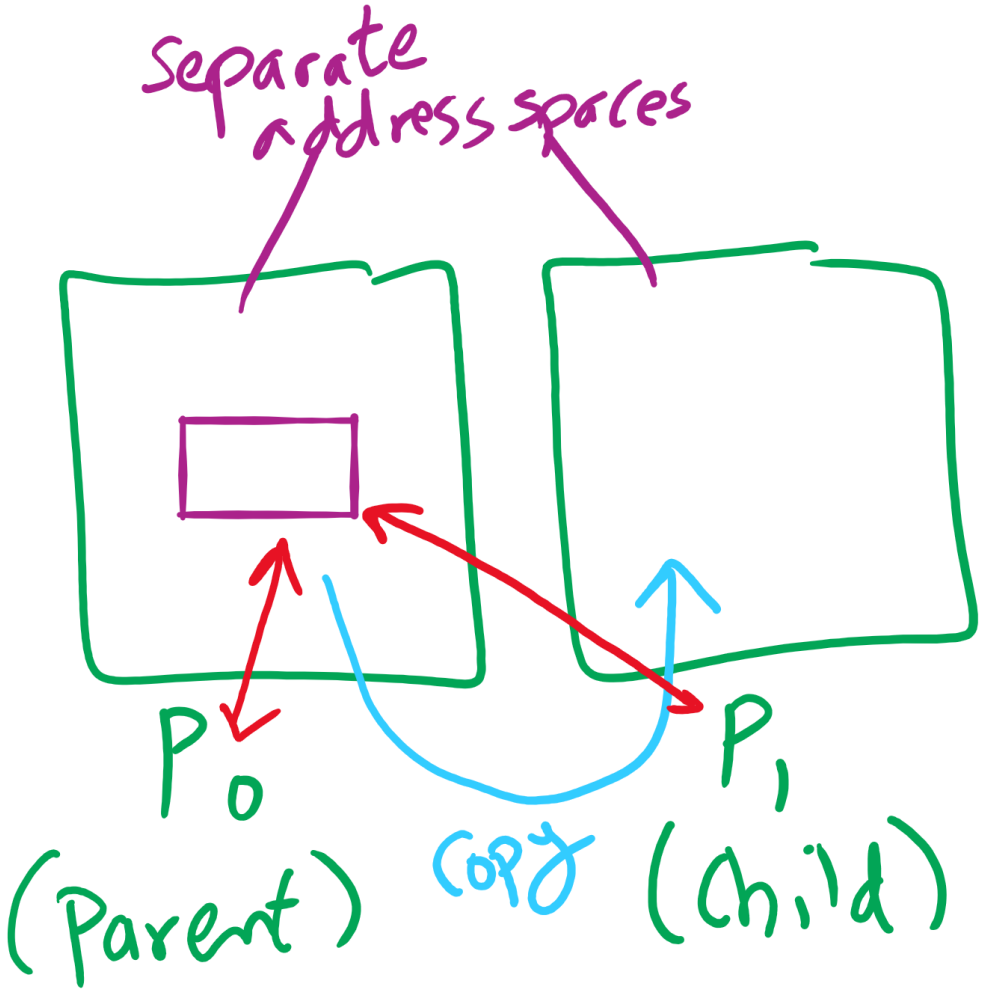


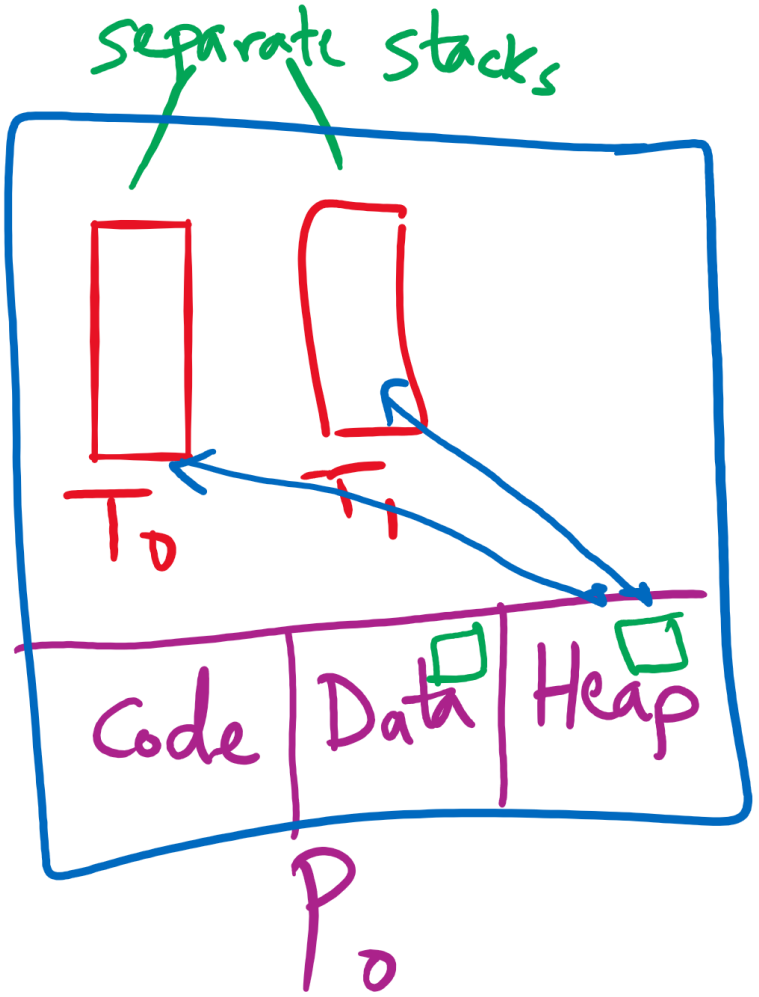


fork()'s of fork()'s



Process vs. Thread





fork() example

```c
int main(){
  int a, b, c, x, y, z;
  printf("Start\n");
  x = fork();
  y = fork();
  if(x>0){
    z = fork();

  } else {
    a = fork();
  }
  if(z > 0 && a ==0){
   b = fork();
  }
  fork();
  printf("End\n");
  return 0;
}
```

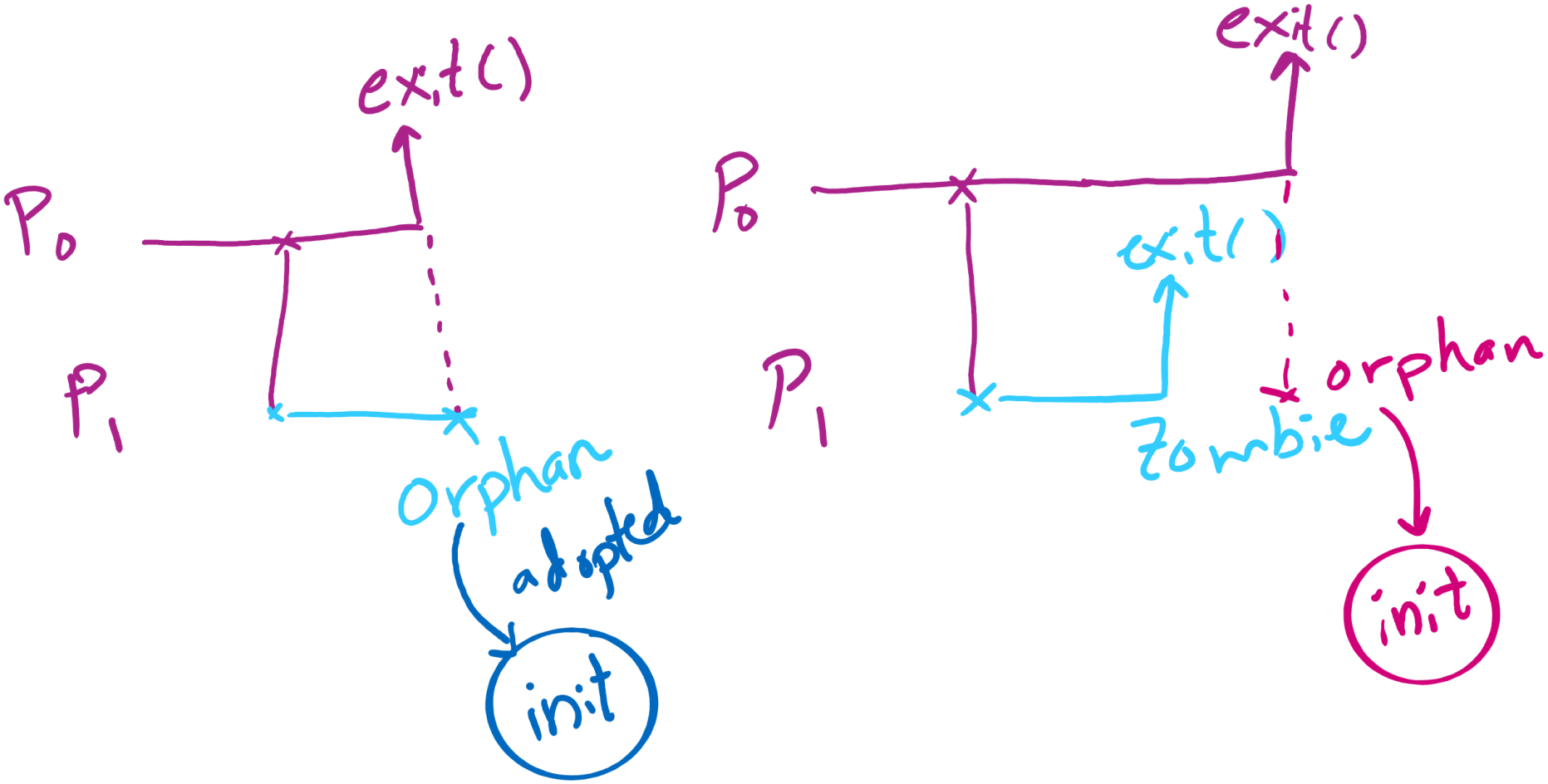
Process Termination

- Via **`exit`**(), **`abort`**(), or **`kill()`** syscalls
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- When a process terminates
  - If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
  - If parent terminated without invoking **`wait`**, process is an **orphan**
    - adopted by the **`init`** process

Orphan vs. Zombie Processes



Benefits of Orphan Processes

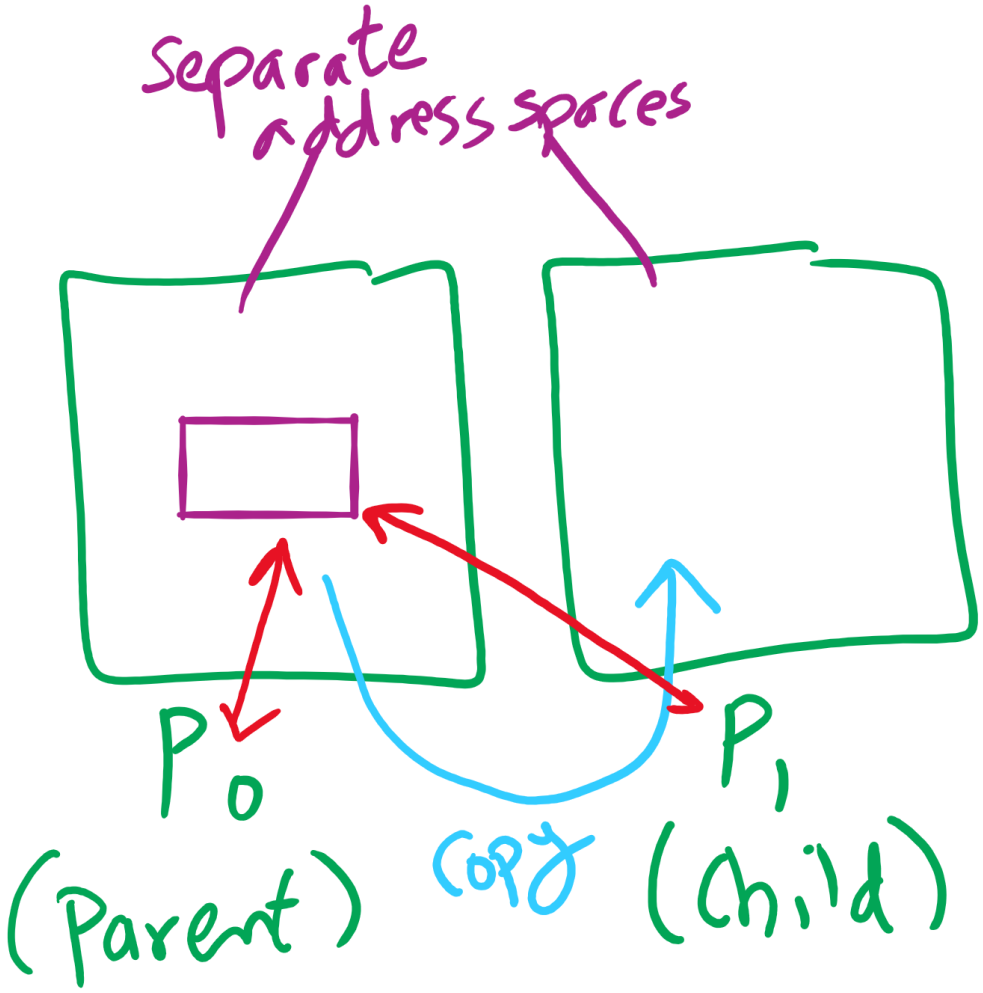
- Allow a long-running job to continue running even after session (e.g., ssh connection) ends.
  - The nohup command does that
- Create daemon processes
  - Long-running background processes adopted by the init process.

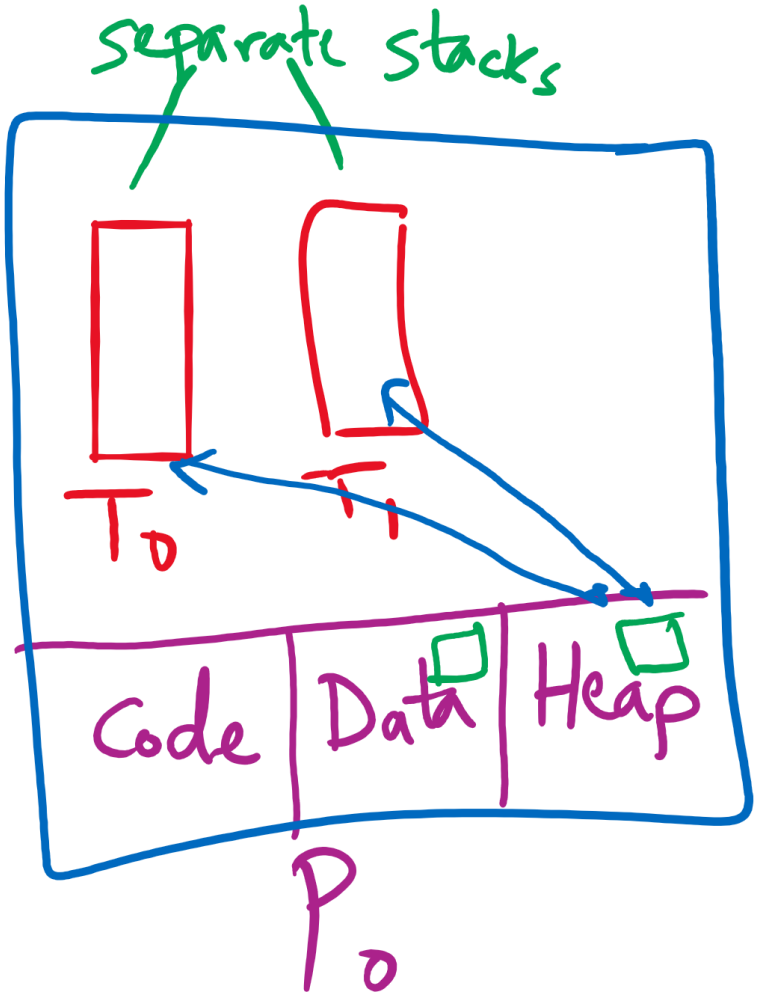
Thread Synchronization

Synchronization issues apply to *threads* as well

- Threads can share data easily (same address space)
- Other two issues apply to threads

Process vs. Thread

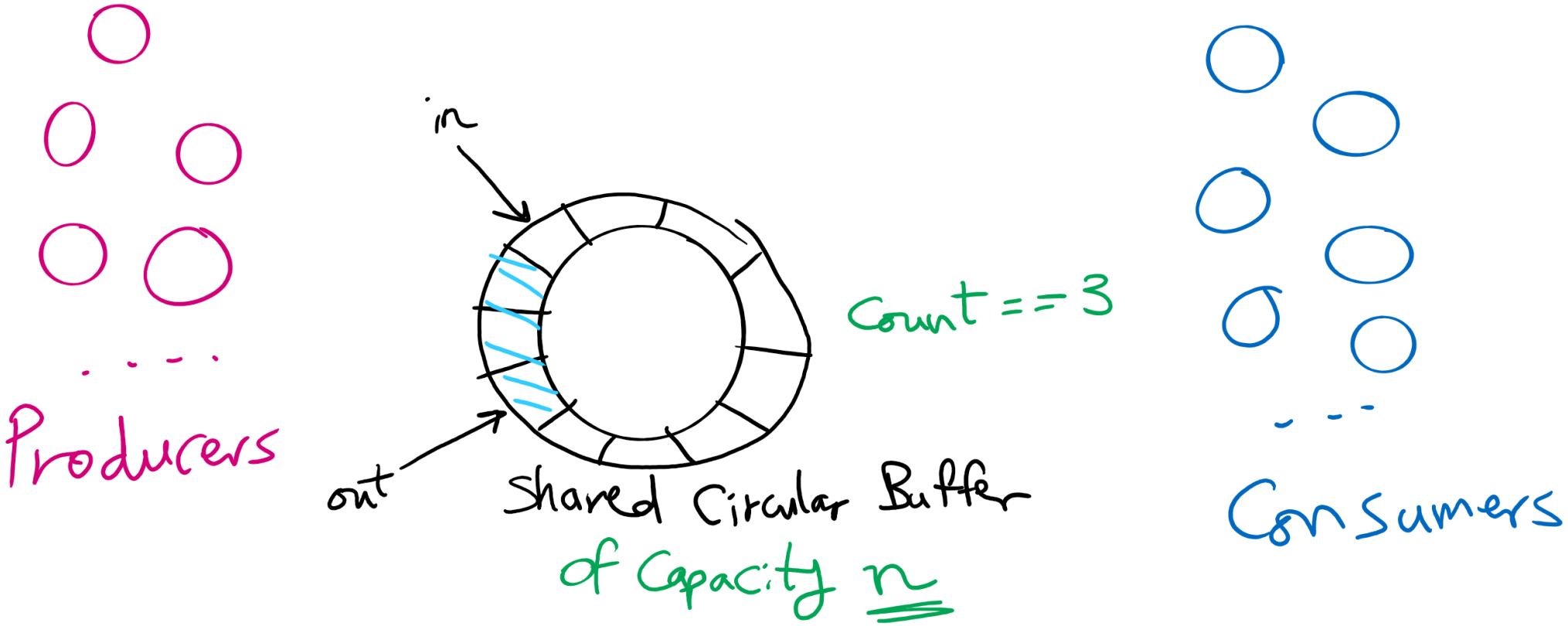




Bounded Buffer Problem

- Bounded Buffer Problem
  - aka Producers Consumers Problem
- A shared circular-array buffer with capacity *n*
- A set of *producer* processes/threads
- As set of *consumer* processes/threads
- Requirements:
  - Never exceed the buffer capacity
  - Producers wait if the buffer is full
  - Consumers wait if the buffer is empty

Produces Consumers Problem

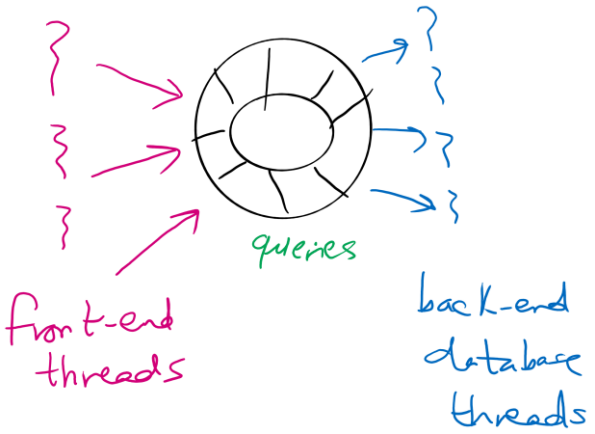


Producers Consumers Problem is everywhere!

- Access to User Interface elements in mobile applications
  - Main UI thread
  - Background threads
    - e.g., download files, check email on mail server
  - Background threads send requests to UI thread to update UI elements
  - Requests are stored in a shared bounded buffer
  - Which threads are the producers?
  - Who threads are the consumers?

Producers Consumers Problem is everywhere!

- Web Server
  - front-end processes/threads that interact with the HTTP connection from the client (e.g., browser)
  - Back-end processes/threads that execute database queries
  - Queries are inserted by front-end processes into a shared buffer
  - Which threads are the producers?
  - Who threads are the consumers?



Solving Producers Consumers using Semaphores

```
Semaphore empty(n), full(0)
Mutex Sem(1);
```

Producer

```
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
Count++
up(sem)
up(full)
```

Consumer

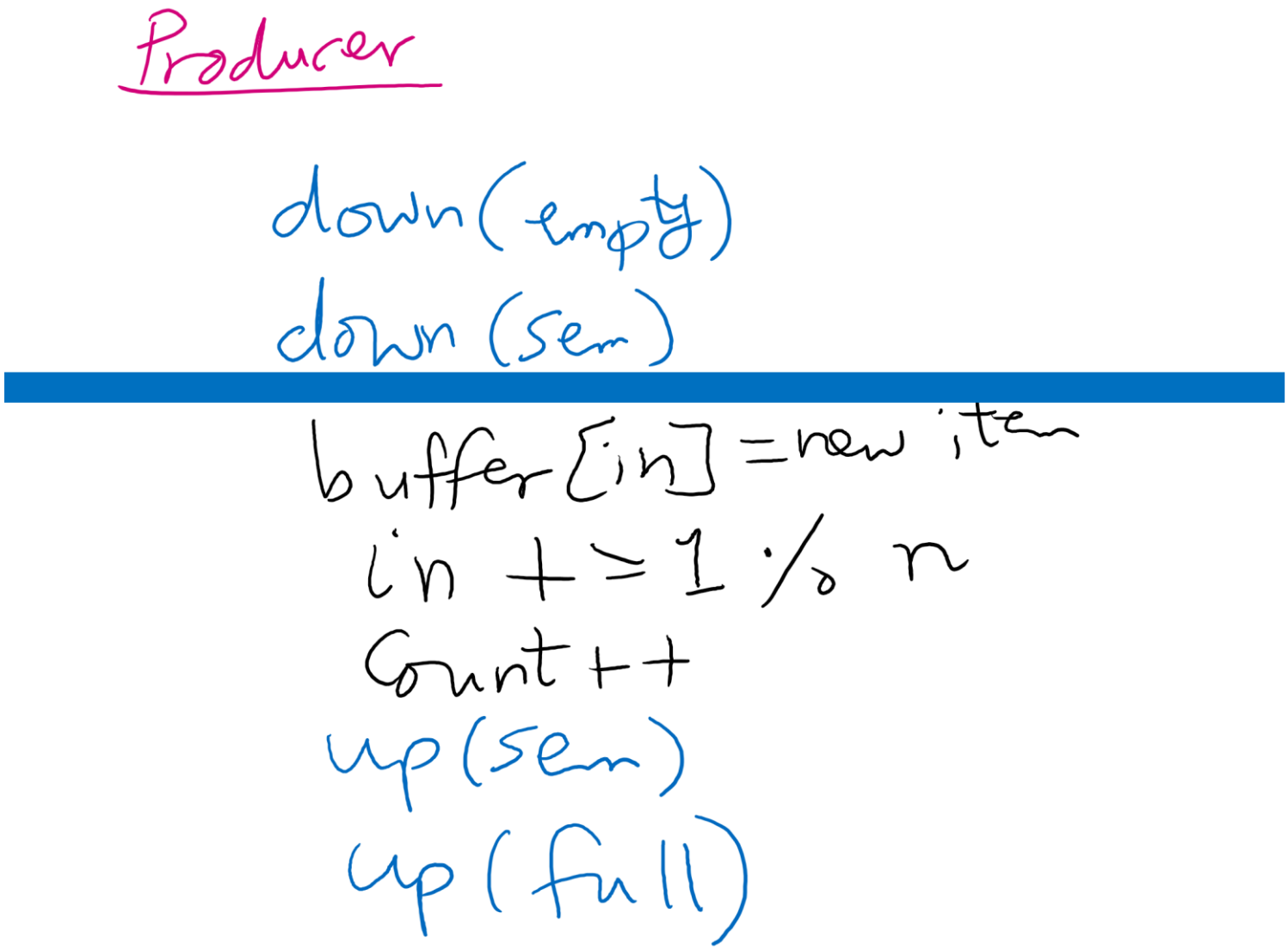
```
down(full)
down(sem)
item = buffer[out]
out += 1 % n
Count--
up(sem)
up(empty)
```

How to trace the solution?

Let’s define some events.

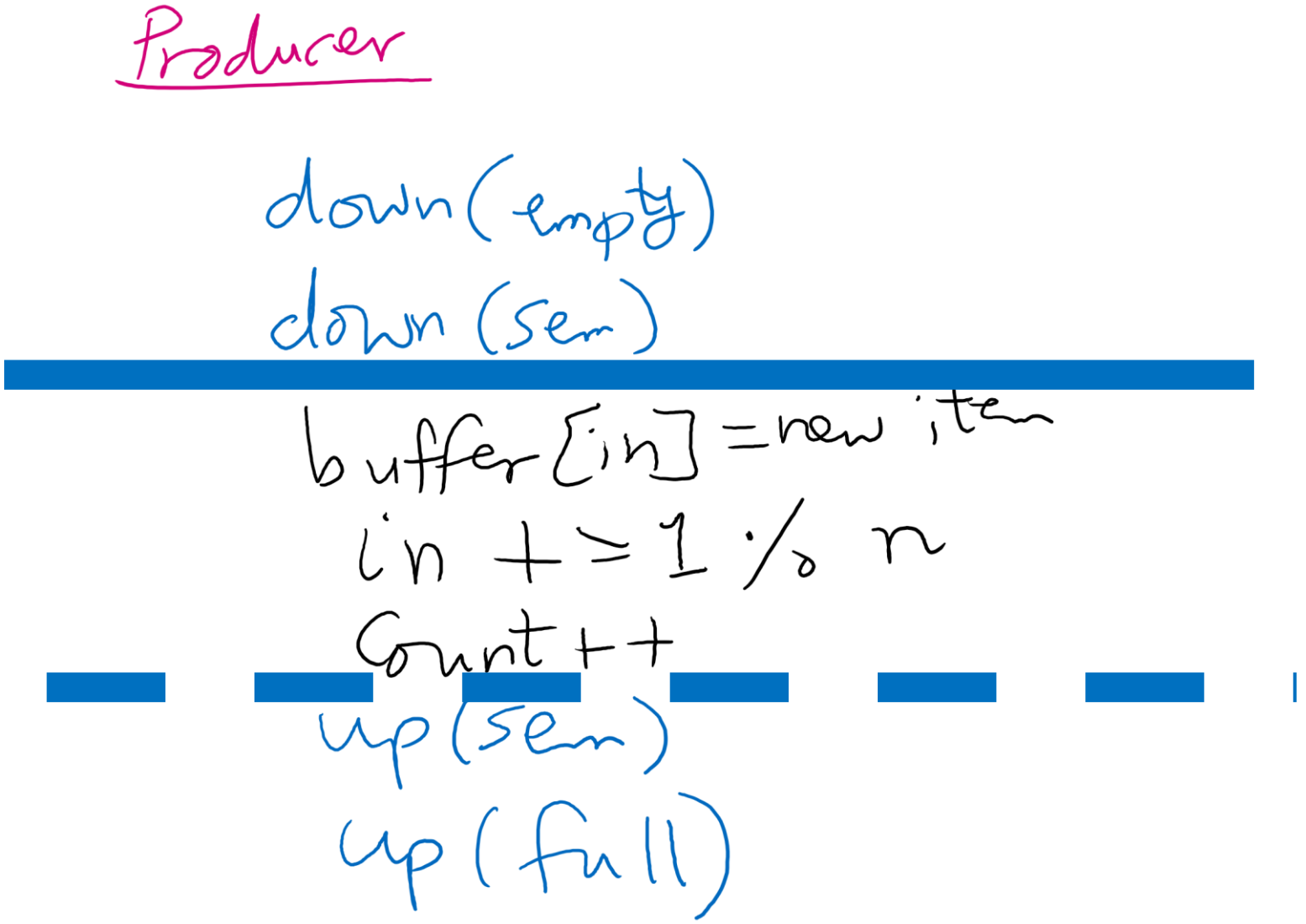
Producer arrives

Moves as far as possible until the solid line



Producer enters

Moves as far as possible past the solid line and until the dashed line



Producer leaves

Moves as far as possible until the dotted line

Producer

down(empty)

down(sem)

buffer[in] = new item

in += 1 % n

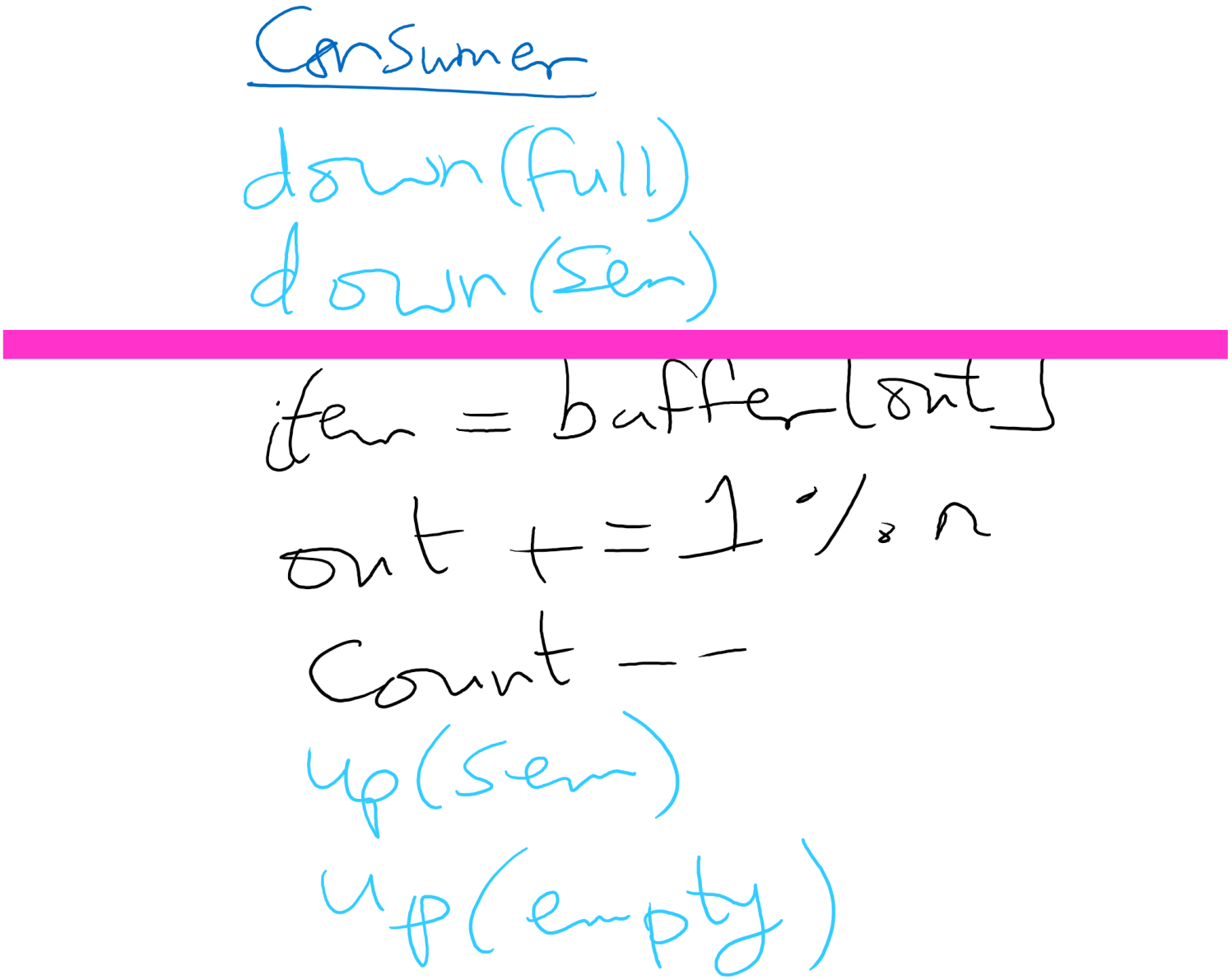
count++

up(sem)

up(full)

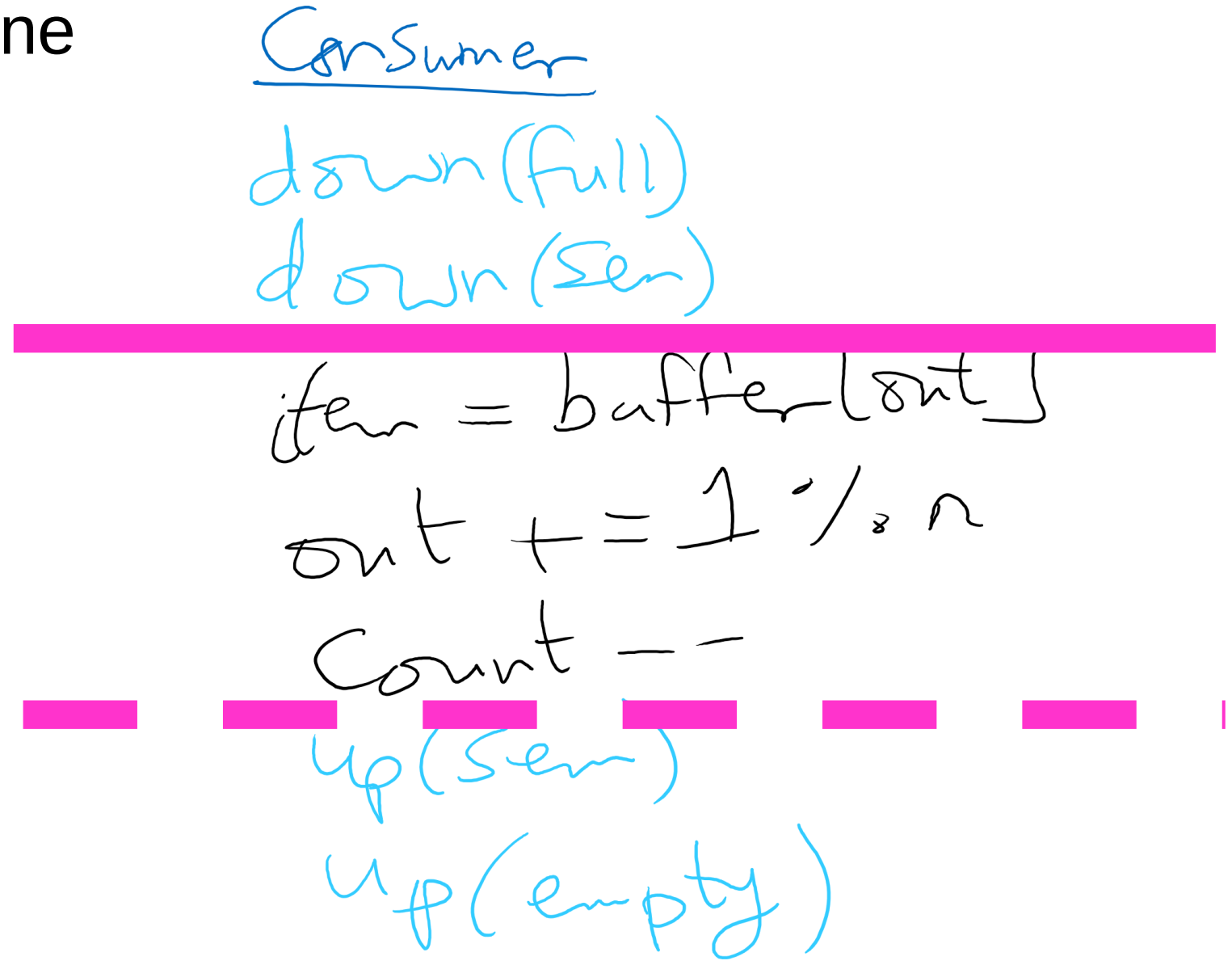
Consumer arrives

Moves as far as possible until the solid line



Consumer enters

Moves as far as possible past the solid line and until the dashed line



Consumer leaves

Moves as far as possible past the dashed line and until the dotted line

Consumer

down(full)

down(sem)

item = buffer[out]

out += 1 % n

count--

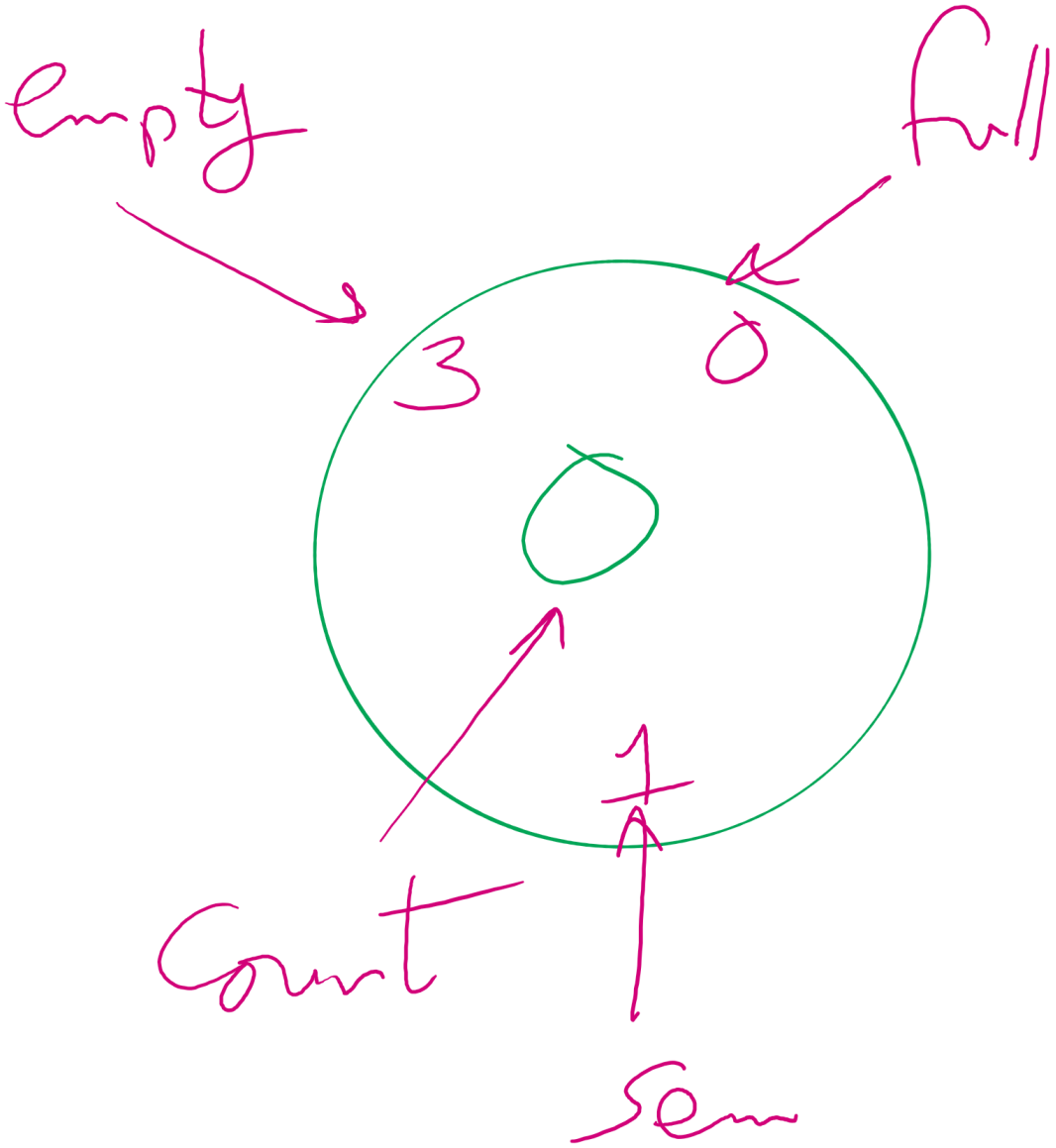
up(sem)

up(empty)

Tracing

Given a sequence of events, is the sequence *feasible*?
If yes, what is the *system state* at the end of the sequence?

System State

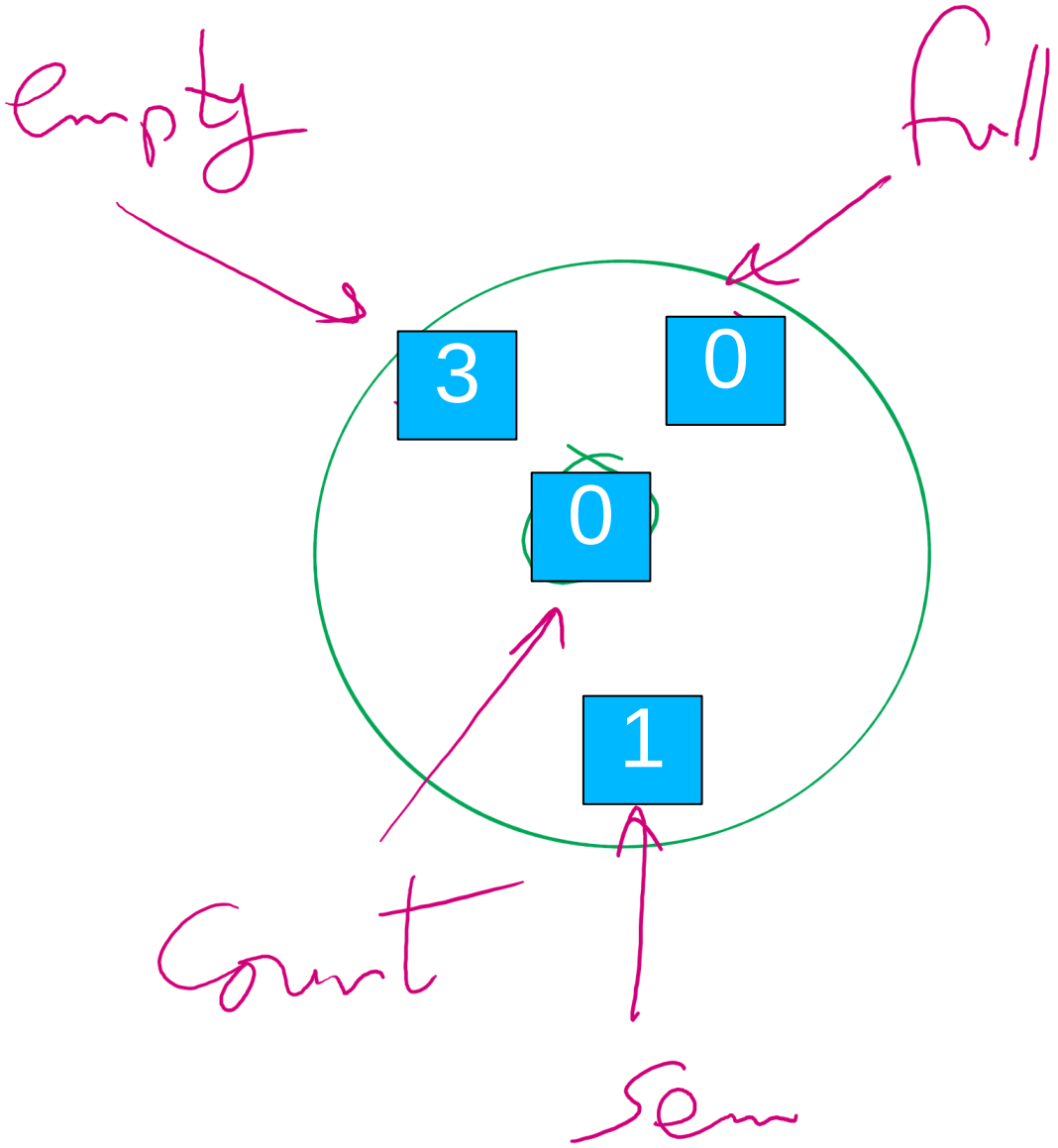


Example 1

n == 3

Producer 0 arrives
Producer 0 enters
Producer 1 arrives
Producer 2 arrives
Consumer 0 arrives
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves

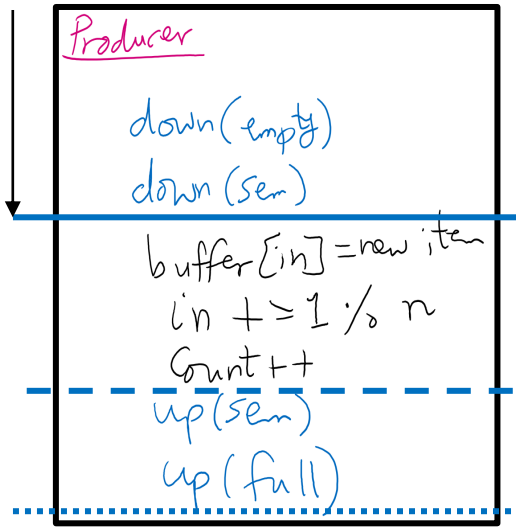
Initial state

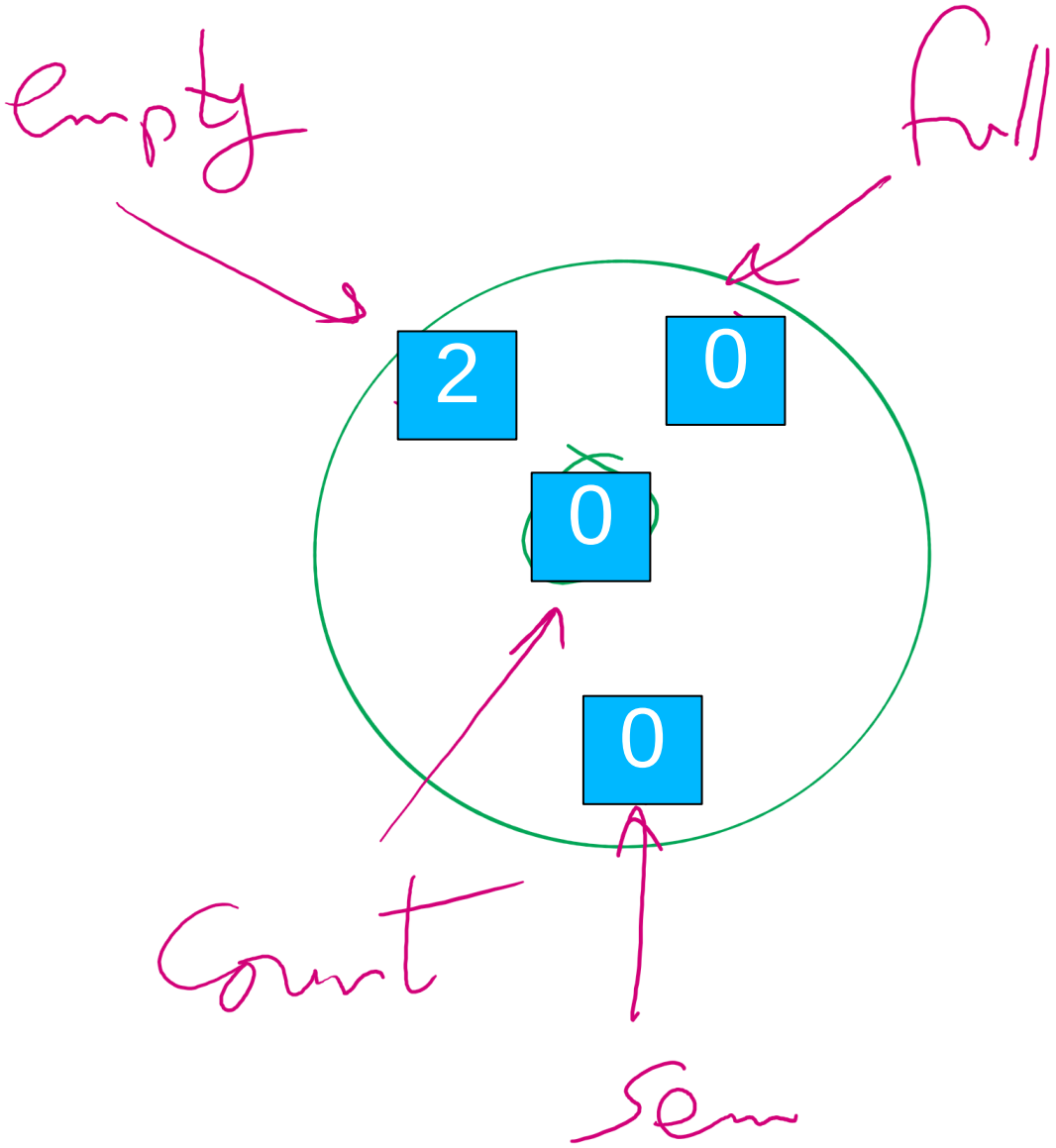


n == 3

Producer 0 arrives
Producer 0 enters
Producer 1 arrives
Producer 2 arrives
Consumer 0 arrives
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves

P0 arrives

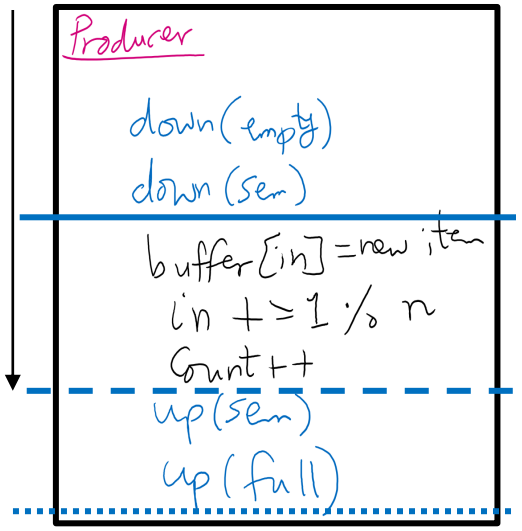


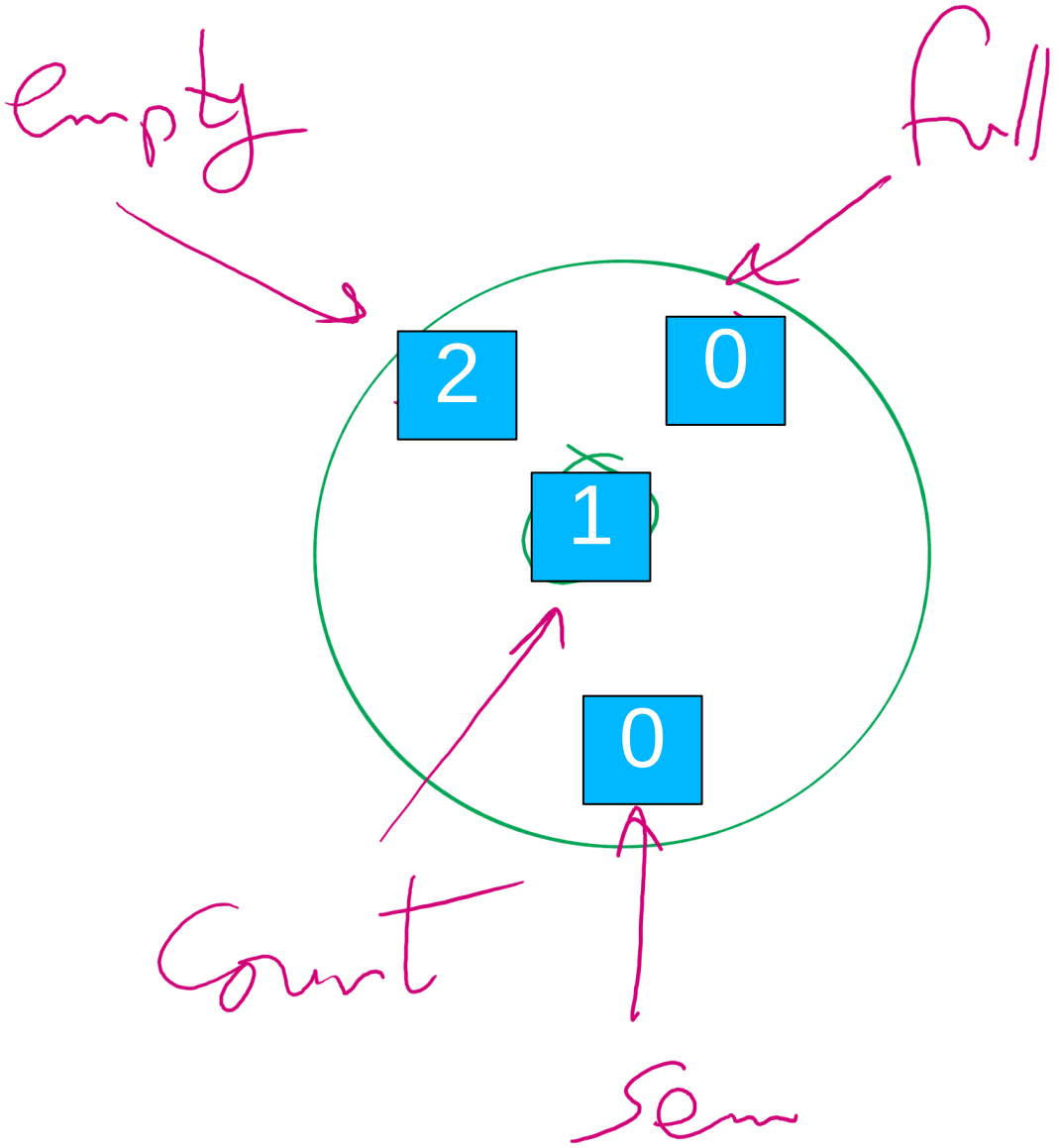


n == 3

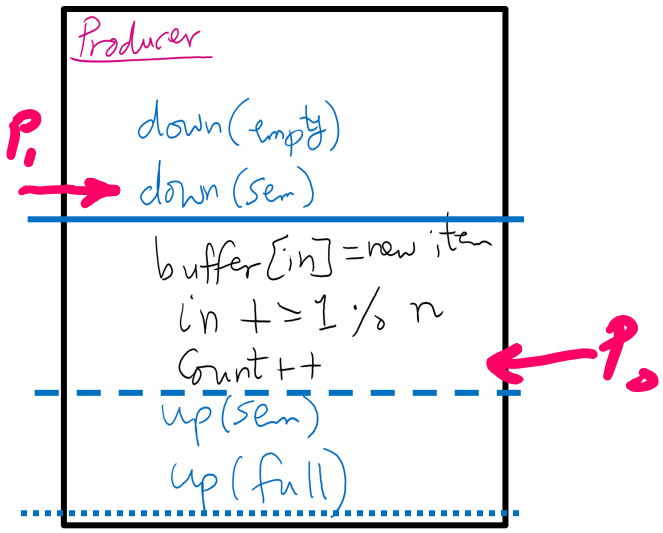
Producer 0 arrives
Producer 0 enters
Producer 1 arrives
Producer 2 arrives
Consumer 0 arrives
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves

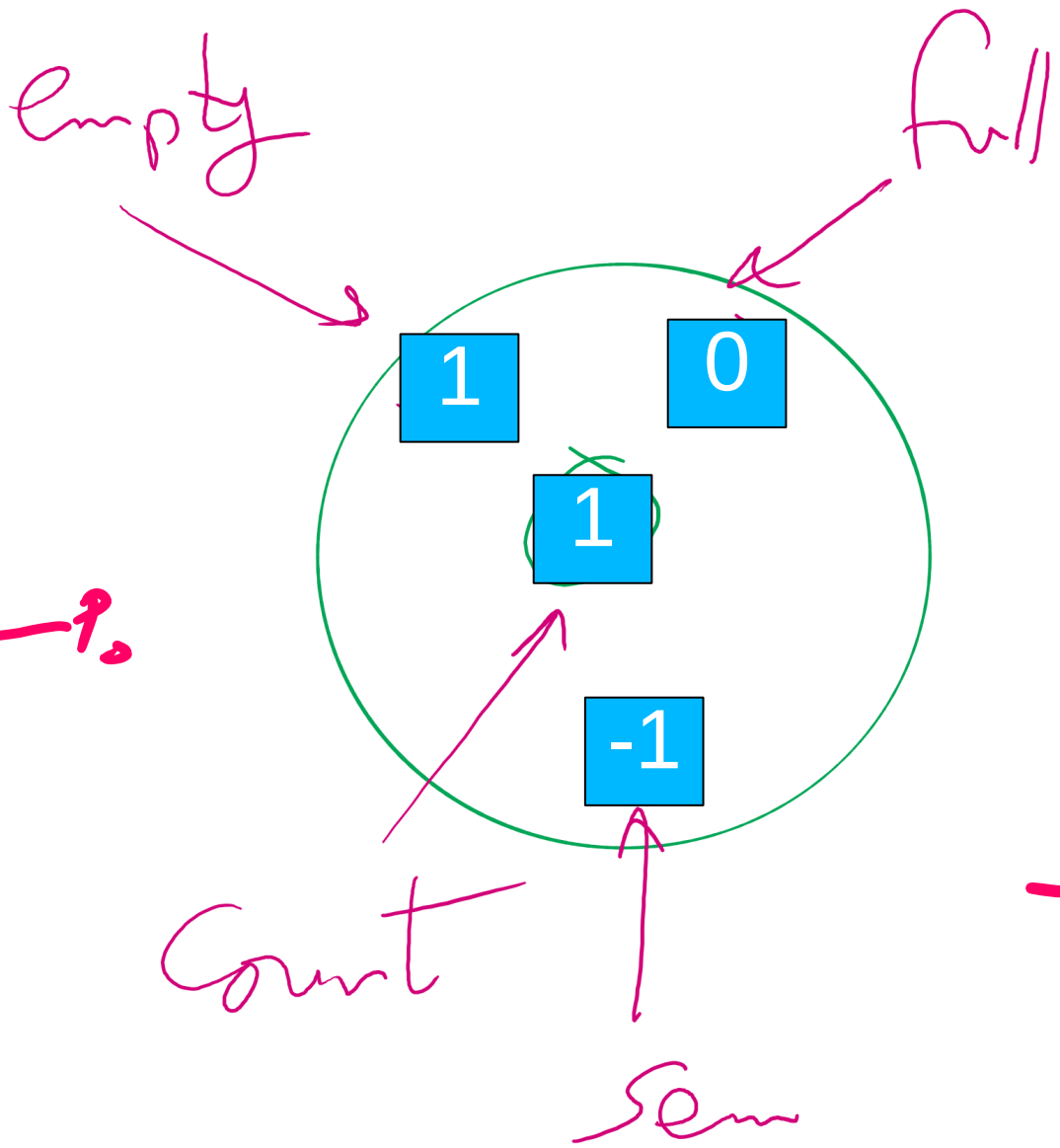
P0 enters


Producer
down(empty)
down(sem)
buffer[in] = new item
in = (in + 1) % n
count++
up(sem)
up(full)


empty
full
2
0
1
0
count
sem

P1 arrives

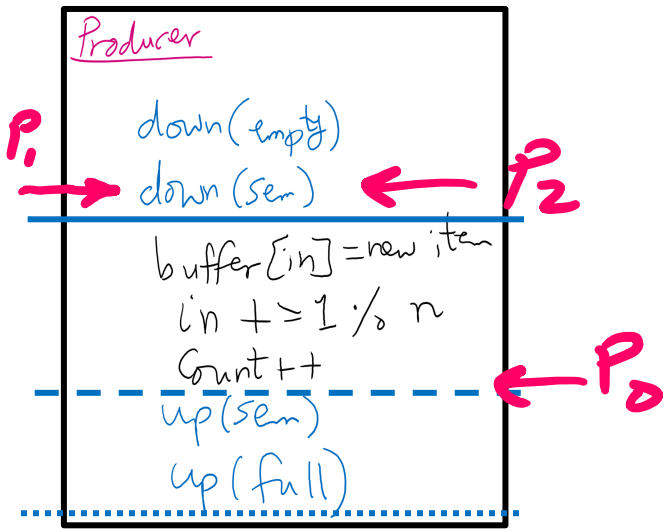


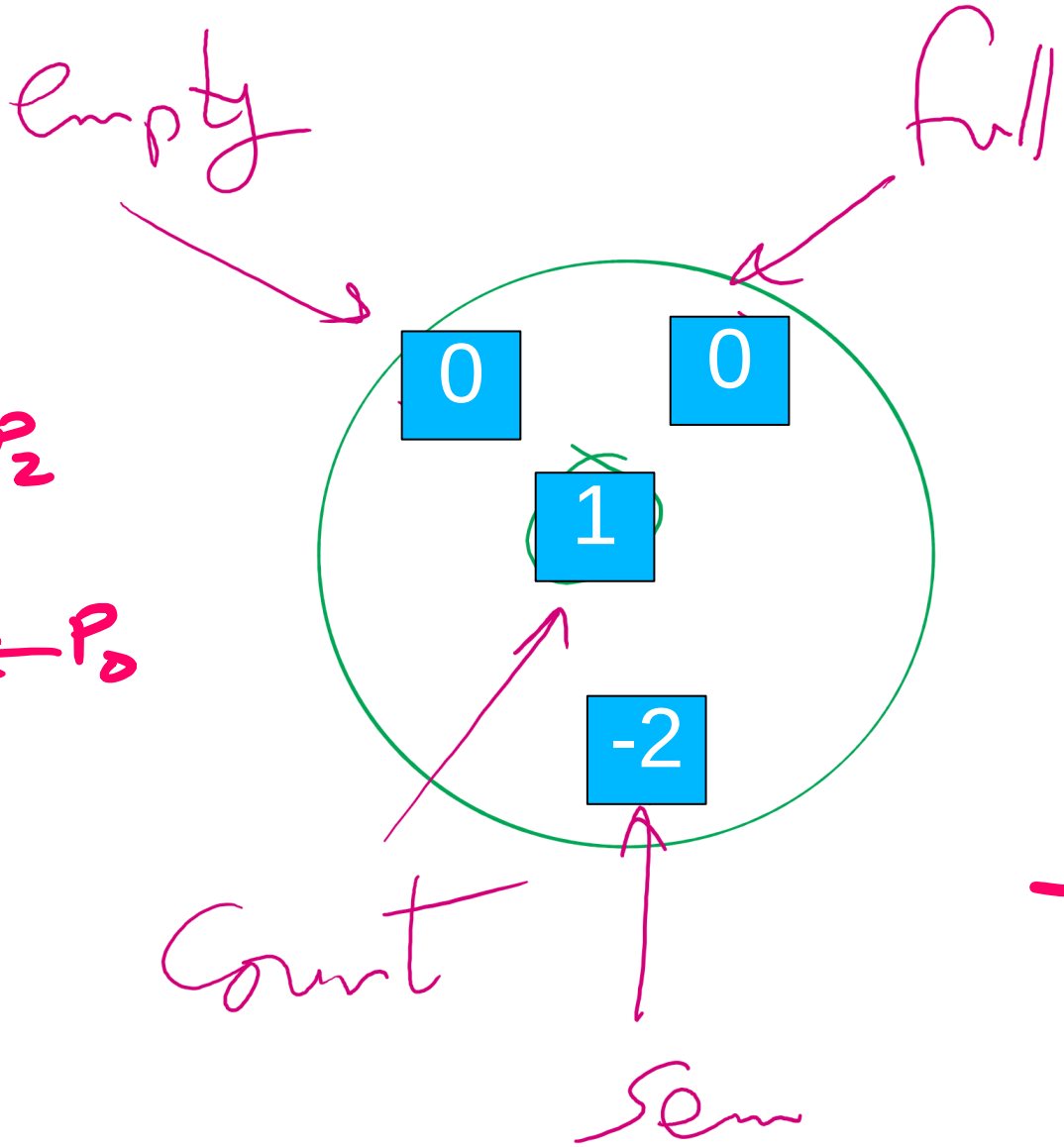


Sem Queue

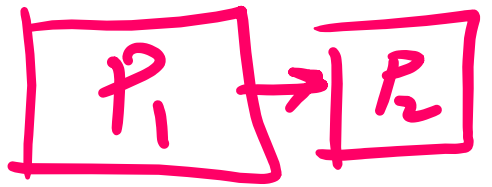
P1

P2 arrives

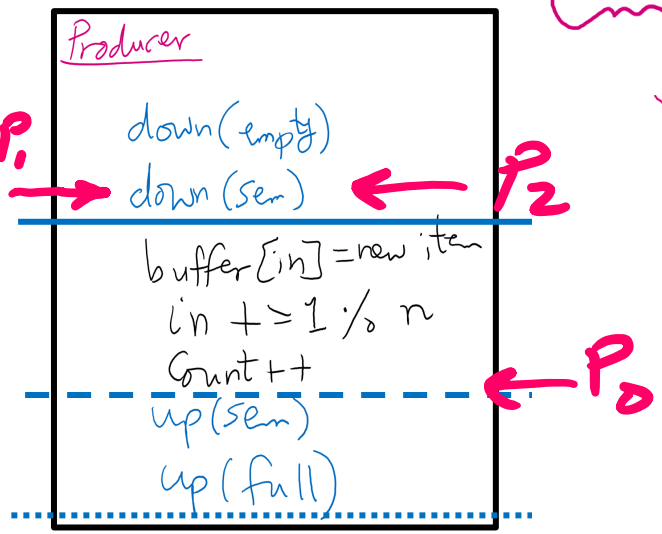


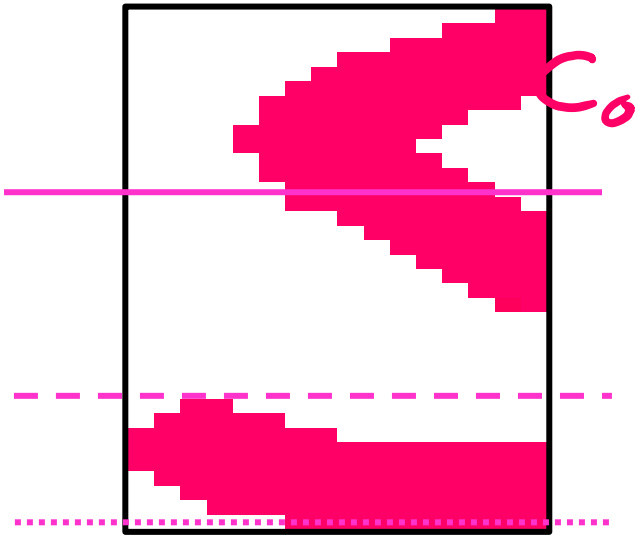


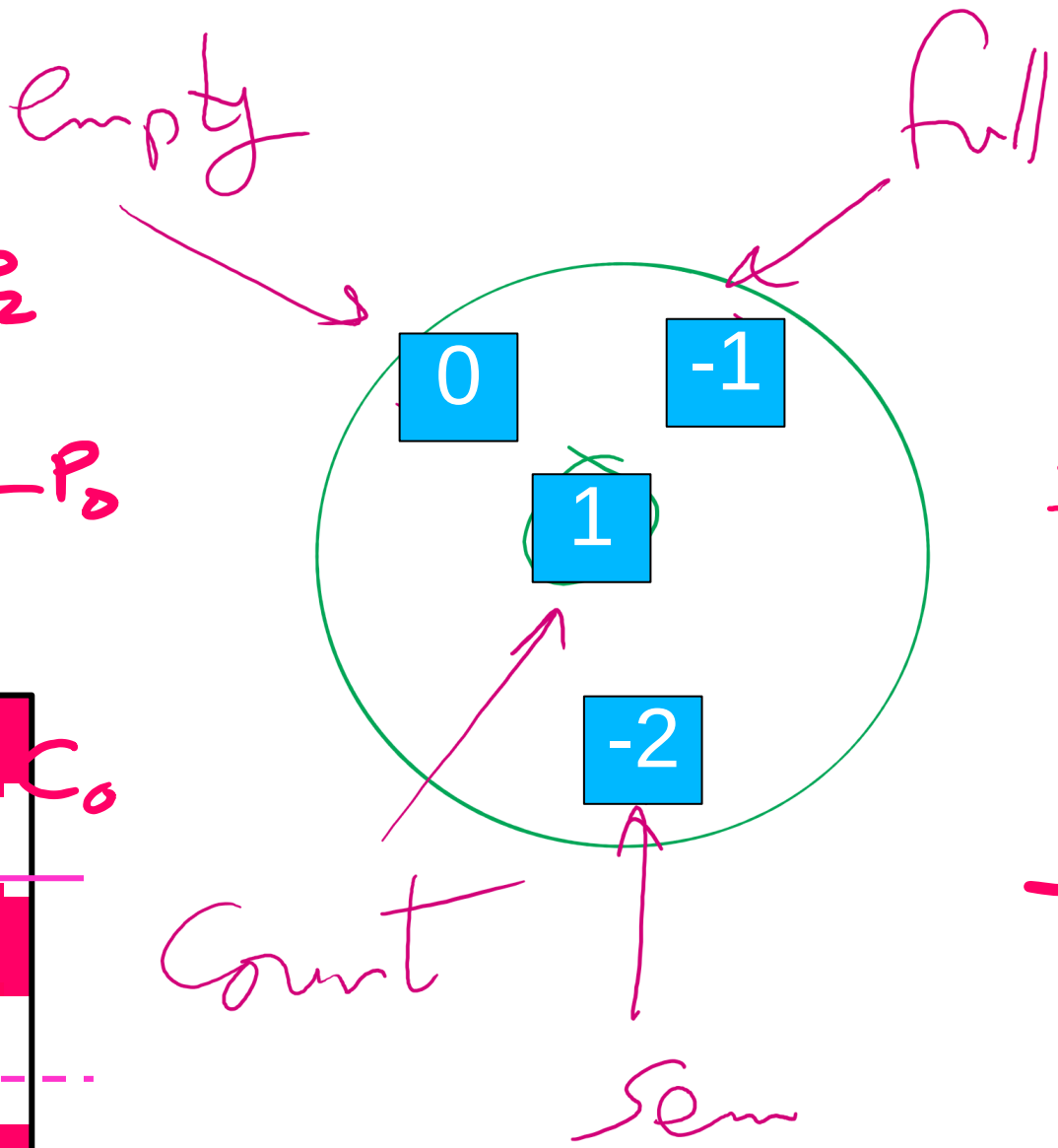
Sem Queue

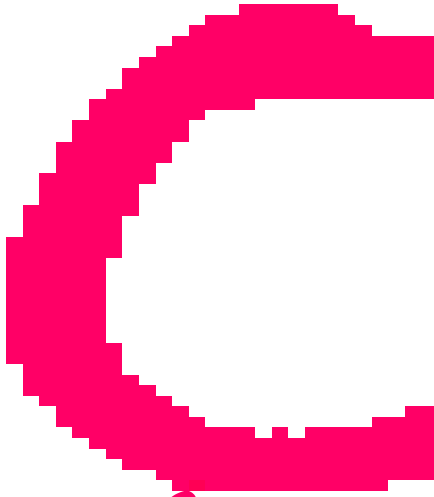


C0 arrives









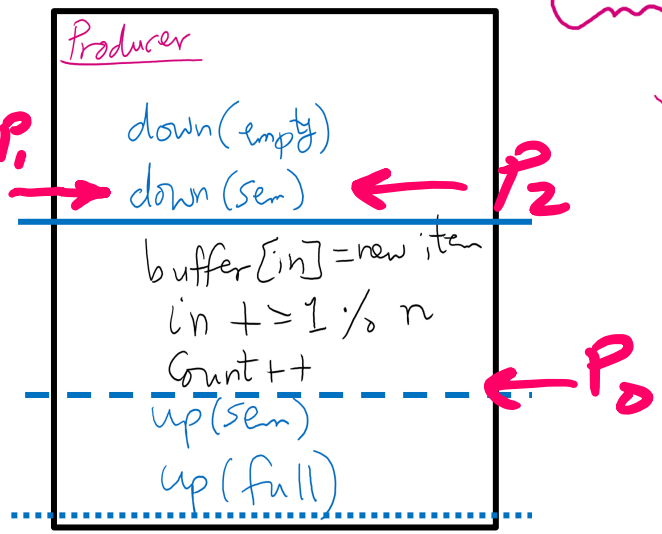
full Queue

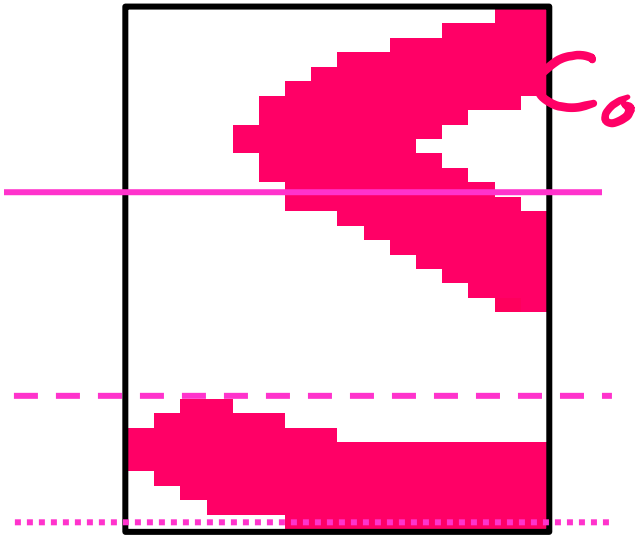
C0

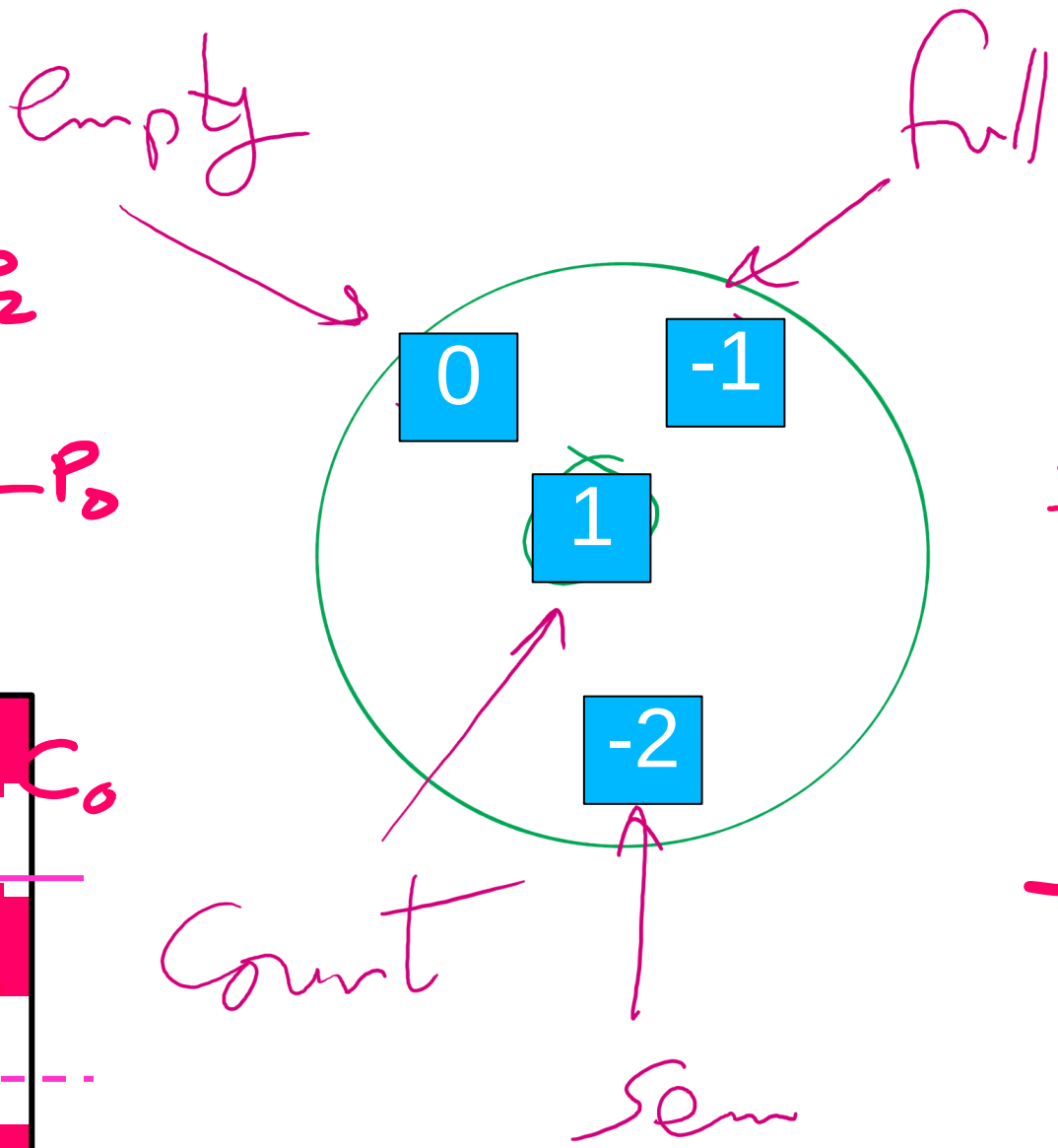
Sem Queue

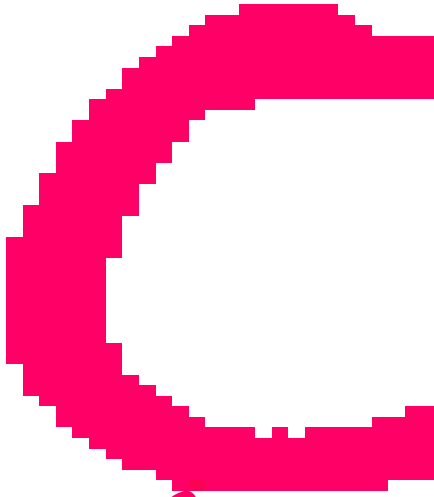
P1 → P2

P0 leaves









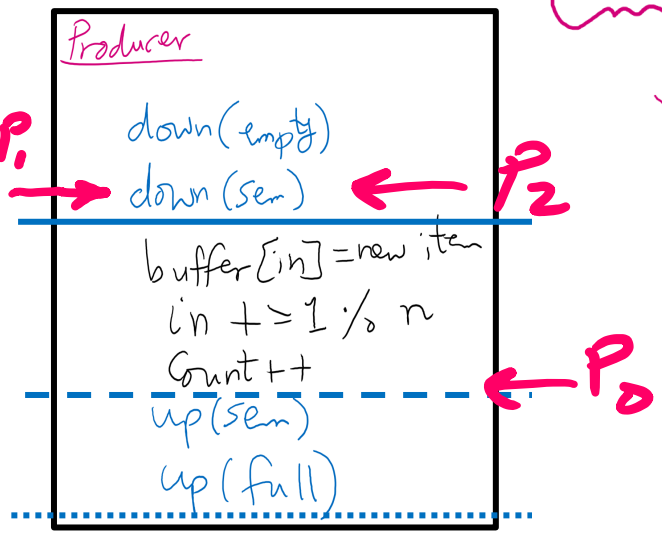
full Queue

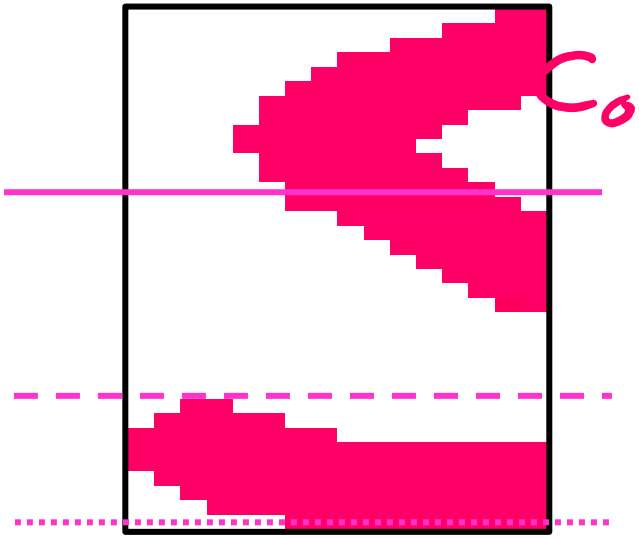
C0

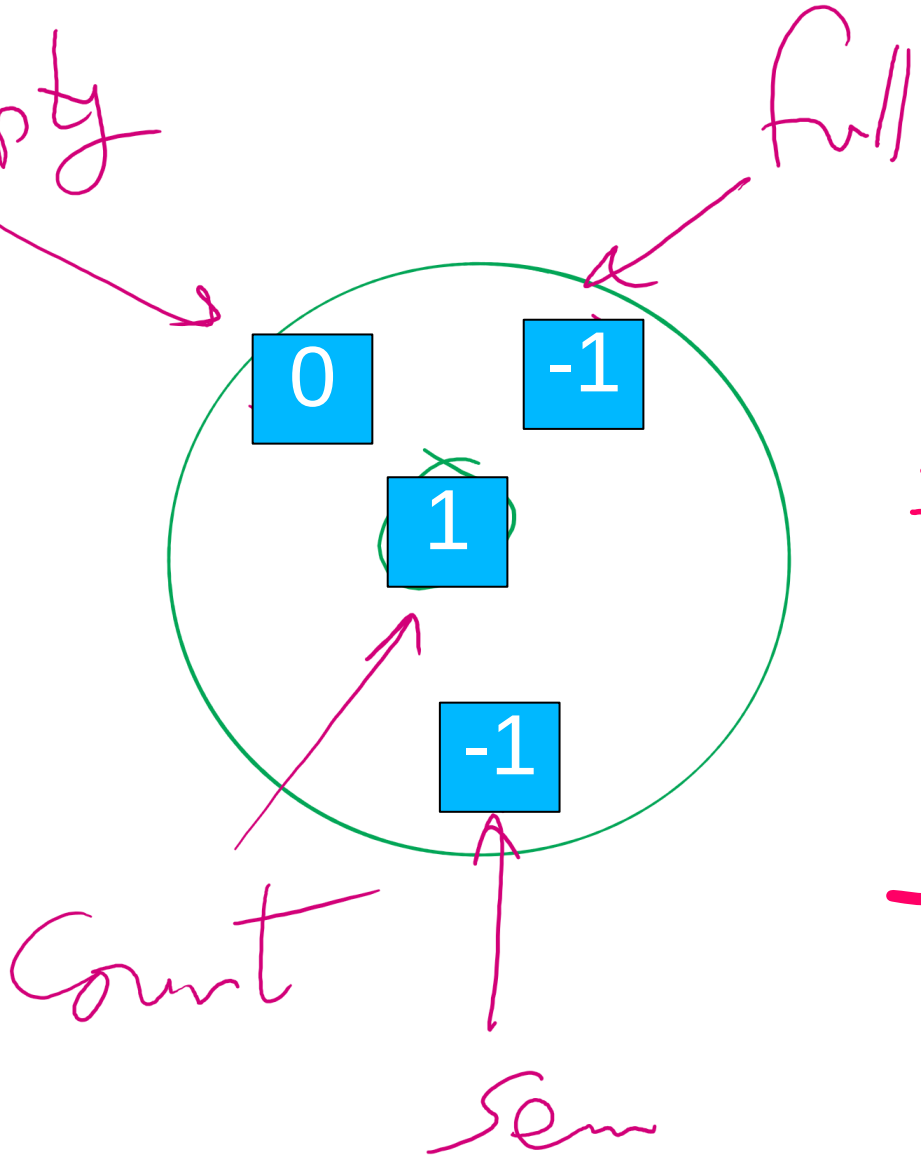
Sem Queue

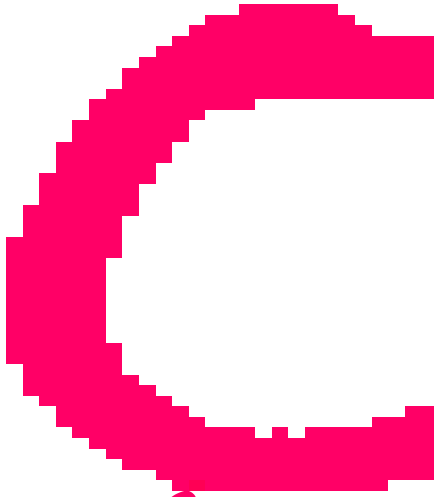
P1 → P2

P0 leaves









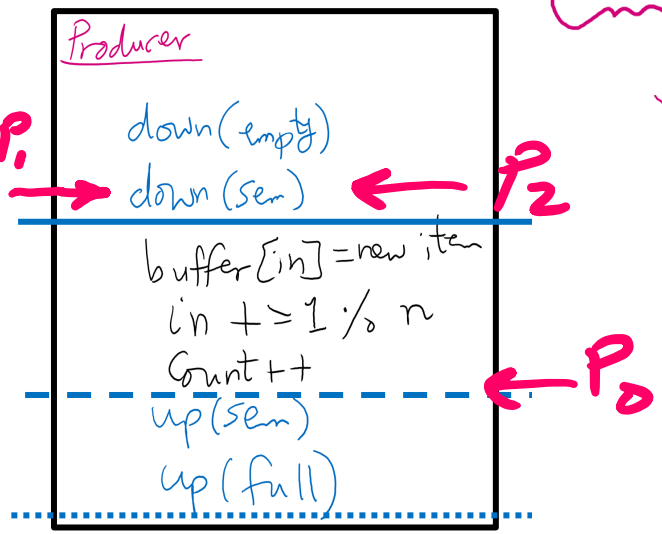
full Queue

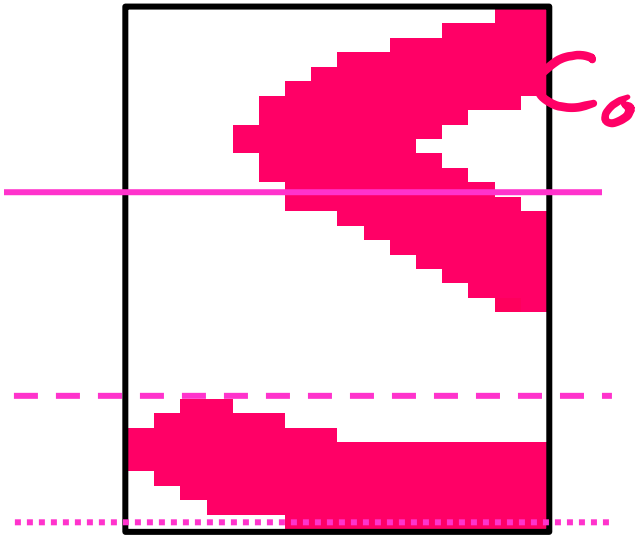
C0

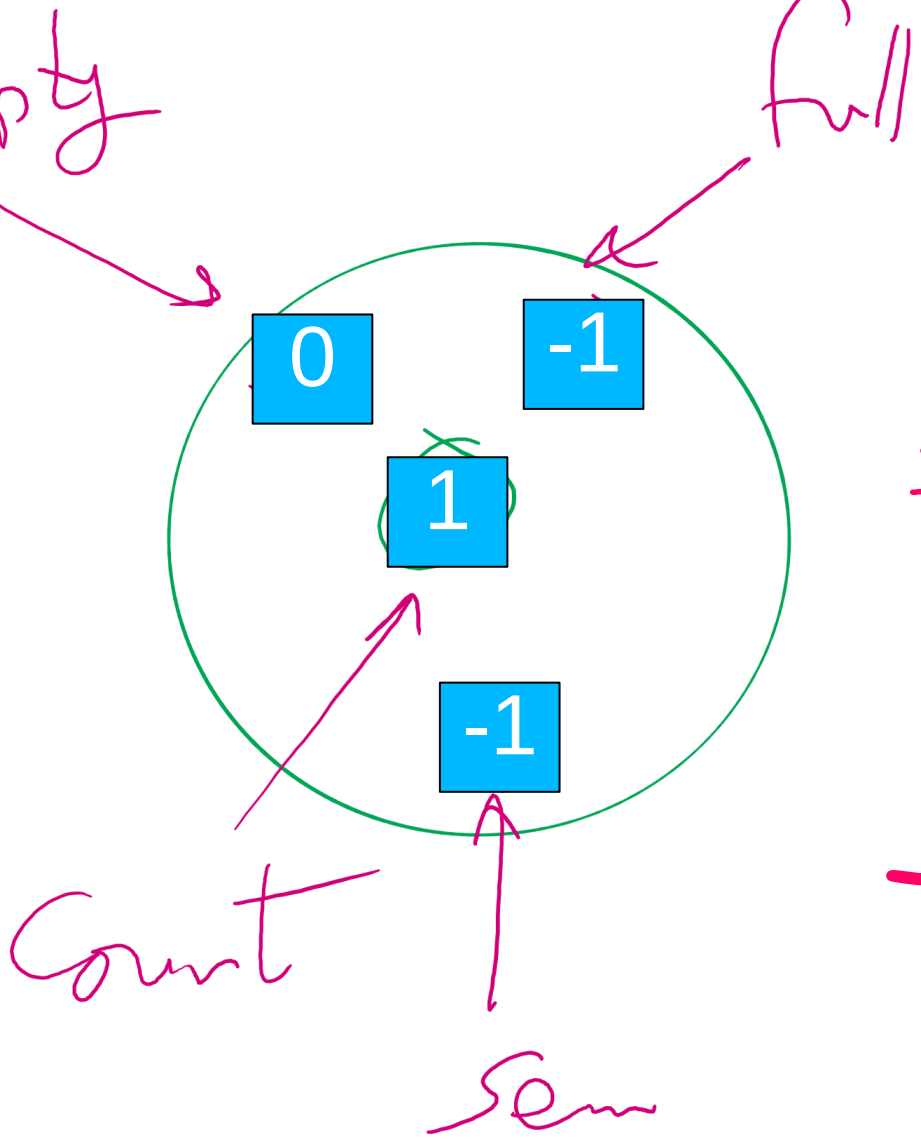
Sem Queue

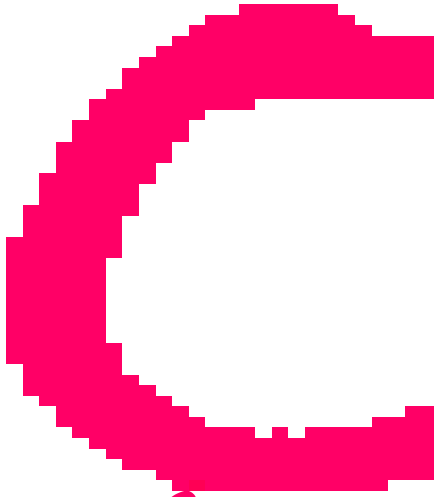
P1 → P2

P0 leaves









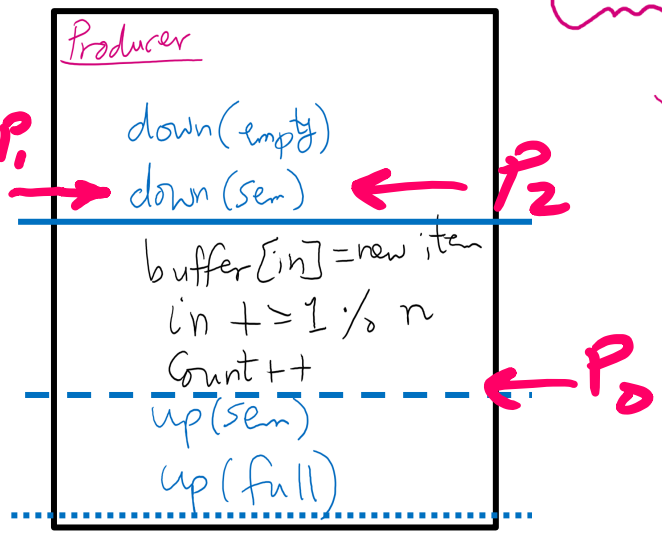
full Queue

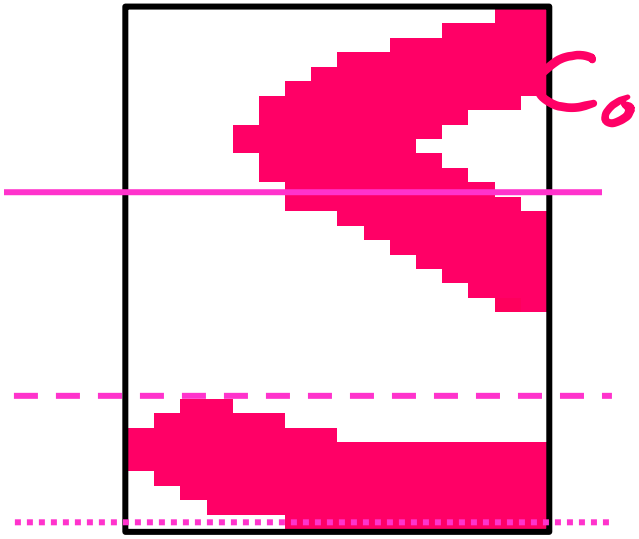
C0

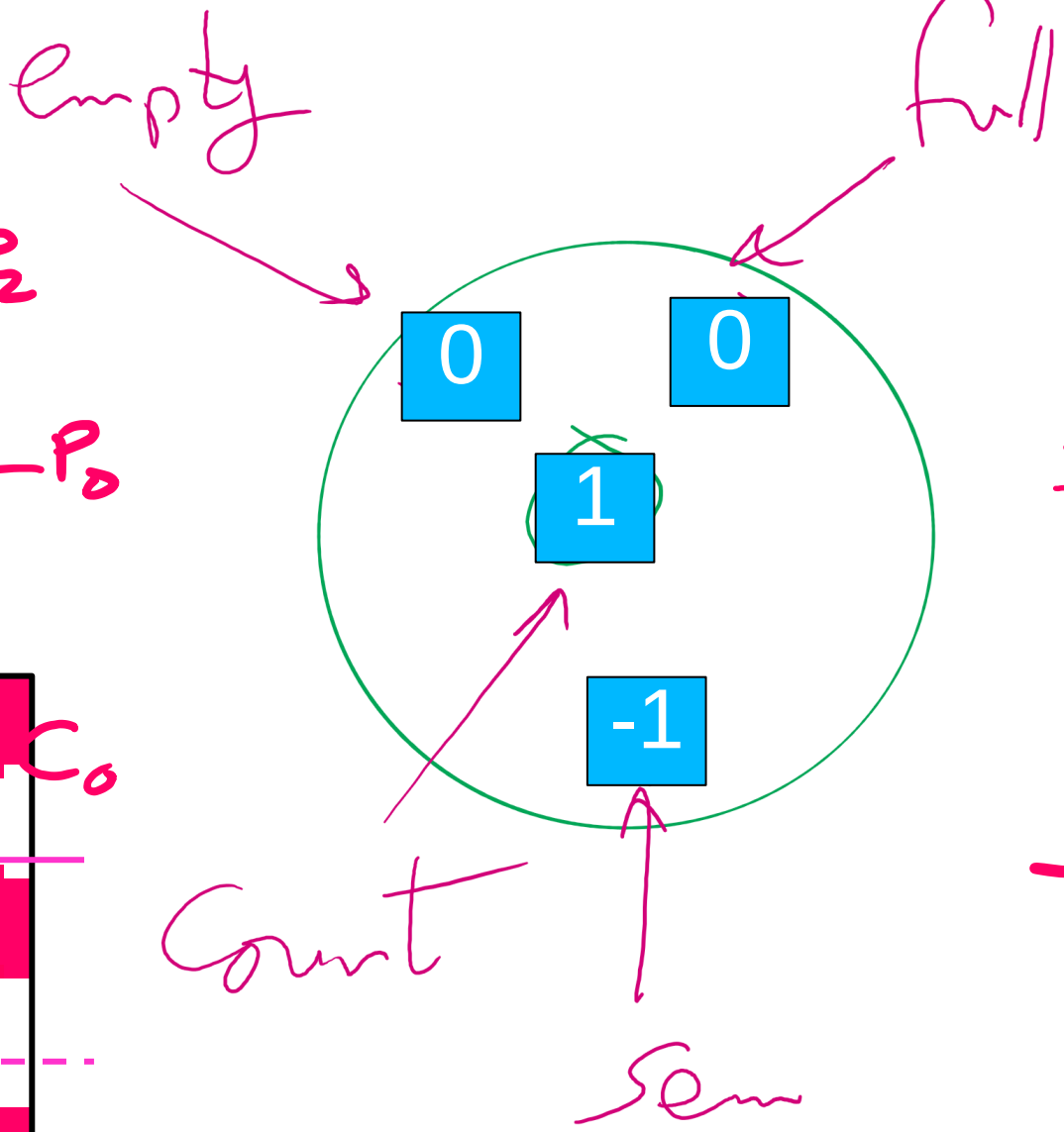
Sem Queue

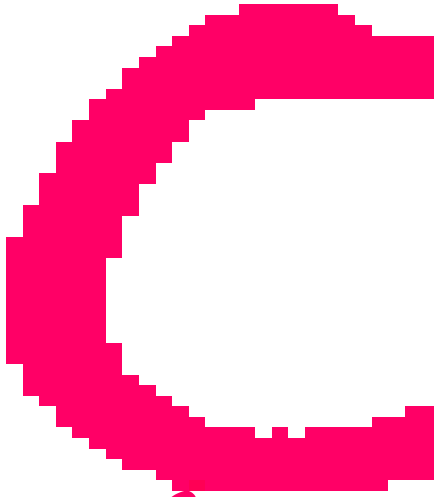
P2

P0 leaves









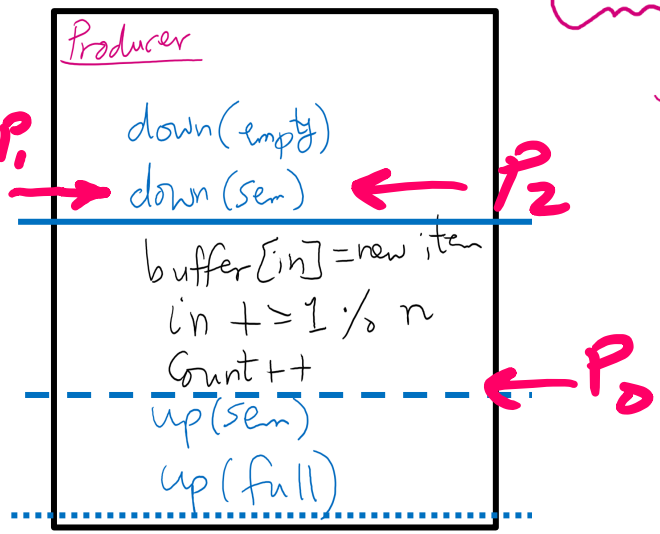
full Queue

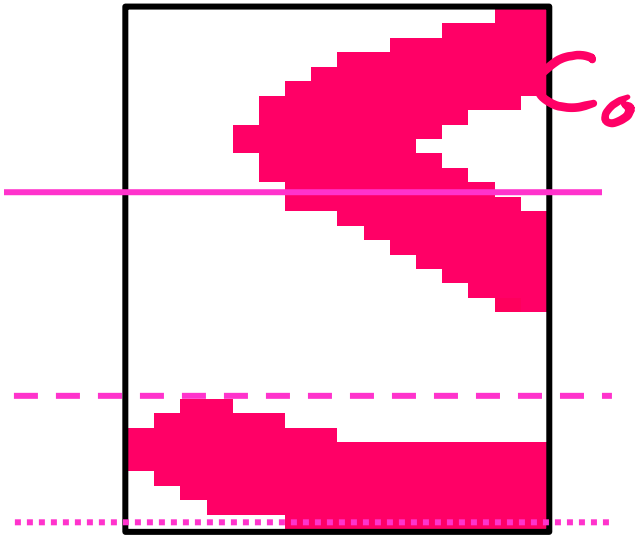
C0

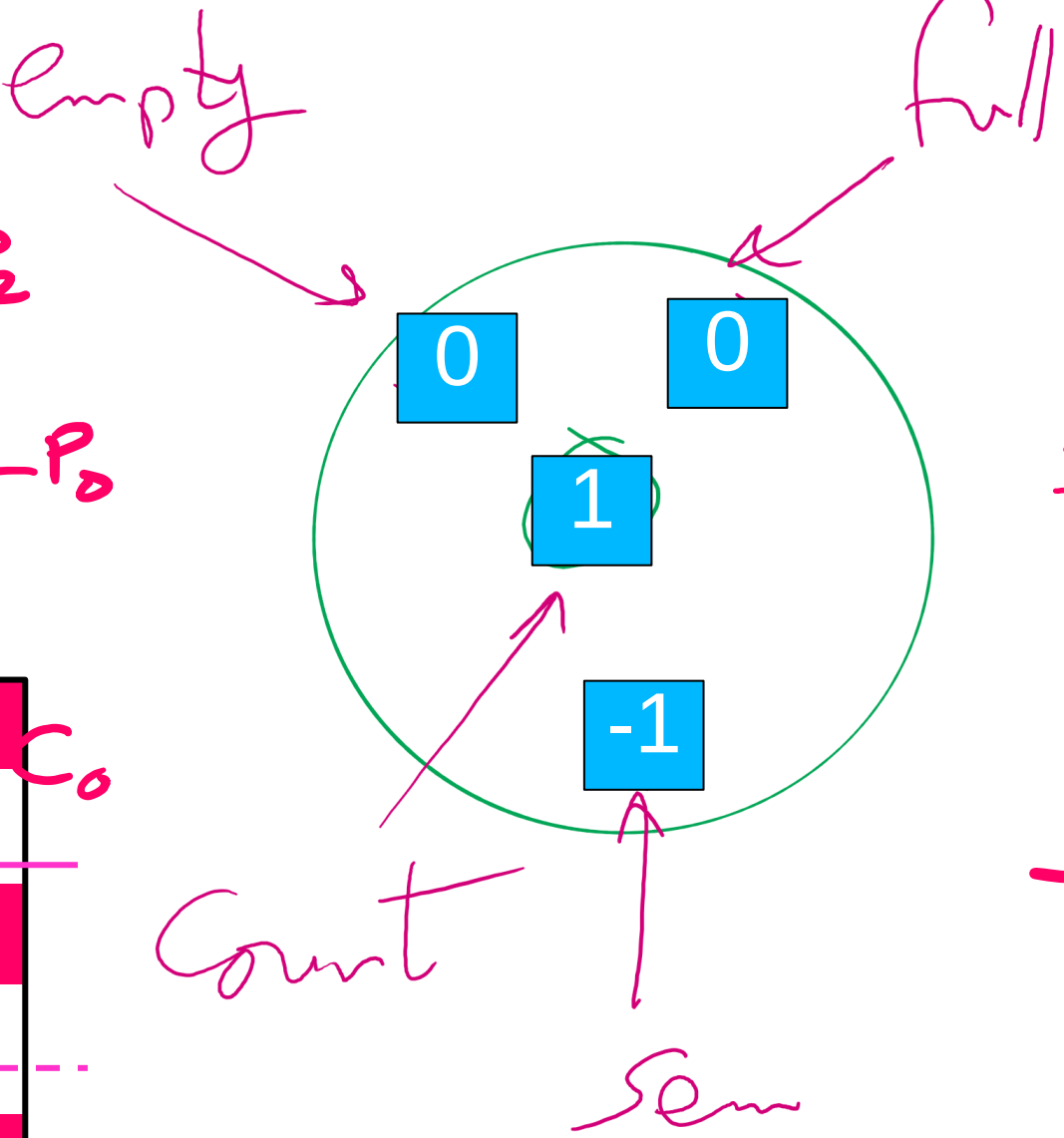
Sem Queue

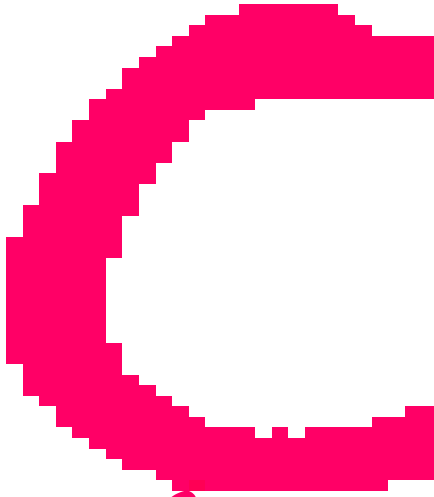
P2

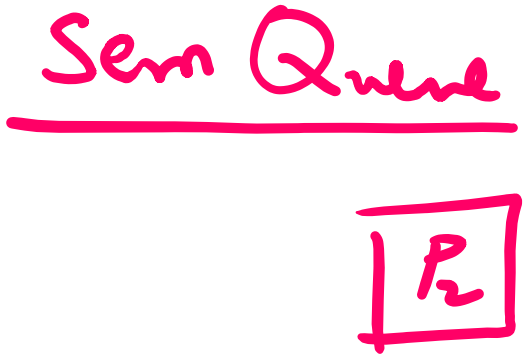
C0 enters


Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P1
P2
P0

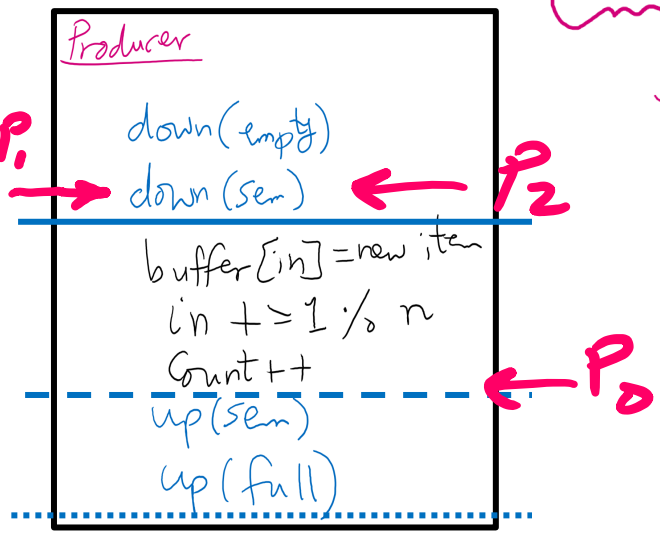

C0

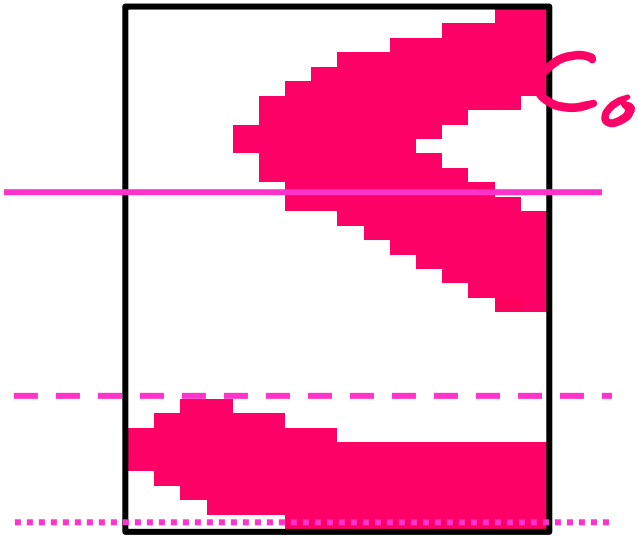

empty
full
0
0
1
-1
count
sem

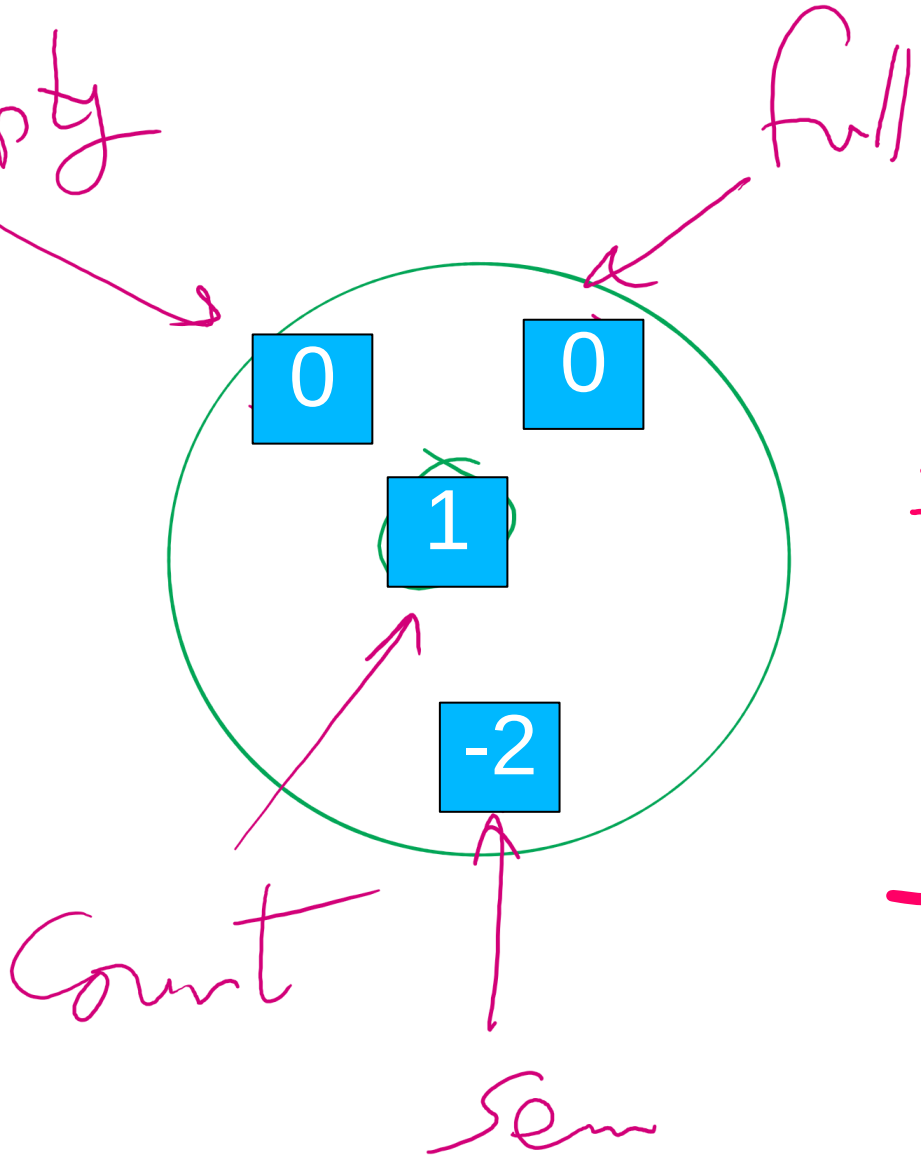

full Queue

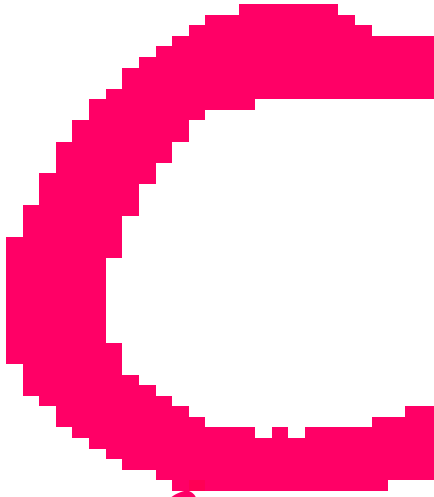

Sem Queue
P2

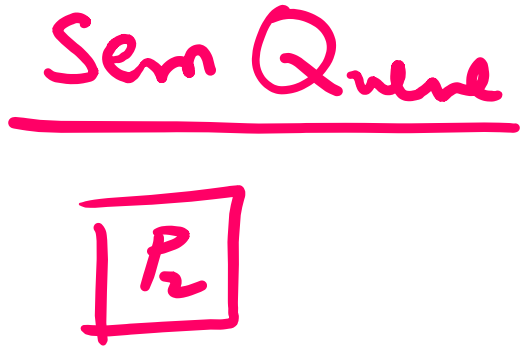
C0 enters


Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P1
P2
P0

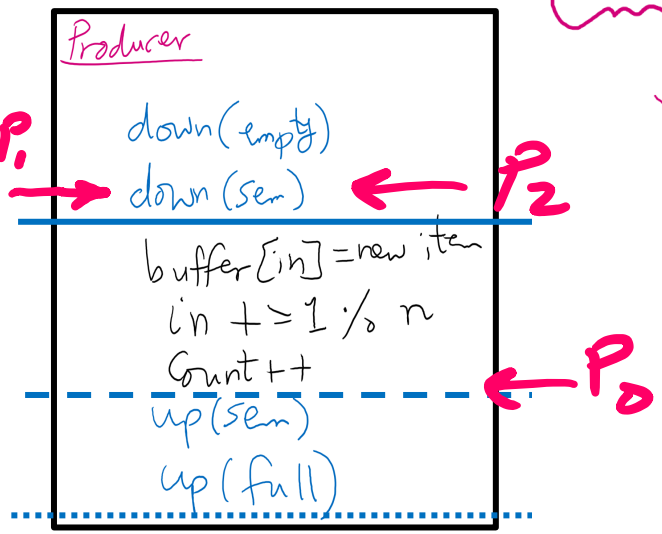

C0

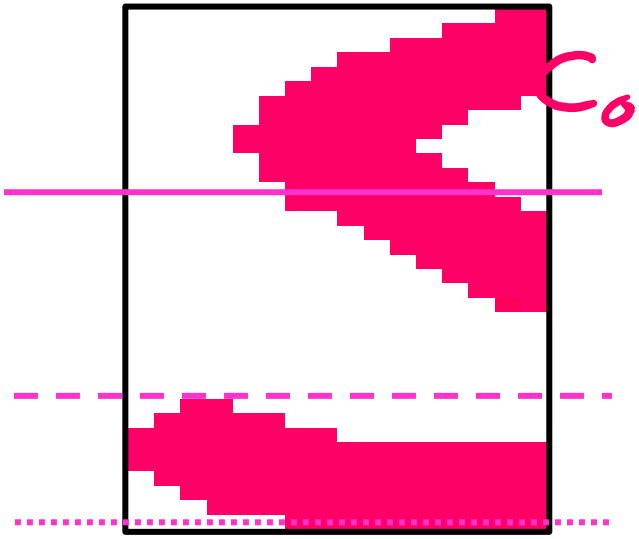

empty
full
0
0
1
-2
count
sem

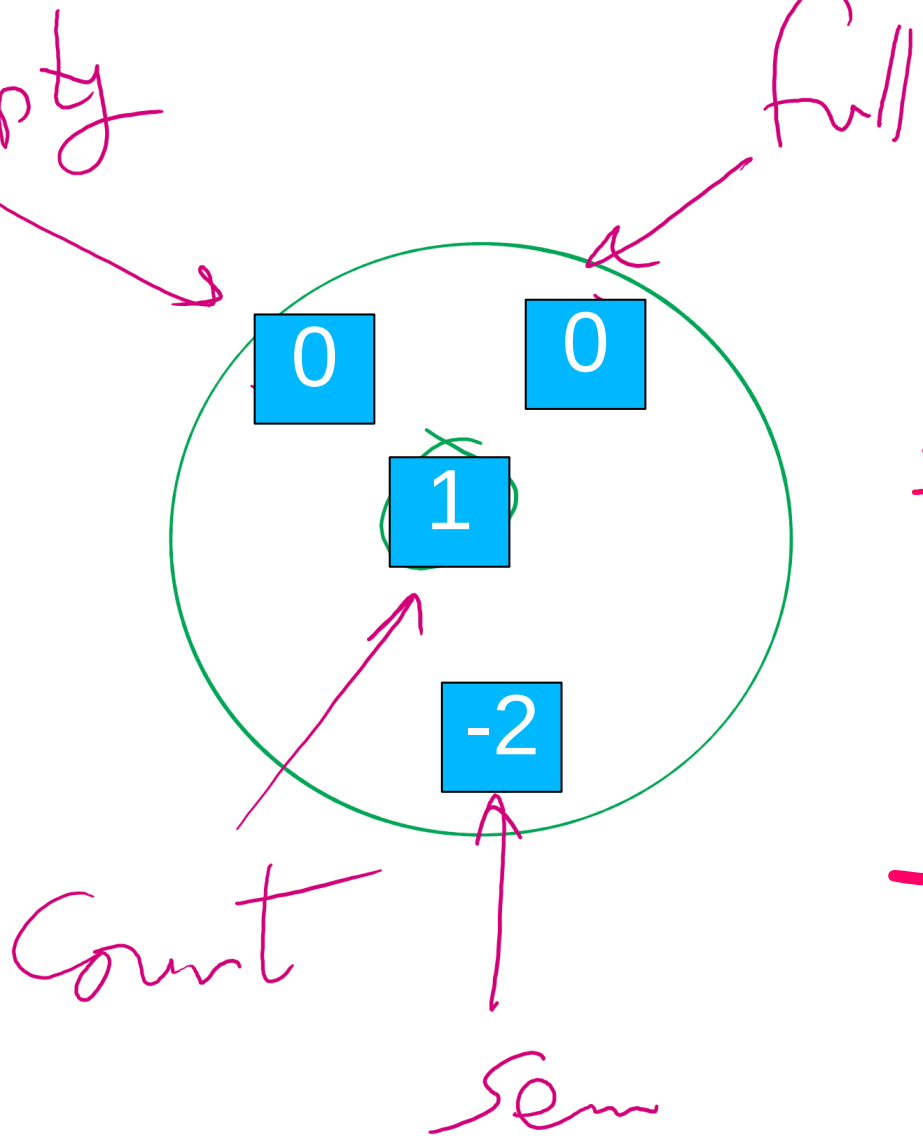

full Queue

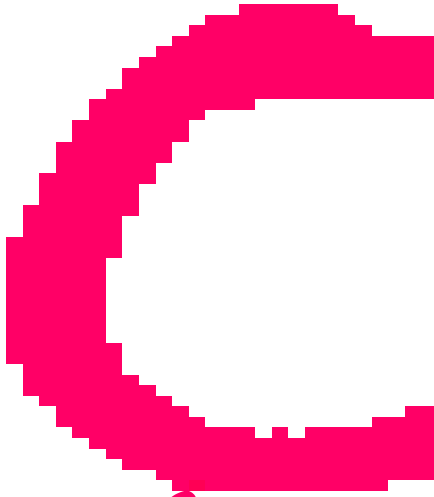

Sem Queue
P2

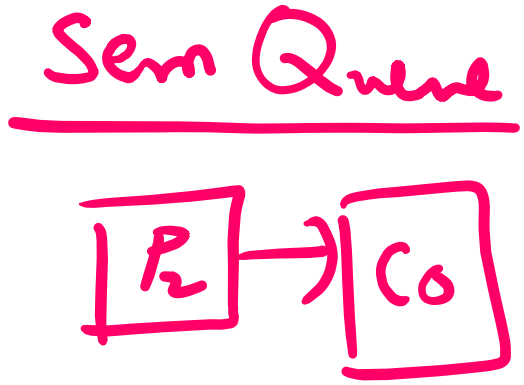
C0 enters


Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P1
P2
P0

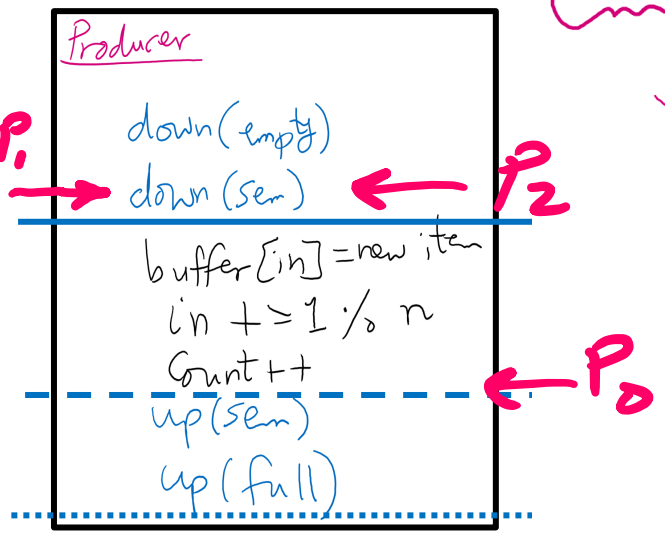

C0

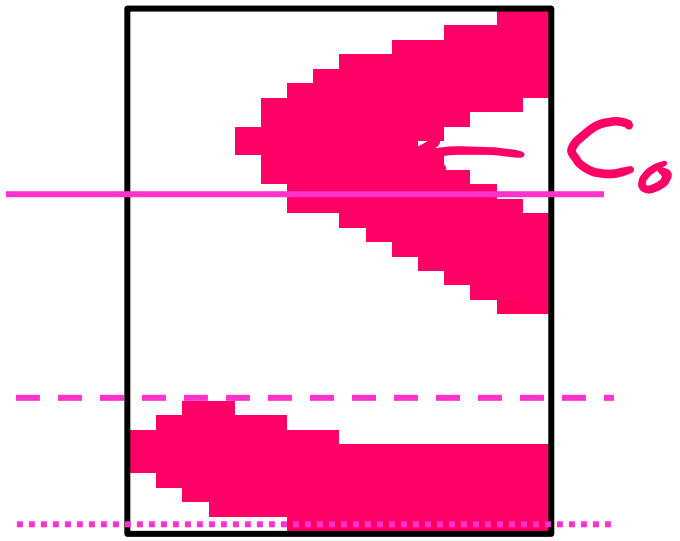

empty
full
0
0
1
-2
count
sem

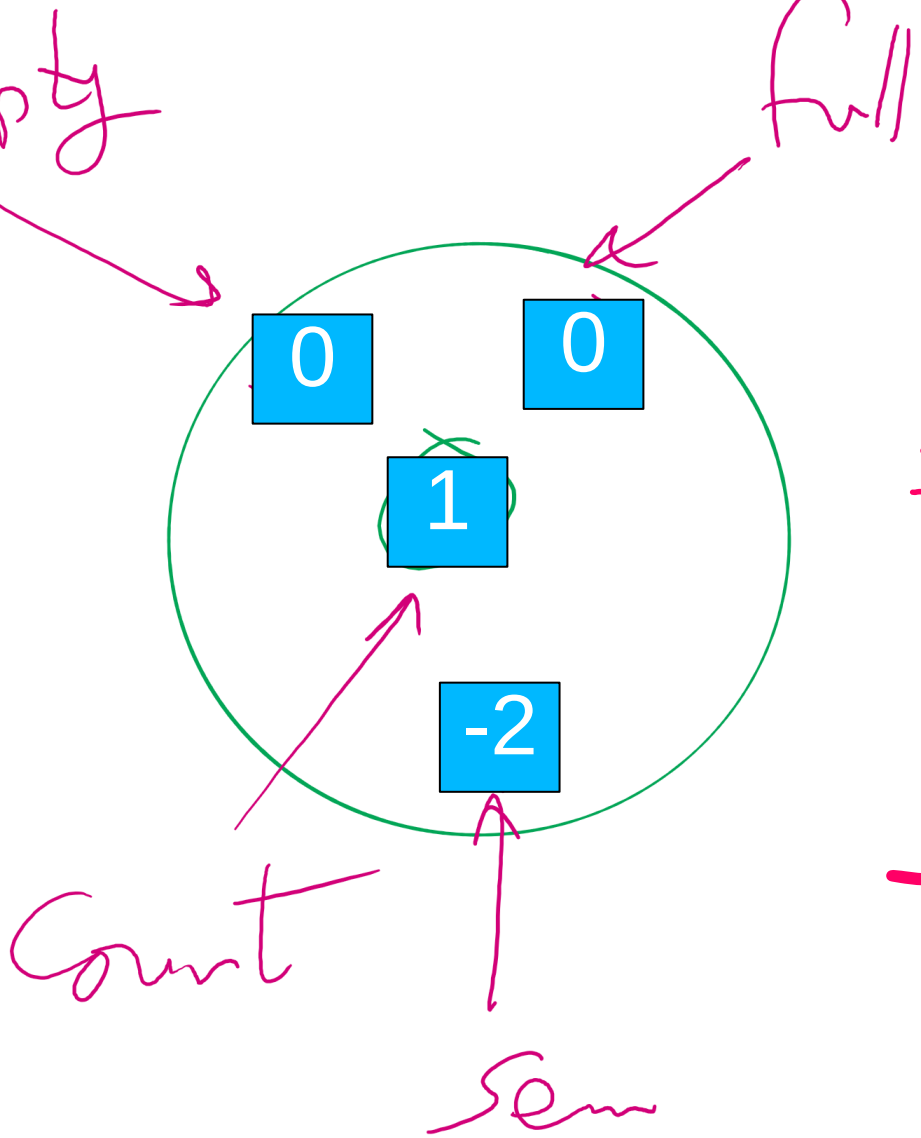

full Queue

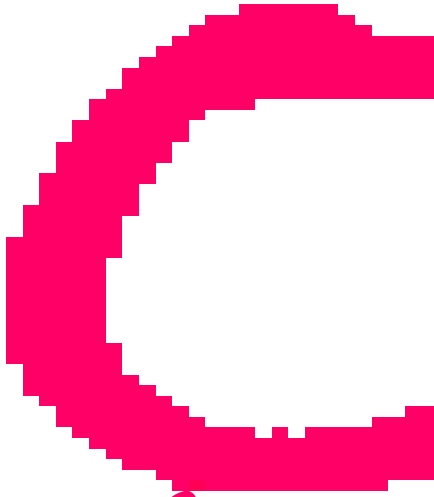

Sem Queue
P2
C0

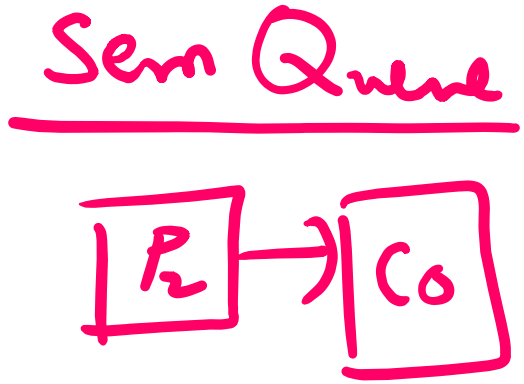
Can C0 enter?


Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P1
P2
P0


C0


empty
full
0
0
1
-2
count
sem


full Queue


Sem Queue
P2
C0

Example 2

n = = 1

Consumer 0 arives

Producer 0 arives

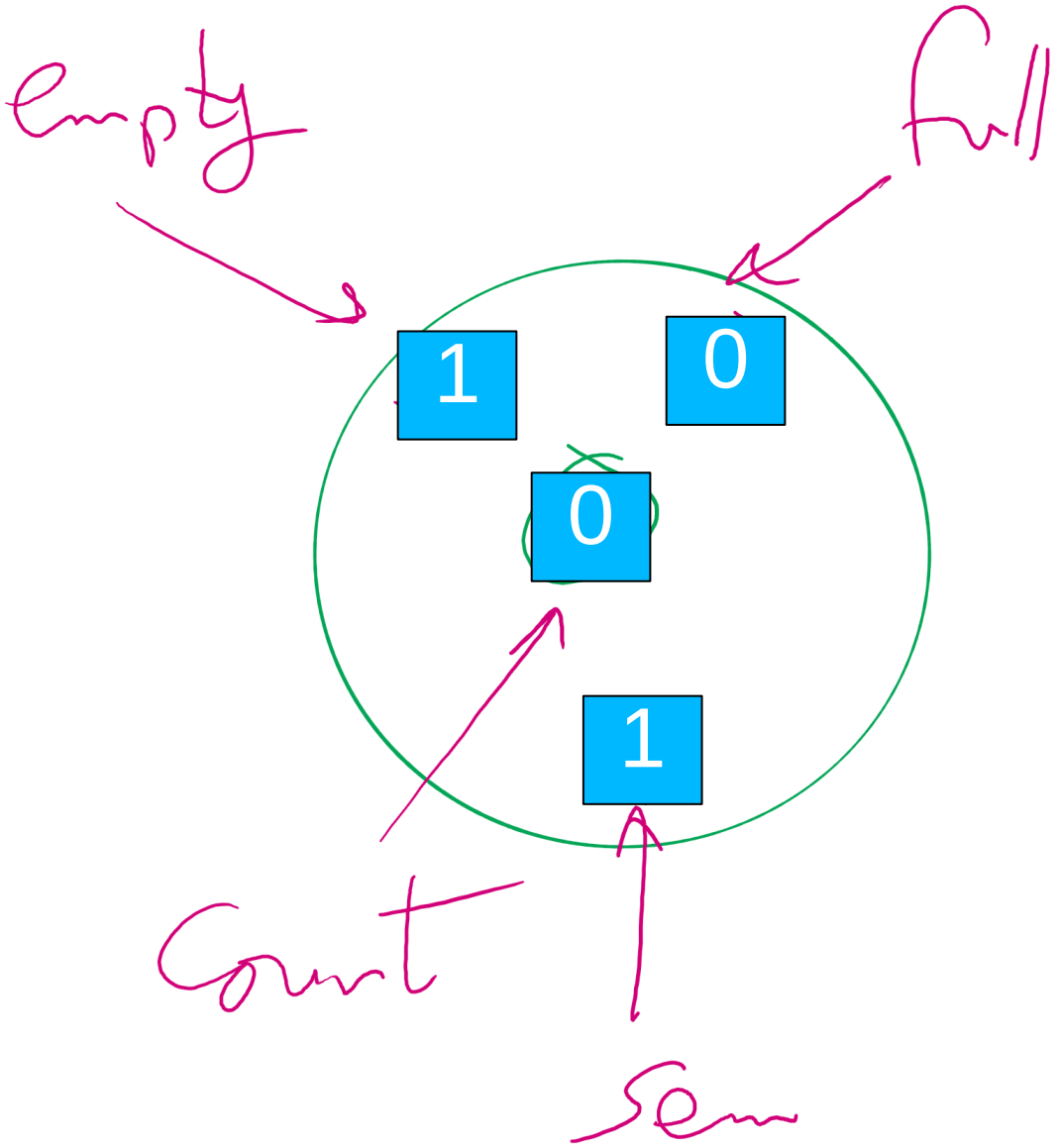
Producer 0 enters

Producer 0 leaves

Consumer 0 enters

Consumer 0 leaves

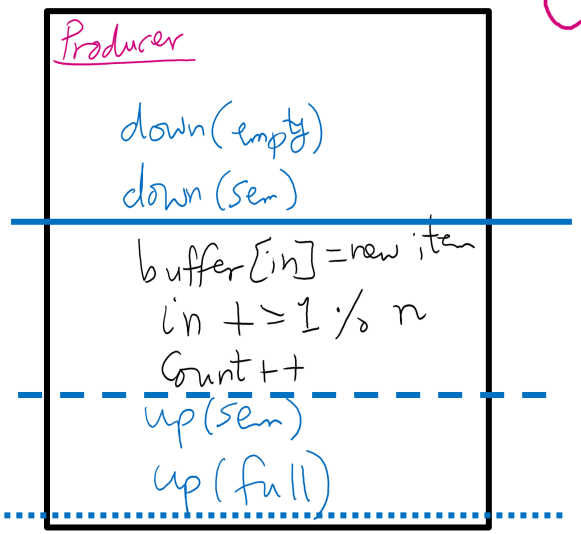
Initial state

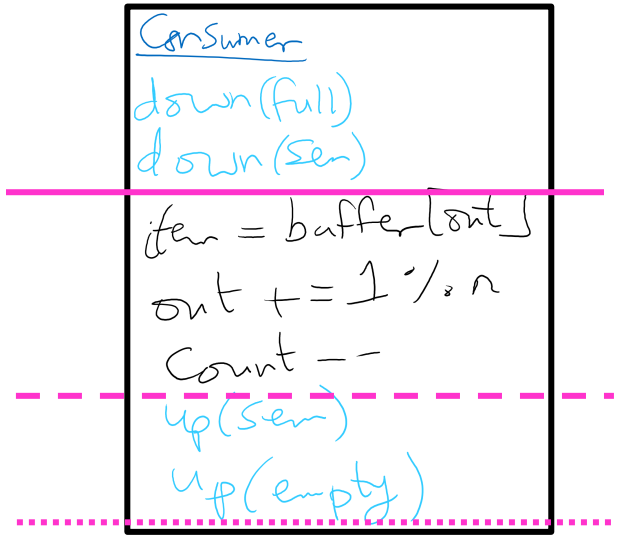


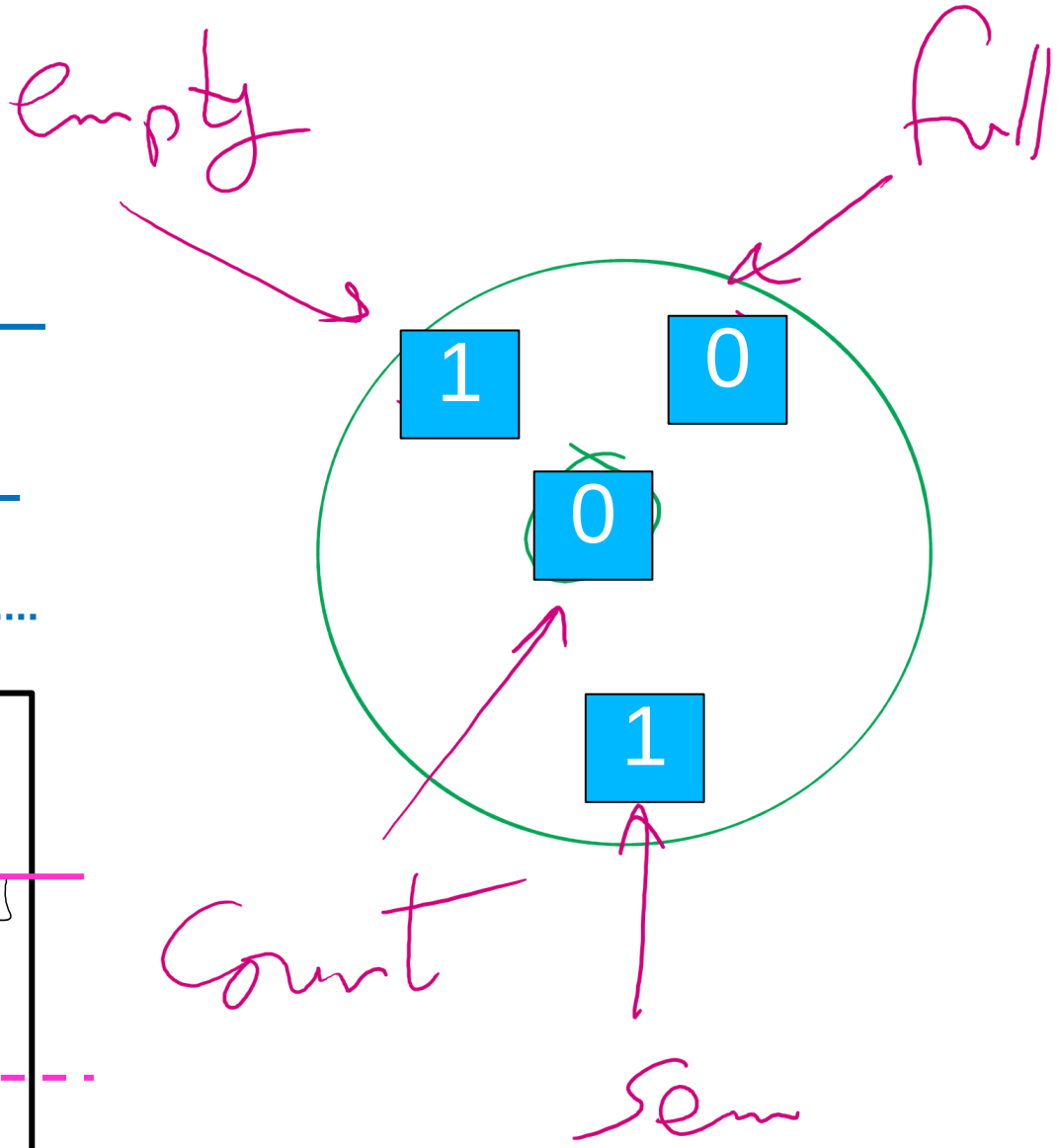
n == 1

Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves

C0 arrives



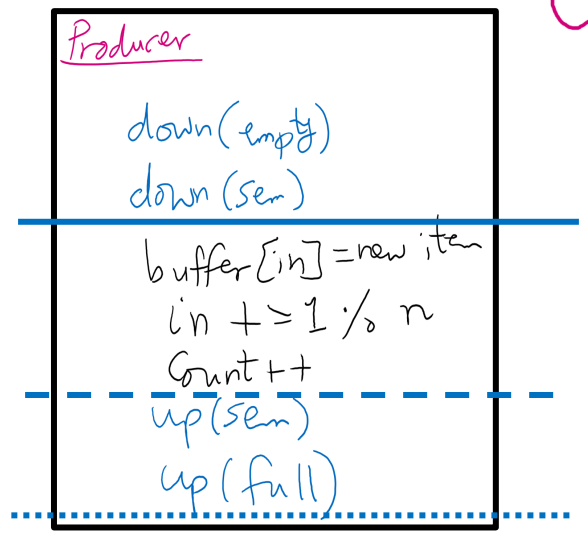


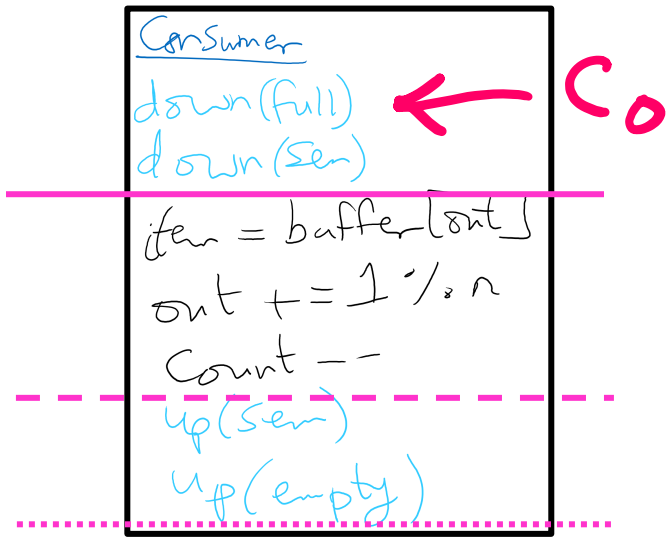


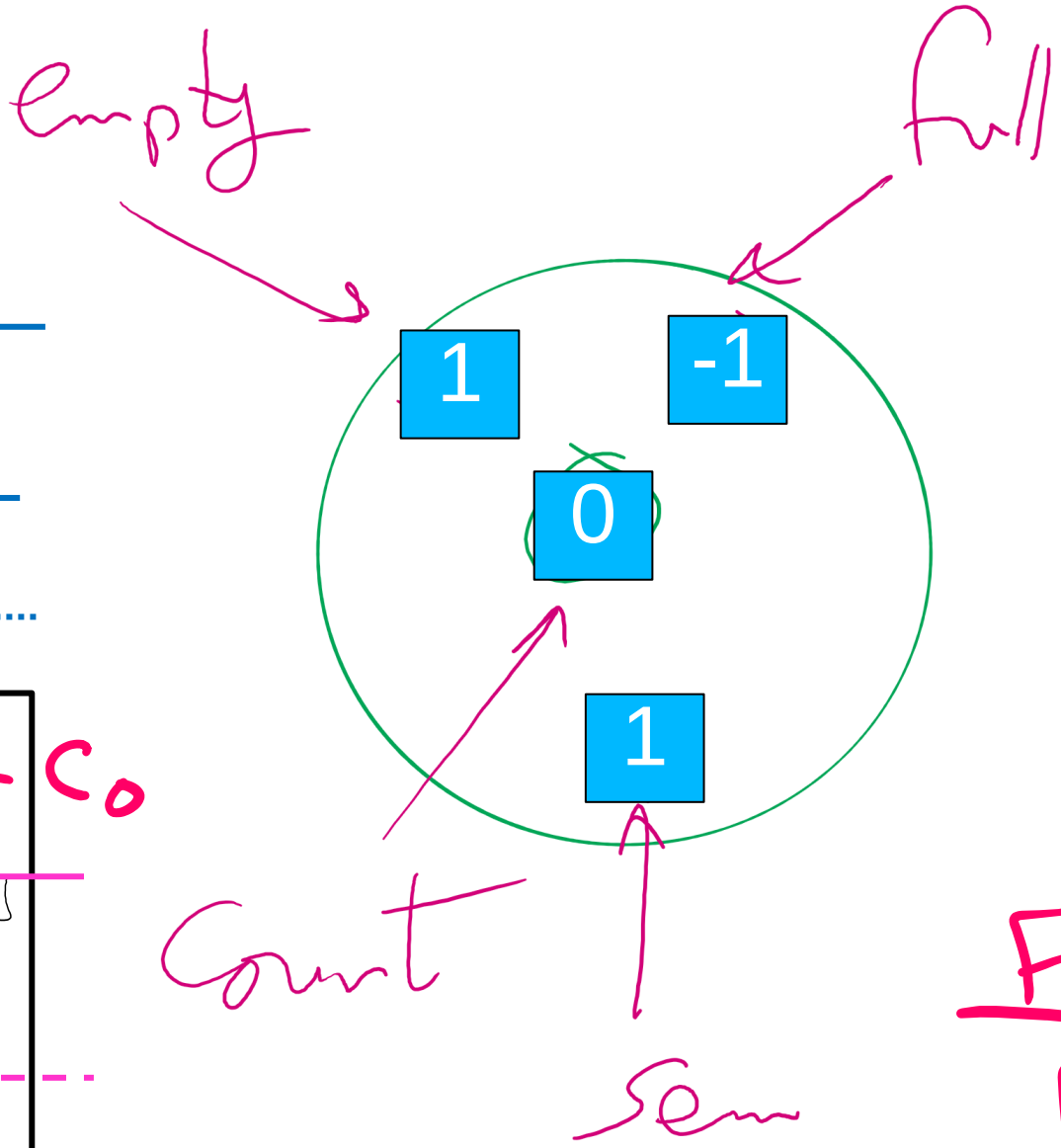
n == 1

Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves

C0 arrives







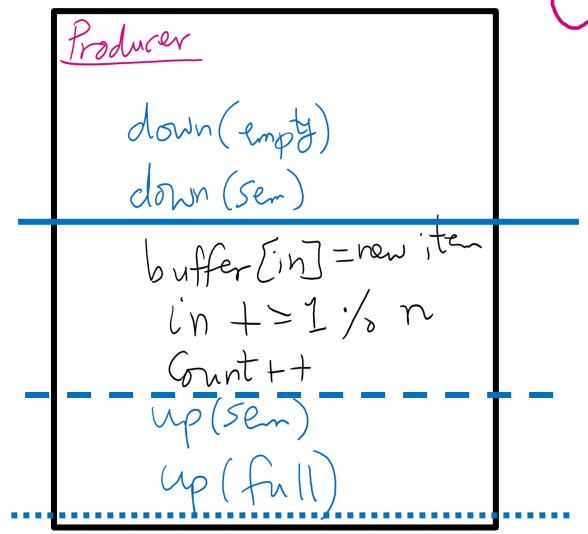
n == 1

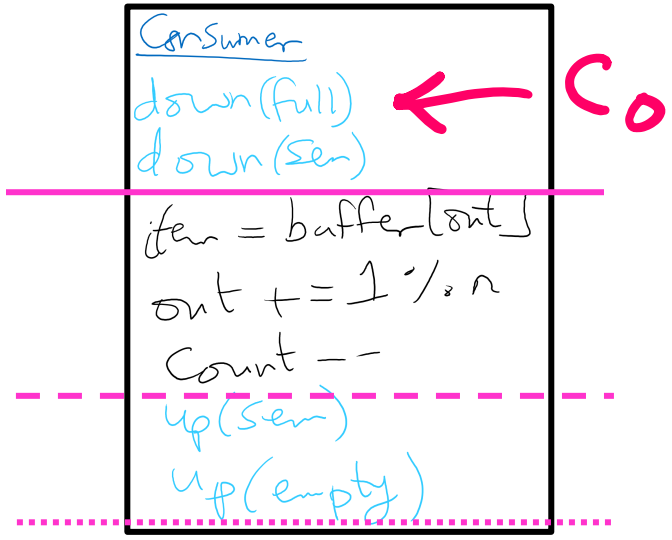
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves

Full Queue

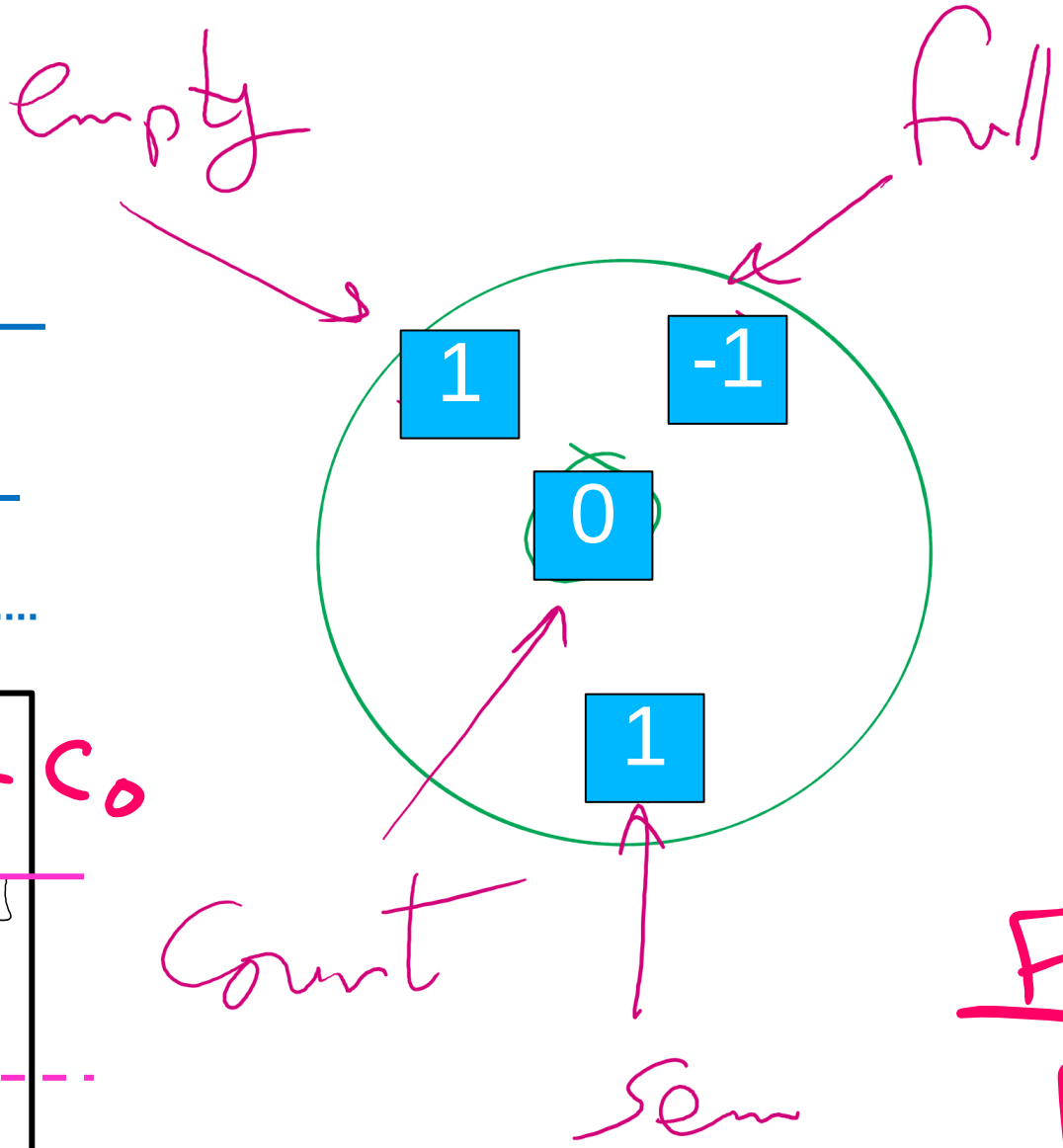
C0

P0 arrives





C0



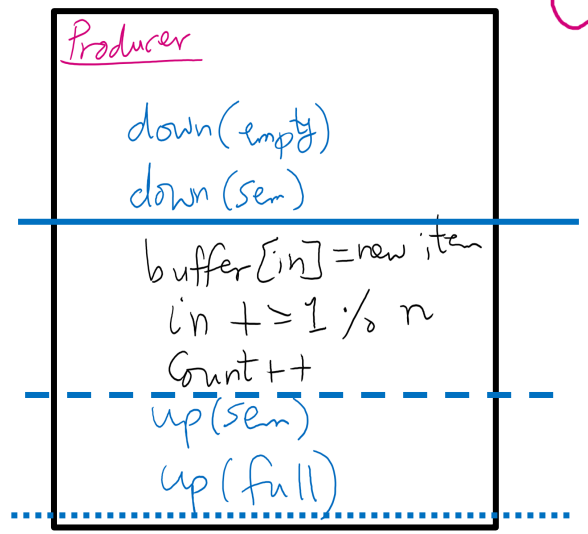
n == 1

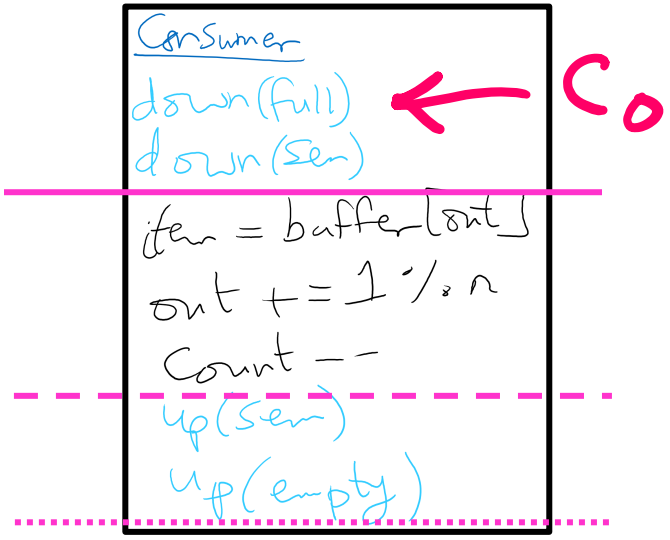
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves

Full Queue

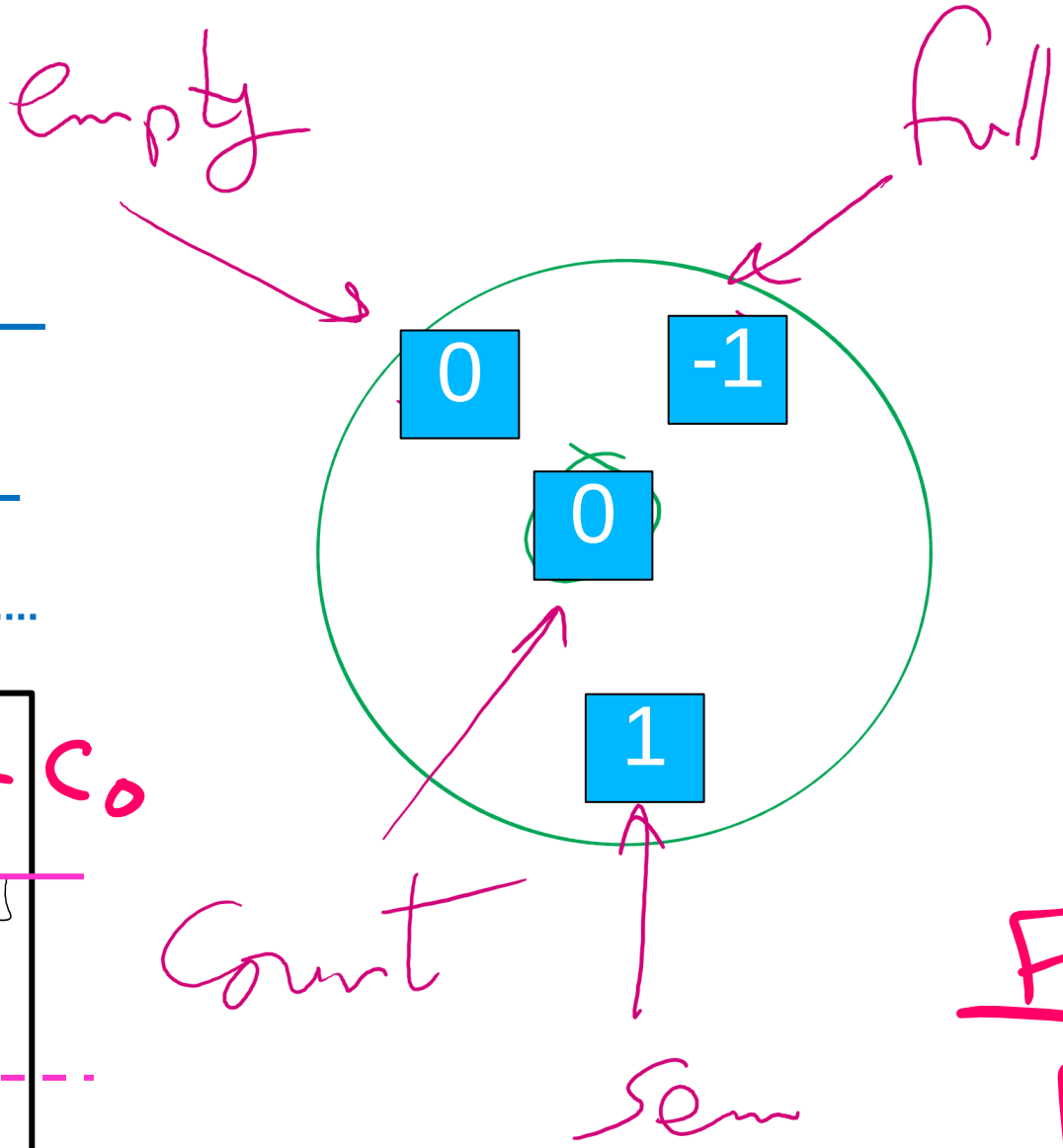
C0

P0 arrives





C0



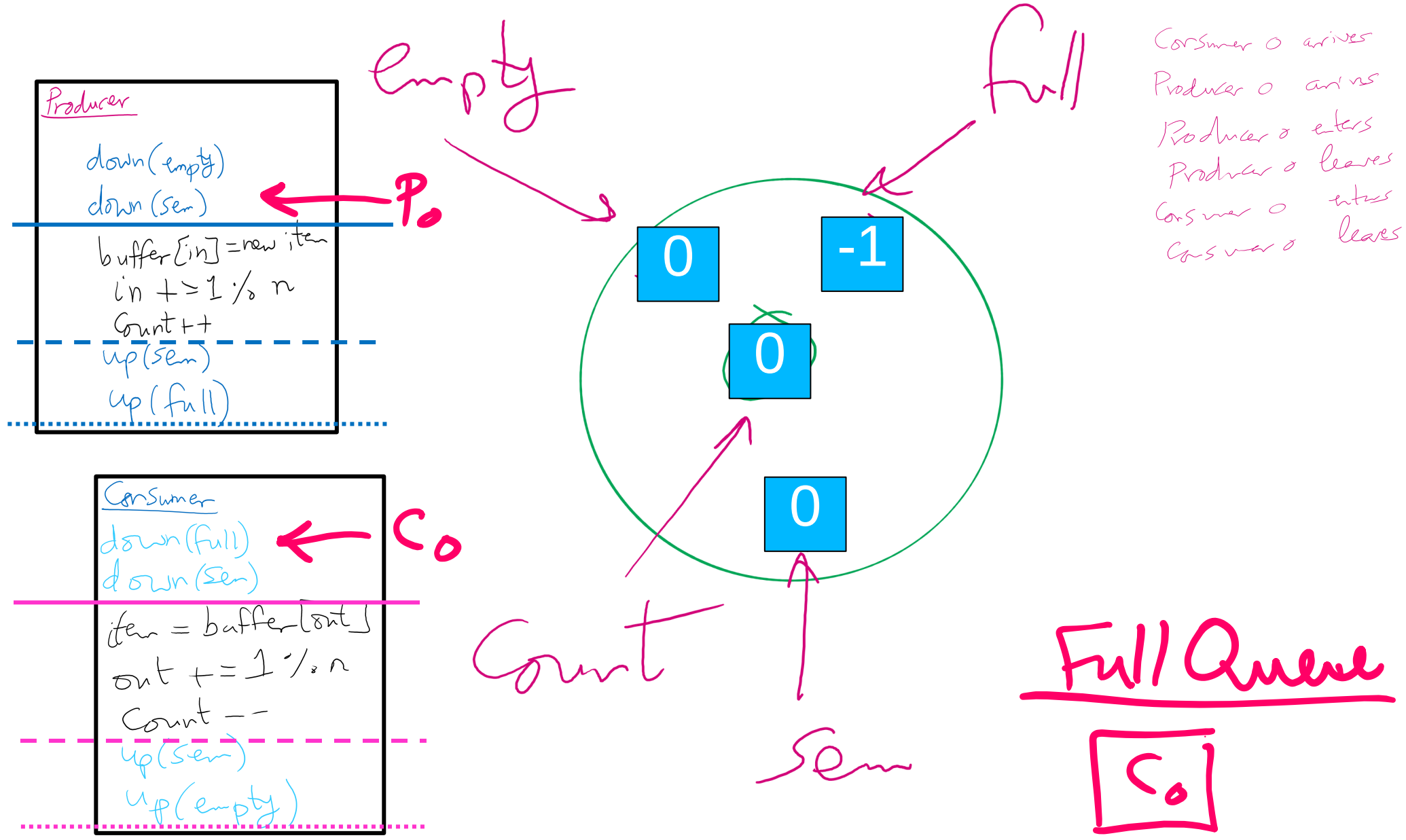
n == 1

Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves

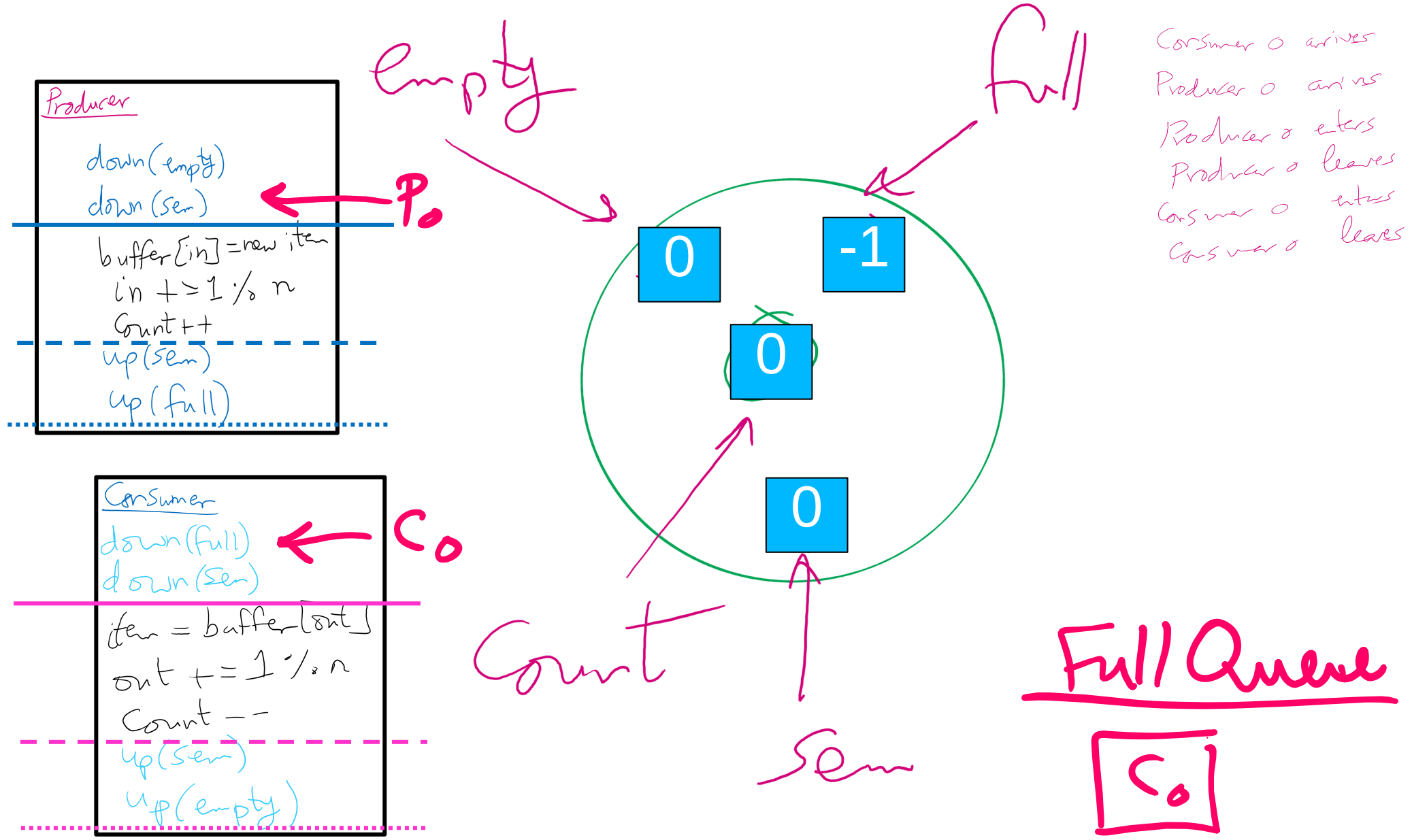
Full Queue

C0

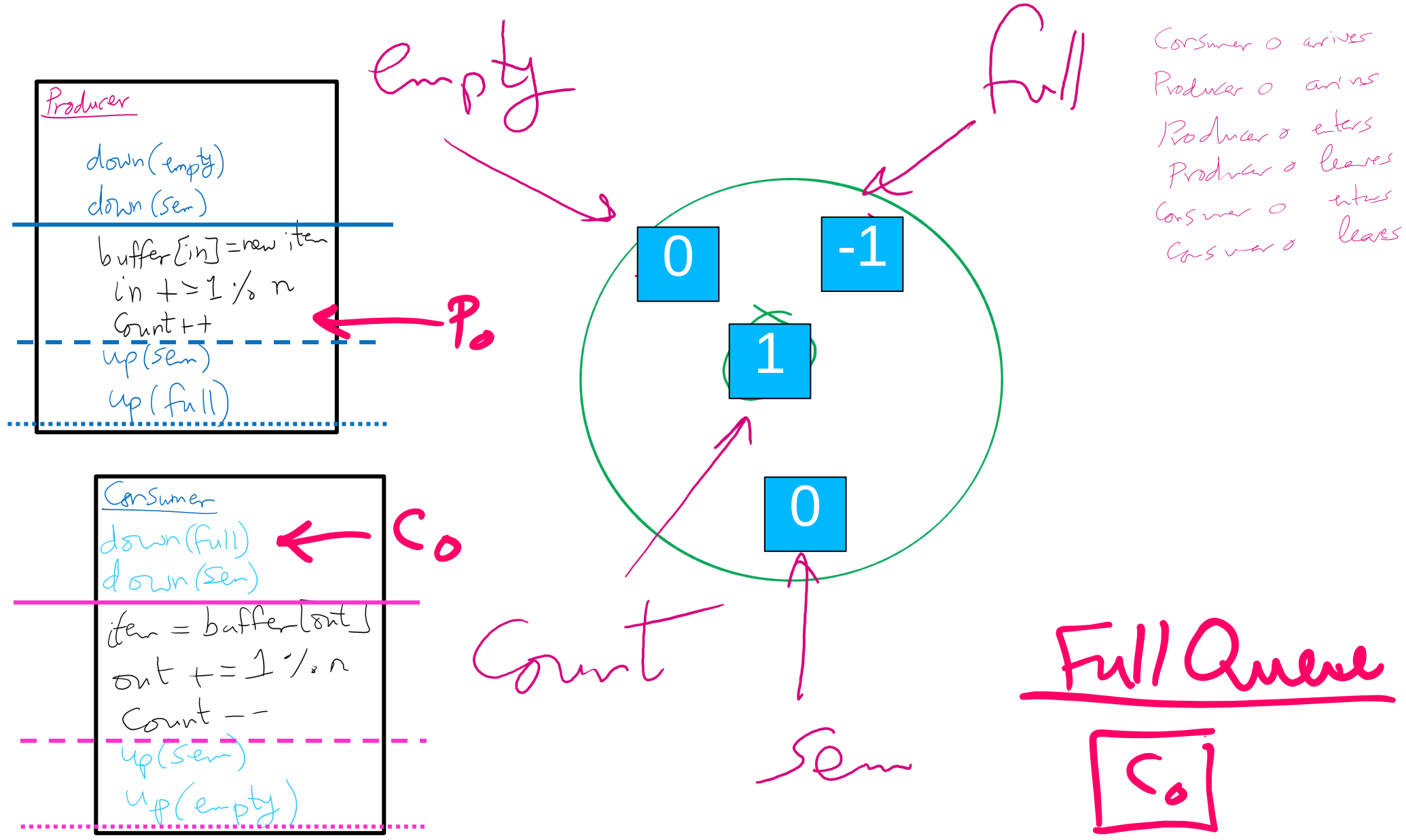
P0 arrives


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
0
-1
0
0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0
count
sem
Full Queue
C0

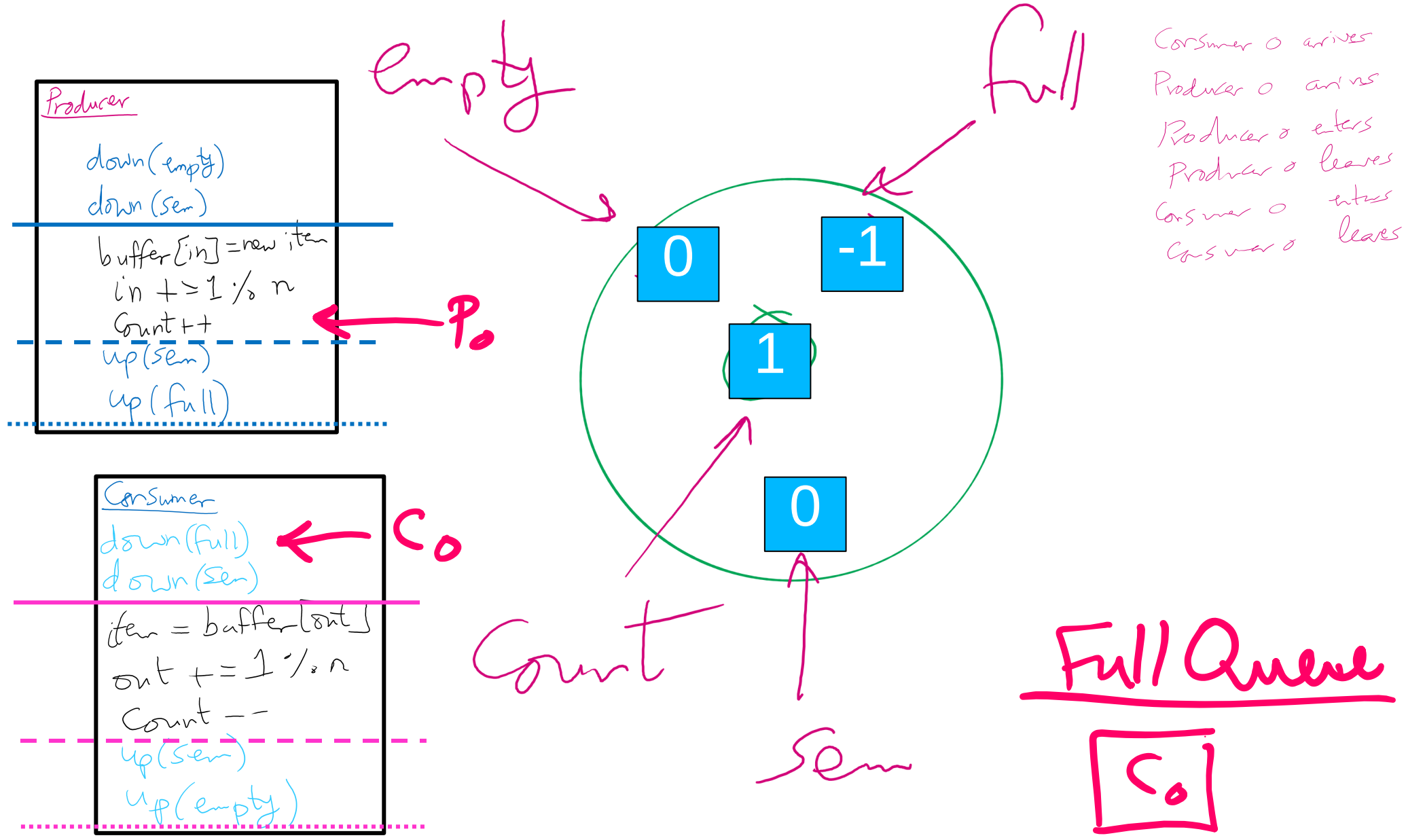
P0 enters


Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
Count++
up(sem)
up(full)
P0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
Count--
up(sem)
up(empty)
C0
empty
full
0
-1
0
0
Count
sem
n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
Full Queue
C0

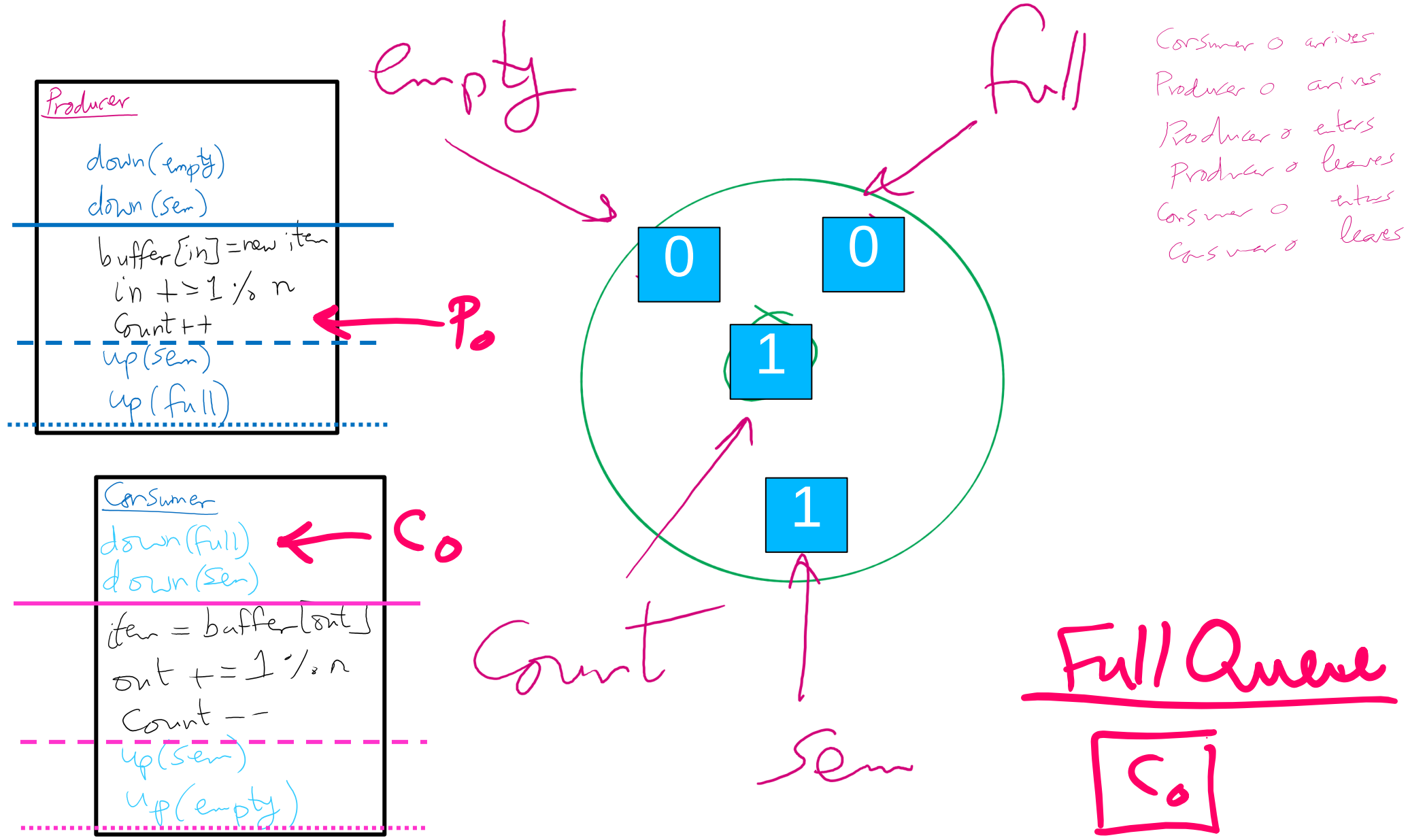
P0 enters


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
0
-1
1
0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0
count
sem
Full Queue
C0

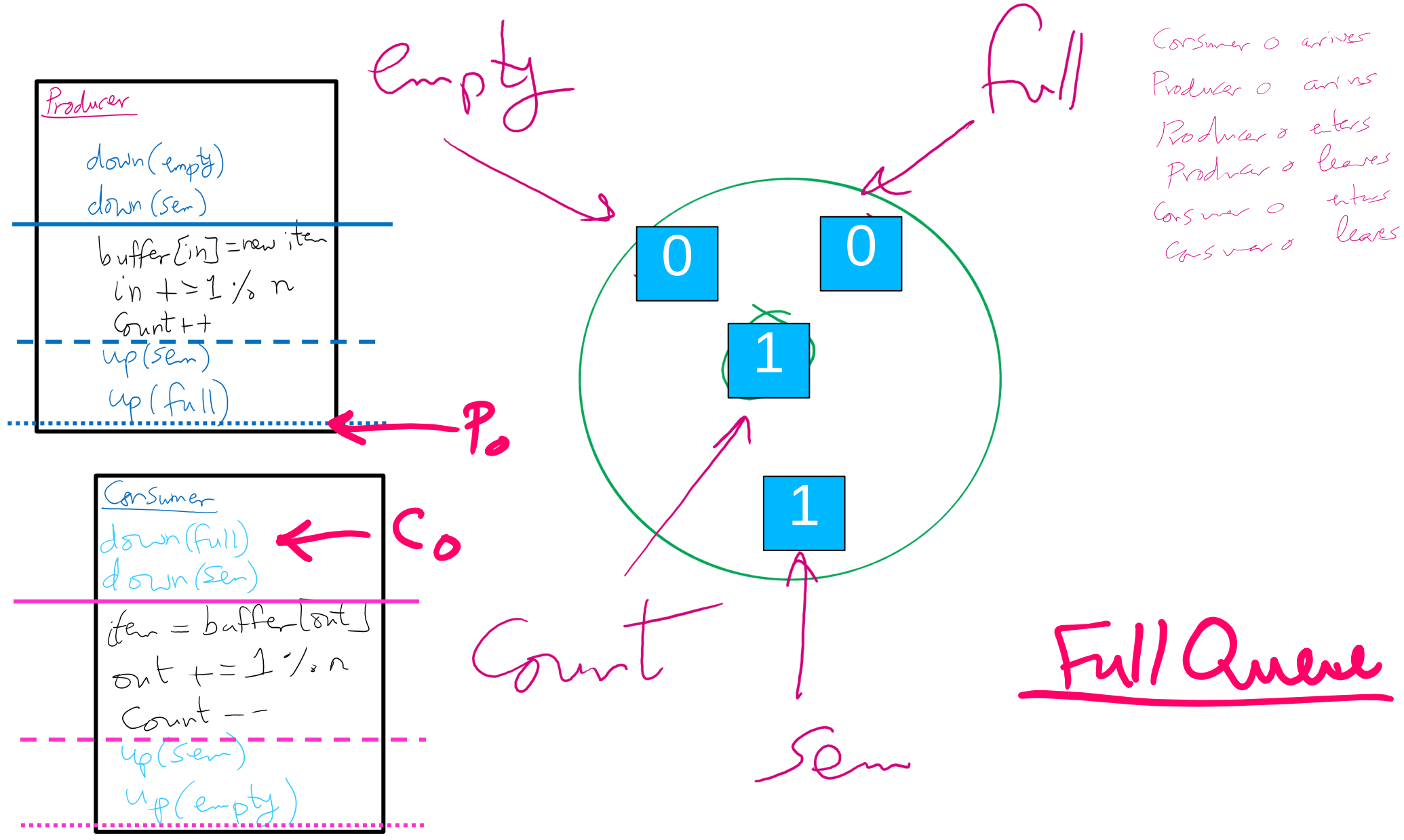
P0 leaves


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
0
-1
1
0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0
count
sem
Full Queue
C0

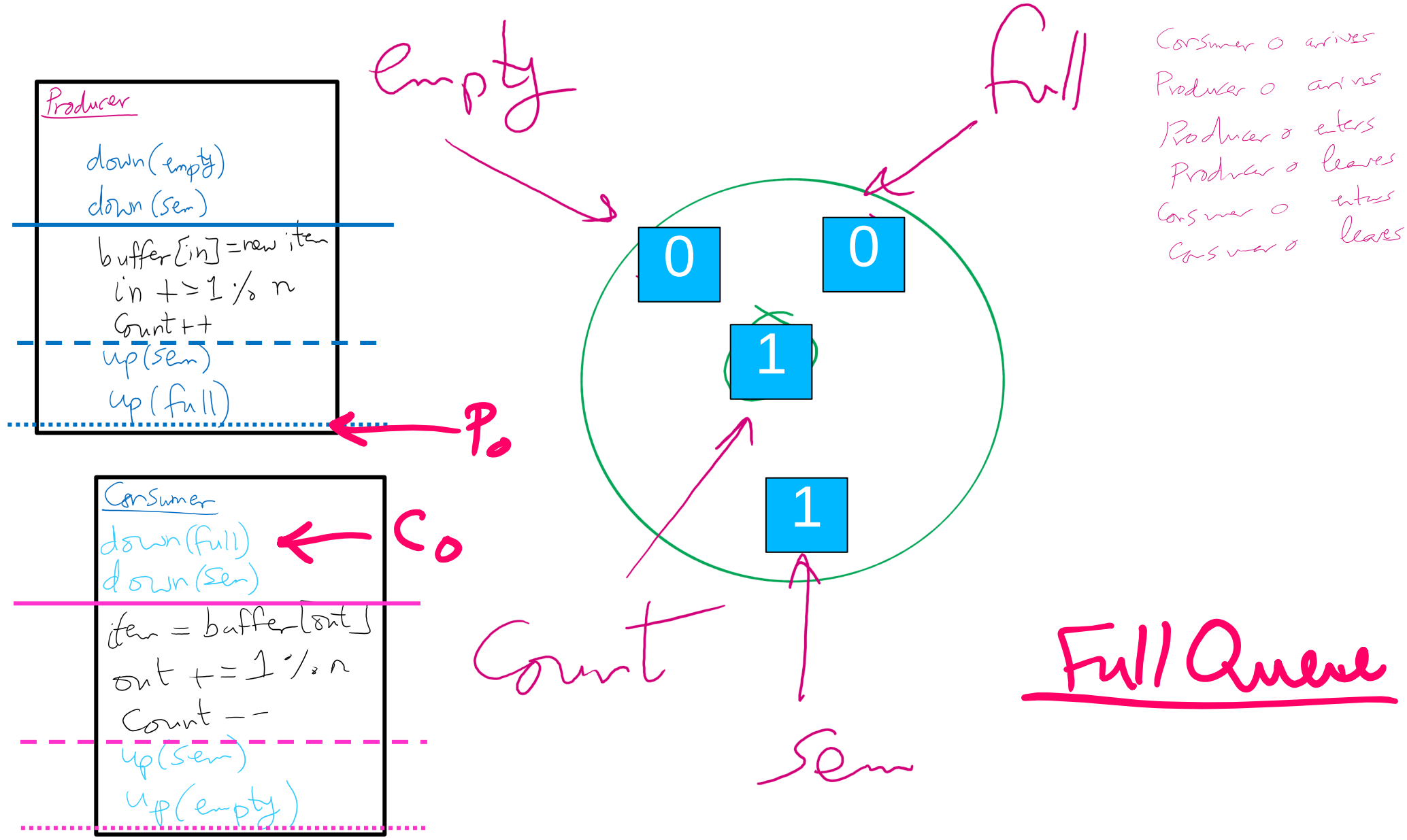
P0 leaves


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
Count++
up(sem)
up(full)
P0
0
0
1
1
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
Count--
up(sem)
up(empty)
C0
Count
sem
Full Queue
C0

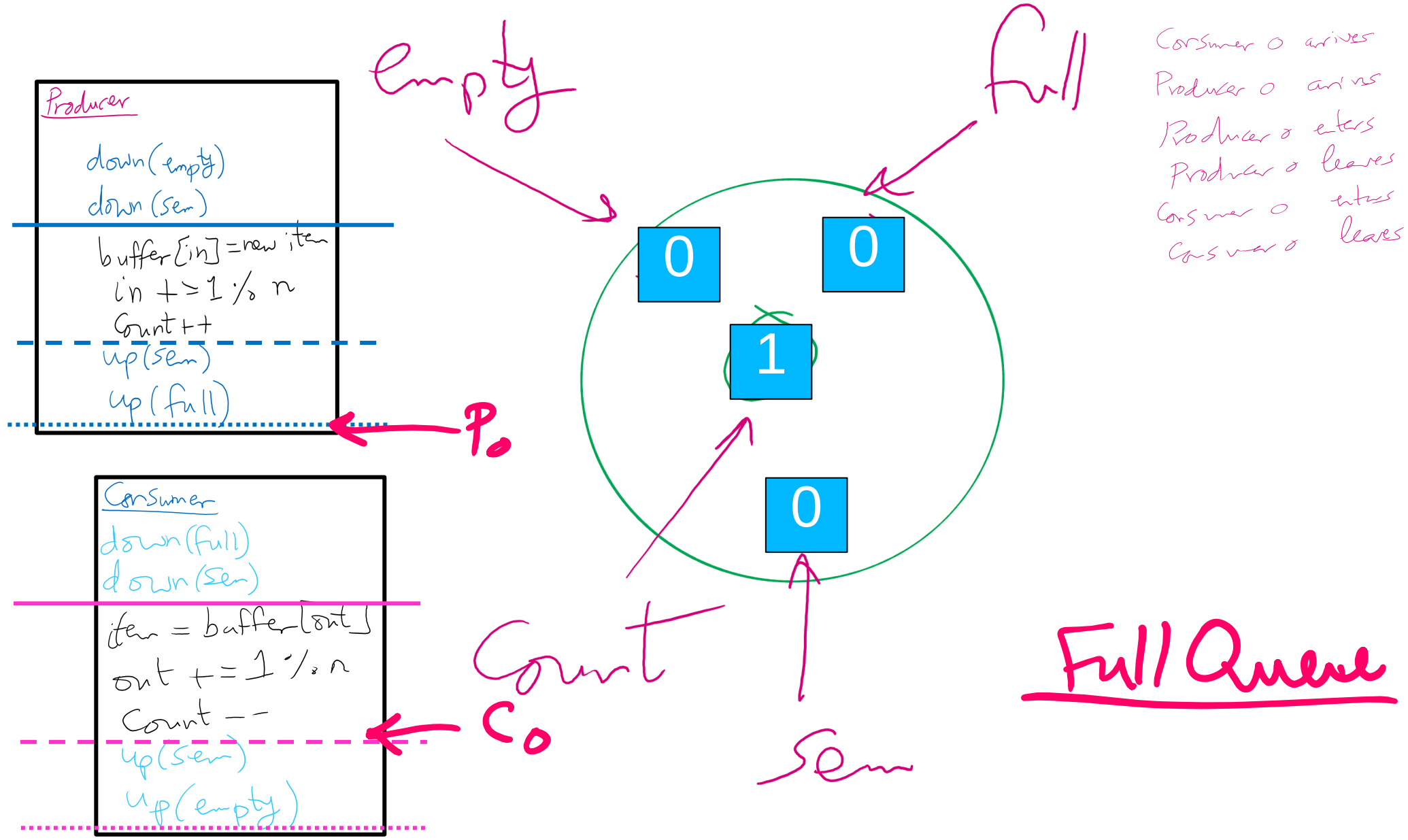
P0 leaves


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
Count++
up(sem)
up(full)
P0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
Count--
up(sem)
up(empty)
C0
0
0
1
1
Count
sem
Full Queue

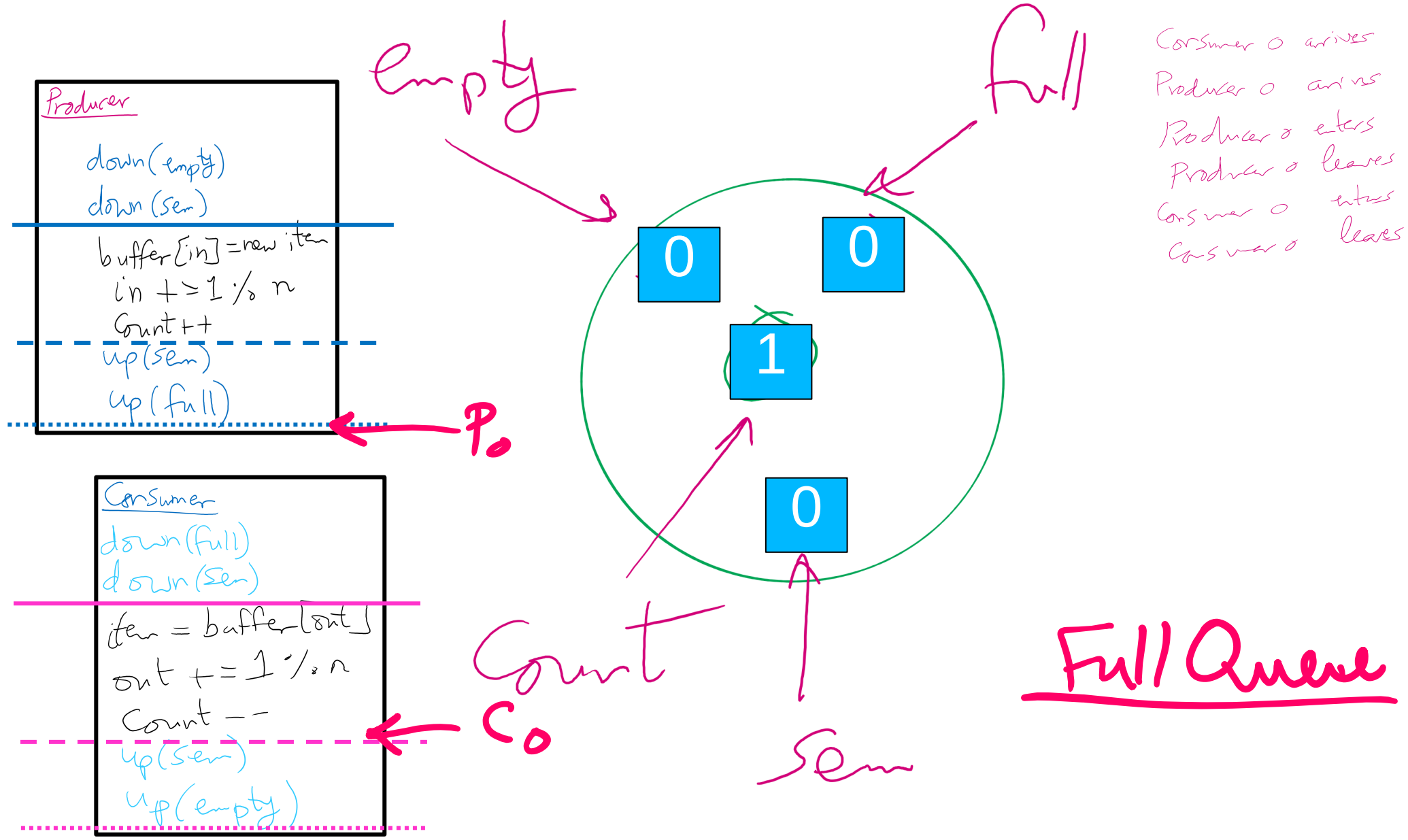
C0 enters


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
Count++
up(sem)
up(full)
P0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
Count--
up(sem)
up(empty)
C0
0
0
1
1
Count
sem
Full Queue

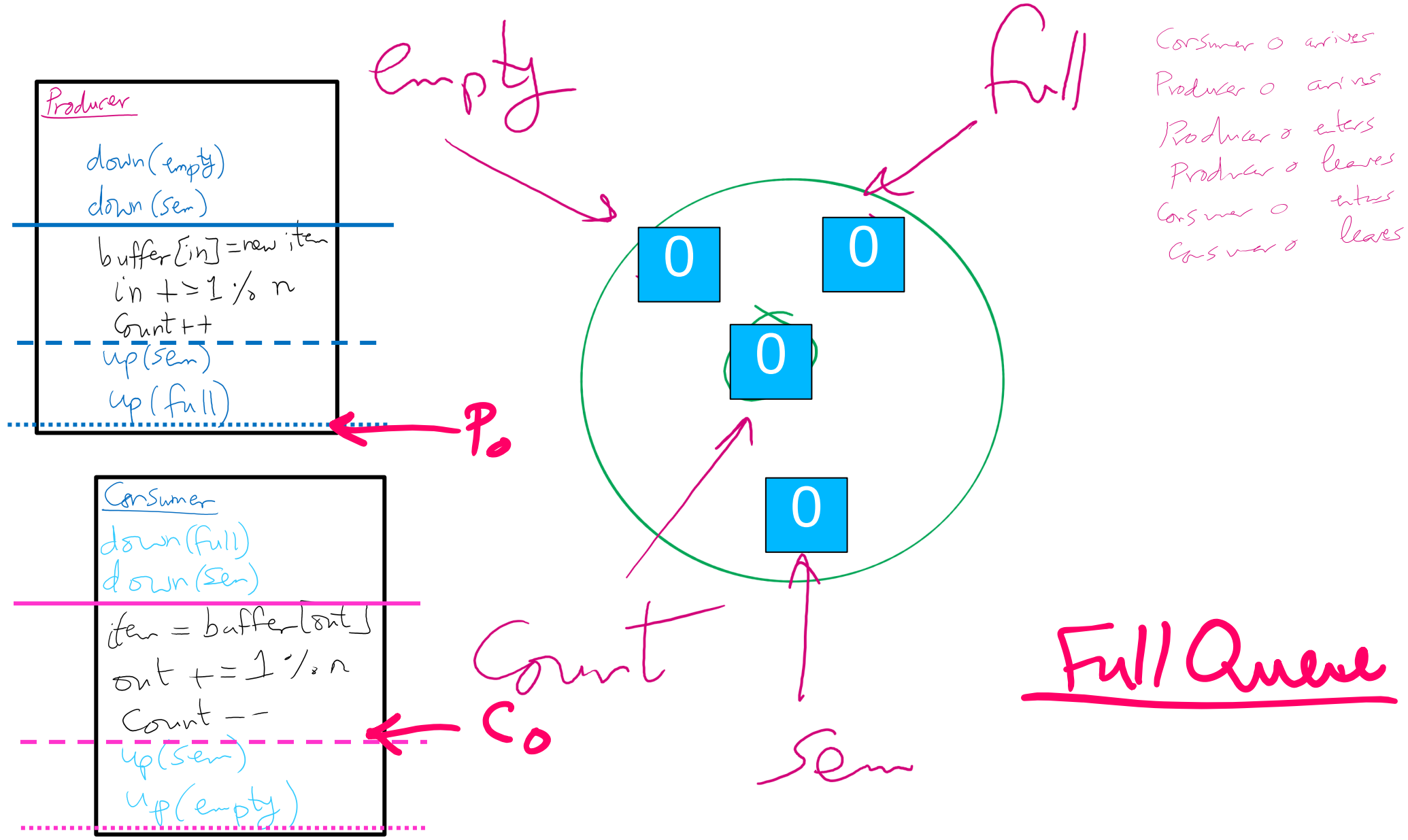
C0 enters


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0
empty
full
0
0
1
0
count
sem
Full Queue

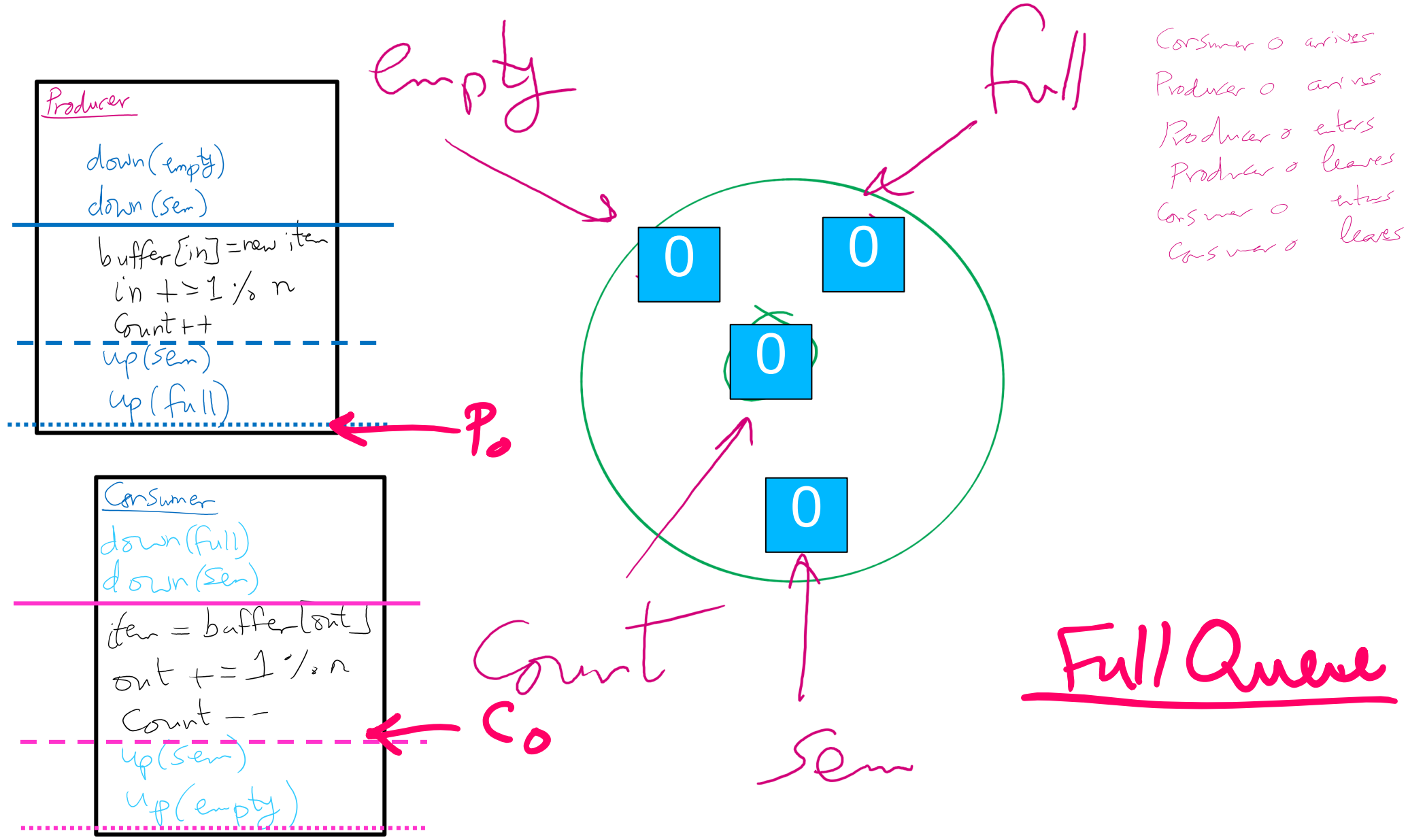
Can C0 enter?


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
0
0
1
0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0
count
sem
Full Queue

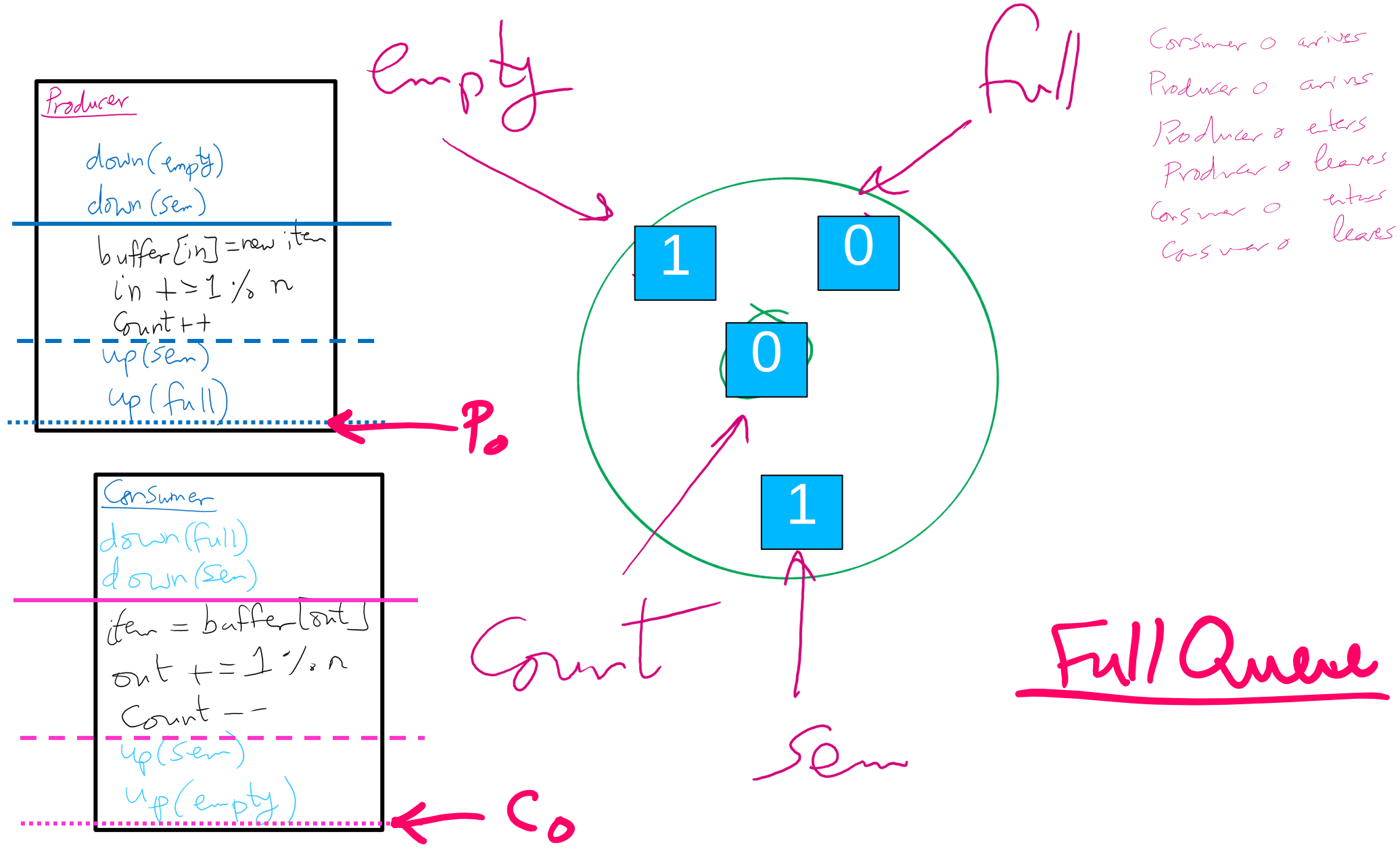
C0 enters


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
0
0
0
0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0
count
sem
Full Queue

C0 leaves


Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0
empty
full
0
0
0
0
count
sem
n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
Full Queue

C0 leaves


n == 1
Consumer 0 arrives
Producer 0 arrives
Producer 0 enters
Producer 0 leaves
Consumer 0 enters
Consumer 0 leaves
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
Count++
up(sem)
up(full)
P0
1
0
0
1
Count
sem
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
Count--
up(sem)
up(empty)
C0
Full Queue

Example 3

$n == 2$

C_0 arrives

C_1 arrives

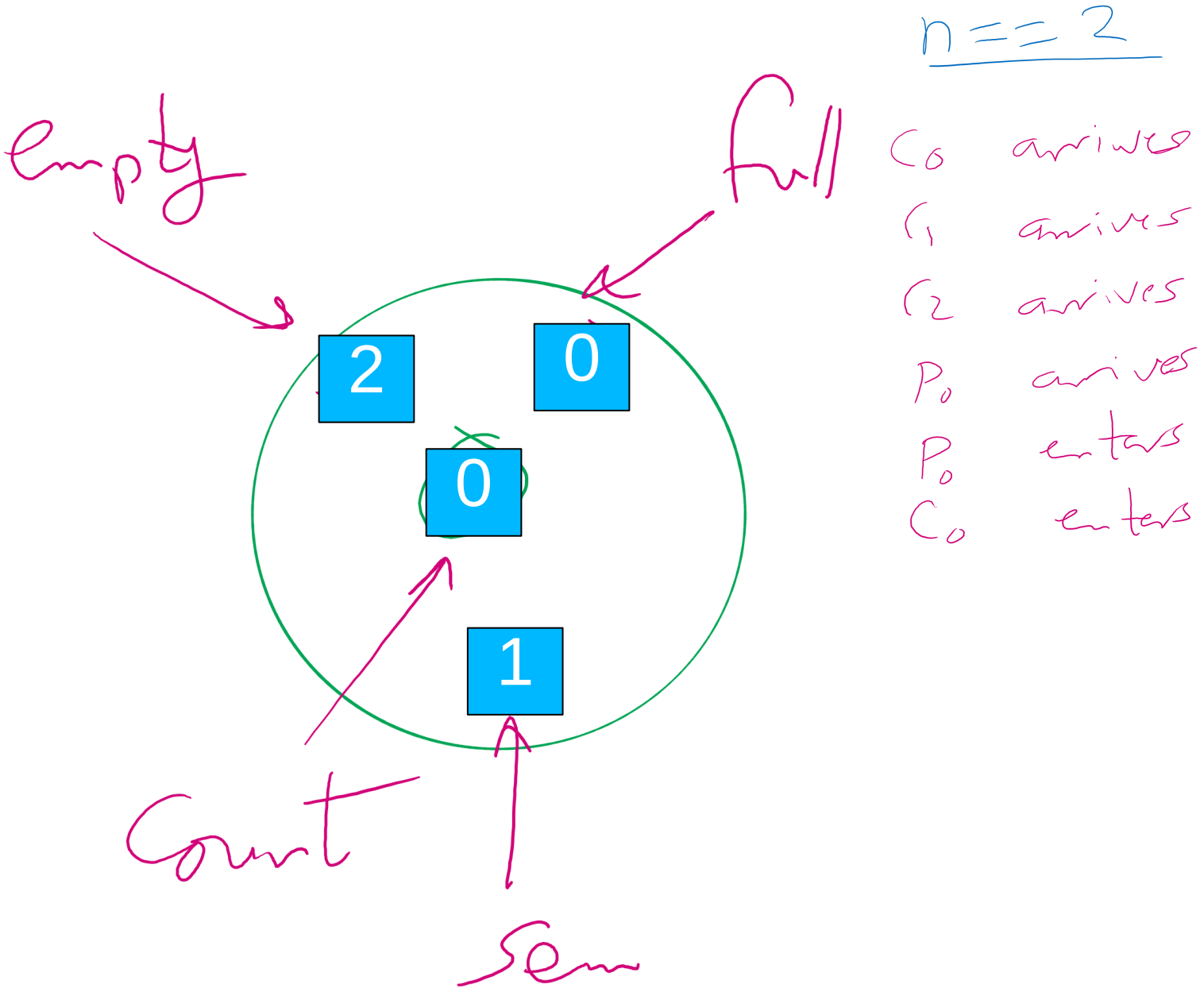
C_2 arrives

P_0 arrives

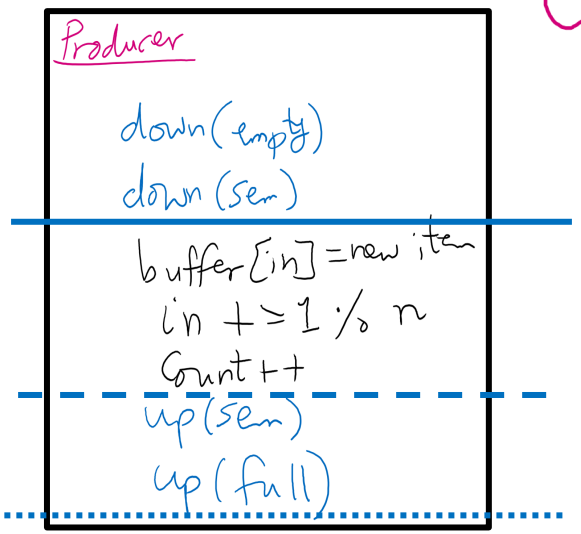
P_0 enters

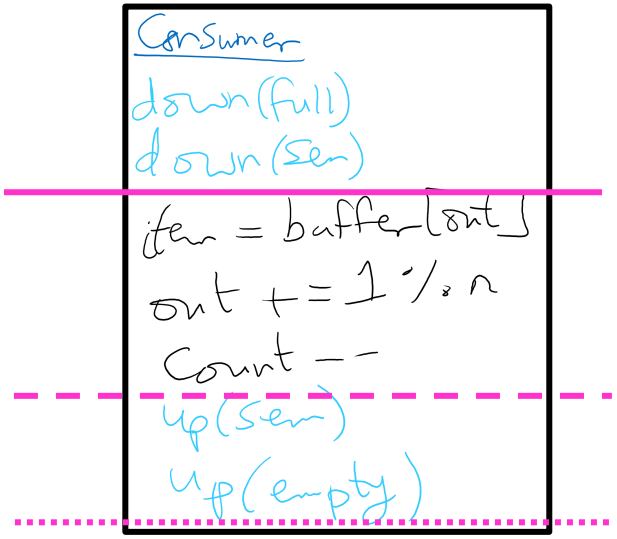
C_0 enters

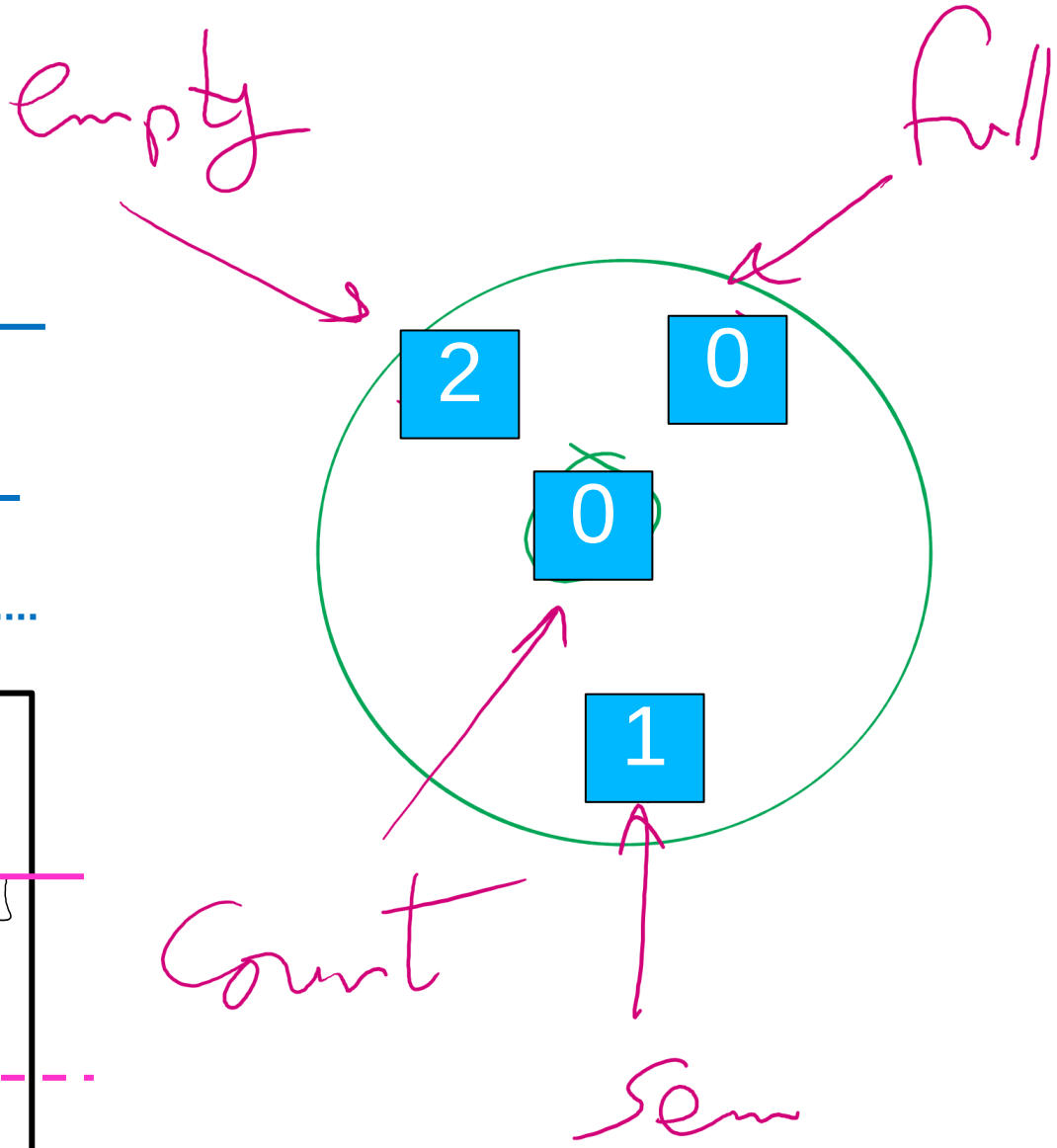
Initial state



C0 arrives



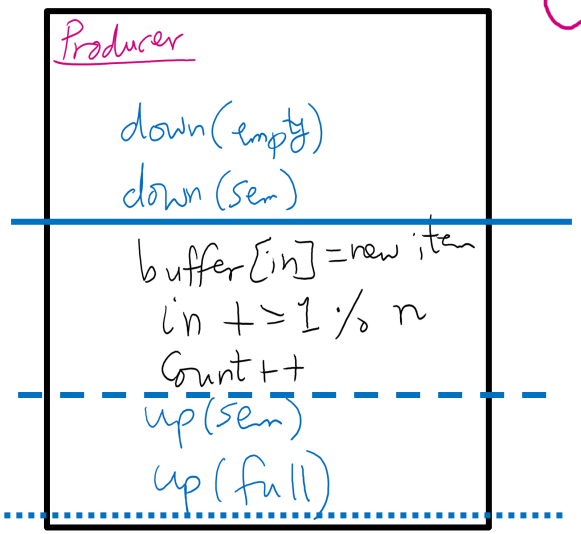


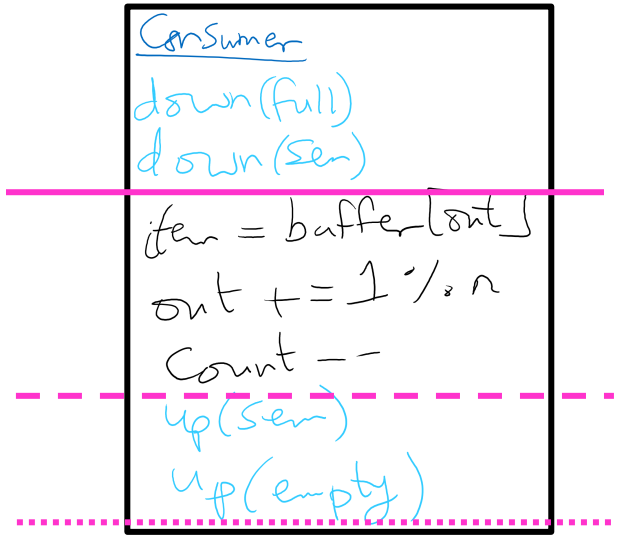


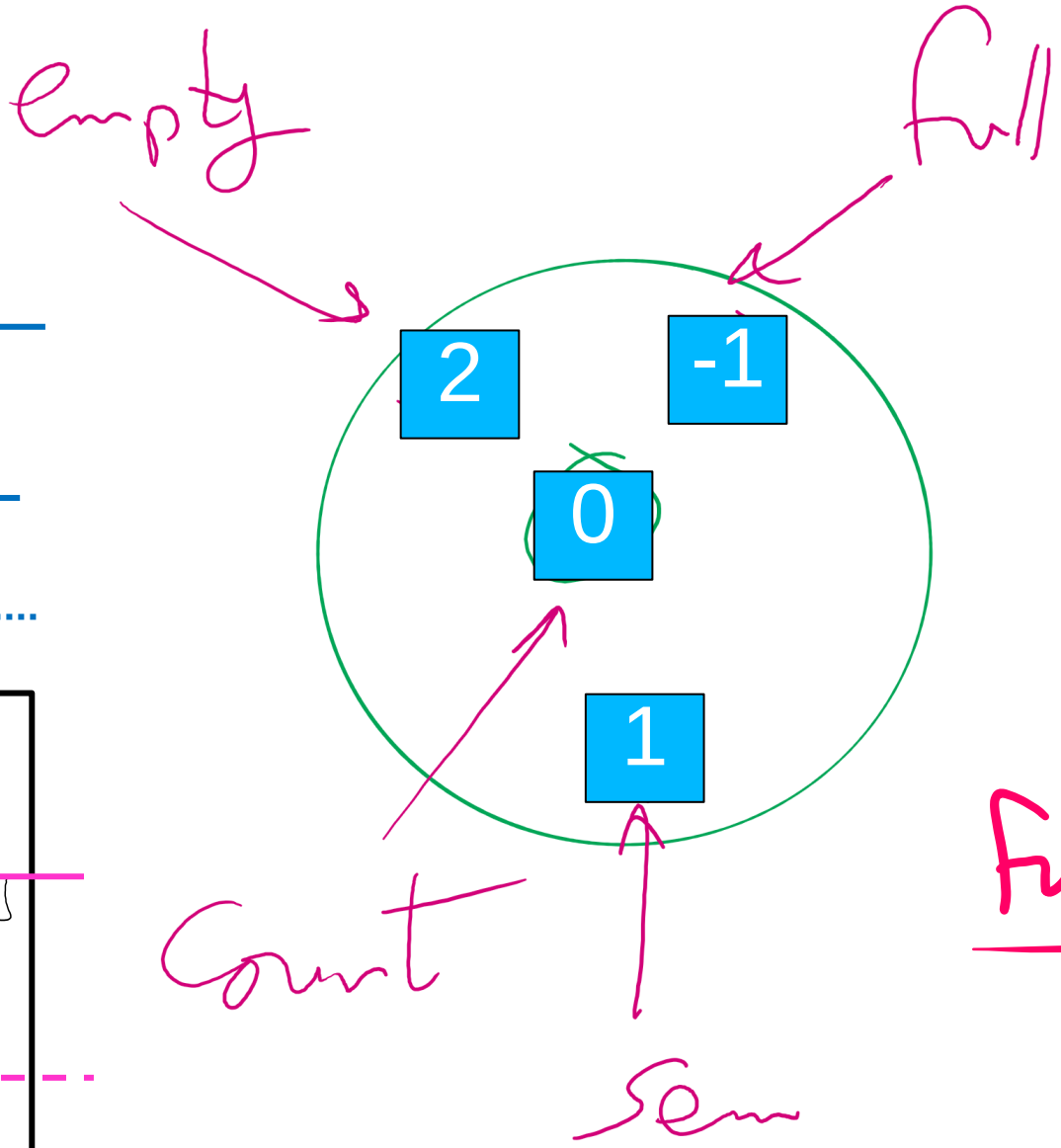
n == 2

C_0 arrives
C_1 arrives
C_2 arrives
P_0 arrives
P_0 enters
C_0 enters

C0 arrives



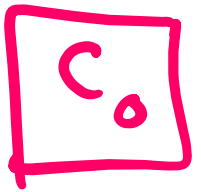




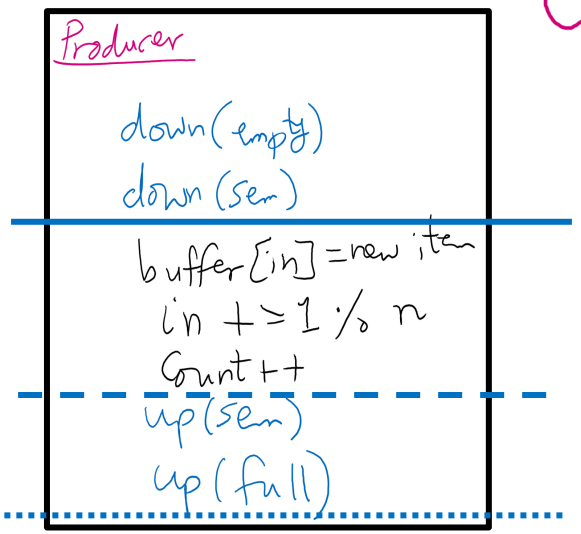
n == 2

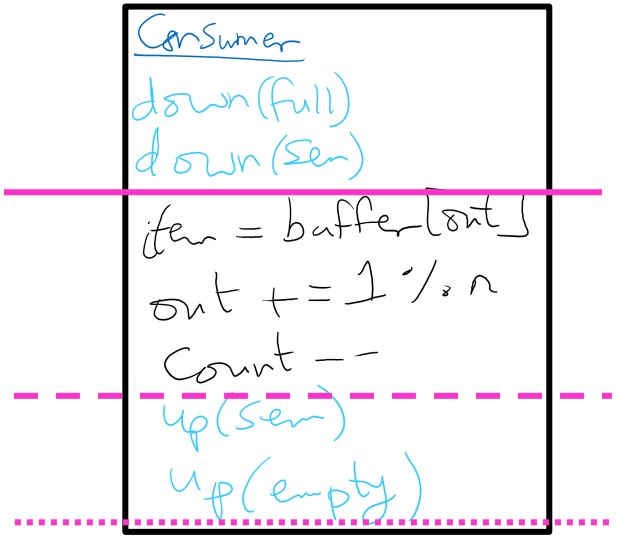
C0 arrives
C1 arrives
C2 arrives
P0 arrives
P0 enters
C0 enters

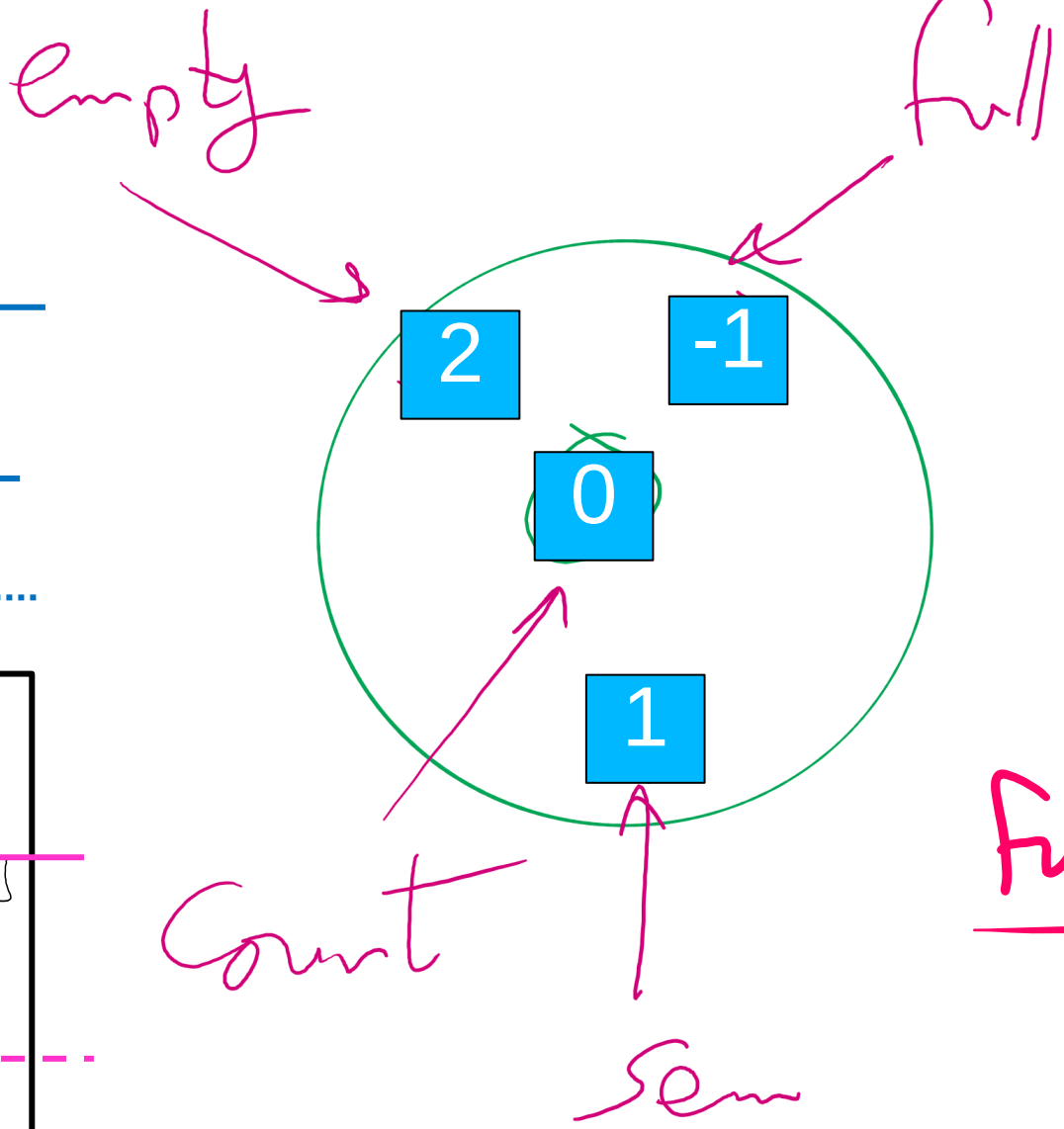
Full Queue



C1 arrives



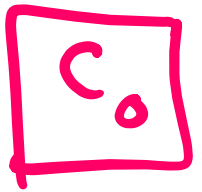




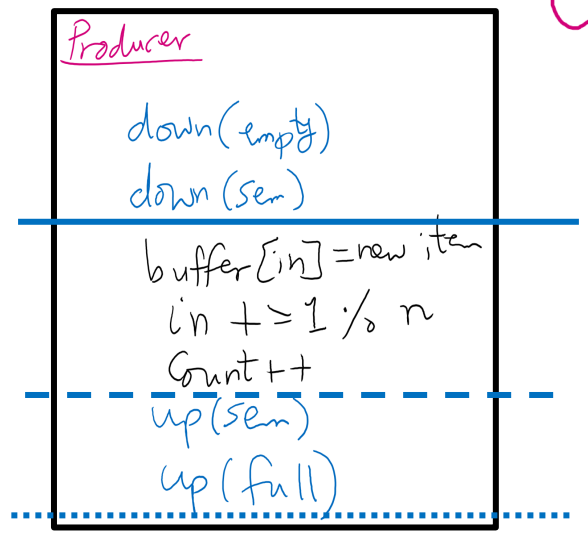
n == 2

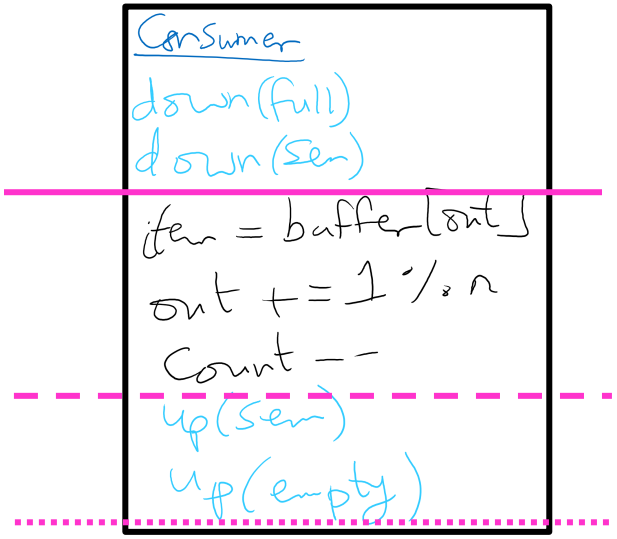
C_0 arrives
C_1 arrives
C_2 arrives
P_0 arrives
P_0 enters
C_0 enters

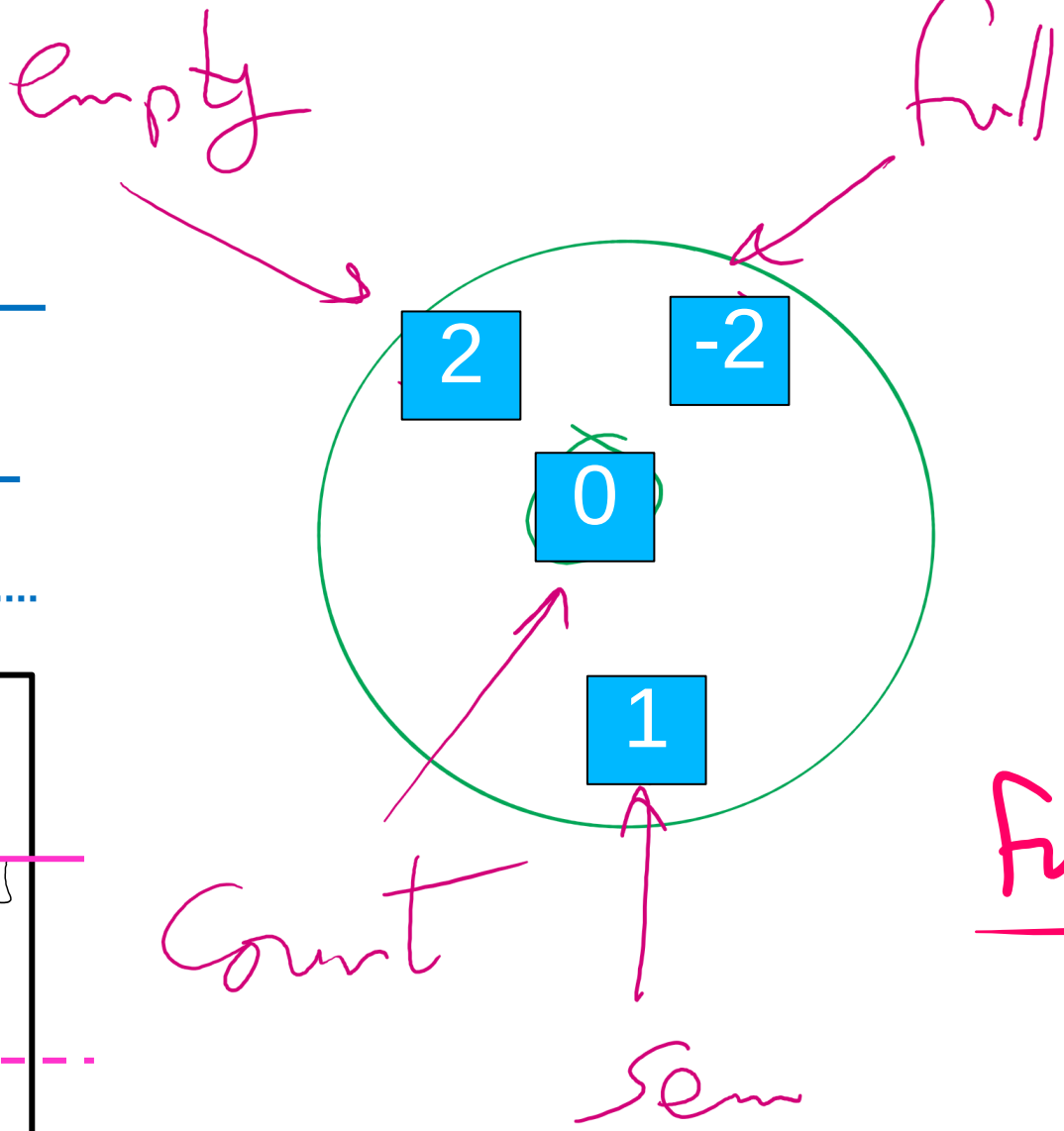
Full Queue



C1 arrives



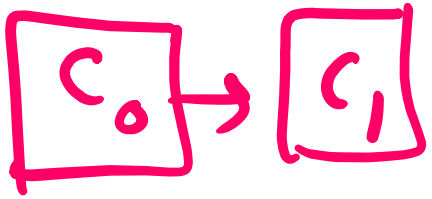




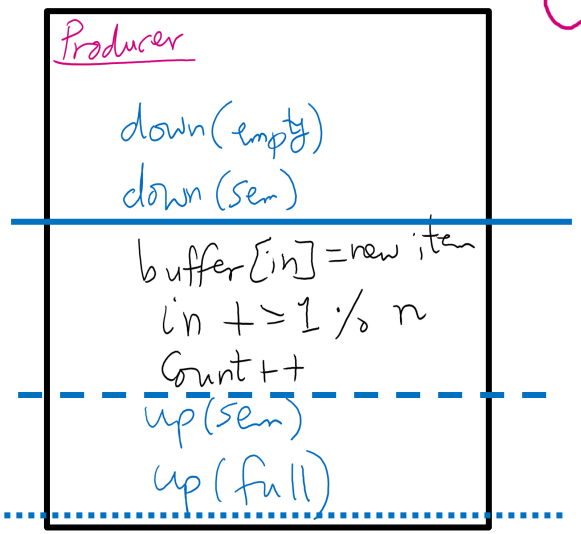
n == 2

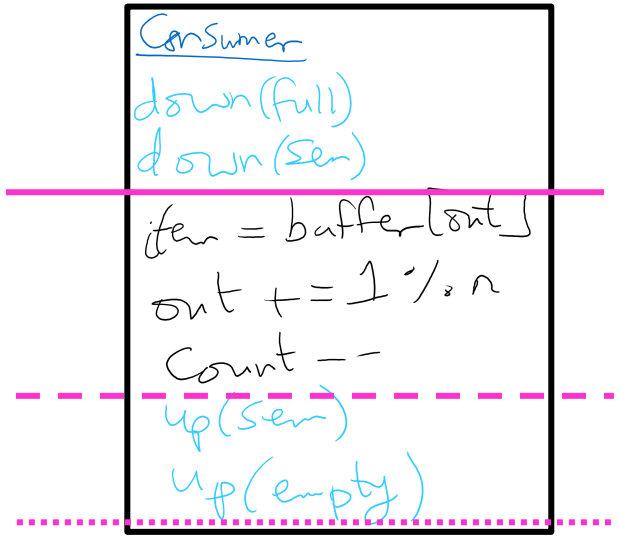
C_0 arrives
C_1 arrives
C_2 arrives
P_0 arrives
P_0 enters
C_0 enters

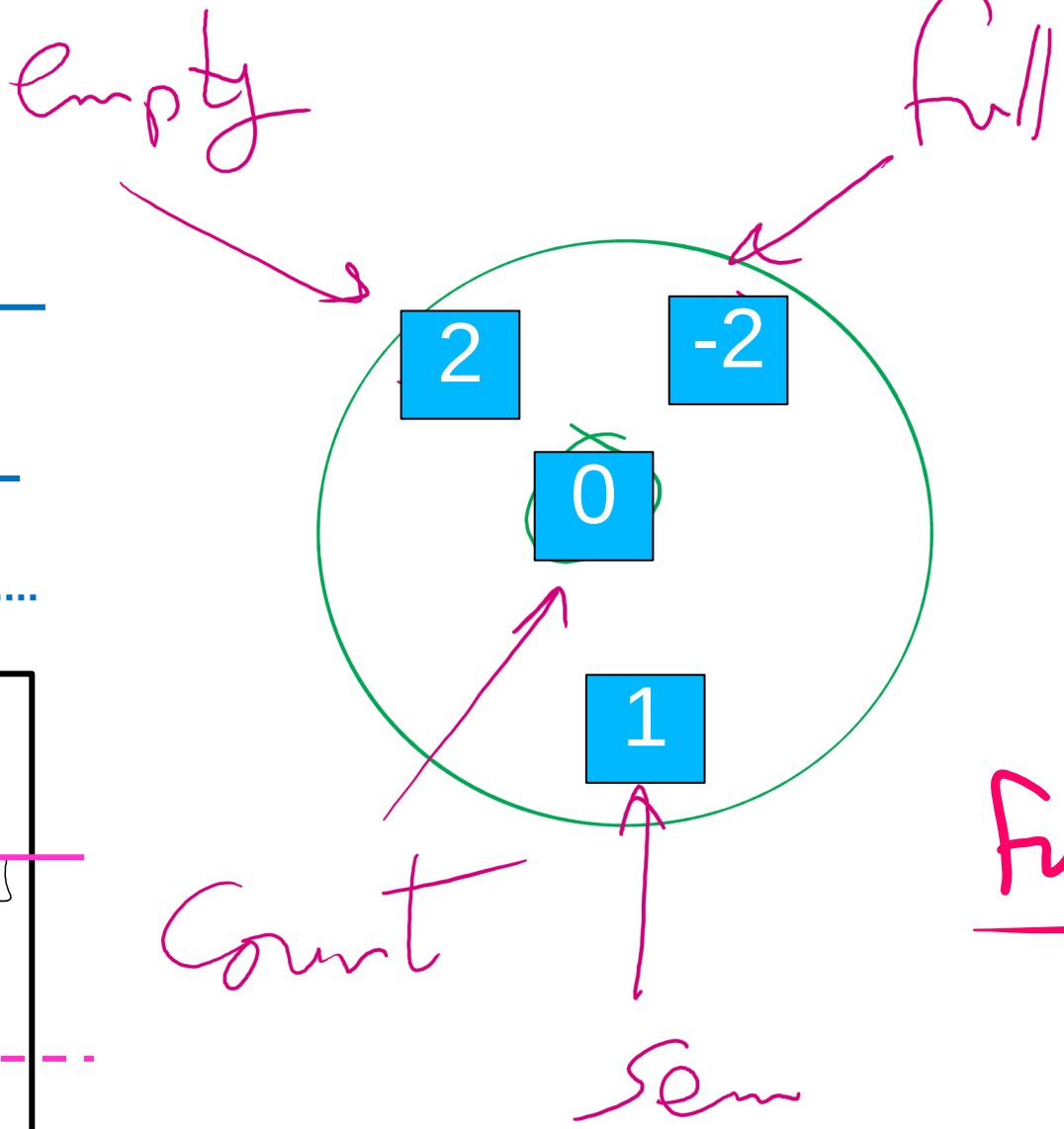
Full Queue



C2 arrives



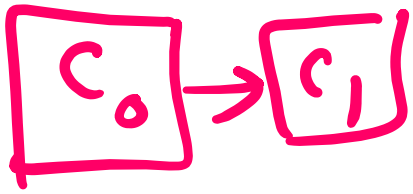




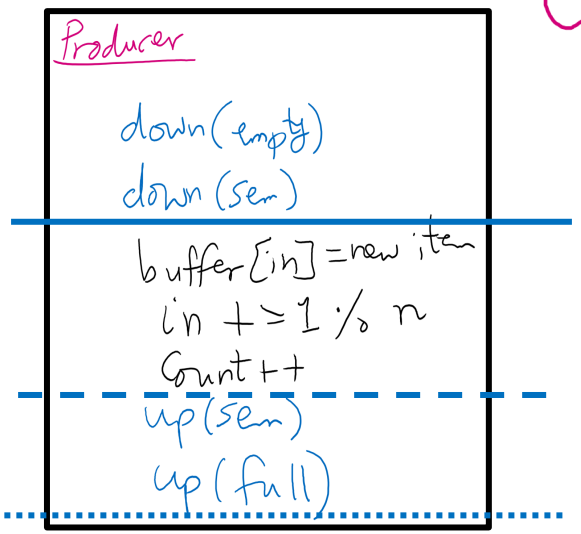
n == 2

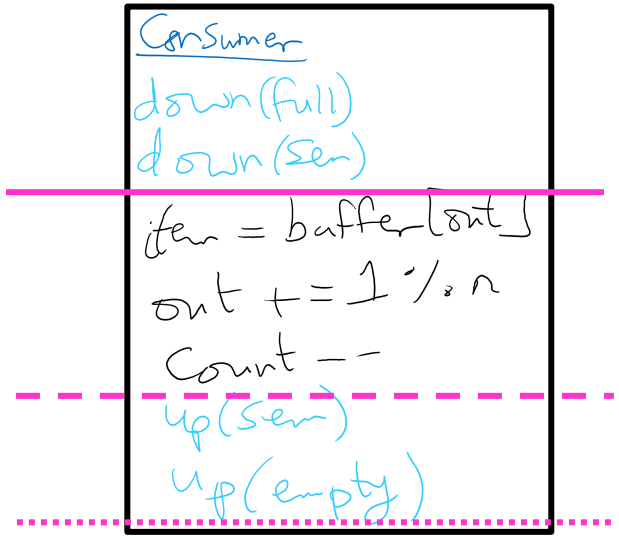
C_0 arrives
C_1 arrives
C_2 arrives
P_0 arrives
P_0 enters
C_0 enters

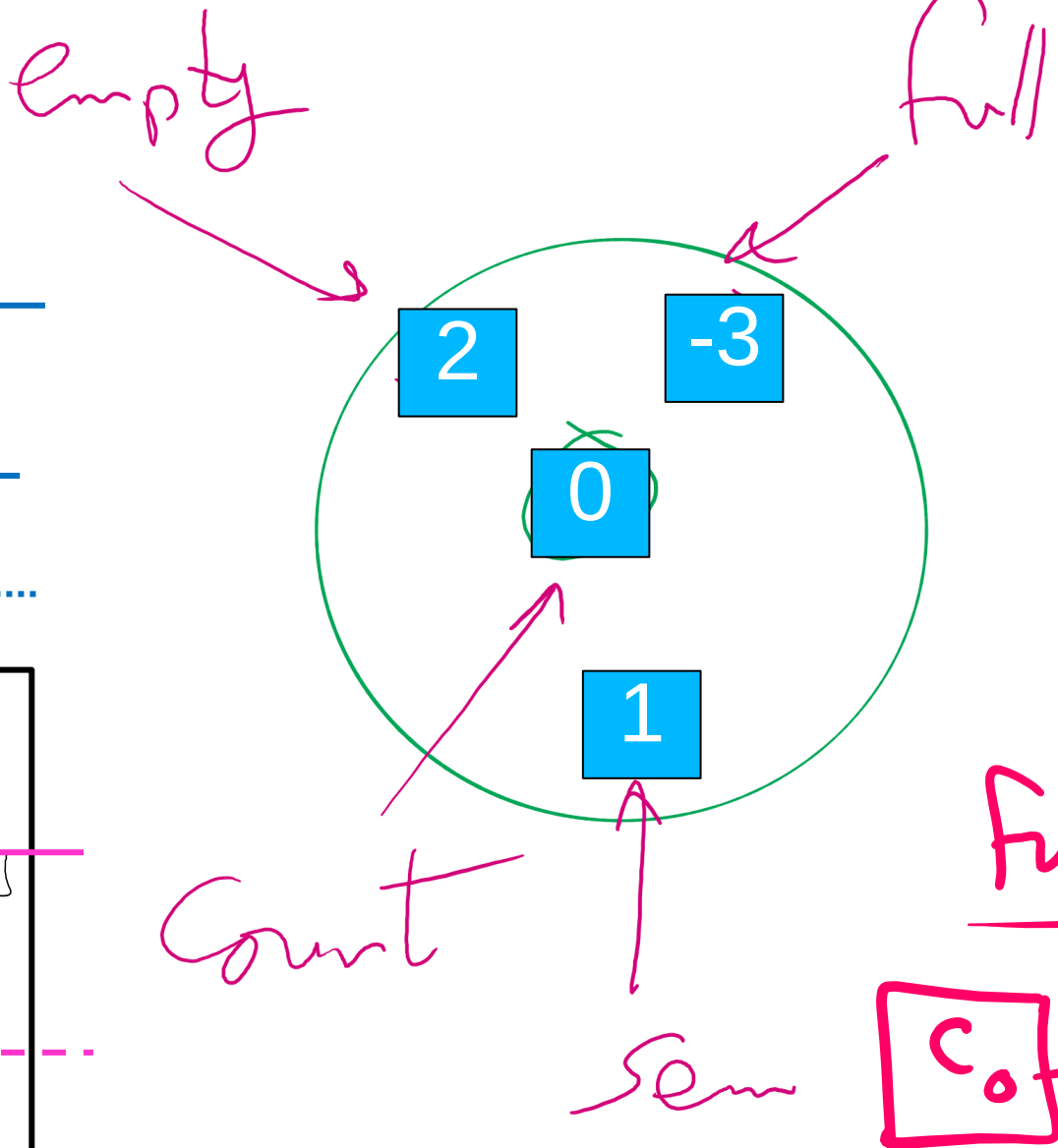
Full Queue



C2 arrives



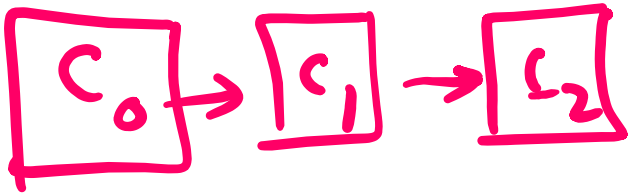




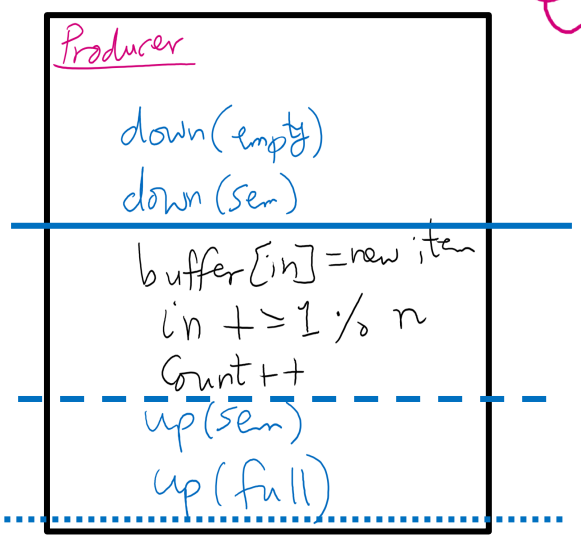
n == 2

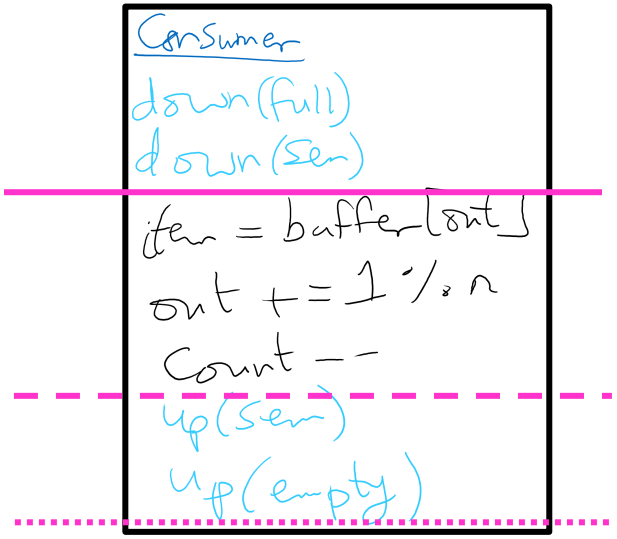
C_0 arrives
C_1 arrives
C_2 arrives
P_0 arrives
P_0 enters
C_0 enters

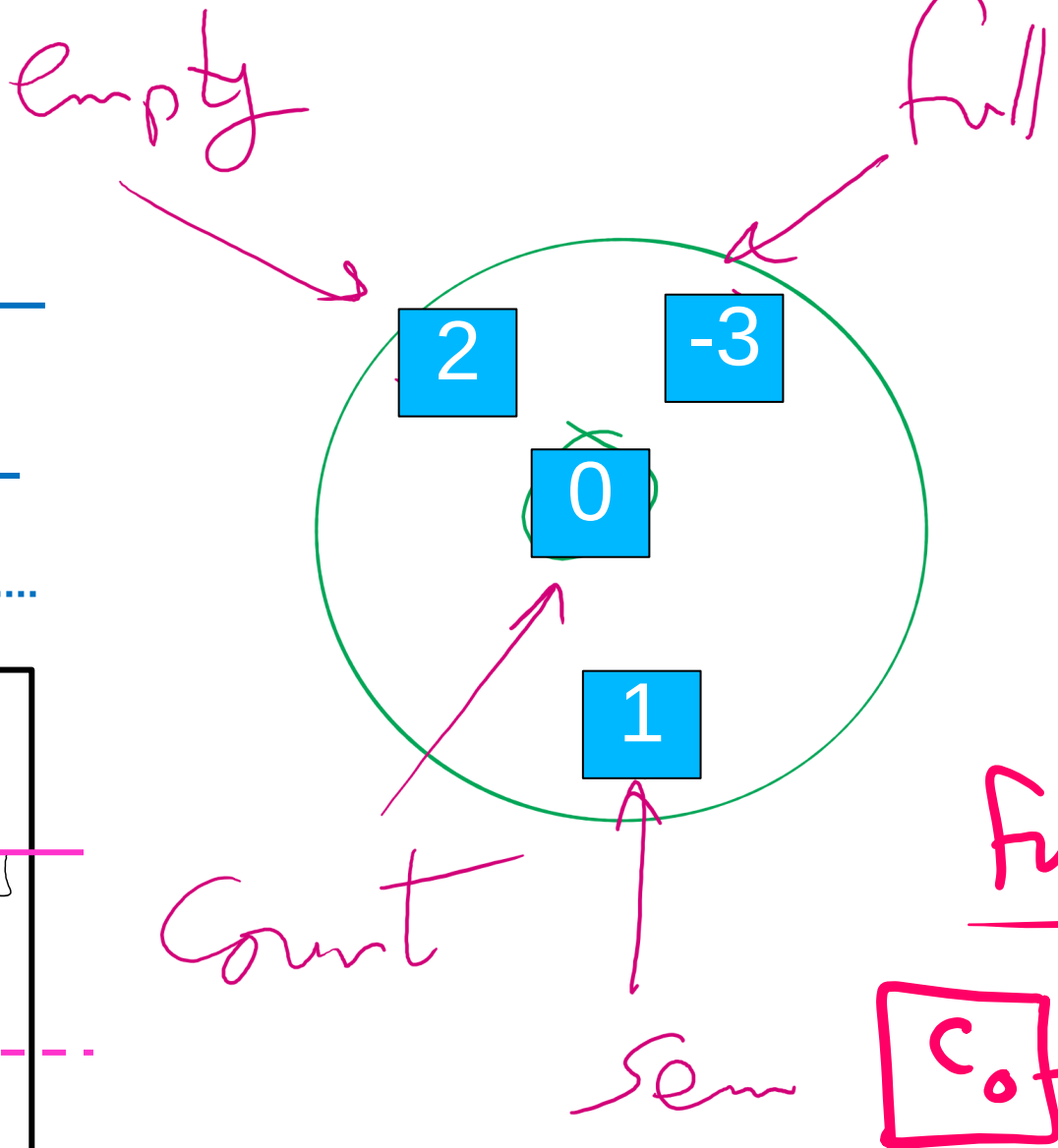
Full Queue



P0 arrives



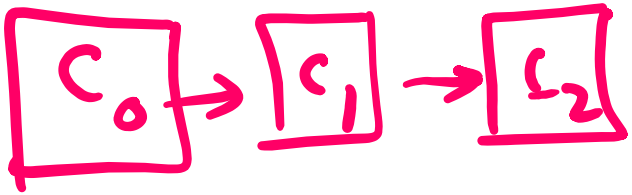




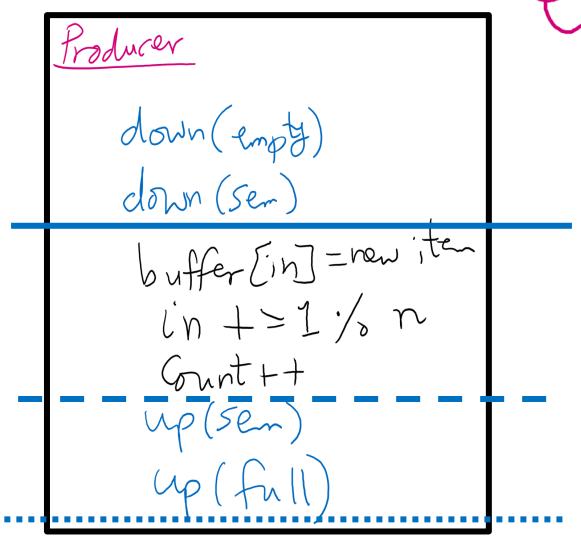
n == 2

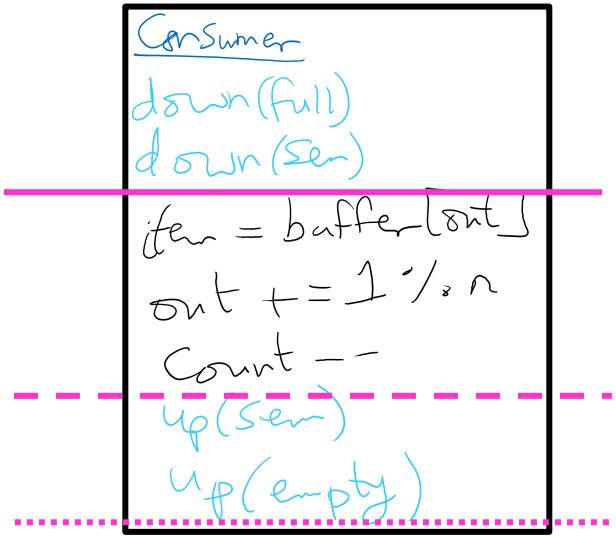
C_0 arrives
C_1 arrives
C_2 arrives
P_0 arrives
P_0 enters
C_0 enters

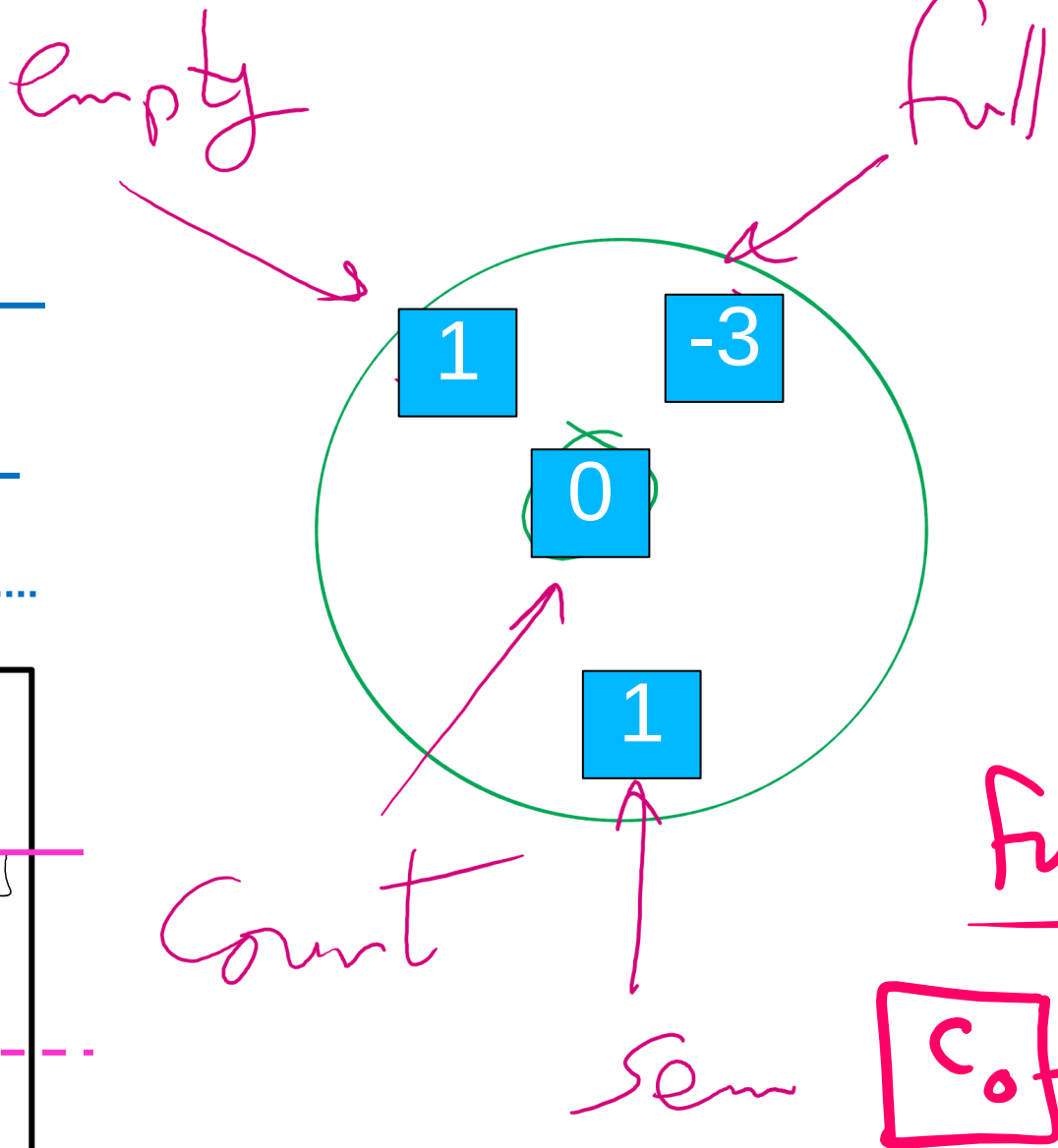
Full Queue



P0 arrives



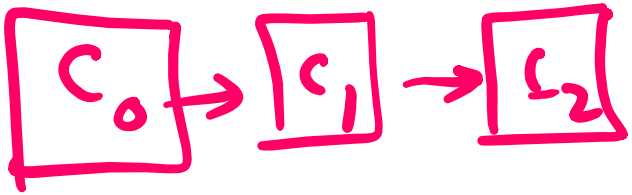




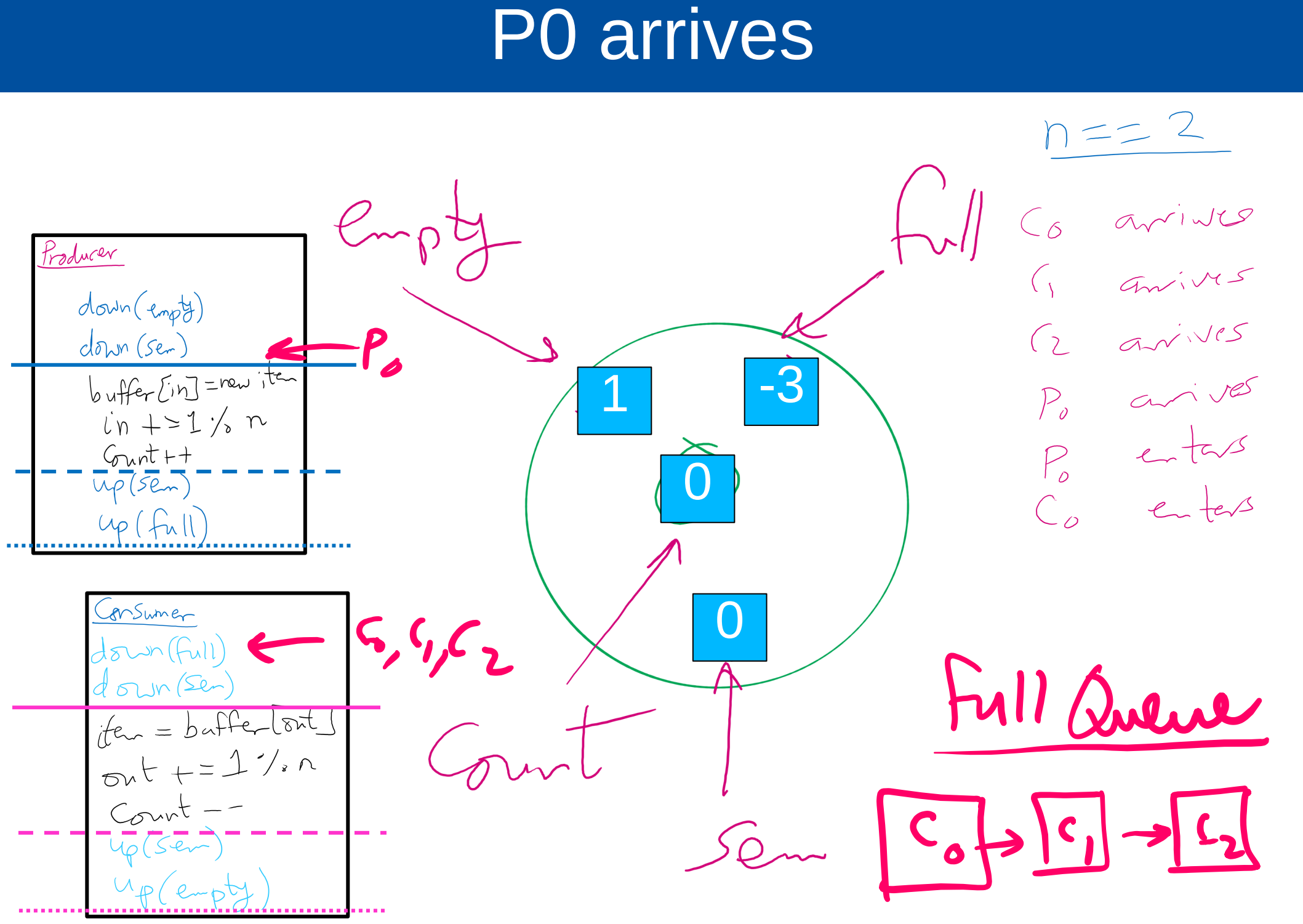
n == 2

C_0 arrives
C_1 arrives
C_2 arrives
P_0 arrives
P_0 enters
C_0 enters

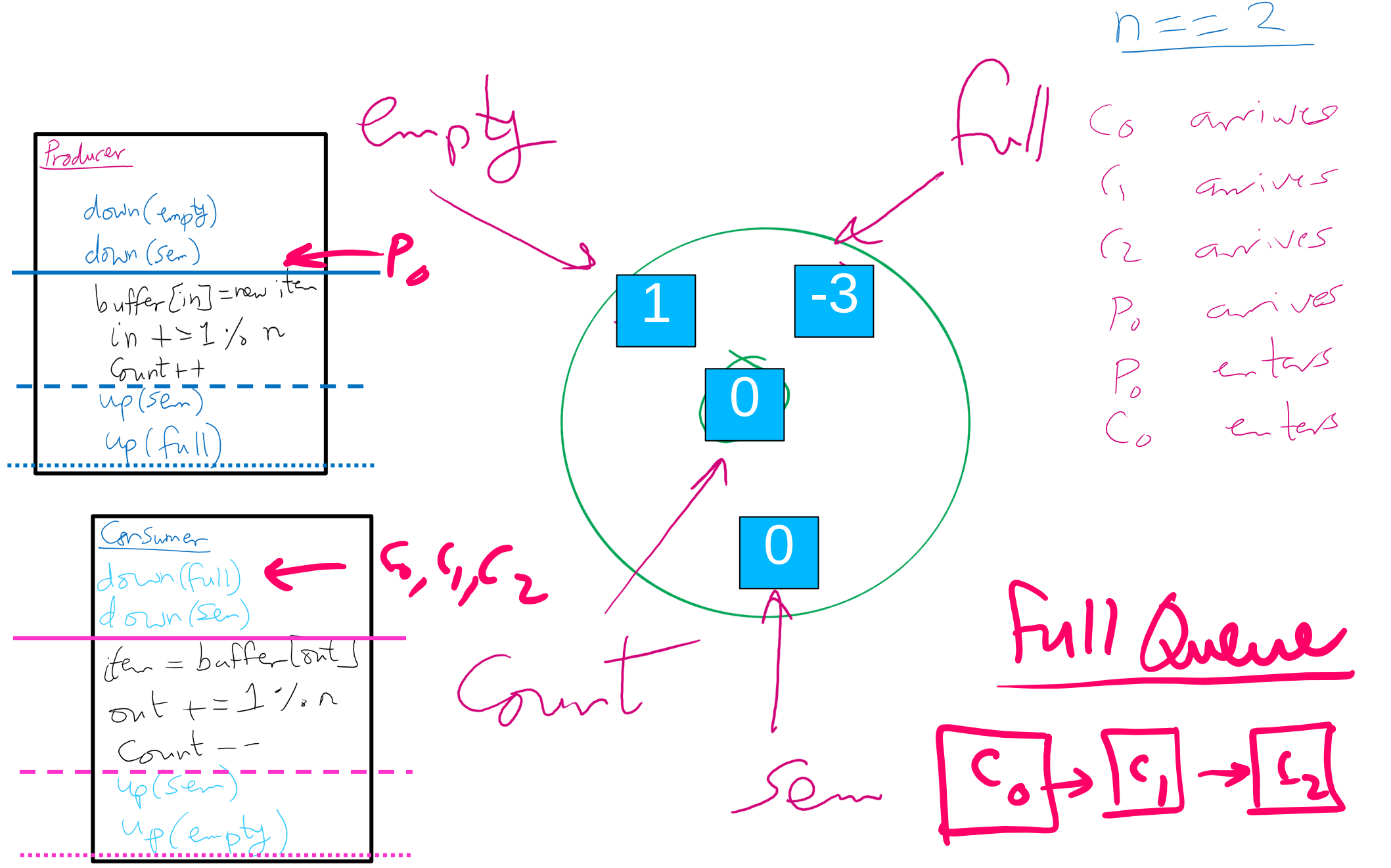
Full Queue



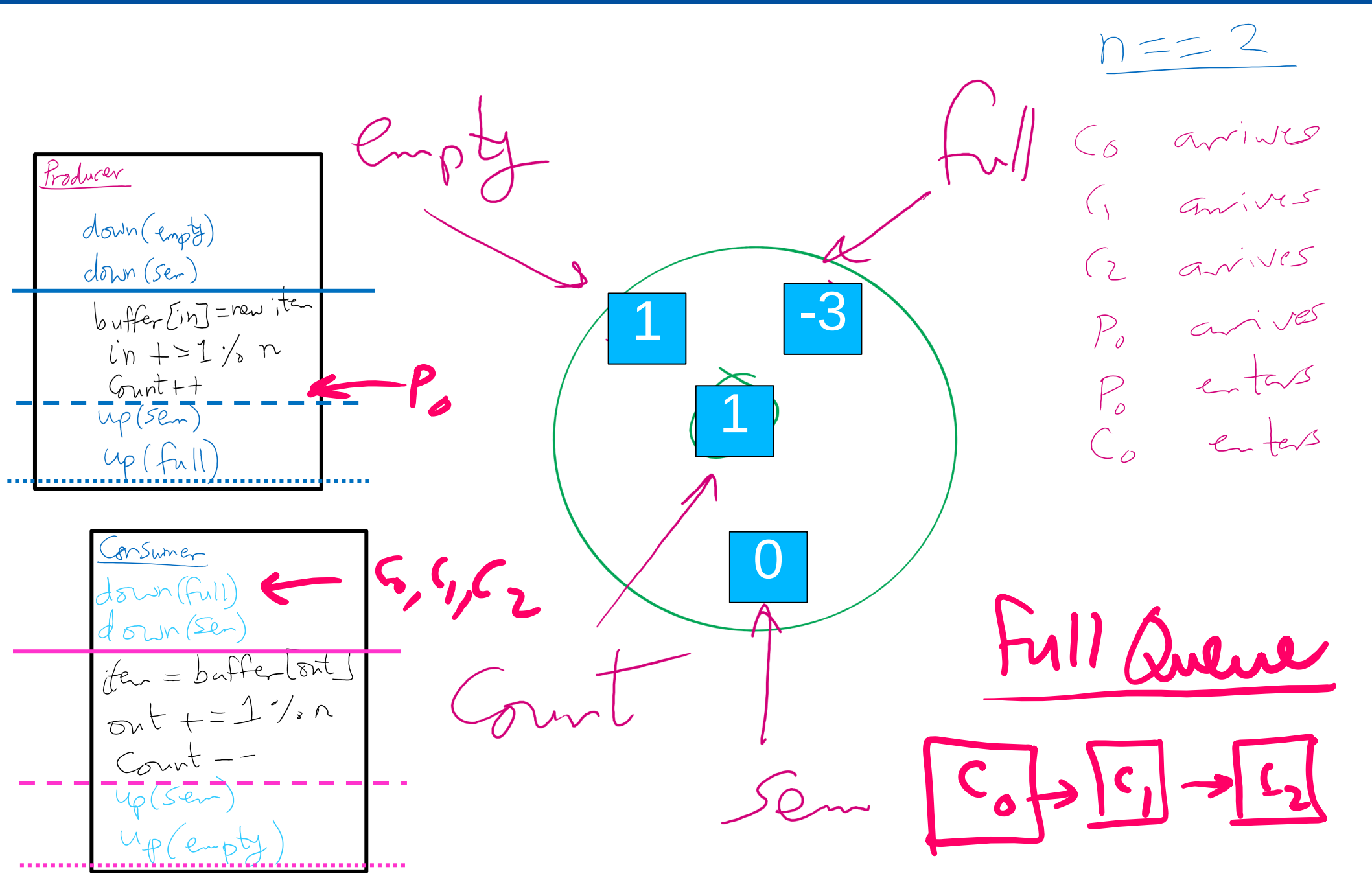
P0 arrives


n == 2
C0 arrives
C1 arrives
C2 arrives
P0 arrives
P0 enters
C0 enters
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
1
-3
0
0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0, C1, C2
count
sem
Full Queue
C0
C1
C2

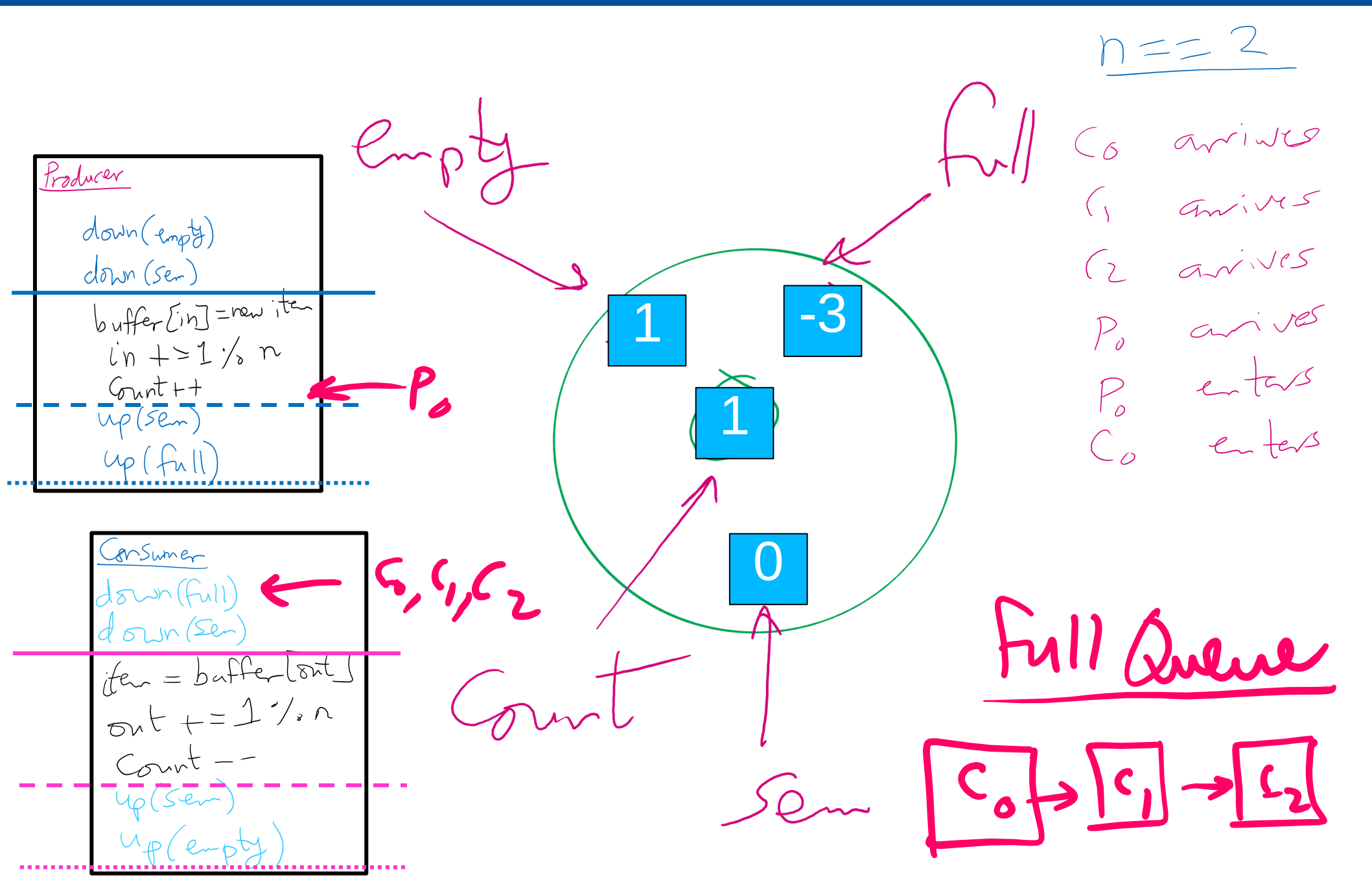
P0 enters


n == 2
C0 arrives
C1 arrives
C2 arrives
P0 arrives
P0 enters
C0 enters
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
Count++
up(sem)
up(full)
P0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
Count--
up(sem)
up(empty)
C0, C1, C2
empty
full
1
-3
0
0
Count
sem
Full Queue
C0
C1
C2

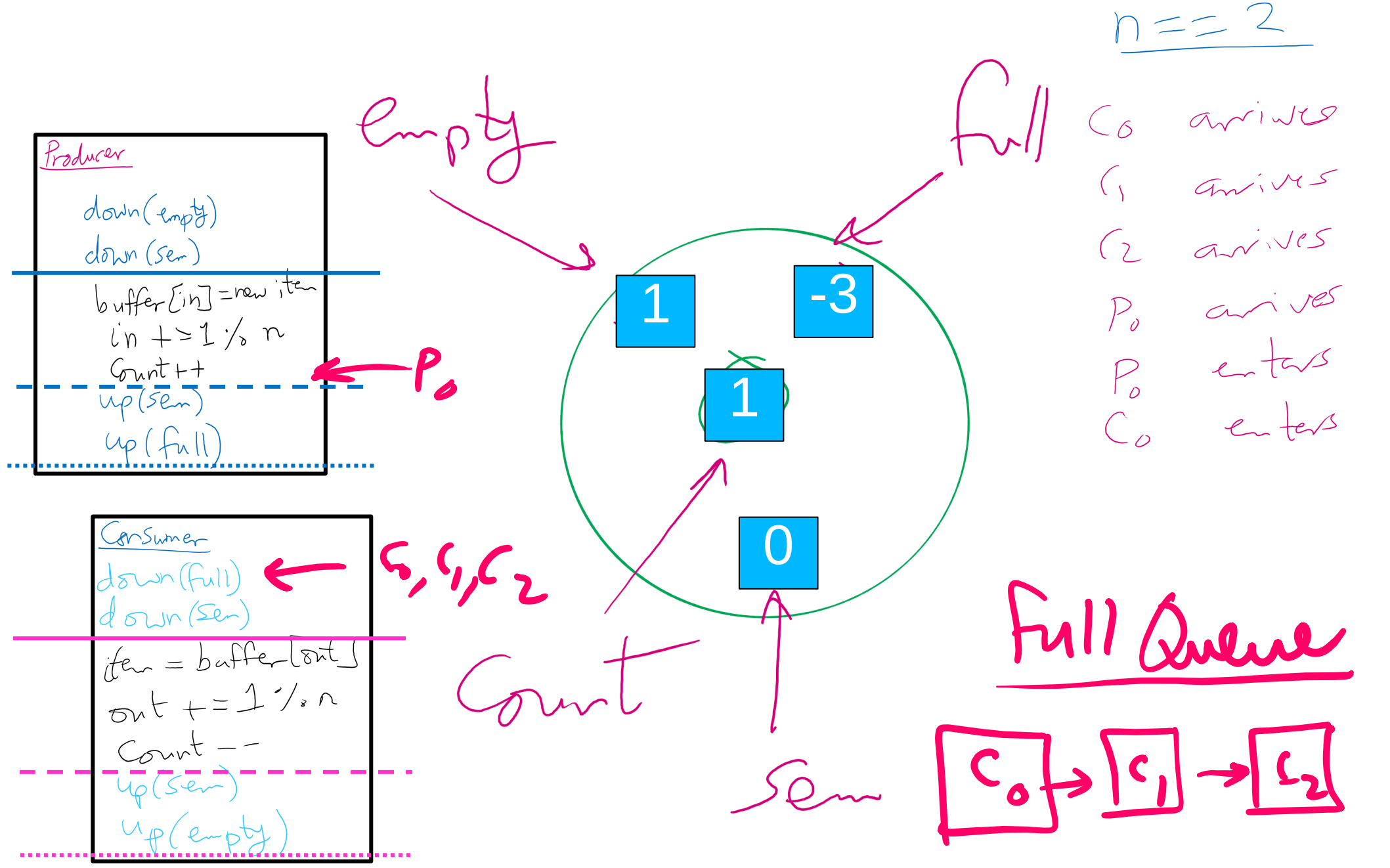
P0 enters


n == 2
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0, C1, C2
1
-3
1
0
count
sem
C0 arrives
C1 arrives
C2 arrives
P0 arrives
P0 enters
C0 enters
Full Queue
C0
C1
C2

C0 enters


n == 2
C0 arrives
C1 arrives
C2 arrives
P0 arrives
P0 enters
C0 enters
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
1
-3
1
0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0, C1, C2
count
sem
Full Queue
C0
C1
C2

Can C0 enter?


n == 2
C0 arrives
C1 arrives
C2 arrives
P0 arrives
P0 enters
C0 enters
empty
full
Producer
down(empty)
down(sem)
buffer[in] = new item
in += 1 % n
count++
up(sem)
up(full)
P0
Consumer
down(full)
down(sem)
item = buffer[out]
out += 1 % n
count--
up(sem)
up(empty)
C0, C1, C2
1
-3
1
0
count
sem
Full Queue
C0
C1
C2