G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Semaphores

Dr Jason Atkin

Last Lectures

- Critical sections and Mutual Exclusion
  - Spinlocks
  - Round-robin
  - Decker's Algorithm
  - Peterson's Algorithm
  - Busy-wait consumed CPU time
- System support
  - CRITICAL_SECTION objects
  - Mutex objects
  - Owner by the thread – no re-entry problems
  - Actually 'sleep' and 'wake' the thread/process

This lecture

- Semaphores
  – Locks with a counting mechanism

- The theory of semaphores
- Using semaphores

- Problems where semaphores can help
  – Producer - consumer

Semaphore

- A counting mechanism
  - A counter and two atomic operations upon it
- Could think of it as a limited resource which threads can ask for
  - Wait(), p() : try to get some of the resource
  - Signal(), v() : indicate that you have finished with the resource
- Differences from mutexes:
  - Counter, not just a binary flag for locked or not
  - Not owned by the thread
    - wait()/p() twice decrements by two, can deadlock yourself
- Binary semaphore : only one can have it

More formally: Semaphores

- A *semaphore* s is an integer variable
  - which can take only non-negative values
- Once it has been given its initial value, the only permissible operations on s are the atomic actions:
- ***P***(s) : if $s > 0$ then $s = s - 1$, else ***suspend*** execution of the process that called ***P***(s)
  - Sometimes called wait() or lock()
- ***V***(s) : if some process p is suspended by a previous ***P***(s) on this semaphore then resume p, else $s = s + 1$
  - Sometimes called signal() or release()

- A *general semaphore* can have any non-negative value
- A *binary semaphore* is one whose value is always 0 or 1

P and ***V*** as atomic actions

- Reading and writing the semaphore value needs to be in a *critical section*, or atomic action:
  - ***P*** and ***V*** operations on the same semaphore must be *mutually exclusive*
  - e.g., suppose we have a semaphore, *s*, which has the value 1, and two processes simultaneously attempt to execute ***P*** on *s*:
    - only one of these operations will be able to complete before the next ***V*** operation on *s;*
    - the other process attempting to perform a ***P*** operation is suspended.
  - Semaphore operations on different semaphores do not need to be mutually exclusive

V() on binary semaphores

- Binary semaphores have values 0 or 1
- Effects of performing a ***V*** operation on a binary semaphore which has a current value of 1 are implementation dependent:
  - operation may be ignored
  - may increment the semaphore
  - may throw an exception
- We will assume that a ***V*** operation on a binary semaphore which has value 1 does not increment the value of the semaphore.
- Note: Windows applies a maximum on any semaphore, set when created

Example implementation

Integer Variable: SEM

Initialisation: SEM = <Initial count>

Implementation of p()

Lock Critical Section

If SEM > 0

SEM = SEM - 1

Unlock Critical Section

Return success

Unlock Critical Section

Spin, wait for SEM>0 **and repeat** (or return failure)

Implementation of v()

Lock Critical Section

SEM = SEM + 1

Unlock Critical Section

Note: With atomic increment/decrement, the critical section is implicit

Resuming suspended processes

- Note that the definition of ***V*** doesn’t specify which process is woken up if more than one process has been suspended on the same semaphore
- This has implications for the fairness of algorithms implemented using semaphores and upon properties like Eventual Entry

Windows support : CreateSemaphore

- MSDN, Using Semaphores: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686946%28v=vs.85%29.aspx
- Create/access the object and store the handle

```
HANDLE ghSemaphore;
ghSemaphore = CreateSemaphore(
      NULL,           // default security attributes
      MAX_SEM_COUNT,  // initial count
      MAX_SEM_COUNT,  // maximum count
      NULL);          // unnamed semaphore
```

- Use the semaphore
  - WaitForSingleObject(), ReleaseSemaphore()
- Close the handle at the end

```
CloseHandle(ghSemaphore);
```

Windows support : wait and release

```
switch (WaitForSingleObject(
            ghSemaphore, // semaphore handle
            10000L) )    // Timeout – could be zero
{
   case WAIT_OBJECT_0:
      printf("Thread %d: wait succeesszn", GetCurrentThreadId());
      if ( !ReleaseSemaphore(
            ghSemaphore,  // handle to semaphore
            1,            // increase count by one
            NULL) )       // not interested in previous count
      { /* Handle error */ }
      break;

   case WAIT_TIMEOUT:
      printf("Thread %d: wait timed out\n", GetCurrentThreadId());
      break;
   }
```

Maximum simultaneous executions

Using semaphores for maximum counts

- You could use a binary semaphore as a simple method to ensure mutual exclusion
  - Only one thread can have it at once

- A semaphore with initial value of n can ensure that only n threads can access a section of code (or resource) at once
  - Each calls p() before it enters (or needs the resource) and v() when it finishes with it

Using a semaphore to protect a resource

Semaphore Count: 2

T1	T2	T3	T4
→ init	→ init	→ init	→ init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P()	P()
Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 2

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P()	P()
Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 1

T1	T2	T3	T4
init	→ init	→ init	→ init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P()	P()
→ Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 1

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P()	P()
Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P()	P()
Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	→ init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	→ P() **Blocked**	P()
→ Use res	→ Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	→ P() **Blocked**	→ P() **Blocked**
→ Use res	→ Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	→ P() **Blocked**	→ P() **Blocked**
Use res	→ Use res	Use res	Use res
→ V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 1

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P() **Blocked**	P() **Blocked**
Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 1

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	→ P()	→ P() **Blocked**
Use res	→ Use res	Use res	Use res
V()	V()	V()	V()
→ rem()	rem()	rem()	rem()

Note: you don't know which one will get unblocked. T3 here.

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P()	→ P() **Blocked**
Use res	→ Use res	→ Use res	Use res
V()	V()	V()	V()
→ rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P()	P() **Blocked**
Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
→ P() **`Blocked`**	P()	P()	→ P() **`Blocked`**
Use res	→ Use res	→ Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P() **Blocked**	P()	P()	P() **Blocked**
Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 1

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
→ P()	P()	P()	→ P() **Blocked**
Use res	→ Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	→ rem()	rem()

Note: you don't know which one will get unblocked. T1 here.

Using a semaphore to protect a resource

Semaphore Count: 0

T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	P()	P() **Blocked**
Use res	Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Using a semaphore to protect a resource

Semaphore Count: 0

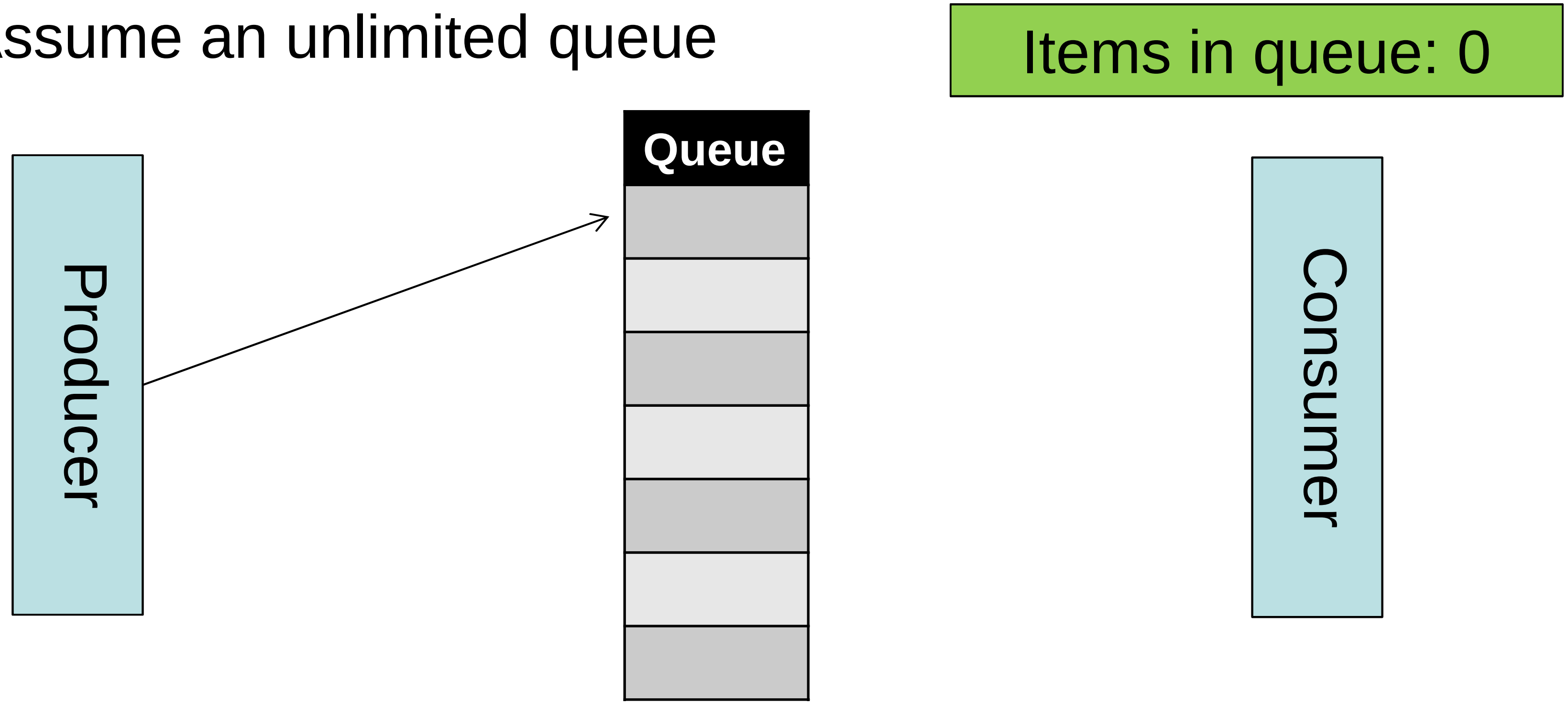
T1	T2	T3	T4
init	init	init	init
Loop:	**Loop:**	**Loop:**	**Loop:**
P()	P()	→ P() **Blocked**	→ P() **Blocked**
→ Use res	→ Use res	Use res	Use res
V()	V()	V()	V()
rem()	rem()	rem()	rem()

Etc… The number of threads between the p() and v() is limited

Consumer-Producer

Consumer-Producer

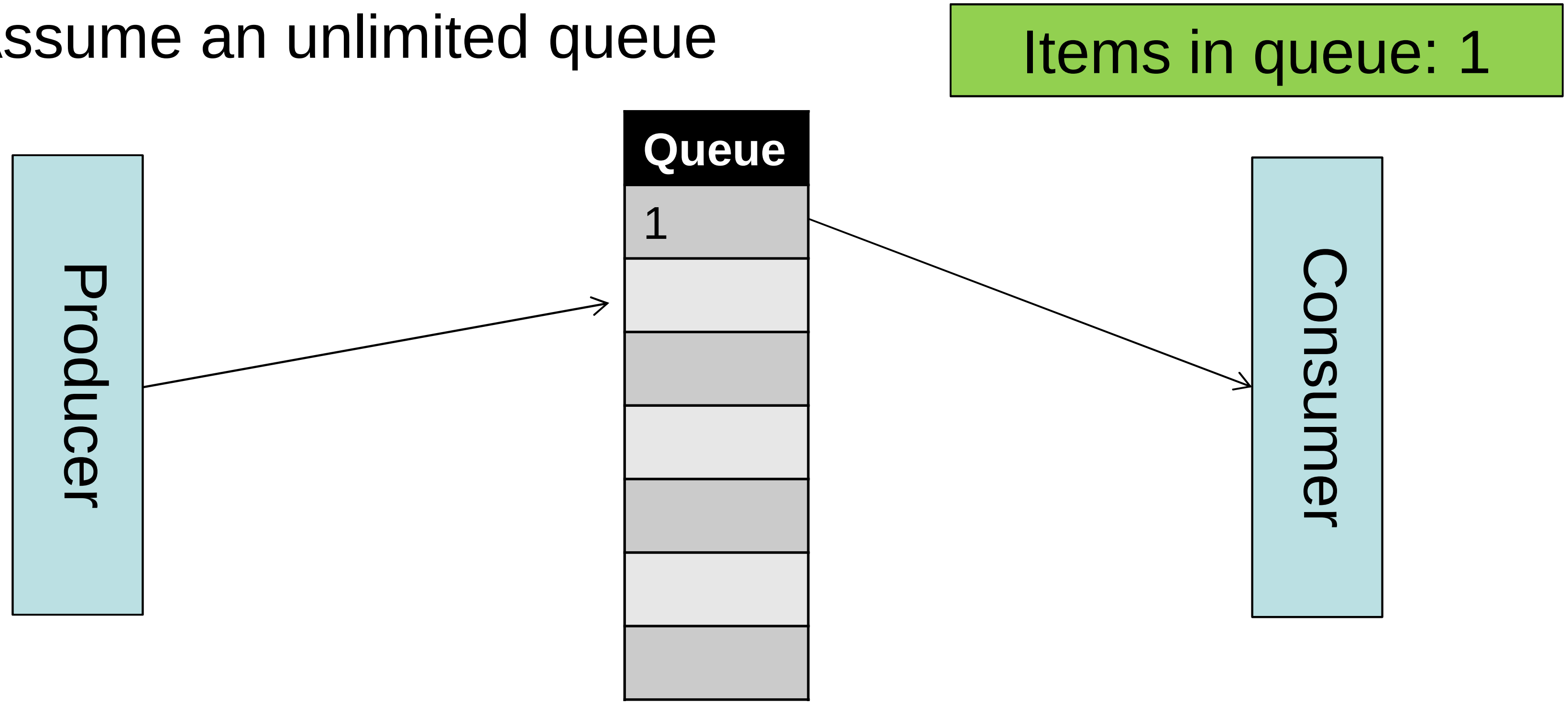
- Assume an unlimited queue



- Consumer cannot consume a product until it has been produced
- How can we use a (single) semaphore to prevent consumption before production?

Consumer-Producer

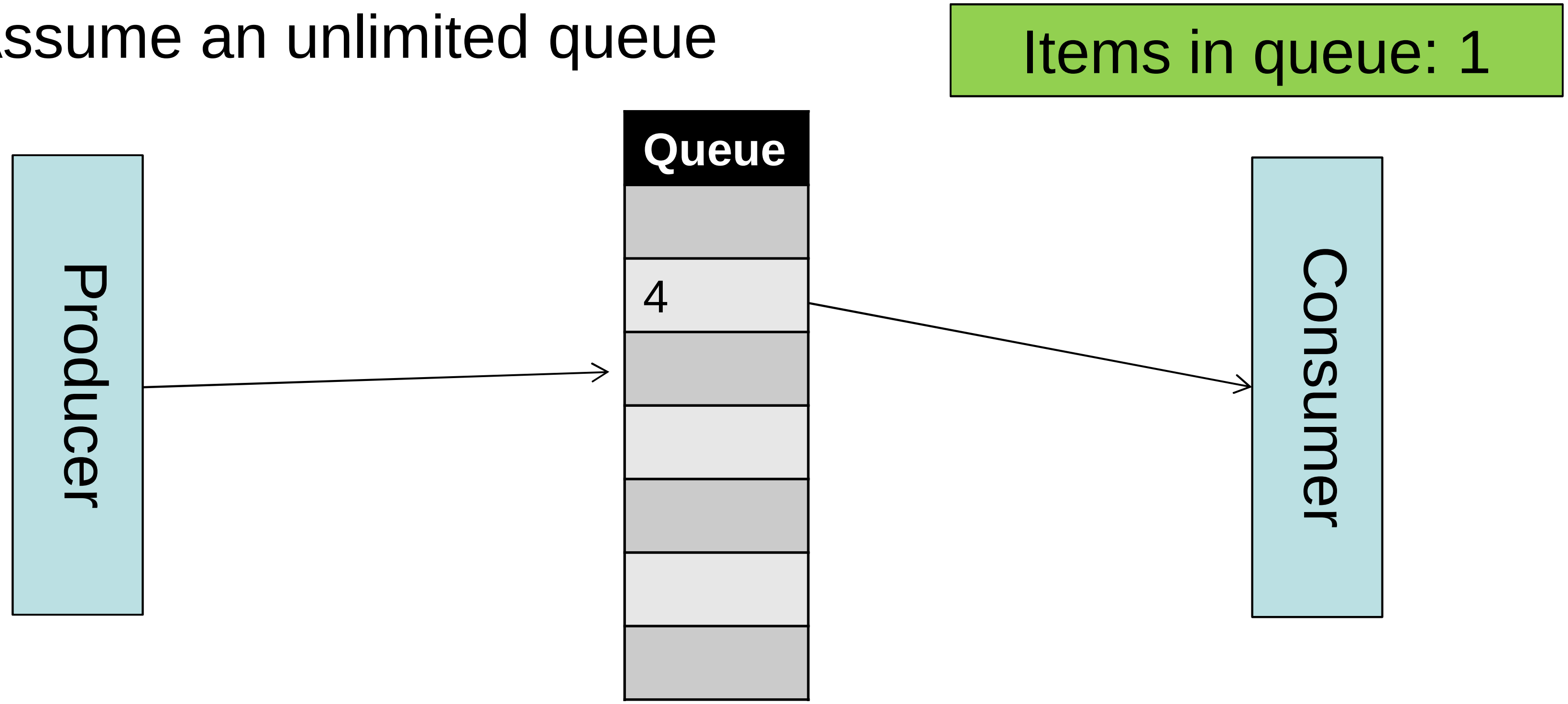
- Assume an unlimited queue



- Consumer cannot consume a product until it has been produced
- How can we use a (single) semaphore to prevent consumption before production?

Consumer-Producer

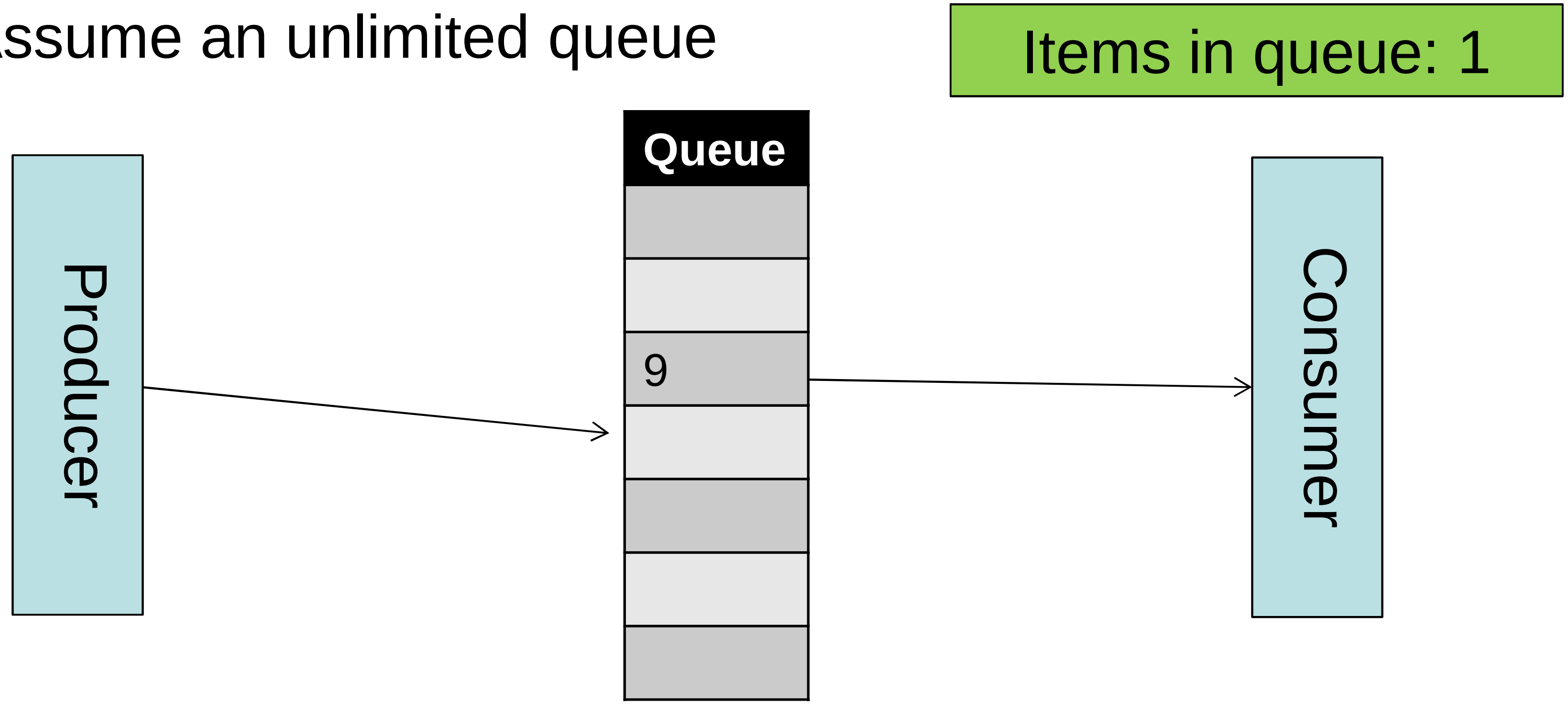
- Assume an unlimited queue



- Consumer cannot consume a product until it has been produced
- How can we use a (single) semaphore to prevent consumption before production?

Consumer-Producer

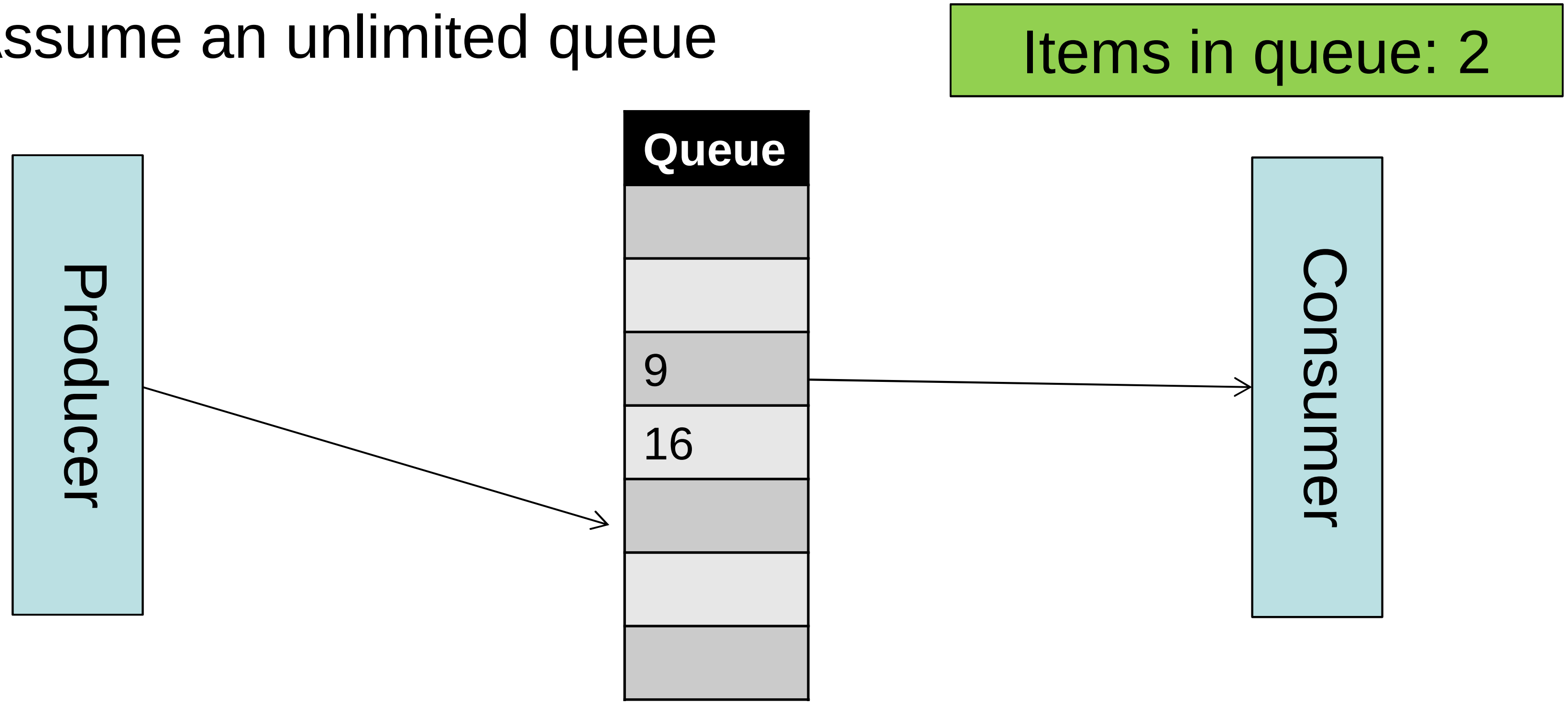
- Assume an unlimited queue



- Consumer cannot consume a product until it has been produced
- How can we use a (single) semaphore to prevent consumption before production?

Consumer-Producer

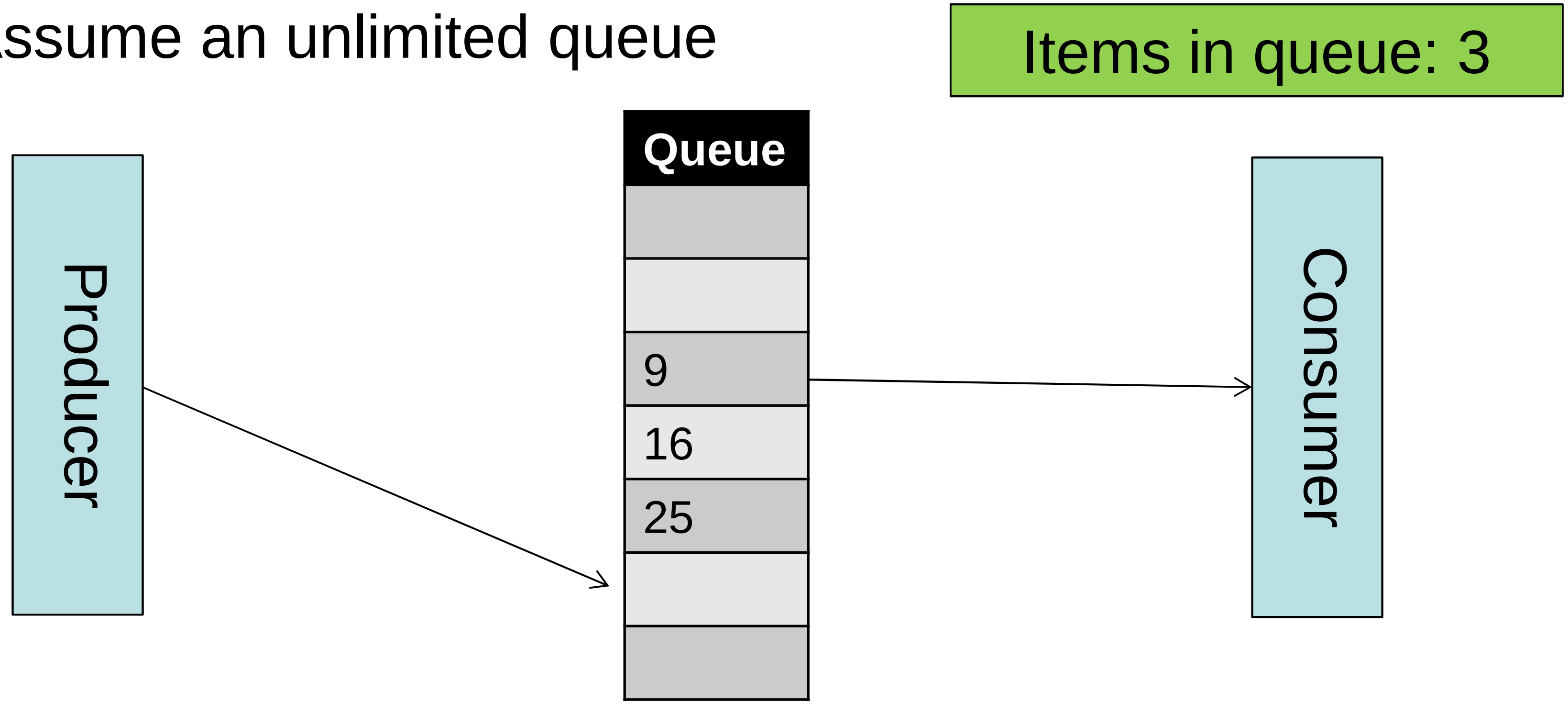
- Assume an unlimited queue



- Consumer cannot consume a product until it has been produced
- How can we use a (single) semaphore to prevent consumption before production?

Consumer-Producer

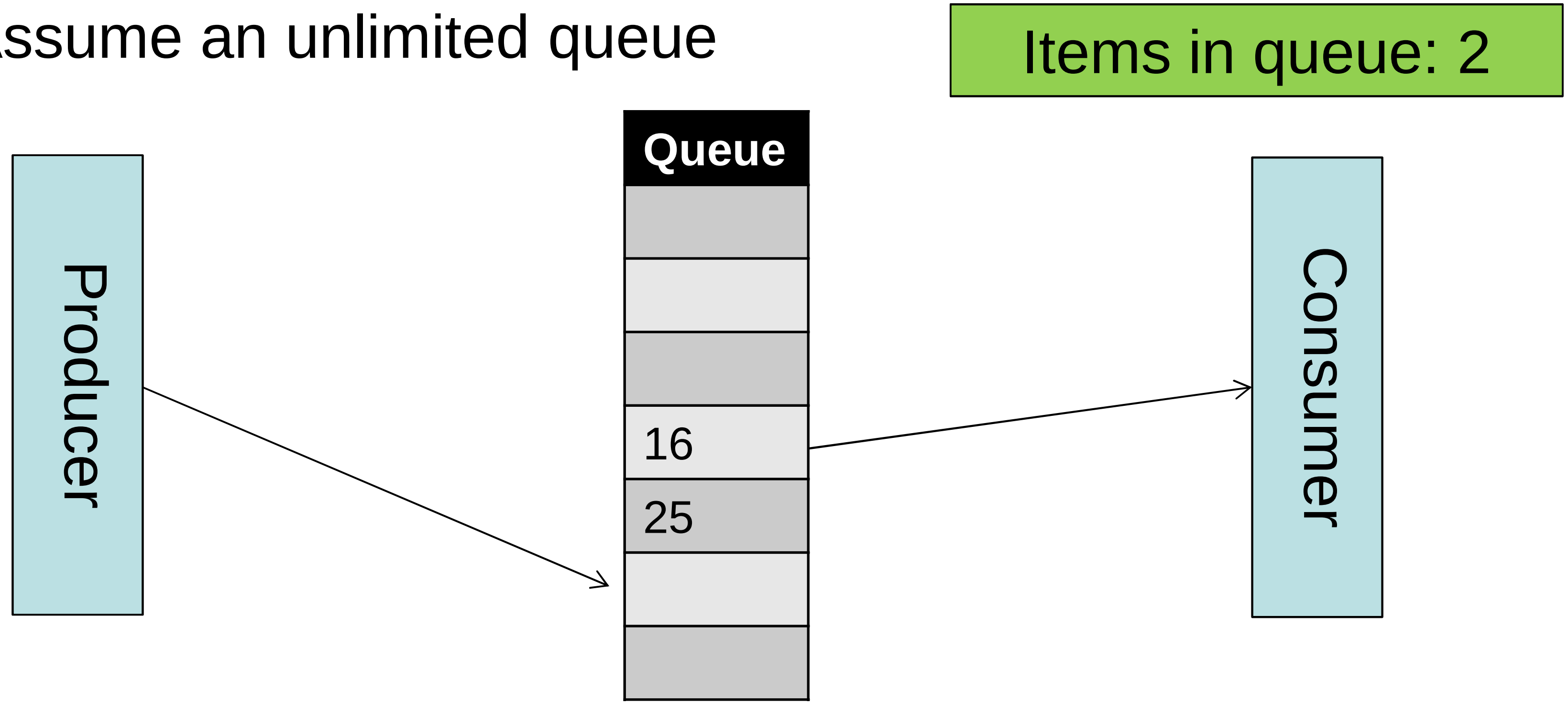
- Assume an unlimited queue



- Consumer cannot consume a product until it has been produced
- How can we use a (single) semaphore to prevent consumption before production?

Consumer-Producer

- Assume an unlimited queue



- Producer creates the items
- Consumer cannot consume a product until it has been produced – **how do we stop this?**

Answer…

Answer

- Use a semaphore as a counter
- When producer produces an item, it increments the semaphore (v(), signal())
  - Like ‘release’ on semaphore
- Consumer uses p() to consume the item
  - Blocks if count is zero
  - Decrements by 1 if the item is consumed – tracking the items still left to be consumed

Limiting the queue size

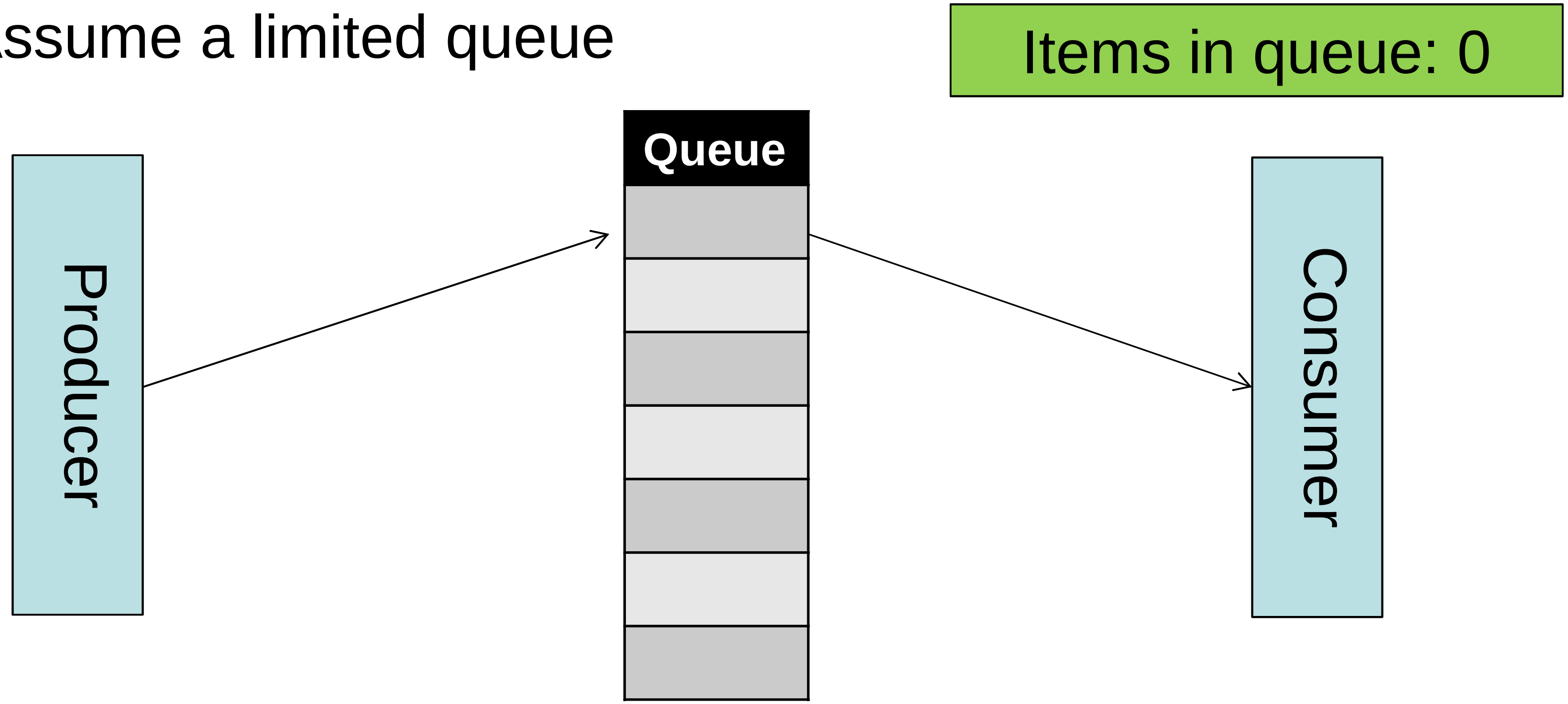
Limited queue size

- The previous example assumed an unlimited buffer size

- If we have a buffer size of n, how can we make the producer stop when the buffer gets full, to allow the consumer to empty it a bit?

Rolling buffer

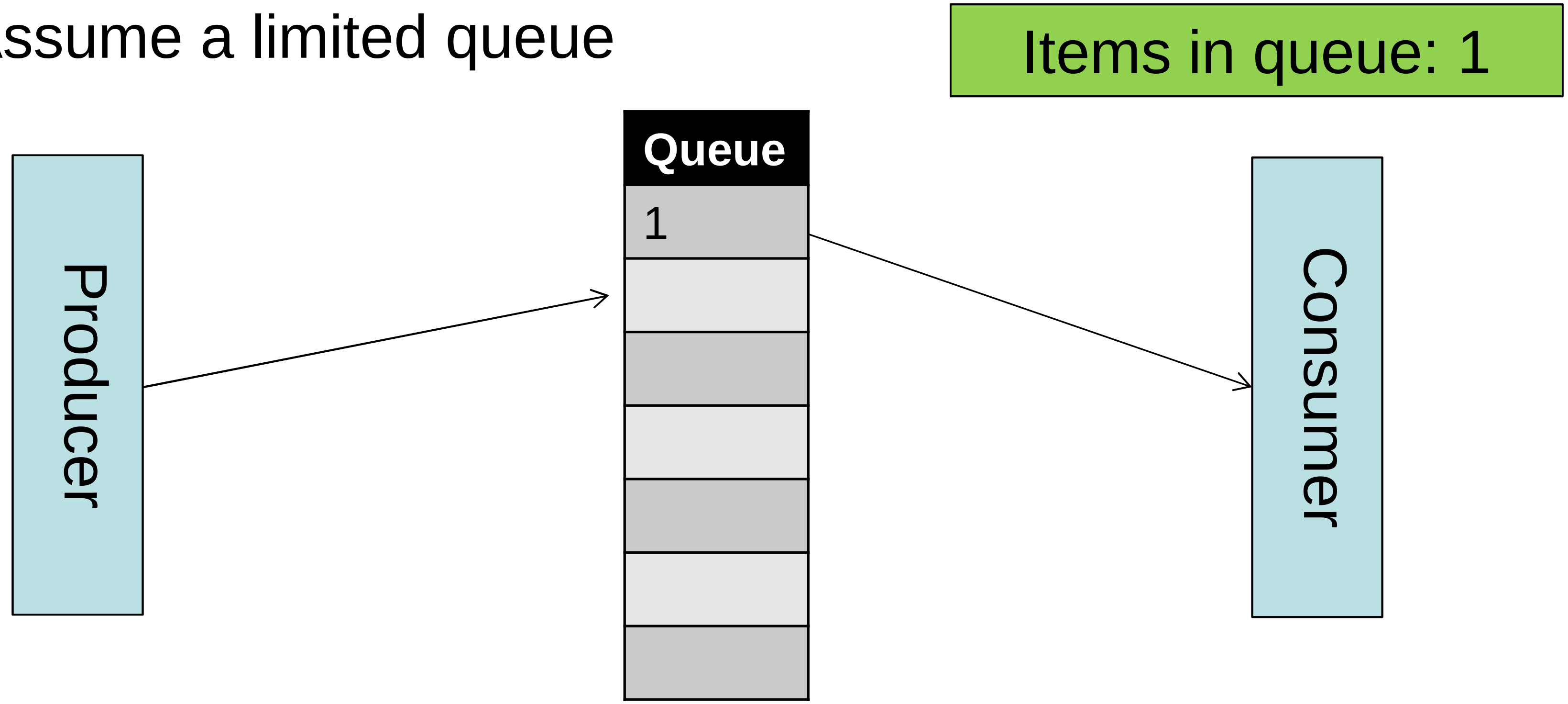
- Assume a limited queue



- When queue is filled, it rolls around, as it gets emptied

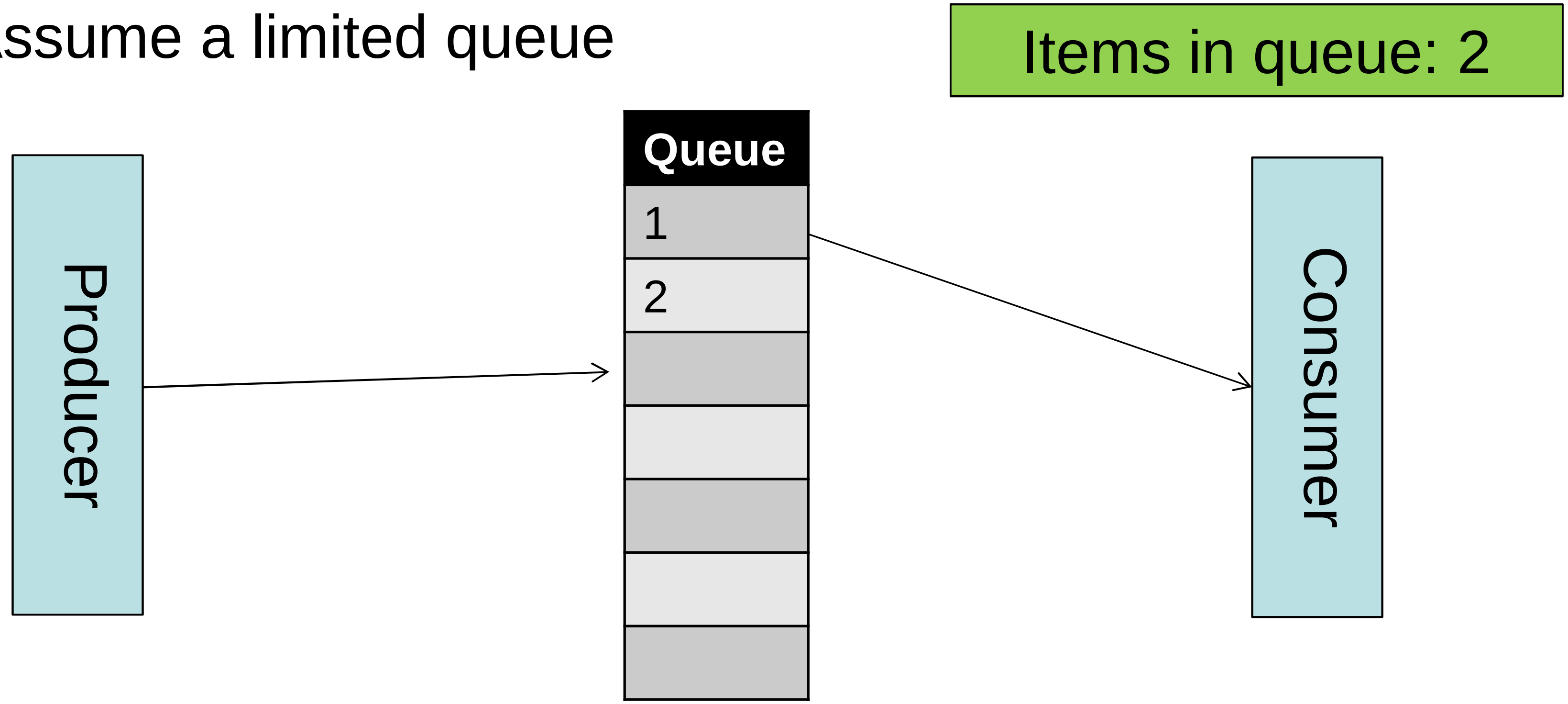
Rolling buffer

- Assume a limited queue



Rolling buffer

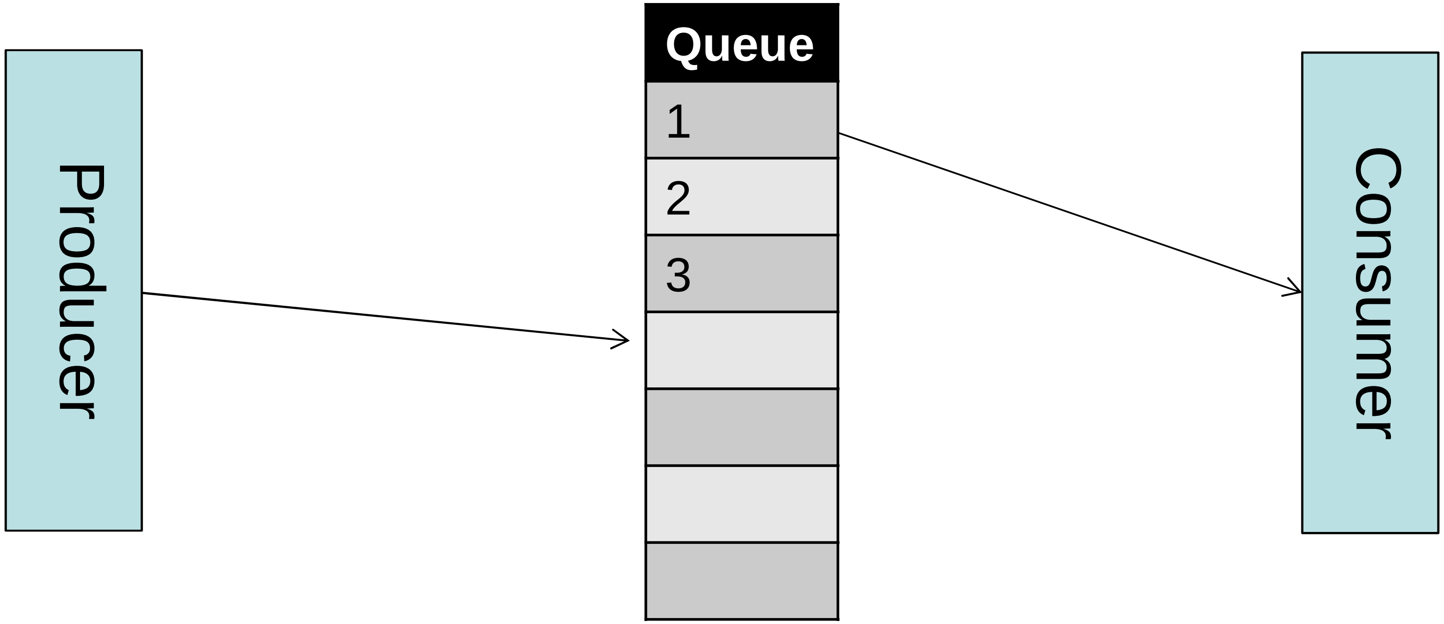
- Assume a limited queue



Rolling buffer

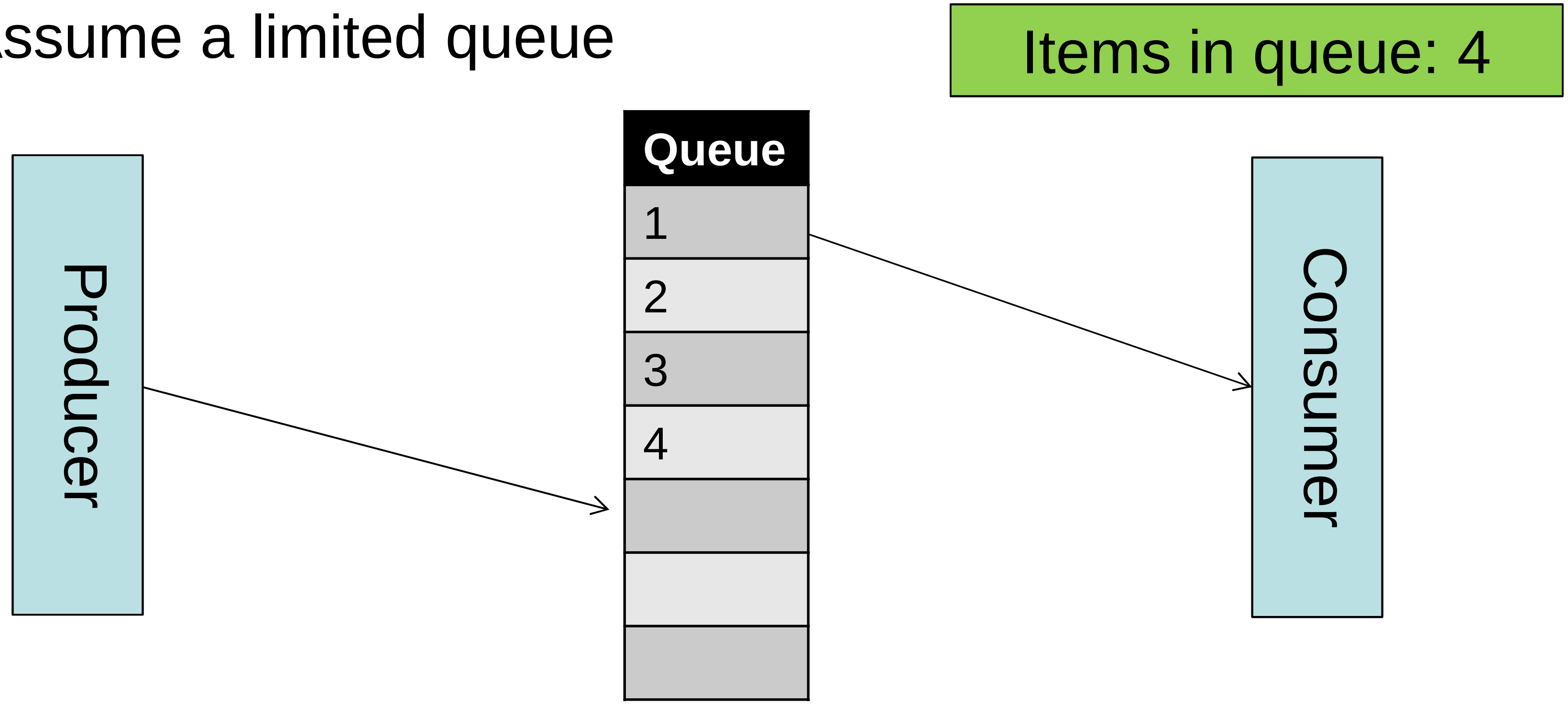
- Assume a limited queue

Items in queue: 3



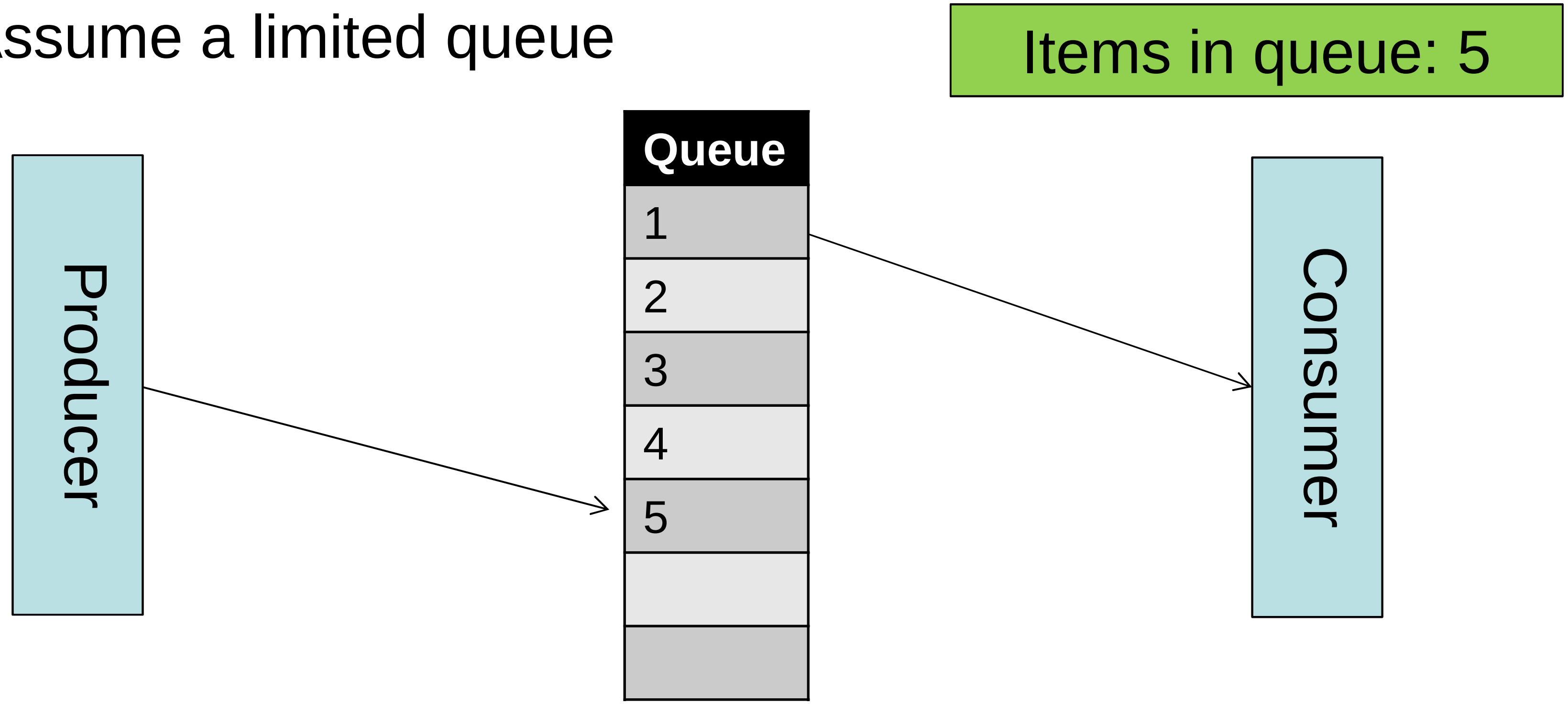
Rolling buffer

- Assume a limited queue



Rolling buffer

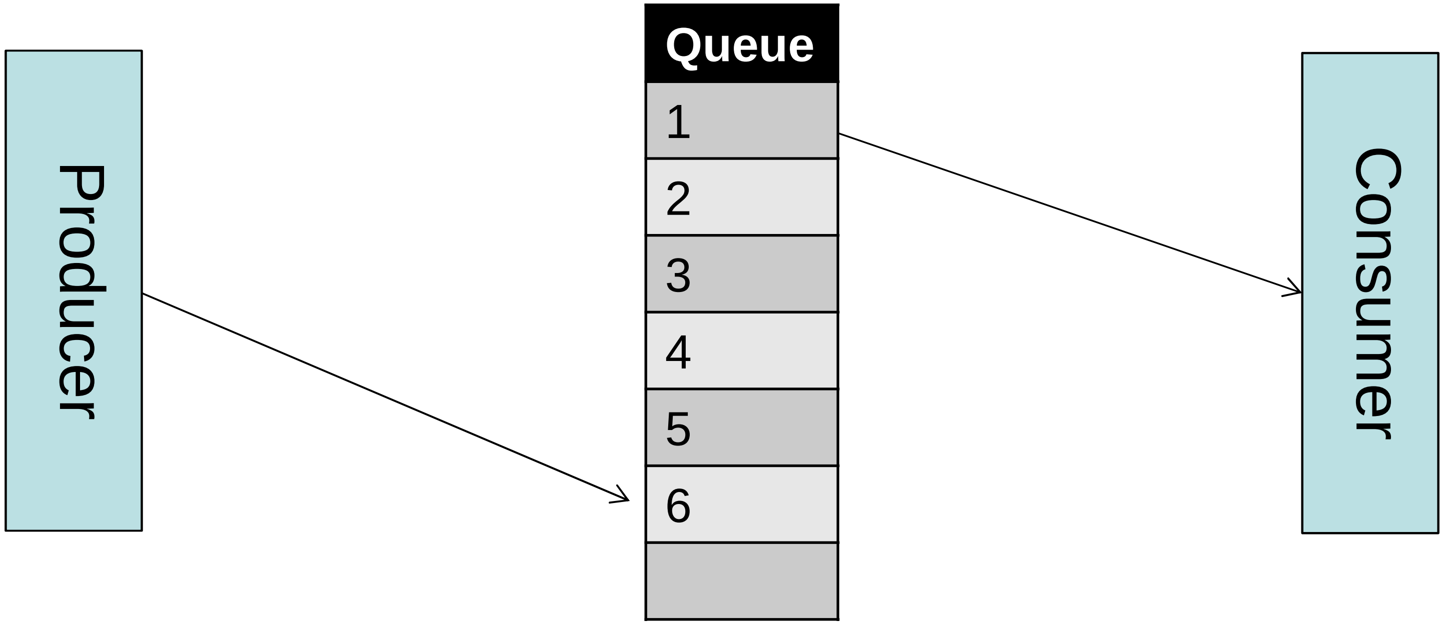
- Assume a limited queue



Rolling buffer

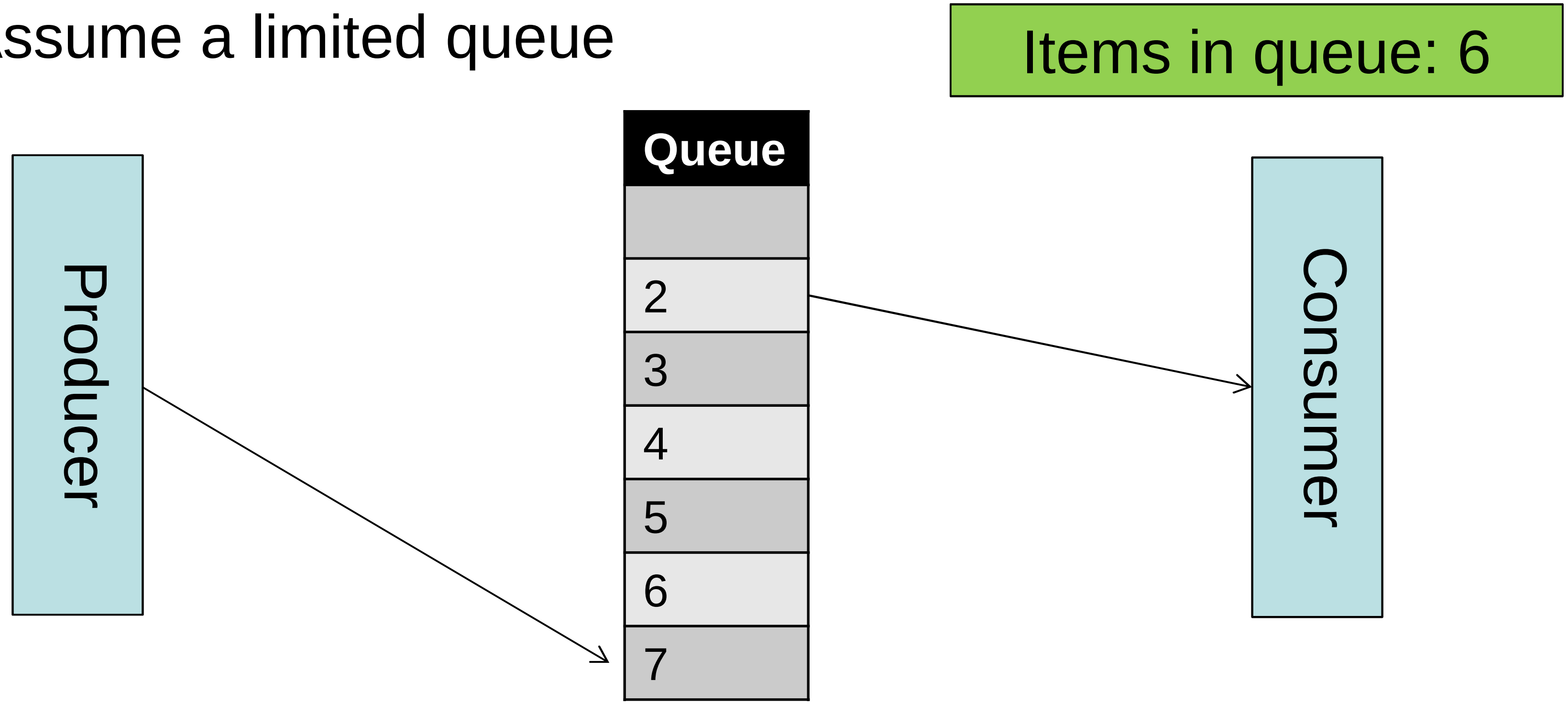
- Assume a limited queue

Items in queue: 6



Rolling buffer

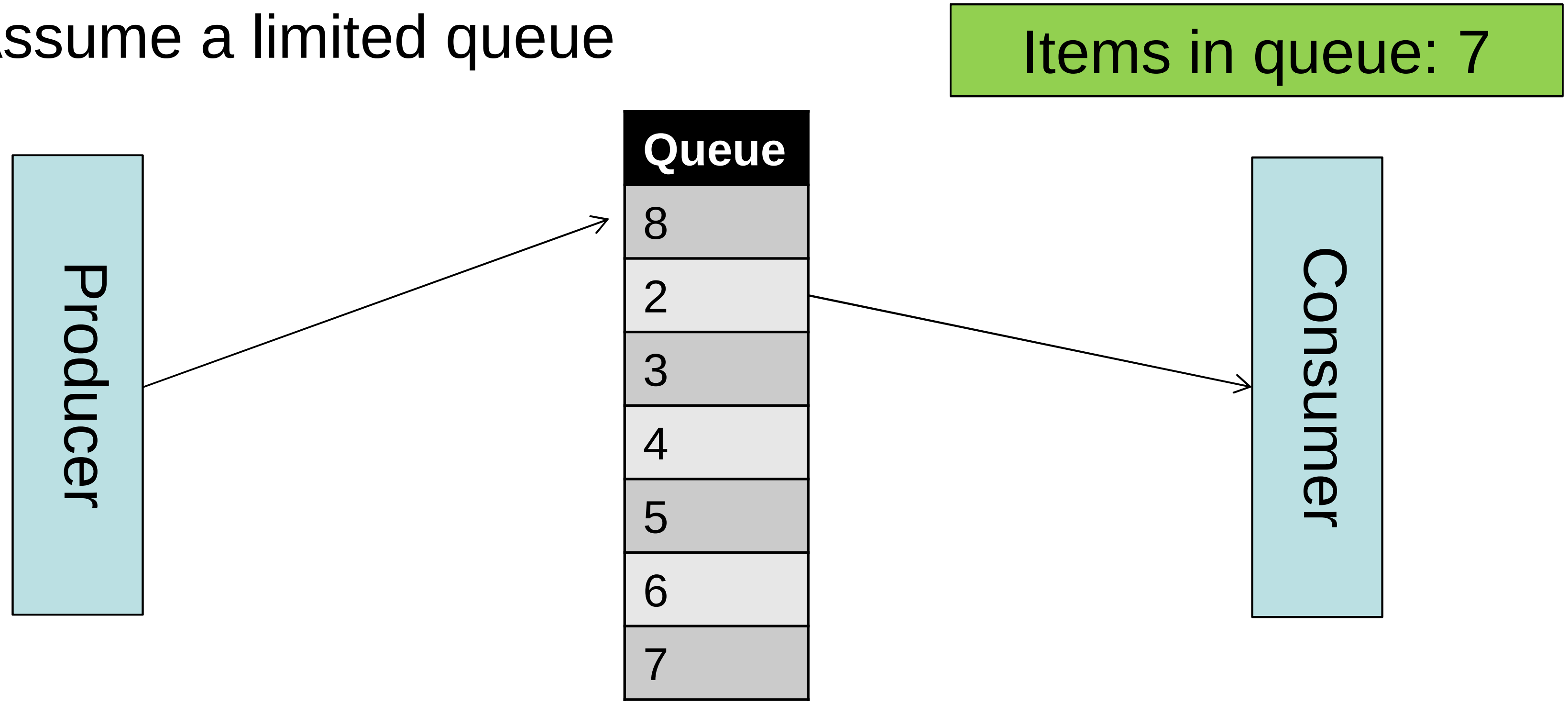
- Assume a limited queue



- Assume that consumer finally got around to consuming one

Rolling buffer

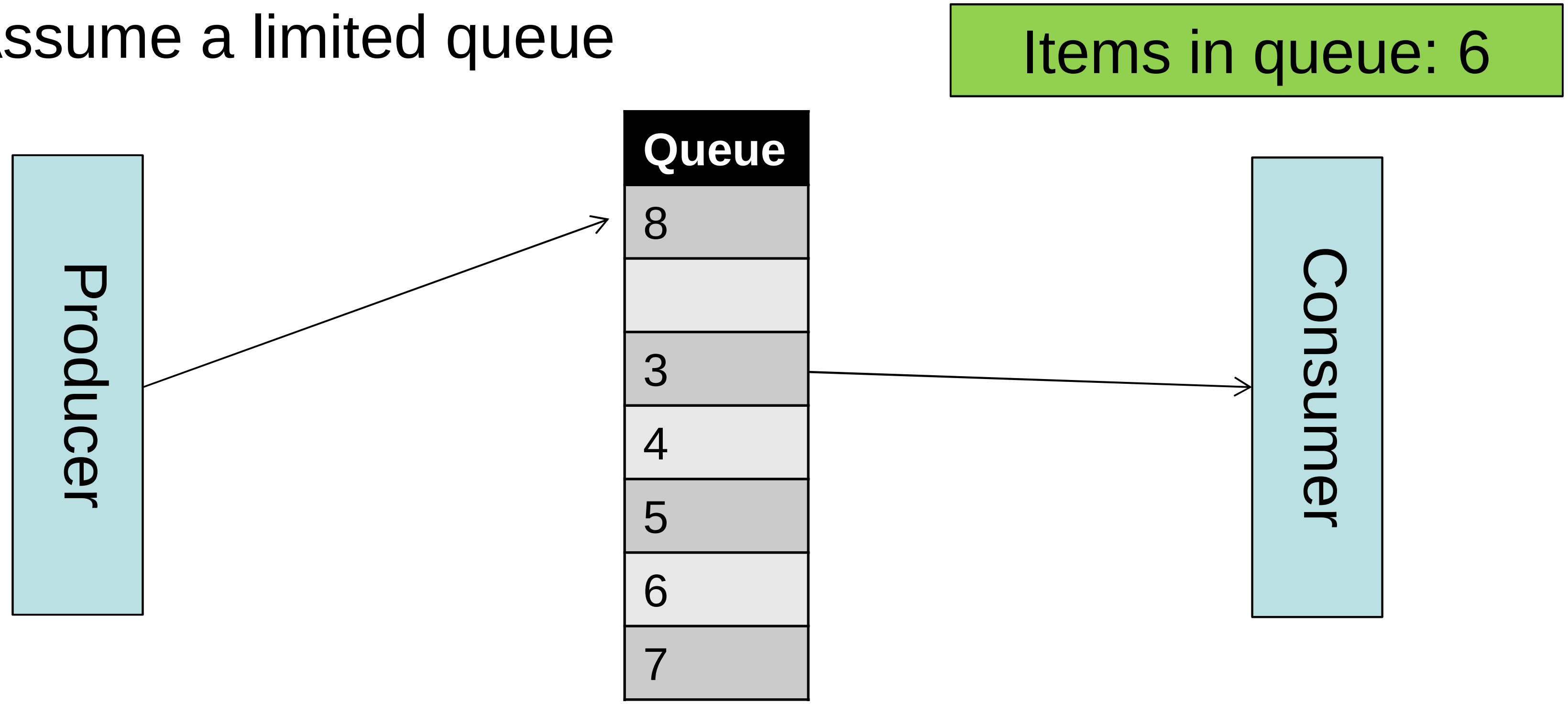
- Assume a limited queue



- Producer cannot now produce any more until the consumer has used some up
- There are no spare spaces to put the values

Rolling buffer

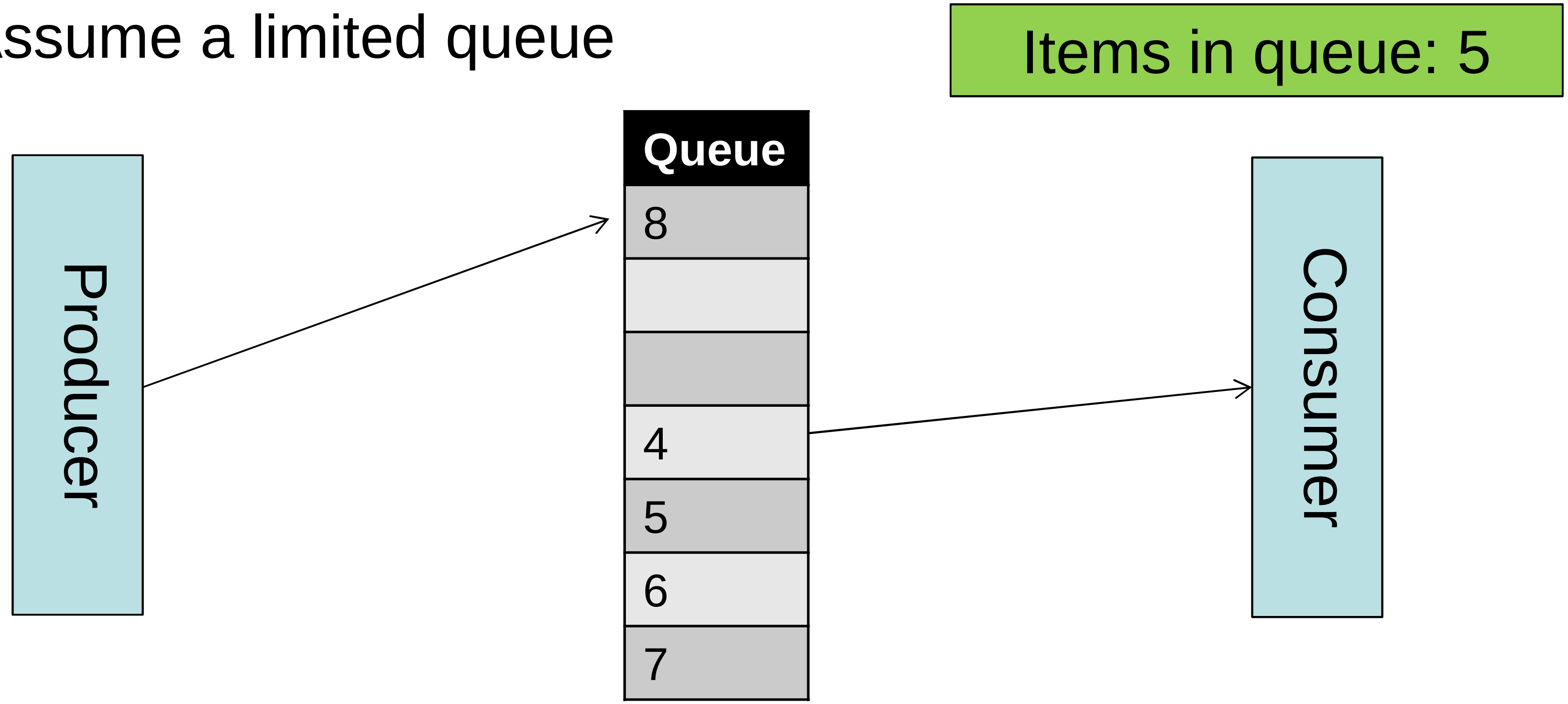
- Assume a limited queue



- Consumer uses one, so there is one spare space
- Producer could continue now

Rolling buffer

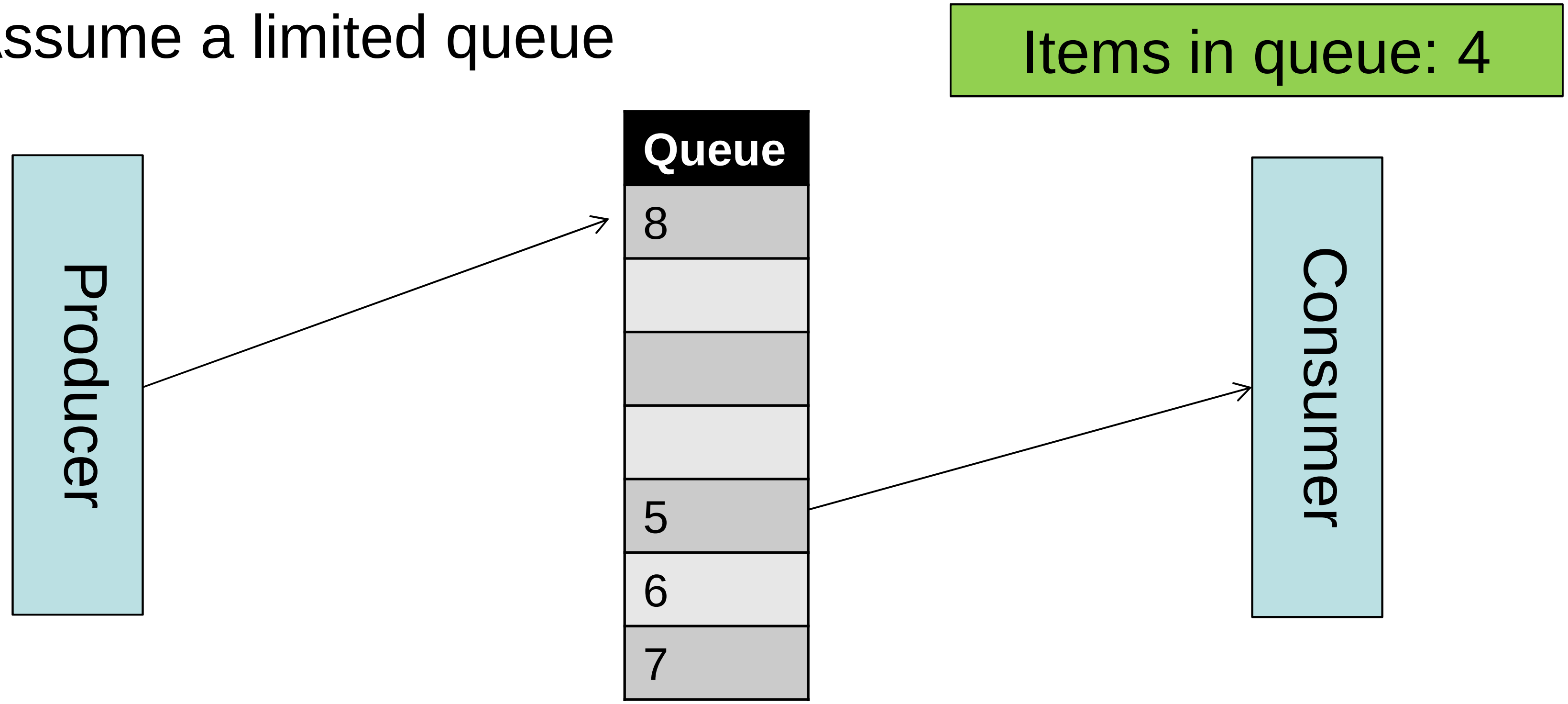
- Assume a limited queue



- Consumer uses another, so there are two spare spaces
- Producer could continue still

Rolling buffer

- Assume a limited queue



- Consumer uses another, so there are three spare spaces now
- Producer could continue still

Question: Semaphore solution?

- Is there any way to count whether the producer can continue?

- Hint: To use a semaphore you want something that will drop to 0 when it has to stop, and be positive when it can continue

Answer

Answer…

- To use a semaphore you want something that will drop to 0 when it has to stop, and be positive when it can continue?
  – The number of spare spaces has this property
- Initially the number of spare spaces is the size of the buffer
- As items are produced, the spaces go down by one
- As items are consumed, the spaces go up again, by one at a time

Producer-Consumer Semaphores

Producer

- SpacesSemaphore.p()
  - Decrement spaces if there is at least one, or block
- Put item into buffer
- ProductSemaphore.v()
  - Record that an item is ready for production

Consumer

- ProductSemaphore.p()
  - Decrement number of product if there is one
- Remove item from buffer
- SpacesSemaphore.v()
  - Record that an extra space is now available

Two semaphores:

SpacesSemaphore: Initial = number of spaces = buffer size

ProductSemaphore: Initial = 0 (no items initially)

Summary

- Semaphores are useful for implementing counters
- Always count DOWN to ZERO
- Block if attempting to get a lock / call p() / call wait() if the counter is already zero
- You can implement them using any mutual exclusion code or atomic operations
  - Need access to the counter variable from all threads though
- Often built-in operating system support
- No guarantee of which waiting thread will be awoken when the count goes up
  - Note: Java implementation supports first-come-first-served too
- Consumer-producer model is easy to enforce using semaphores – unlimited or limited buffer space

Next lecture

- Deadlock and livelock situations
- Inter-process communication
- Synchronous vs asynchronous communication
- Window messages again
- Sockets