# G52OSC
# OPERATING SYSTEMS AND CONCURRENCY

## Java Critical Sections and Monitors

Dr Jason Atkin

# Last Lecture

- Java
  - Thread Creation
    - Anonymous classes
  - Memory model
  - Atomic variables
  - Volatile
    - Memory gates

# Spin locks with `volatile`

```
// Process 1
init1;
while(true) {
  // entry protocol
  c1 = true;
  turn = 2;
  while (c2 && turn == 2);
  crit1;
  // exit protocol
  c1 = false;
  rem1;
}
```

```
// Process 2
init2;
while(true) {
  // entry protocol
  c2 = true;
  turn = 1;
  while (c1 && turn == 1);
  crit2;
  // exit protocol
  c2 = false;
  rem2;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn = 1;
```

Try the sample and you should find that it worked, without having to add any explicit memory barriers

# Equivalent behaviour - optimising

```
// Process 1
init1;
while(true) {
  // entry protocol
   c1 = true;
   turn = 2;
   while (c2 && turn == 2);
   crit1;
   // exit protocol
   c1 = false;
   rem1;
}
```

```
// Process 1
init1;
while(true) {
  // entry protocol
   turn = 2;
   c1 = true;
   c1 = false;
   while (c2 && turn == 2);
   crit1;
   // exit protocol
   rem1;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn = 1;
```

Equivalent behaviour
In a single-threaded process

# Volatile

- In C volatile stops caching of the variable in local memory
  – Forces the value to be written to memory and/or read from memory immediately
- In Java it does more
  – Prevents the local caching
  – AND tells compiler not to re-order the accesses to volatile variables
  – AND writes op codes into the program to stop the processor re-ordering the operations

# Coursework and Labs

- Note: the four labs will take a while to finish
  - This is deliberate, but will make the coursework shorter
- A 15% coursework for a 20 credit module (equivalent 30% coursework for 10 credit) could take a while to complete
- I included the time to do the labs, which may take you longer than the coursework itself
- You learn how to do things in the labs, then apply your knowledge to the coursework (part 1)
  - **Learning and understanding takes time**
  - **Do not leave doing the labs until the last minute!!!**

# This Lecture

- Fairness and spin locks
- Synchronisation
  - Locks
  - Wait/notify
- Monitors

# Reminder: Peterson's algorithm

```
// Process 1
init1;
while(true) {
  // entry protocol
   c1 = true;
   turn = 2;
   while (c2 && turn == 2);
   crit1;
   // exit protocol
   c1 = false;
   rem1;
}
```

```
// Process 2
init2;
while(true) {
  // entry protocol
   c2 = true;
   turn = 1;
   while (c1 && turn == 1);
   crit2;
   // exit protocol
   c2 = false;
   rem2;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn = 1;
```

To work the algorithm depends upon variables being set in this order and no reordering occurring

# Fairness

- A *weakly fair* scheduling policy guarantees that if a process requests to enter its critical section (and does not withdraw the request), the process will *eventually* enter its critical section
  - i.e. wait long enough – don’t stop waiting
- A *strongly fair* scheduling policy guarantees that if a process makes enough requests it will eventually succeed
  - Does not have to keep waiting
  - Can do something else between requests

# Properties of the Java scheduler

- Java makes ***no*** promises about scheduling or fairness, and does not even strictly guarantee that threads make forward progress:
  - Most Java implementations display some sort of weak, restricted or probabilistic fairness properties with respect to executing runnable threads
  - **However you can't depend on this**
- There may be no guarantee that the any specific thread has a chance to run

# `Thread` priorities

- **`Threads`** have *priorities* which **heuristically** influence schedulers:
  – Each thread has a priority in the range `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`
  – By default, each new thread has the same priority as the thread that created it
  – The initial thread associated with a main method by default has priority `Thread.NORM_PRIORITY`
  – The current priority of a thread can be accessed by the method `getPriority` and set via the method `setPriority, e.g.:`

```
Thread.currentThread().setPriority(Thread.MAX_PRIORITY)
```

- When there are more runnable threads than CPUs, a scheduler is generally biased in favour of threads with higher priorities
  – But it may not make much difference

# Archetypical mutual exclusion

- We assumed that:
  - the initialisation, critical sections and remainder may be of any size and may take any length of time to execute–each may vary from one pass through the `while` loop to the next;
  - **the critical sections must execute in a finite time; i.e., each process must leave its critical section after a finite period of time**
  - the initialisation and remainder of each process may be infinite
- **If the critical sections don't execute in finite time, the scheduling policy cannot be weakly fair (others could wait for infinite time)**

Java

**synchronized**

# Mutual Exclusion in Java

- Every object in Java can be used as a Mutex
  - And as a condition variable – see later
- Use the keyword **`synchronized`** for this purpose
  - Notice the American spelling!!!   z not s.
- You ‘synchronize’ on an object
  - Either the current object (this)
  - Or you give it an object explicitly
- Mutual exclusion is ensured between any two pieces of code which are **`synchronized`** on the same object (functions synchronize on **`this`**)

# synchronized functions

```
synchronized void procedure1()
{
      for ( int i = 0 ; i < 1000000 ; i++ )
      {
            ++v;
      }
      System.out.println( "Procedure 1 Finished\n" );
}

synchronized void procedure2()
{
      for ( int i = 0 ; i < 1000000 ; i++ )
      {
            ++v;
      }
      System.out.println( "Procedure 2 Finished\n" );
}
```

# **synchronized** blocks

```
void procedure1()
{
        for ( int i = 0 ; i < 1000000 ; i++ )
        {
                synchronized ( this )
                {
                        ++v;
                }
        }
        System.out.println( "Procedure 1 Finished\n" );
}
```

… etc for the other procedure

- The code within the block is synchronized, rather than the entire function

# Java `synchronized`

- You can consider a synchronized block/function to work as a mutex lock
  - Obtain lock in order to enter the block/function
  - Release the lock automatically when leaving the block/function
- Same problems as mutexes
  - E.g. deadlock issues (next slide)
- Avoids you having to remember to unlock the mutex
- You can use different objects if you want different mutexes to protect different critical sections

# Deadlock example

```
class SynchronizedClass {
   synchronized void f1( SynchronizedClass ob )
   {  ob.f2(this);     }

   synchronized void f2( SynchronizedClass ob )
   {  i = 2;      }
}

SynchronizedClass o1 = new SynchronizedClass();
SynchronizedClass o2 = new SynchronizedClass();

public void go()
{
   new Thread() { public void run() {
      for ( int i = 0 ; i < 100 ; i++ ) o1.f1(o2);
            } }.start();
   new Thread() { public void run() {
      for ( int i = 0 ; i < 100 ; i++ ) o2.f1(o1);
            } }.start();
}
```

`f1()` on the first object calls `f2()` on the second object

Both methods are synchronized

o1 is first
o2 is second

o2 is first
o1 is second

# Reasons for deadlock

**Thread 1**

```
for ( int i = 0
      ; i < 100
      ; i++ )
      o1.f1(o2);
```

- Calls f1 on o1
    - Locks o1
  - Calls f2() on o2
    - Locks o2

**Thread 2**

```
for ( int i = 0
      ; i < 100
      ; i++ )
      o2.f1(o1);
```

- Calls f1 on o2
    - Locks o2
  - Calls f2() on o1
    - Locks o1

# Reasons for deadlock

**Thread 1**

```
for ( int i = 0
      ; i < 100
      ; i++ )
      o1.f1(o2);
```

- Calls f1 on o1
    - Locks o1
  - Calls f2() on o2
    - Locks o2

**Thread 2**

```
for ( int i = 0
      ; i < 100
      ; i++ )
      o2.f1(o1);
```

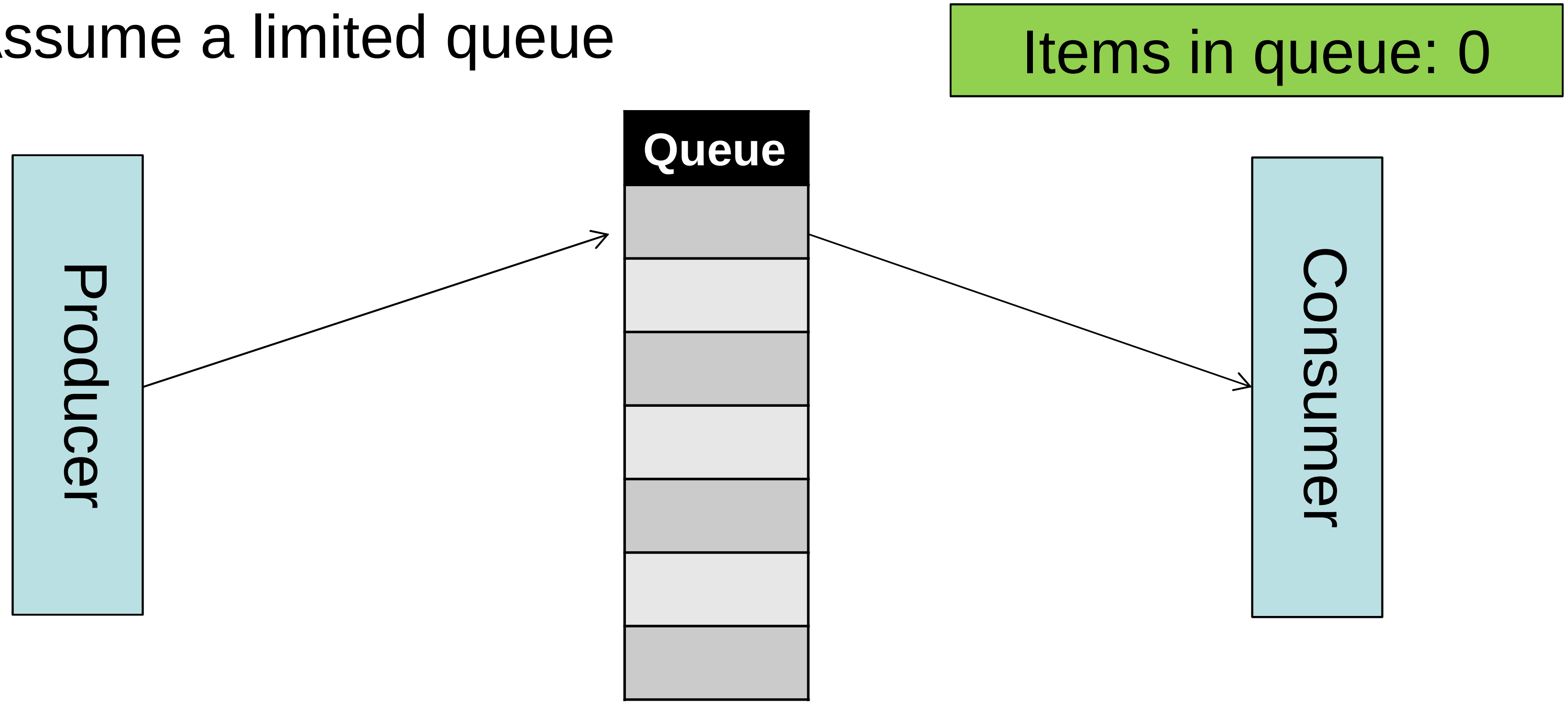
- Calls f1 on o2
    - Locks o2
  - Calls f2() on o1
    - Locks o1

# Data encapsulation

- Advantage of synchronized functions:
  - You can make your data private
  - Make all methods which access it synchronized
  - Then you will never have issues with interference between threads
- Java goes further than this, allowing simple wait and signal/notify facilities
  - And more complex facilities to implement more realistic monitors – later
- **Example: Producer-Consumer problem**

# Rolling buffer

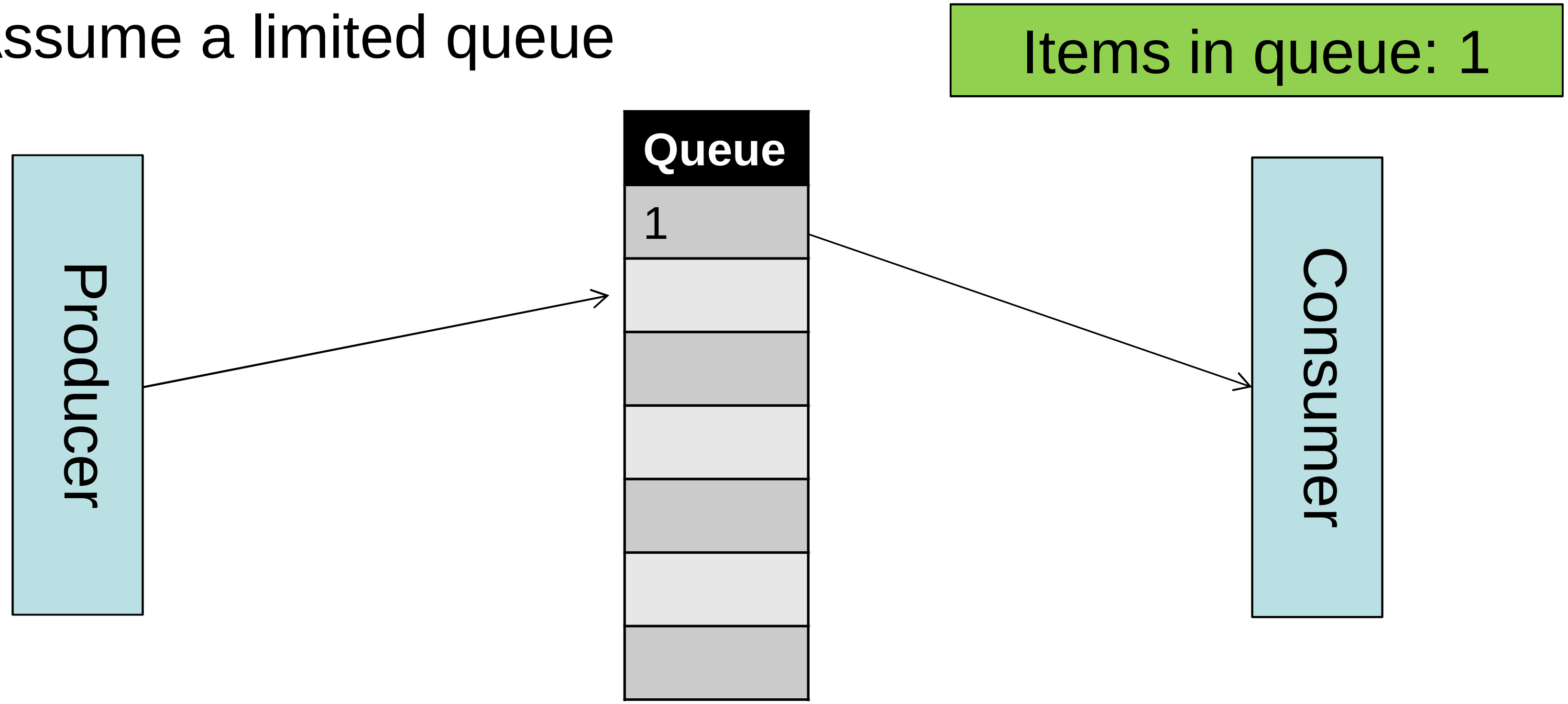
- Assume a limited queue




- When queue is filled, it rolls around, to start again
- As long as there is enough room
  - i.e. consumer has consumed some by then

# Rolling buffer

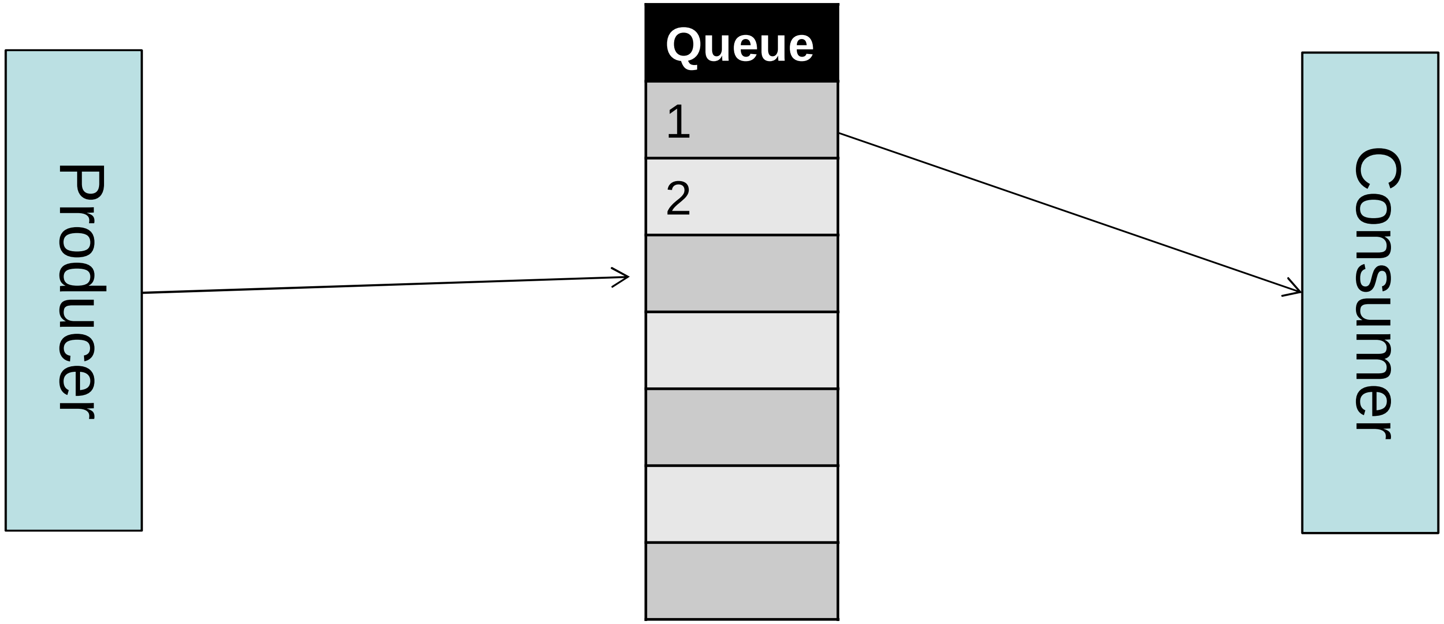
- Assume a limited queue



# Rolling buffer

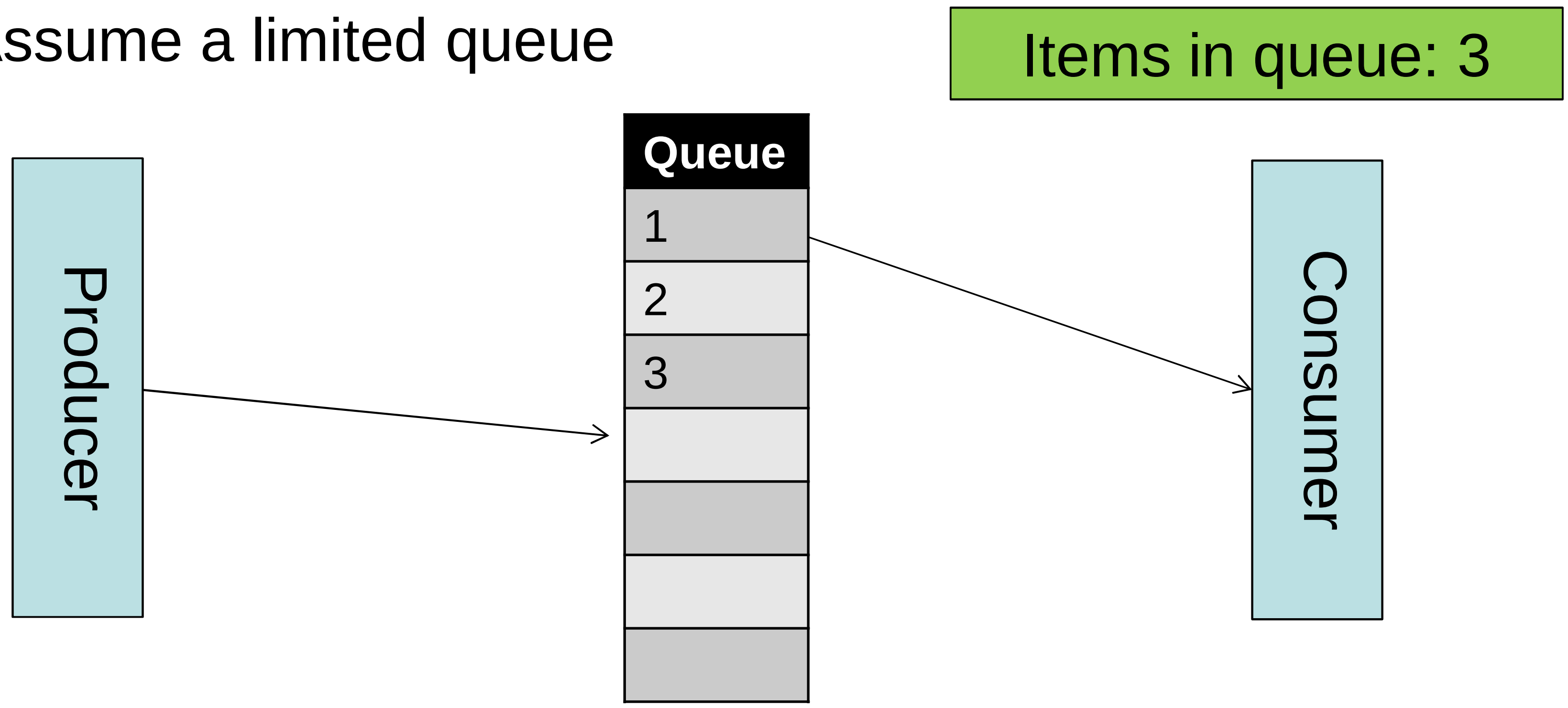
- Assume a limited queue

Items in queue: 2



# Rolling buffer

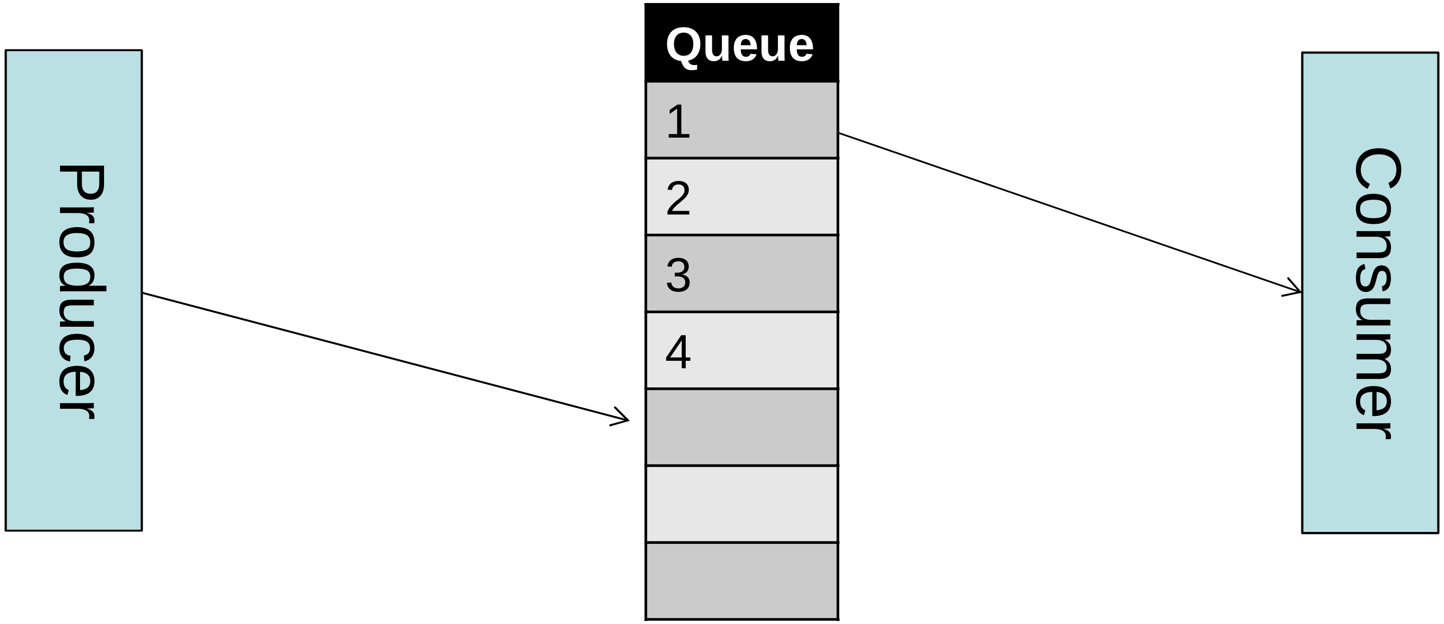
- Assume a limited queue



# Rolling buffer

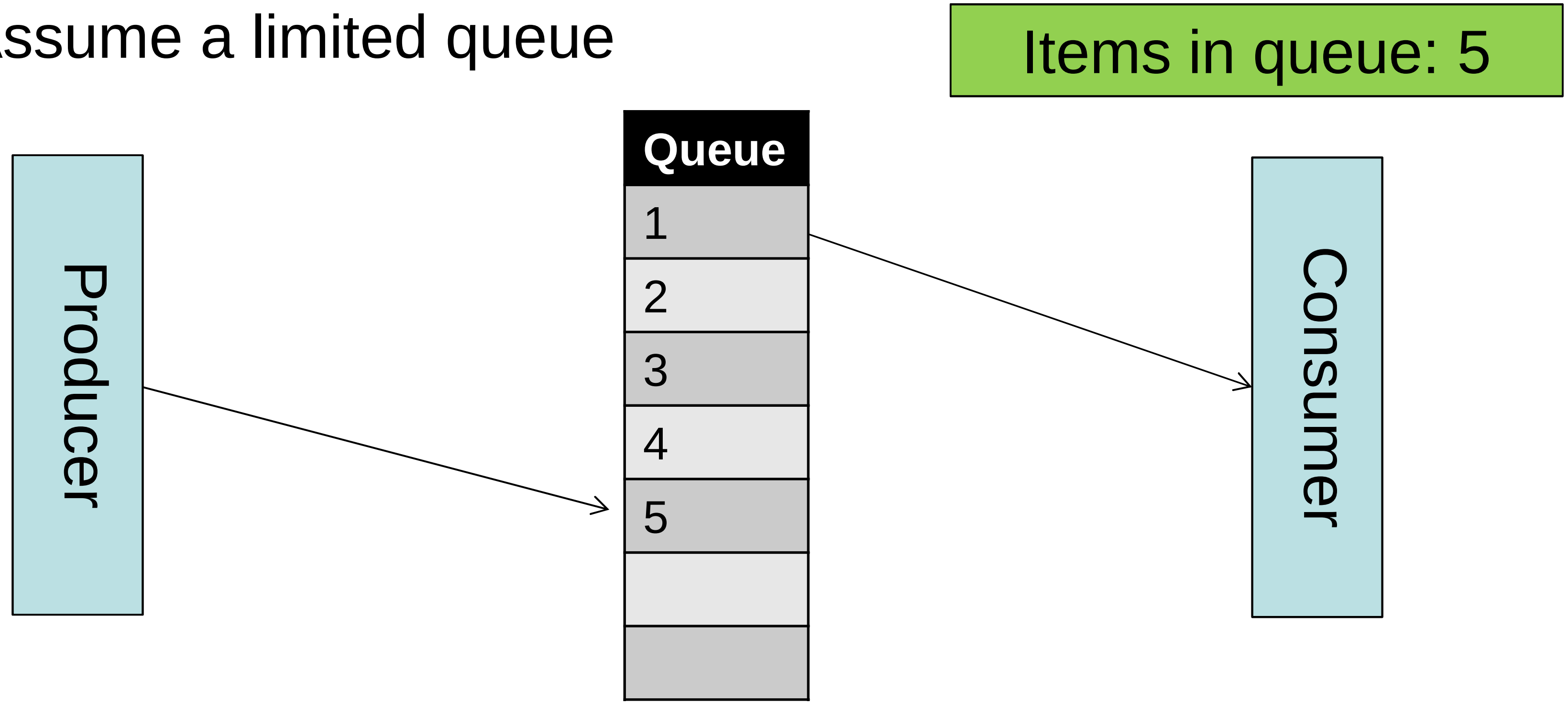
- Assume a limited queue

Items in queue: 4



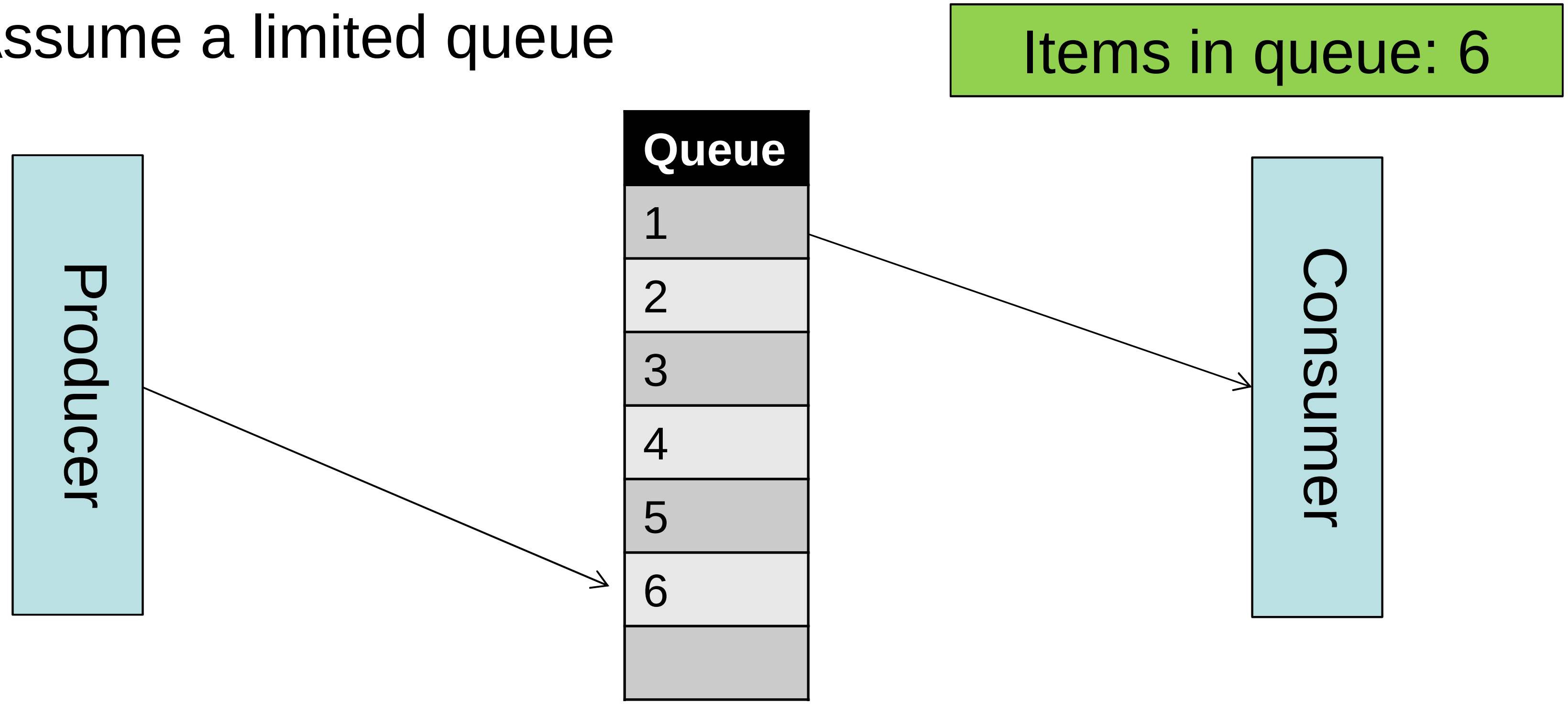
# Rolling buffer

- Assume a limited queue



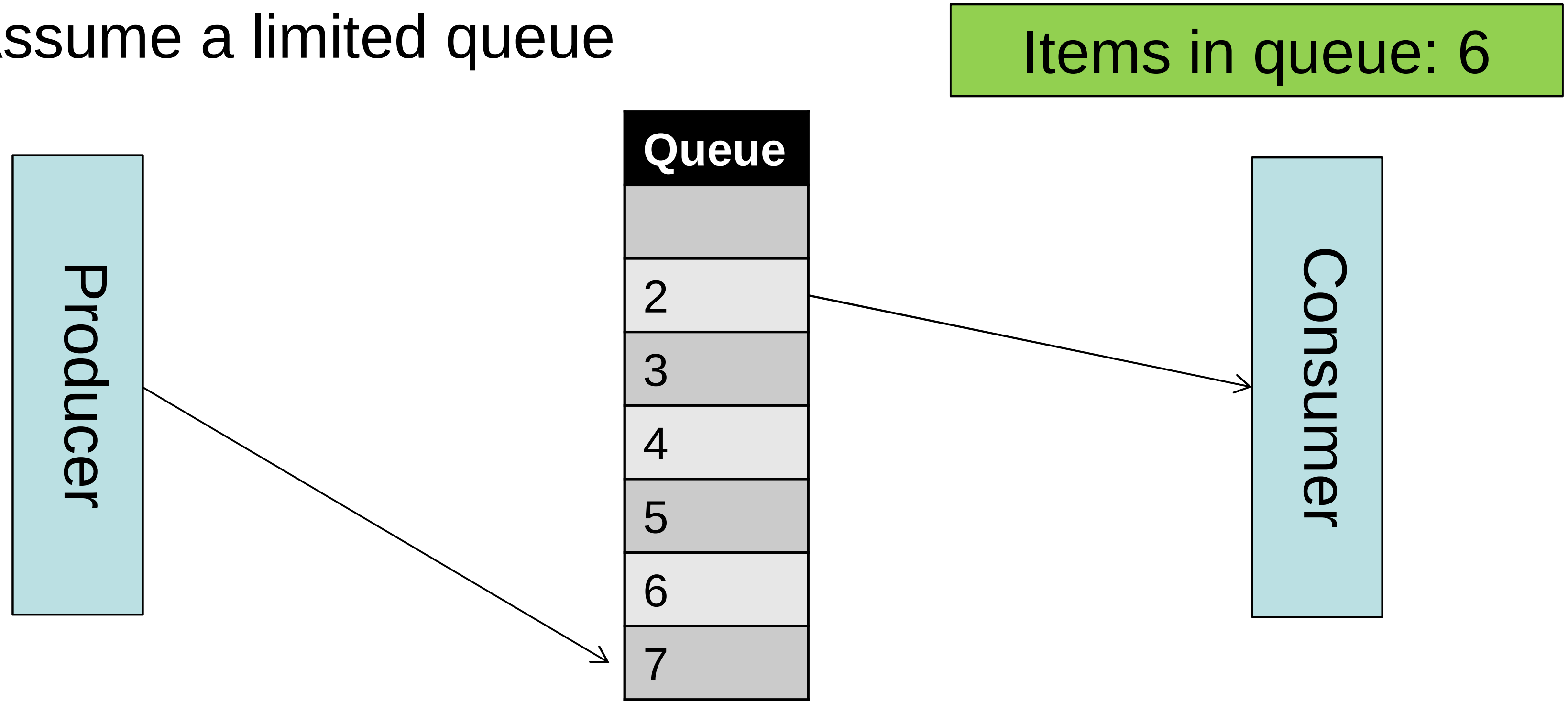
# Rolling buffer

- Assume a limited queue



# Rolling buffer

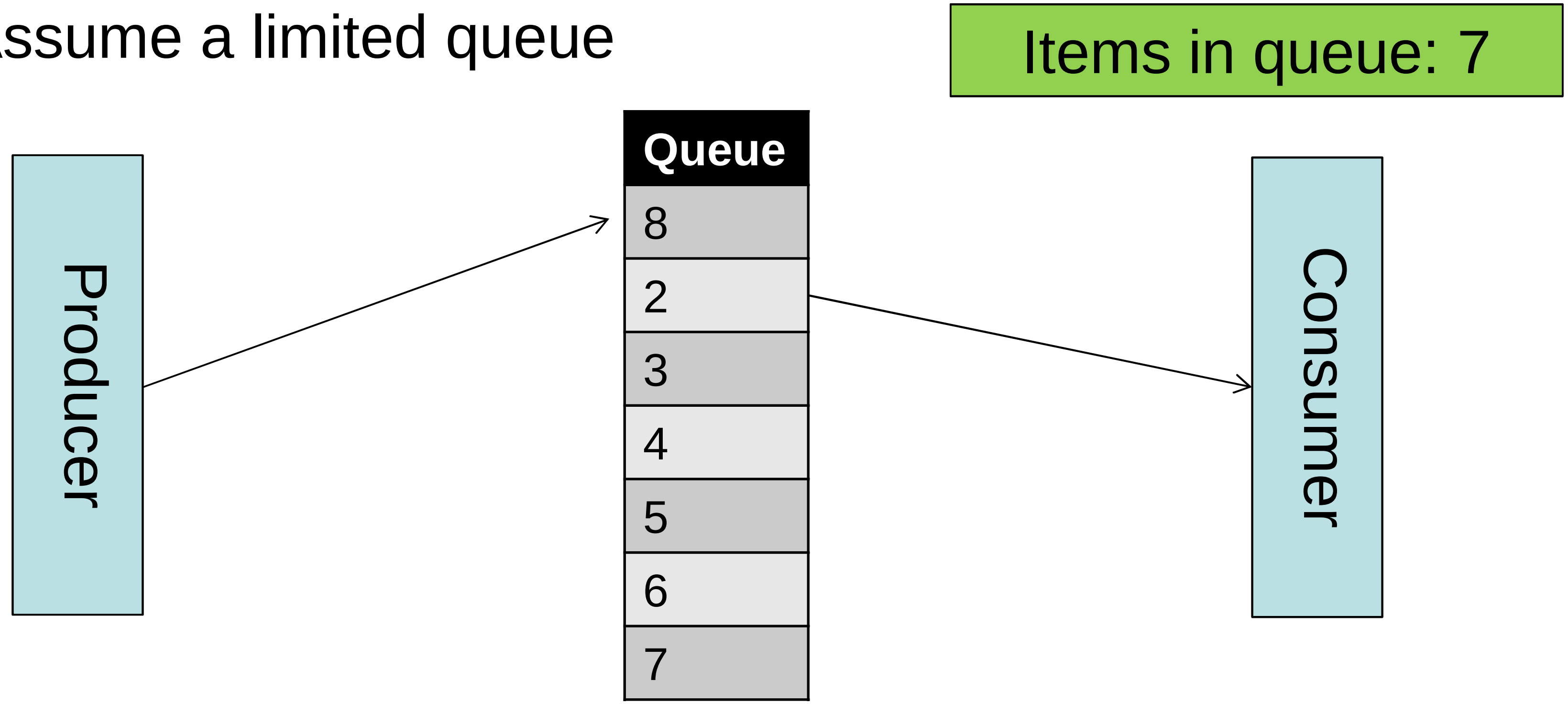
- Assume a limited queue




- Assume that consumer finally got around to consuming one

# Rolling buffer

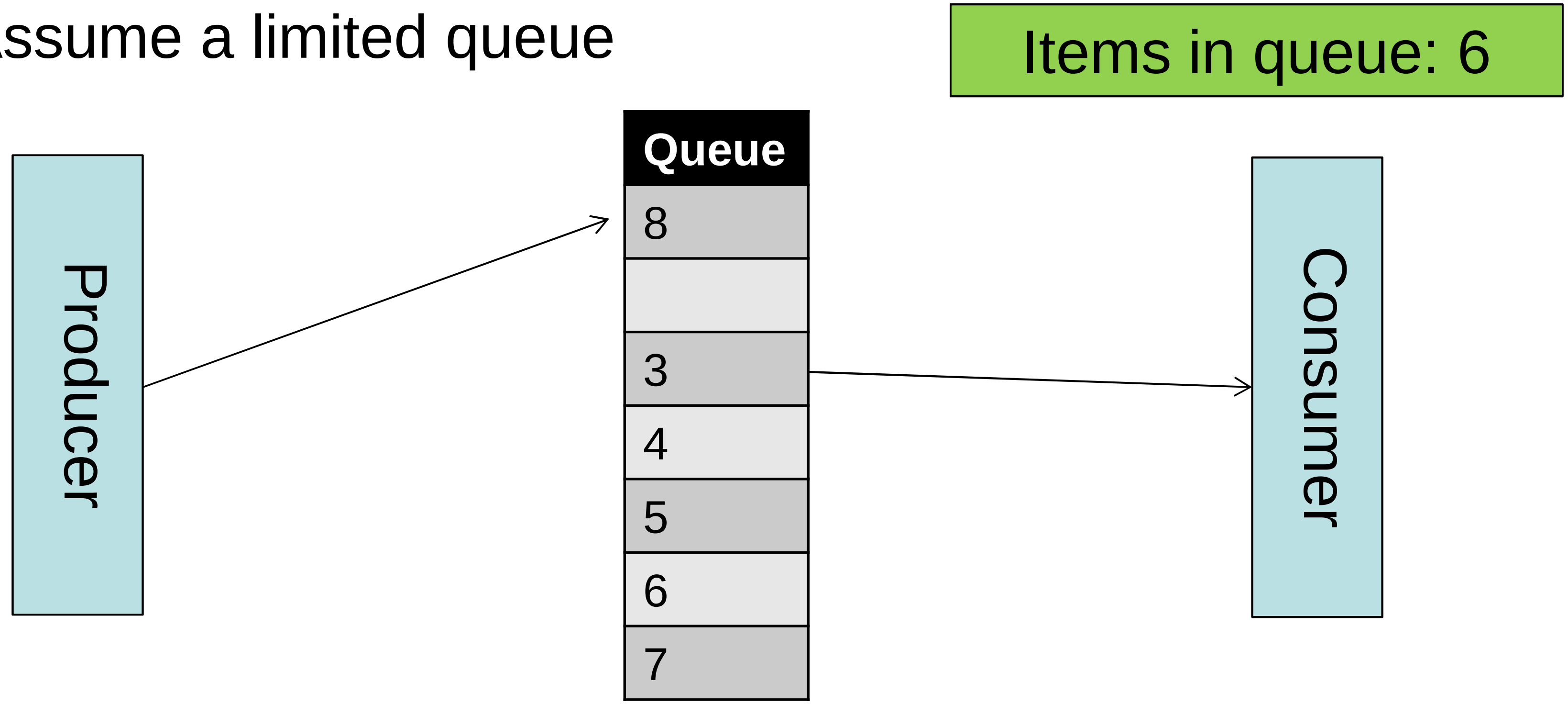
- Assume a limited queue




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

# Rolling buffer

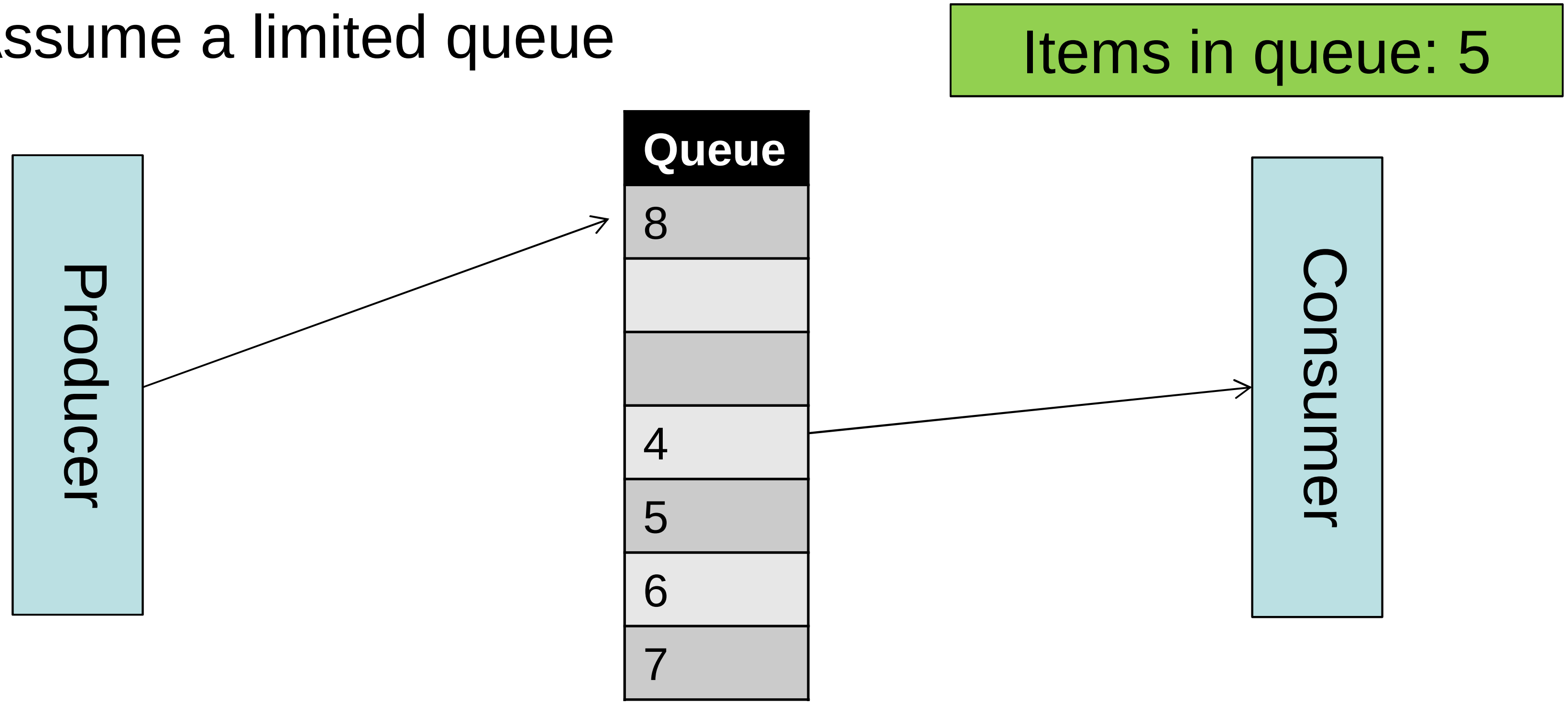
- Assume a limited queue




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

# Rolling buffer

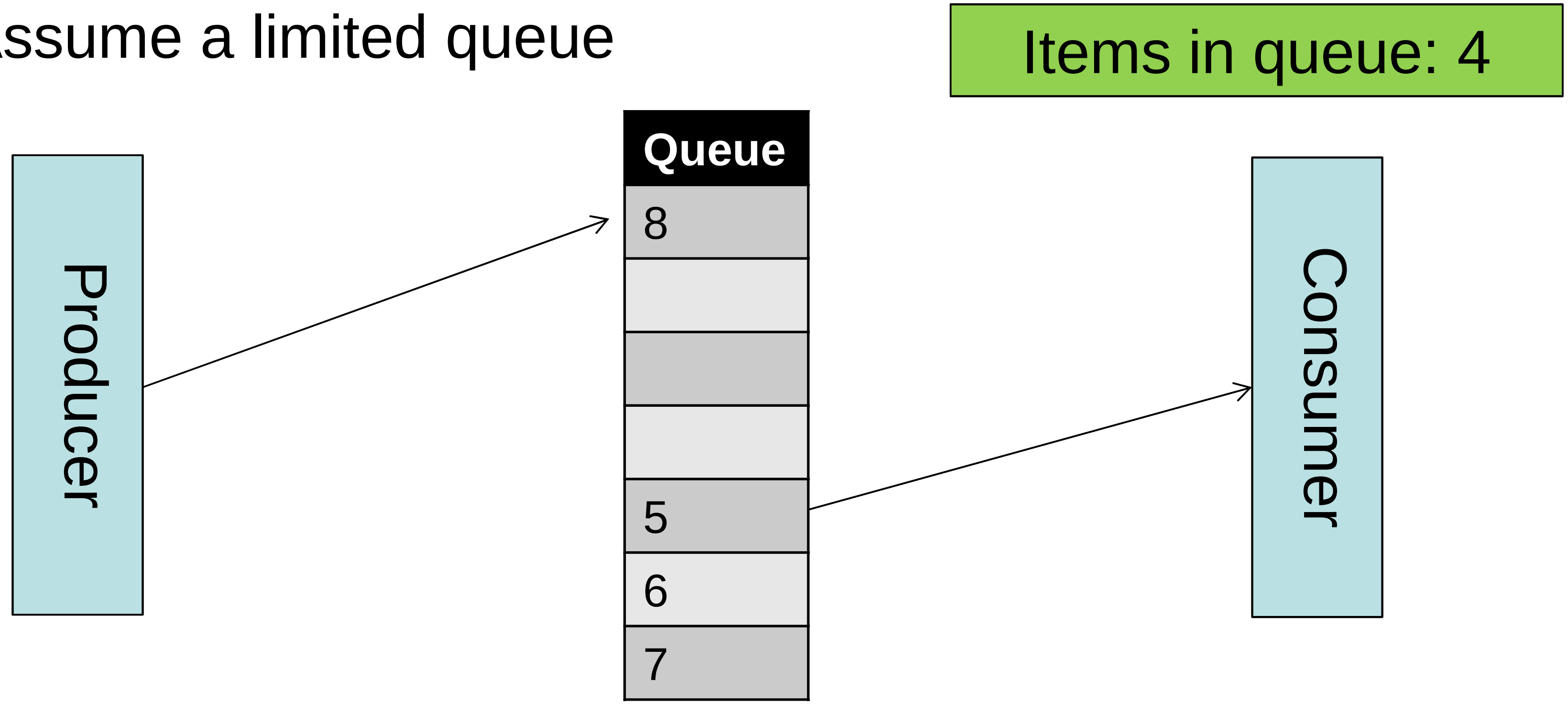
- Assume a limited queue




- 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

# Example source code extract

```
final int BUFFER_SIZE = 8;
volatile int iCount = 0;
volatile int iInsertionPosition = 0;
volatile int iRemovalPosition = 0;
volatile int[] arrayItems = new int[BUFFER_SIZE];

synchronized boolean ProduceItem( int i  )
{
       if ( iCount >= BUFFER_SIZE )  // Is there any space?
              return false;

       arrayItems[iInsertionPosition] = i;   // Store item
       ++iInsertionPosition;         // Update next insert posn
       if ( iInsertionPosition >= BUFFER_SIZE ) // Wrap around
              iInsertionPosition = 0; // Back to start

       ++iCount; // Increment count of items
       return true;
}
```

Returns failure if no space

Note: could loop, checking the variable rather than return but then we don't release the lock!

# Similarly for the consumer…

```
synchronized int ConsumeItem()
{
        if ( iCount <= 0 )
                return -1; // No item to consume
        // Get item
        int iThisItem = arrayItems[iRemovalPosition];

        ++iRemovalPosition; // Increment removal position
        // Wrap around if we get to the end
        if ( iRemovalPosition >= BUFFER_SIZE )
                iRemovalPosition = 0;

        --iCount; // Reduce count of items available

        return iThisItem;
}
```

Returns failure if no item

# Overview of usage

## Producer

- **While there is no space**
  - **Wait and then try again**
- Produce item
- Increment count of items

## Consumer

- **While there are no items**
  - **Wait and then try again**
- Retrieve/consume item
- Decrement count of items

- Producer can only produce items when there is space to do so
- Consumer can only consume items when there are items on the queue
- Problem is the spinning at the beginning
- **It would be useful to be able to wait for an item to be produced (or space to be available) rather than to keep trying**

# Wait and notify/signal let us do that

- Enter the synchronized block
- Check your condition to continue
- If you have to wait, call wait()
  - wait() releases the lock that you have until it is awoken
  - When it ‘wakes up’ again it will have to wait to get that lock before continuing
- Anything which changes the condition it is waiting on must call notify, to wake it up again
  - When it wakes up, it must re-check the condition
  - If it is still not true then go back to sleep ( wait() )
- You can have multiple condition variables (queues)
  - Basic Java implementation assumes just one condition variable
  - Condition variable is associated with an object – wait on object
  - You have to wait() on the same object you synchronized on

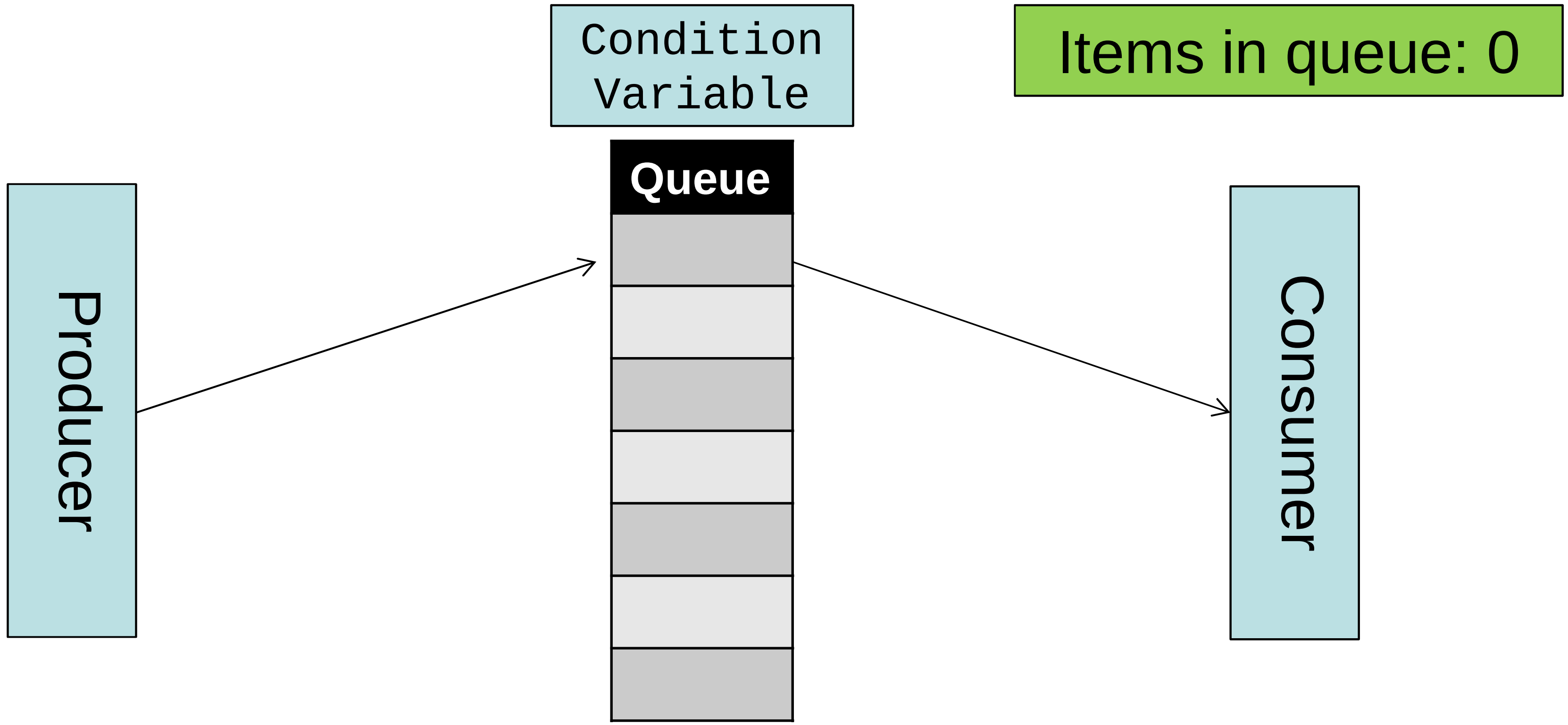
# Consumer

```
synchronized int ConsumeItem()
{
   // Do the wait-notify block first
   while ( iCount <= 0 )
      try { wait(); } catch (InterruptedException e) {}
   // Note: doing this.wait() to wait on current object

   int iThisItem = arrayItems[iRemovalPosition]; // Get item

   ++iRemovalPosition; // Increment removal position
   if ( iRemovalPosition >= BUFFER_SIZE ) // Wrap around?
      iRemovalPosition = 0;

   --iCount; // Decrement count of items stored

   // Tell any waiting producers that it is worth carrying on now
   notifyAll();

   return iThisItem;
}
```

# Producer

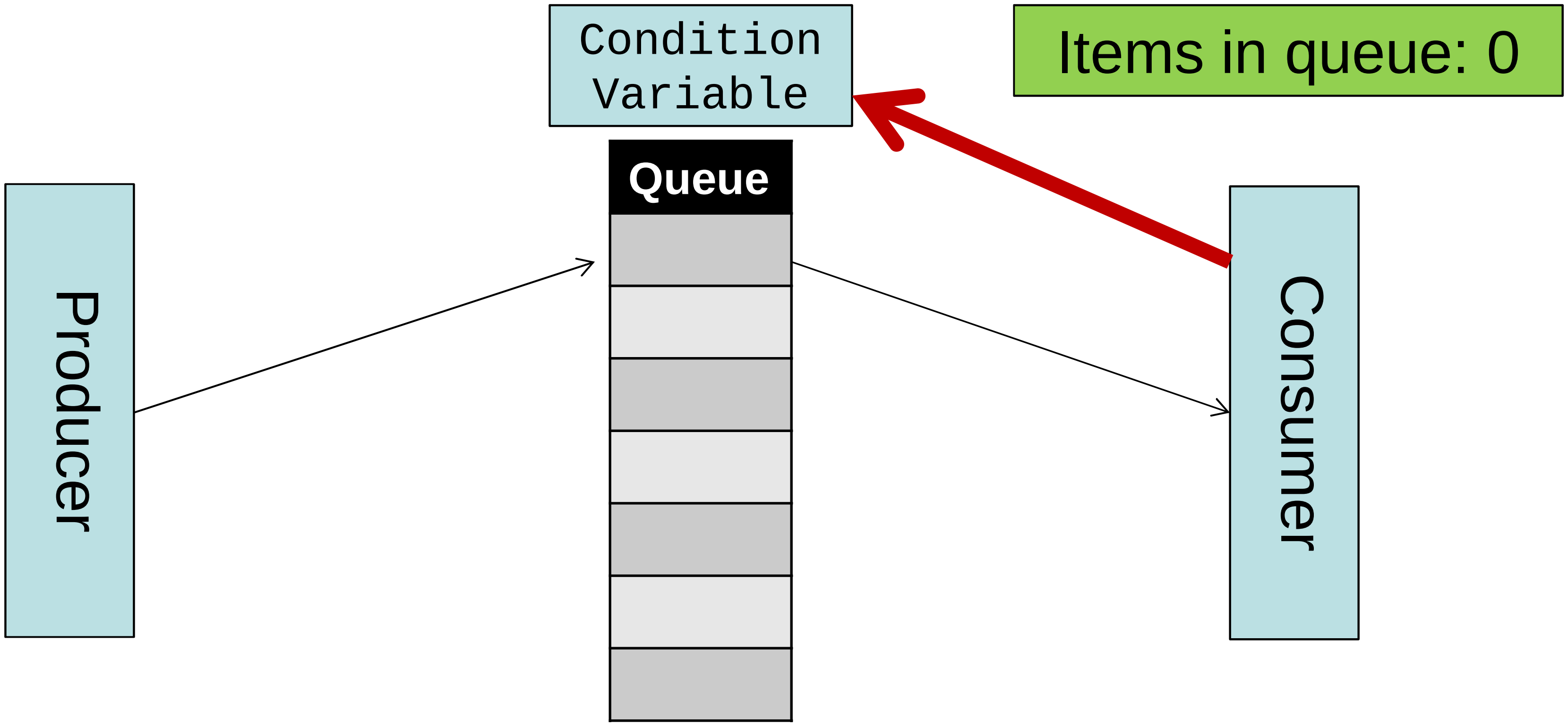
```
synchronized boolean ProduceItem( int i )
{
   // Do the wait-notify block first
   while ( iCount >= BUFFER_SIZE )
      try { wait(); } catch (InterruptedException e) { }

   arrayItems[iInsertionPosition] = i; // Insert item

   ++iInsertionPosition; // Increment insertion position
   if ( iInsertionPosition >= BUFFER_SIZE ) // Wrap around?
      iInsertionPosition = 0;

   ++iCount; // Increment count of items

   // Tell any waiting consumers that it is worth carrying on now
   notifyAll();

   return true;
}
```

# Producer-Consumer



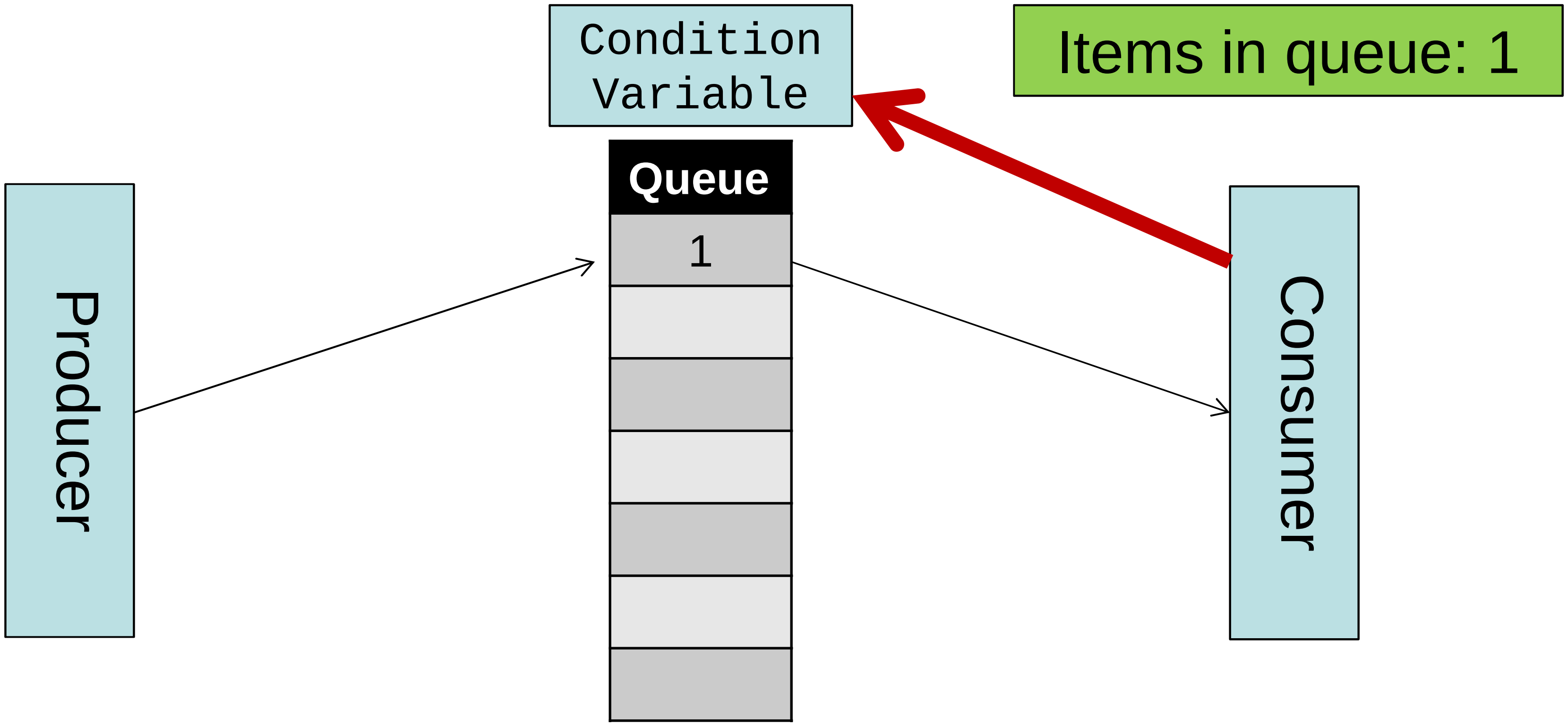

- Consumer tries to consume
- Nothing in the queue …

# Producer-Consumer



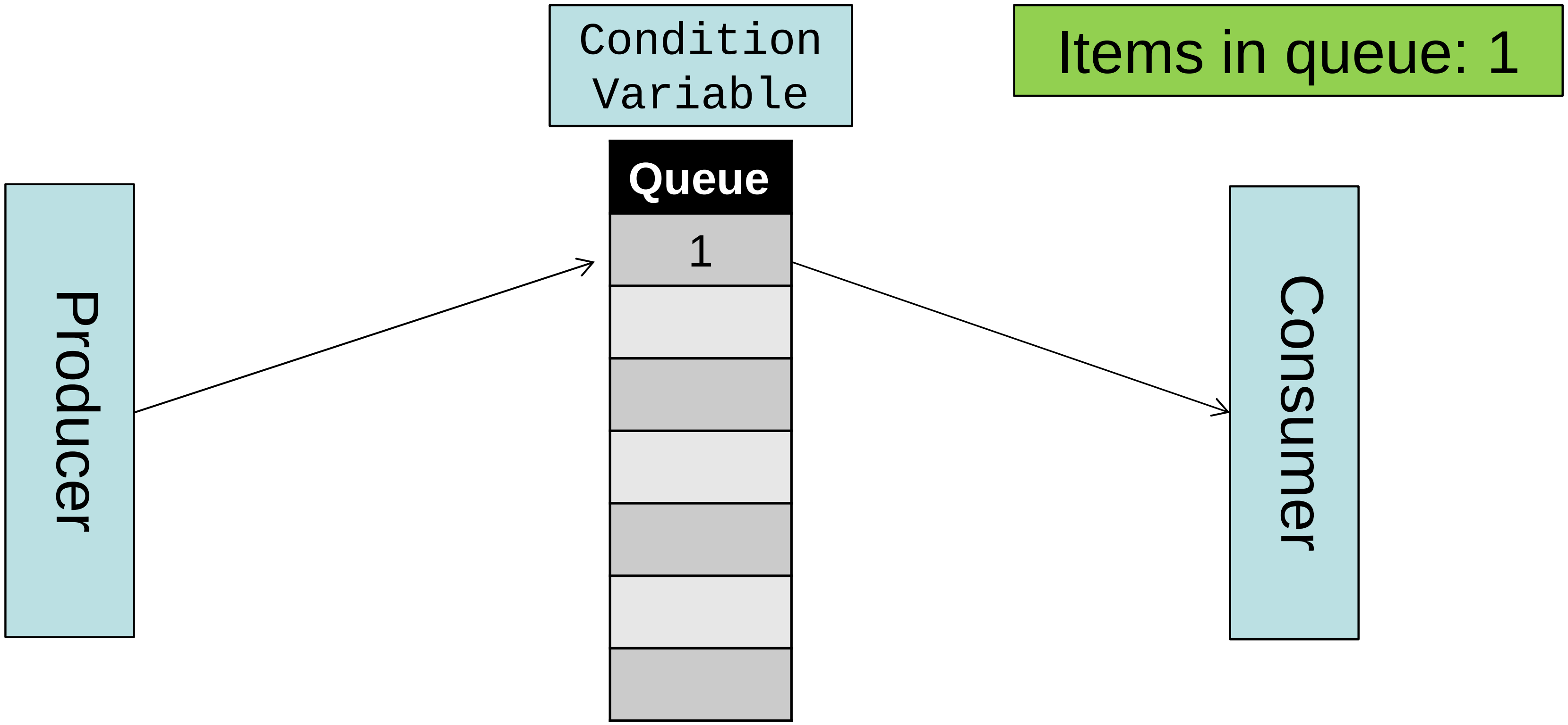

- Consumer tries to consume
- Nothing in the queue
- Consumer issues wait() on the condition variable

# Producer-Consumer



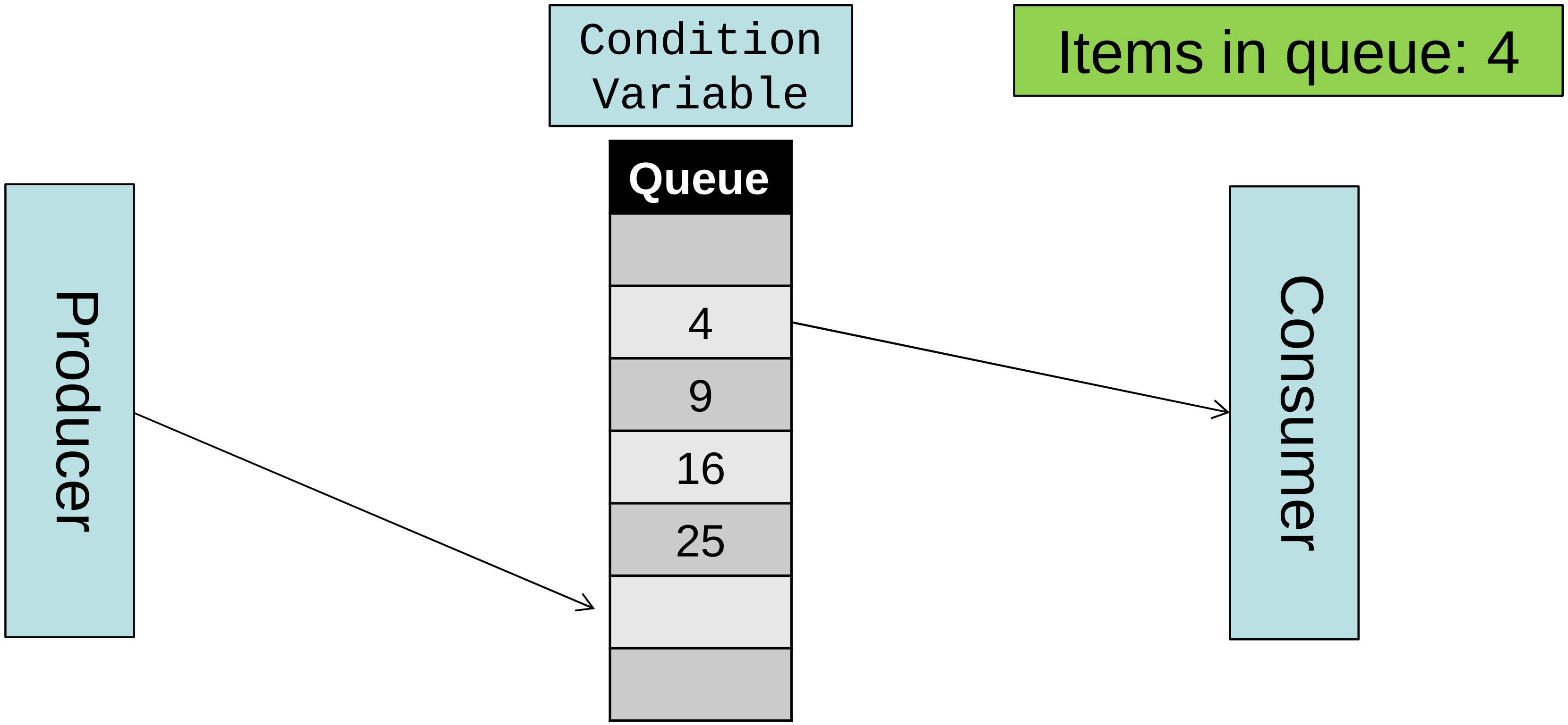

- Producer produces an item
- Producer calls notify() on the condition variable
  - Often called ‘signal’ rather than notify

# Producer-Consumer



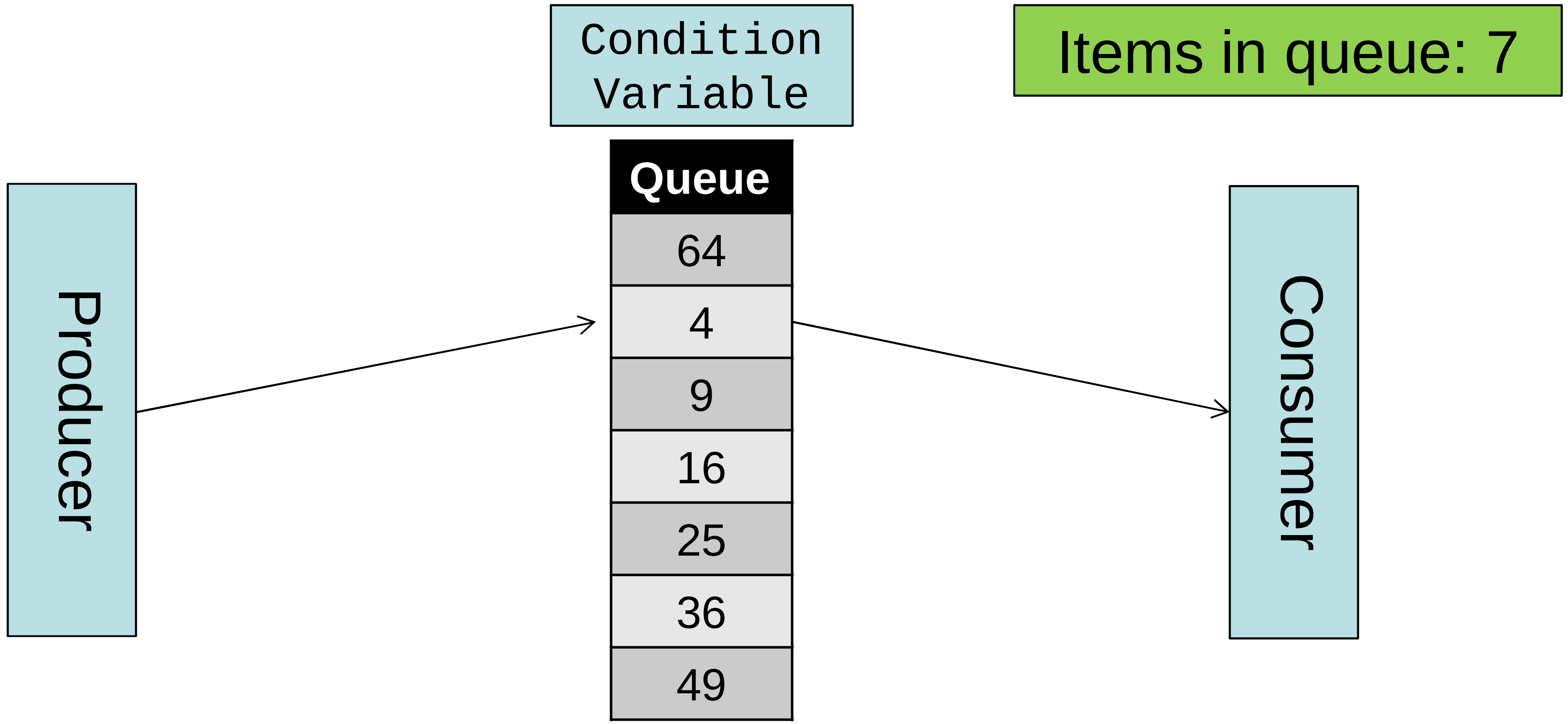

- Consumer wakes up and wants the lock so that it can continue
- When Producer leaves the synchronized function, consumer can enter its own, and consume the item

# Producer-Consumer



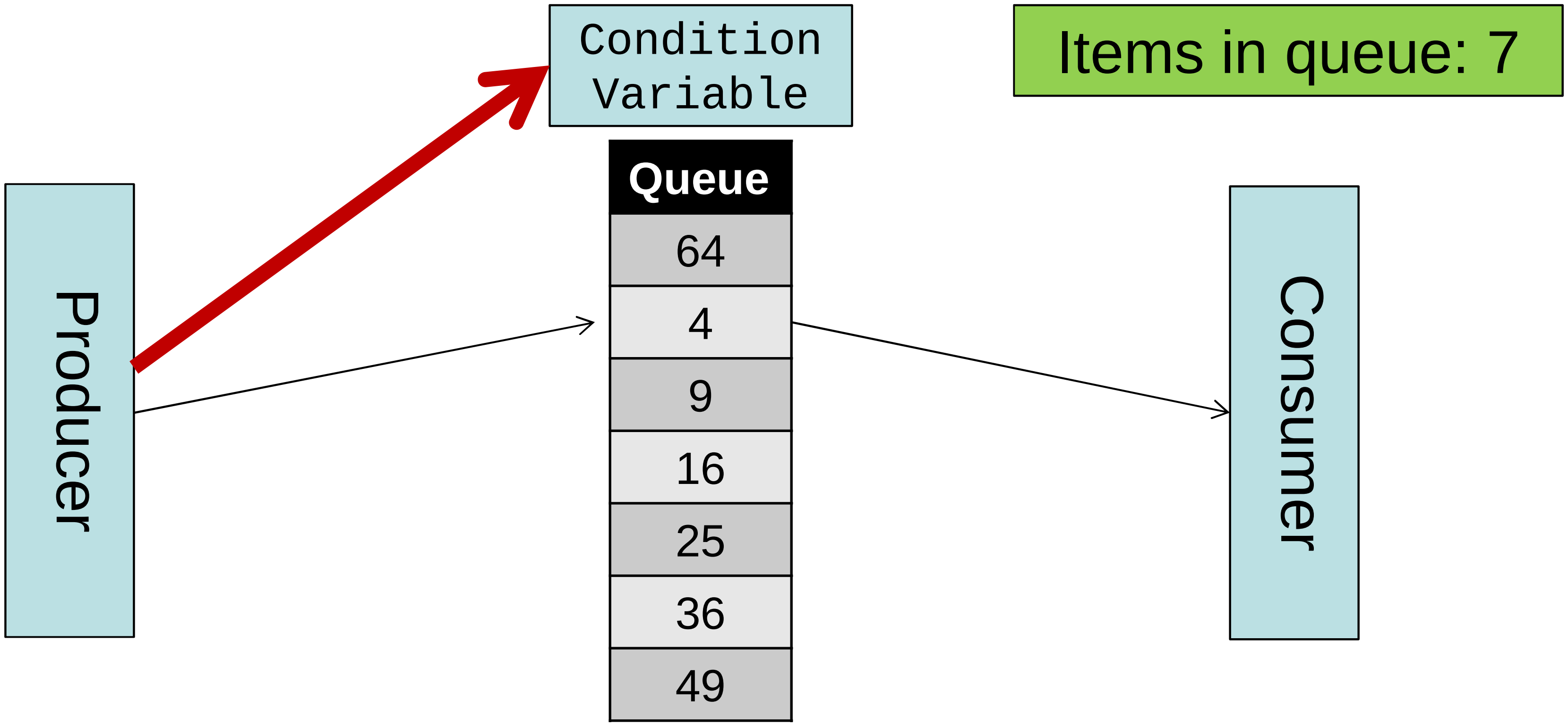

- Producer keeps producing…

# Producer-Consumer



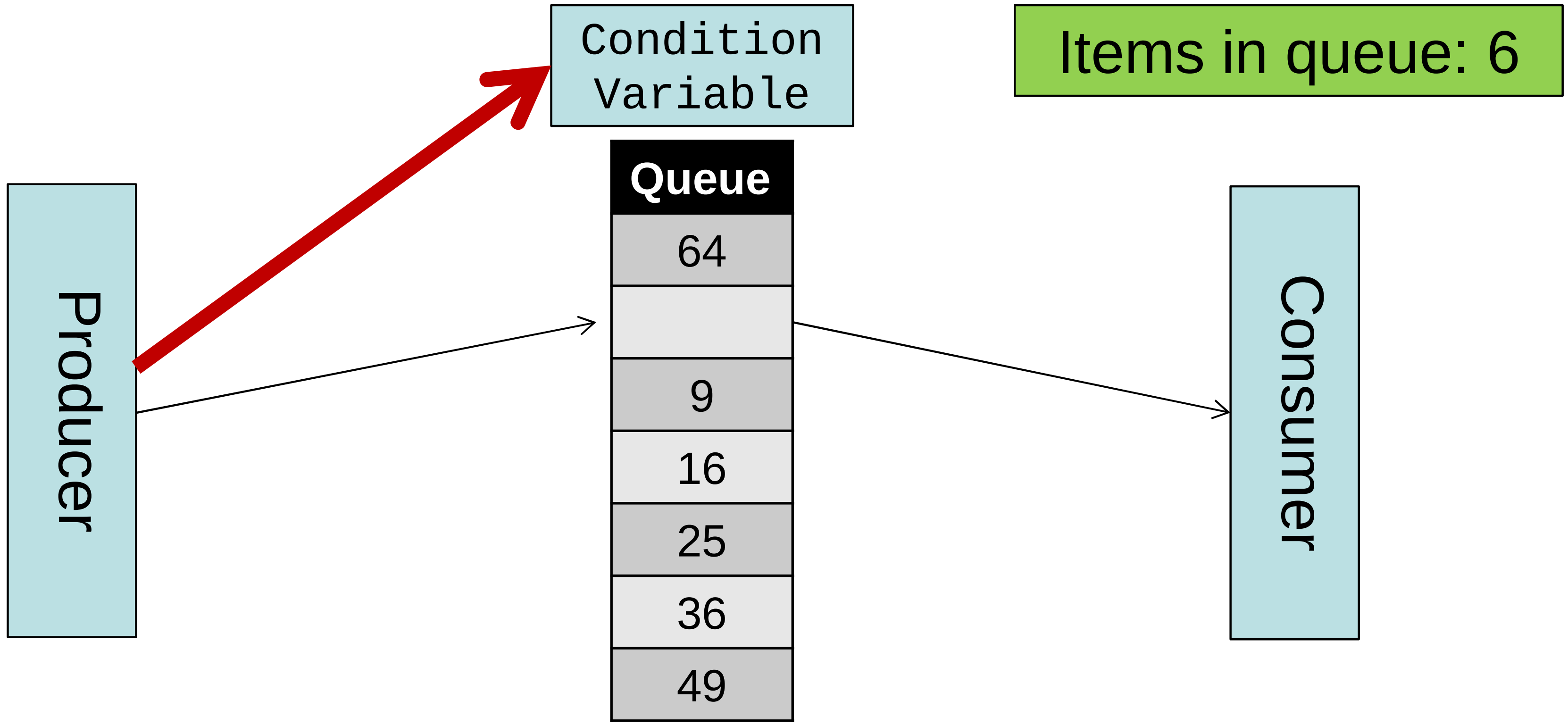

- Producer keeps producing…
- At a later point, the Producer finds that the queue is full…

# Producer-Consumer



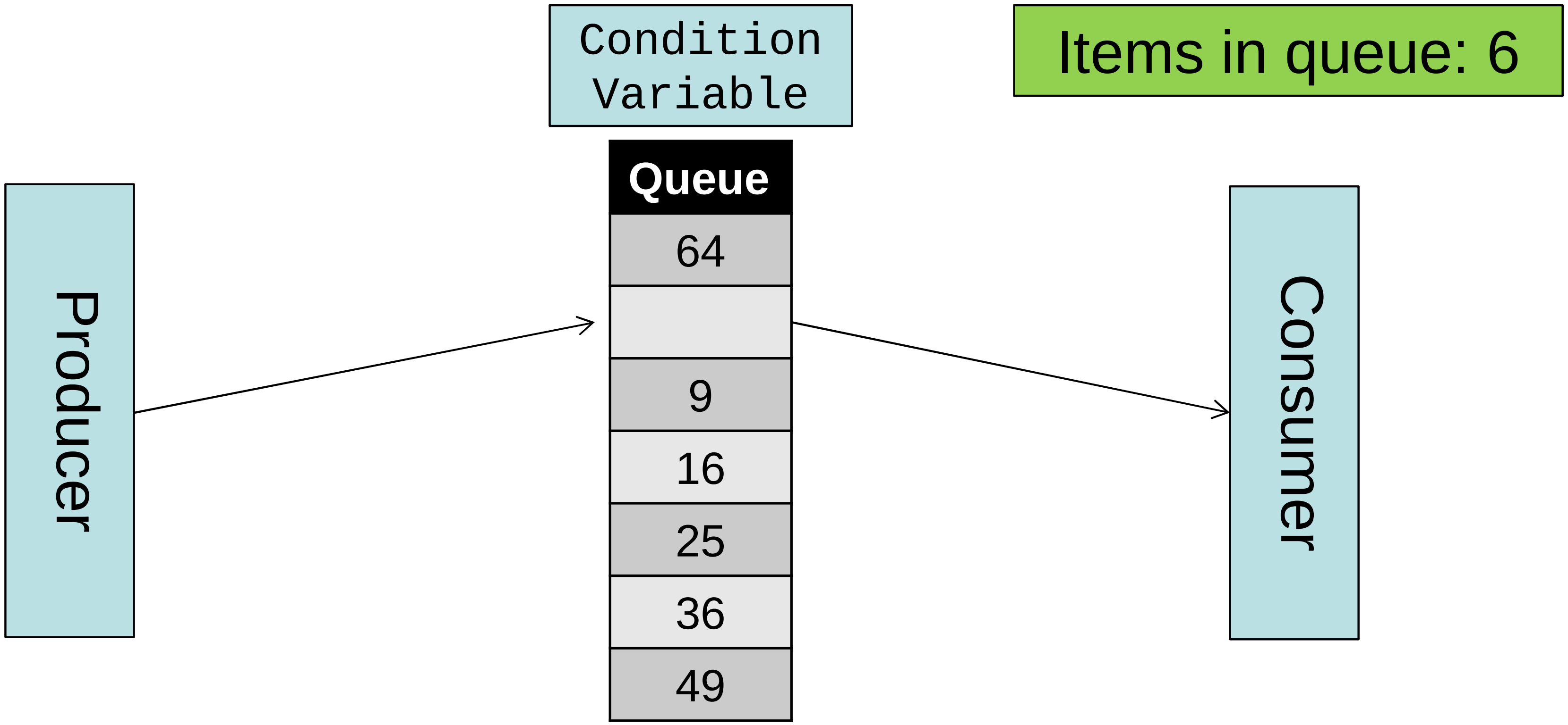

- Producer keeps producing
- At a later point, the Producer finds that the queue is full
- Producer calls Wait() on the condition variable

# Producer-Consumer




- Consumer will eventually consume a product, making room for a new one
- **Every** time consumer has consumed a product it calls notify() on the Condition Variable

# Producer-Consumer




- Consumer calls notify
- Producer is awakened and will check for space
  - When it can get the lock / enter the synchronized section again
  - If still no space it will wait() again

# wait() and notify()

- The thing you wait on is called a **condition variable**
  - In Java any object can be used for this
- In the basic Java implementation you need to have locked the object that you are using as a condition variable (i.e. synchronized on it)
  - i.e. you cannot have more than one condition variable associated with the object
- The wait will unlock it, awakening will re-lock it
- More Java complex concurrency classes allow this though (e.g. ReentrantLock)
- Now we are ready to understand monitors…

# Monitors

# Monitors as abstract data types

- A *monitor* is an abstract data type representing a shared resource and operations to protect and manipulate it
- Monitors (conceptually) *encapsulate* the shared resource
- A monitor implements a shared data structure together with the operations which manipulate the data structure
- Think “private data and public access methods”
- Monitors have four components:
  - A set of *private variables* which represent the state of the resource (the data to protect)
  - A set of *monitor procedures* which provide the public interface to the resource (the functions/methods you can call)
  - A set of *condition variables* used to implement condition synchronisation (e.g. a queue of waiting threads)
  - *Initialisation code* which initialises the private variables

# Next Lecture

- More Monitors
  - The theory
  - Implementing full monitors
- More concurrency in Java