G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Monitors II

Dr Jason Atkin

Design Patterns and Solution Methods

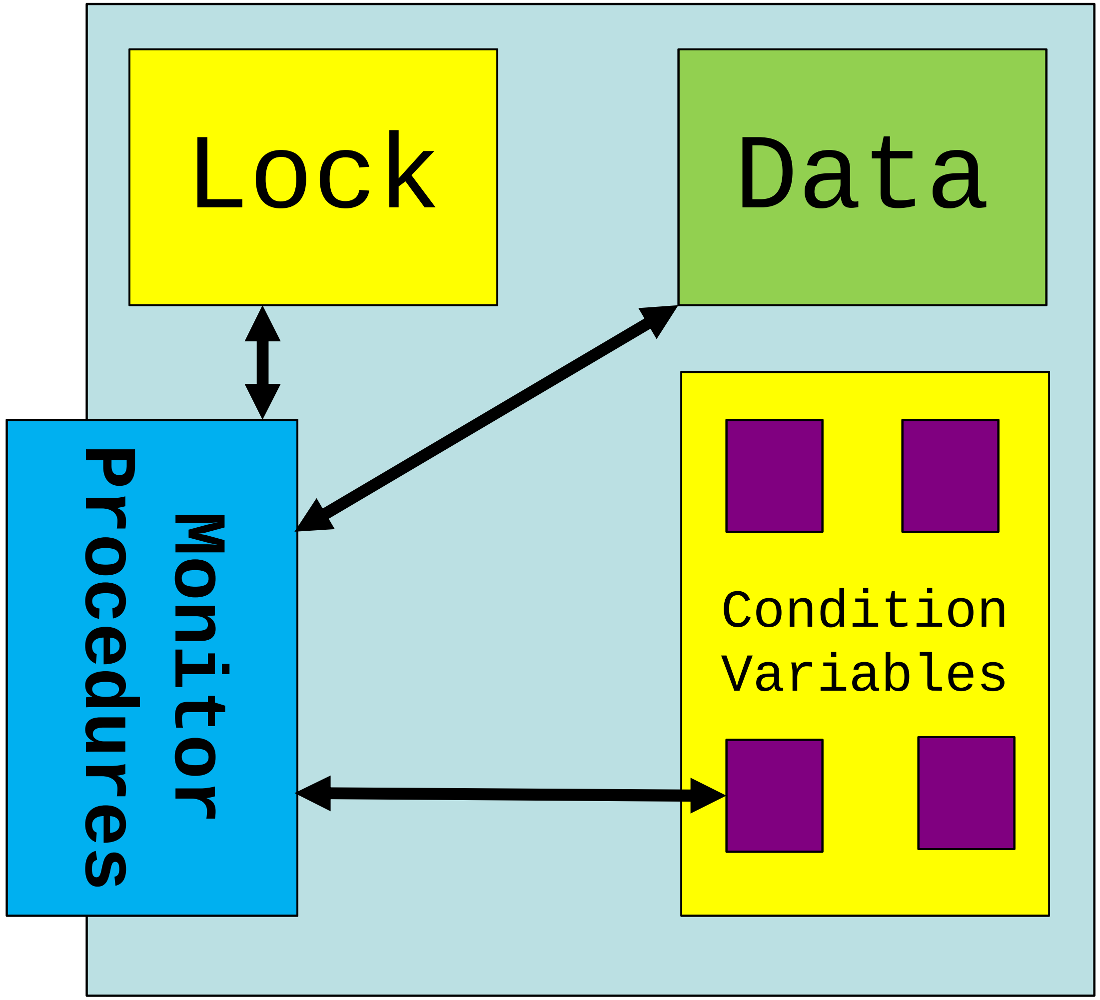
- We have seen a number of concurrency problems:
  - Mutual Exclusion and Critical Sections
  - Asynchronous message passing (PostMessage)
  - Synchronous requests (SendMessage)
  - Limiting by a count (Semaphores)
  - Producer-Consumer
  - Dining Philosophers and Reader-Writer still to come
- And a number of tools to use
  - Mutex (and critical section) objects
  - Semaphores (counts but no ownership)
  - Monitors (lock and wait/notify)
  - Spinlocks
- These represent common ‘design patterns’ that you see in concurrent programs, so have practical applications
- One aim of this module is to recognise the problems and solutions, to avoid working out how to solve them yourself, or making mistakes

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

Structure of a monitor

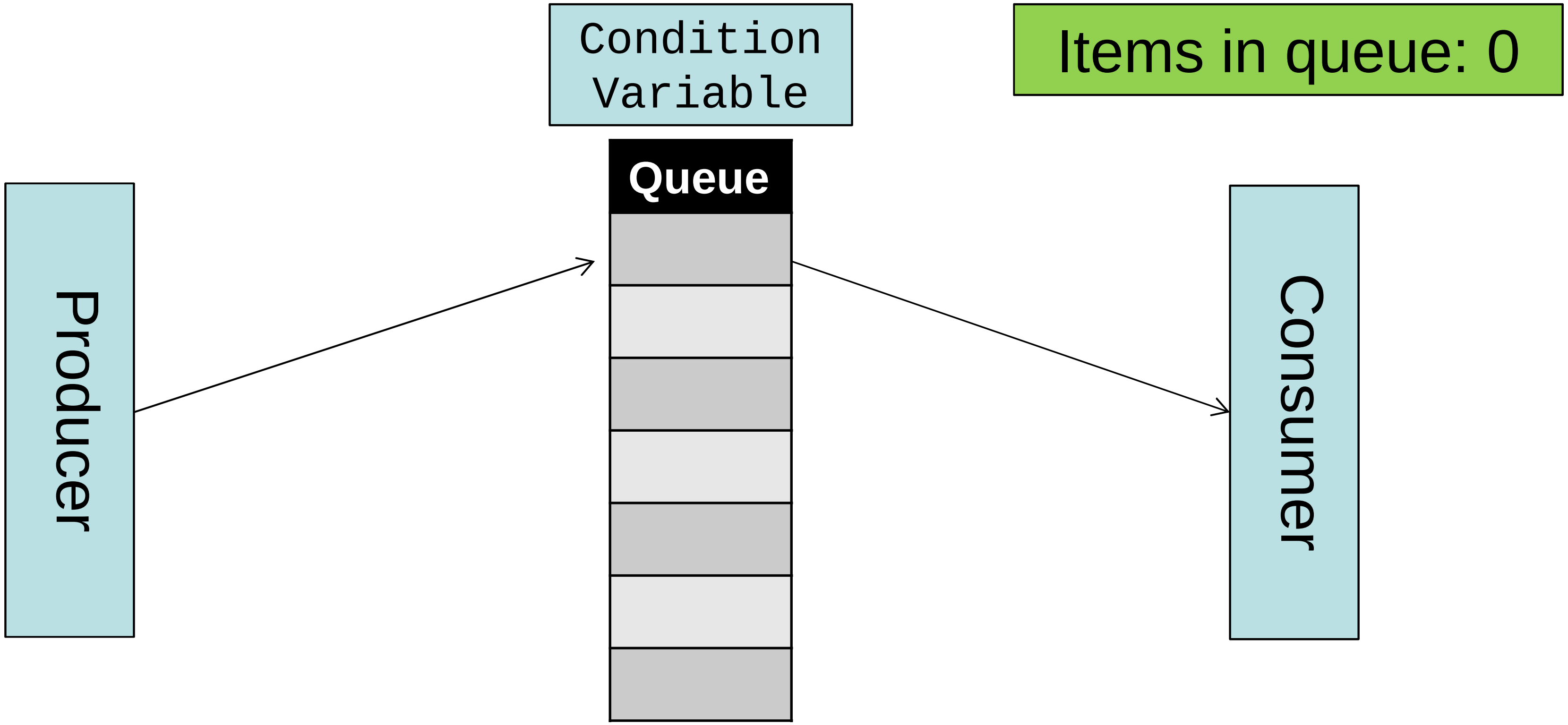
- Private data
- Public methods
- Locks
- Condition variables



Reminder: Consumer

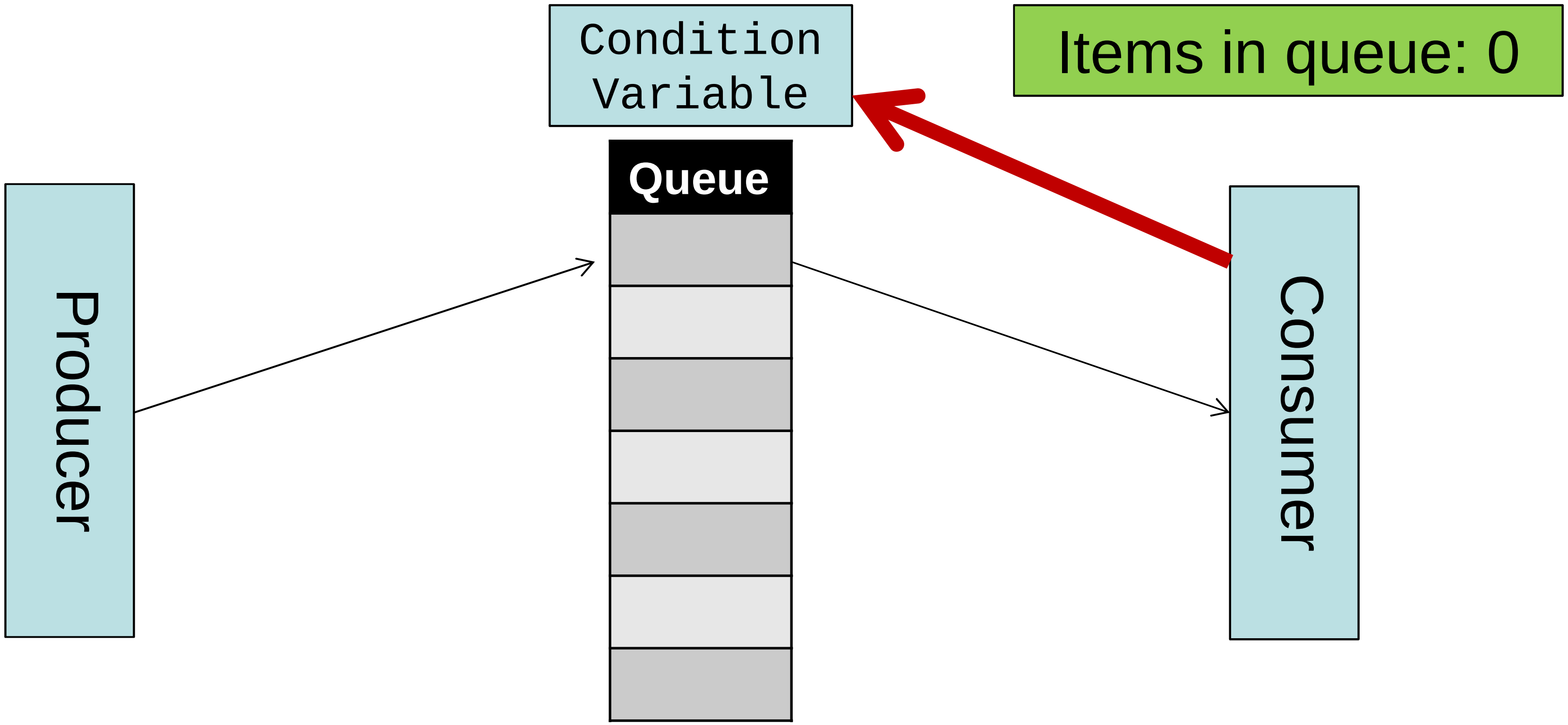
```
synchronized int ConsumeItem()
{
   // Do the wait-notify block first
   while ( iCount <= 0 )
      try { wait(); } catch (InterruptedException e) {}
   // Note: doing this.wait() to wait on current object

   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;
}
```

Reminder: Producer-Consumer



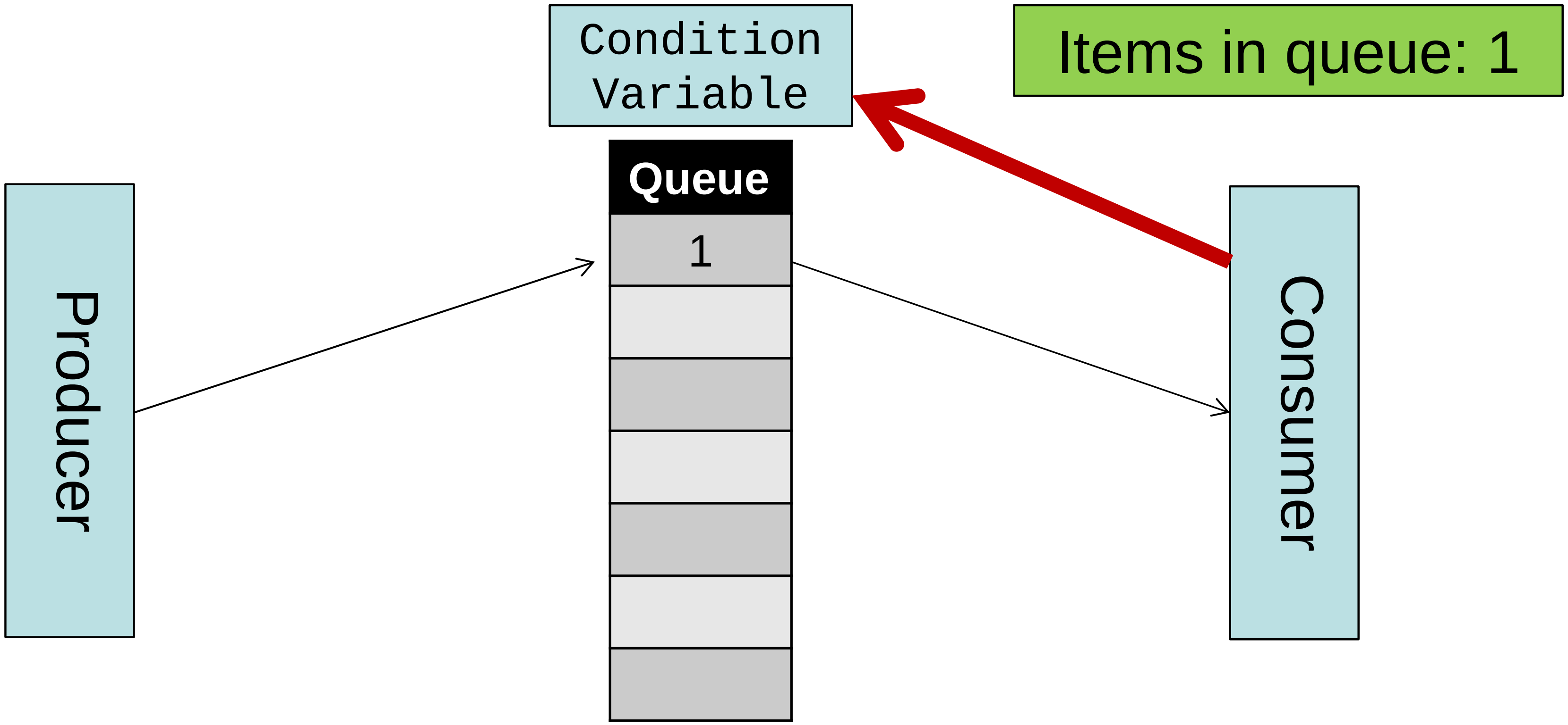
- Consumer tries to consume
- Nothing in the queue …

Reminder: Producer-Consumer



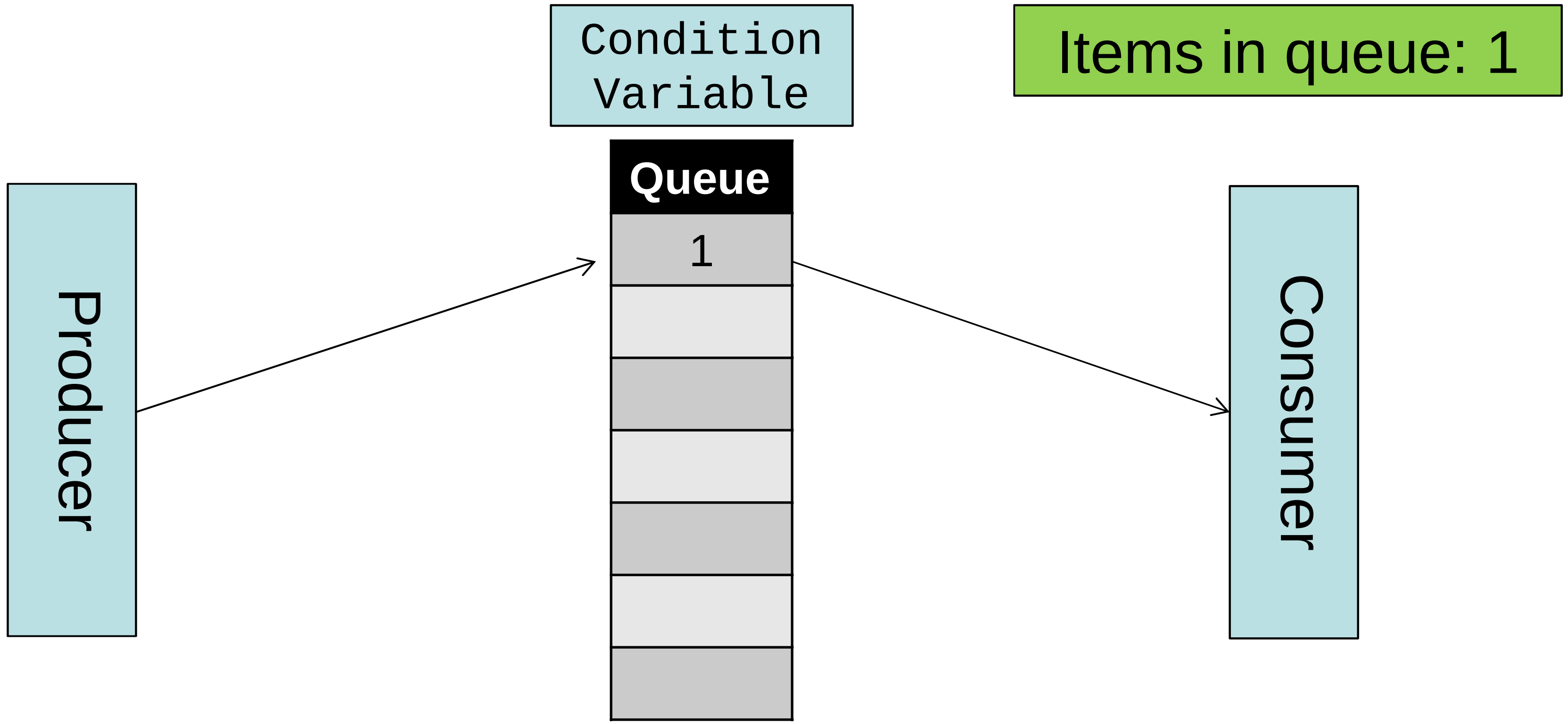
- Consumer tries to consume
- Nothing in the queue
- Consumer issues wait() on the condition variable

Reminder: Producer-Consumer



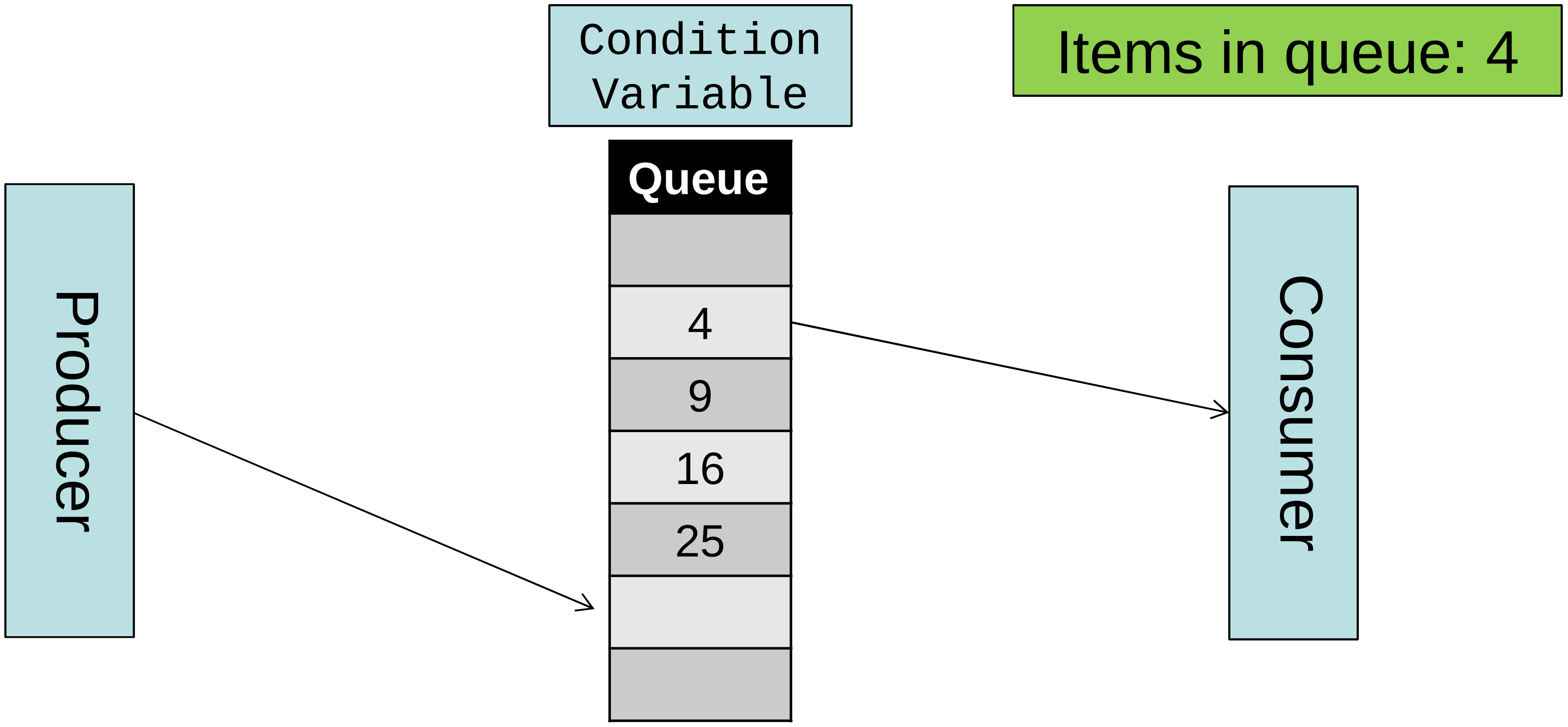
- Producer produces an item
- Producer calls notify() on the condition variable
  - Often called ‘signal’ rather than notify

Reminder: Producer-Consumer



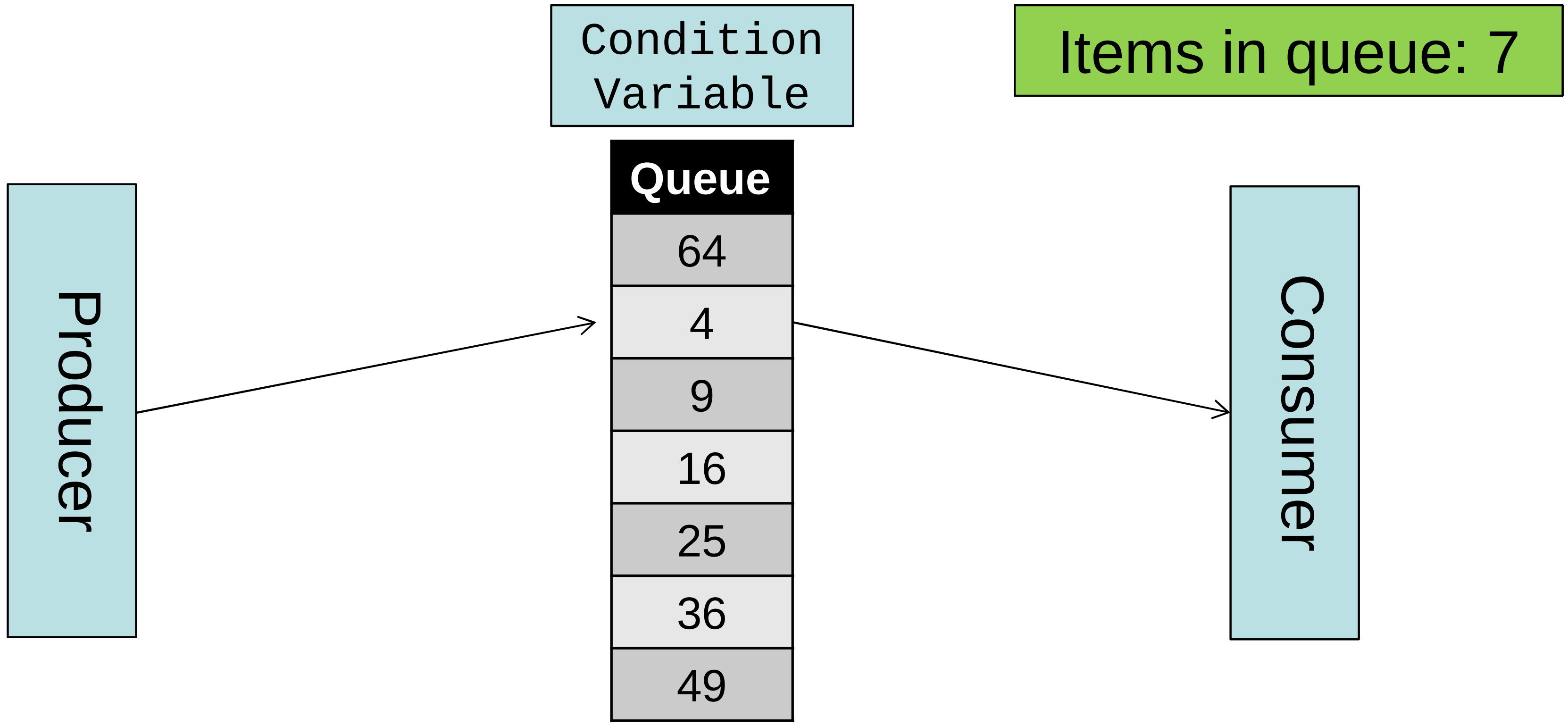
- Consumer wakes up and wants the lock so that it can continue
- When Producer leaves the synchronized function, consumer can enter its own, and consume the item

Reminder: Producer-Consumer



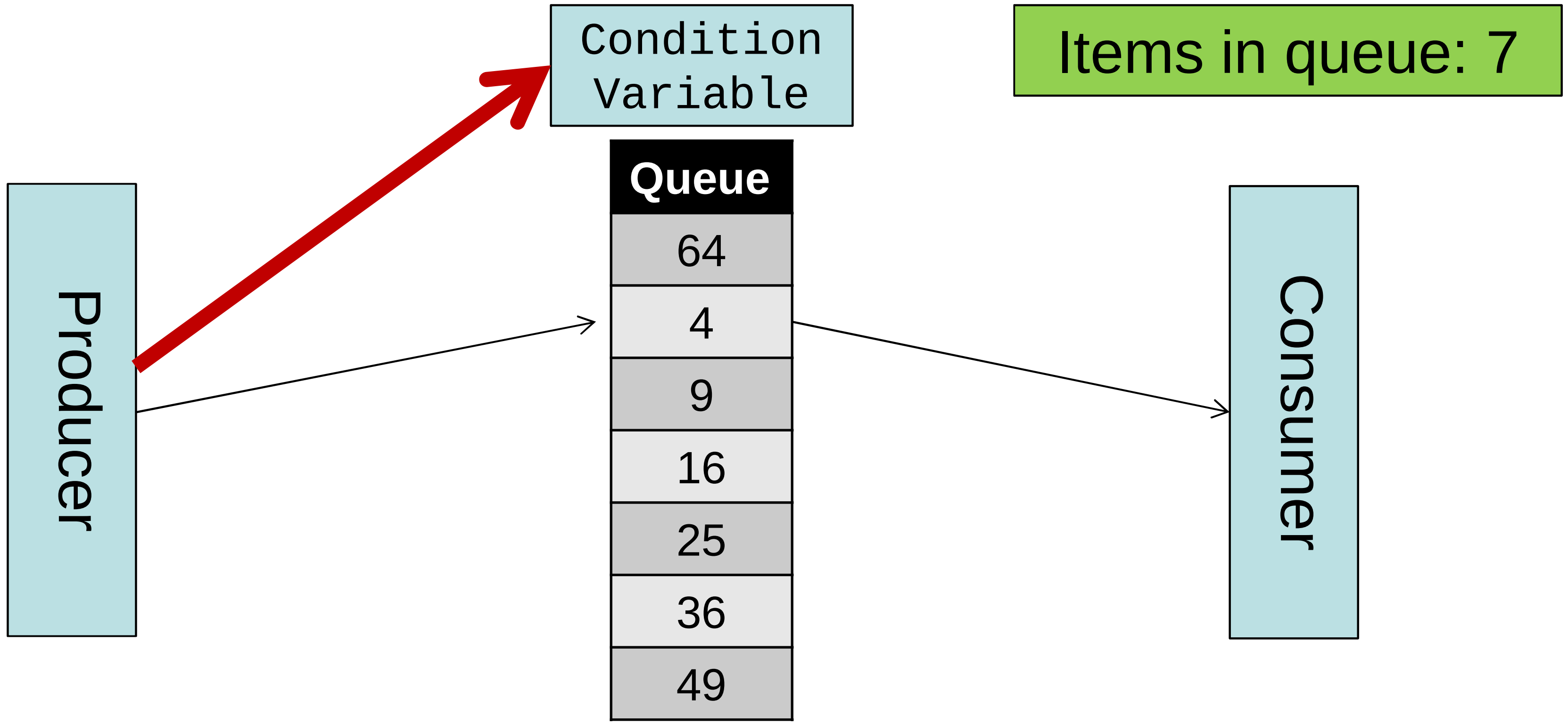
- Producer keeps producing…

Reminder: Producer-Consumer



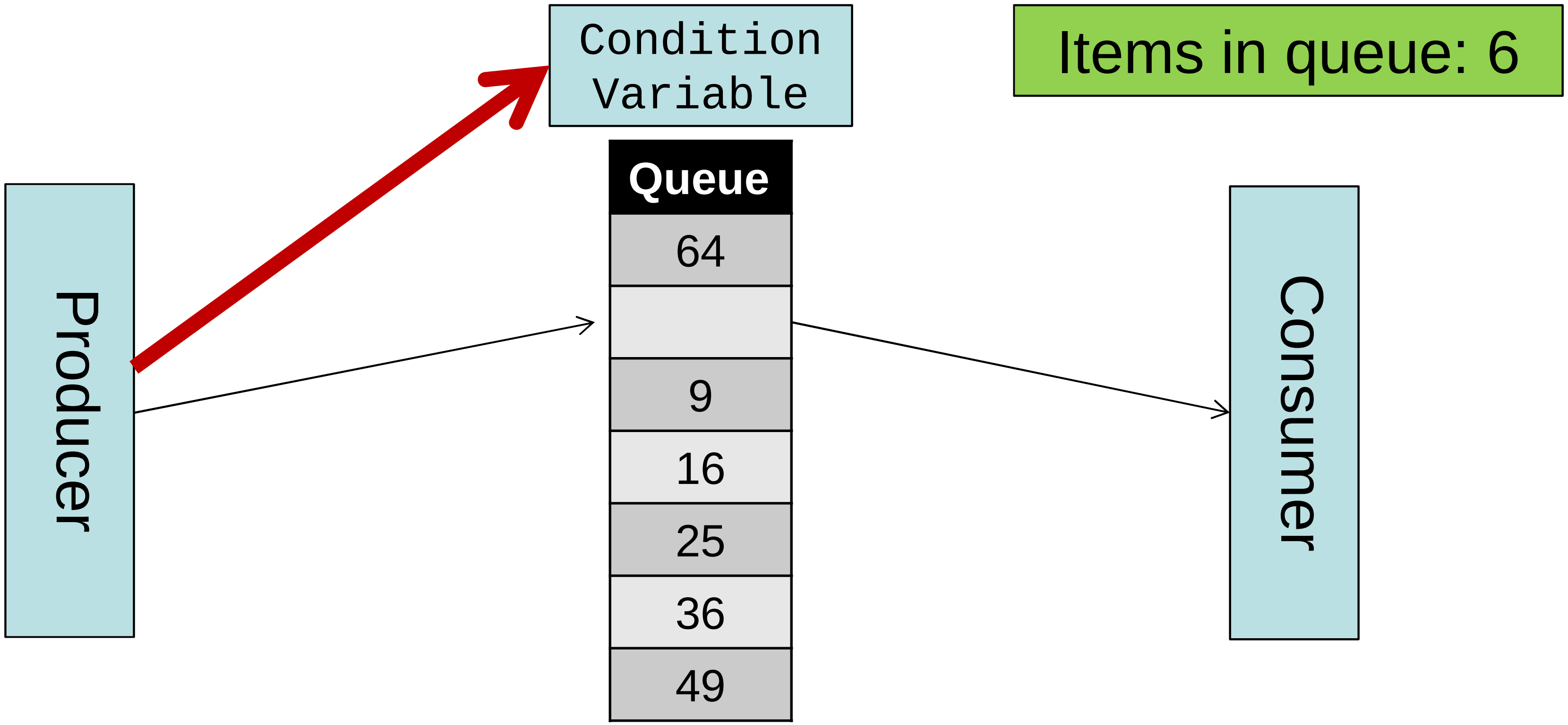
- Producer keeps producing…
- At a later point, the Producer finds that the queue is full…

Reminder: Producer-Consumer



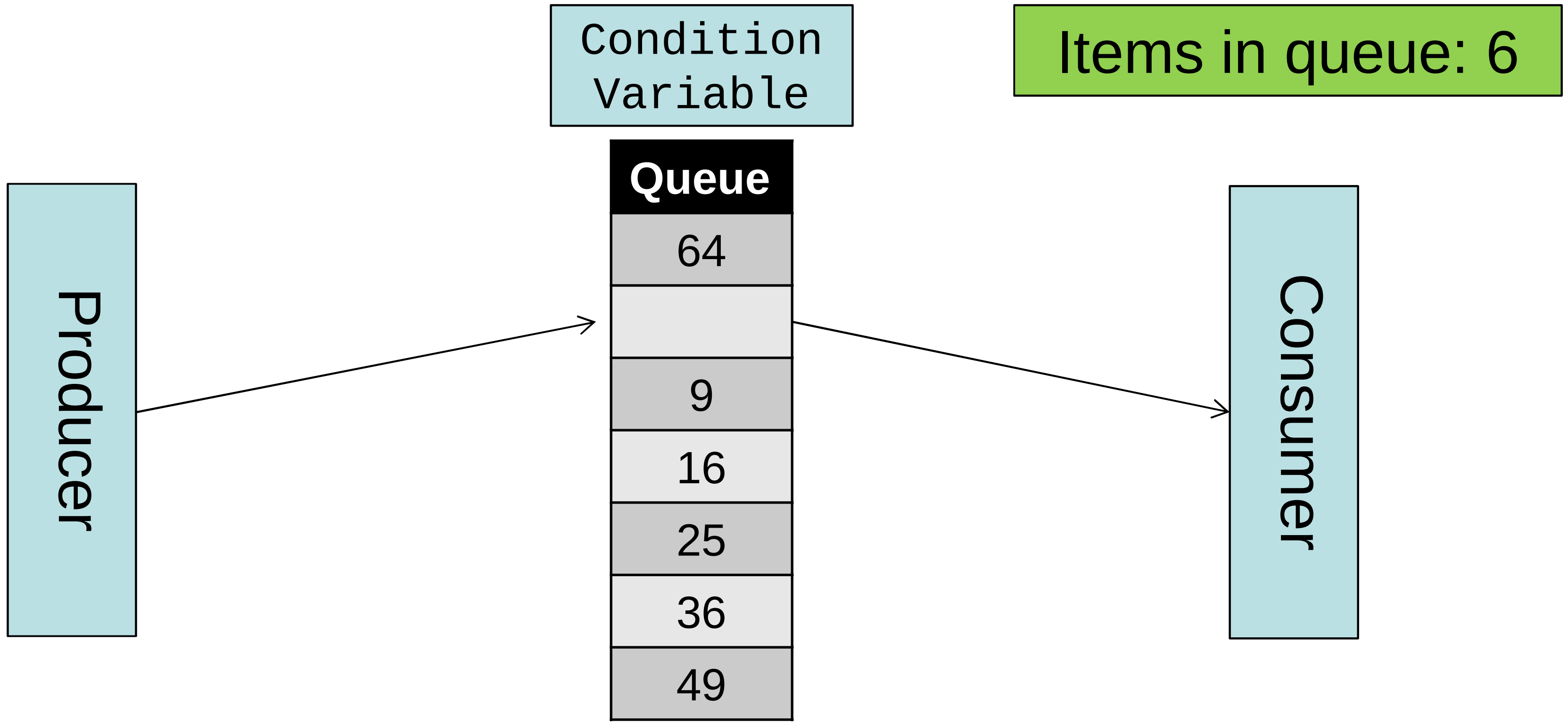
- Producer keeps producing
- At a later point, the Producer finds that the queue is full
- Producer calls Wait() on the condition variable

Reminder: 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

Reminder: 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

Monitor procedures

- *Monitor procedures* manipulate the values of the private monitor variables:
  - Conceptually: only the names are visible outside the monitor, not their implementation
  - A thread can only read or change the value of a private monitor variable by calling a monitor procedure
    - The private monitor variables are shared by all of the monitor procedures
    - Monitor procedures may also have their own local variables—each procedure call gets its own copy of these
  - Statements within monitor procedures (or initialisation code) may not access variables **declared outside** the monitor (unless passed as arguments to a monitor procedure)

Mutual exclusion

- At most *one* instance of *one* monitor procedure may be active in a monitor at a time:
  - If one thread already executing a monitor procedure, any further threads which call a procedure of the same monitor will **block** and be placed on the ***entry queue*** for the monitor
  - When thread in the monitor completes the monitor procedure call, one of the blocked threads on the entry queue is awakened & given mutual exclusion
  - Entry queues **usually** defined to be FIFO, so first thread to block will be the next one to enter the monitor
- This solves the critical section problem, since:
  - all shared state is held in private monitor variables
  - all communication between threads is via monitor procedures
  - so, any access to shared state is guaranteed mutually exclusive
- Different classes of critical section can be implemented by using a different monitor for each class

Condition variables

- Synchronisation within monitors is achieved using monitor procedures and condition variables:
  - Mutual exclusion is **implicit** with monitor procedures
  - Condition synchronisation (ensuring that any conditions are met, e.g. items exist to consume) must be programmed **explicitly** using *condition variables*
    - i.e. you need to write to code yourself to do this

- *Condition variables* are used to **delay** a thread that can't safely execute a monitor procedure until the monitor's state satisfies some boolean condition:
  - Condition variables are not visible outside the monitor
    - So only monitor procedures can check them
  - The only access to them is via *special monitor operations* within monitor procedures (e.g. wait() and notify()/signal())

Condition synchronisation

- The **value** of a condition variable is a ***delay queue*** of blocked processes waiting on a condition
  - If a call to a monitor procedure can't proceed until a condition is satisfied, the process *waits* on **the corresponding** condition variable
  - When another process executes a monitor procedure that makes the condition true, it *signals* to the process(es) waiting on **the corresponding** condition variable
- We assume that the following operations are defined for a condition variable v:
  - **`wait`**`(v)`: wait at the end of the delay queue for v
  - **`signal`**`(v)`: wake the process at the front of the delay queue for v and continue
  - **`signal_all`**`(v)`: wake all the processes on the delay queue for v and continue
  - **`empty`**`(v)`: true if the delay queue for v is empty

The `signal` and `wait` operations

- The process must have exclusive access to the monitor in order to wait
  - Must be in a monitor procedure
- If a process detects that it can't proceed, it blocks on a condition variable v by executing:
  - **`wait`**`(v);`
- The blocked process **relinquishes exclusive access to the monitor** and is appended to the end of the delay queue for v
- Processes blocked on a condition variable v are woken up when some *other* process performs a **`signal`** operation on the variable:
  - **`signal`**`(v);`
- The first process is awoken (assuming FIFO)
- If the delay queue for v is empty, **`signal`** does nothing
  - This is *unlike* semaphores, where if no process was waiting on the semaphore, a ***V*** operation for a semaphore increments the semaphore
- Note: In Java the default signal function is called notify()

Wait implementation

- Get a lock, on entering monitor procedure
- Test condition to continue
  - E.g. items available to consume
- Loop while condition is not true:
  - Unlock the lock you have locked
  - Wait on queue until signalled
  - Get a lock again (before re-testing condition)
- When the loop condition succeeds, you have the lock and can continue
  - E.g. to consume an item

Potential problem – must be atomic

- Get a lock, on entering monitor procedure
- Test condition to continue
  - E.g. items available to consume
- Loop while condition is not true:
  - Unlock the lock you have locked
  - Wait on queue until signalled
  - Get a lock again (before re-testing condition)
- When the loop condition succeeds, you have the lock and can continue
  - E.g. to consume an item

Lost notify problem

- Unlock the lock you have locked
- Wait on queue until signalled

- If these two operations do **not** operate in an atomic environment, you can lose the notification and never get awoken
  - You unlock
  - Notification is sent
  - You wait on queue for the notification
  - **Too late! It was sent before then**

Signalling disciplines

- When a monitor procedure calls **`signal`** on a condition variable, it wakes up the **first** blocked process in the delay queue waiting on the condition
  - *Signal and Wait:* the signaller waits until some later time and the signalled process executes now
  - *Signal and Continue:* the signaller continues and the signalled process executes at some later time

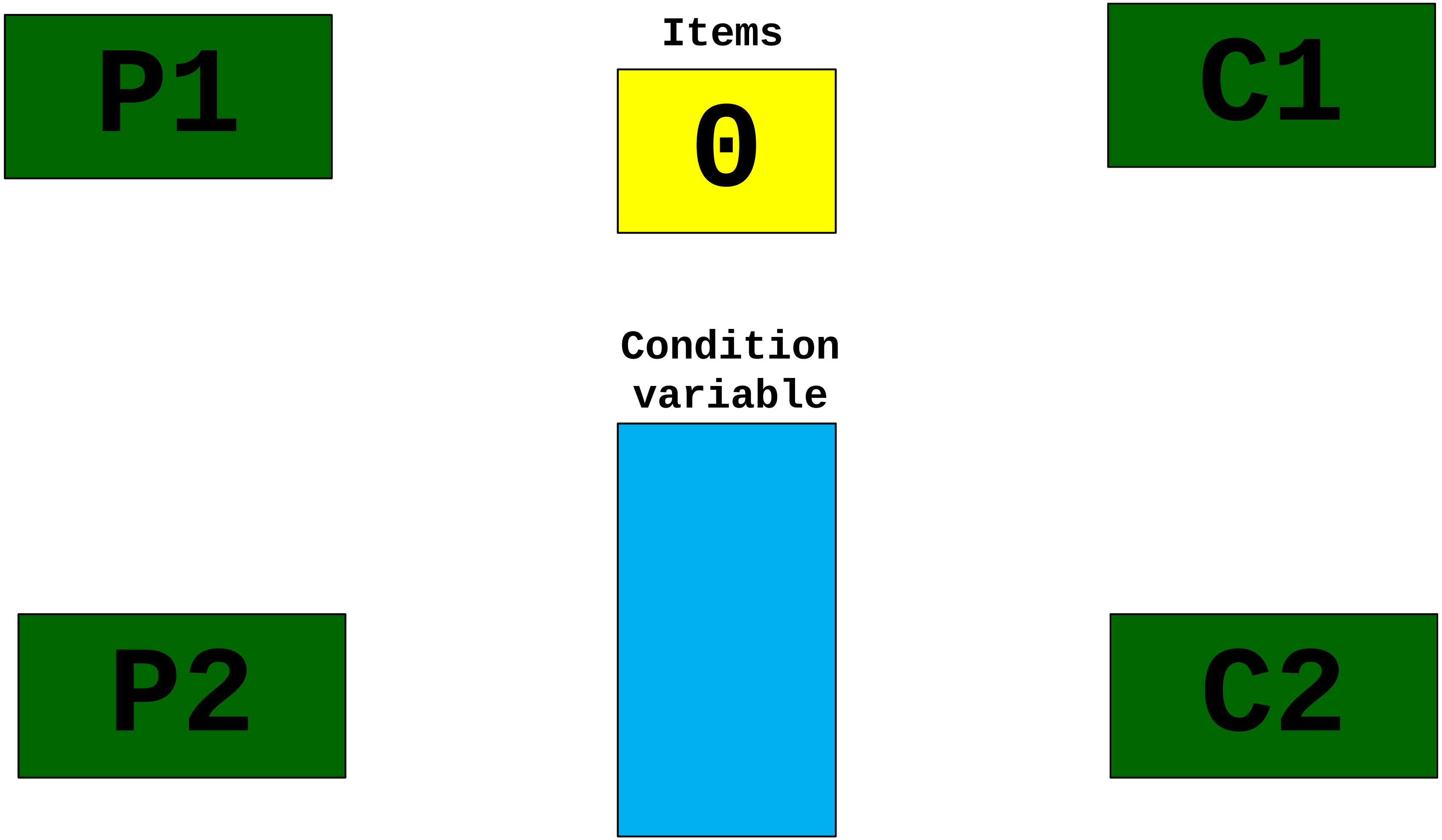
Monitors & Java

- (Cut-down) monitors form the basis of Java's support for (shared memory) concurrency:
  - *Mutual exclusion* can be implemented in Java using the **`synchronized`** keyword
  - A synchronized method (or block) is executed under mutual exclusion with all other synchronized methods on the same object
  - Java provides basic operations for *condition synchronisation*: `wait()`, `notify()`, `notifyAll()`
  - Each Java object has a single (implicit) condition variable and delay queue, the *wait set*
  - Java uses the *signal and continue* signalling discipline

Shared Condition Variable Issues

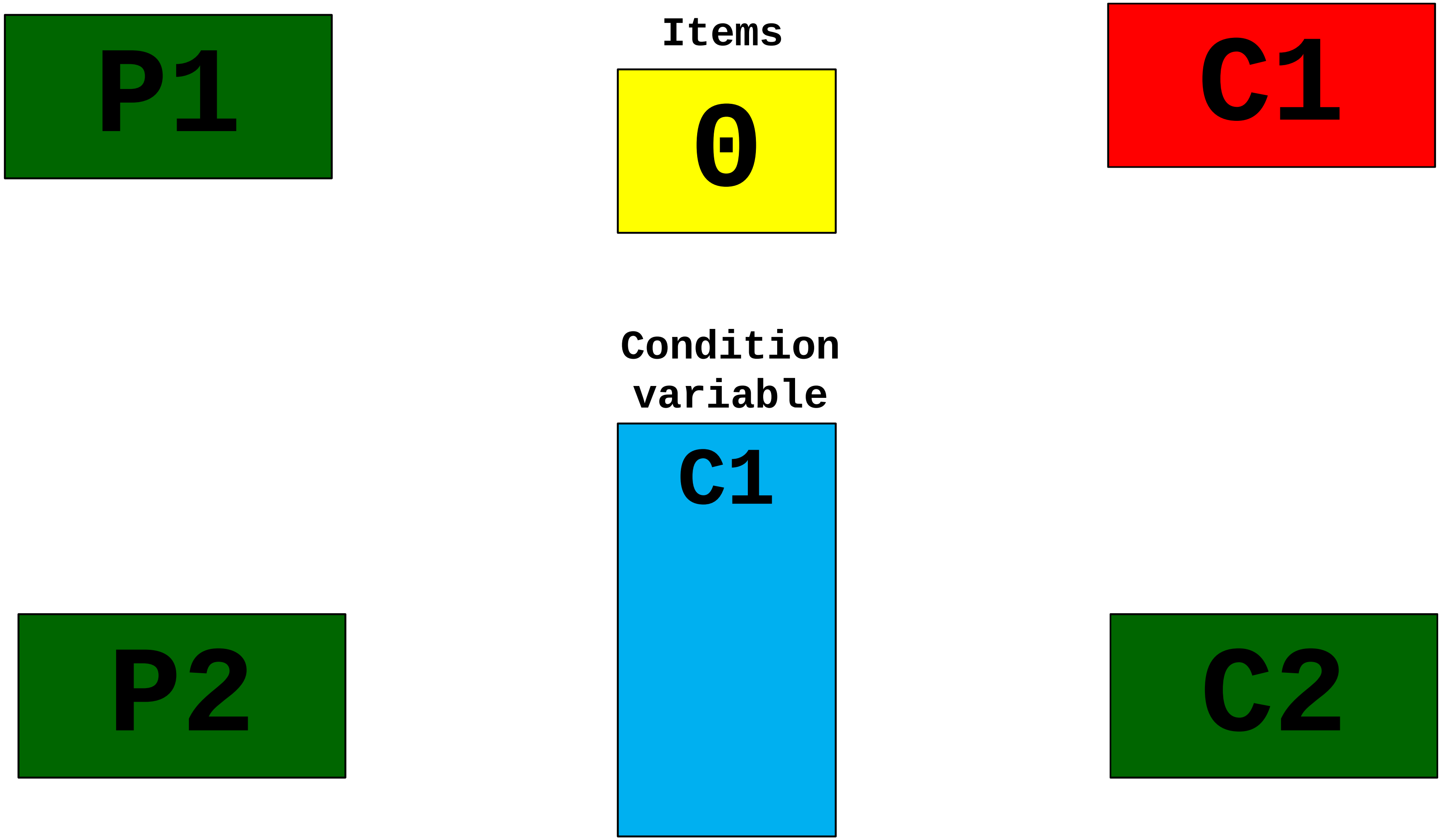
- **Another loss of notification problem can occur when a single condition variable is used for multiple purposes**
- **E.g. consider a single entry buffer for the producer-consumer problem (only one space)**
- Assume two producers and two consumers
- Consumer 1 attempts to remove an item and blocks
- Consumer 2 attempts to remove an item and blocks
- Producer 1 produces an item and fills the space
  - Notify is sent, waking a blocking thread – assume consumer 1 in this case
- Producer 2 gets in before consumer 1, attempts to produce an item and blocks
- Producer 1 attempts to produce another item and blocks
- Consumer 1 runs and consumes and item, then signals
  - Consumer 2 was first on the queue so gets the notify

Lost Notify Problem



C1 attempts to consume item

Lost Notify Problem



C2 attempts to consume item

Lost Notify Problem

P1

Items

0

C1

Condition variable

C1

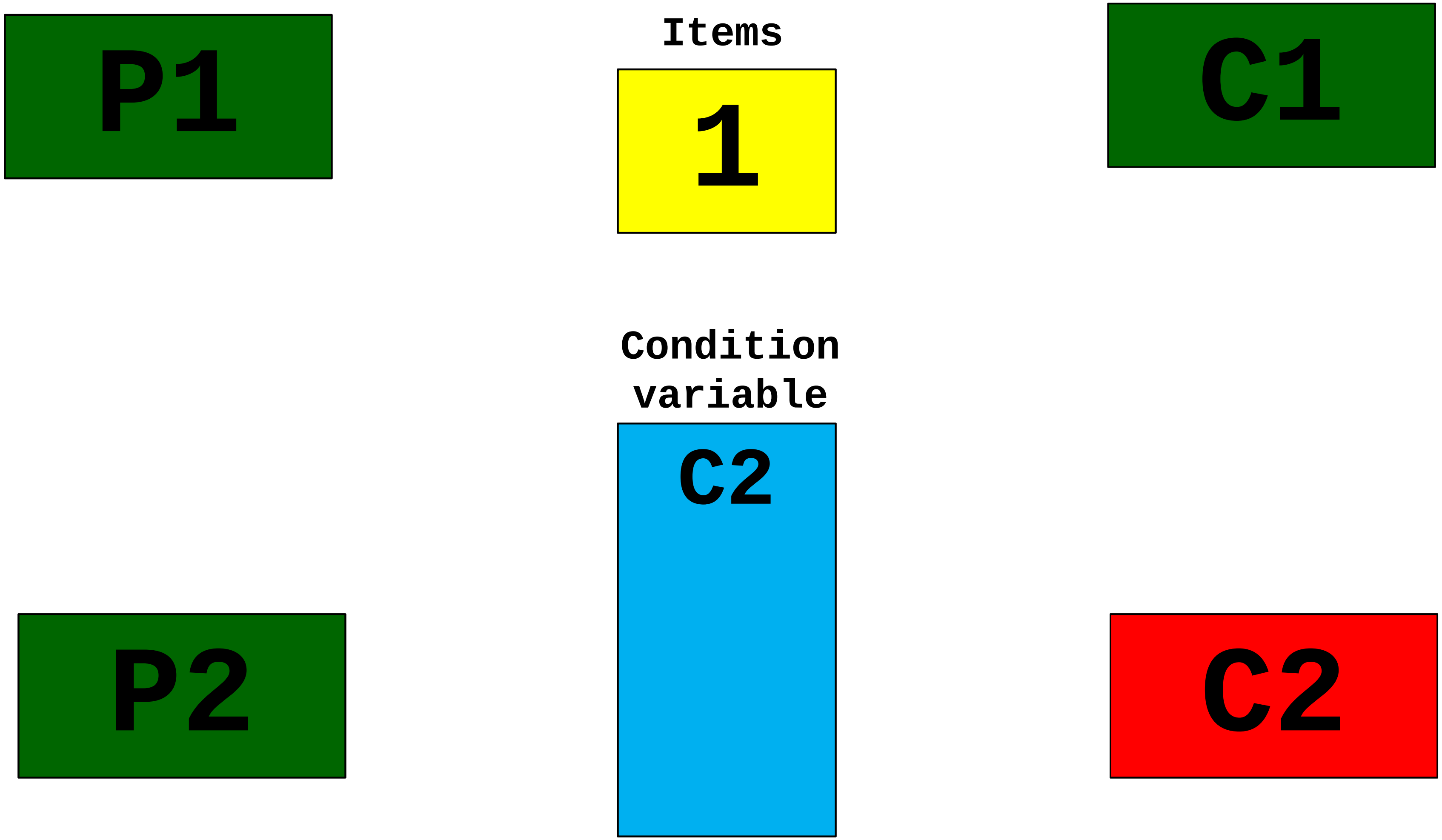
C2

P2

C2

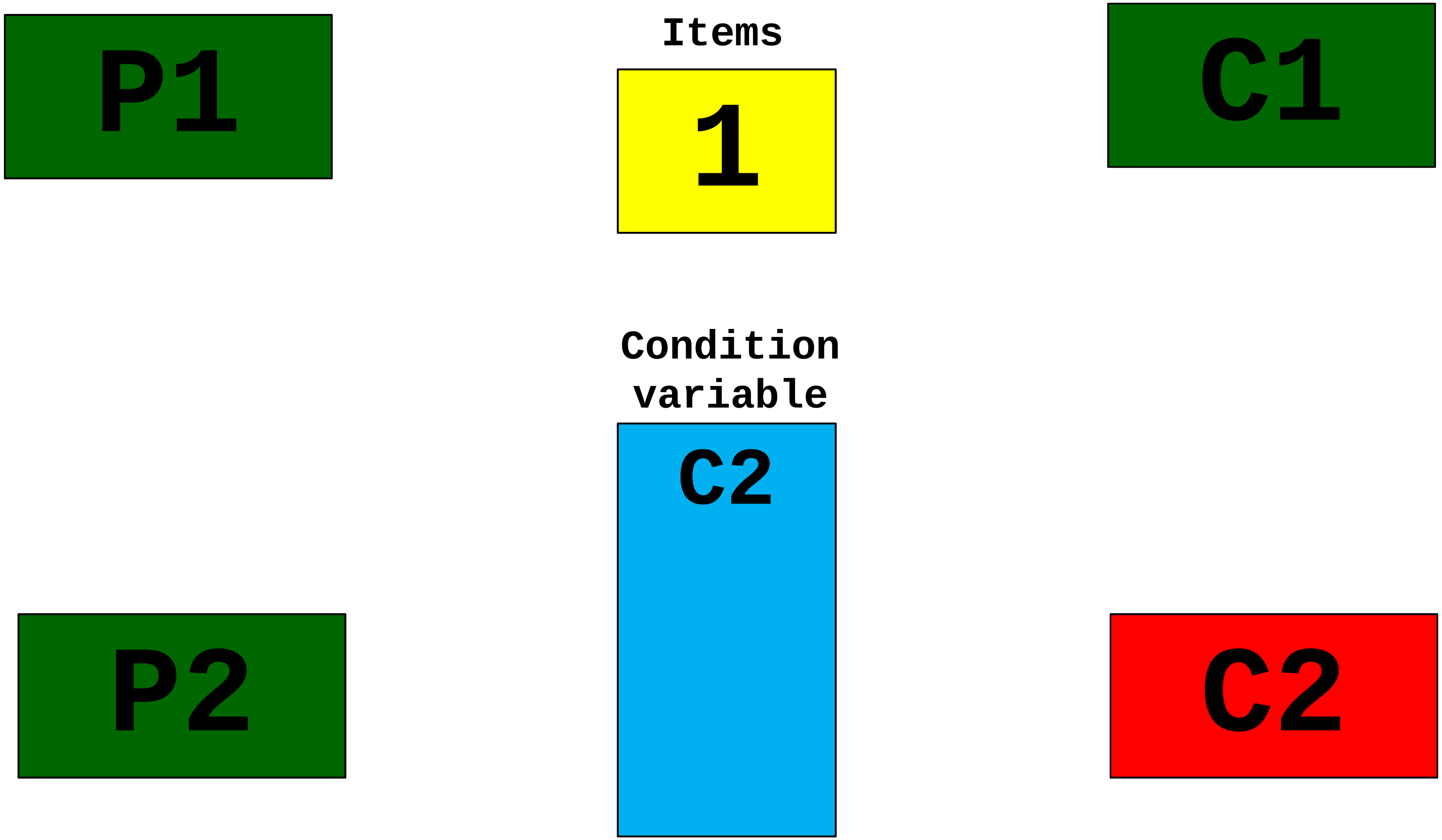
P1 produces an item

Lost Notify Problem



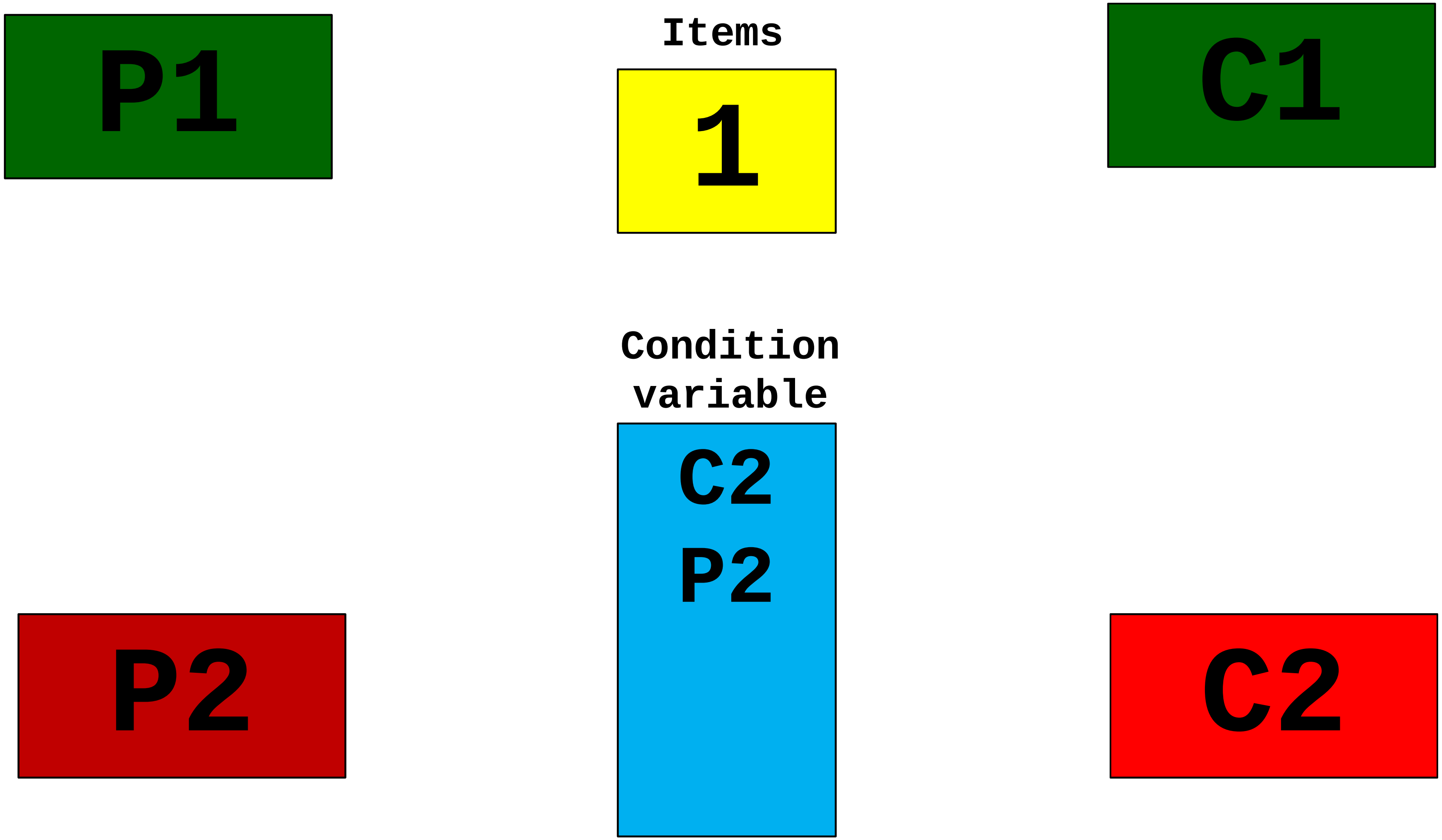
C1 has been notified

Lost Notify Problem



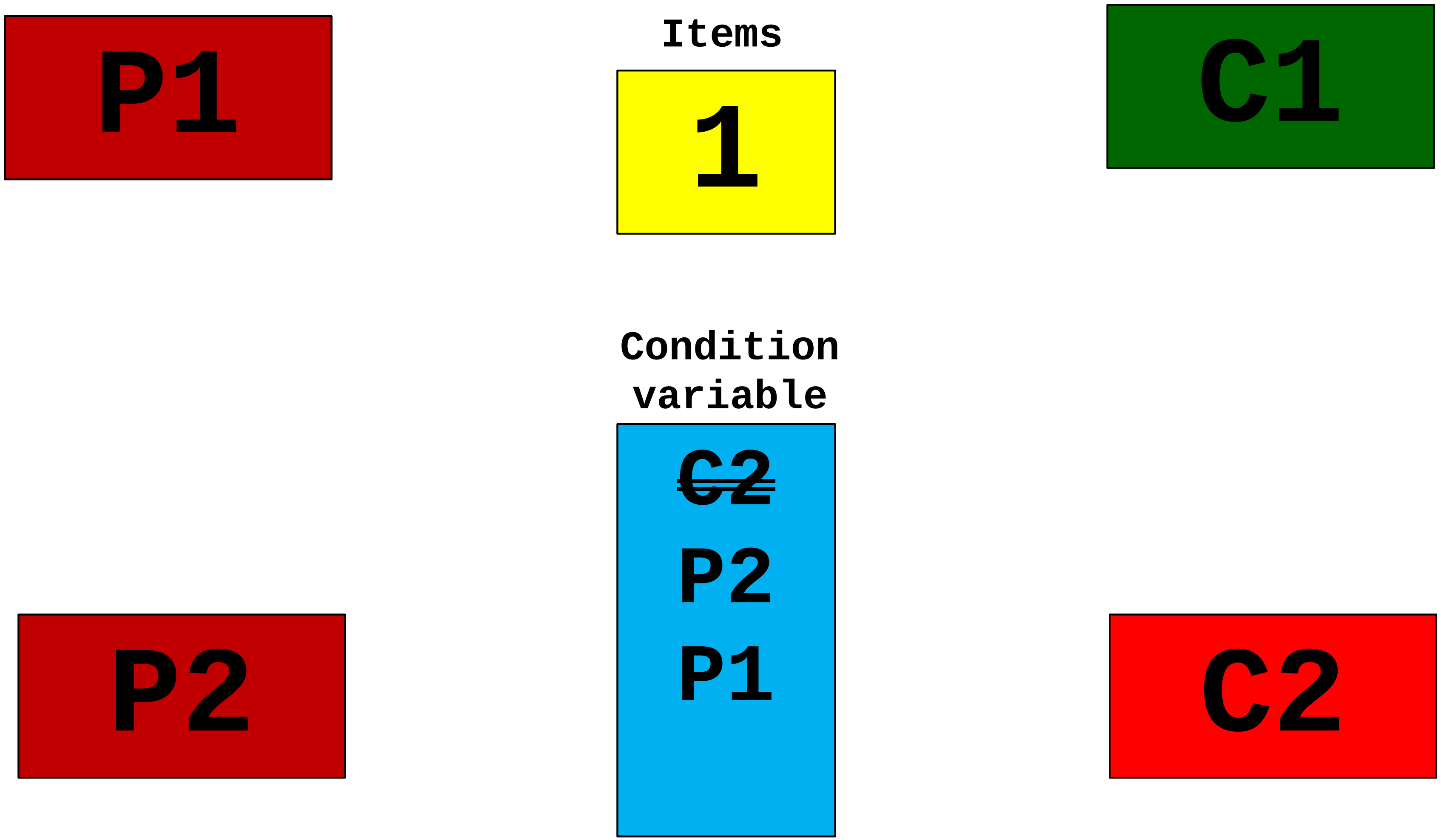
P2 attempts to produce an item

Lost Notify Problem



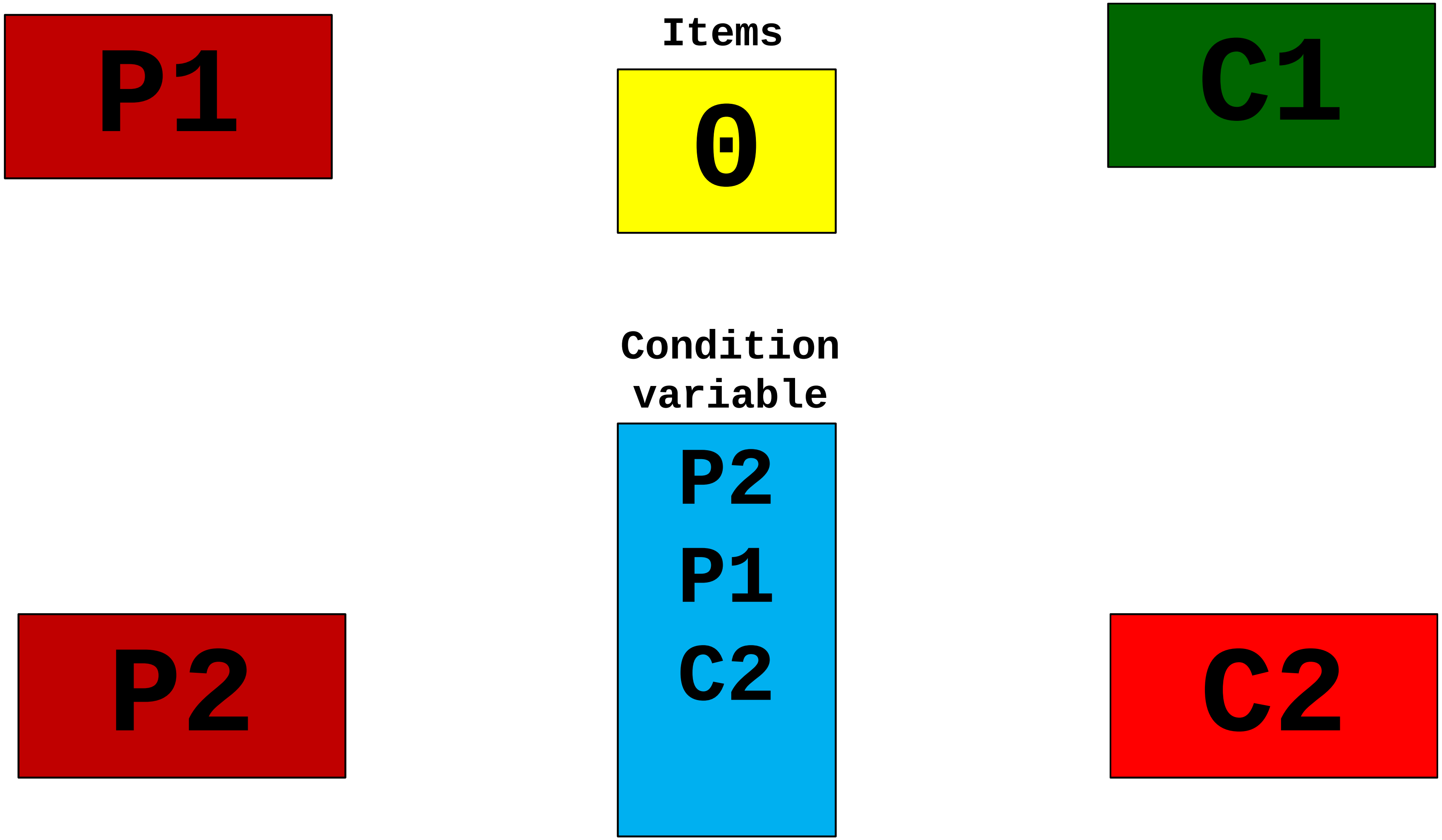
P1 attempts to produce an item

Lost Notify Problem



C1 consumes the item, and notifies … C2

Lost Notify Problem



C2 gets notified, then goes back to sleep

Lost Notify Problem

P1

Items

0

C1

Both producers are asleep, waiting to be notified that there is a space

Nothing will **ever** notify them!!!

P2

C2

C2 gets notified, then goes back to sleep

Repeat: Loss of notify problem

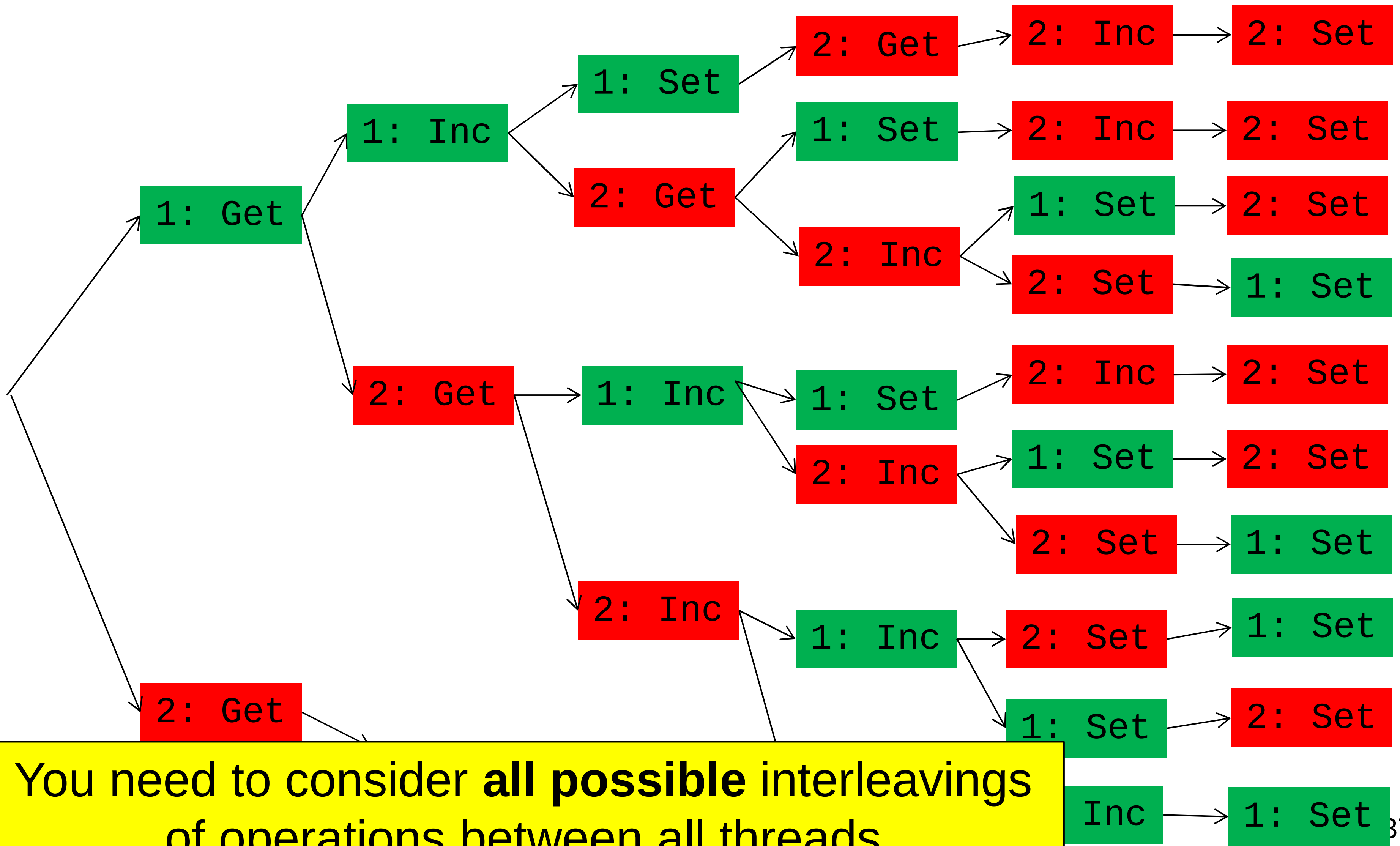
- **Another loss of notification problem can occur when a single condition variable is used for multiple purposes**

- Assumes two producers and two consumers, single entry in buffer
- Consumer 1 attempts to remove an item and blocks
- Consumer 2 attempts to remove an item and blocks
- Producer 1 produces an item and fills the space
  - Notify is sent, waking a blocking thread – assume consumer 1 in this case
- Producer 2 gets in before consumer 1, attempts to produce an item and blocks
- Producer 1 attempts to produce another item and blocks
- Consumer 1 runs and consumes and item, then signals
  - Consumer 2 was first on the queue so gets the notify

Problem and solutions

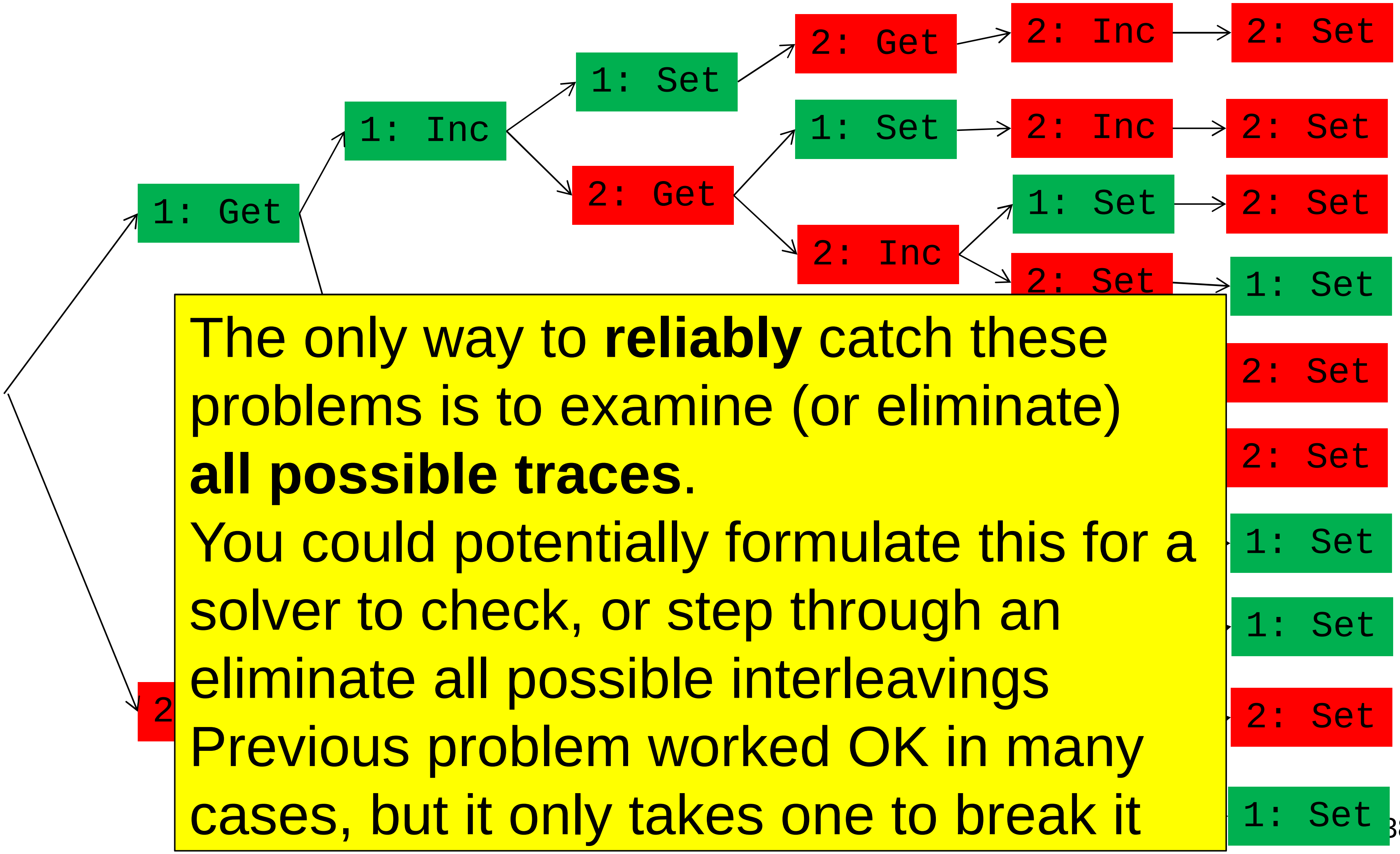
- The shared notification queue (condition variable) is a problem
  - Something which cannot use the notification (and will go straight back into a waiting state) gets the notification
- Potential solutions:
  - Separate queues (condition variables)
  - NotifyAll() – awaken all waiting threads
    - May waste time waking those who cannot proceed
    - Will lose the FIFO properties – will depend who re-waits first
  - Provide a timeout on the wait
    - Bit of a ‘hack’/workaround – wastes time waiting for timeout or awakening unnecessarily

Aside: the importance of traces



You need to consider **all possible** interleavings of operations between all threads

Aside: the importance of traces



The only way to **reliably** catch these problems is to examine (or eliminate) **all possible traces**.

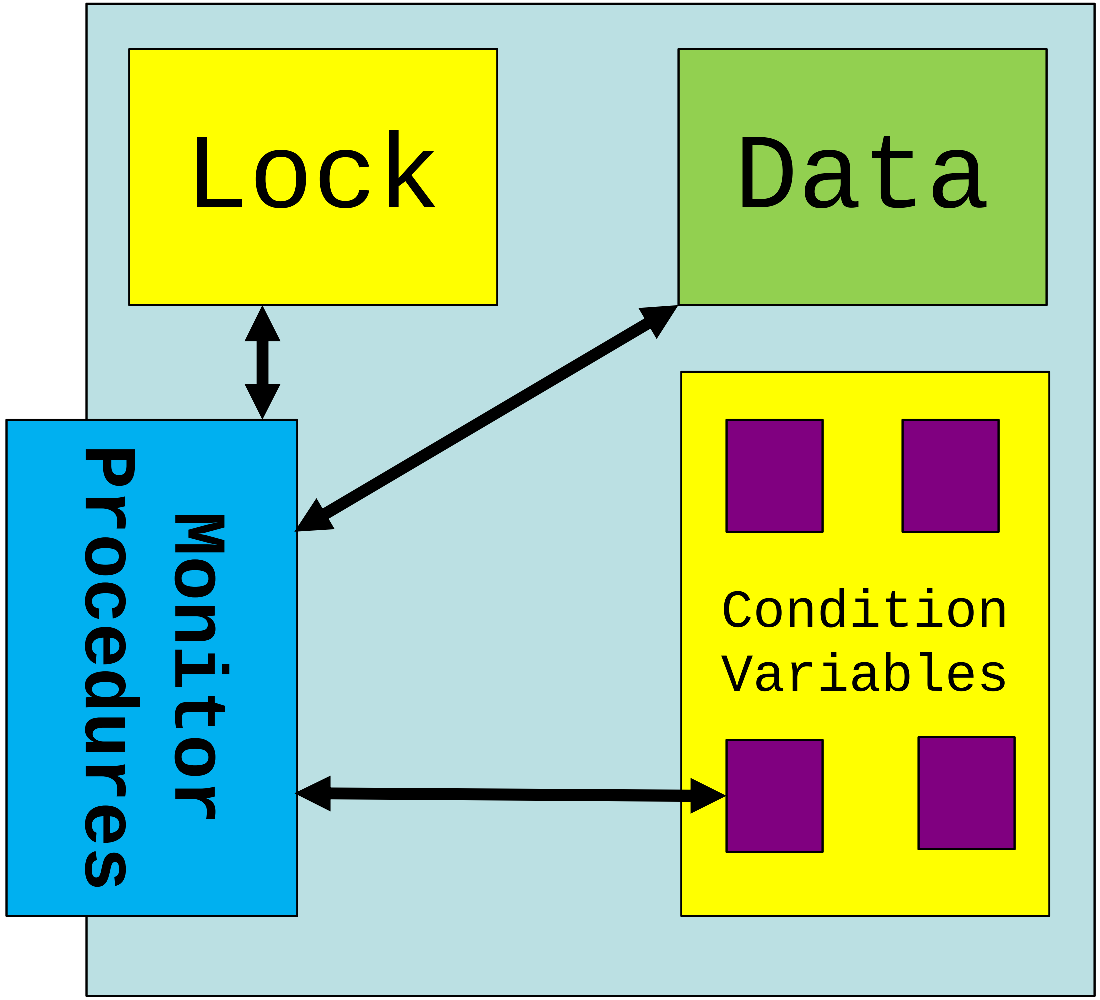
You could potentially formulate this for a solver to check, or step through an eliminate all possible interleavings

Previous problem worked OK in many cases, but it only takes one to break it

Multiple wait-queues (condition variables)

Reminder: Structure of a monitor

- Private data
- Public methods
- Locks
- Condition variables



Reminder: 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;
}
```

Synchronisation so far

- Using **`synchronized`** functions is useful
  - Safe – you KNOW that the object is synchronised while you are in the function
  - And that you release the lock at the right time
    - Automatically when you leave the function
- But there are times when you need more
  - Many other Java classes providing other synchronisation facilities
  - Browse the java.util.concurrent package
  - Will see Futures and ThreadPools tomorrow
  - But now…

ReentrantLock and Conditions

- It is useful to have multiple condition variables for a single lock
  - Need some atomic support – to wait on condition variable and unlock the lock simultaneously (atomically)
    - Otherwise risk losing notification (as we saw)
- Java **`ReentrantLock`** is useful for this:

```
Lock lock = new ReentrantLock();
Condition condSpaceAvailable = lock.newCondition();
Condition condItemAvailable = lock.newCondition();
```

- Create the lock object
- Create Condition objects ***for that lock***

Producer

```
lock.lock();
try {
    while ( iCount >= BUFFER_SIZE )
        try {  condSpaceAvailable.await(); }  // Wait for a space
        catch (InterruptedException e) { }

    // Insert item, move indices, etc

    // Tell any waiting consumers that it is worth carrying on now
    condItemAvailable.signal();
    return true;
}
finally // Ensure that this really is done, however you exit this try
{
    lock.unlock();
}
```

- try {} finally {} to ensure that the unlock will always be done, no matter what else happens

Consumer

```
lock.lock();
try {
        while ( iCount <= 0 )
                try { condItemAvailable.await(); }  // Wait for a space
                catch (InterruptedException e) { }

        // Insert item, move indices, etc

        // Tell any waiting consumers that it is worth carrying on now
        condSpaceAvailable.signal();
        return true;
}
finally // Ensure that this really is done, however you exit this try
{
        lock.unlock();
}
```

await()

- **`await()`** not **`wait()`**,
  - The lock is an object so **`wait()`** already exists for it
- **`signal()`** not **`notify()`** for the same reason
- We have seen already that the await must unlock the lock it is associated with, and then block, waiting for the condition variable to be set
  - The condition has to be associated with a lock
  - Has to happen as an atomic operation to avoid loss of notification
- We have two conditions:
  - Producer (a)waits on one and signals the other
  - Consumer does the opposite

Additional advantages of ReentrantLock

- There are functions to:
  - Retrieve a list of threads which are waiting for a lock, or waiting on a condition associated with the lock
  - Test whether the lock is currently held by current thread
  - Find out which thread currently holds the lock
  - Try to lock it, and return false rather than blocking if the lock is not attained
- Also have ability to set fair ordering policy
  - Potentially less efficient, but avoids starvation

Doing this in C?

C is not object oriented ☹

Two problems with fully implementing monitors in C:

- The fact that C is not object oriented gives us some problems hiding the data
  - Encapsulation in an object oriented language allows us to make the data private and restrict the access
  - We will have to ensure that the programmer does not manipulate the data except through our functions
- The lack of an atomic ‘wait and unlock’ causes some problems
  - Potential loss of notification
  - I added a timeout just in case (wake up and try again)

Windows Events

- Windows provides events, which we can use as (unordered) condition variables
- CreateEvent() : create an event object
  - Can have a name or by nameless
  - Can be set to manual reset or auto-clear
- CloseHandle() : release the object
- SignalEvent(): set it to signalled
- ResetEvent() : clear the signalled state
- WaitFor{Multiple|Single}Object: wait for event to be signalled, as usual

Example

- I have provided a sample monitor implementation in C for the producer-consumer problem:

```
struct MyProducerConsumerMonitor
{
     HANDLE MyEventProduce;
     HANDLE MyEventConsume;
     HANDLE MyMutex;
     volatile int iItemCount;
     volatile int iConsumerPosition;
     volatile int iProducerPosition;
     volatile int aiBuffer[BUFFER_SIZE];
};
```

- It is entirely up to you whether you look at the sample – nothing new beyond the Java one and probably more complex, but seeing the fundamentals/basic building blocks may help your understanding

Spinlock Example

```
void ProducerSpinLock(
             MyProducerConsumerMonitor* pMonitor,
             int iNewItem )
{
       LockMutex( pMonitor );
       while ( pMonitor->iItemCount >= BUFFER_SIZE )
       {
             UnlockMutex( pMonitor );
             Sleep( 1 ); // Give something else a chance
             LockMutex( pMonitor );
       }
       …
       UnlockMutex( pMonitor );
```

- If there is no room to add the new item, then unlock the mutex, sleep and then lock it again

With a wait on event

```
void ProducerSpinLock(
            MyProducerConsumerMonitor* pMonitor,
            int iNewItem )
{
      LockMutex( pMonitor );
      while ( pMonitor->iItemCount >= BUFFER_SIZE )
      {
            UnlockMutex( pMonitor );
            WaitOnEvent( pMonitor->MyEventProduce,
                        5000/* 5 second timeout*/ );
            LockMutex( pMonitor );
      }

      // Implement production

      SignalEvent( pMonitor->MyEventConsume );
      UnlockMutex( pMonitor );
}
```

Next Lecture

- Futures
- ThreadPools
- Example problem

Remaining lecture slots

Week		
This week	Monitors	Threadpools Futures Dining Philosophers
Next week	Readers and writers	Pulling it all together
Last week before Easter	Exam and revision	Extra office hour
First week back	Nothing	Past exam questions? You choose!
Second week back	Group project open day?	Past exam questions? You choose!

Aside: One person took advantage of the extra lab helpers yesterday!