Bounded Buffers

- A higher level abstraction than shared domains or simple messages
- But not quite as high level as RPC
- A buffer that allows writers to put messages in
- And readers to pull messages out
- FIFO
- Unidirectional
  - One process sends, one process receives
- With a buffer of limited size

SEND and RECEIVE With Bounded Buffers

- For SEND(), if buffer is not full, put the message into the end of the buffer and return
  - If full, block waiting for space in buffer
  - Then add message and return
- For RECEIVE(), if buffer has one or more messages, return the first one put in
  - If there are no messages in buffer, block and wait until one is put in

Practicalities of Bounded Buffers

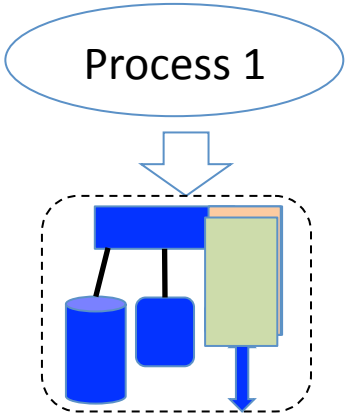
- Handles problem of not having infinite space
- Ensures that fast sender doesn't overwhelm slow receiver
- Provides well-defined, simple behavior for receiver
- But subject to some synchronization issues
  - The producer/consumer problem
  - A good abstraction for exploring those issues

The Bounded Buffer

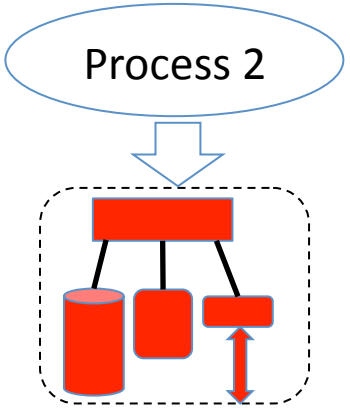
Process 1 is the writer

Process 2 is the reader

What could possibly go wrong?



A fixed size buffer



Process 1 SENDs a message through the buffer

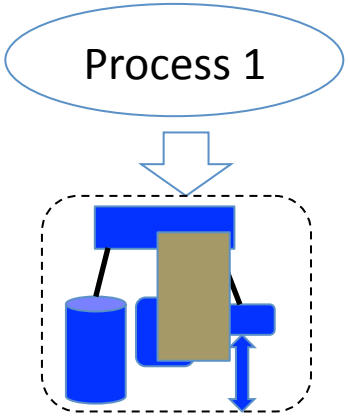
More messages are sent

And received

Process 2 RECEIVEs a message from the buffer

One Potential Issue

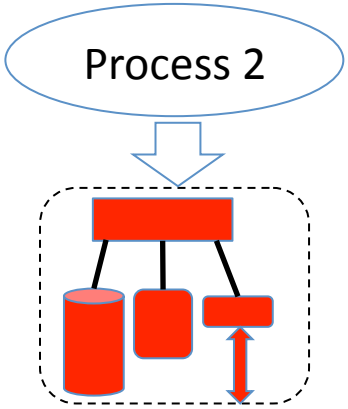
What if the buffer is full?



But the sender wants to send another message?

The sender will need to wait for the receiver to catch up

An issue of *sequence coordination*



Another sequence coordination problem if receiver tries to read from an empty buffer

Handling Sequence Coordination Issues

- One party needs to wait
  - For the other to do something
- If the buffer is full, process 1's SEND must wait for process 2 to do a RECEIVE
- If the buffer is empty, process 2's RECEIVE must wait for process 1 to SEND
- Naively, done through *busy loops*
  - Check condition, loop back if it's not true
  - Also called *spin loops*

Implementing the Loops

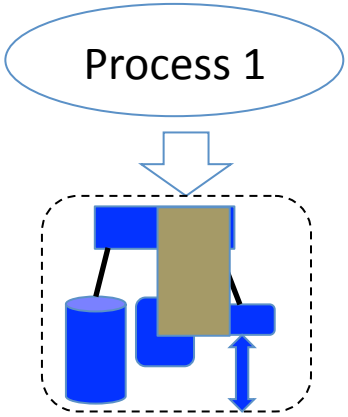
- What exactly are the processes looping on?
- They care about how many messages are in the bounded buffer
- That count is probably kept in a variable
  - Incremented on SEND
  - Decremented on RECEIVE
  - Never to go below zero or exceed buffer size
- The actual system code would test the variable

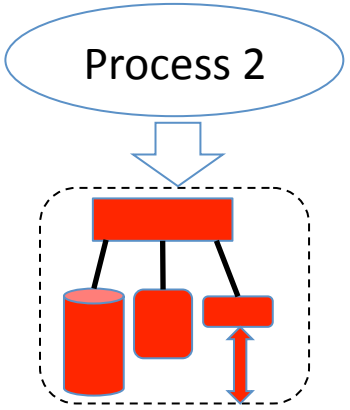
A Potential Danger

Process 1 wants to SEND

Process 2 wants to RECEIVE

Concurrency's a bitch





Process 1 checks `BUFFER_COUNT`

3

Process 2 checks `BUFFER_COUNT`

5

`BUFFER_COUNT`

3

Why Didn't You Just Say `BUFFER_COUNT=BUFFER_COUNT-1`?

- These are system operations
- Occurring at a low level
- Using variables not necessarily in the processes' own address space
  - Perhaps even RAM memory locations
- The question isn't, can we do it right?
- The question is, what must we do if we <u>are</u> to do it right?

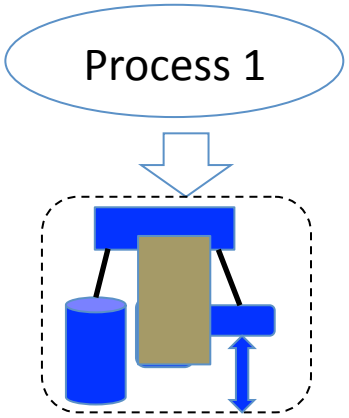
One Possible Solution

- Use separate variables to hold the number of messages put into the buffer
- And the number of messages taken out
- Only the sender updates the `IN` variable
- Only the receiver updates the `OUT` variable
- Calculate buffer fullness by subtracting `OUT` from `IN`
- Won't exhibit the previous problem
- ***When working with concurrent processes, it's safest to only allow one process to write each variable***

Multiple Writers and Races

- What if there are multiple senders and receivers sharing the buffer?
- Other kinds of concurrency issues can arise
  - Unfortunately, in non-deterministic fashion
  - Depending on timings, they might or might not occur
  - Without synchronization between threads/ processes, we have no control of the timing
  - Any action interleaving is possible

A Multiple Sender Problem



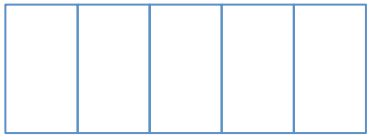
Process 1 wants to SEND

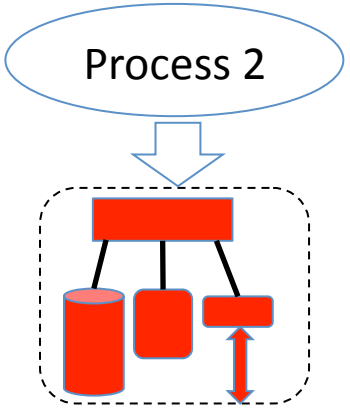
Processes 1 and 3 are senders

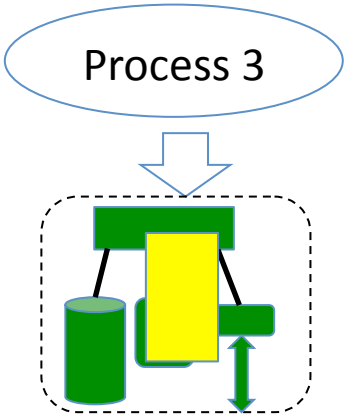
Process 2 is a receiver

There's plenty of room in the buffer for both

But . . .





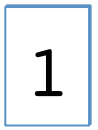


The buffer starts empty

We're in trouble:

Process 3 wants to SEND

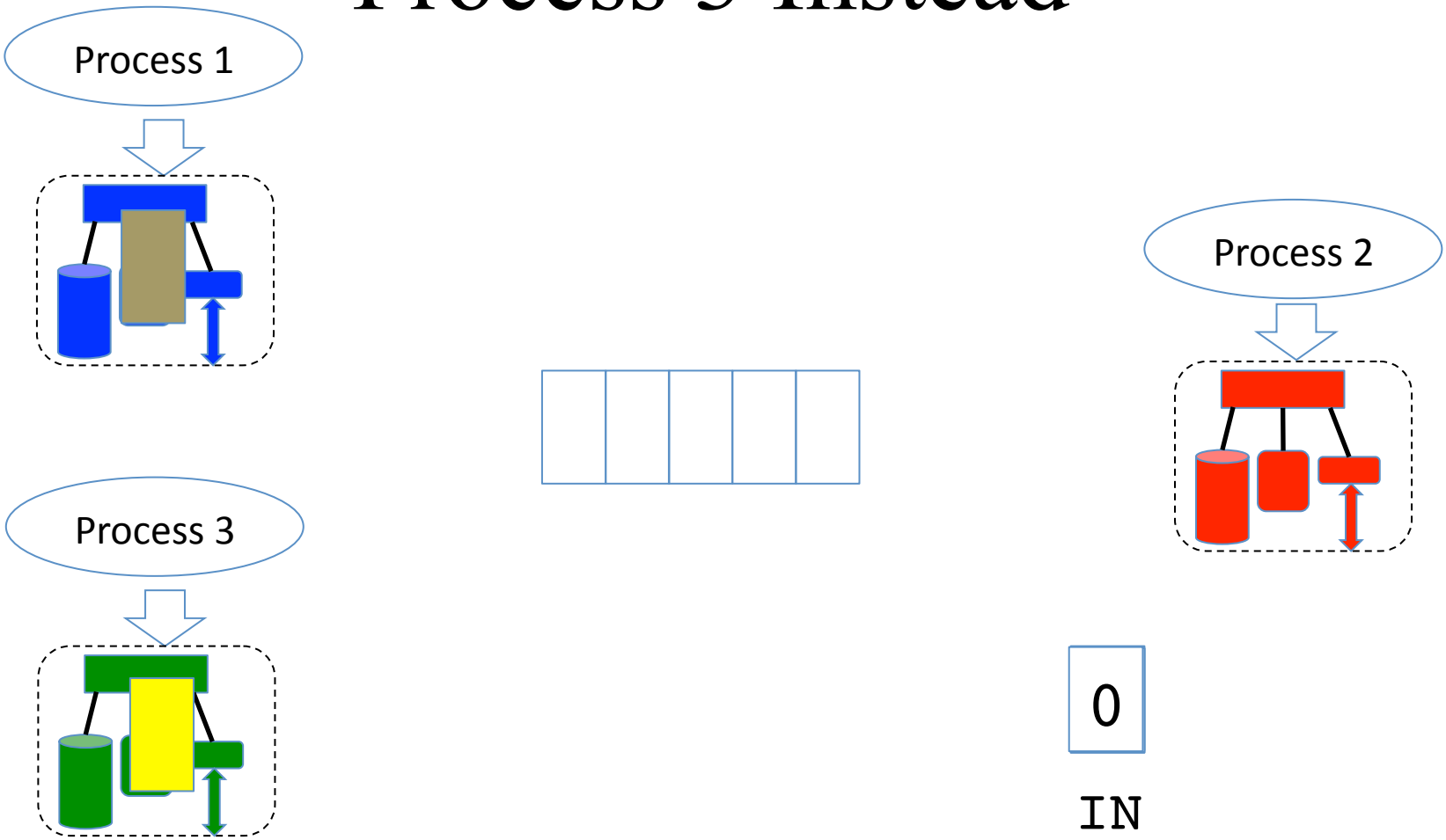
We overwrote process 1's message



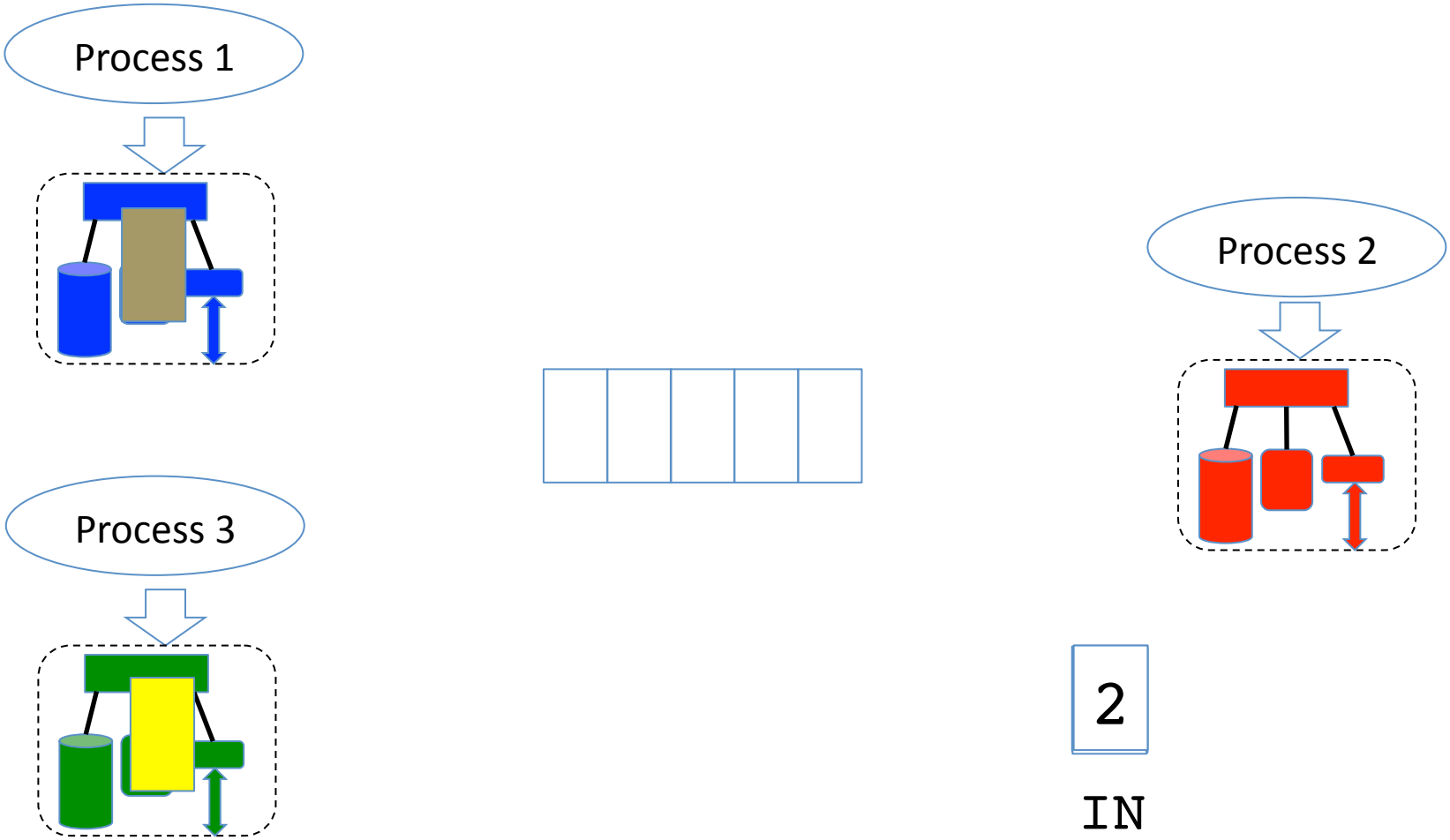
The Source of the Problem

- Concurrency again
- Processes 1 and 3 executed concurrently
- At some point they determined that buffer slot 1 was empty
  - And they each filled it
  - Not realizing the other would do so
- Worse, it's timing dependent
  - Depending on ordering of events

Process 1 Might Overwrite Process 3 Instead



Or It Might Come Out Right



Race Conditions

- Errors or problems occurring because of this kind of concurrency
- For some ordering of events, everything is fine
- For others, there are serious problems
- In true concurrent situations, either result is possible
- And it's often hard to predict which you'll get
- Hard to find and fix
  - A job for the OS, not application programmers

How Can The OS Help?

- By providing abstractions not subject to race conditions
- One can program race-free concurrent code
  - It's not easy
- So having an expert do it once is better than expecting all programmers to do it themselves
- An example of the OS hiding unpleasant complexities

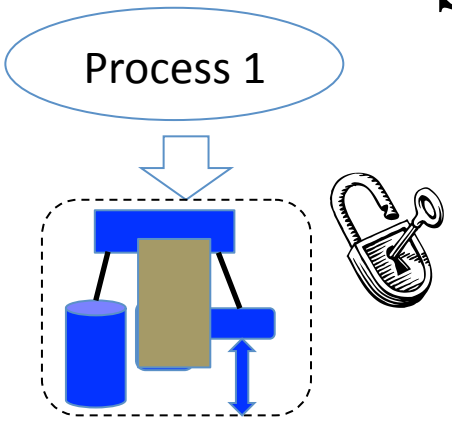
Locks

- A way to deal with concurrency issues
- Many concurrency issues arise because multiple steps aren't done atomically
  - It's possible for another process to take actions in the middle
- Locks prevent that from happening
- They convert a multi-step process into effectively a single step one

What Is a Lock?

- A shared variable that coordinates use of a shared resource
  - Such as code or other shared variables
- When a process wants to use the shared resource, it must first ACQUIRE the lock
  - Can't use the resource till ACQUIRE succeeds
- When it is done using the shared resource, it will RELEASE the lock
- ACQUIRE and RELEASE are the fundamental lock operations

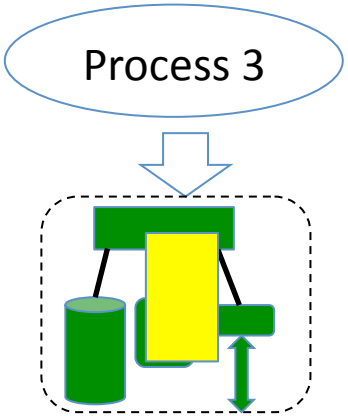
Using Locks in Our Multiple Sender Problem



To use the buffer properly, a process must:

1. Read the value of `IN`
2. If `IN < BUFFER_SIZE`, store message
3. Add 1 to `IN`

WITHOUT INTERRUPTION!



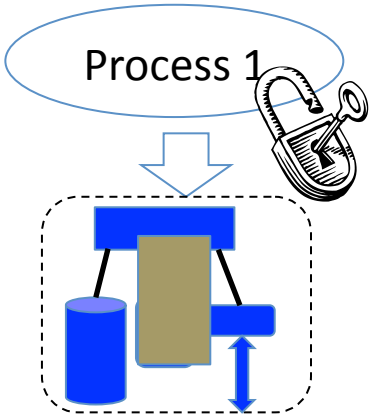
So associate a lock with those steps

`IN`

IN = 0

0 < 5 ✔

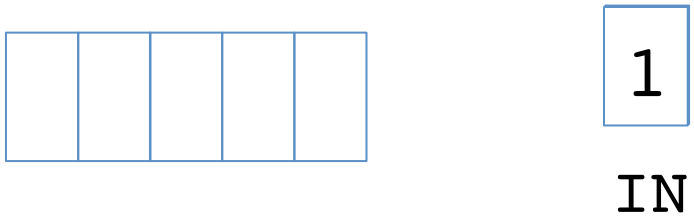
The Lock in Action

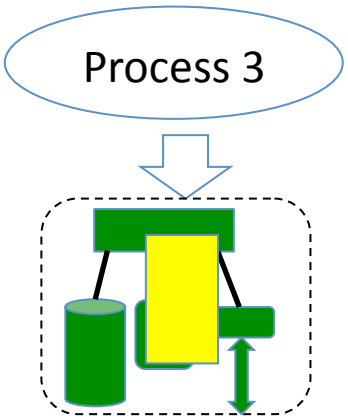


Process 1 executes ACQUIRE on the lock

Let's assume it succeeds

Now process 1 executes the code associated with the lock



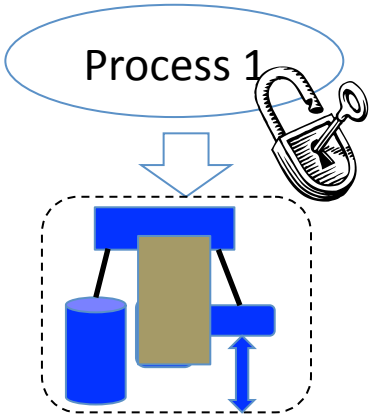


1. Read the value of `IN`
2. If `IN < BUFFER_SIZE`, store message
3. Add 1 to `IN`

Process 1 now executes RELEASE on the lock

What If Process 3 Intervenes?

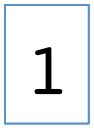
`IN = 0`



Let's say process 1 has the lock already

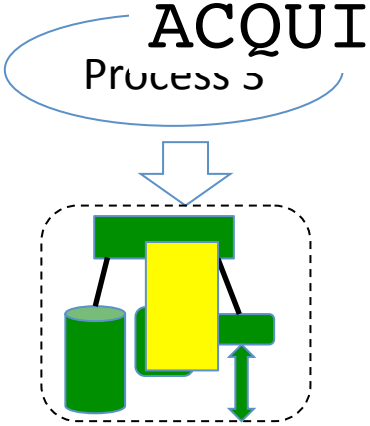
And has read `IN`

So process 1 can safely complete the `SEND`



`IN`

`ACQUIRE()`



Now, before process 1 can execute any more code, process 3 tries to `SEND`

Before process 3 can go ahead, it needs the lock

But that `ACQUIRE` fails, since process 1 already has the lock

Locking and Atomicity

- Locking is one way to provide the property of *atomicity* for compound actions
  - Actions that take more than one step
- Atomicity has two aspects:
  - Before-or-after atomicity
  - All-or-nothing atomicity
- Locking is most useful for providing before-or-after atomicity

Before-Or-After Atomicity

- As applied to a set of actions *A*
- If they have before-or-after atomicity,
- For all other actions, each such action either:
  - Happened before the entire set of *A*
  - Or happened after the entire set of *A*
- In our bounded buffer example, either the entire buffer update occurred first
- Or the entire buffer update came later
- Not partly before, partly after

Using Locks to Avoid Races

- Software designer must find all places where a race condition might occur
  - If he misses one, he may get errors there
- He must then properly use locks for all processes that could cause the race
  - If he doesn't do it right, he might get races anyway
- Since neither is trivial to get right, OS should provide abstractions to handle proper locking