Process Communications, Synchronization, and Concurrency

CS 111

Operating System Principles

Peter Reiher

Outline

- Process communications issues
- Synchronizing processes
- Concurrency issues
  - Critical section synchronization

Processes and Communications

- Many processes are self-contained
- But many others need to communicate
  - Often complex applications are built of multiple communicating processes
- Types of communications
  - Simple signaling
    - Just telling someone else that something has happened
  - Messages
  - Procedure calls or method invocation
  - Tight sharing of large amounts of data
    - E.g., shared memory, pipes

Some Common Characteristics of IPC

- Issues of proper synchronization
  - Are the sender and receiver both ready?
  - Issues of potential deadlock
- There are safety issues
  - Bad behavior from one process should not trash another process
- There are performance issues
  - Copying of large amounts of data is expensive
- There are security issues, too

Desirable Characteristics of Communications Mechanisms

- Simplicity
  - Simple definition of what they do and how to do it
  - Good to resemble existing mechanism, like a procedure call
  - Best if they're simple to implement in the OS
- Robust
  - In the face of many using processes and invocations
  - When one party misbehaves
- Flexibility
  - E.g., not limited to fixed size, nice if one-to-many possible, etc.
- Free from synchronization problems
- Good performance
- Usable across machine boundaries

Blocking Vs. Non-Blocking

- When sender uses the communications mechanism, does it block waiting for the result?
  - Synchronous communications
- Or does it go ahead without necessarily waiting?
  - Asynchronous communications
- Blocking reduces parallelism possibilities
  - And may complicate handling errors
- Not blocking can lead to more complex programming
  - Parallelism is often confusing and unpredicatable
- Particular mechanisms tend to be one or the other

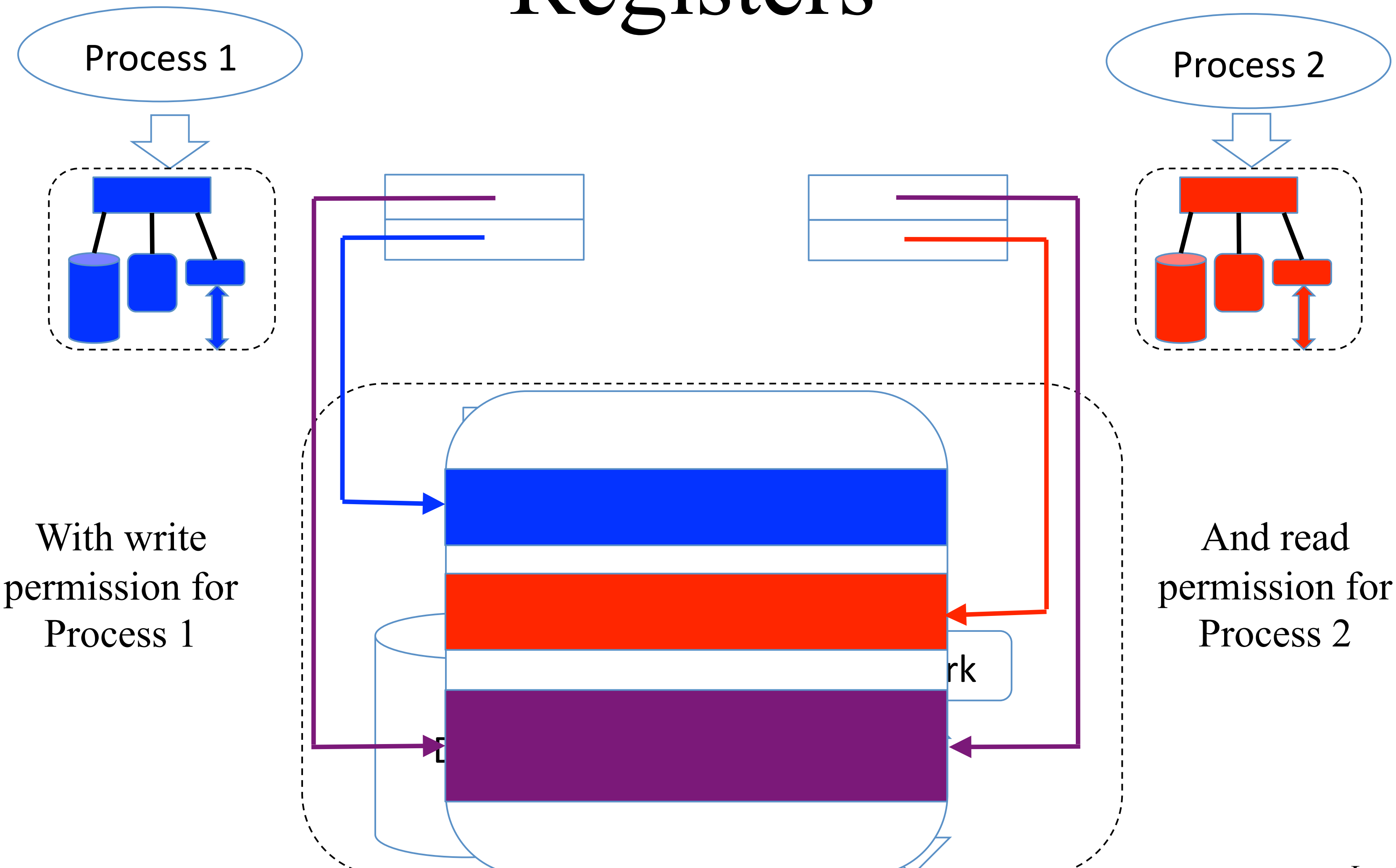
Communications Mechanisms

- Shared memory
- Messages
- RPC
- More sophisticated abstractions
  - The bounded buffer

Shared Memory

- Everyone uses the same pool of RAM anyway
- Why not have communications done simply by writing and reading parts of the RAM?
  - Sender writes to a RAM location
  - Receiver reads it
  - Give both processes access to memory via their domain registers
- Conceptually simple
- Basic idea cheap to implement
- Usually non-blocking

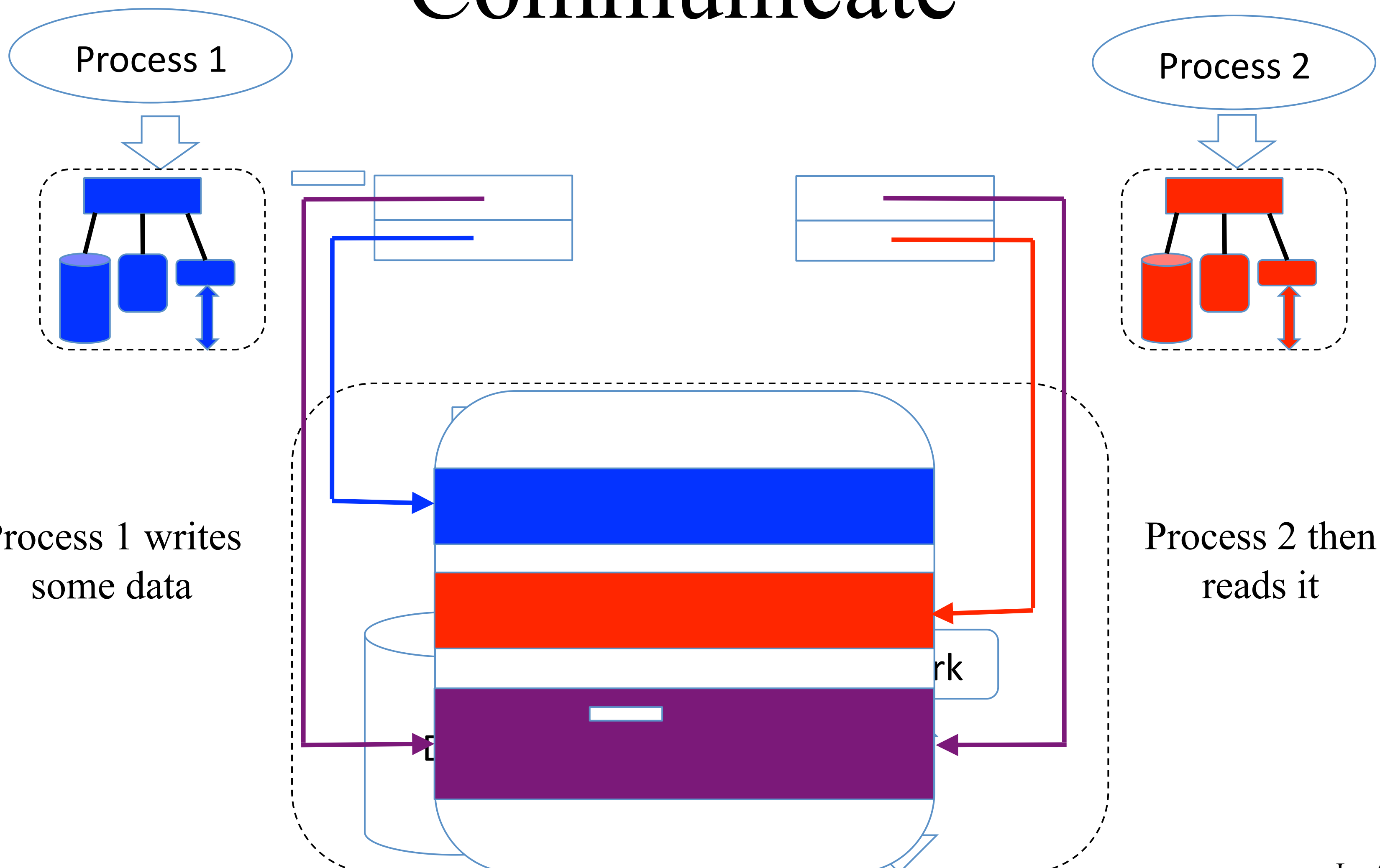
Sharing Memory With Domain Registers



With write permission for Process 1

And read permission for Process 2

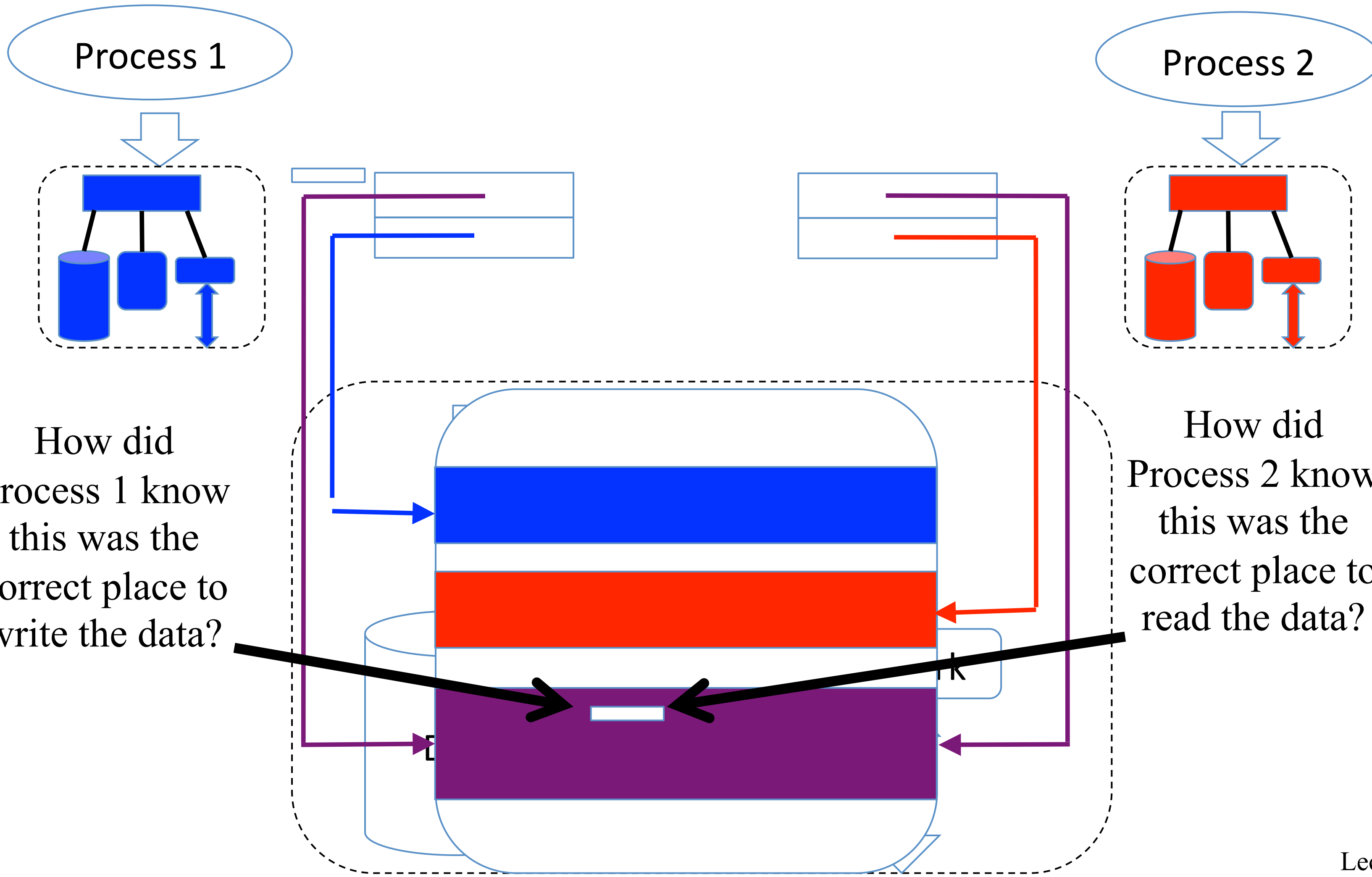
Using the Shared Domain to Communicate



Process 1 writes some data

Process 2 then reads it

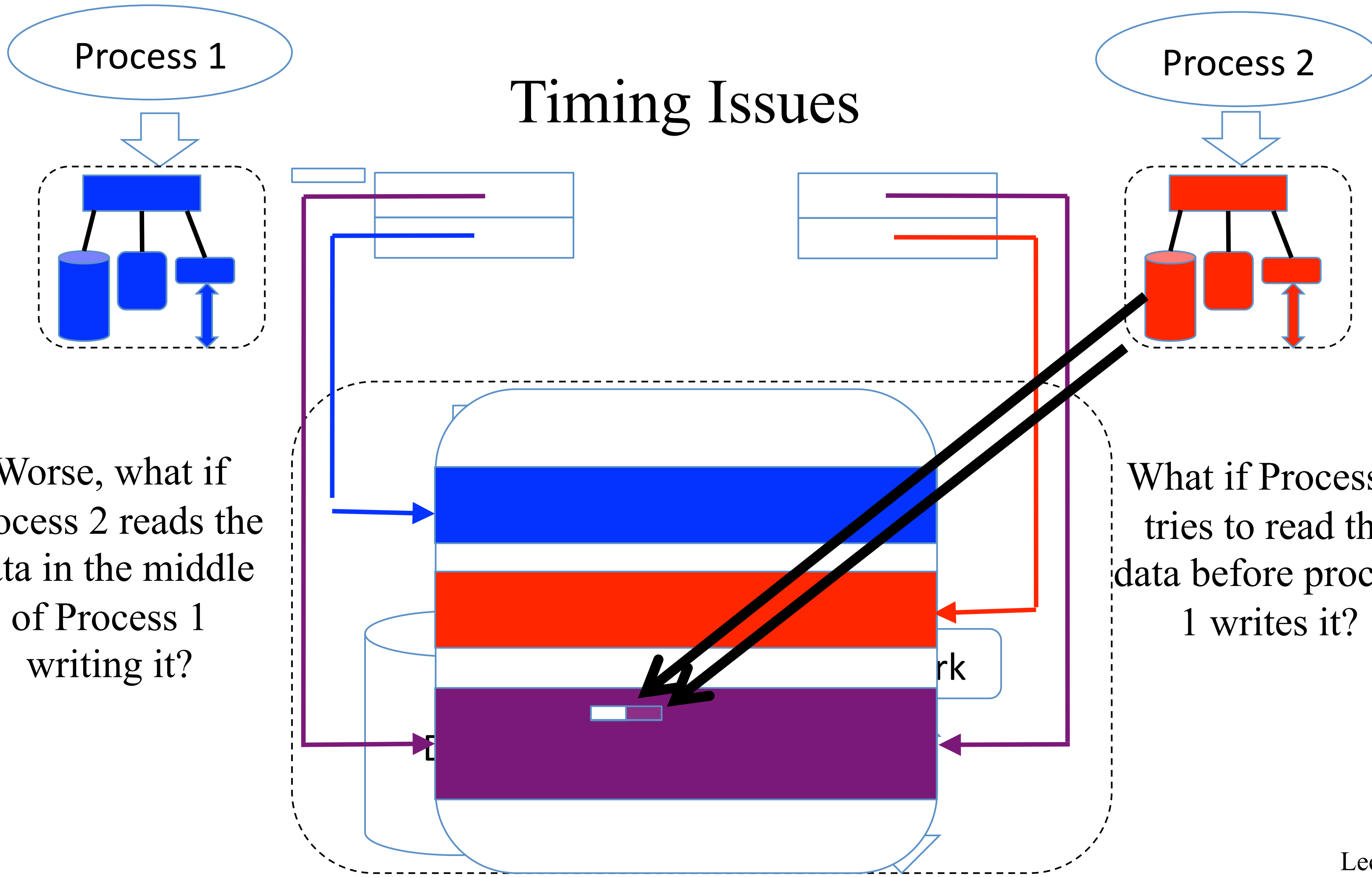
Potential Problem #1 With Shared Domain Communications



How did Process 1 know this was the correct place to write the data?

How did Process 2 know this was the correct place to read the data?

Potential Problem #2 With Shared Domain Communications



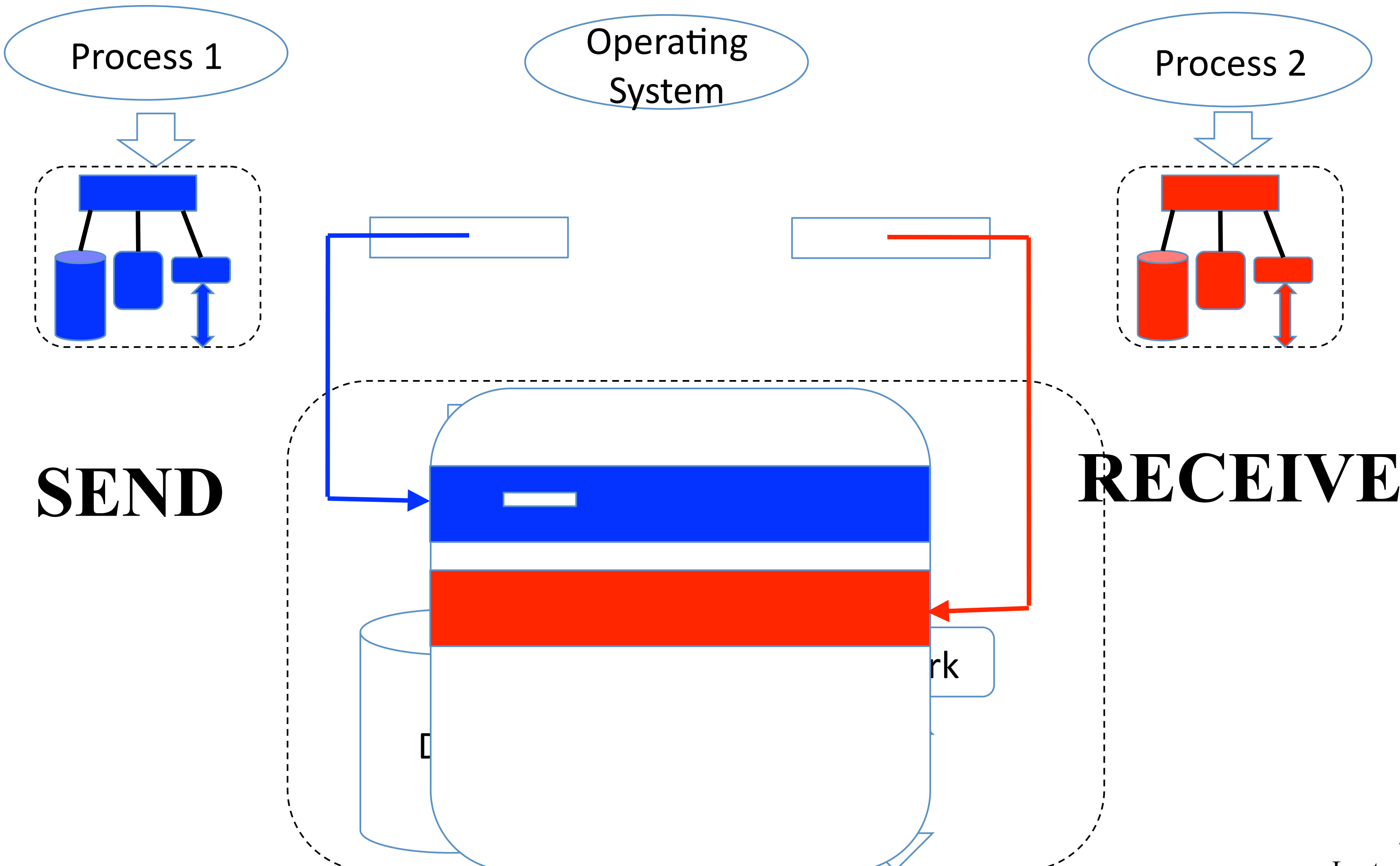
Worse, what if Process 2 reads the data in the middle of Process 1 writing it?

What if Process 2 tries to read the data before process 1 writes it?

Messages

- A conceptually simple communications mechanism
- The sender sends a message explicitly
- The receiver explicitly asks to receive it
- The message service is provided by the operating system
  - Which handles all the "little details"
- Usually non-blocking

Using Messages



Advantages of Messages

- Processes need not agree on where to look for things
  - Other than, perhaps, a named message queue
- Clear synchronization points
  - The message doesn't exist until you SEND it
  - The message can't be examined until you RECEIVE it
  - So no worries about incomplete communications
- Helpful encapsulation features
  - You RECEIVE exactly what was sent, no more, no less
- No worries about size of the communications
  - Well, no worries for the user; the OS has to worry
- Easy to see how it scales to multiple processes

Implementing Messages

- The OS is providing this communications abstraction
- There's no magic here
  - Lots of stuff needs to be done behind the scenes by OS
- Issues to solve:
  - Where do you store the message before receipt?
  - How do you deal with large quantities of messages?
  - What happens when someone asks to receive before anything is sent?
  - What happens to messages that are never received?
  - How do you handle naming issues?
  - What are the limits on message contents?

Message Storage Issues

- Messages must be stored somewhere while waiting delivery
  - Typical choices are either in the sender's domain
    - What if sender deletes/overwrites them?
  - Or in a special OS domain
    - That implies extra copying, with performance costs
- How long do messages hang around?
  - Delivered ones are cleared
  - What about those for which no RECEIVE is done?
    - One choice: delete them when the receiving process exits

Remote Procedure Calls

- A more object-oriented mechanism
- Communicate by making procedure calls on other processes
  - "Remote" here really means "in another process"
  - Not necessarily "on another machine"
- They aren't in your address space
  - And don't even use the same code
- Some differences from a regular procedure call
- Typically blocking

RPC Characteristics

- Procedure calls are primary unit of computation in most languages
  - Unit of information hiding and interface specification
- Natural boundary between client and server
  - Turn procedure calls into message send/receives
- Requires both sender and receiver to be playing the same game
  - Typically both use some particular RPC standard

RPC Mechanics

- The process hosting the remote procedure might be on same computer or a different one
- Under the covers, use messages in either case
- Resulting limitations:
  - No implicit parameters/returns (e.g. global variables)
  - No call-by-reference parameters
  - Much slower than procedure calls (TANSTAAFL)
- Often used for client/server computing

RPC Operations

- Client application links to local procedures
  - Calls local procedures, gets results
  - All RPC implementation is inside those procedures
- Client application does not know about details
  - Does not know about formats of messages
  - Does not worry about sends, timeouts, resends
  - Does not know about external data representation
- All generated automatically by RPC tools
  - The key to the tools is the interface specification
- Failure in callee doesn't crash caller

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

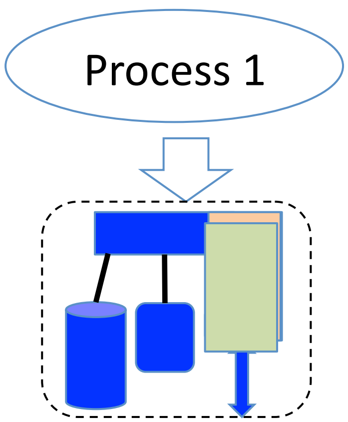
- Handles problem of not having infinite space
- Ensures that fast sender doesn't overwhelm slow receiver
- Provides well-defined, simple behavior for receiver
- But subject to some synchronization issues
  - The producer/consumer problem
  - A good abstraction for exploring those issues

The Bounded Buffer

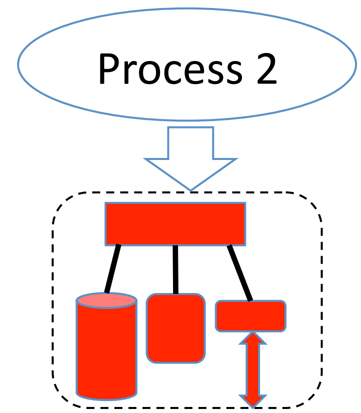
Process 1 is the writer

Process 2 is the reader

What could possibly go wrong?



A fixed size buffer



Process 1 SENDs a message through the buffer

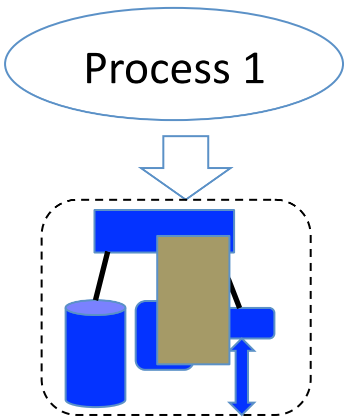
More messages are sent

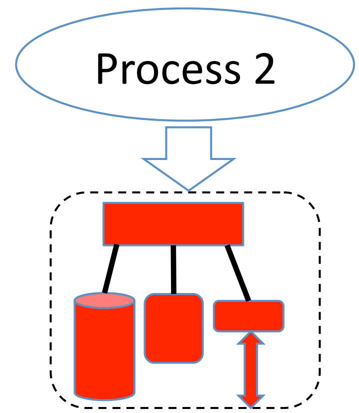
And received

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

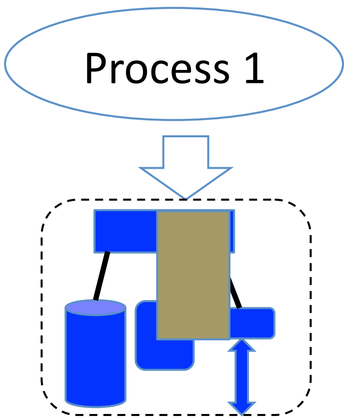
- What exactly are the processes looping on?
- They care about how many messages are in the bounded buffer
- That count is probably kept in a variable
  - Incremented on SEND
  - Decremented on RECEIVE
  - Never to go below zero or exceed buffer size
- The actual system code would test the variable

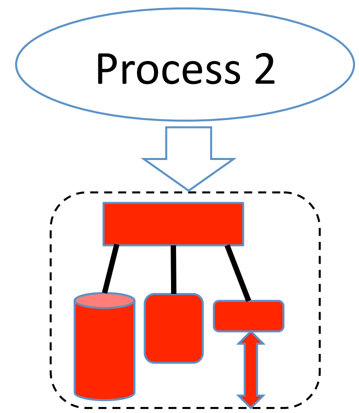
A Potential Danger

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?

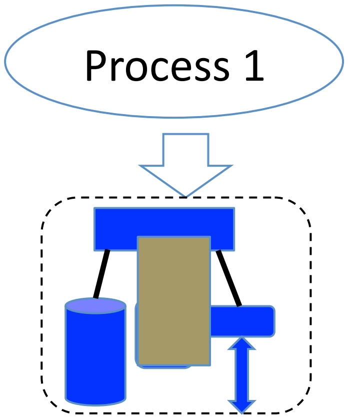
One Possible Solution

- 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



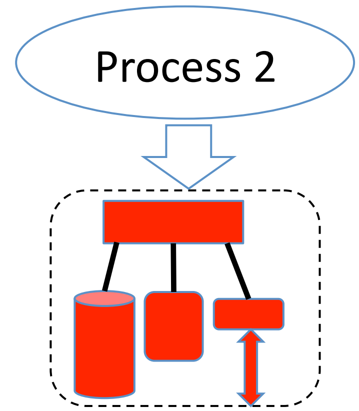
Process 1 wants to SEND

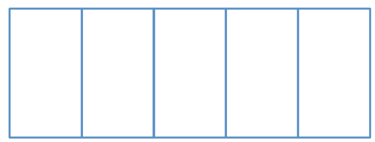
Processes 1 and 3 are senders

Process 2 is a receiver

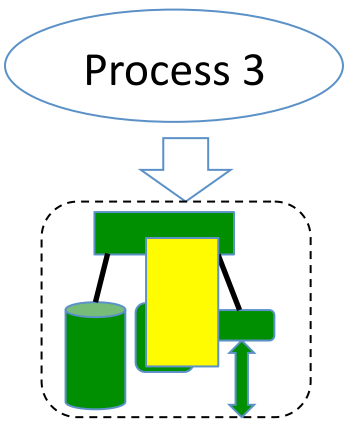
There's plenty of room in the buffer for both

But . . .





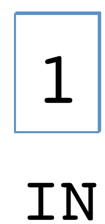
The buffer starts empty



Process 3 wants to SEND

We're in trouble:

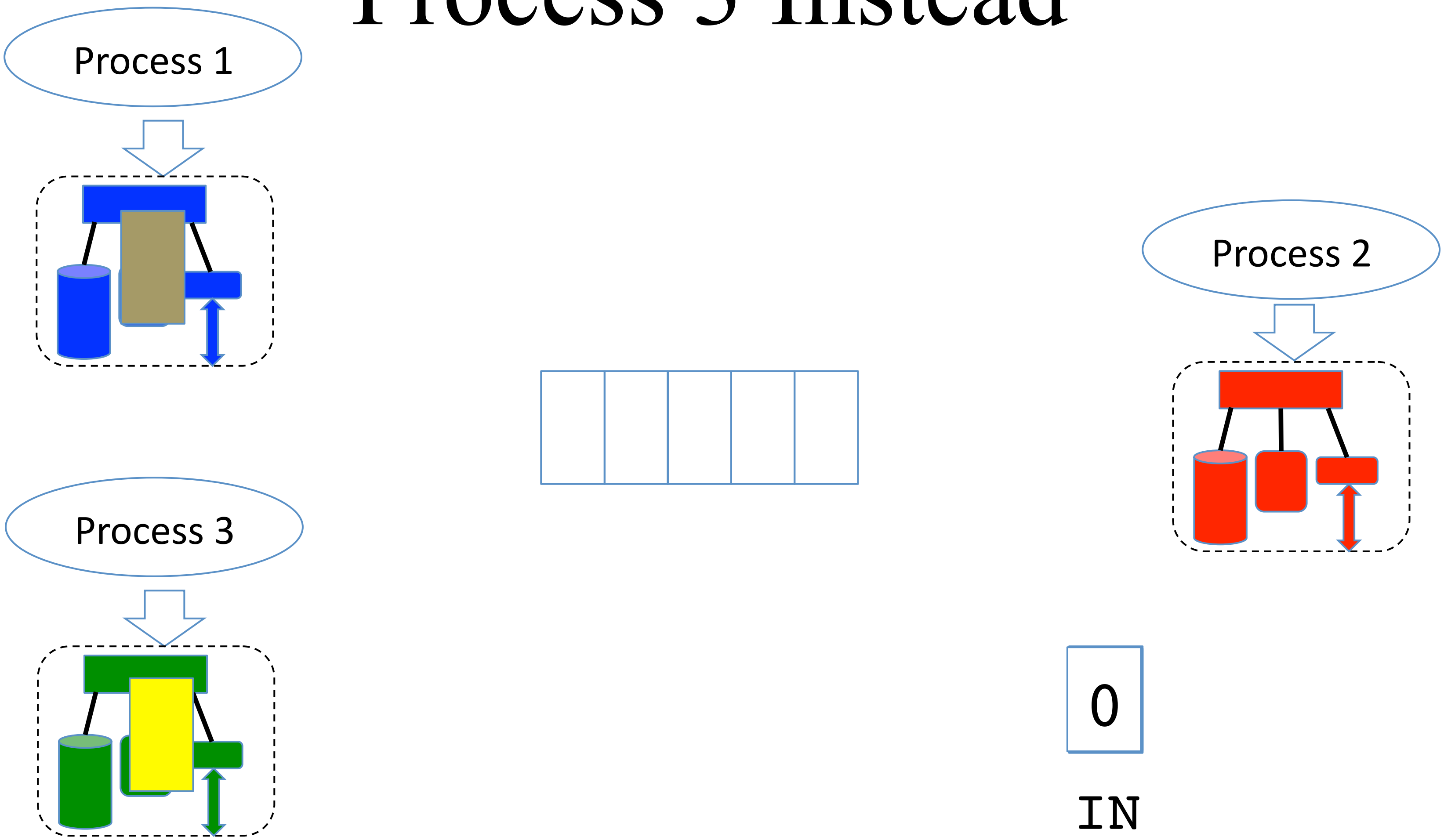
We overwrote process 1's message



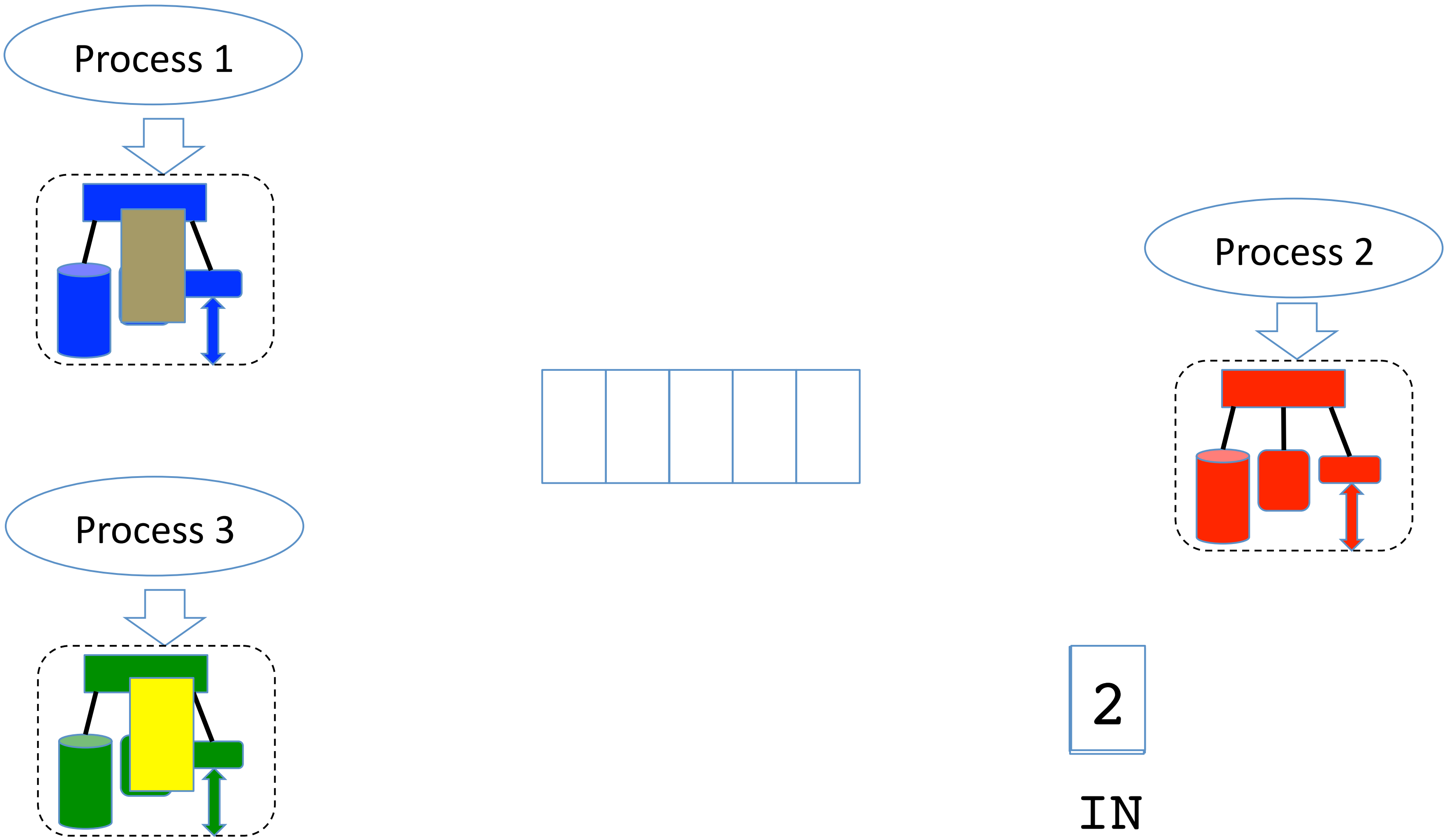
The Source of the Problem

- 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

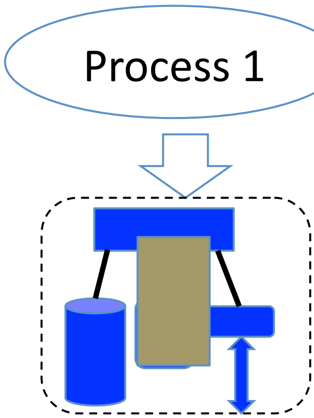
Locks

- 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

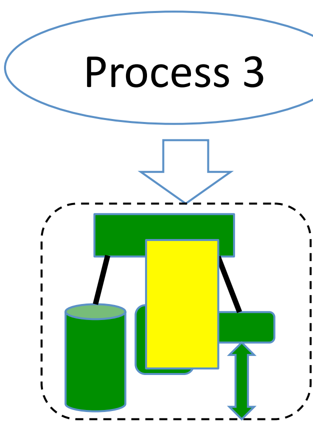
Using Locks in Our Multiple Sender Problem



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

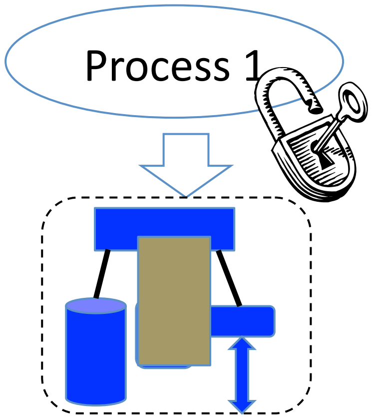
0

`IN`

`IN = 0`

`0 < 5` ✔

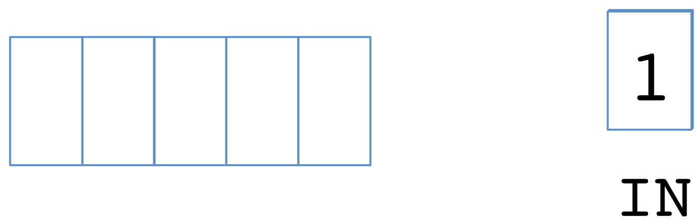
The Lock in Action

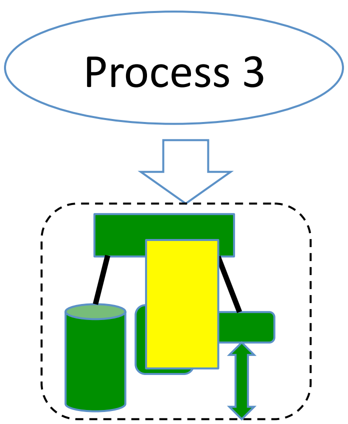


Process 1 executes ACQUIRE on the lock

Let's assume it succeeds

Now process 1 executes the code associated with the lock



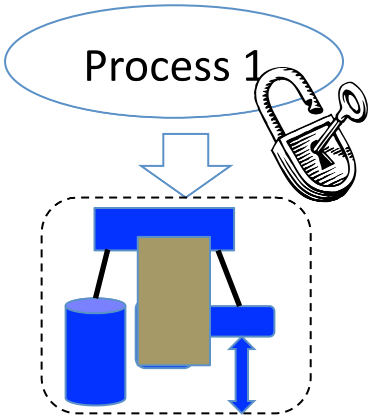


1. Read the value of `IN`
2. If `IN < BUFFER_SIZE`, store message
3. Add 1 to `IN`

Process 1 now executes RELEASE on the lock

IN = 0

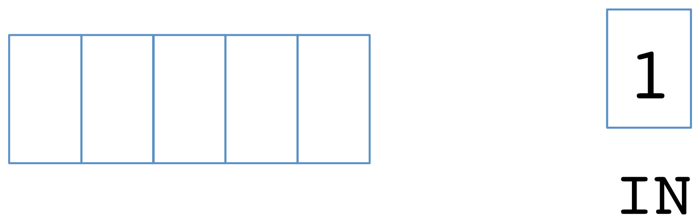
What If Process 3 Intervenes?



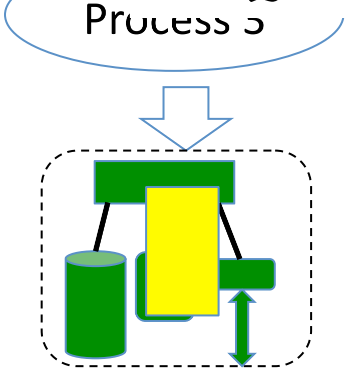
Let's say process 1 has the lock already

And has read `IN`

So process 1 can safely complete the `SEND`



`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

Parallelism and Concurrency

- Running parallel threads of execution has many benefits and is increasingly important
- Making use of parallelism implies concurrency
  - Multiple actions happening at the same time
  - Or perhaps appearing to do so
- That's difficult, because if two execution streams are not synchronized
  - Results depend on the order of instruction execution
  - Parallelism makes execution order non-deterministic
  - Understanding possible outcomes of the computation becomes combinatorially intractable

Solving the Parallelism Problem

- There are actually two interdependent problems
  - Critical section serialization
  - Notification of asynchronous completion
- They are often discussed as a single problem
  - Many mechanisms simultaneously solve both
  - Solution to either requires solution to the other
- But they can be understood and solved separately

The Critical Section Problem

- A *critical section* is a resource that is shared by multiple threads
  - By multiple concurrent threads, processes or CPUs
  - By interrupted code and interrupt handler
- Use of the resource changes its state
  - Contents, properties, relation to other resources
- Correctness depends on execution order
  - When scheduler runs/preempts which threads
  - Relative timing of asynchronous/independent events

The Asynchronous Completion Problem

- Parallel activities happen at different speeds
- Sometimes one activity needs to wait for another to complete
- The *asynchronous completion problem* is how to perform such waits without killing performance
  - Without wasteful spins/busy-waits
- Examples of asynchronous completions
  - Waiting for a held lock to be released
  - Waiting for an I/O operation to complete
  - Waiting for a response to a network request
  - Delaying execution for a fixed period of time

Critical Sections

- What is a critical section?
- Functionality whose proper use in parallel programs is critical to correct execution
- If you do things in different orders, you get different results
- A possible location for undesirable non-determinism

Basic Approach to Critical Sections

- Serialize access
  - Only allow one thread to use it at a time
  - Using some method like locking
- Won’t that limit parallelism?
  - Yes, but . . .
- If true interactions are rare, and critical sections well defined, most code still parallel
- If there are actual frequent interactions, there isn’t any real parallelism possible
  - Assuming you demand correct results

Critical Section Example 1: Updating a File

Process 1

```
remove("database");
fd = create("database");
write(fd,newdata,length);
close(fd);
```

Process 2

```
fd = open("database",READ);
count = read(fd,buffer,length);
```

```
remove("database");
fd = create("database");
                          fd = open("database",READ);
                          count = read(fd,buffer,length);
write(fd,newdata,length);
close(fd);
```

- Process 2 reads an empty database
  - This result could not occur with any sequential execution

Critical Section Example 2: Multithreaded Banking Code

Thread 1

```
load r1, balance   // = 100
load r2, amount1 // = 50
add r1, r2             // = 150
store r1, balance  // = 150
```

```
load r1, b
load r2, a
add r1, r
```

Thread 2

```
load r1, balance    // = 100
load r2, amount2 // = 25
sub r1, r2              // = 75
store r1, balance   // = 75
```

The $25 debit was lost!!!

```
load r1, balance    // = 100
load r2, amount2 // = 25
sub r1, r2              // = 75
store r1, balance   // = 75
```

CONTEXT SWITCH!!!

```
store r1, balance  // = 150
```

amount1 **50**   balance **150**   amount2 **25**

r1 **75**

r2 **50**

These Kinds of Interleavings Seem Pretty Unlikely

- To cause problems, things have to happen exactly wrong
- Indeed, that's true
- But modern machines execute a billion instructions per second
- So even very low probability events can happen with frightening frequency
- Often, one problem blows up everything that follows

Can't We Solve the Problem By Disabling Interrupts?

- Much of our difficulty is caused by a poorly timed interrupt
  - Our code gets part way through, then gets interrupted
  - Someone else does something that interferes
  - When we start again, things are messed up
- Why not temporarily disable interrupts to solve those problems?
  - Can't be done in user mode
  - Harmful to overall performance
  - Dangerous to correct system behavior

Another Approach

- Avoid shared data whenever possible
  - No shared data, no critical section
  - Not always feasible
- Eliminate critical sections with *atomic instructions*
  - Atomic (uninteruptable) read/modify/write operations
  - Can be applied to 1-8 contiguous bytes
  - Simple: increment/decrement, and/or/xor
  - Complex: test-and-set, exchange, compare-and-swap
  - What if we need to do more in a critical section?
- Use atomic instructions to implement locks
  - Use the lock operations to protect critical sections

Atomic Instructions – Compare and Swap

A C description of machine instructions

```
bool compare_and_swap( int *p, int old, int new ) {
  if (*p == old) {    /* see if value has been changed      */
      *p = new;        /* if not, set it to new value        */
      return( TRUE);   /* tell caller he succeeded           */
  } else               /* value has been changed        */
    return( FALSE);    /* tell caller he failed              */
}

if (compare_and_swap(flag,UNUSED,IN_USE) {
      /* I got the critical section! */
} else {
      /* I didn't get it.  */
}
```

Solving Problem #2 With Compare and Swap

Again, a C implementation

```
int current_balance;
writecheck( int amount ) {
  int oldbal, newbal;
  do {
      oldbal = current_balance;
      newbal = oldbal - amount;
      if (newbal < 0) return (ERROR);
  } while (!compare_and_swap( &current_balance, oldbal, newbal))
...
}
```

Why Does This Work?

- Remember, `compare_and_swap()` is atomic
- First time through, if no concurrency,
  - `oldbal == current_balance`
  - `current_balance` was changed to `newbal` by `compare_and_swap()`
- If not,
  - `current_balance` changed after you read it
  - So `compare_and_swap()` didn't change `current_balance` and returned FALSE
  - Loop, read the new value, and try again

Will This Really Solve the Problem?

- If compare & swap fails, loop back and re-try
  - If there is a conflicting thread isn't it likely to simply fail again?
- Only if preempted during a four instruction window
  - By someone executing the same critical section
- Extremely low probability event
  - We will very seldom go through the loop even twice

Limitation of Atomic Instructions

- They only update a small number of contiguous bytes
  - Cannot be used to atomically change multiple locations
    - E.g., insertions in a doubly-linked list
- They operate on a single memory bus
  - Cannot be used to update records on disk
  - Cannot be used across a network
- They are not higher level locking operations
  - They cannot “wait” until a resource becomes available
  - You have to program that up yourself
    - Giving you extra opportunities to screw up

Implementing Locks

- Create a synchronization object
  - Associated it with a critical section
  - Of a size that an atomic instruction can manage
- Lock the object to seize the critical section
  - If critical section is free, lock operation succeeds
  - If critical section is already in use, lock operation fails
    - It may fail immediately
    - It may block until the critical section is free again
- Unlock the object to release critical section
  - Subsequent lock attempts can now succeed
  - May unblock a sleeping waiter

Criteria for Correct Locking

- How do we know if a locking mechanism is correct?
- Four desirable criteria:
  1. Correct mutual exclusion
     - Only one thread at a time has access to critical section
  2. Progress
     - If resource is available, and someone wants it, they get it
  3. Bounded waiting time
     - No indefinite waits, guaranteed eventual service
  4. And (ideally) fairness
     - E.g. FIFO