Process Communications, Synchronization, and Concurrency

CS 111
Operating Systems
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

- Signals
- Sharing memory
- Messages
- RPC
- More sophisticated abstractions
  - The bounded buffer

Signals

- A very simple (and limited) communications mechanism
- Essentially, send an interrupt to a process
  - With some kind of tag indicating what sort of interrupt it is
- Depending on implementation, process may actually be interrupted
- Or may have some non-interrupting condition code raised
  - Which it would need to check for

Properties of Signals

- Unidirectional
- Low information content
  - Generally just a type
  - Thus not useful for moving data
- Not always possible for user processes to signal each other
  - May only be used by OS to alert user processes
  - Or possibly only through parent/child process relationships

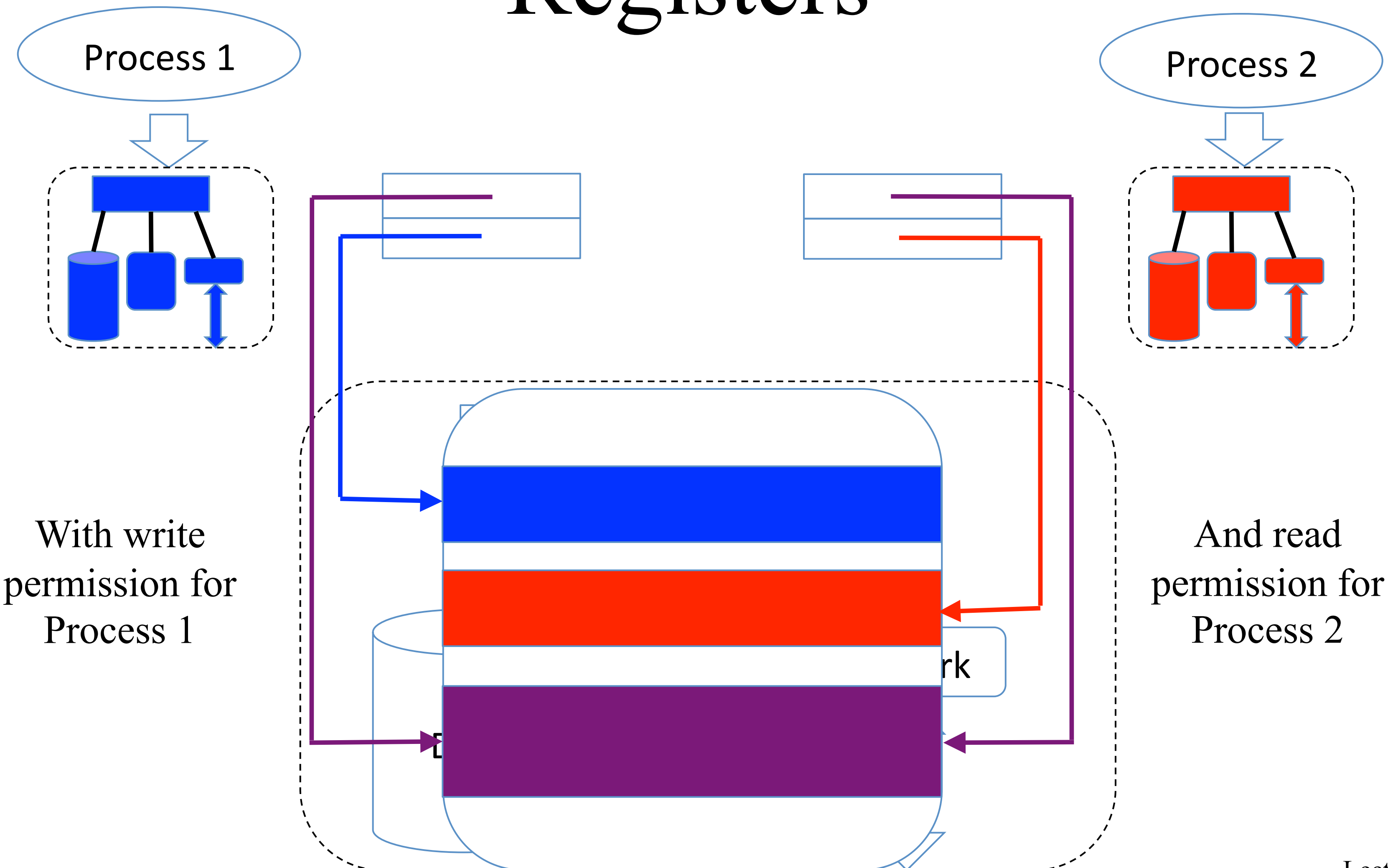
Implementing Signals

- Typically through the trap/interrupt mechanism
- OS (or another process) requests a signal for a process
- That process is delivered a trap or interrupt implementing the signal
- There's no associated parameters or other data
  - So no need to worry about where to put or find that

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

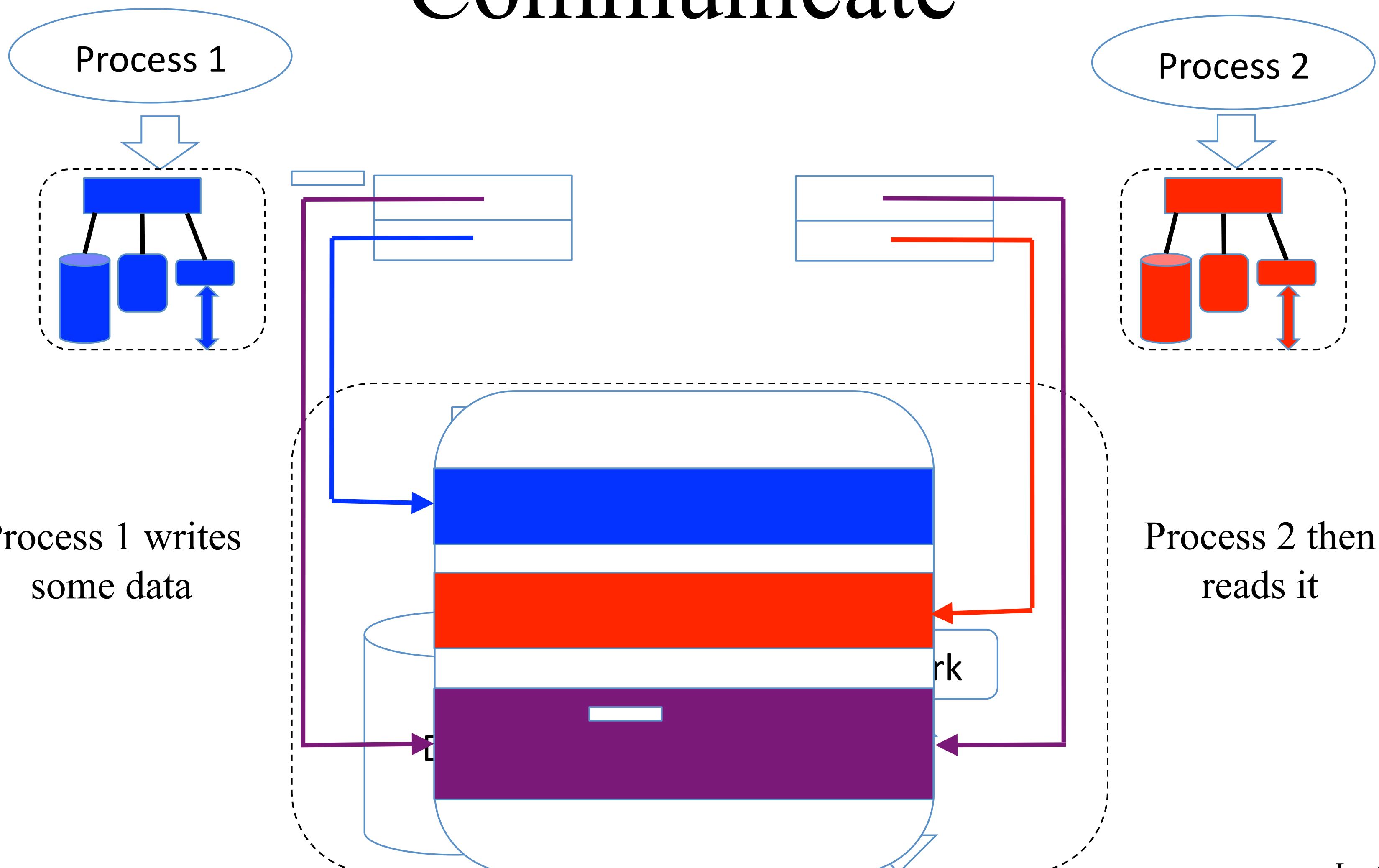
Sharing Memory With Domain Registers



With write permission for Process 1

And read permission for Process 2

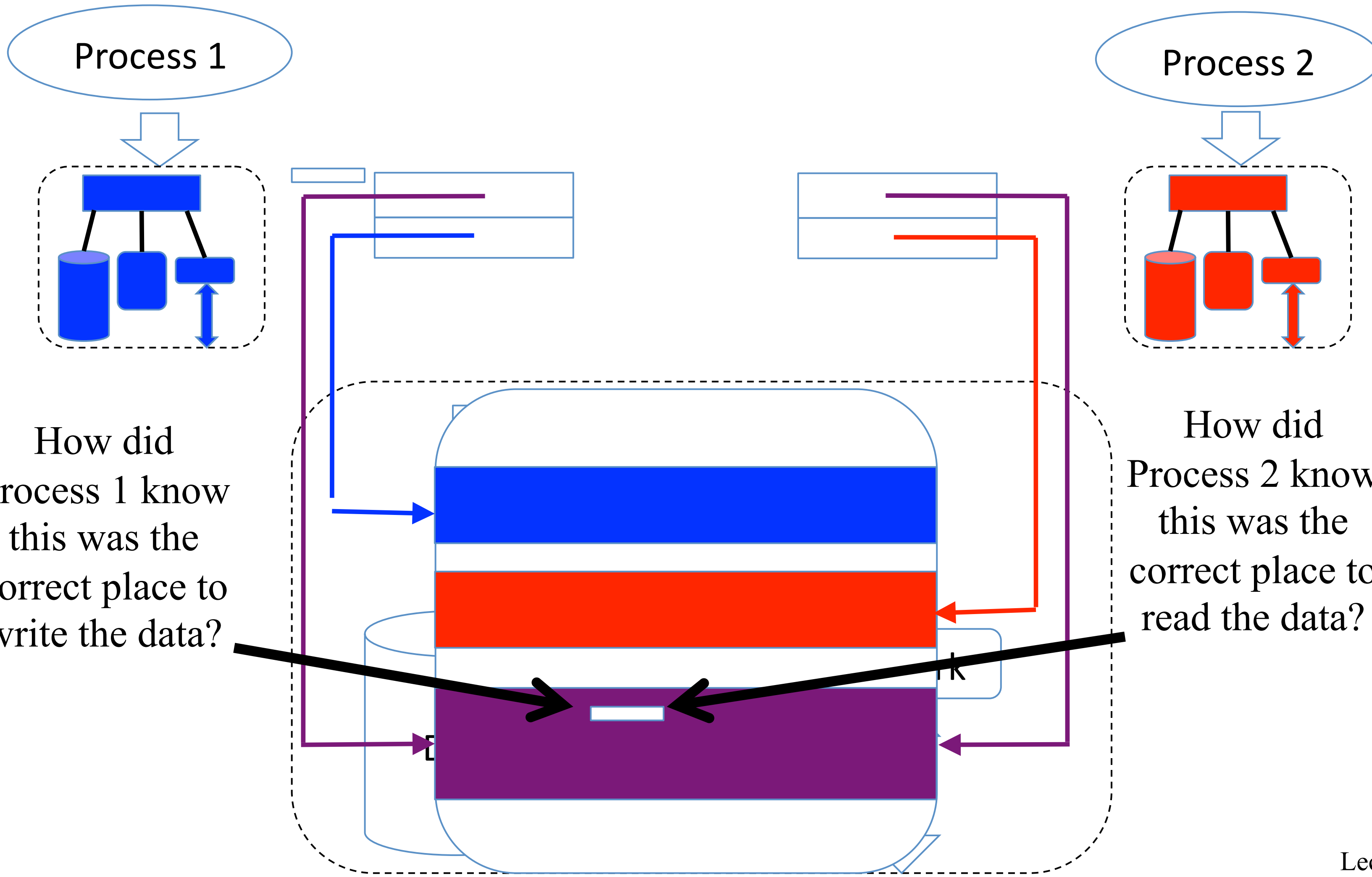
Using the Shared Domain to Communicate



Process 1 writes some data

Process 2 then reads it

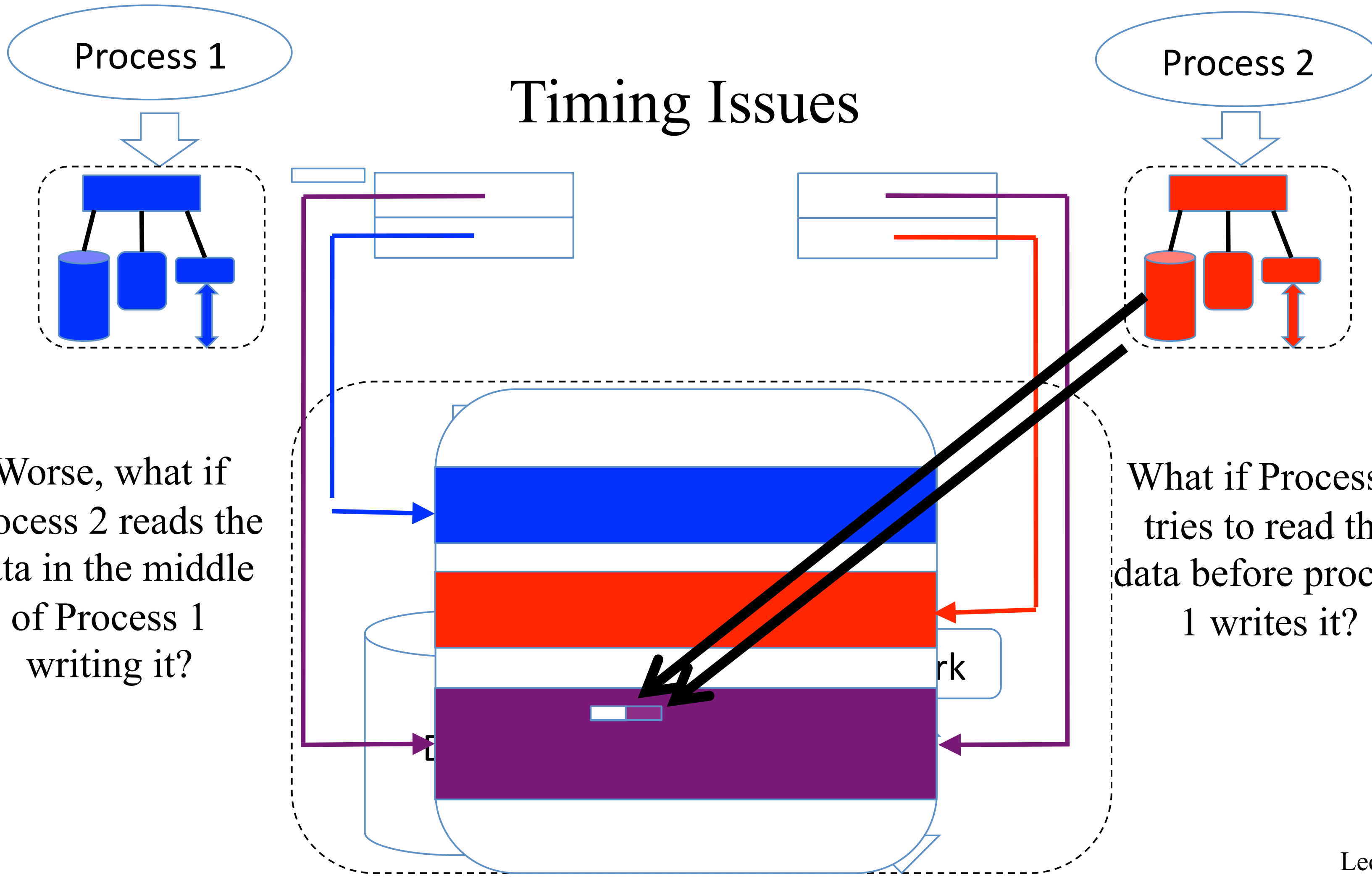
Potential Problem #1 With Shared Domain Communications



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



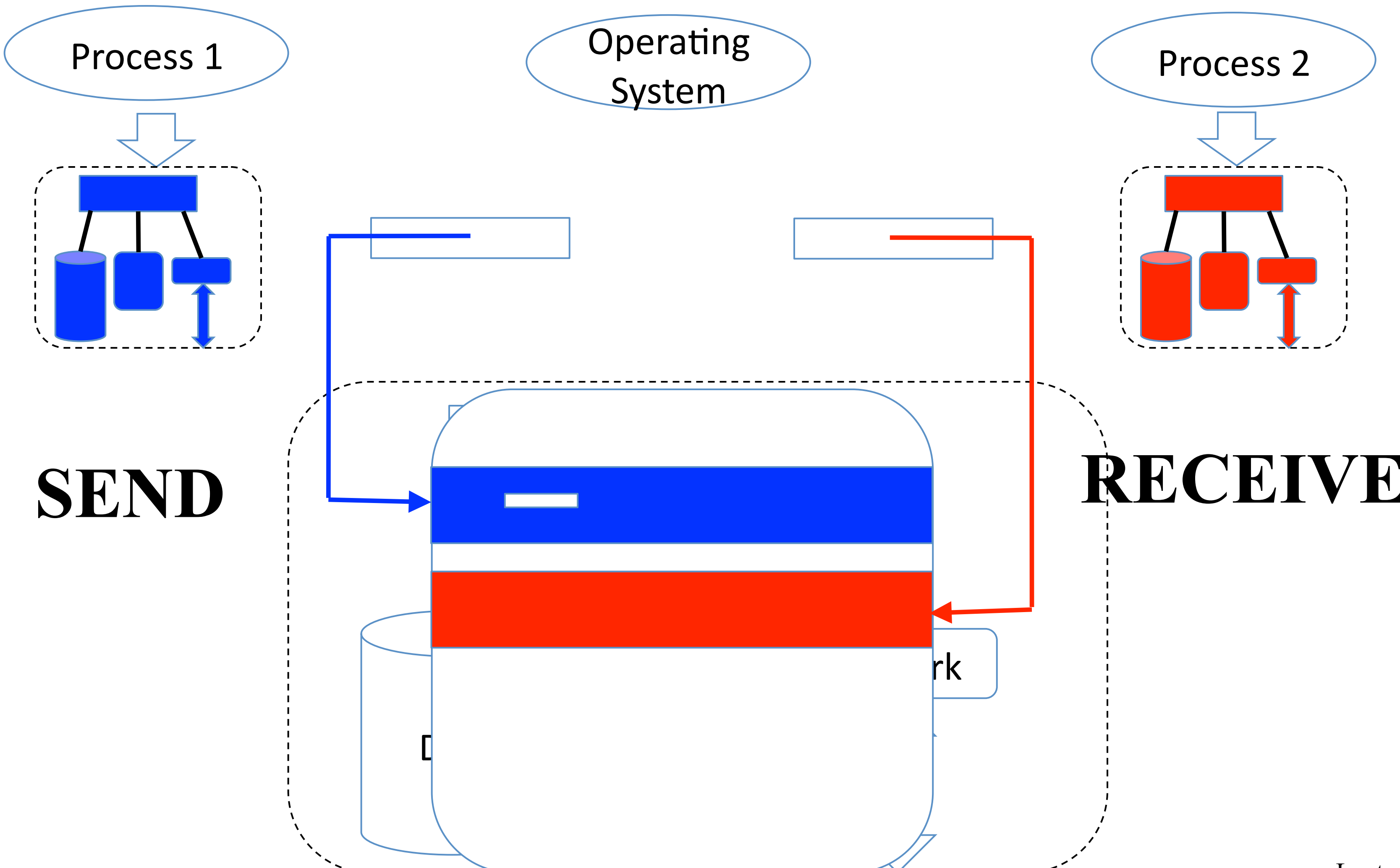
Worse, what if Process 2 reads the data in the middle of Process 1 writing it?

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, resents
  - 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