EEL-4736/5737 Principles of Computer System Design

Lecture Slides 2

Textbook Chapter 2

Fundamental Abstractions of Computer Systems

Fundamental Abstractions

- There is an unlimited number of potential abstractions one can conceive for computer systems
- Fundamental abstractions:
  - Memory
  - Interpreter
  - Communication link

Fundamental abstractions

- These abstractions cover the fundamental functions one expect from a computer system:
  - Storage and recall
  - Processing
  - Communication
- Pervasive: in underlying hardware devices, in large-scale distributed systems
- Designers elaborate on these fundamental abstractions, casting them in different ways for different applications

Fundamental abstractions

- A primary method by which components interact is by using references – *names*
  - Names appear in the interface of all three fundamental abstractions, as well as in higher-level counterparts
- Let us consider each abstraction
  - What/how it does
  - Its interface
  - How names are used for interconnection

Memory

- Remembers data for use in computation
  - Storage
- Fundamental abstract model:

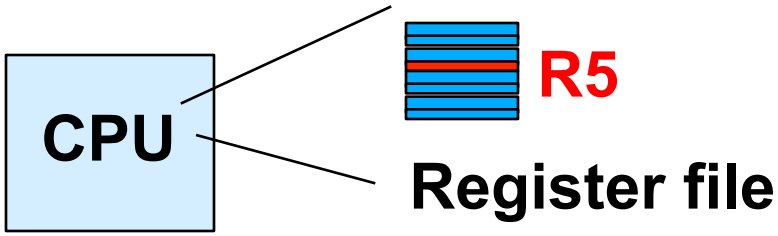
  WRITE (name, value)

  Value <- READ (name)
- Property:
  - Durability: length of time storage medium holds data that is stored
  - Volatile: needs energy supply to retain data

Example

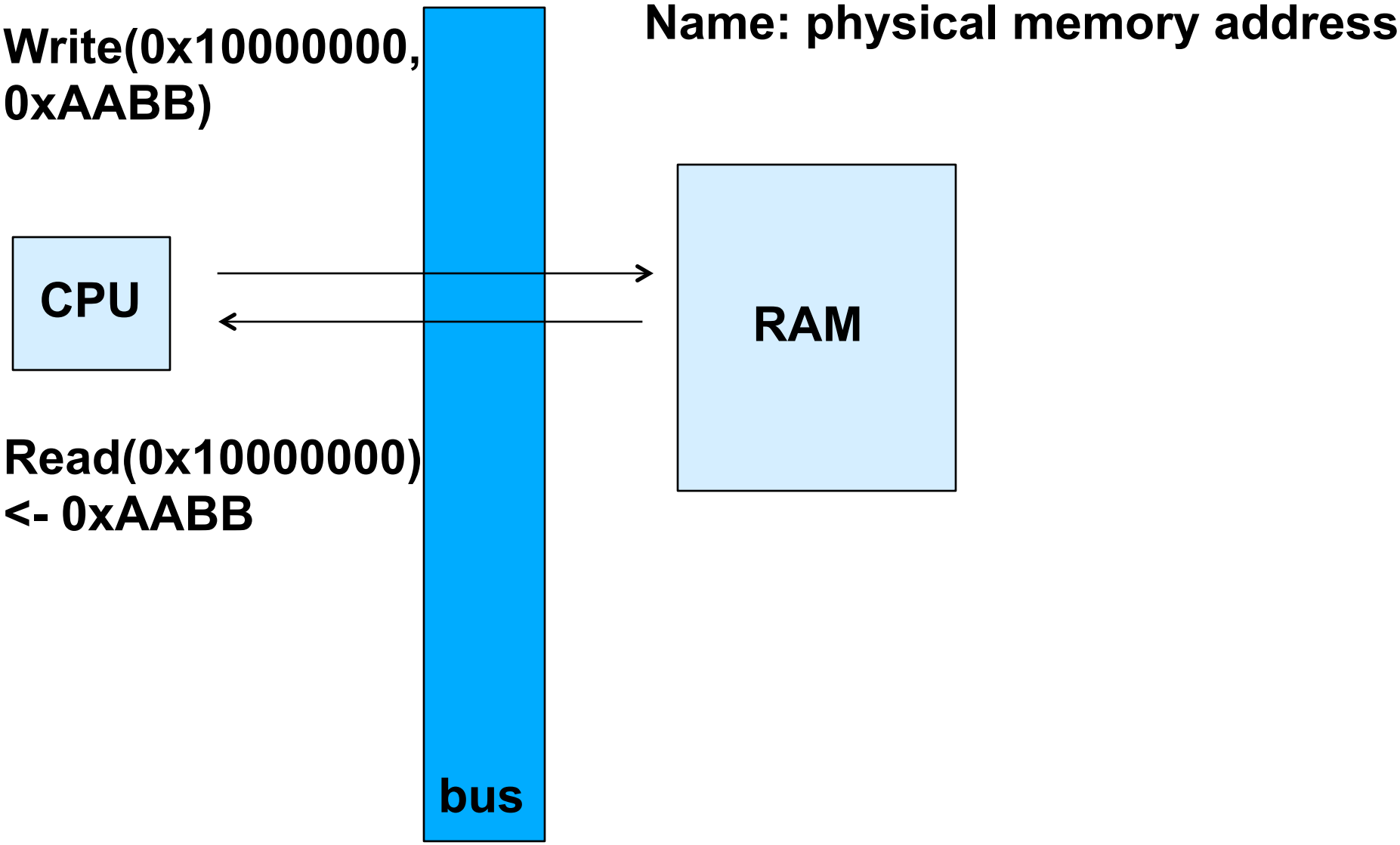
Name: register number

Write(R5, 0xAABB)

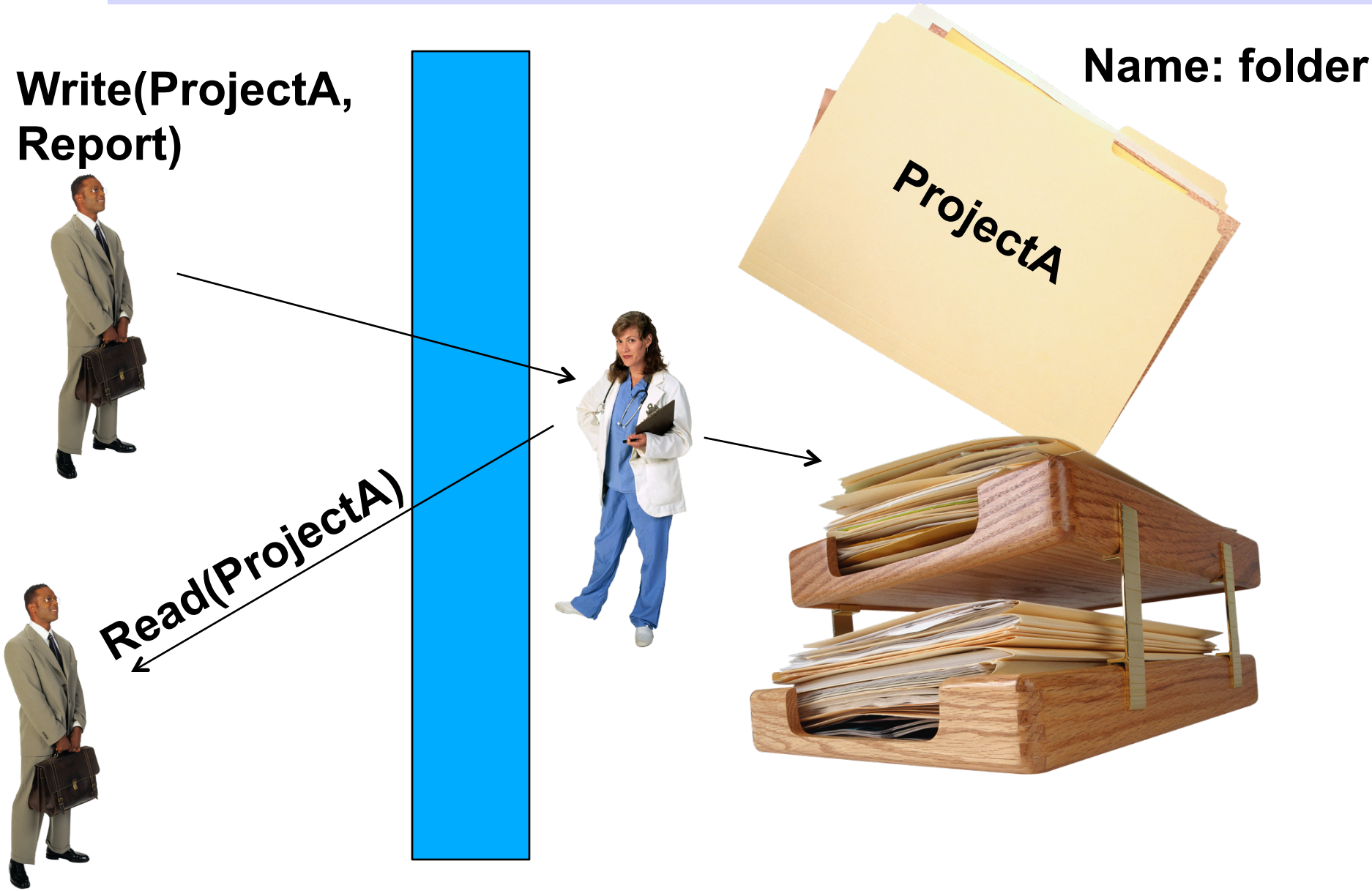


Read(R5) <- 0xAABB

Example


Name: physical memory address
Write(0x10000000, 0xAABB)
CPU
RAM
Read(0x10000000) <- 0xAABB
bus

Example



Read/write coherence, atomicity

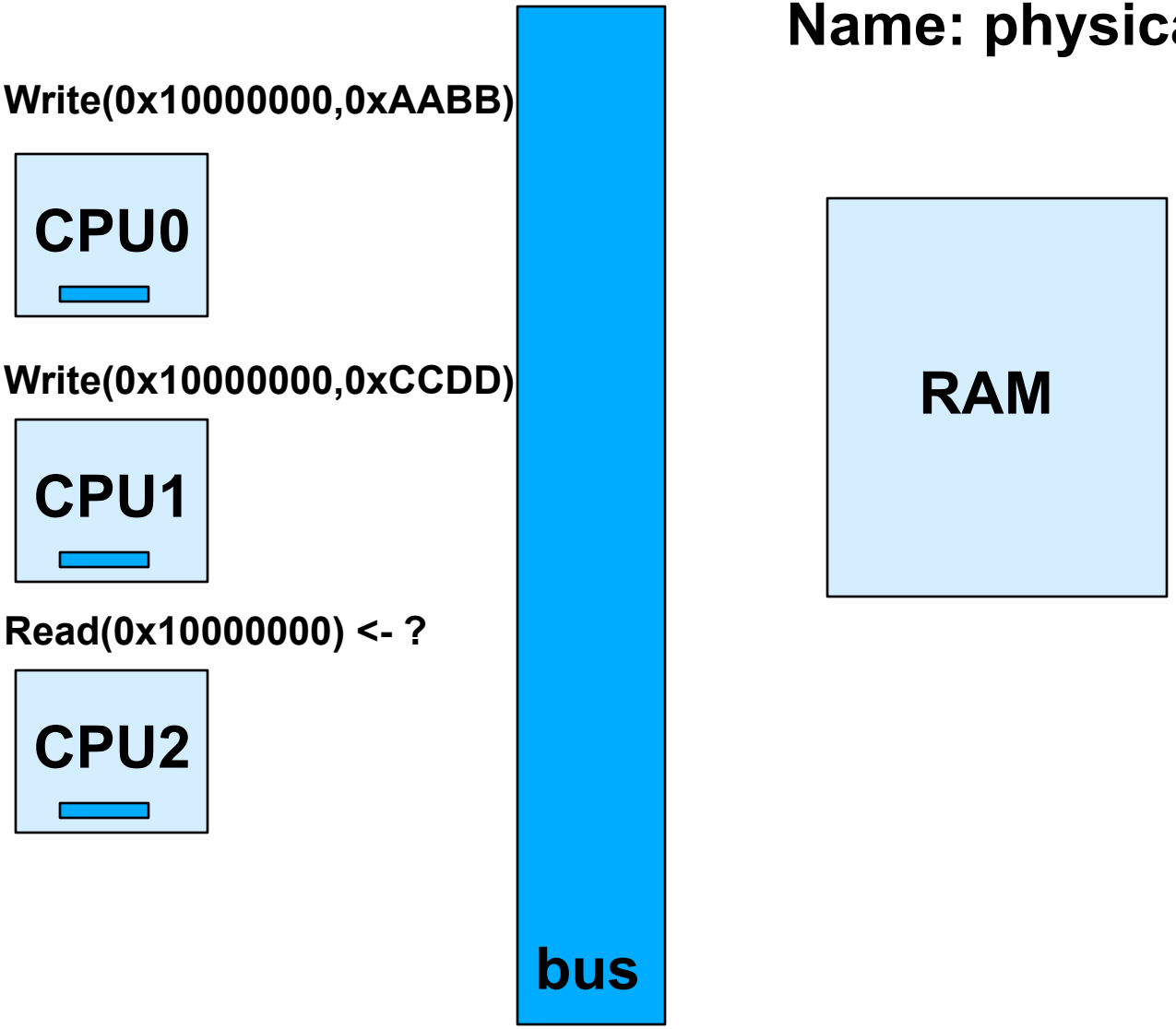
- Read/write coherence:
  - Result of a READ of a named cell is the same as the most recent WRITE to the same cell
- Before/after atomicity:
  - Result of every READ or WRITE is as if the READ or WRITE occurred either completely before or completely after a previous READ or WRITE
- Intuitive behaviors you would expect from a memory where there is one writer and only one storage cell for each name

Read/write coherence, atomicity

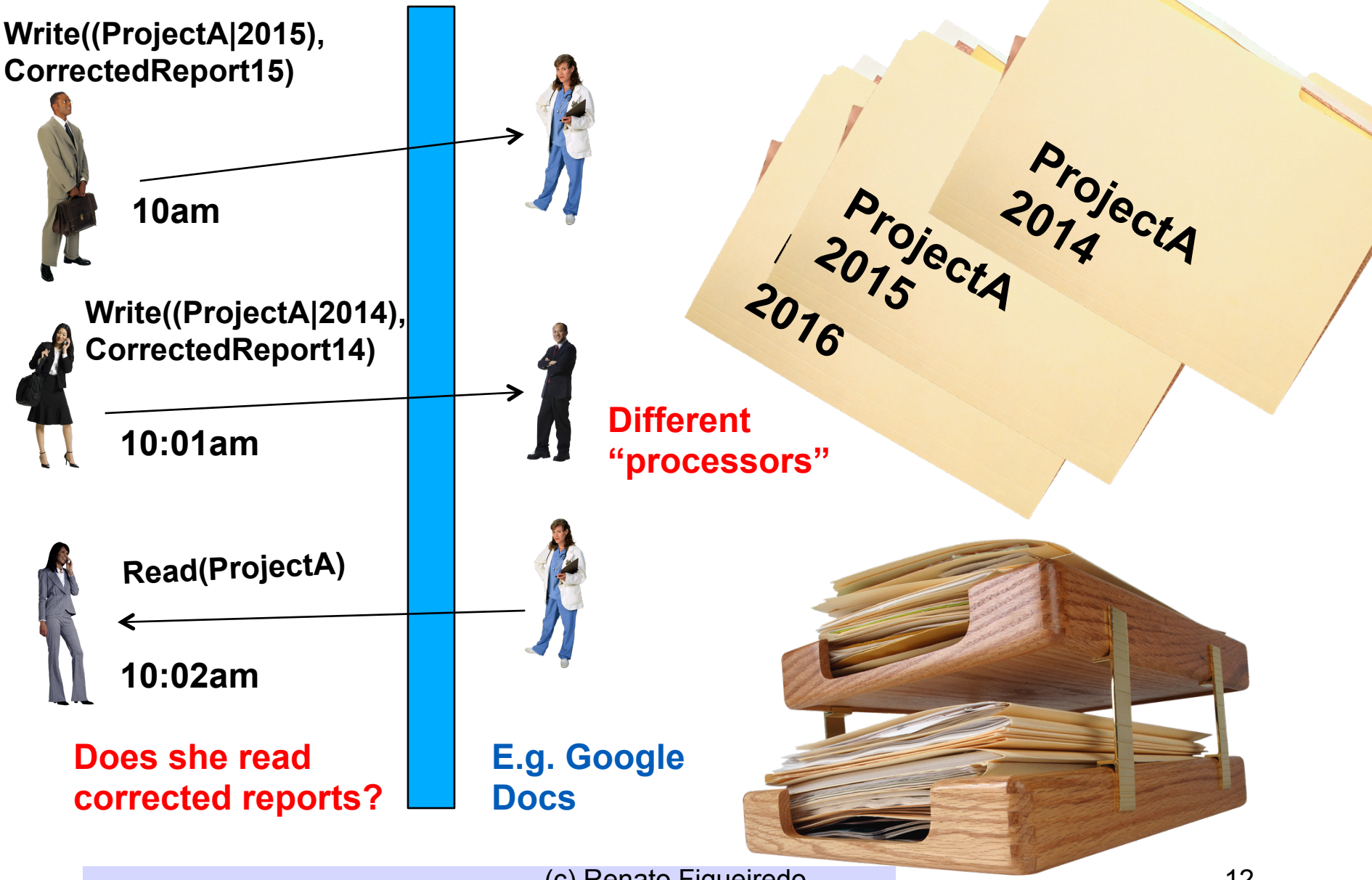
- But it's not so simple
  - Concurrent systems
    - Two subsystems initiate READ/WRITE to the same cell at about the same time – arbitration, ordering
  - Remote storage
    - Same concerns, amplified by long delays. Which write was first? Failures?
  - Cell size != value size
    - Stored value may span across multiple storage cells; implications on before/after atomicity
  - Replicated storage
    - For performance and availability reasons, may desire to keep multiple copies – e.g. caches

Example

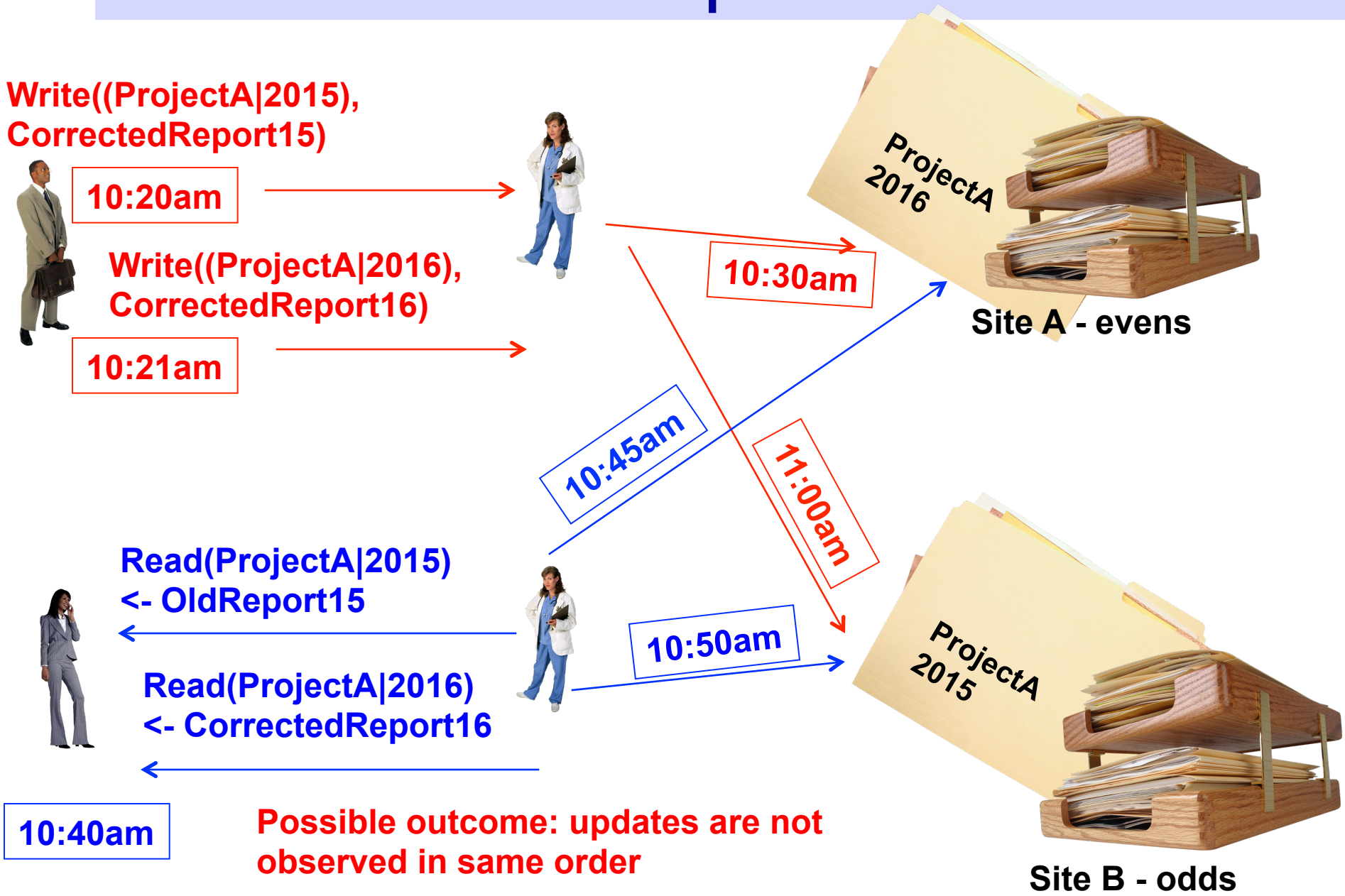
Name: physical memory address



Example


Write((ProjectA|2015), CorrectedReport15)
10am
Write((ProjectA|2014), CorrectedReport14)
10:01am
Different "processors"
Read(ProjectA)
10:02am
Does she read corrected reports?
E.g. Google Docs
ProjectA 2014
ProjectA 2015
2016

Example


Write((ProjectA|2015), CorrectedReport15)
10:20am
Write((ProjectA|2016), CorrectedReport16)
10:21am
10:30am
ProjectA 2016
Site A - evens
10:45am
11:00am
Read(ProjectA|2015) <- OldReport15
Read(ProjectA|2016) <- CorrectedReport16
10:50am
ProjectA 2015
Site B - odds
10:40am
Possible outcome: updates are not observed in same order

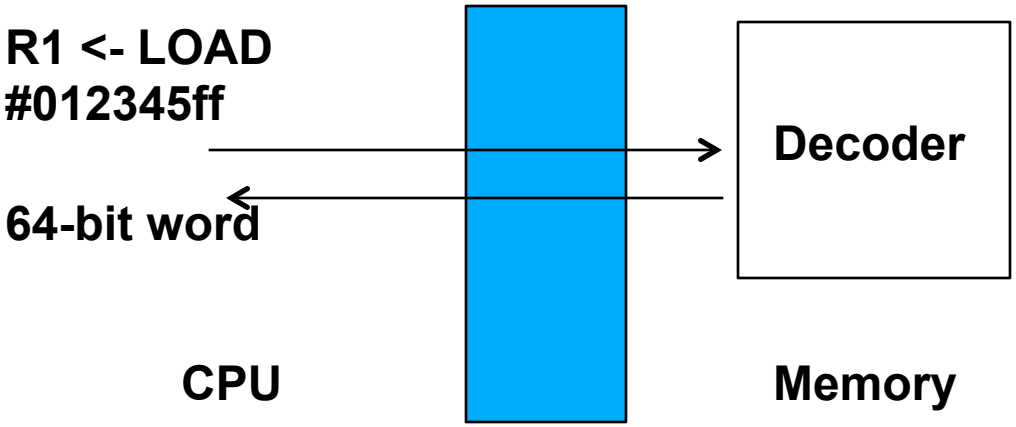
Memory latency

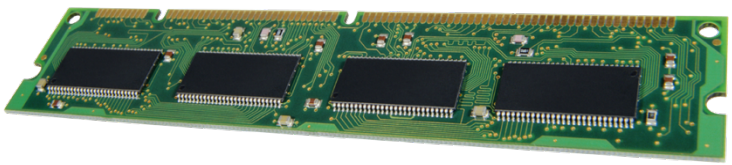
- How long does a READ or WRITE take to complete?
  - RAM: few to hundreds of nanoseconds
  - Hard disk: milliseconds
- Random access
  - Latency is largely independent of name
  - Non-random access devices (e.g. hard disks), latency depends on previous accesses
    - Mechanical delay of moving disk head

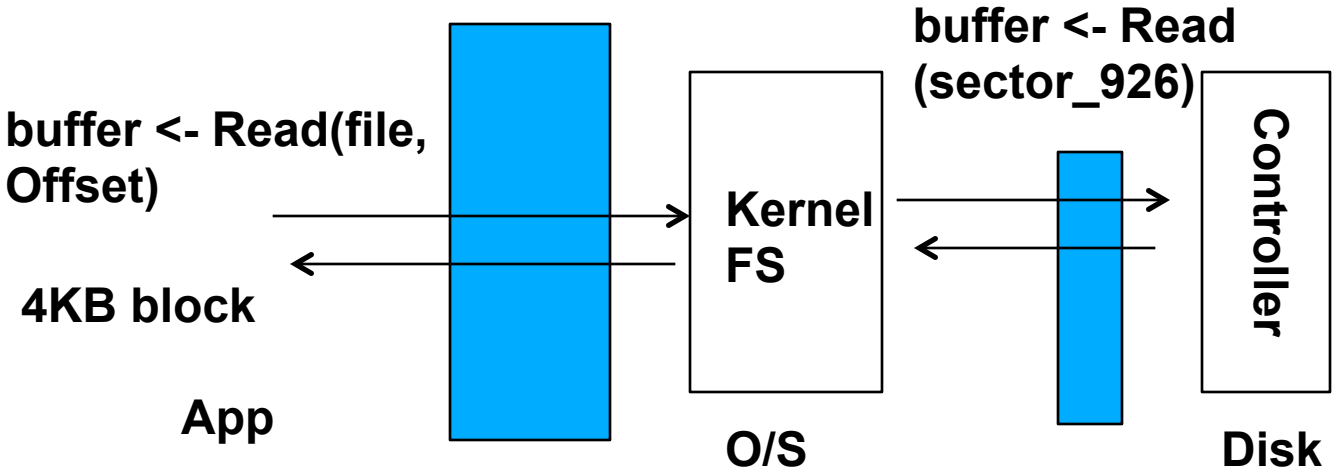
Memory

- What are valid names? Values?
  - Depends on where abstraction is used
- At the physical level
  - Names and values are constrained and designed for efficient hardware manipulation
- Higher level – more flexibility
  - E.g. file names and contents

Example


R1 <- LOAD
#012345ff
64-bit word
Decoder
CPU
Memory



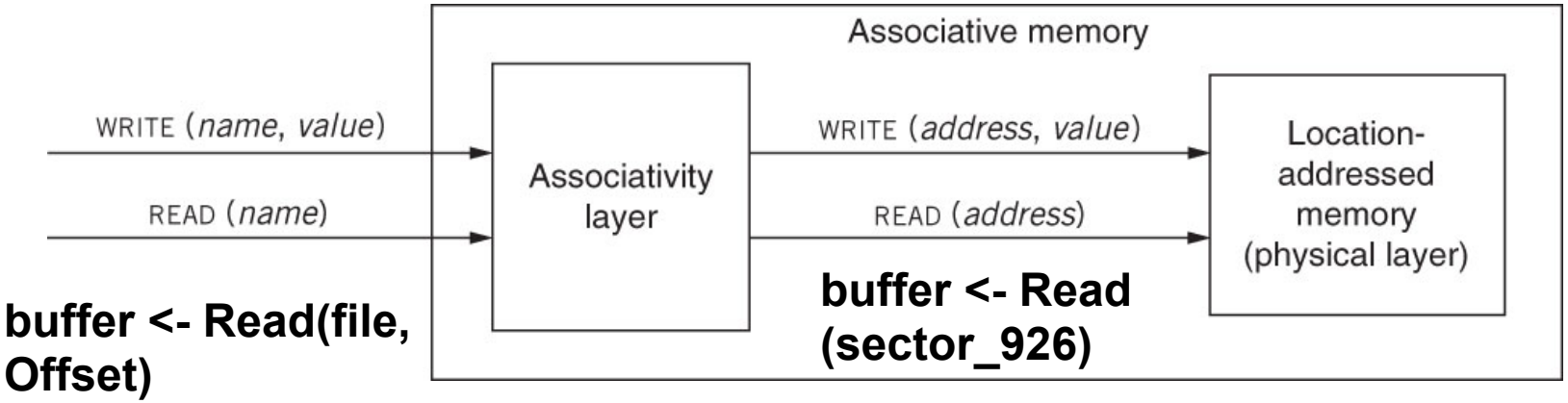

buffer <- Read(file,
Offset)
4KB block
App
Kernel
FS
O/S
buffer <- Read
(sector_926)
Controller
Disk

Location-addressed Memory

- Simpler to design hardware that maps geometric coordinates to/from consecutive integers
  - Efficient hardware decoding
  - Efficient layout of cell arrays
  - Efficient readout/multiplexing
- *Location-addressed memory*
  - Names are integer *addresses;* set: *address space* (or name space)

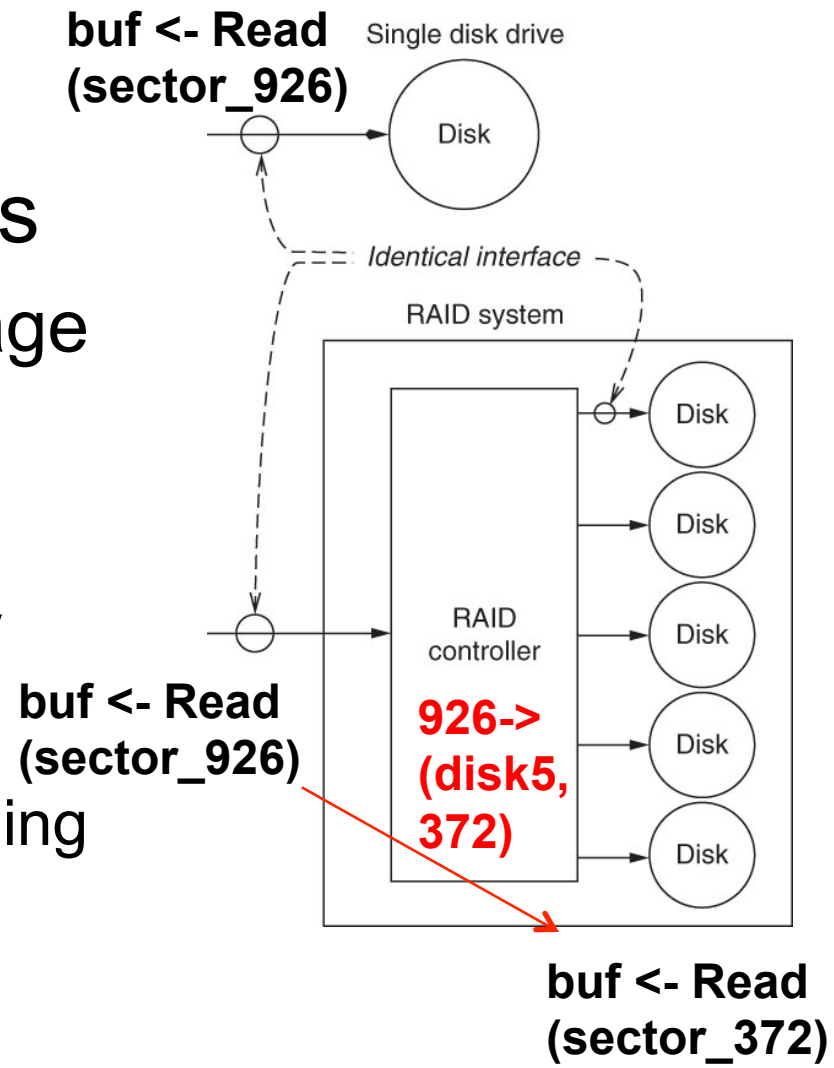
Associative Memory

- Location-addressed
  - Hardware-efficient, but not user-friendly
- Associative memory accepts unconstrained names



Example - RAID

- Redundant Array of Inexpensive Disks
- Combine multiple disks
  - Larger aggregate storage
    - Larger name space
  - Improved performance
    - Read/write concurrently
  - Improved durability
    - Replicated storage; coding



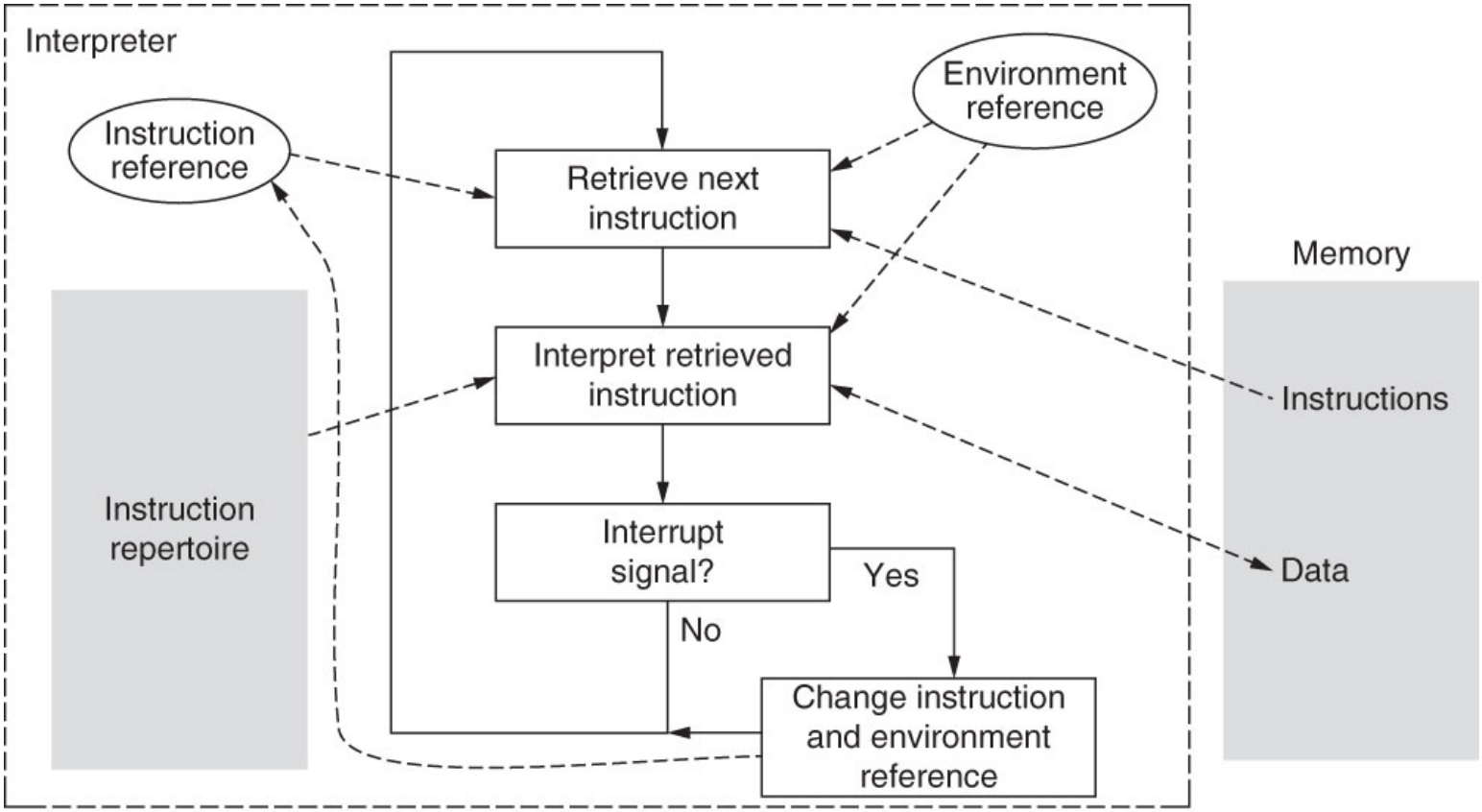
Interpreters

- *Active* elements of a computer system
  - Perform the computation actions
- Examples:
  - A microprocessor; a disk controller
  - A Python or JavaScript interpreter
  - A virtual machine monitor

Interpreter abstraction

- Three core components:
  - A *repertoire*
    - Defines set of possible actions
  - An *instruction reference*
    - Names the location from where to find the next instruction to interpret
  - An *environment reference*
    - Tells the interpreter where to find its environment – i.e. the current state in which to perform the action of the current instruction

Interpreter loop




```
procedure INTERPRET()
    do forever
        instruction ← READ (instruction_reference)
        perform instruction in the context of environment_reference
        if interrupt_signal = TRUE then
            instruction_reference ← entry point of INTERRUPT_HANDLER
            environment_reference ← environment ref of INTERRUPT_HANDLER
```

Example

Write(ProjectA, Report)

Read(ProjectA)

Check window, listen to request
Read/Write? Search folder by name (others? deny)
Match?
- **Y: Copy contents from/to folder**
- **N: Notify not found**

Phone call from supervisor?
- **Y: Instruction from supervisor**
  - **e.g. shred sensitive docs**

"interpreter"
implements read/write instructions, handles supervisor interrupt

Example - processors

- General purpose processor is an implementation of an interpreter
- Instruction reference
  - Program counter – a register
- Environment reference
  - Registers - small set of location-addressed memory with fast access
    - (Main memory is part of the overall environment)
- Repertoire
  - Instruction set architecture (ISA)
  - ADD, SUB, JUMP, LOAD, STORE, … opcodes

Example - processors

0x44440000 LOAD R1, 0x2000FF00

R1 <- READ(0x2000FF00)

0x44440004 STORE R1, 0x2000FA00

WRITE(0x2000FA00,R1)

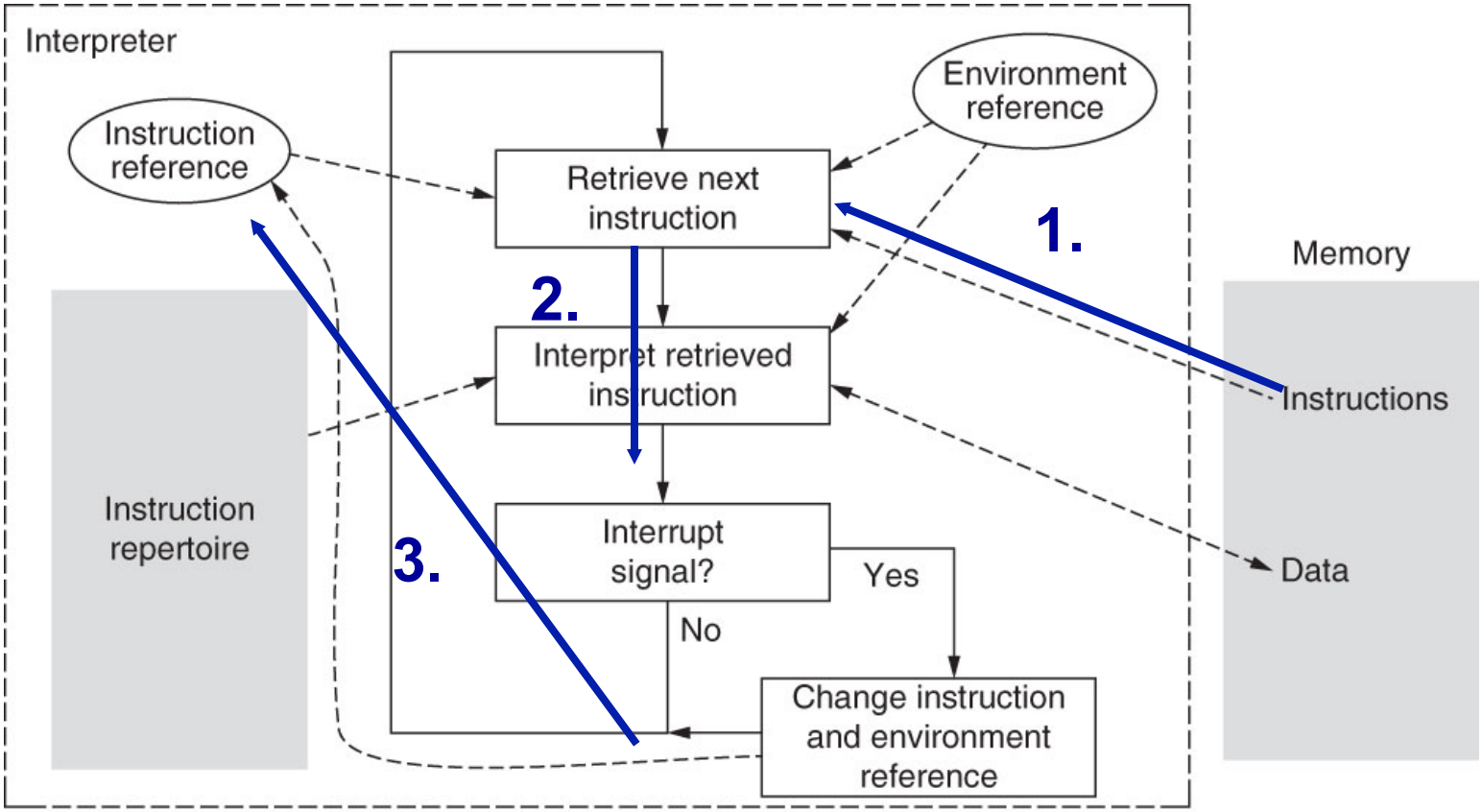
PC: instruction reference

LOAD, STORE: repertoire

R1: environment reference determines which to use

Virtual memory: specialized register (page table map) part of environment reference, determines which physical address the name 0x2000FF00 maps to

Processor



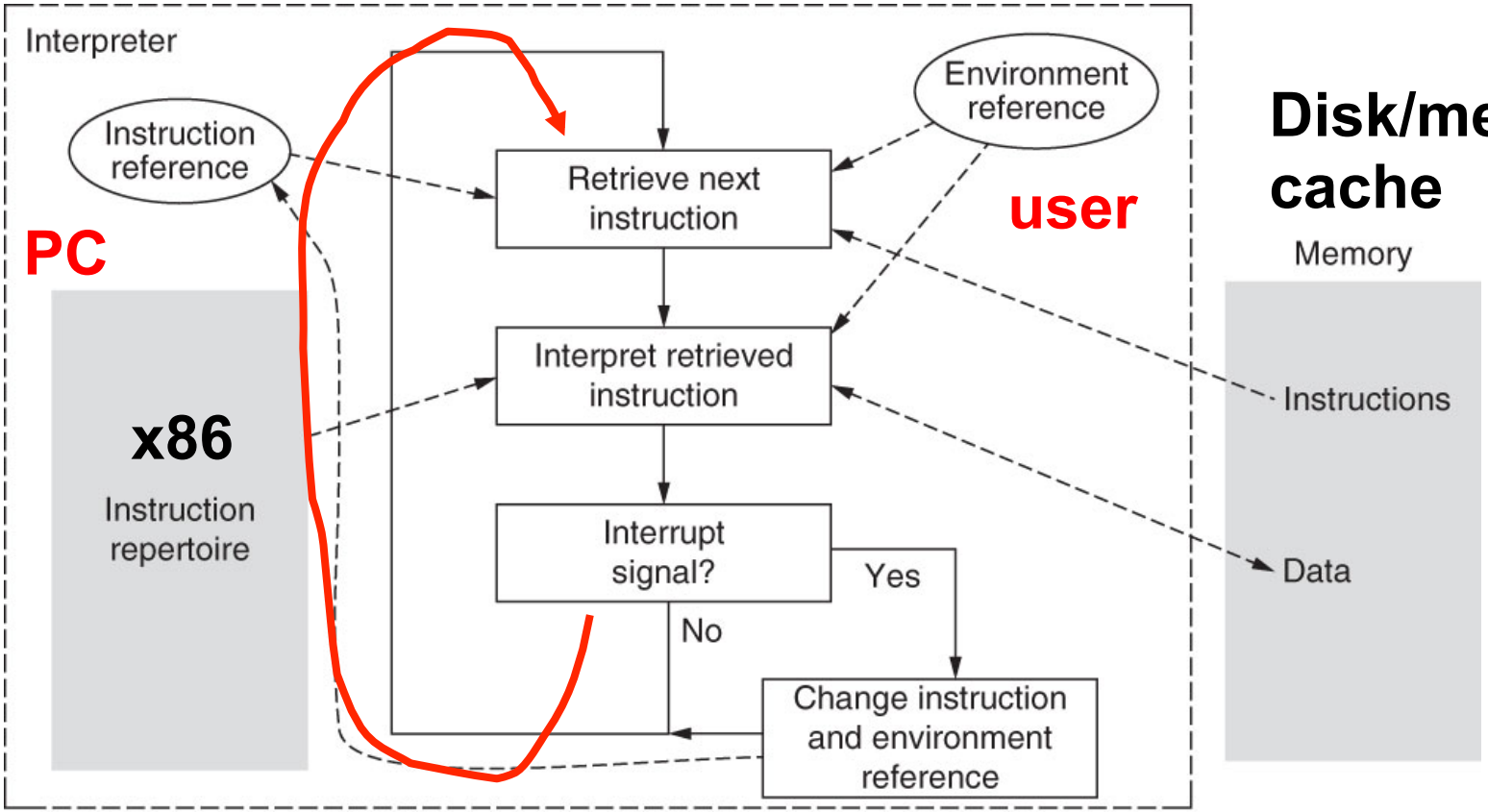

```
procedure INTERPRET()
    do forever
        instruction ← READ (instruction_reference)
        perform instruction in the context of environment_reference
        if interrupt_signal = TRUE then
            instruction_reference ← entry point of INTERRUPT_HANDLER
            environment_reference ← environment ref of INTERRUPT_HANDLER
```

1. Fetch

2. Decode, Exec, commit

3. Next PC

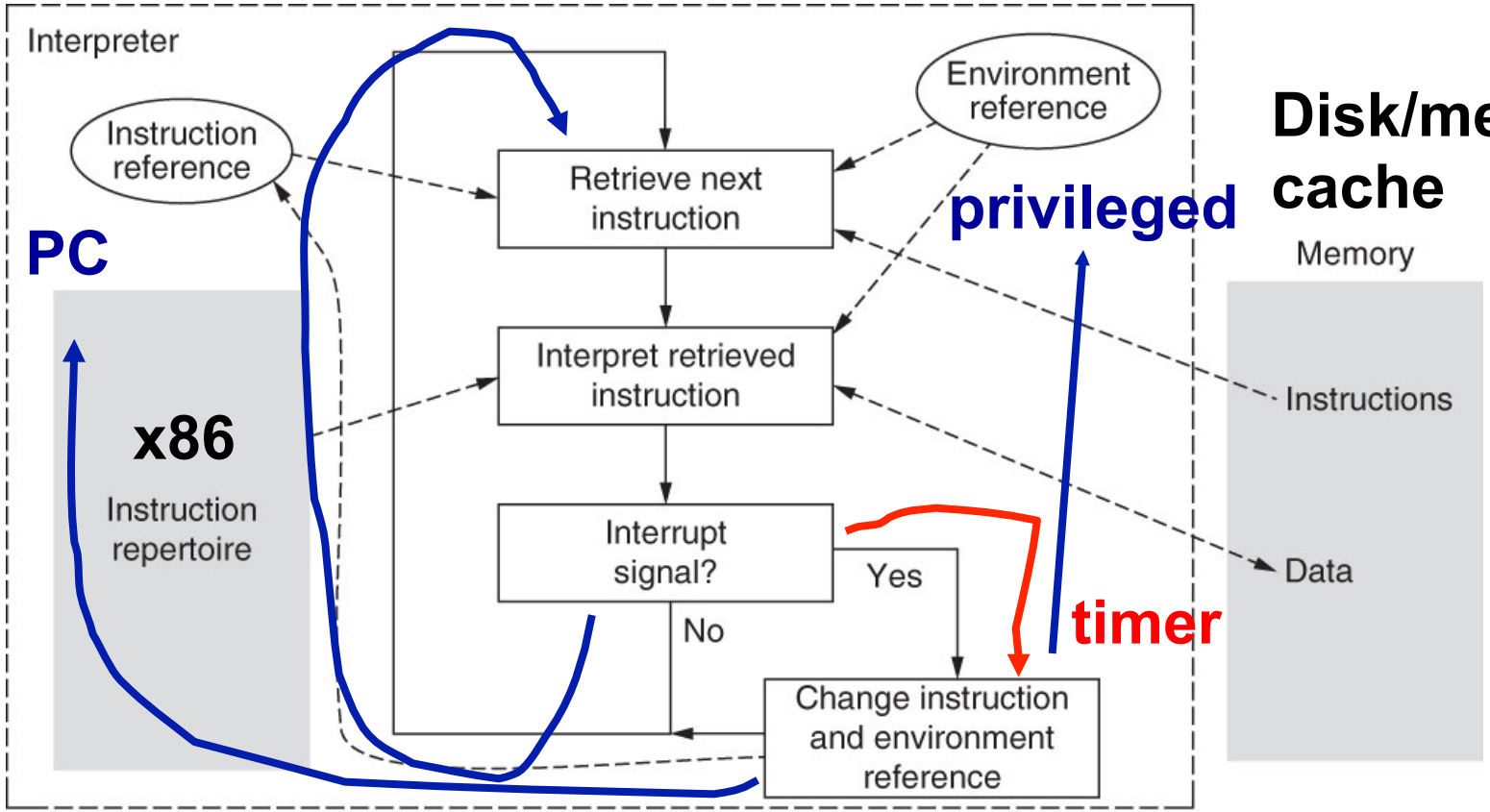
Typical interpreter flow



10ms of Powerpoint – millions of instructions

```
procedure INTERPRET()
    do forever
        instruction ← READ (instruction_reference)
        perform instruction in the context of environment_reference
        if interrupt_signal = TRUE then
            instruction_reference ← entry point of INTERRUPT_HANDLER
            environment_reference ← environment ref of INTERRUPT_HANDLER
```

Interrupt, environment change



Operating system Interrupt handler

```
procedure INTERPRET()
    do forever
        instruction ← READ (instruction_reference)
        perform instruction in the context of environment_reference
        if interrupt_signal = TRUE then
            instruction_reference ← entry point of INTERRUPT_HANDLER
            environment_reference ← environment ref of INTERRUPT_HANDLER
```

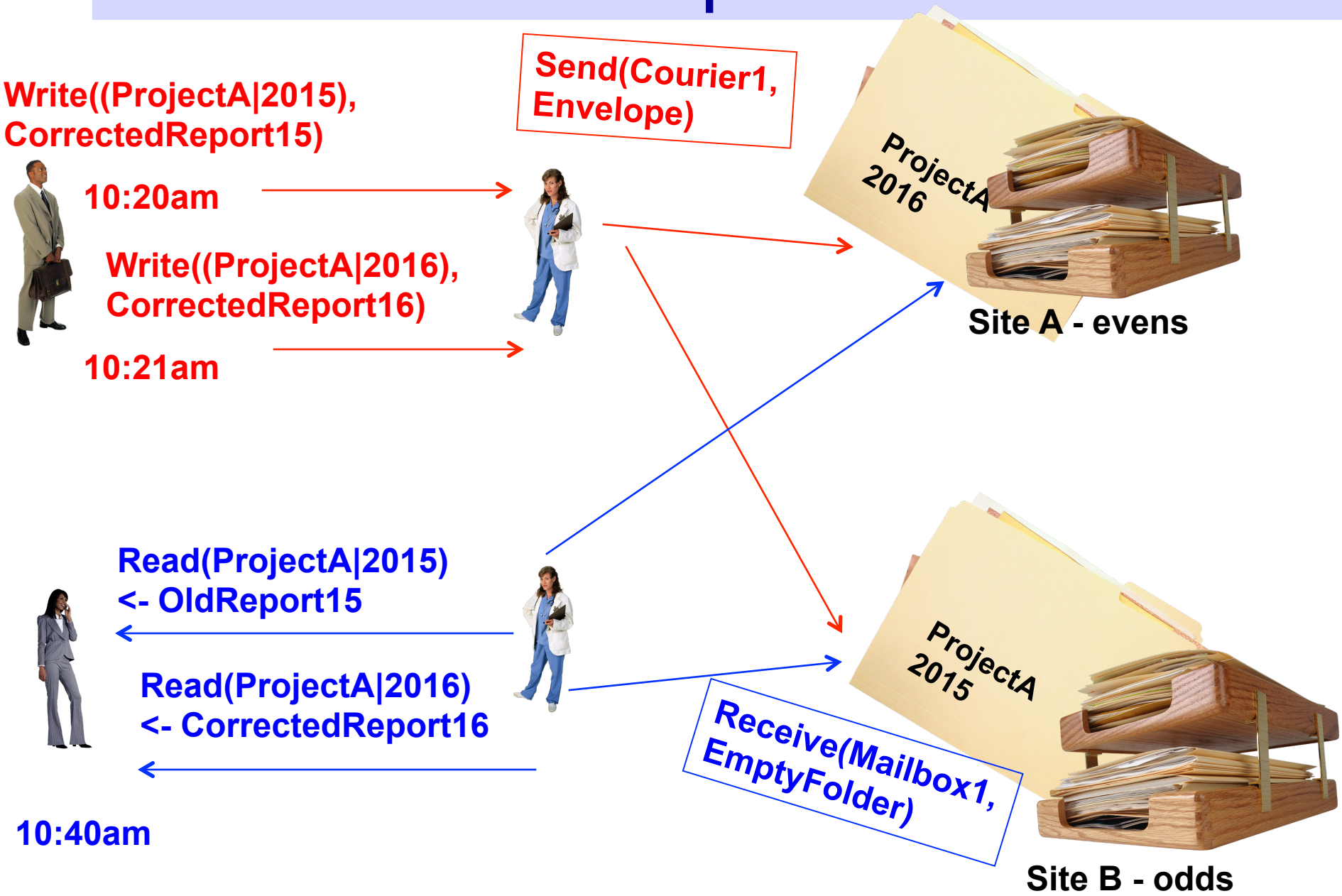
Communication Links

- Communication link provides a way for information to move between physically separate components
  - Physical: twisted pair, optical fiber
  - Higher-level: Ethernet, USB, Internet, UNIX pipe
- Abstraction:
  - SEND (link_name, out_message_buffer)
  - RECEIVE (link_name, in_message_buffer)

Examples

- Link_name: a network socket
- Message_buffer: memory block
- Different from memory READ and WRITE?
  - Message may not be delivered at all
  - Time to complete can be unpredictable
  - Faults, data integrity threats
  - Time and size of message delivery may not be known
- In some scenarios, useful to hide SEND/ RECEIVE behind a READ/WRITE abstraction
  - E.g. distributed shared memory parallel computers

Example


Write((ProjectA|2015), CorrectedReport15)
10:20am
Write((ProjectA|2016), CorrectedReport16)
10:21am
Send(Courier1, Envelope)
ProjectA 2016
Site A - evens
Read(ProjectA|2015) <- OldReport15
Read(ProjectA|2016) <- CorrectedReport16
10:40am
Receive(Mailbox1, EmptyFolder)
ProjectA 2015
Site B - odds

Reading

- Textbook, remaining of chapter 2