EEL-4736/5737 Principles of Computer System Design

Lecture Slides 4

Textbook Chapter 2

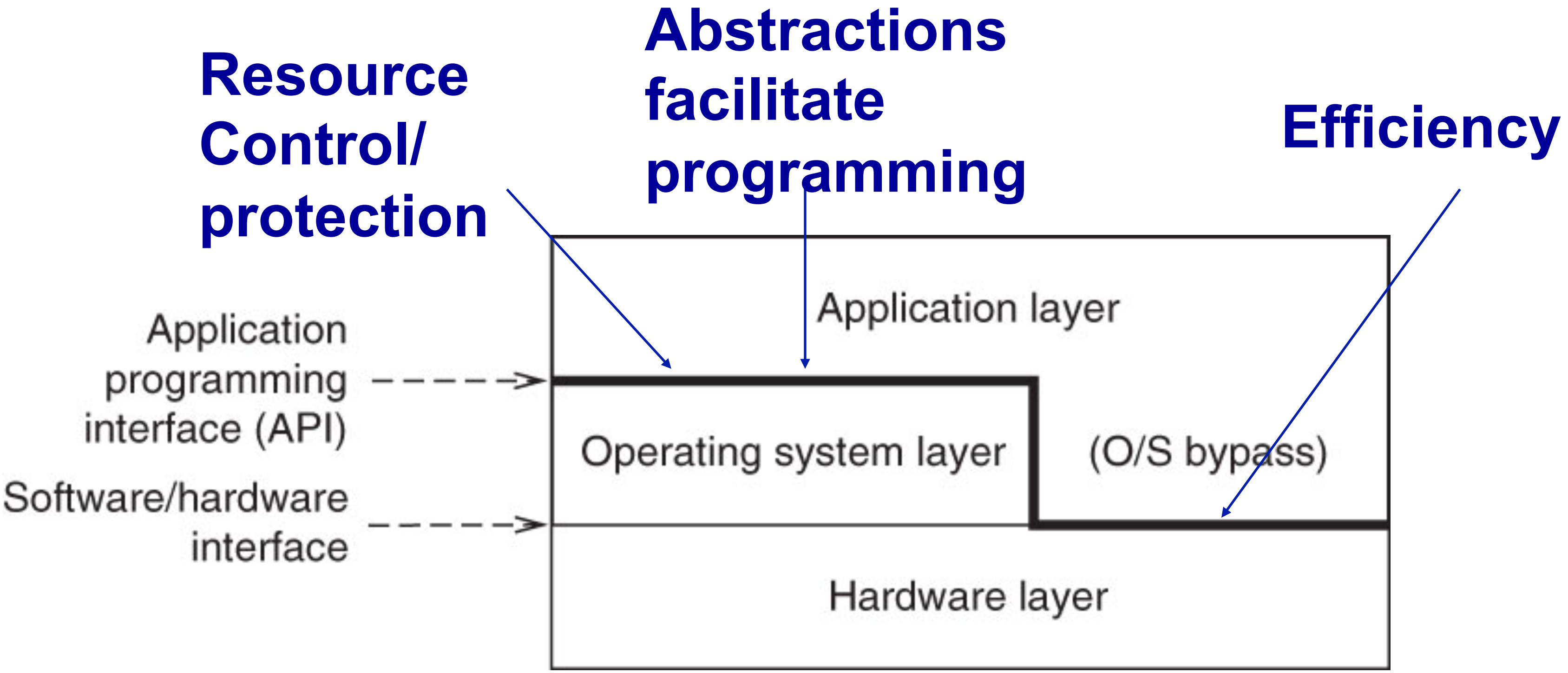
Organizing Computer Systems with Names and Layers

Introduction

- Example of a layered system: O/S
- What constitutes an operating system?
- Set of software (kernel, applications, libraries) that facilitates programmers and users to do their job
  - Provide convenient abstractions to facilitate programming
  - Handles interface with, allocation and scheduling of hardware devices
  - We'll study O/Ss in more depth from a system design perspective in chapter 5

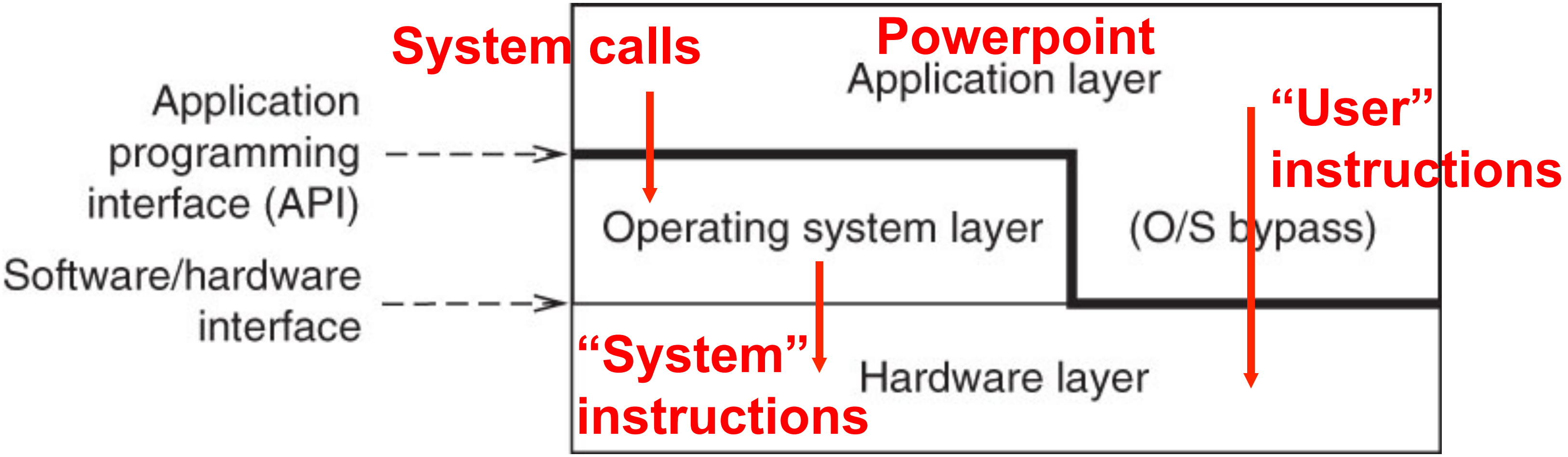
Operating system and layers

- Coarse-grain layers in typical computer
  - Several layers within each

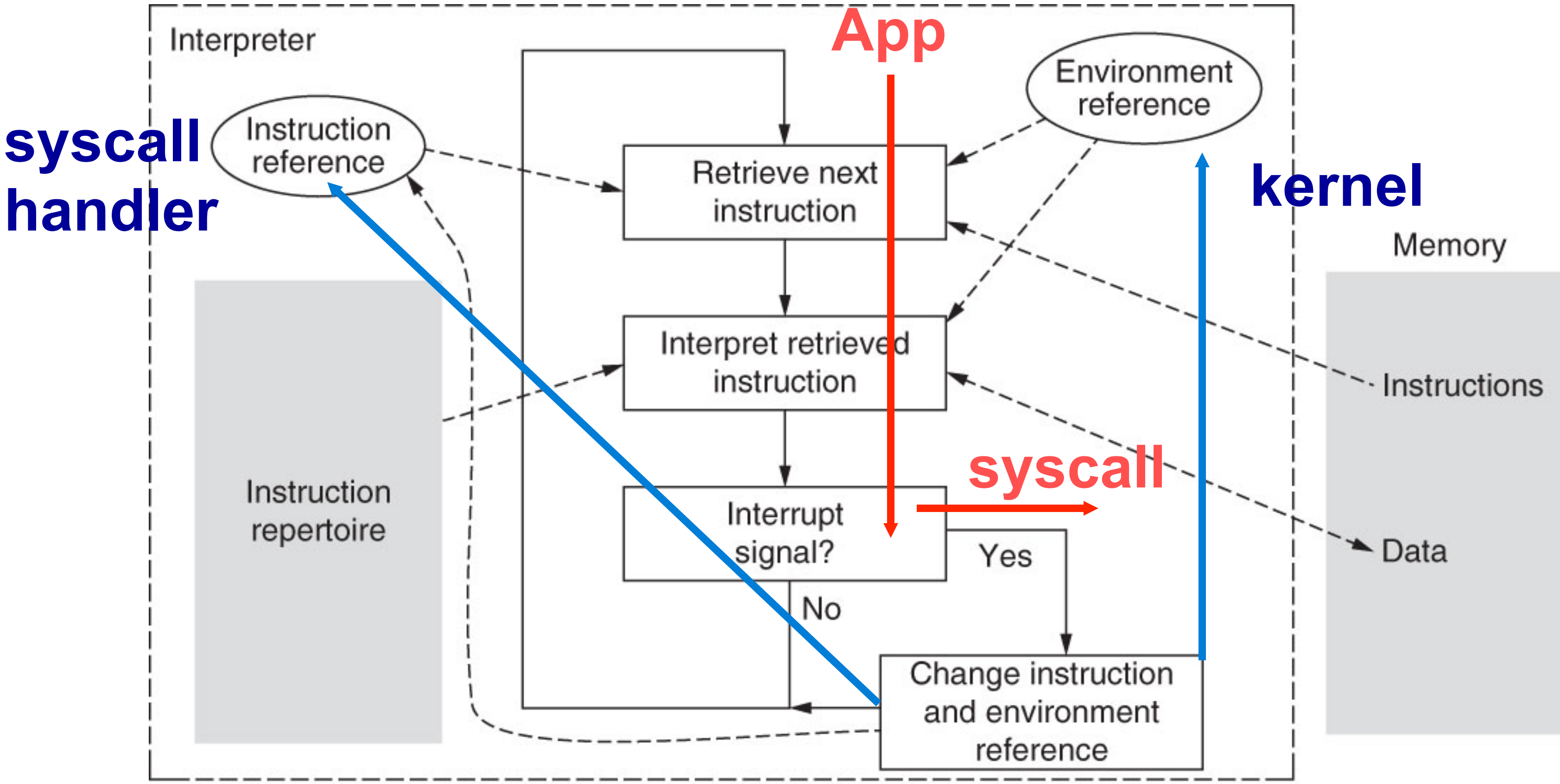


Operating system and layers

- Example



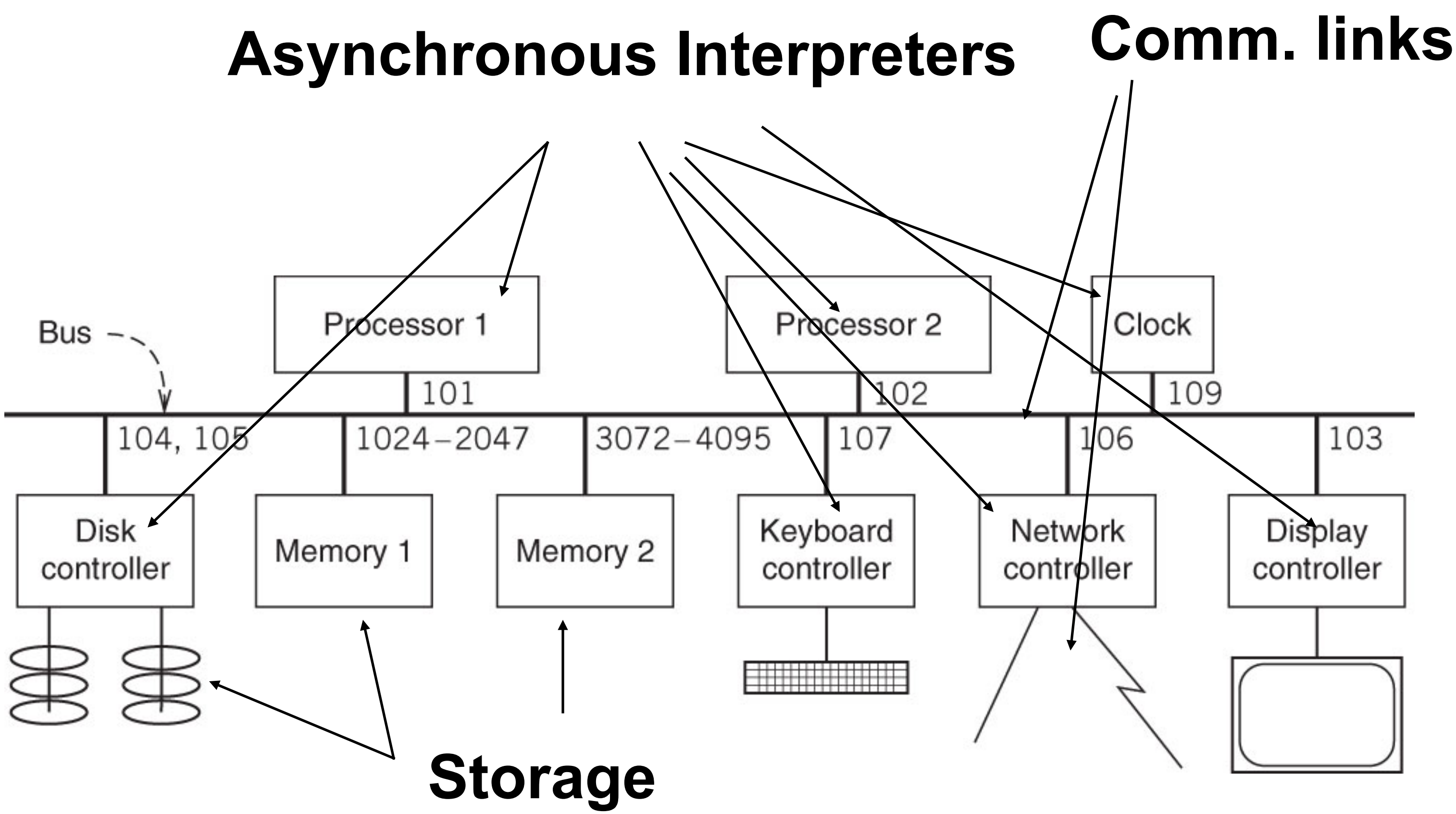
System call – synchronous intr




```
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
```

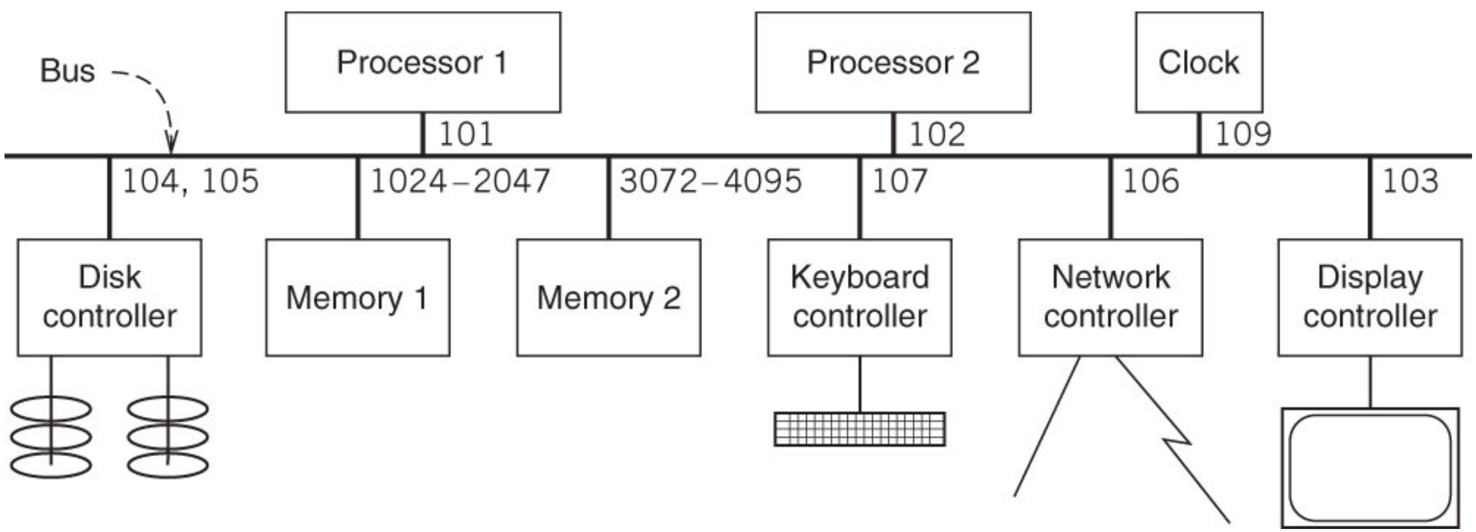
A hardware layer - Bus

- Implement low-level versions of the three fundamental abstractions



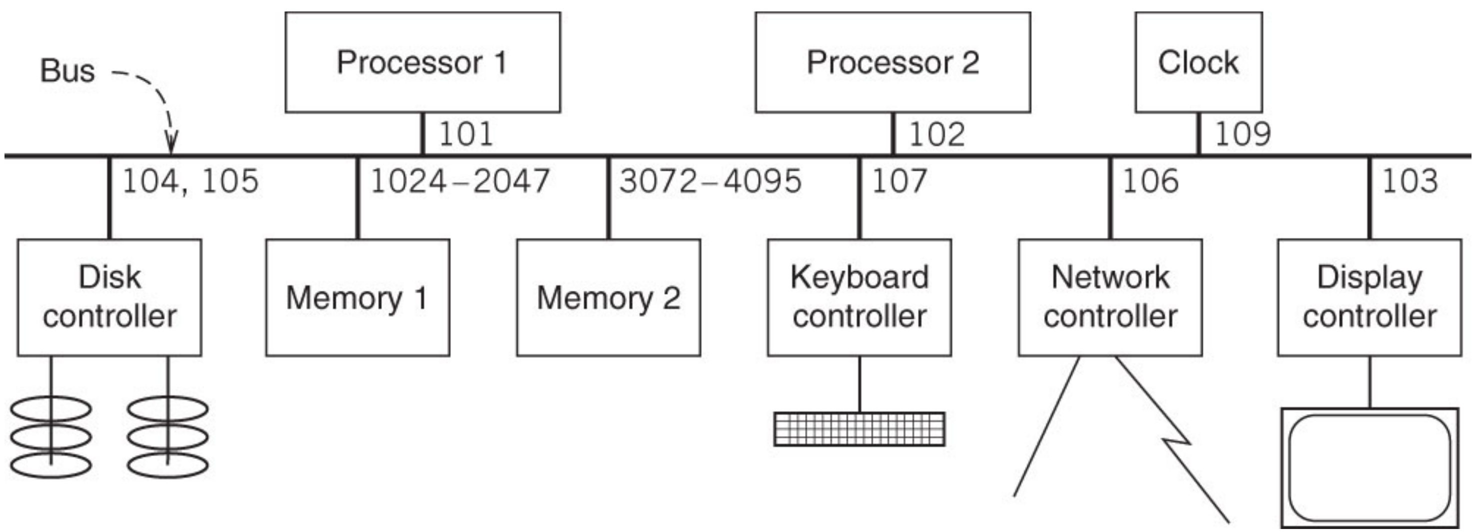
Examples

- Each device has an interface to the bus with a unique name (address)
- Broadcast communication link
  – SEND reaches all devices; RECEIVE sees all messages; no need to route



Examples

- Name space partitioning
  - N-bits: integer 0 to $2^n - 1$
    - In the example, memories use a range of address space

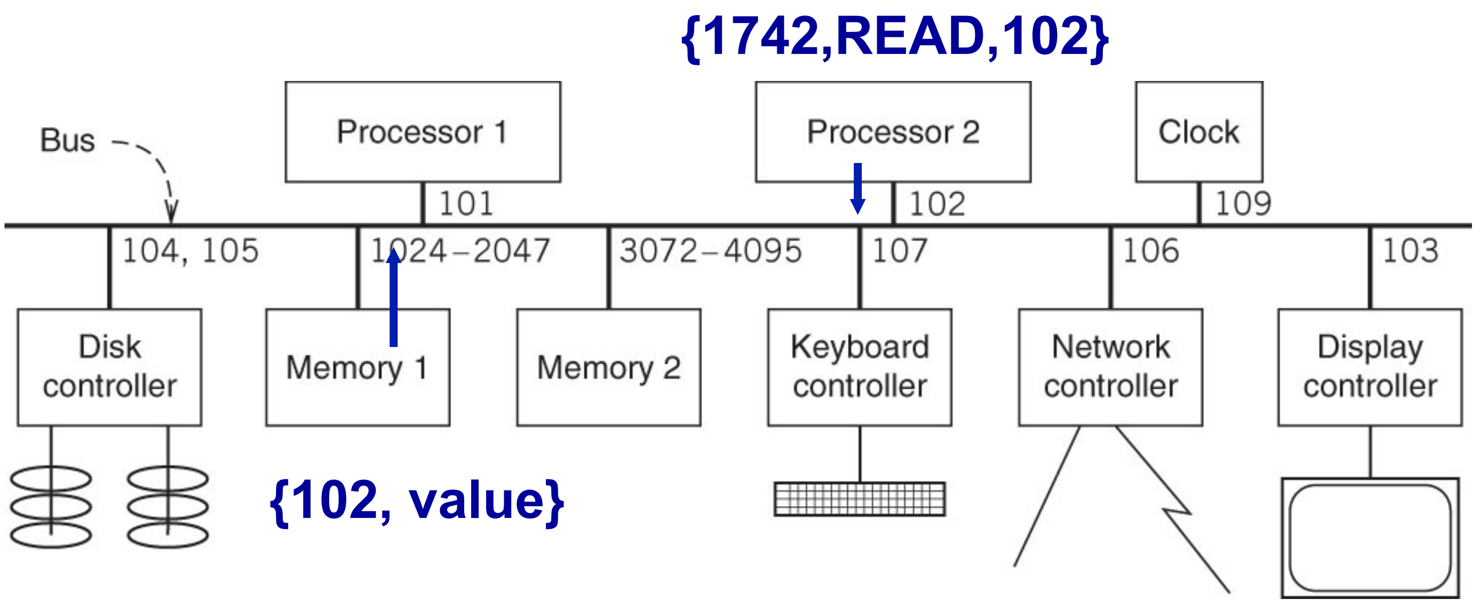


Pseudo-code convention

- total <- a + b
  - Assignment
- a(11..15)
  - Bits 11..15 from string a
- x => y: {M}
  - Message with contents M sent from x to y
  - {a,b,c} – message contains named fields marshaled in some way the recipient understands

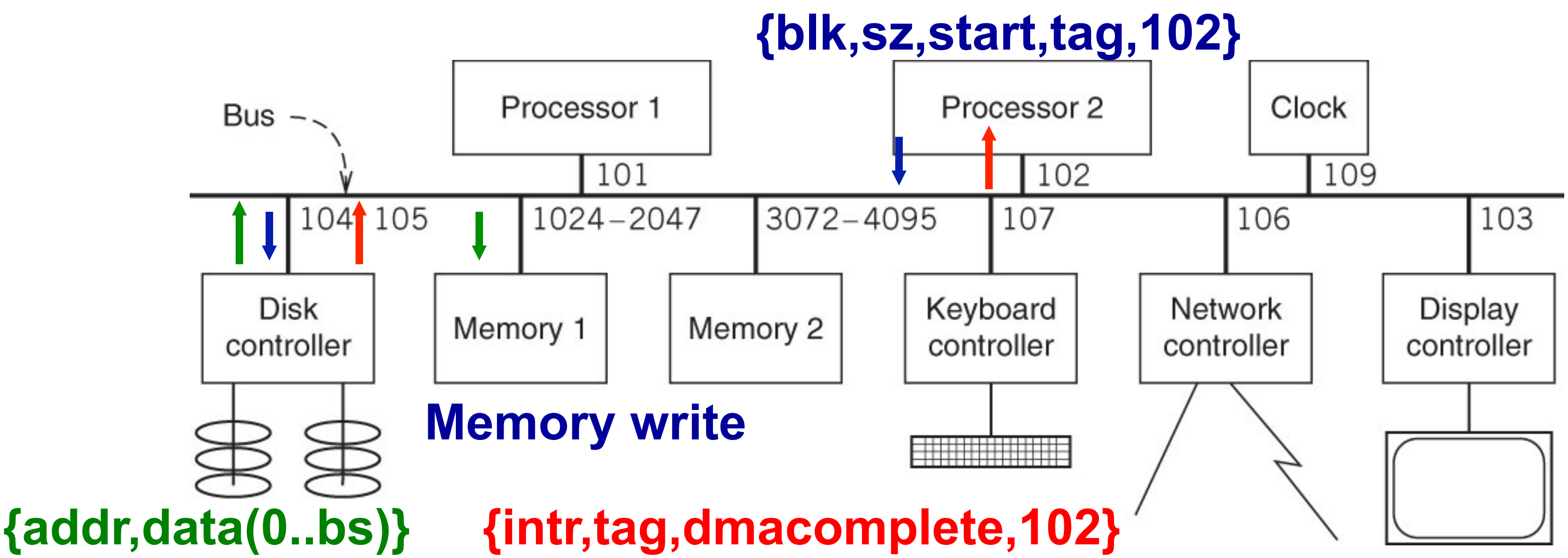
Examples

- Processor 2 interprets LOAD 1742, R1
  - P2 => all bus modules: {1742, READ, 102}
- Memory module 1
  - value <- READ(1742)   (internally)
  - M1 => all bus modules: {102, value}



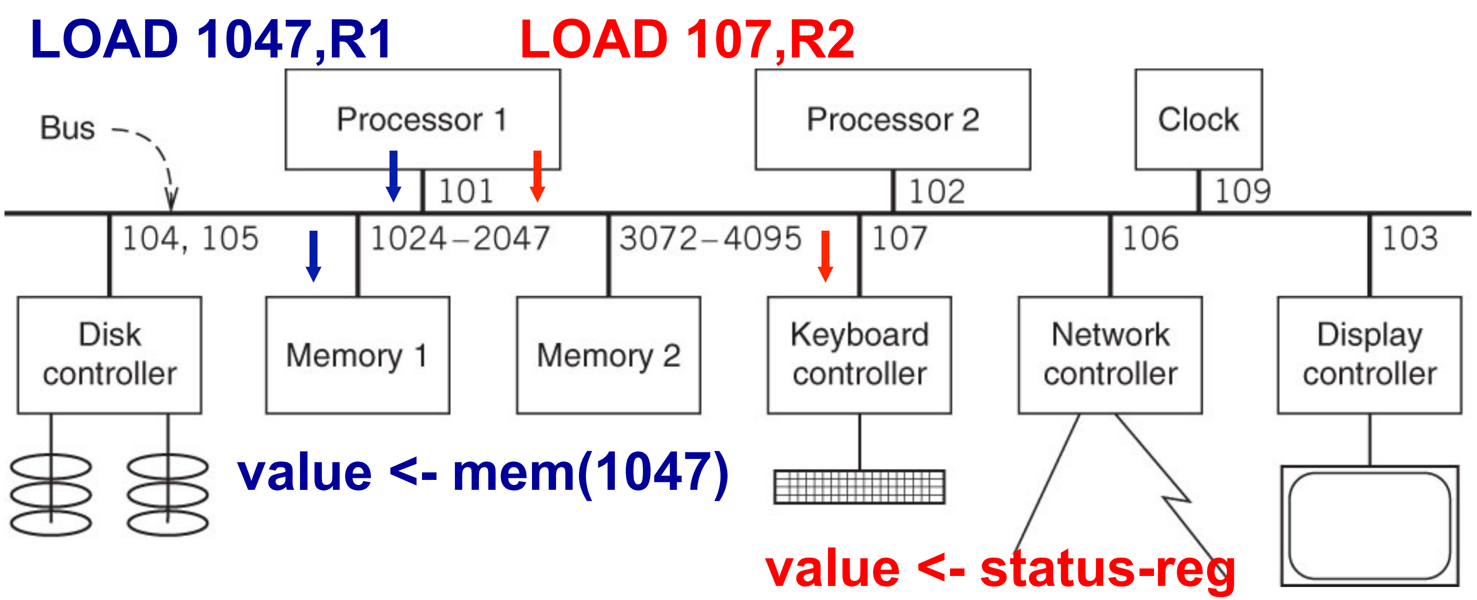
Examples

- Direct memory access – decoupled transfers
  - E.g. disk read: P2 sends a message to device 104:
    - Which block to read, and size
    - Reference to starting memory address
  - Disk controller reads block from disk
    - Sends *data* message directly to memory module
    - Sends *interrupt* message to P2



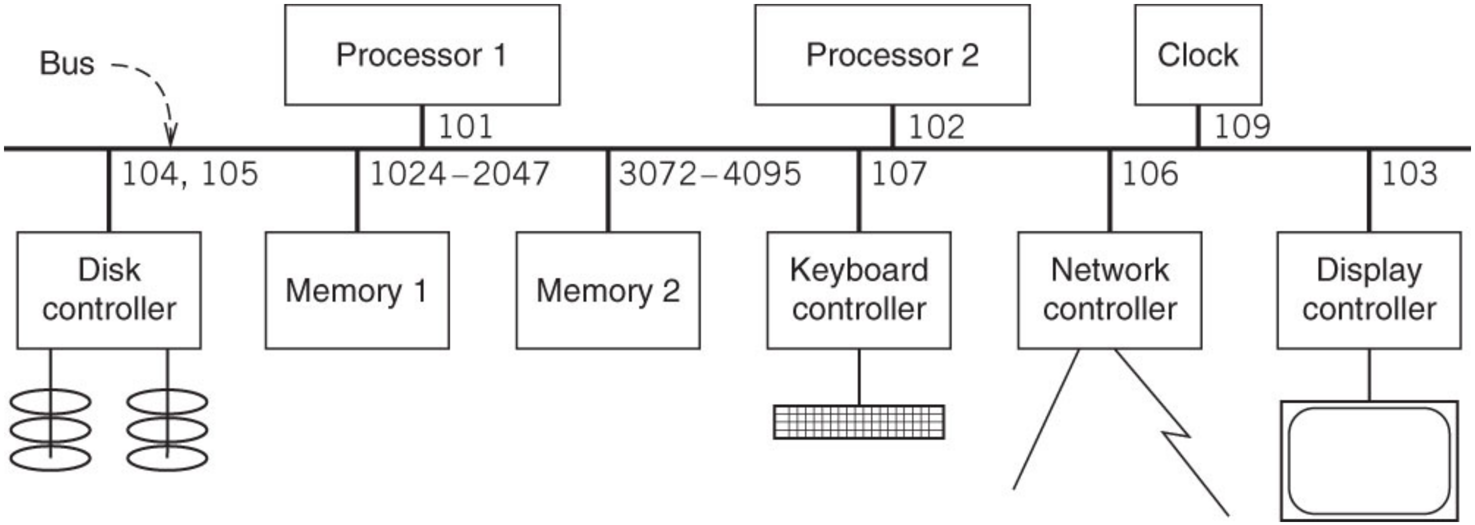
Memory-mapped I/O

- Memory-mapped I/O
  - Processor-issued loads/stores – messages on the bus
    - Subset of addresses bound to devices
    - E.g. DMA controller exposes addresses for device control registers



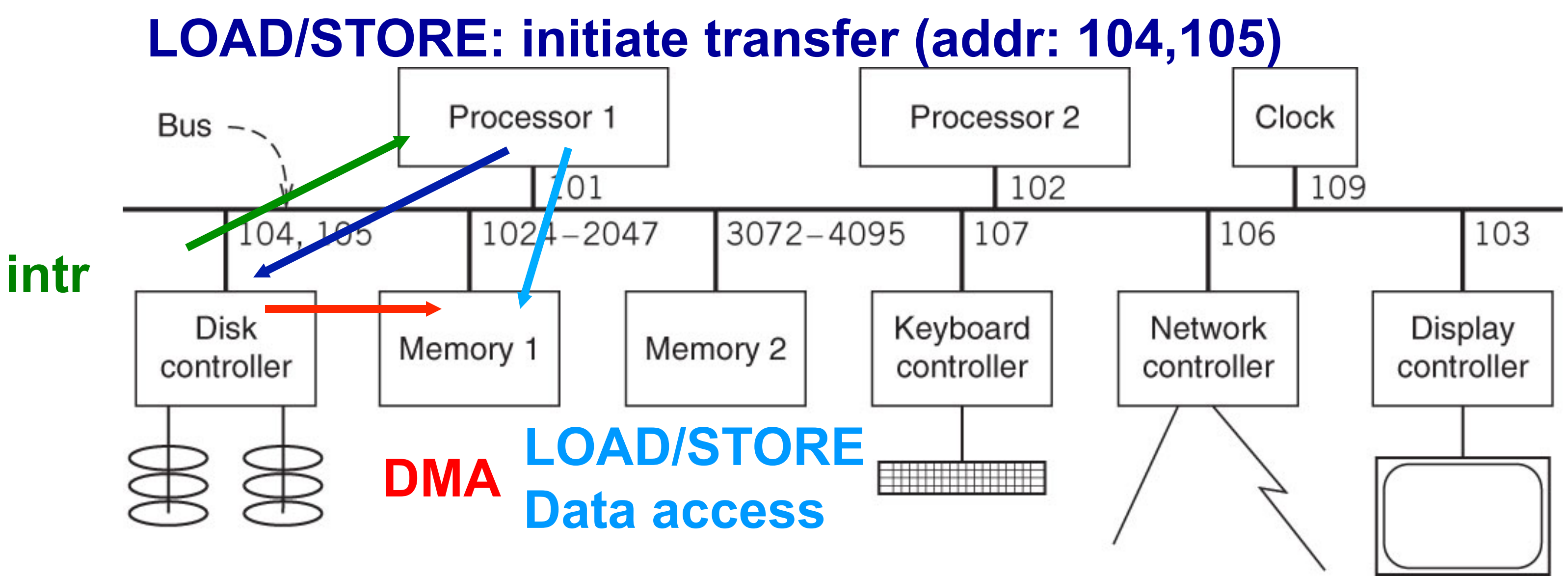
Addressing devices

- Disk – storage
  - Should disk storage be exposed as memory-mapped I/O?
- Remote main memory

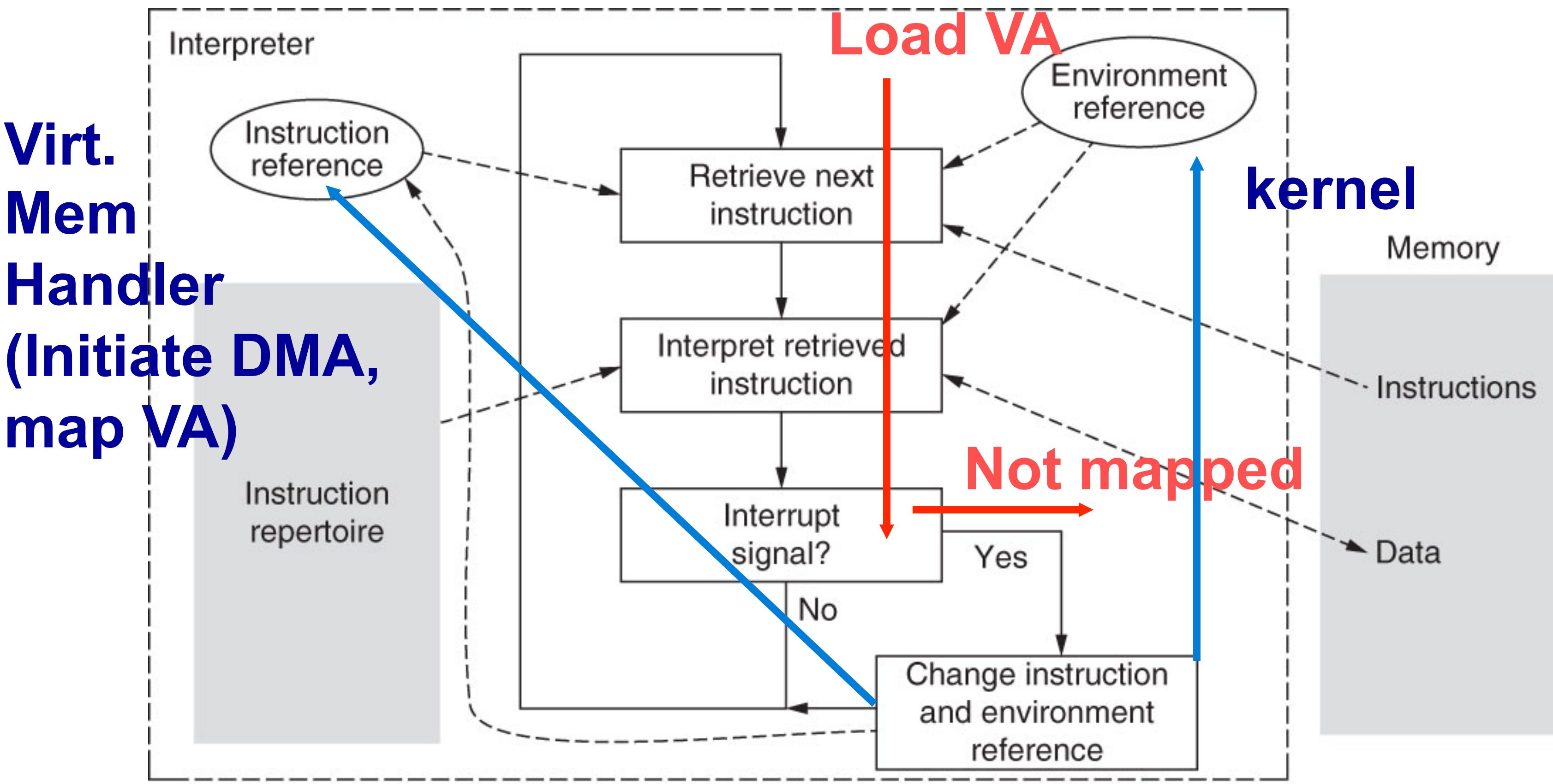


Virtual memory

- A widely-used approach - virtual memory
  - Memory-mapped I/O used to program devices – disk controllers, DMA
  - Transfers at a large granularity (blocks, pages vs. words – efficient)

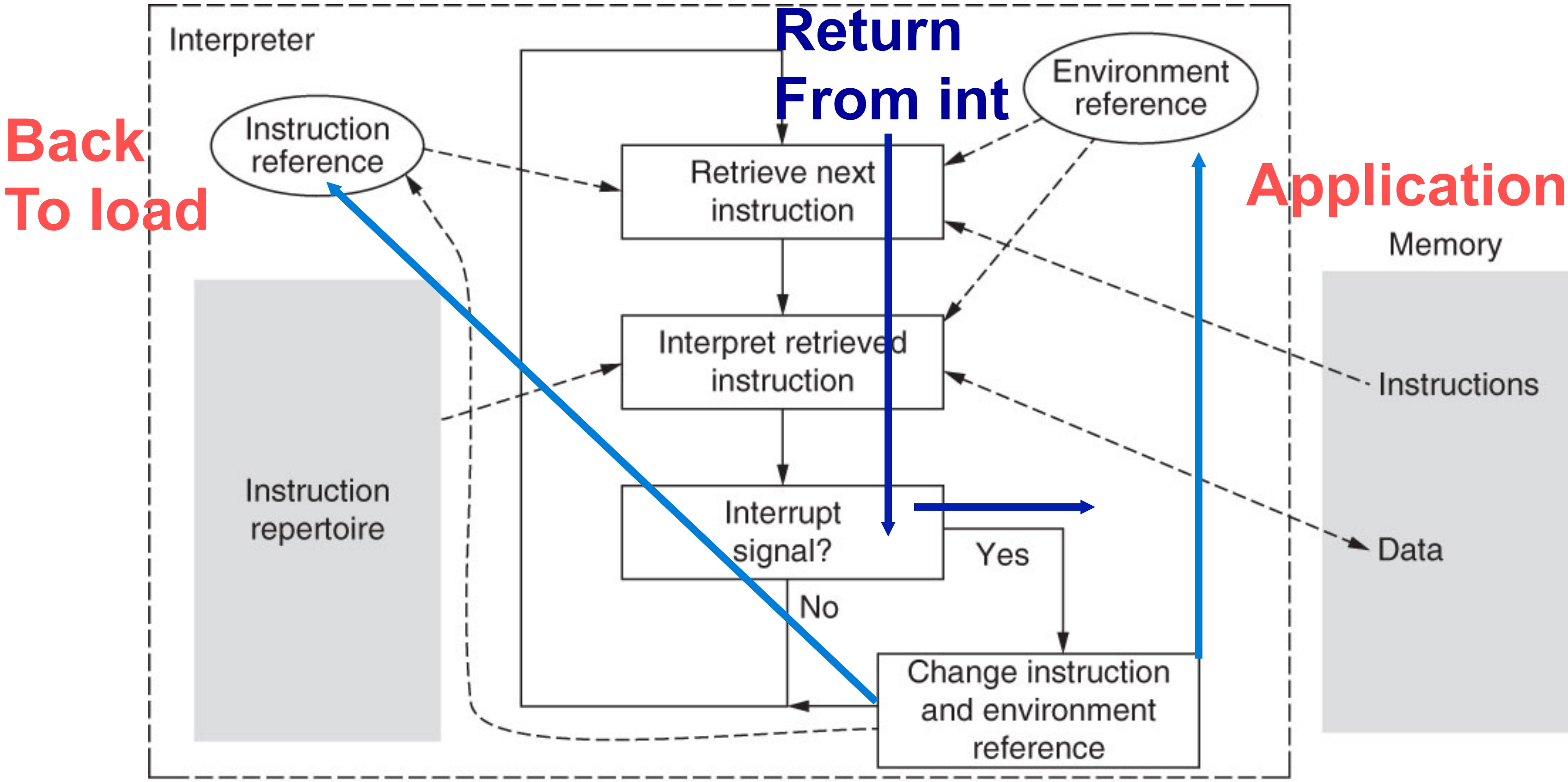


Virtual memory and interpreter



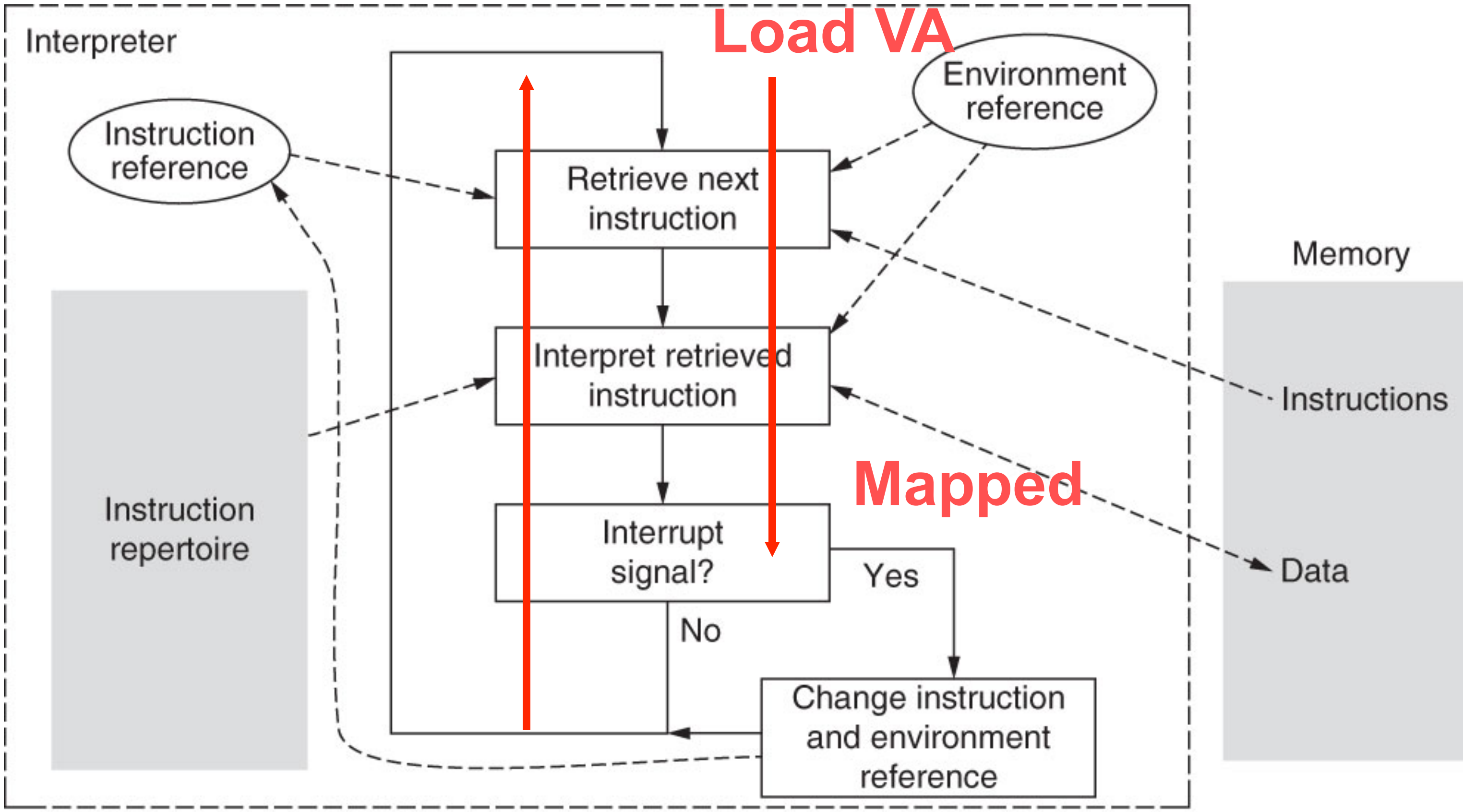

```
procedure INTERPRET()
    do forever
        instruction ← READ (instruction_reference)
        perform instruction in the context of environment_reference
        if interrupt_signal = TRUE then
            instruction_reference ← entry point of INTERRUPT_HANDLER
            environment_reference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

Virtual memory and interpreter




```
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
```

Virtual memory and interpreter




```
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
```

A software layer: File abstraction

- High-level version of memory abstraction
  - A file holds an array of bytes
  - It is durable
    - How long depends on many factors
  - It has a name
    - Can refer to it later;
    - Can be used to share information
- The abstraction can be used in ways other than storing data with durability
  - E.g. a device as a file

File abstraction – core API

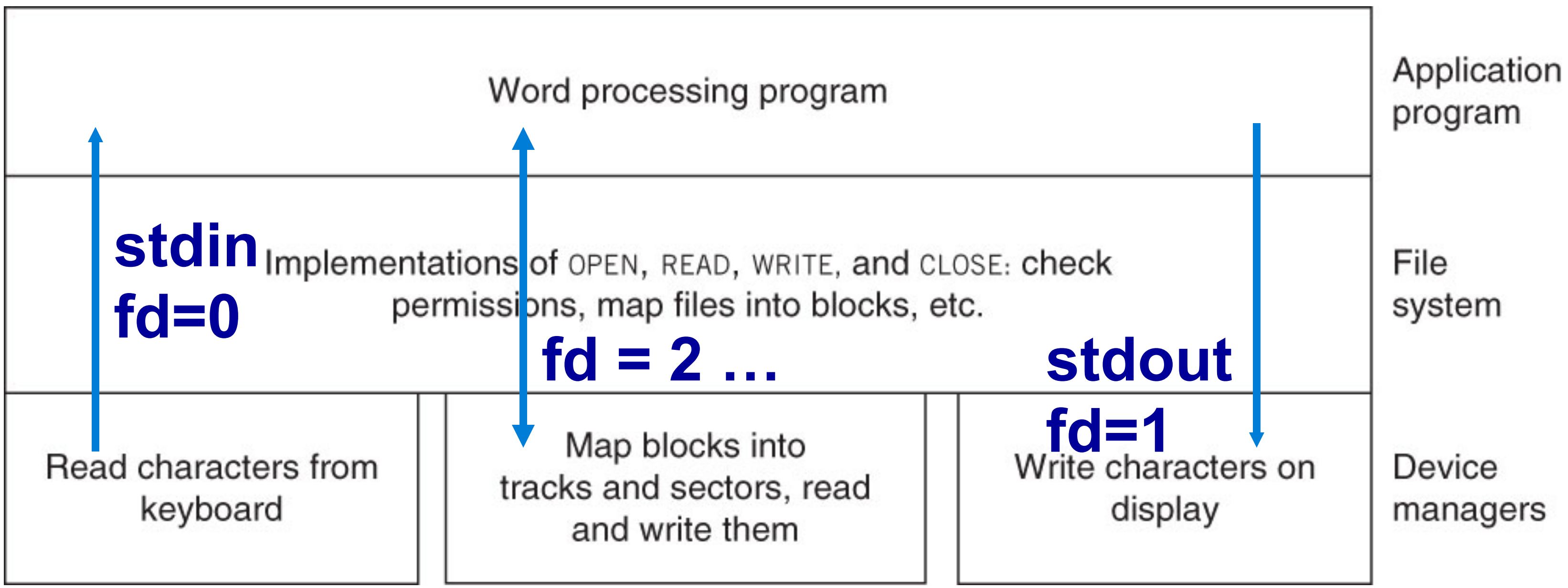
- OPEN
  - Translate file name to name within a local name space to be used by subsequent operations
    - E.g. an integer file descriptor
- READ, WRITE
  - Retrieve/store data from/to an open file
- CLOSE
  - Unbinds file name from descriptor; may also store data on stable storage
- Why OPEN and CLOSE?

File abstraction - APIs

- Need for OPEN/CLOSE
  - Efficiency
    - Replace (long) filename with short integer
    - Saves space – system call stack
    - Saves time – name resolution, permission check
  - Atomicity and concurrency control
    - Can determine which objects share a file at a given point in time
      - E.g. you can block an OPEN if file is already opened for before-after atomicity
    - Can be used to deal with faults
      - Crash before a CLOSE?

File abstraction uses

- E.g. UNIX standard in/standard out



Reading

- Section 2.5