KOÇ UNIVERSITY

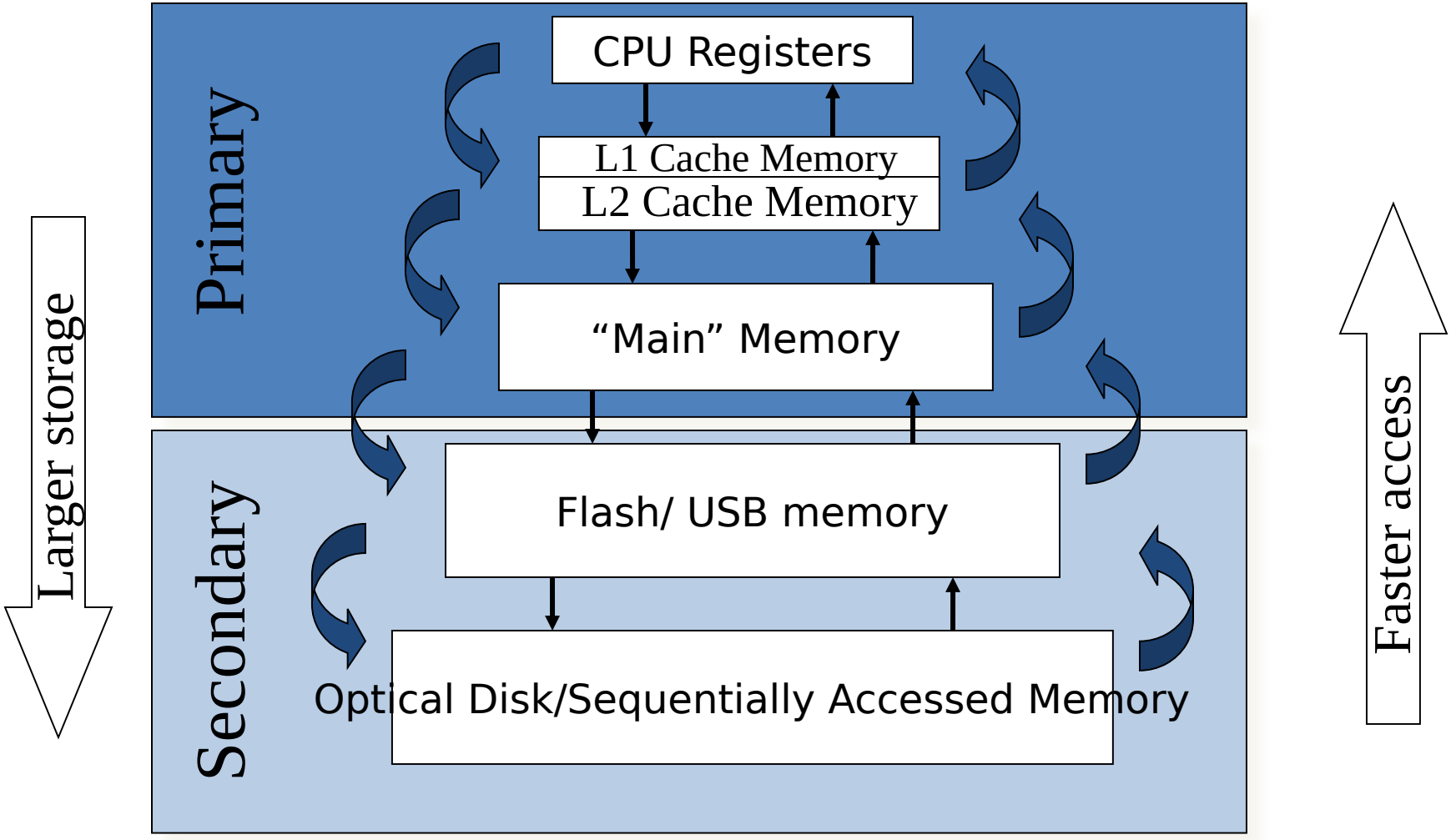
Final Exam Review Slides

(Lecture 17 onward only
Refer to Midterm Review slides for previous lectures)

Didem Unat

COMP304 - Operating Systems (OS)

Contemporary Memory Hierarchy



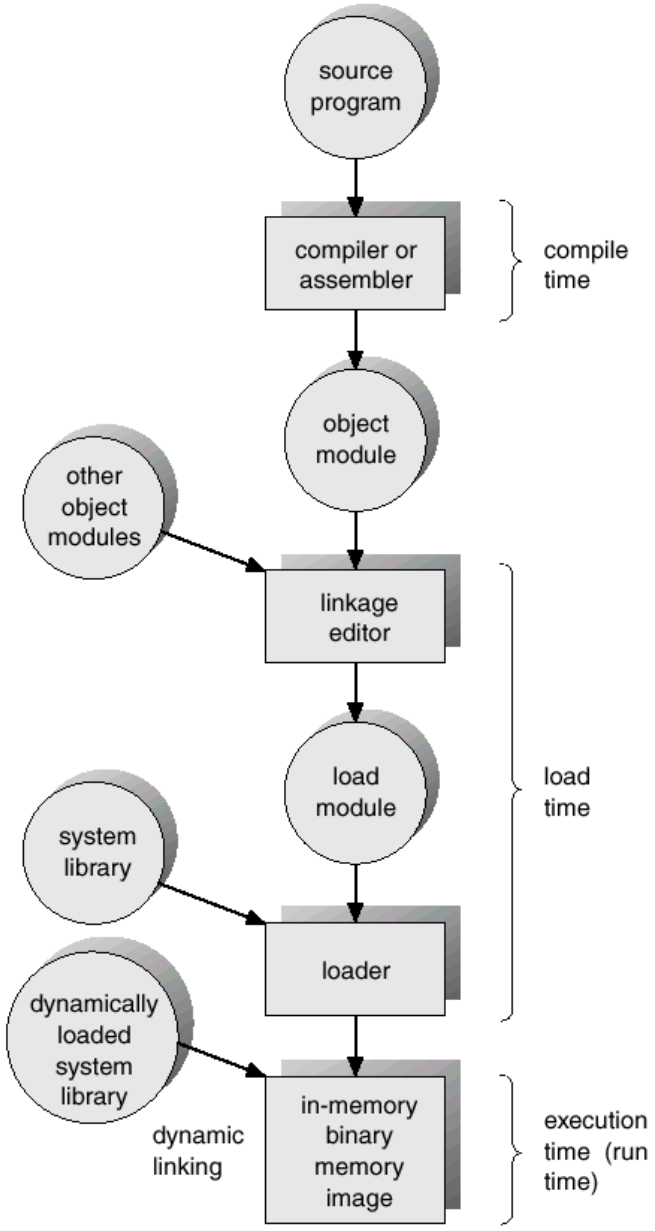
Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a stall
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

How do programs specify memory addresses?

- Absolute code
  - If you know where the program gets loaded (any relocation is done at link time)
- Position independent code
  - All addresses are relative
- Dynamically relocatable code
  - Relocated at load time
- Or ... use logical addresses
  - Absolute code with addresses translated at run time
  - Need special memory translation hardware

Address Binding



Address binding of instructions and data to memory addresses can
happen at three different stages.

Compile time:

If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes.

Load time:

Must generate **relocatable** code if memory location is not known at compile time.

Execution time:

Binding delayed until run time if the process can be moved during its execution from one memory segment to another.

Need hardware support for address maps (e.g., **base** and **limit**

Logical vs. Physical Address Space

- For proper memory management
  - There is **logical address space and physical address space**
  - **Logical address** – generated by the CPU; also referred to as **virtual address**.
  - **Physical address** – address seen by the memory unit.

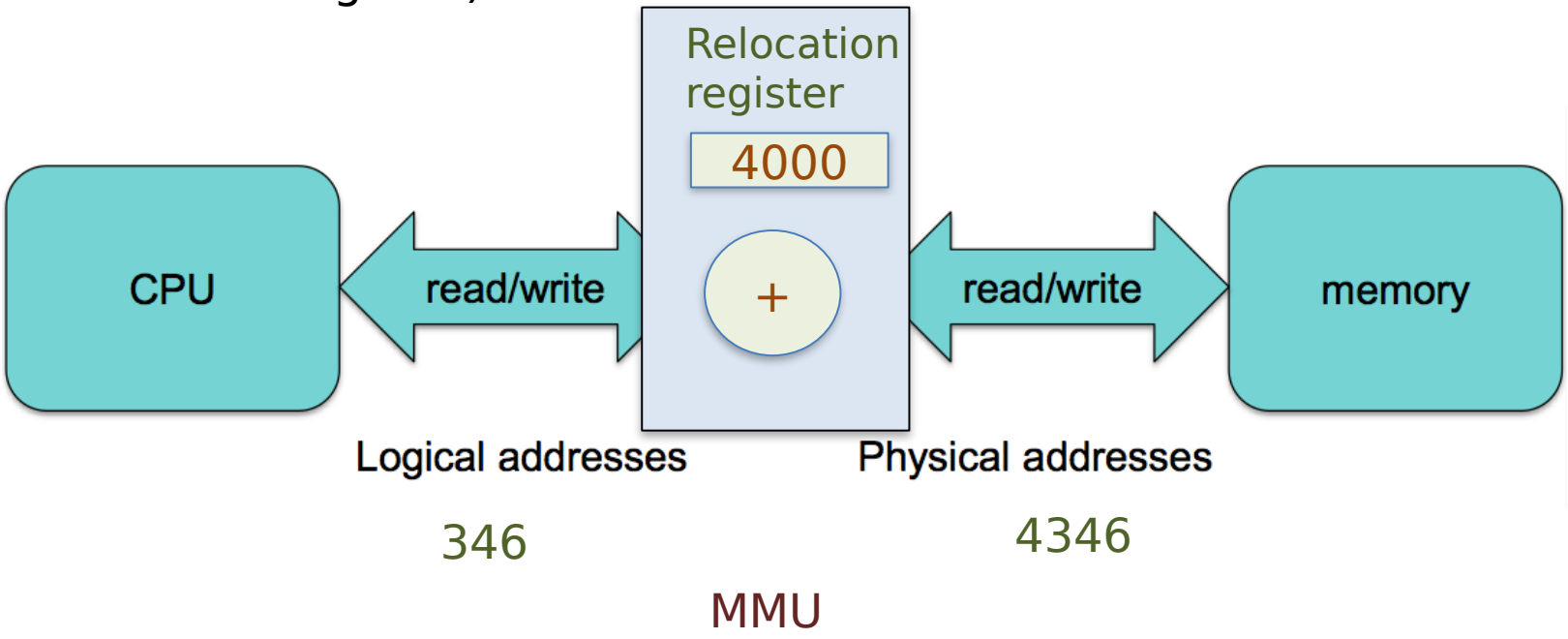
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;

- Logical (virtual) and physical addresses differ in execution-time address-binding scheme.

- The user program deals with logical addresses; it never sees the real physical addresses.
  - Why?

Relocation Register

- User program only generates logical addresses and thinks that the process runs in location 0 to max.
- In fact, it runs R+0 to R+max - R is the base register (now called relocation register)

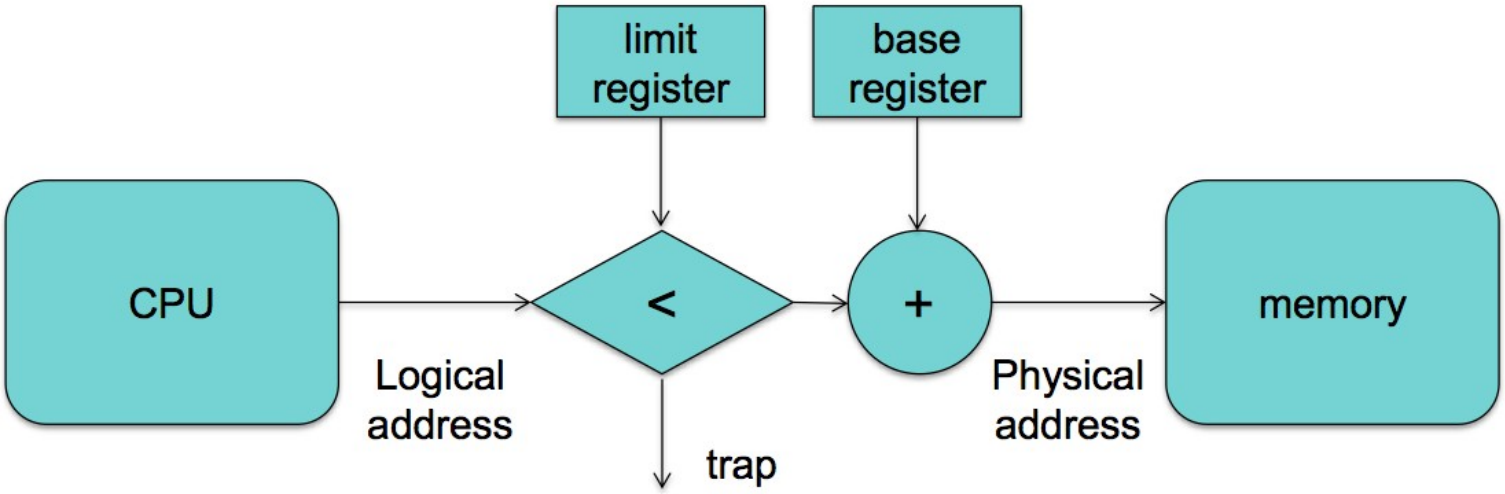


Relocatable Addressing

Base & limit

– *Physical address = logical address + base register*

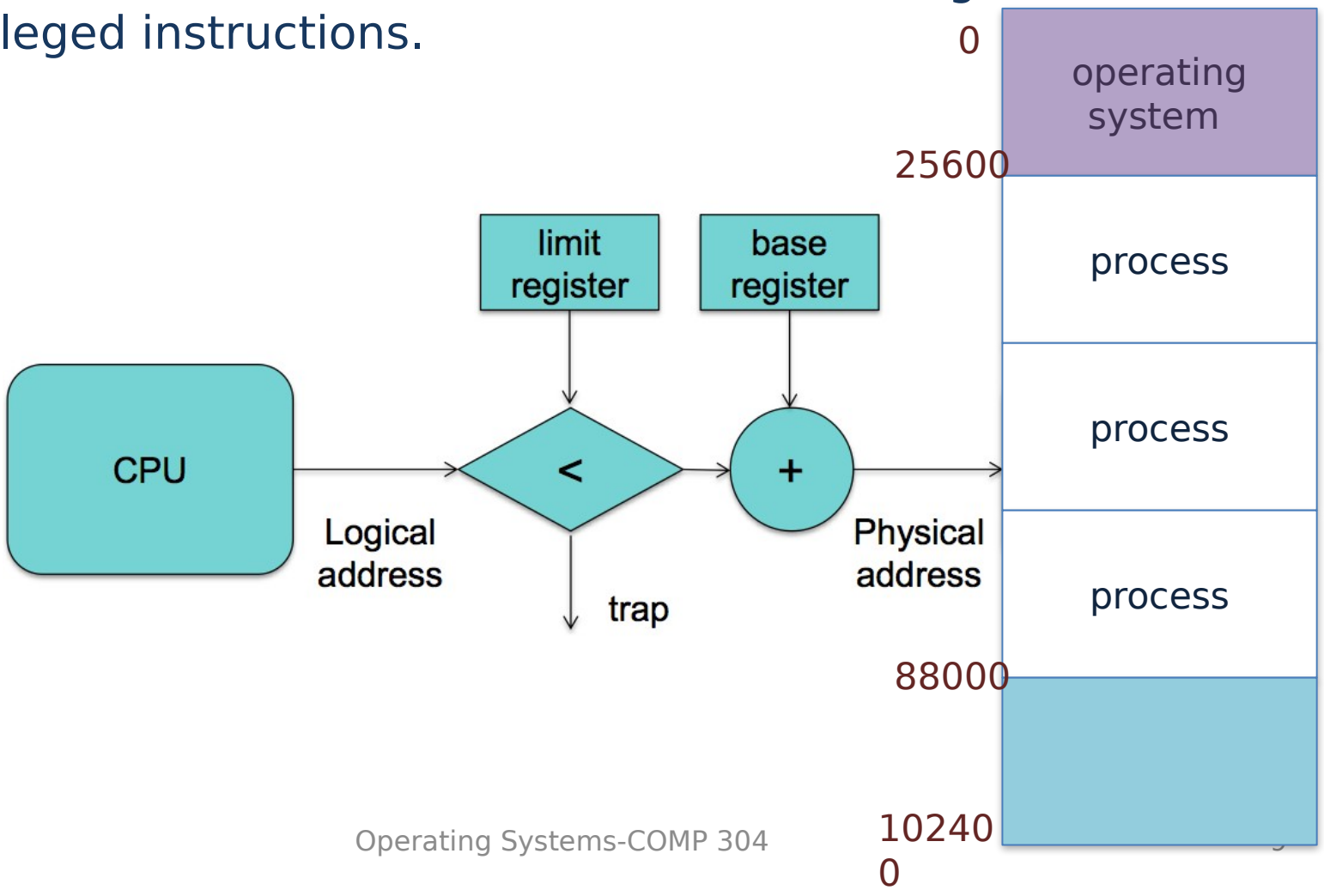
– But first check that: *logical address < limit*



Memory management unit (MMU) maps the logical address dynamically by adding the value in the relocation register.

Hardware Protection

- When executing in kernel mode, the operating system has unrestricted access to both kernel and user's memory.
- The load instructions for the *base* and *limit* registers are privileged instructions.



Memory Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Three methods:
  - Contiguous memory allocation
  - Segmentation
  - Paging

Contiguous Allocation

- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base (relocation) register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU (memory management unit) maps logical address dynamically

Dynamic Storage-Allocation Problem

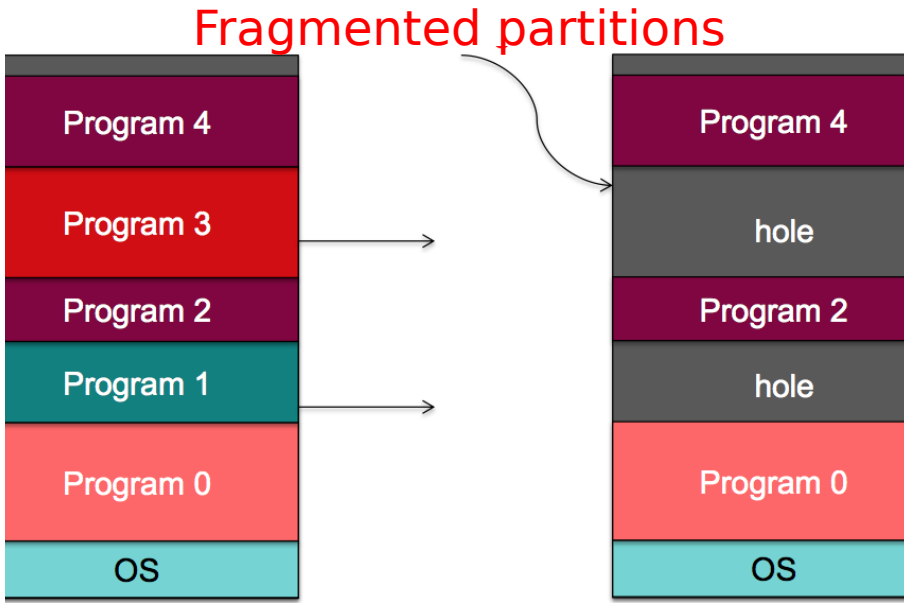
w to satisfy a request of size n from a list of free hol

- **First-fit**: Allocate the **first** hole that is big enough.
- **Best-fit**: Allocate the **smallest** hole that is big enough; must search entire list, unless ordered by size.
  - Produces the smallest leftover hole.
- **Worst-fit**: Allocate the **largest** hole; must a

First-fit and best-fit perform better than worst-fit in terms of speed and storage utilization, however it causes fragmentation.

Fragmentation

- **External Fragmentation**
  - total memory space exists to satisfy a request, but it is not contiguous
  - Also a common problem in disk as well



- **Internal Fragmentation**
  - Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Need more memory?

- What if a process needs more memory?
  – Always allocate some extra memory just in case
    – Find a hole big enough to relocate the process

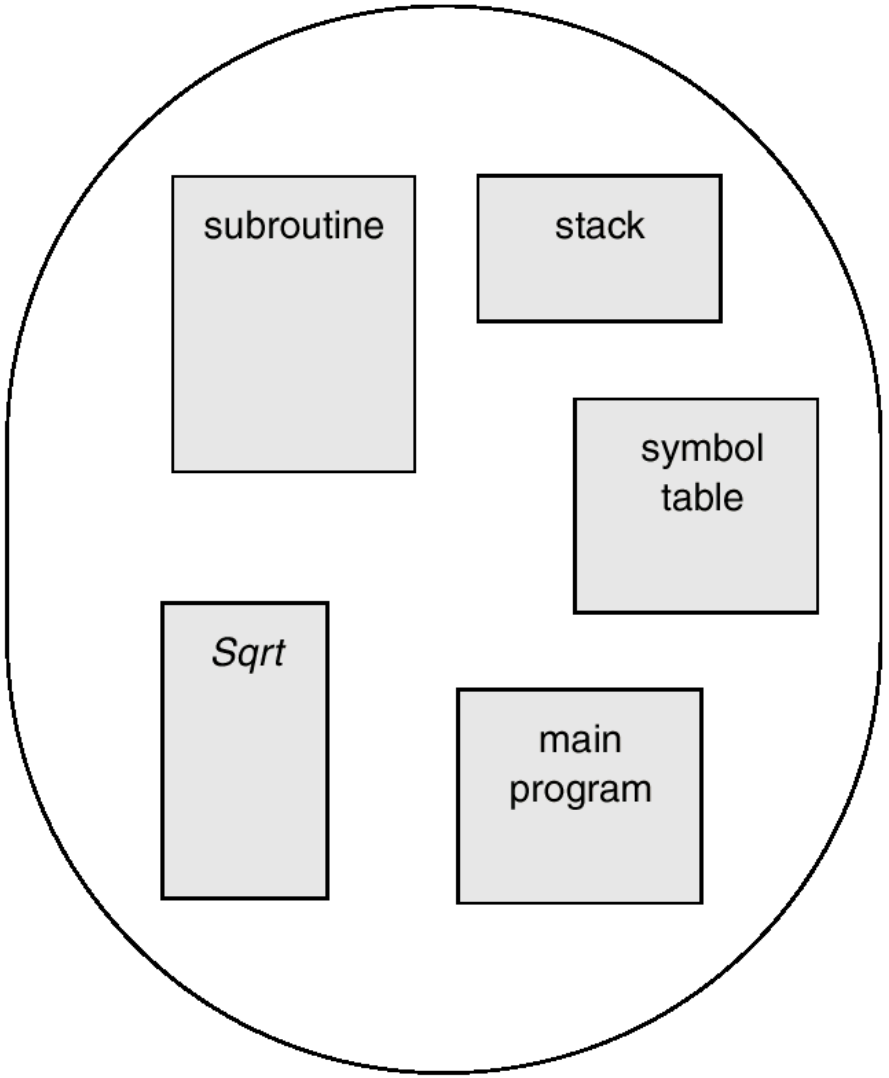
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible **only** if relocation is dynamic, and is done at execution time.

Segmentation

- Memory allocation mechanism that supports user view of memory.
- Users prefer to view memory as a collection of variable-sized segments – similar to programmer's view of memory
- A program is a collection of segments. A segment is a logical unit such as:

  main program,
  function,
  method,
  object,
  local variables, global variables,

User's View of a Program



logical address space

Logical address space is a collection of segments

Segmentation Architecture

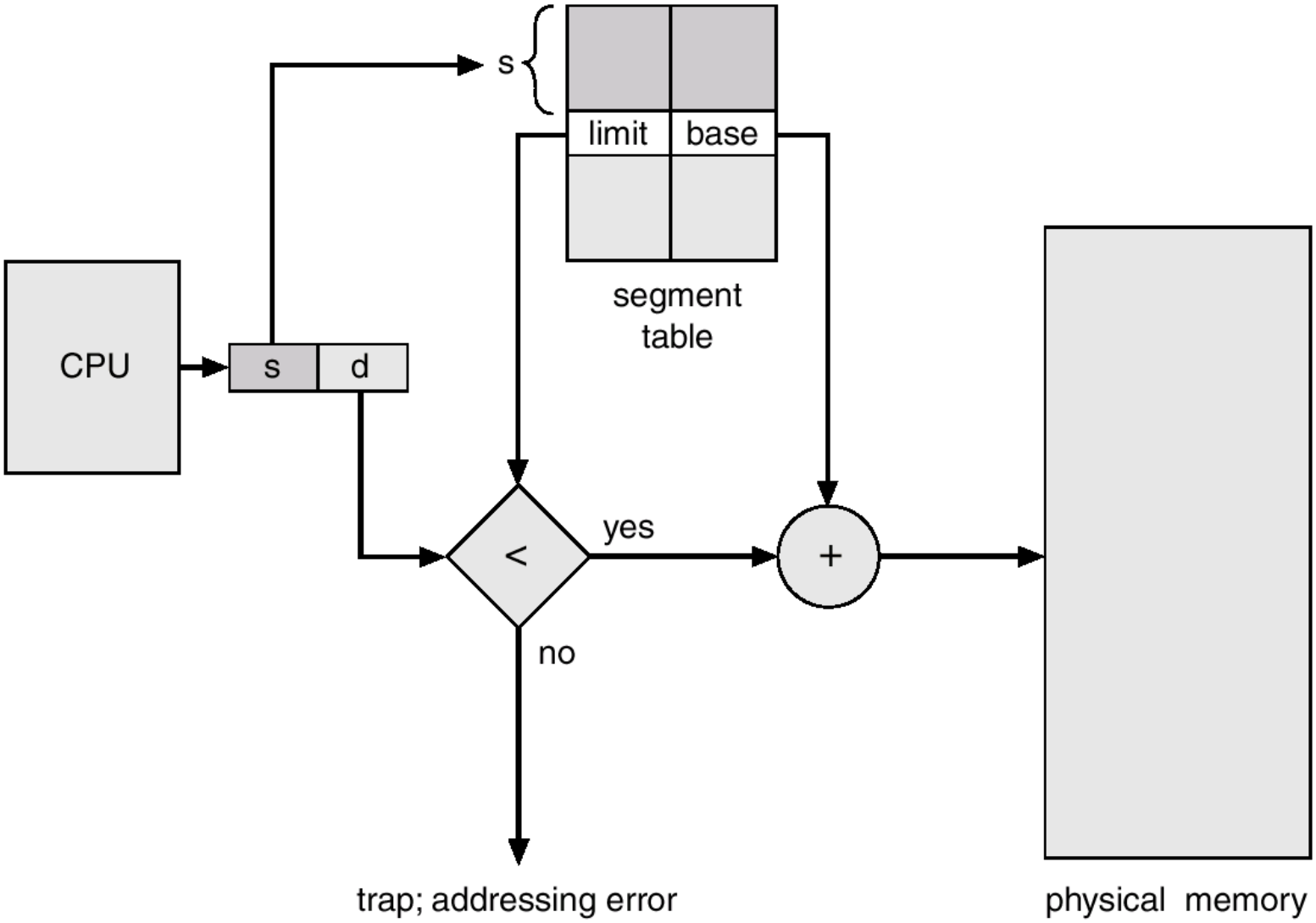
- Logical address consists of a two tuple:

  `<segment-number, offset>`

- Segment table: maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the **segments** reside in memory.
  - limit – specifies the length of the segment.
- Segment-table base register (STBR) points to the segment table's location in memory.
- Segment-table length register (STLR) indicates number of segments used by a program;

  segment number s is legal if s < STLR.

Segmentation Hardware

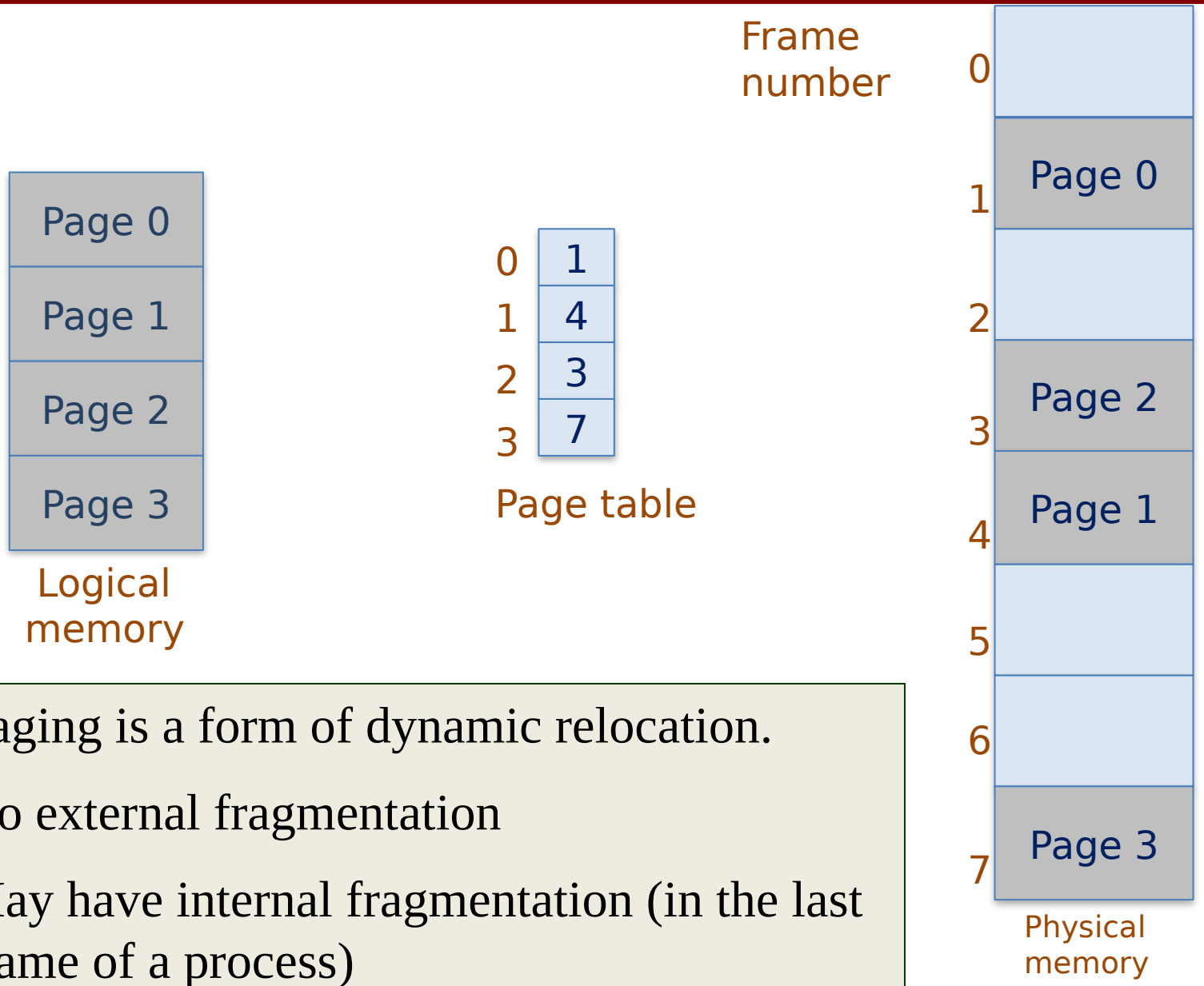


3. Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the space is available.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 16MB).
- Divide logical memory into blocks of the same sized blocks called **pages**.

- Keep track of all free frames.
  - Set up a page table to translate logical to physical addresses.

Paging Model of Logical and Physical Memory



Paging is a form of dynamic relocation.

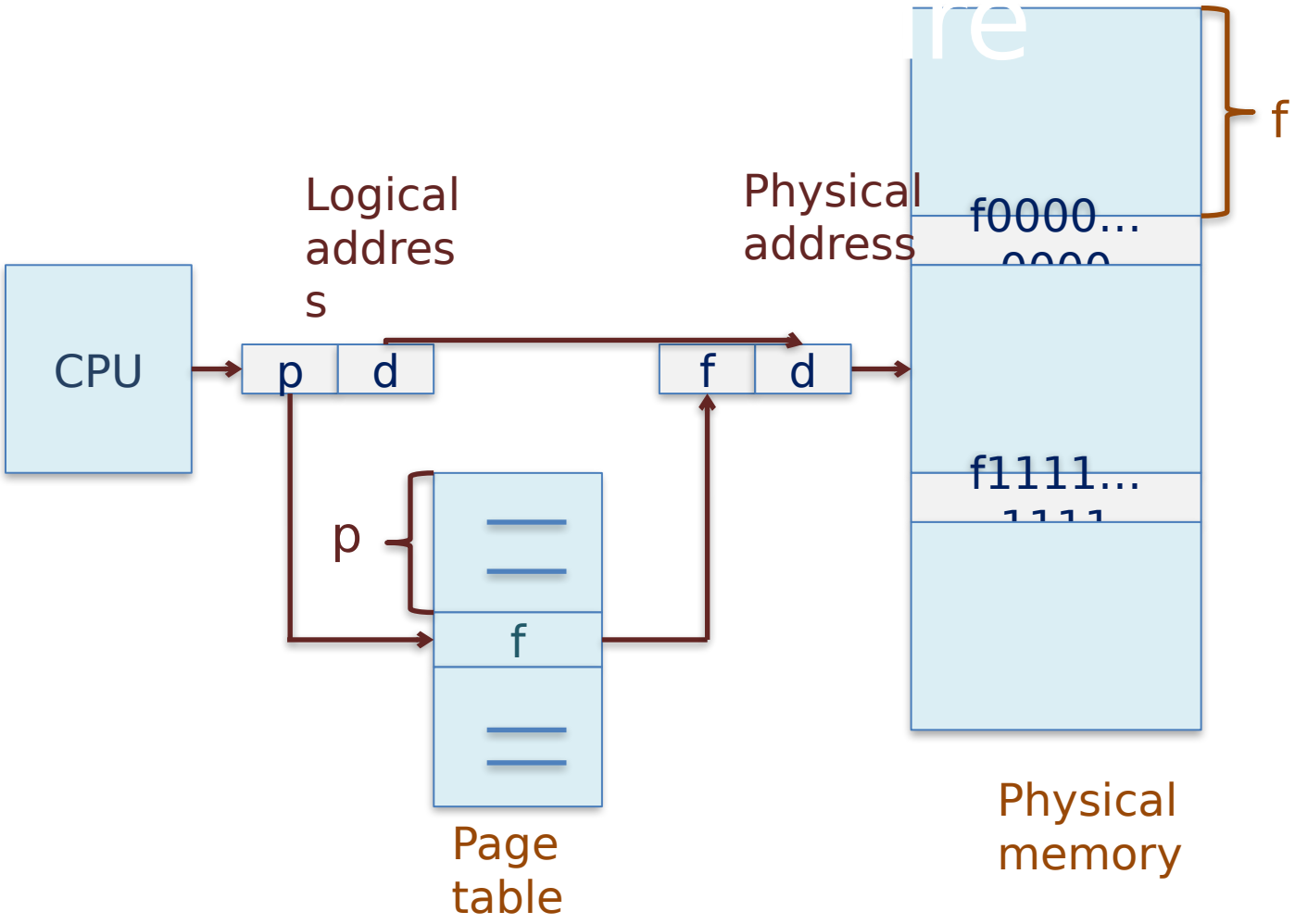
No external fragmentation

May have internal fragmentation (in the last frame of a process)

Address Translation Scheme

- Address generated by CPU (logical address) is divided into:
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory.
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit.

Address Translation



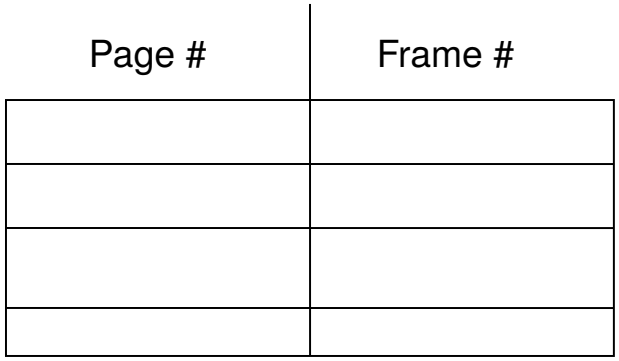
Paging hardware

Implementation of Page Table

- Page table is kept in main memory.
- Page-table base register *(*PTBR) points to the page table.
- Page-table length register (PTLR) indicates size of the page table.
- Problem: Every data/instruction access requires two memory accesses:
  - one for the page table and
  - one for the data/instruction.
- Solution: The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

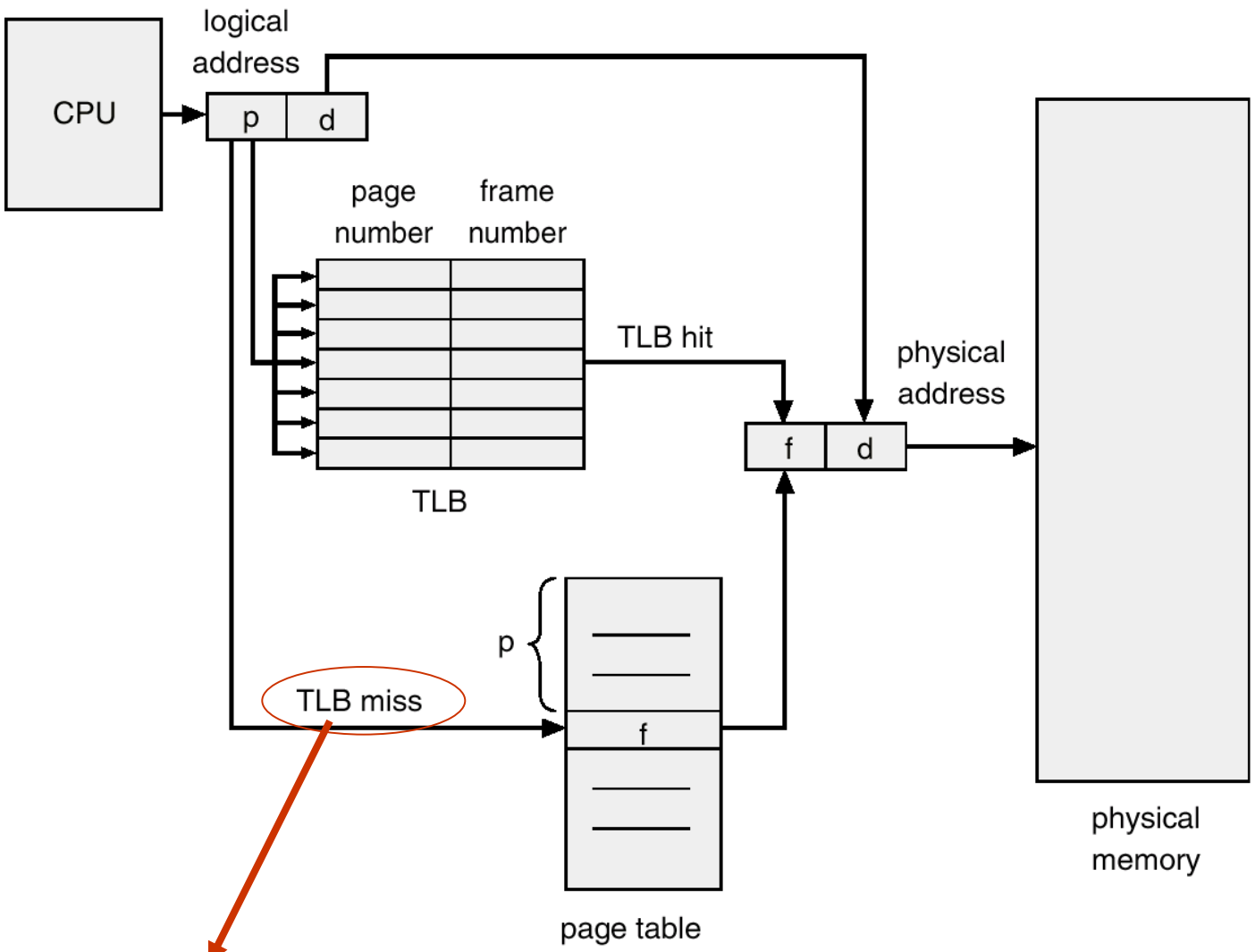
Associative Memory

- Associative memory – parallel search



- Address translation (p, d)
  - If p is in associative register, get frame # out.
  - Otherwise get frame # from page table in memory
- Search is fast
- TLB contains some of the page table entries (64 – 1024) but not all
- Part of the chip's memory management unit (MMU)

Paging Hardware with TLB

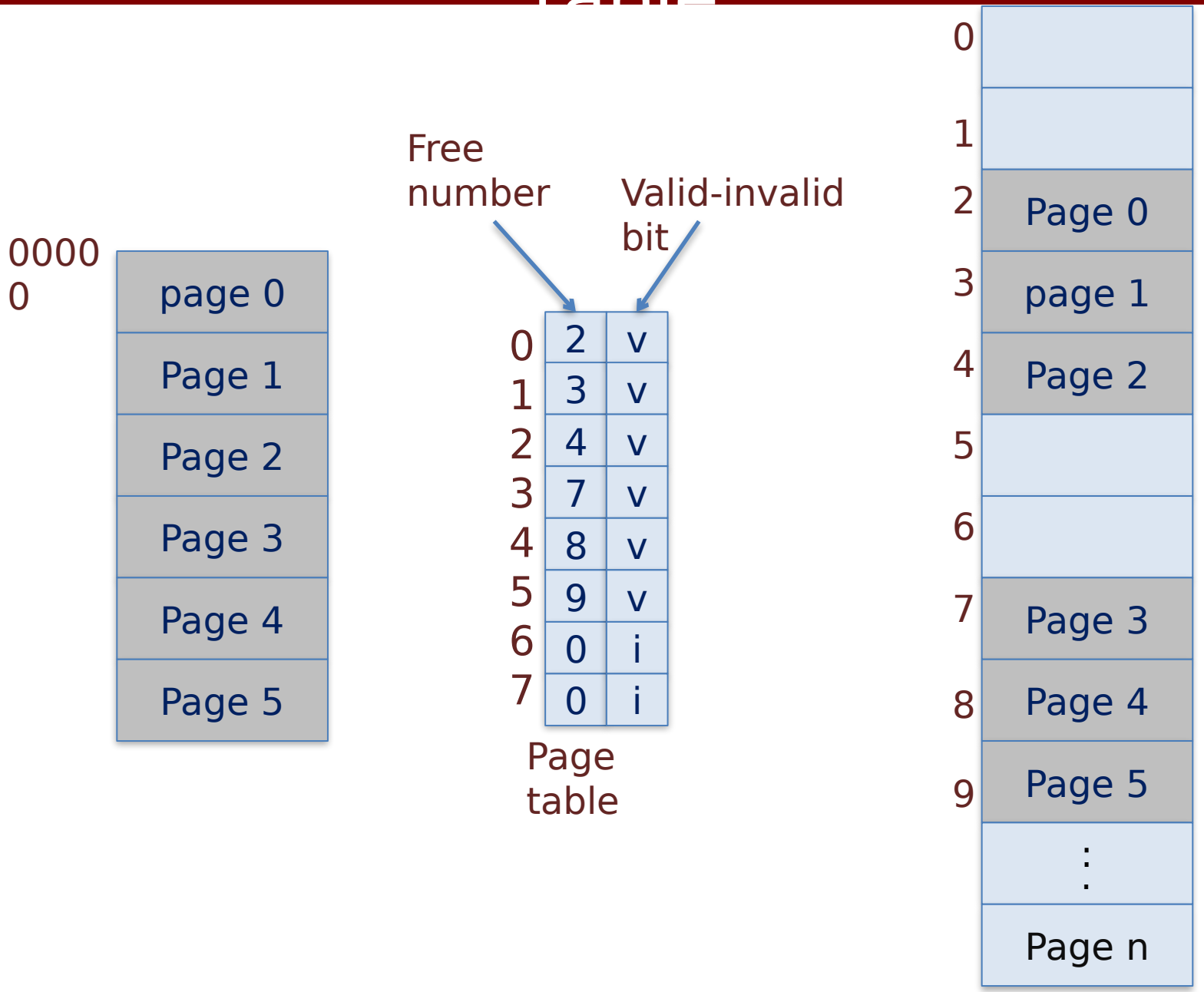


Page # and frame # is added to TLB

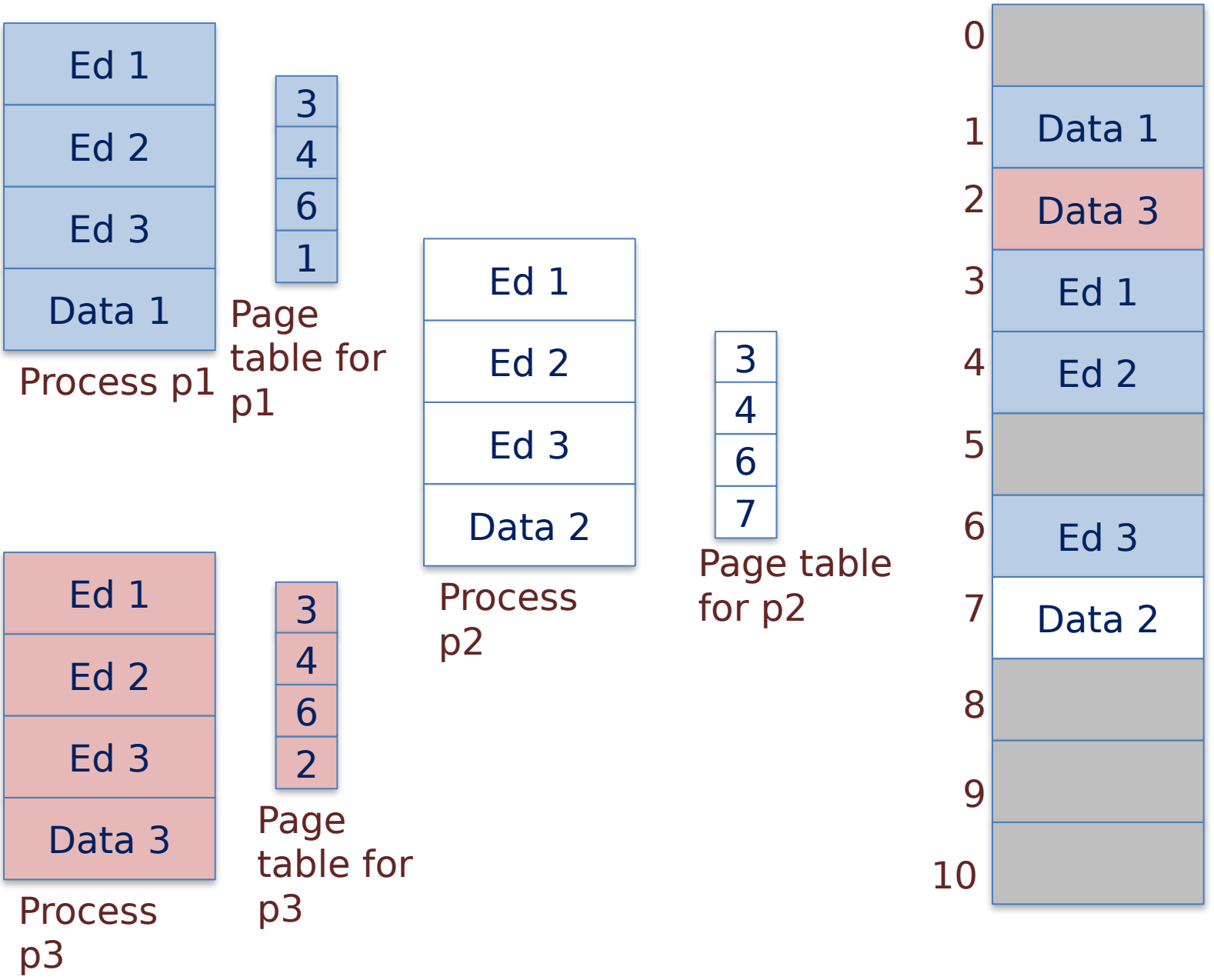
TLB Numbers

- These are typical performance levels of a TLB:
- Size: 12 bits – 4,096 entries
- Hit time: 0.5 – 1 clock cycle
- Miss penalty: 10 – 100 clock cycles
- Miss rate: 0.01 – 1% (20–40% for sparse/graph applications)

Valid (v) or Invalid (i) bit in a page table

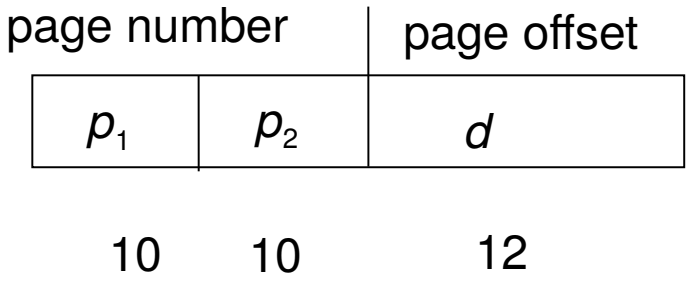


Shared Pages Example



Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:



where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Memory Usage

- **What happens if a process needs more memory than there is available physical memory?**

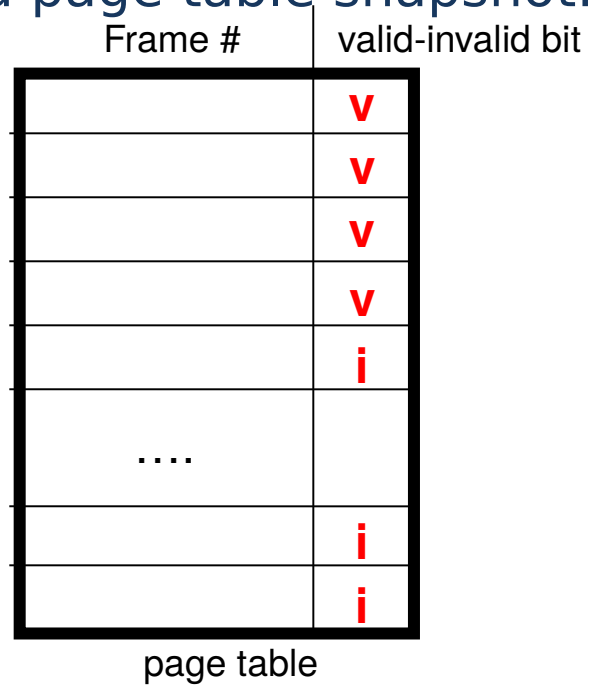
- **Virtual memory**: separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be **much larger than** physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

Demand Paging

- Similar to a paging system with swapping
- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring it to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed

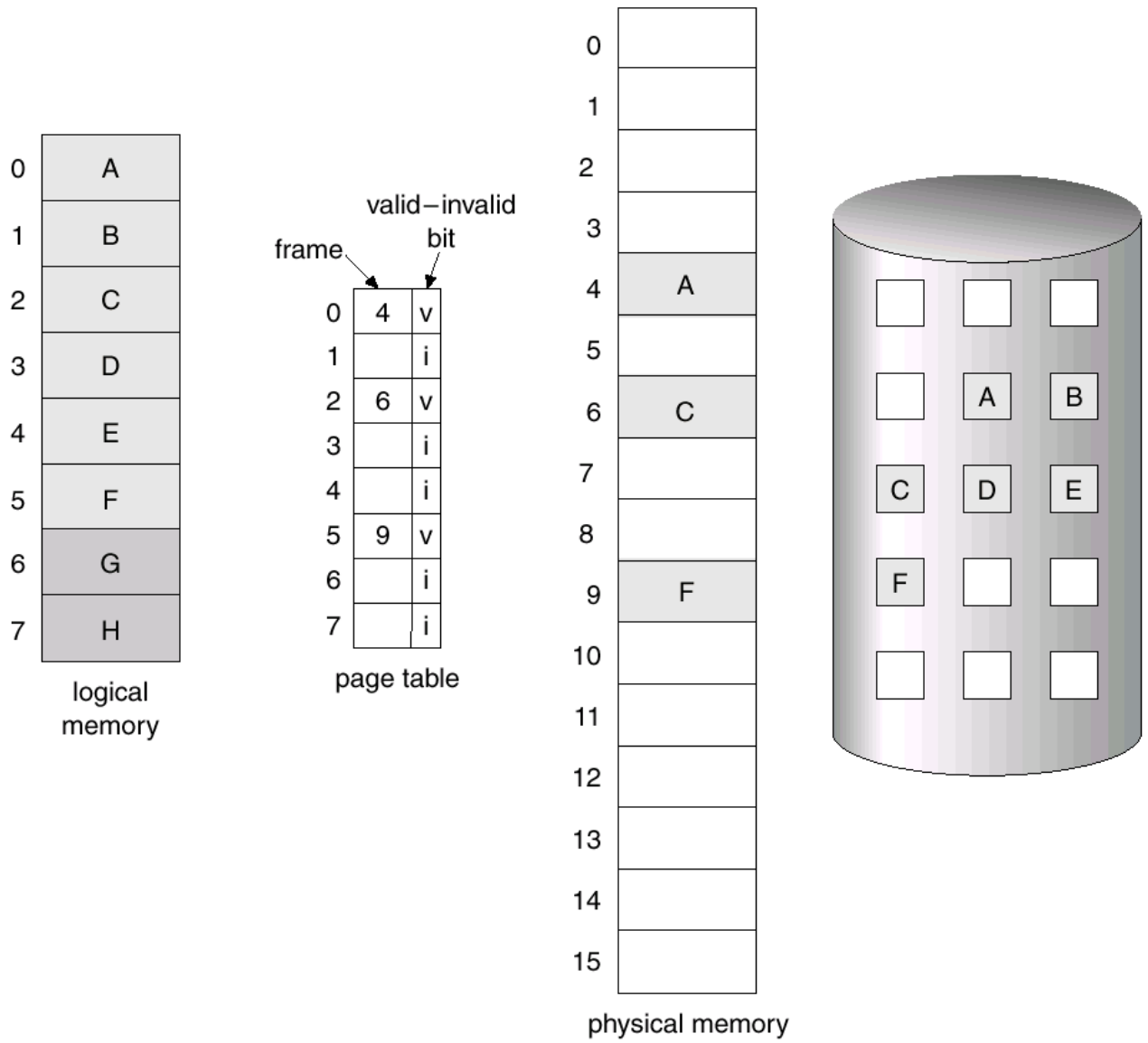
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During address translation, if valid–invalid bit in page table entry
  is **i** ⇒ **page fault**

Page Table when some pages are not in Main Memory

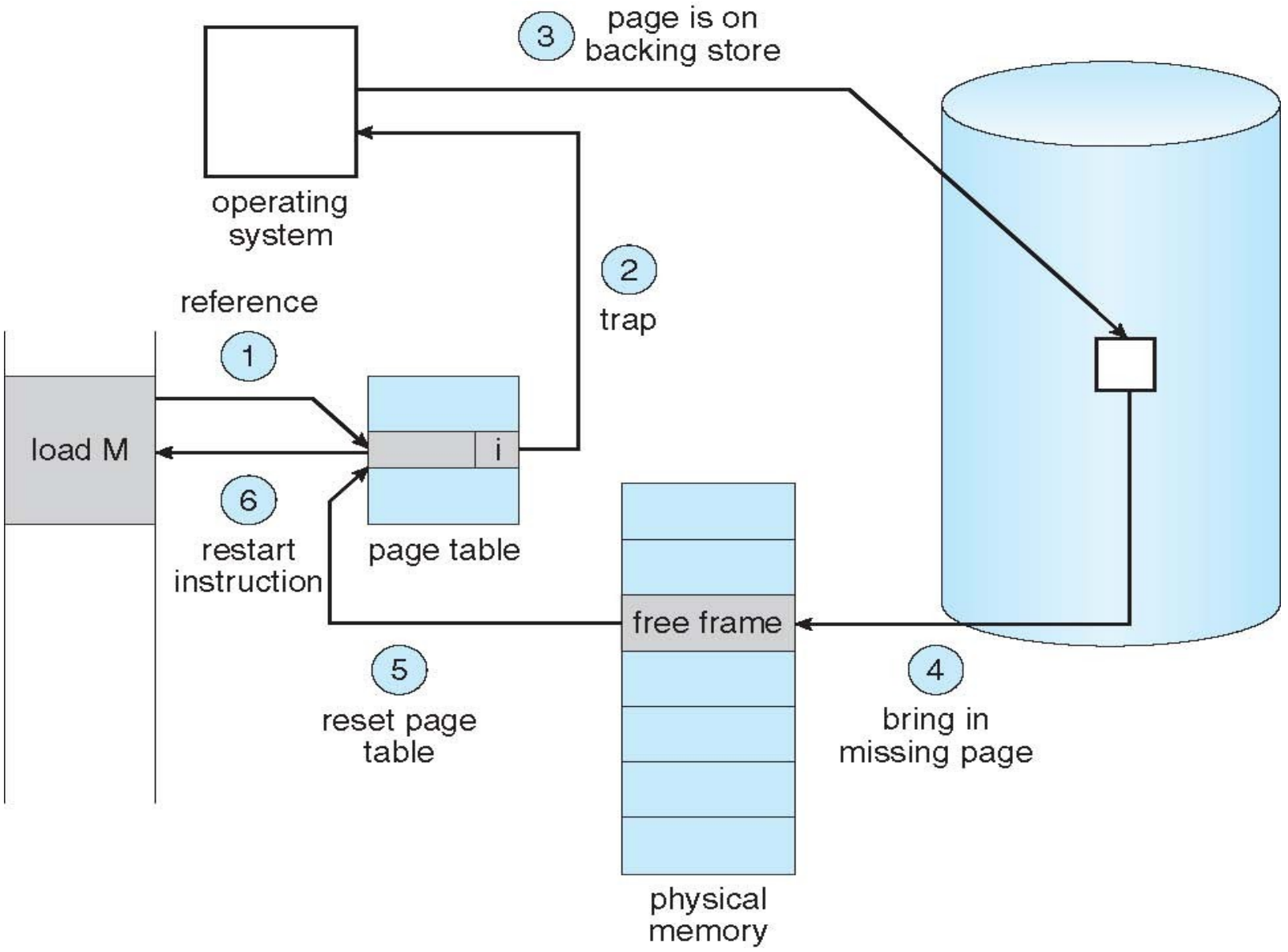


Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

1. Operating system looks at table (kept in Process Control Block) to decide:
   - Invalid reference ⇒ abort
     - out of process's allowed address space
   - Just not in memory
     - The page is in the disk
2. Get an empty frame
3. Swap page into frame
4. Reset table
5. Set validation bit = **v**
6. Restart the instruction of the process that caused the page fault

Steps in Handling a Page Fault



Page Fault (more in detailed)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other process
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other process
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Question?

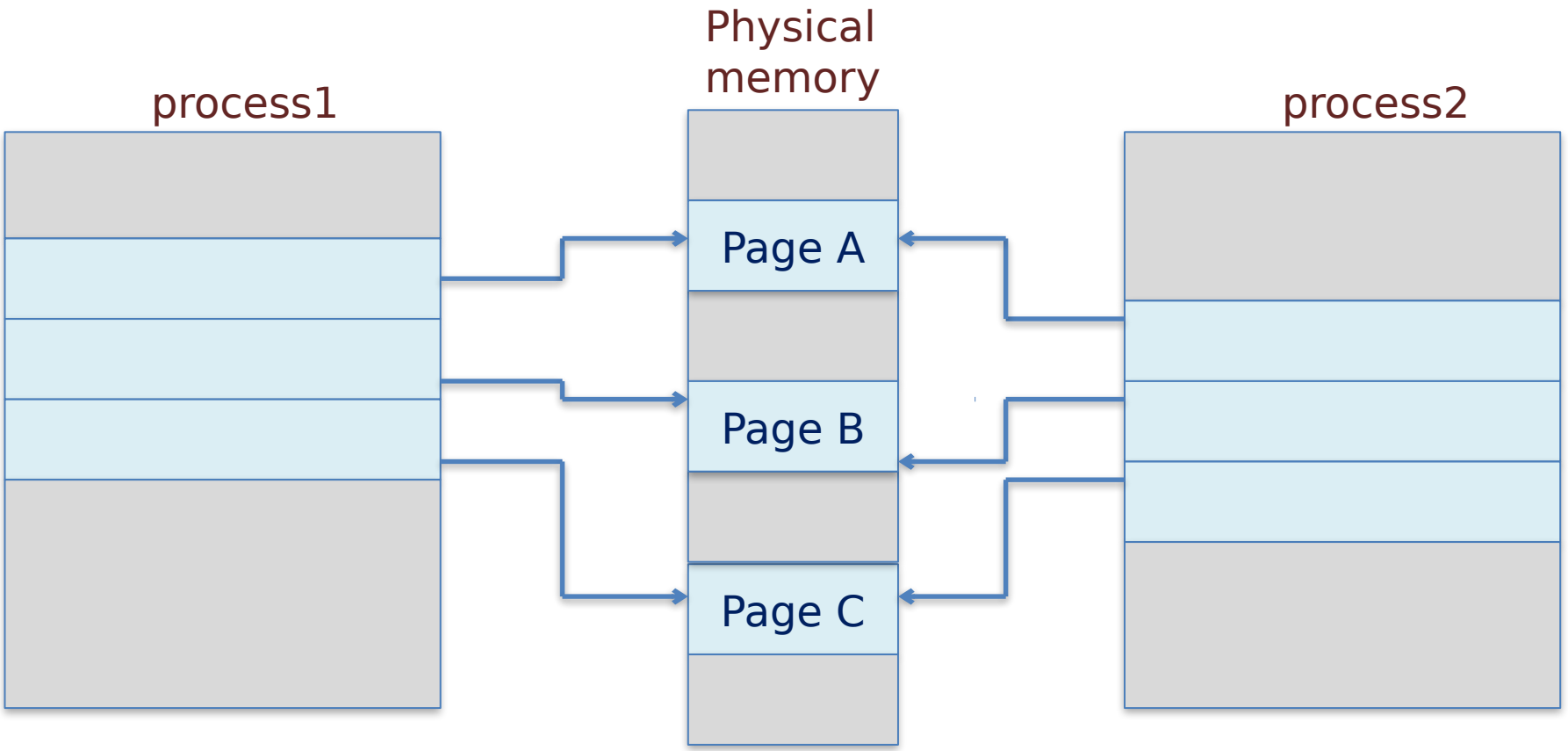
- A thread states ***are Ready, Running, and Blocked***,
  - where a thread is either ready and waiting to be scheduled,
  - is running on the processor, or
  - is blocked (for example, waiting for I/O).
- Assuming a thread is in the Running state
  - Will the thread change state if it incurs a page fault?
    - If so, to what new state?
  - Will the thread change state if it generates a TLB miss that is resolved in the page table?
    - If so, to what new state?

- On a page fault the thread state is set to blocked as an I/O operation is required to bring the new page into memory.
- On a TLB-miss, the thread continues running if the address is resolved in the page table.

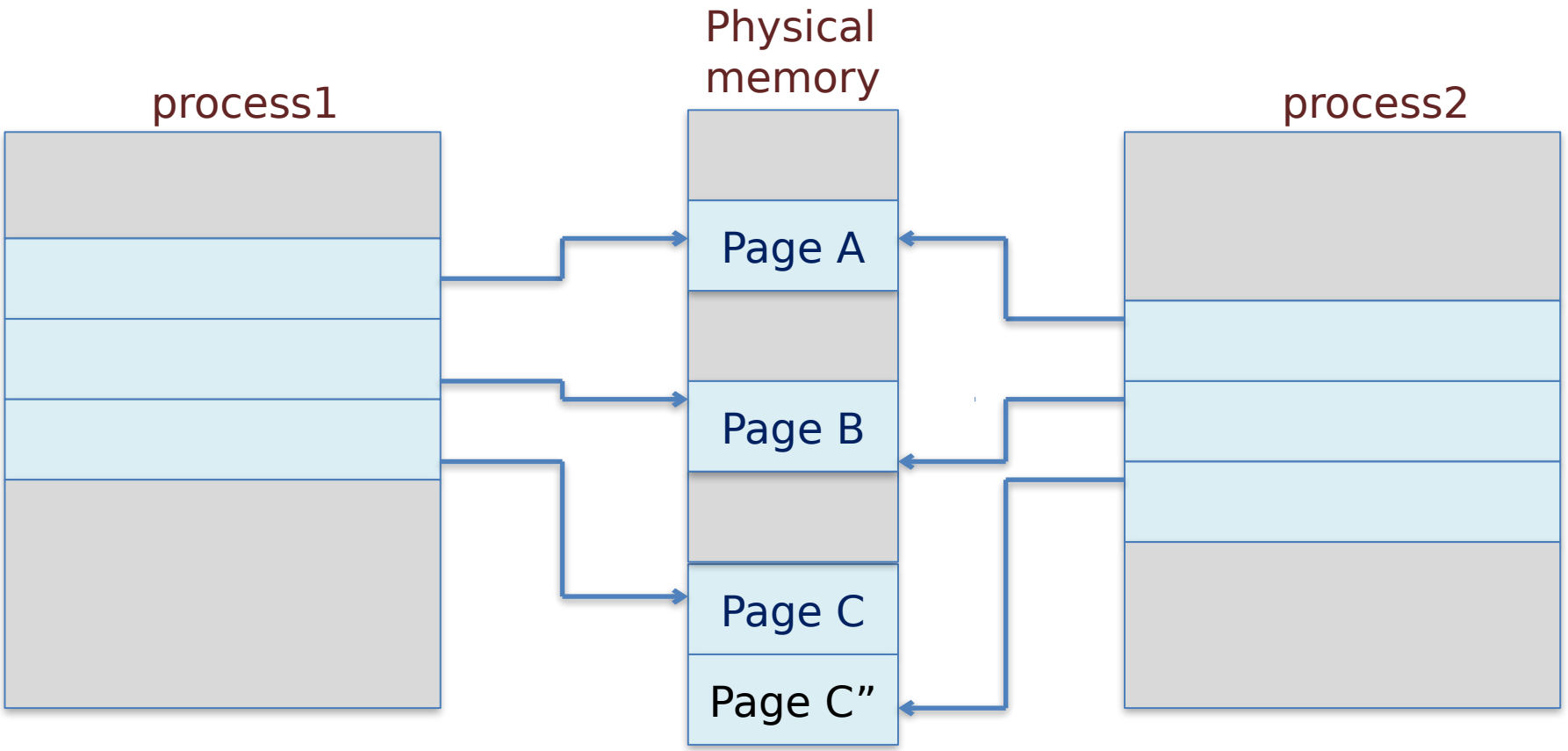
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages

Before Process 2 Modifies Page C

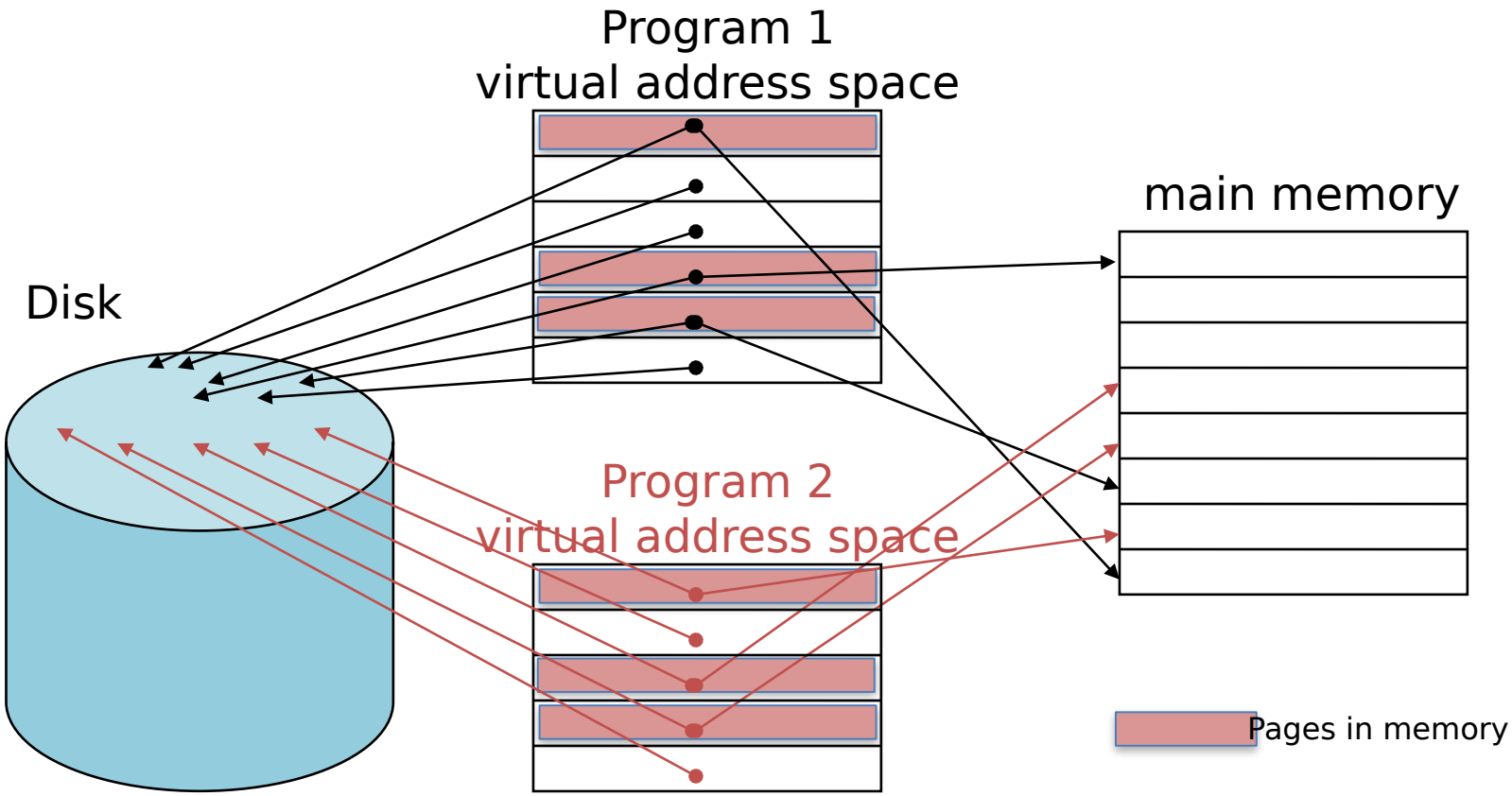


After Process 2 Modifies Page C



Two Processes Sharing Physical Memory

- Each program is compiled into its own address space – a "virtual" address space
  - Address space is divided into pages
- All these pages are in disk but only some of them are in main memory (physical memory) during program execution



Overview of Paging (cont'd)

1. Based on the notion of a **virtual address space**
   - A large, contiguous address space that is only an illusion
     - Virtual address space >> Physical address space
   - Each "program" gets its own separate virtual address space
     - Each **process**, not each thread

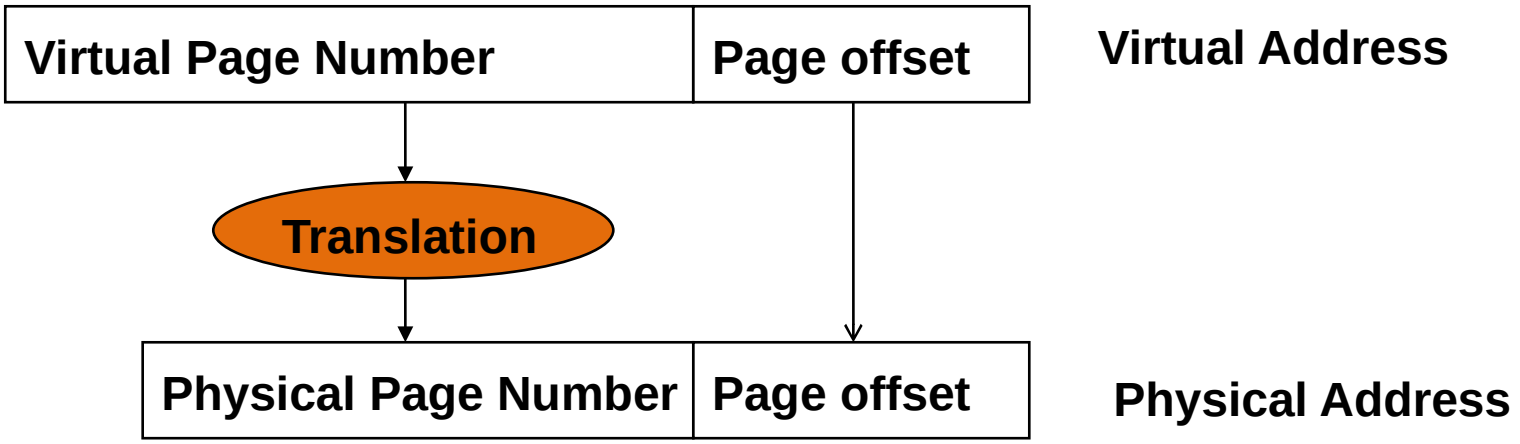
2. Divide the address spaces into fixed-sized **pages**
   - **Virtual page**: A "chunk" of the virtual address space
   - **Physical page**: A "chunk" of the physical address space
     - Also called a **frame**
   - Size of virtual page == Size of physical page

Overview of Paging (cont'd)

3. **Map** virtual pages to physical pages
   - By itself, a virtual page is merely an illusion
     - Cannot actually store anything
     - Needs to be backed-up by a physical page
   - Before a virtual page can be accessed …
     - It must be paired with a physical page
     - I.e., it must be **mapped** to a physical page
     - This mapping is stored somewhere (at OS memory space)
   - On every subsequent access to the virtual page …
     - Its mapping is looked up
     - Then, the access is directed to the physical page

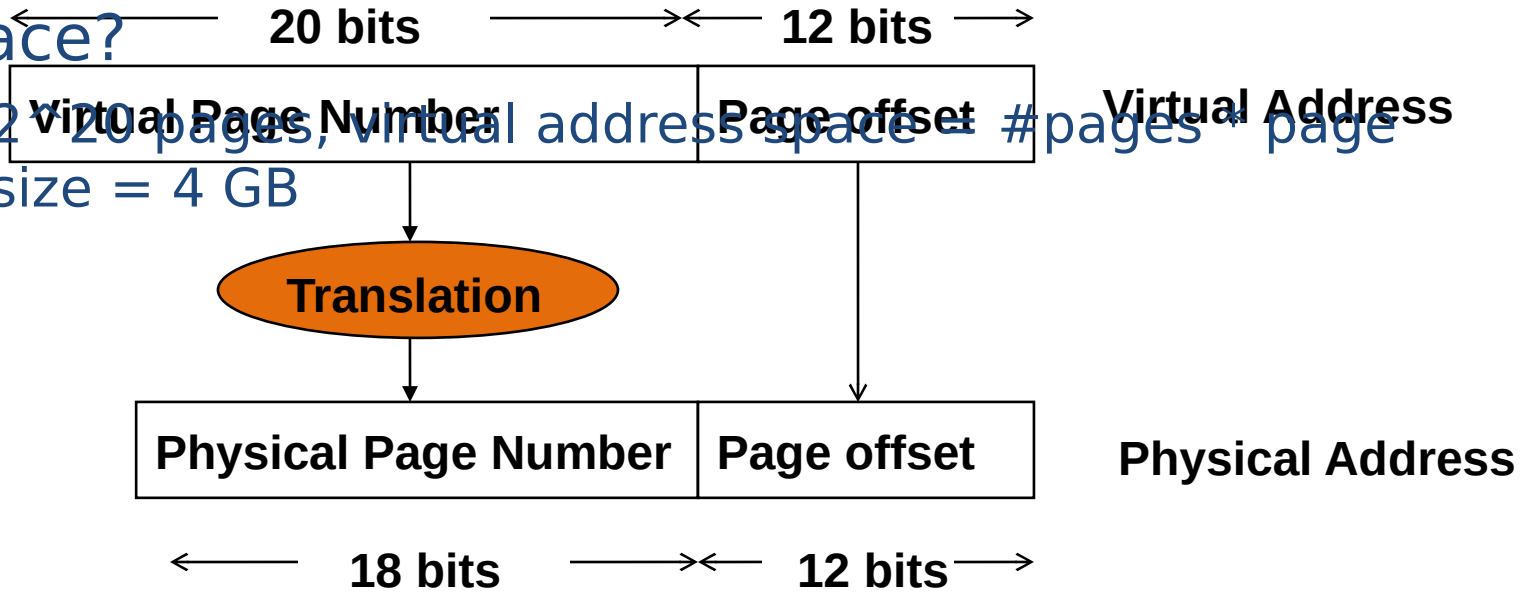
Virtual and Physical Addresses

- So each memory request *first* requires an address translation from the virtual space to the physical space
- Translation is done by a combination of hardware and OS support



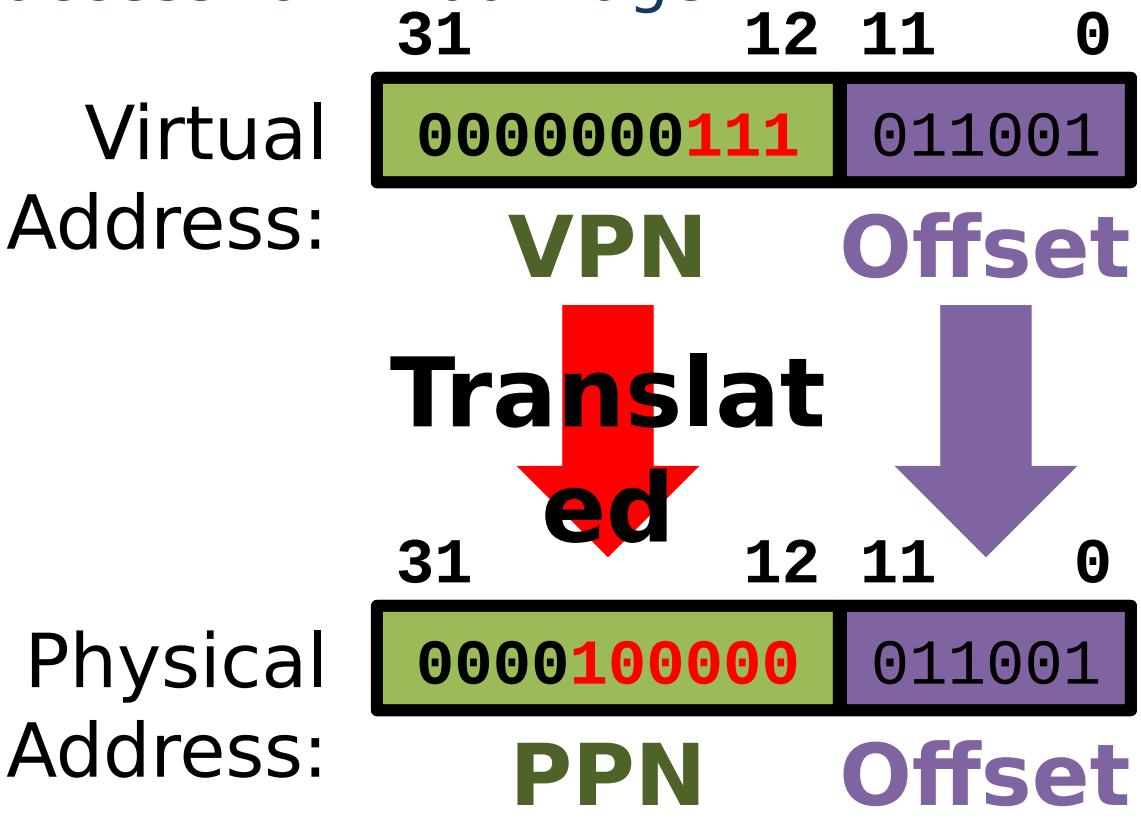
Virtual and Physical Addresses

- What is the page size?
  - 2^12 = 4 KB
- How many pages are allowed in physical memory?
  - 2^18 pages, thus physical address space = #pages * page size = 1 GB
- How many pages are allowed in virtual address space?
  - 2^20 pages, virtual address space = #pages * page size = 4 GB

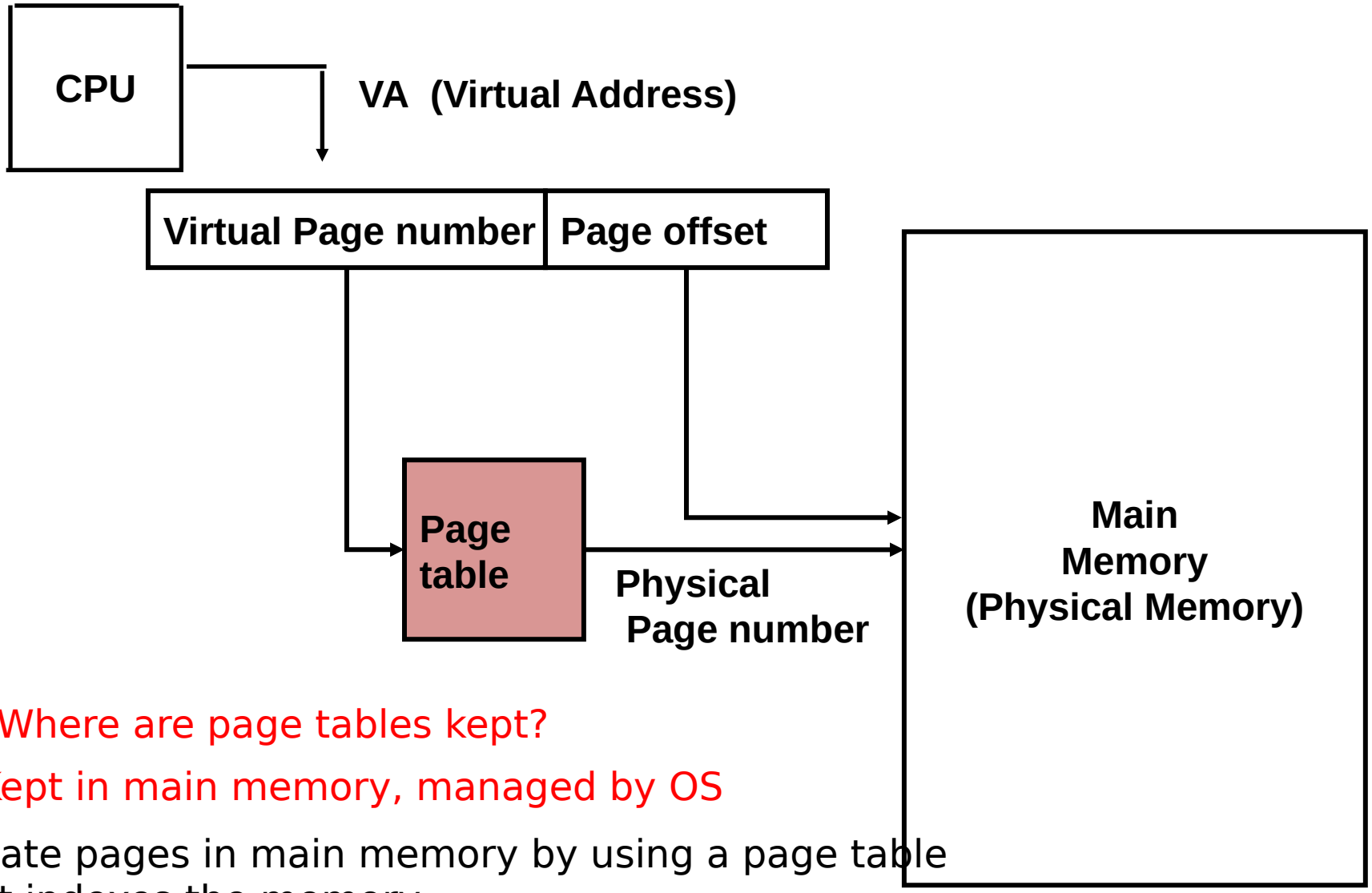


Intel 80386: Translation

- Assume: Virtual Page 7 is mapped to Physical Page 32
- For an access to Virtual Page 7 …



Address Translation



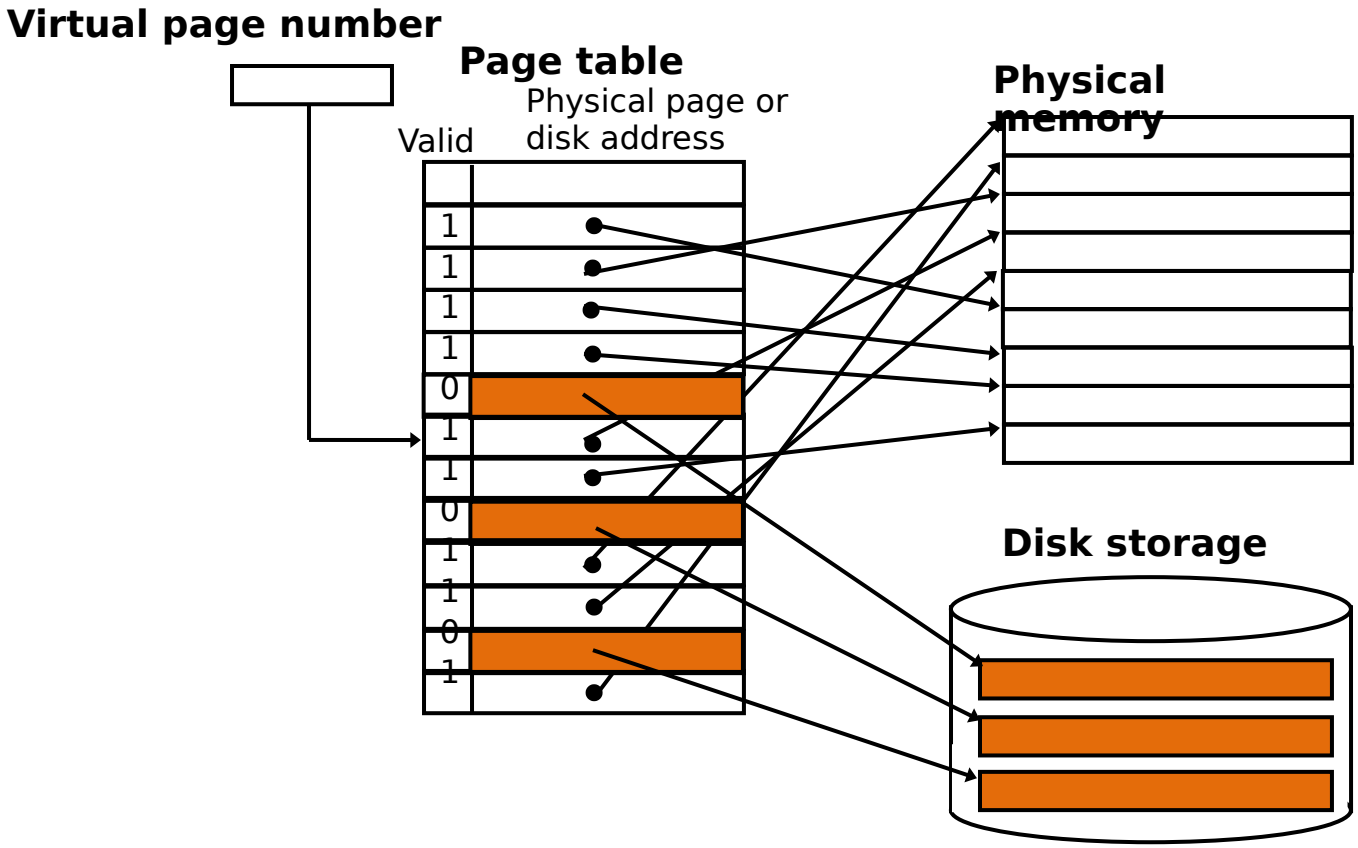
Where are page tables kept?

Kept in main memory, managed by OS

ocate pages in main memory by using a page table
hat indexes the memory

Page Fault

- If the valid bit of the page table is zero, this means that the page is **not in main memory**.
- In this case of a reference to an invalid page, then a page fault occurs, and the missing page is read in from disk.

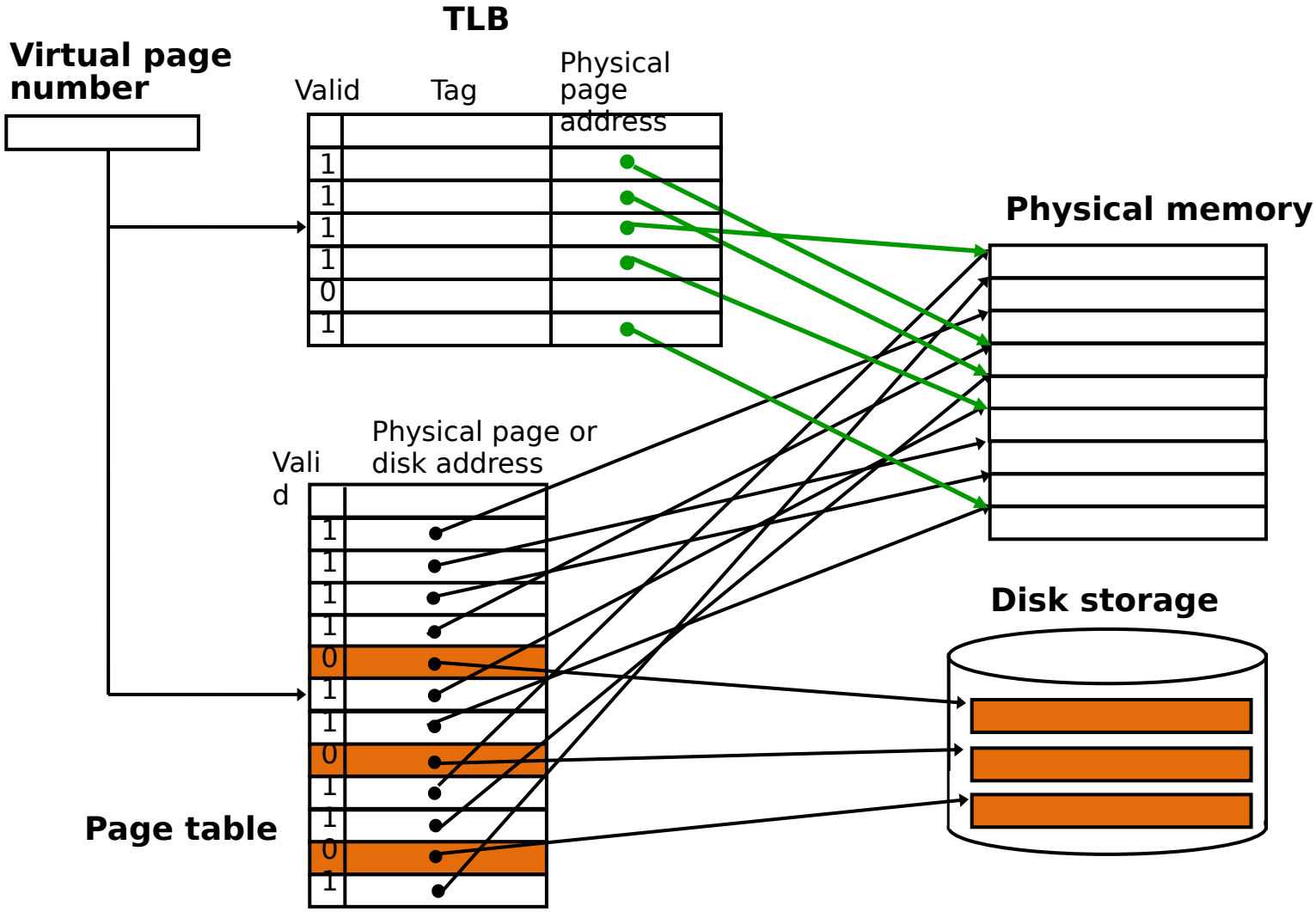


Determining Page Table Size

- Assume
  - 40-bit virtual address
  - 30-bit physical address
  - 8 KB pages
  - Each page table entry is one word (4 bytes)
- How large is the page table?
  - Page offset = 8 KB = 2^13 => 13 bit page offset
  - Virtual page number = 40 - 13 = 27 bits
  - Number of entries in Page Table= number of pages = 2^27
  - Total size = number of entries x bytes/entry
    = 2^27 x 4 = 512 Mbytes

Translation-Lookaside Buffer (TLB)

- A TLB acts as a cache for the page table, by storing physical addresses of pages that have been recently accessed.



Page and Frame Replacement Algorithms

- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace

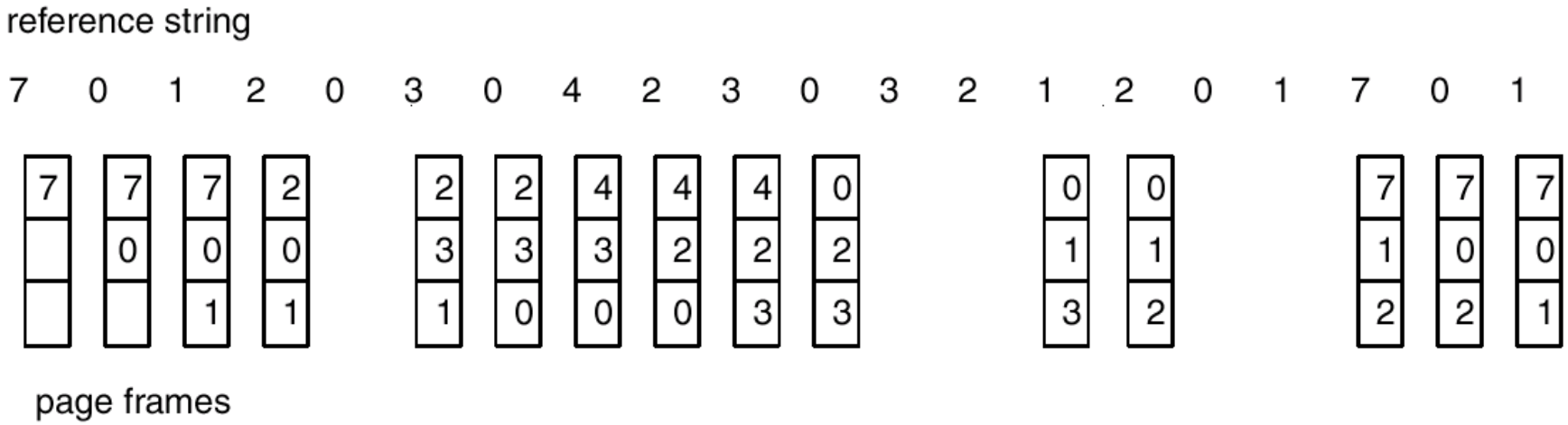
- Evaluate an algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full

1. FIFO Page Replacement

Each page is associated the time when it was brought into memory

When a page must be replaced, the oldest page is chosen

Example: 3 frames/process

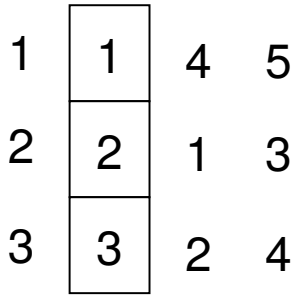


15 page faults

First-in-First-out (FIFO) Algorithm

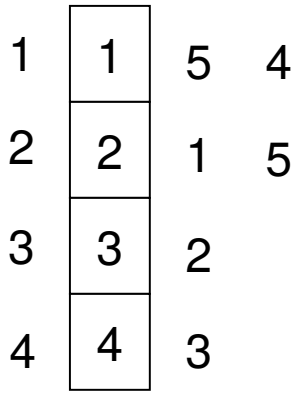
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frames



9 page faults

4 frames



10 page faults

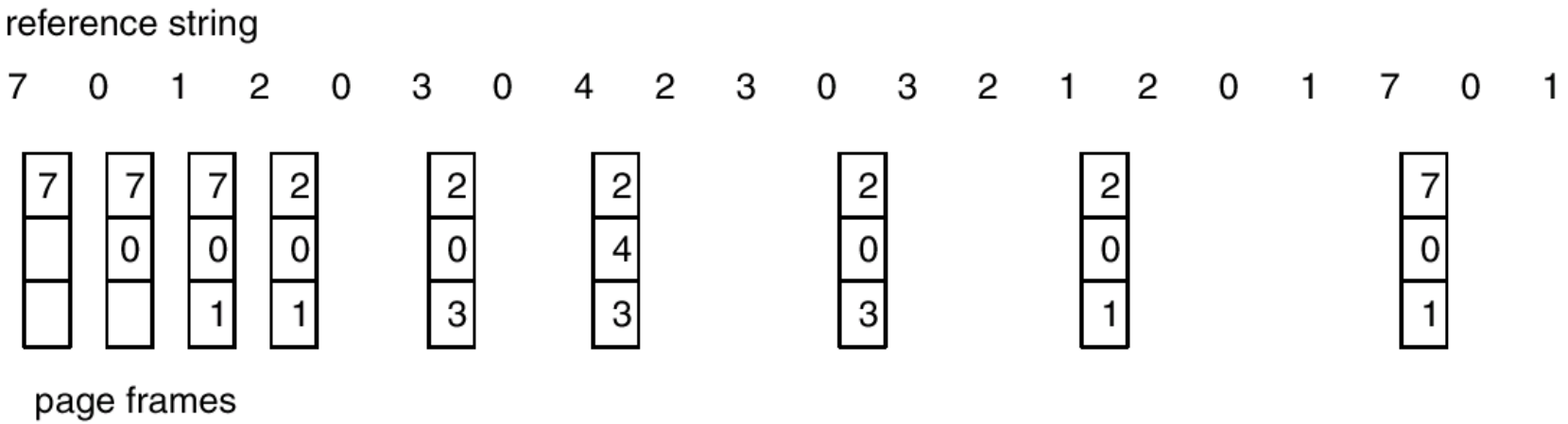
- FIFO Replacement: Belady's Anomaly (page fault rate may *increase* as the number of allocated frames increases!)
- Ideal: more frames ⇒ less page faults

2. Optimal Page Replacement

Replace page that will not be used for the longest period of time.

Lowest page-fault rate of all algorithms

Never suffers from Belady's anomaly

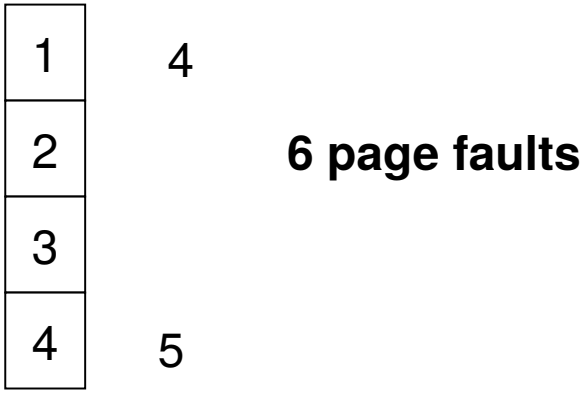


9 page faults

Optimal Algorithm

- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

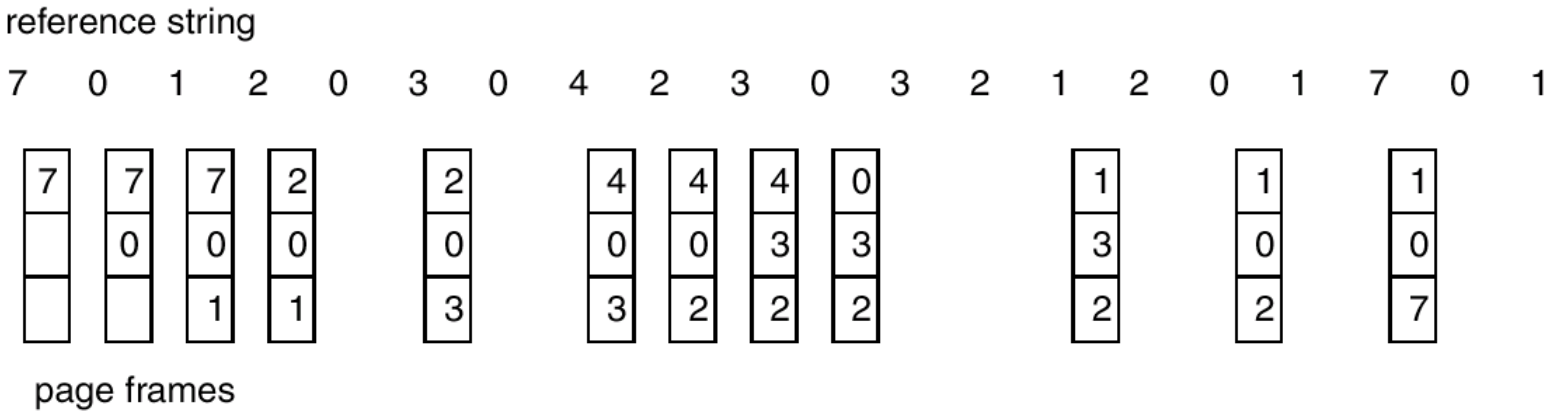


- Requires future knowledge of reference string , which is not possible!
- Used for measuring how well your placement algorithm performs because you cannot be better than optimal

3. Least Recently Used (LRU) Algorithm

Replace page that has not been used for longest period of time.

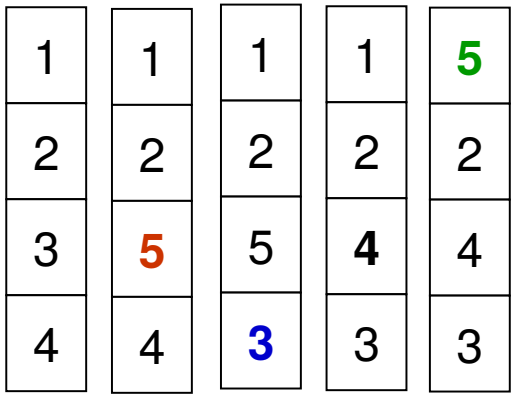
Does not suffer from Belady's anomaly



12 page faults

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



8 page faults

- Counter implementation of LRU
  - Every page entry has a counter; every time a page is referenced, copy the clock into the counter of the page
  - When a page needs to be replaced, look at the counters to determine which one to change (replace the page with the smallest counter) least recently used

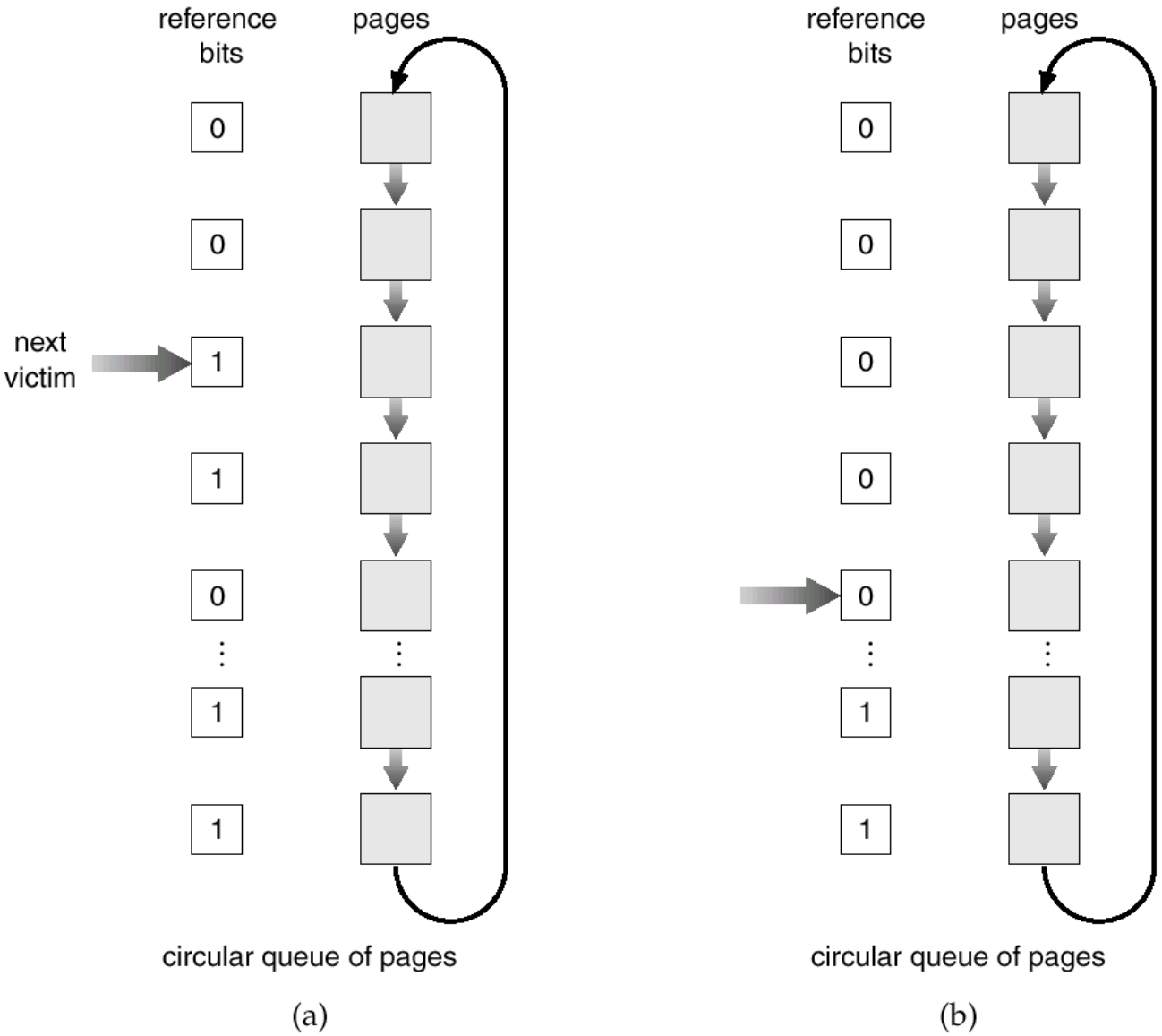
4. LRU Approximation Algorithms

- LRU needs hardware support
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Periodically reset all the bits to zero
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Additional Reference bits**
  - 8 bit shift register
  - At regular intervals OS shifts the reference bit for each page into the high-order bit
  - 1100100 -> 0110010 if not used
  - 1100100 -> 1110010 if used

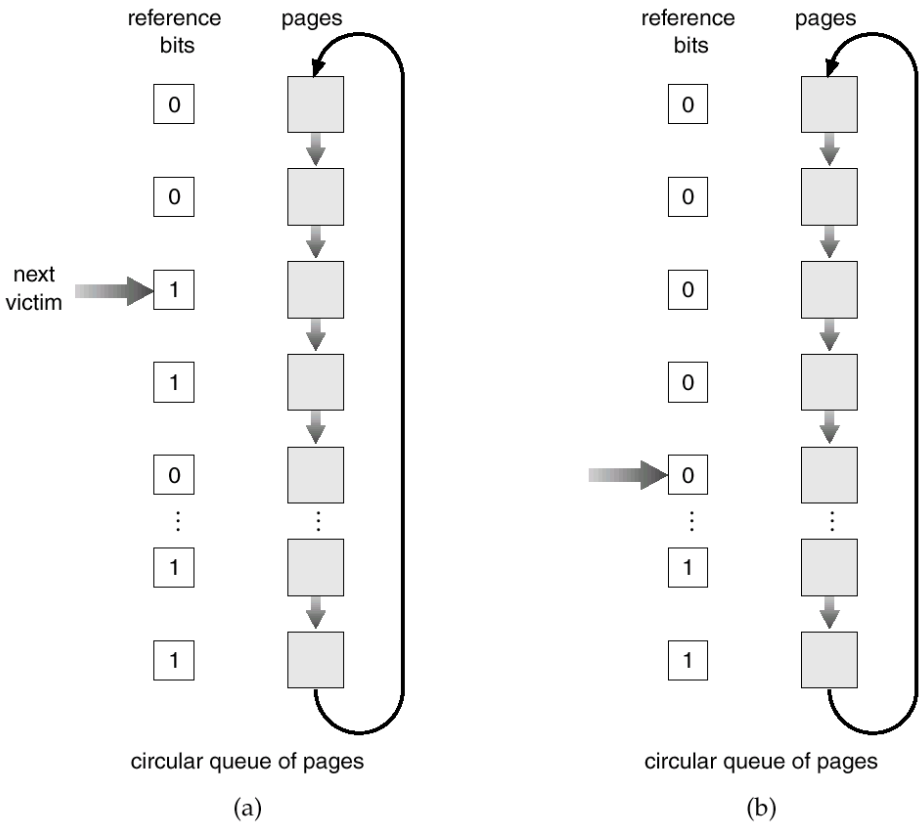
LRU Approximation Algorithms

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - Reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-chance (clock) Page-Replacement Algorithm



Question?



- The pointer indicates the candidate page for replacement.
- What can you say about the system if you notice the following behavior?
  - Pointer is moving fast
  - Pointer is moving slow

5. Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
  - LFU (least frequently used) Algorithm: replaces page with smallest count. Set the counter to zero when a page is moved into memory
  - MFU (most frequently used) Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Allocation of Frames

- How many frames does each process get?
- Each process needs **minimum** number of pages
  - There must be enough frames to hold all the different pages that any single instruction can reference
- Two major allocation methods:
  - fixed allocation (equal, proportional)
  - priority allocation

Fixed Allocation

- Equal allocation:
  - e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation:
  - Allocate according to the size of process.

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

Priority Allocation and Global Replacement

- Priority Allocation
  - Use a proportional allocation method using priorities rather than size.
  - If process P_i generates a page fault,
  - select for replacement one of its frames.
  - select for replacement a frame from a process with lower priority number.

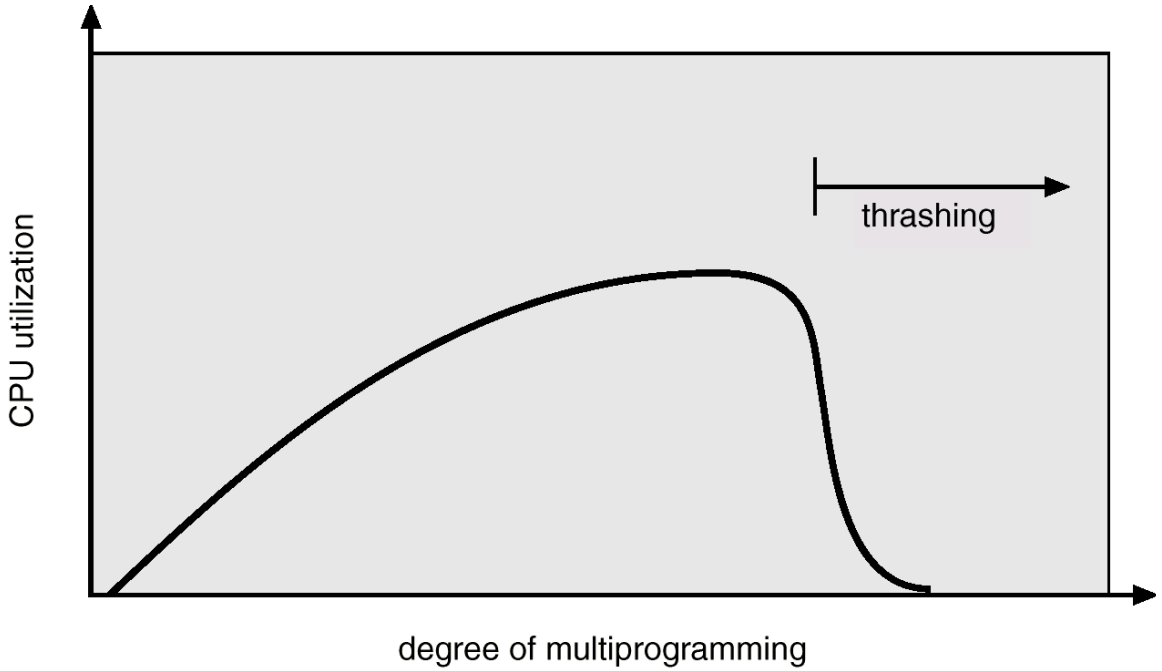
- **Global** replacement:
  - process selects a replacement frame from the set of all frames; one process can take a frame from another.
  - Generally results in greater system throughput
- **Local** replacement:
  - each process selects from only its own set of allocated frames.

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high, leading to
  - Low CPU utilization
  - Operating system thinks that it needs to increase the degree of multiprogramming
    - Why?
  - Another process is added to the system, makes it worse

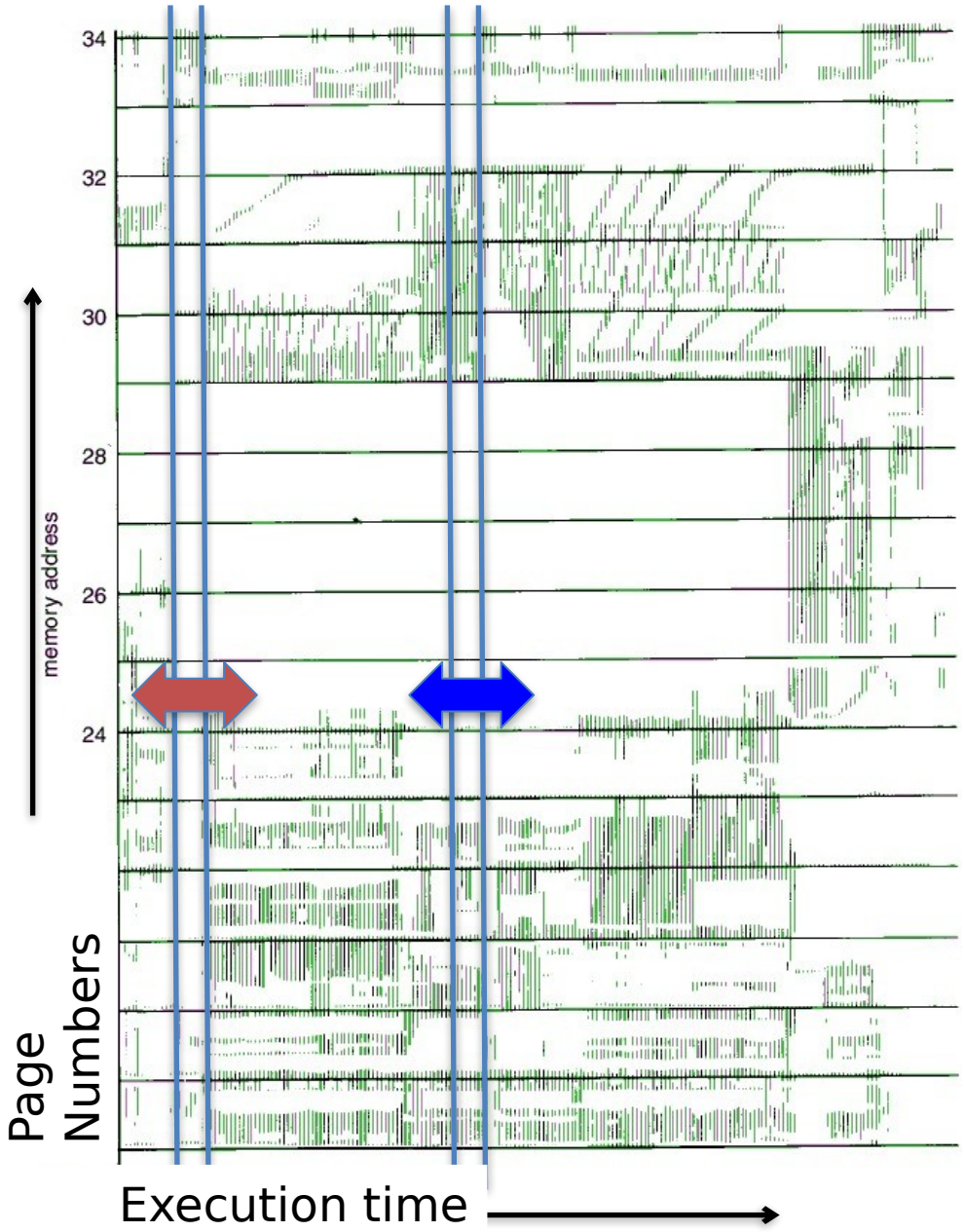
- **Thrashing** ≡ a process is busy swapping pages in and out
- A process is **thrashing** if it is spending more time for paging than executing.

Thrashing



- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another as it executes.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size

Locality in a Memory-Reference Pattern



First time interval, there are few pages that are referenced.

Second time interval, there are many more pages referenced.

Working-Set Model

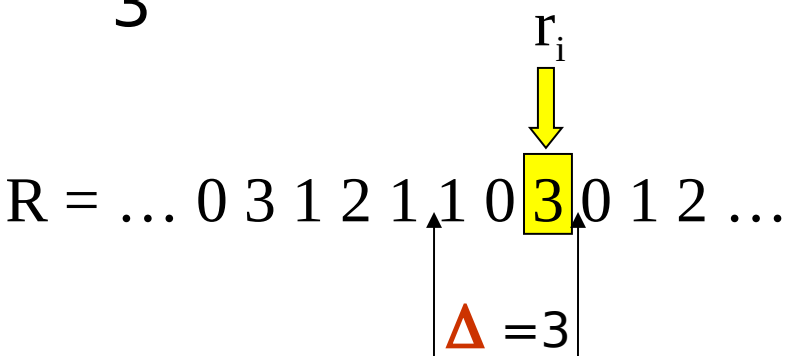
- Working-set model prevents thrashing while keeping the degree of multiprogramming as high as possible optimizes CPU utilization
- To prevent thrashing, provide a process as many frames as it needs.
- Working set model defines the locality model of process execution.
  - An approximation of the set of pages that the process will access in the future

Working-Set Model

- Peter Denning in 1968 defined the working set model of a process
- Δ (working-set window): time interval from *t* to *t*+*i*
  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if $\Delta = \infty \Rightarrow$ will encompass entire program.
- WS_i : working set (pages) of process P_i
- WSS_i (working set size of process P_i): total number of pages referenced in the most recent Δ, (varies in time)
- $D = \Sigma\ WSS_i$: total demand of frames
- m : total number of frames (memory capacity)
- if D > m $\Rightarrow$ Thrashing
- Policy: if D > m, then suspend one of the processes.

Working-Set model

The "Window Size" = Δ = 3



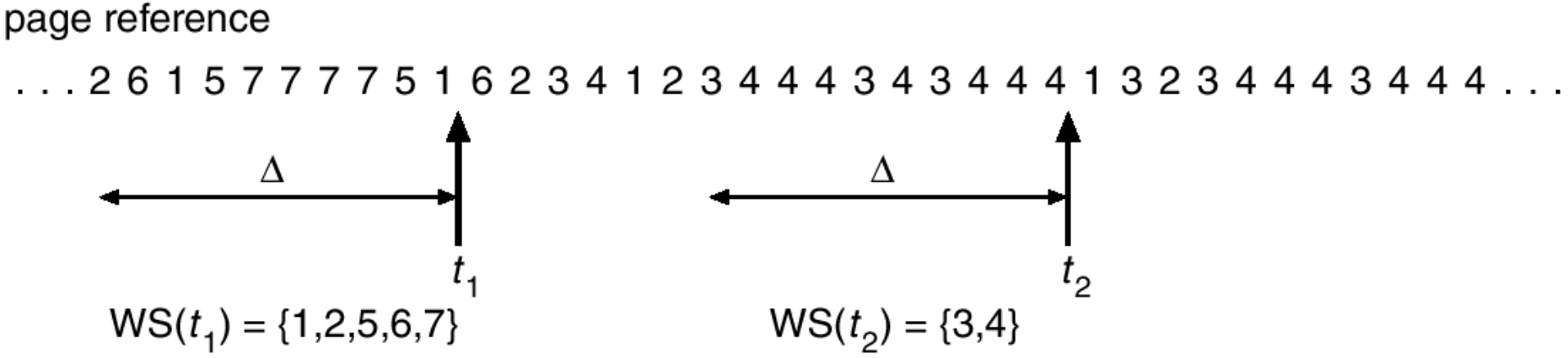
At virtual time i: working set = {0, 1, 3}

At virtual time i-1: working set = {0, 1}

What if $\Delta = 4$?

Working-Set model

$\Delta = 10$



Questions – Program Structure

- Assume page size is 1024 words
- **int A[][] = new int[1024][1024];**
- Each row is stored in one page
- If the OS allocates less than 1024 frames for the program
- Program 1

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i][j] = 0;
```

- Program 2

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i][j] = 0;
```

How many page faults occur in each program?

File

- A collection of related bytes having meaning only to the creator. The file can be "free formed", indexed, structured, etc.
- The file is an entry in a directory.
- The file may have structure (O.S. may or may not know about this.) It's a tradeoff of capabilities versus overhead. For example,
  a) An Operating System understands program image format in order to create a process.
  b) The UNIX **shell** understands how directory files look. (In general the UNIX **kernel** doesn't interpret files.)
  c) Usually the Operating System understands and interprets file types.

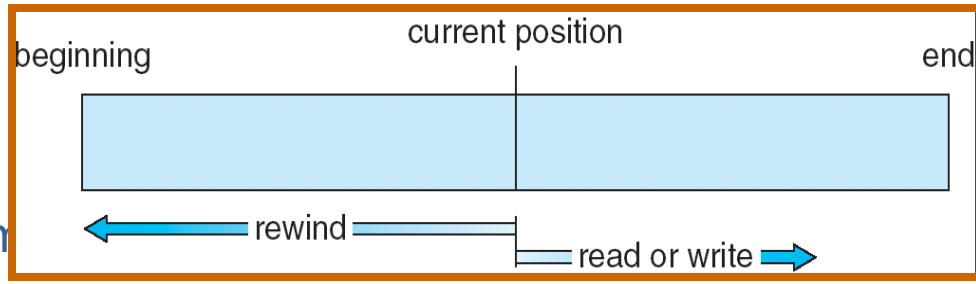
File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring

- Information about files are kept in the directory structure, which is maintained on the disk

File Systems Interface — Access Methods

If files had only one "chunk" of data, life would be simple. But for large files, the files themselves may contain structure, making access faster.



SEQUENTIAL ACCESS

- Implemented by the filesystem
- Data is accessed one record right after the last.
- Reads cause a pointer to be moved ahead by one.
- Writes allocate space for the record and move the pointer to the new End Of File.
- Such a method is reasonable for tape

File Systems Interface

Access Methods

DIRECT ACCESS

- Method useful for disks.
- The file is viewed as a numbered sequence of blocks or records.
- There are no restrictions on which blocks are read/written in any order.
- User now says "read n" rather than "read next".
- "n" is a number relative to the beginning of file, not relative to an absolute physical disk location.

File System Interface

OTHER ACCESS METHODS

Access Methods

Built on top of direct access and often implemented by a user utility.

Indexed ID plus pointer.

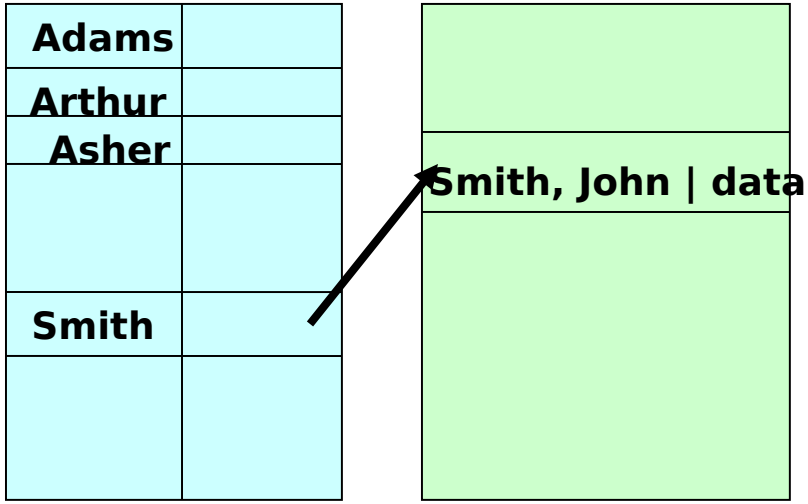
An index block says what's in each remaining block or contains pointers to blocks containing particular items. Suppose a file contains many blocks of data arranged by name alphabetically.

Example 1: Index contains the name appearing as the first record in each block. There are as many index entries as there are blocks.

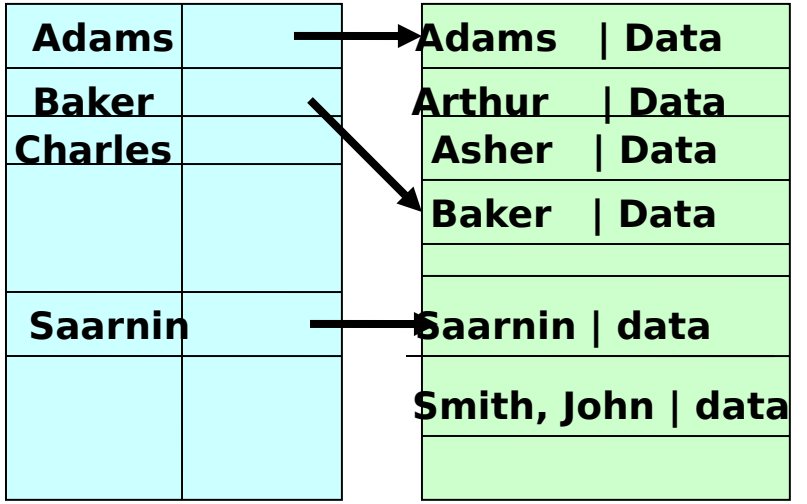
Example 2: Index contains the block number where "A" begins, where "B" begins, etc. Here there are only 26 index entries.

File System Interface — Access Methods

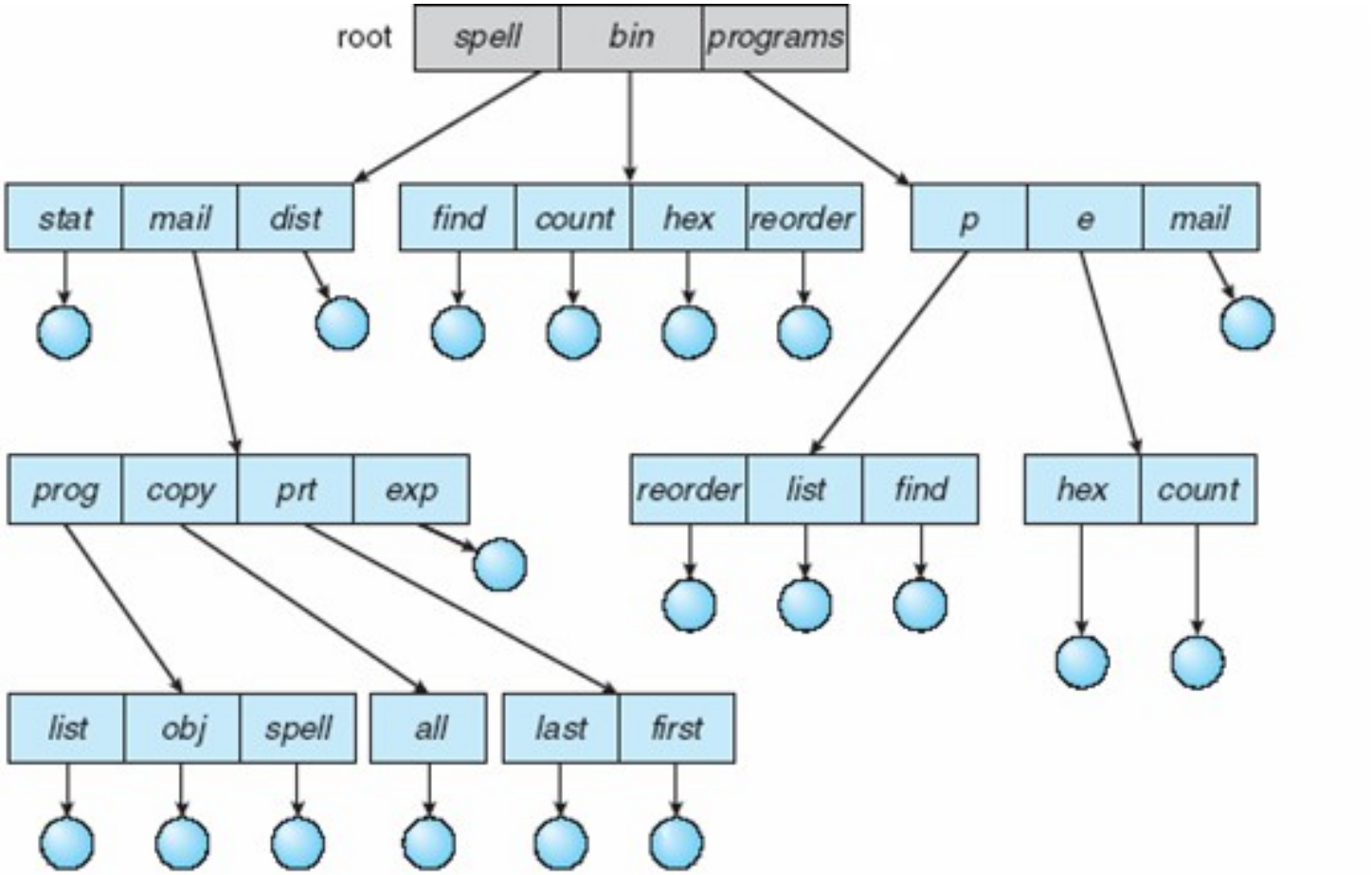
Example 1: Index contains the name appearing as the first record in each block. There are as many index entries as there are blocks.



Example 2: Index contains the block number where "A" begins, where "B" begins, etc. Here there are only 26 index entries.



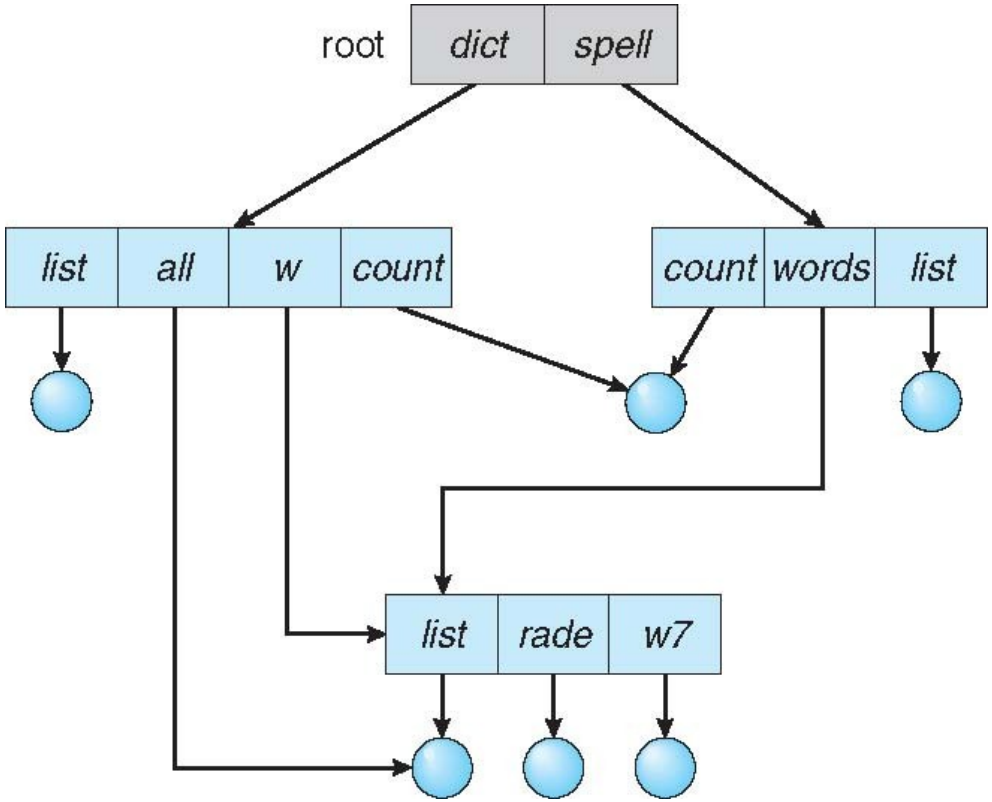
Tree-Structured Directories



Other non-tree structures possible?

Acyclic-Graph Directories

- Have shared subdirectories and files
- How can we implement this?
  - Links

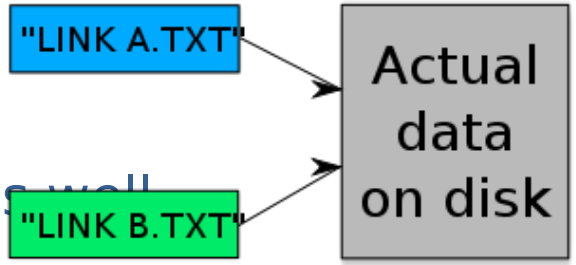


What happens when the link is deleted?

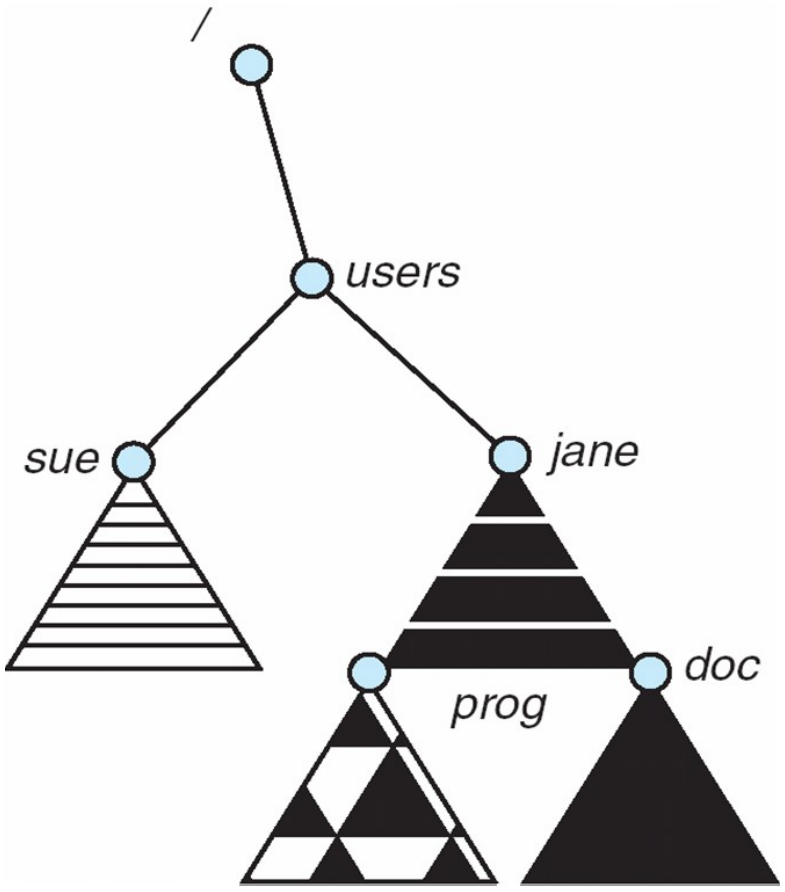
What happens when the original file is deleted?

Symbolic vs Hard Links

- Both are used for aliasing files
  - Multiple names for the same file
- Symbolic Link
  - contains a reference to another file or directory in the form of an absolute or relative path
  - When the link is deleted, file remains
  - When the file is deleted, the link remains – user has to clean up
  - ln -s target_path link_path
- Hard Links
  - Keep a reference (link) count
  - When count is zero, delete the file as well
  - Not recommended to use
  - Only few OSs support with a root access



Mount Point



- Mac OS X searches for a file system on the device at boot time or while the system running
  - **It automatically mounts the file system under the /Volume directory**
- Unix
  - Requires explicit mount
    - Usually under /mnt
  - The ones listed in configuration file containing list of devices are automatically mounted

File Sharing

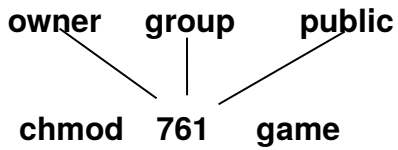
- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- In multi-user system
  - **User IDs** identify users, allowing permissions and protections to be per-user
    Group IDs allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory

Access Lists and Groups in Unix

- Mode of access: read (4), write (2), execute (1)
- Three classes of users on Unix / Linux

	RWX		
a) **owner access**	7	⇒	1 1 1
	RWX		
b) **group access**	6	⇒	1 1 0
	RWX		
c) **public access**	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

`chgrp G game`

Virtual File Systems

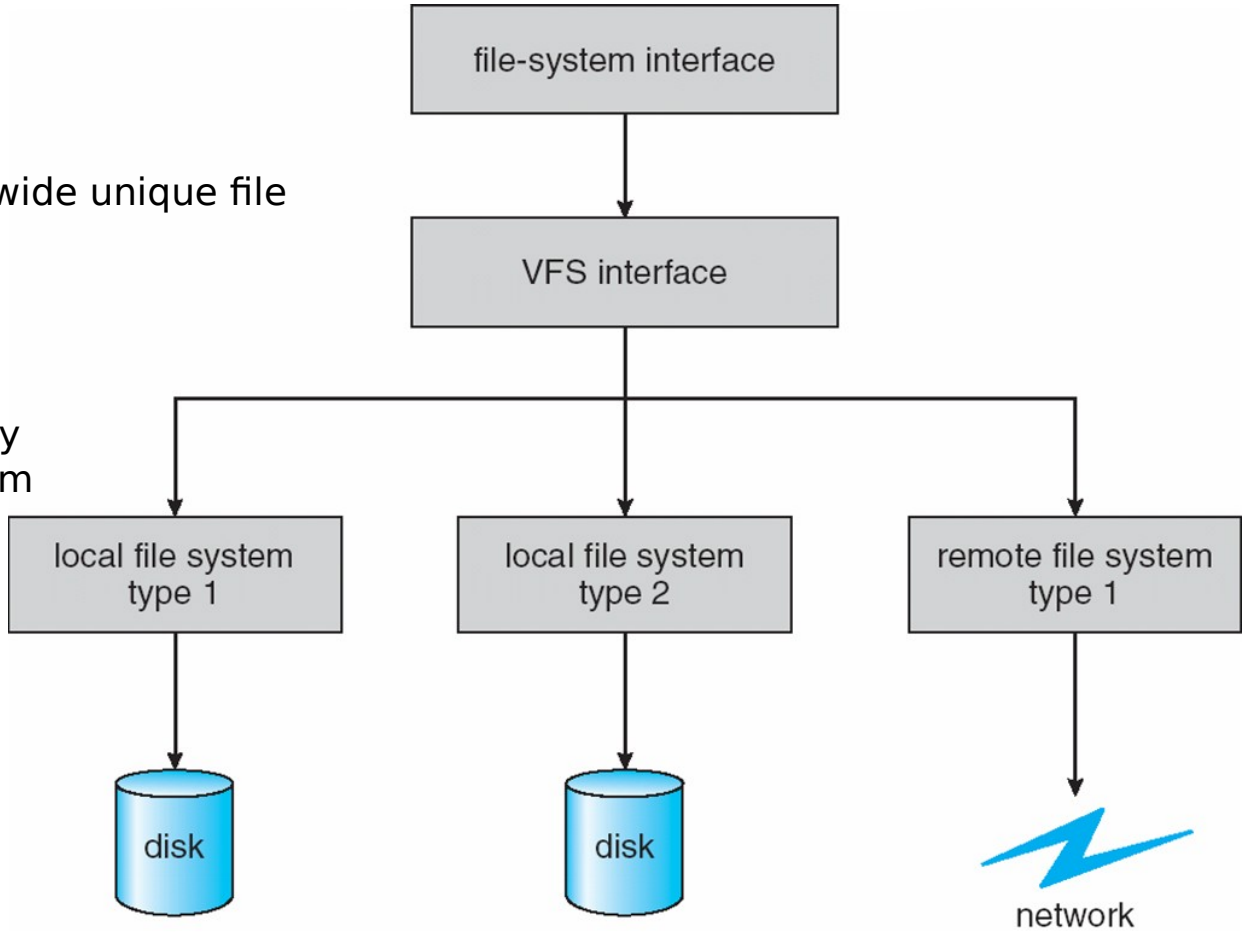
- Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
  - Then dispatches operation to appropriate file system implementation routines
- The API is to the VFS interface, rather than any specific type of file system

Schematic View of Virtual File System

`open(), read(), write(), close()`

`vnode:` network wide unique file

`inode:`
Unique within only
a single file system

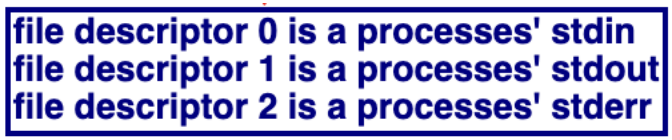


Virtual File System Implementation

- For example, Linux VFS has four object types:
  - Inode object: represents an individual file
  - File object: represents an open file
  - Superblock object: represents entire file system
  - Dentry object: individual directory entry

- VFS defines set of operations on the objects that must be implemented
  - Such as open, close, read, write, ….

File Descriptors

- **Unix I/O: file descriptors**
  - One of the most important resources managed by the kernel is the file
  - A user process and the kernel must agree on names for open file connections. Analogous to the difference between a program and a process, there is a difference between a file and an open "connection" to a file: for example we may have two open connections to the same file, but be in different positions in the file with respect to our next read. So the file name itself isn't really appropriate for communicating which connection you want to read the next byte from. So the OS and the user process refer to each open connection by a number (type int) called a *file descriptor*.
  - Standard i 

ılt to file descriptors 0, 1 and 2

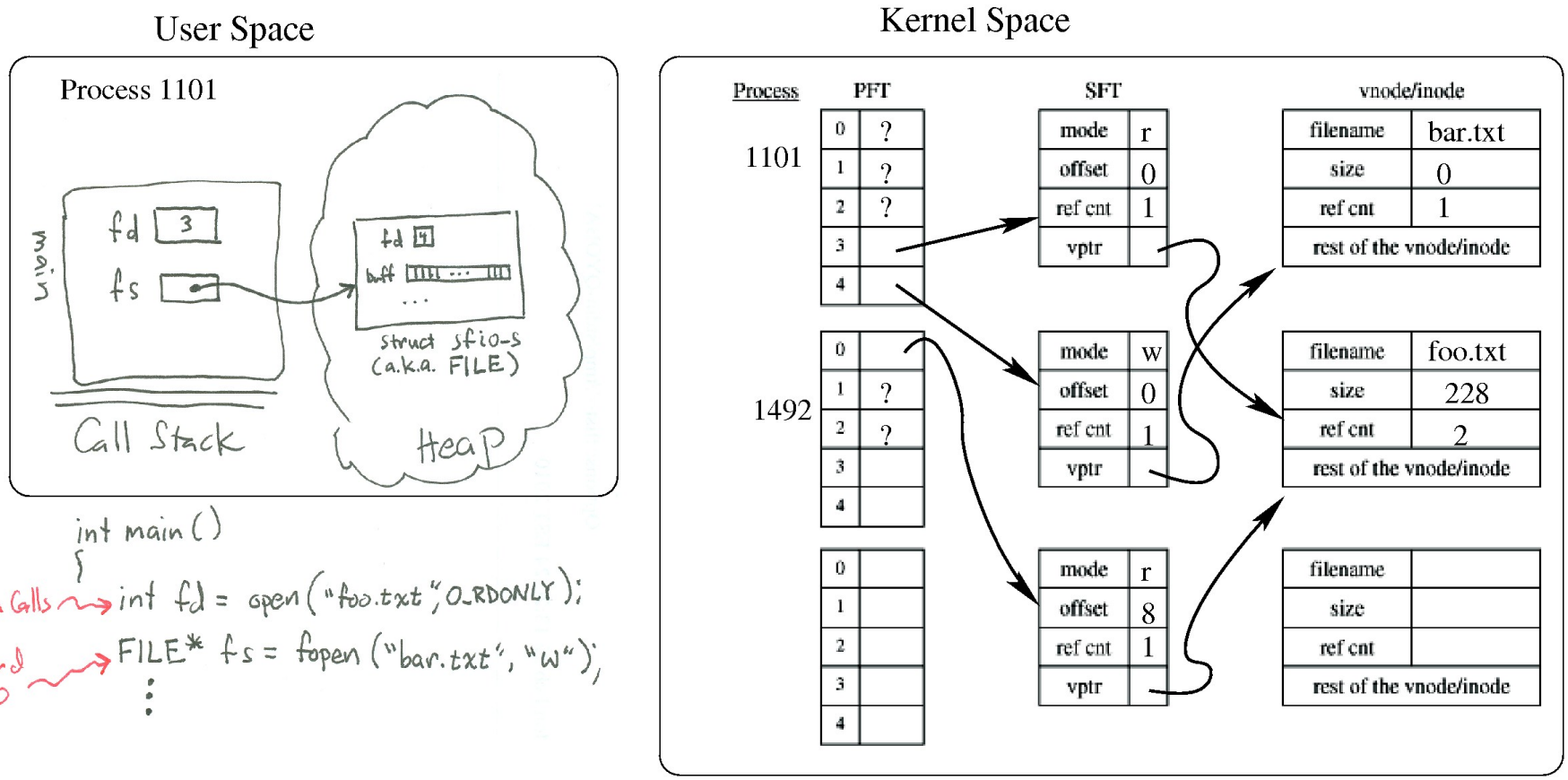
Kernel Space

- **System Open File Table**
  - the kernel keeps a data structure called the *system open-file table* which has an entry for each connection (process-to-file). Each entry contains
    - the connection *status*, e.g. read or write,
    - the *current offset* in the file, and
    - a pointer to a *vnode*, which is the OS's structure representing the file, irrespective of where in the file you may currently be looking.
- **Vnode Table**
  - has an entry for each open file or device.
  - contains information about the type of file and pointers to functions that operate on the file.
  - Typically for files, the vnode also contains a copy of the *inode* for the file, which has "physical" information about the file, e.g. where exactly on the disk the file's data resides.

Physical Drive

- **The physical device: inodes, etc.**
  - a file may be broken up into many data blocks, which may be widely distributed across the physical drive.
  - The *inode* for a file contains the locations of each of the data blocks comprising the file.
  - Directories don't have data blocks, but in a similar fashion have directory blocks, which contain inode/filename pairs; i.e. the names of the files/directories in the directory, along with the inodes for each. Each directory contains entries for "." and ".." --- the current directory and its parent.

System Open File Table



https://www.usna.edu/Users/cs/wcbrown/courses/IC221/classes/L09/Class.html

Reference Counts

- **Several file descriptors** may actually refer to the same system open-file table entries. That entry in the system open-file table can't be removed until *all* of those referencing file descriptors have been closed.
- **Several system open-file table entries** may actually refer to the same vnode table entry. That vnode table entry cannot be removed from the vnode table until *all* of those referencing system open-file table entries have been removed.
- A file may be referenced by several entries in the file system (this comes from "**hard links**", which you can create with the *ln* utility) and the file cannot be removed from the filesystem until *all* of those references to the file have been removed.

Allocation Methods

- Memory is divided into pages, similarly disk is divided in blocks
- An allocation method refers to how disk blocks are allocated for files so that disk space is utilized effectively and files can be accessed efficiently:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

Basic Features: Distributed File Systems (DFS)

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

Data Characteristics

- Applications need streaming access to data
- Batch processing rather than interactive user access.
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
  - Tens of millions of files in a single instance
  - Large data sets and files: gigabytes to terabytes size

- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency
- A map-reduce application or web-crawler application fits perfectly to this model.

Motivation Questions

- **Problem 1:** Data is too big to store on one machine.

- **HDFS:** Store the data on multiple machines!

Motivation Questions

- **Problem 2:** Very high-end machines are too expensive

- **HDFS:** Run on commodity hardware!

Motivation Questions

- **Problem 3:** Commodity hardware will fail!

- **HDFS:** Software is intelligent enough to handle hardware failure!

Fault tolerance

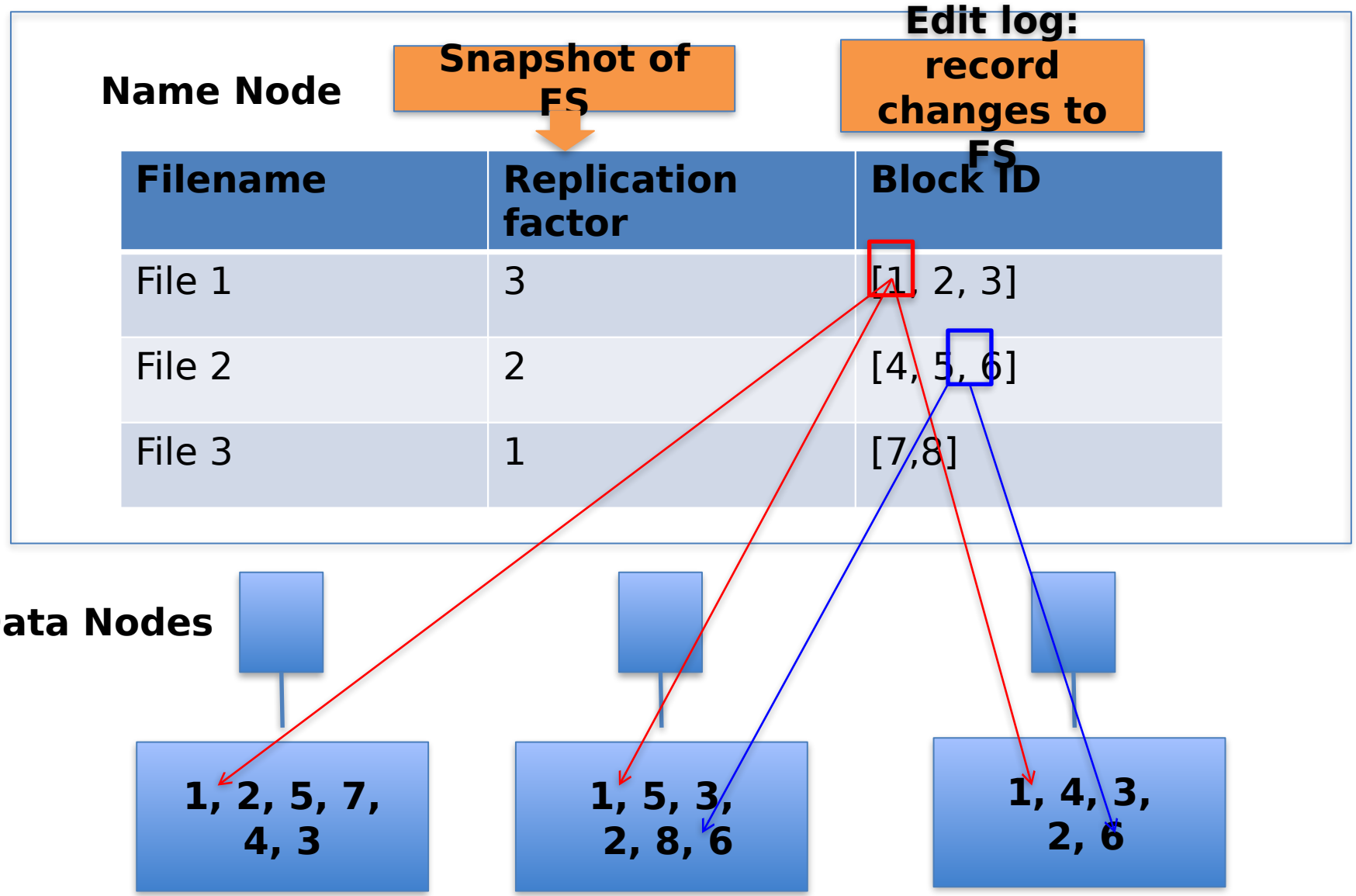
- Failure is the norm rather than exception
- An DFS instance may consist of thousands of server machines, each storing part of the file system’s data.
- Since we have a huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of DFS.

Motivation Questions

- **Problem 4:** What happens to the data if the machine stores the data fails?

- **HDFS:** Replicate the data!

Replication

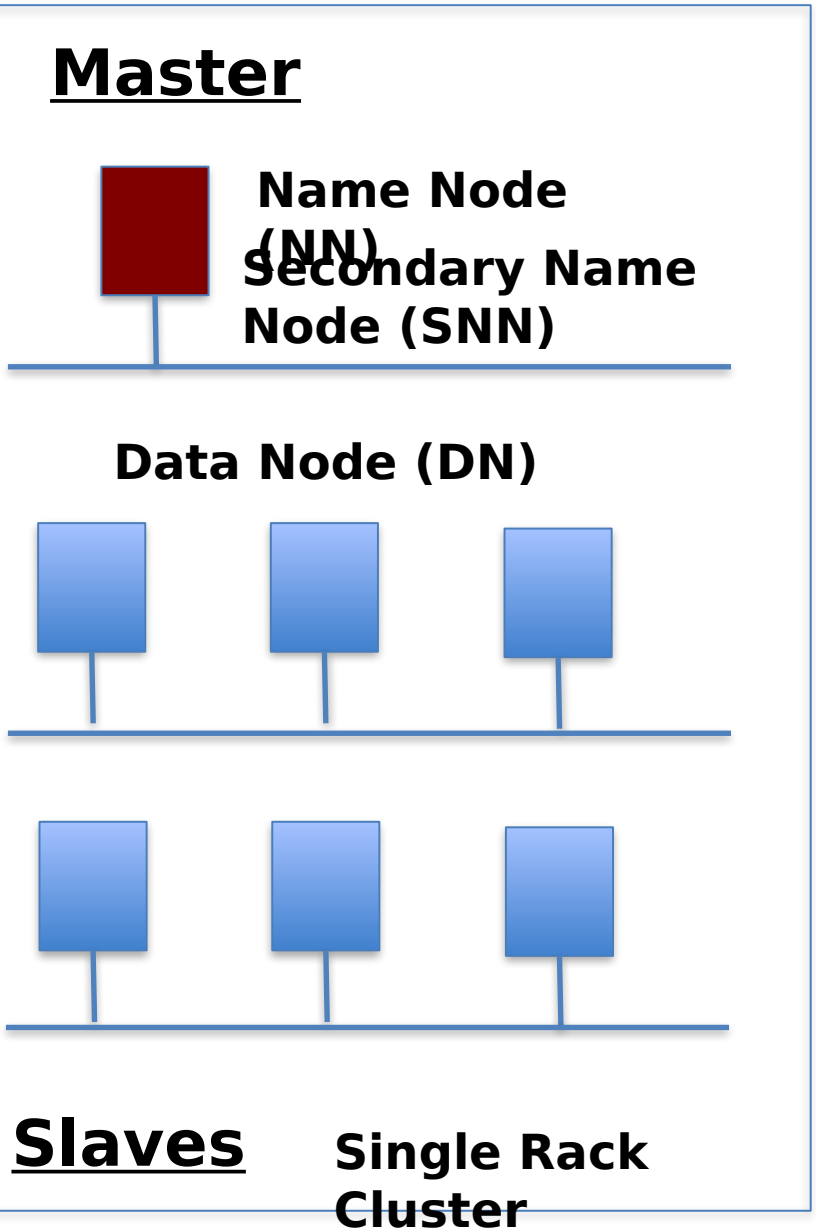


Motivation Questions

- **Problem 5:** How can distributed machines organize the data in a coordinated way?

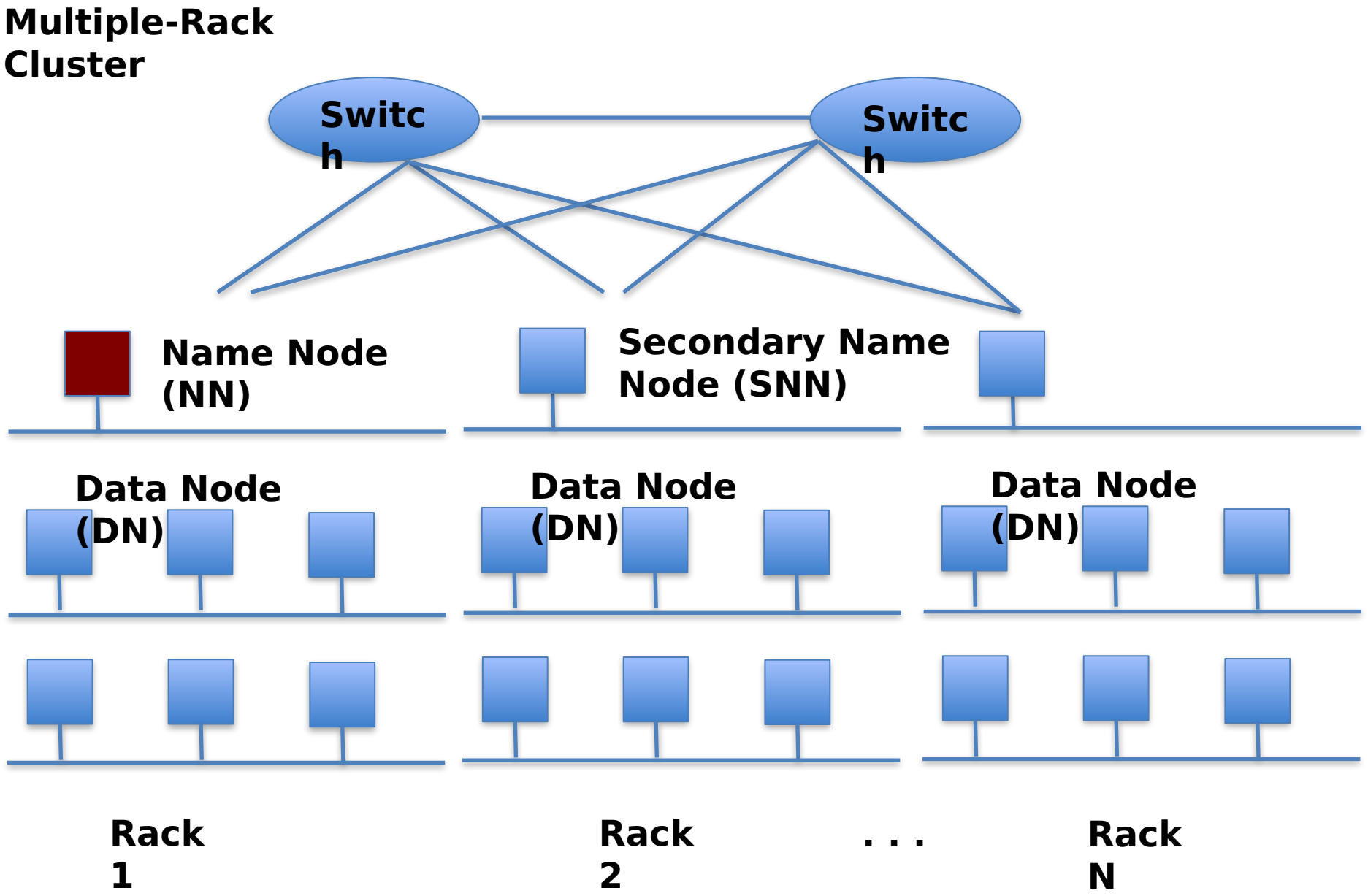
- **HDFS:** Master-Slave Architecture!

HDFS Architecture: Master-Slave

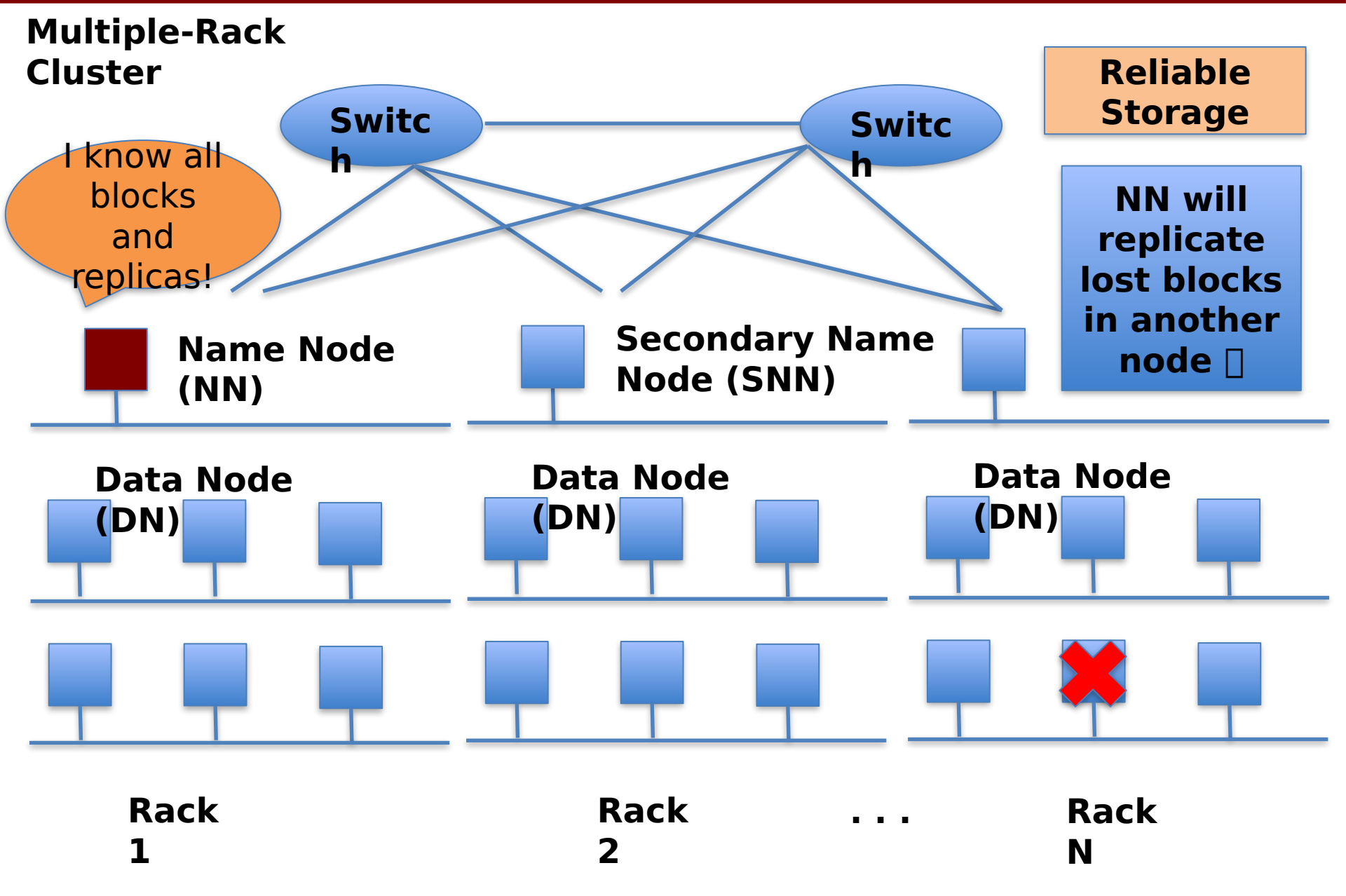


- Name Node: Controller
  - File System Name Space Management
  - Block Mappings
- Data Node: Work Horses
  - Block Operations
  - Replication
- Secondary Name Node:
  - Checkpoint node

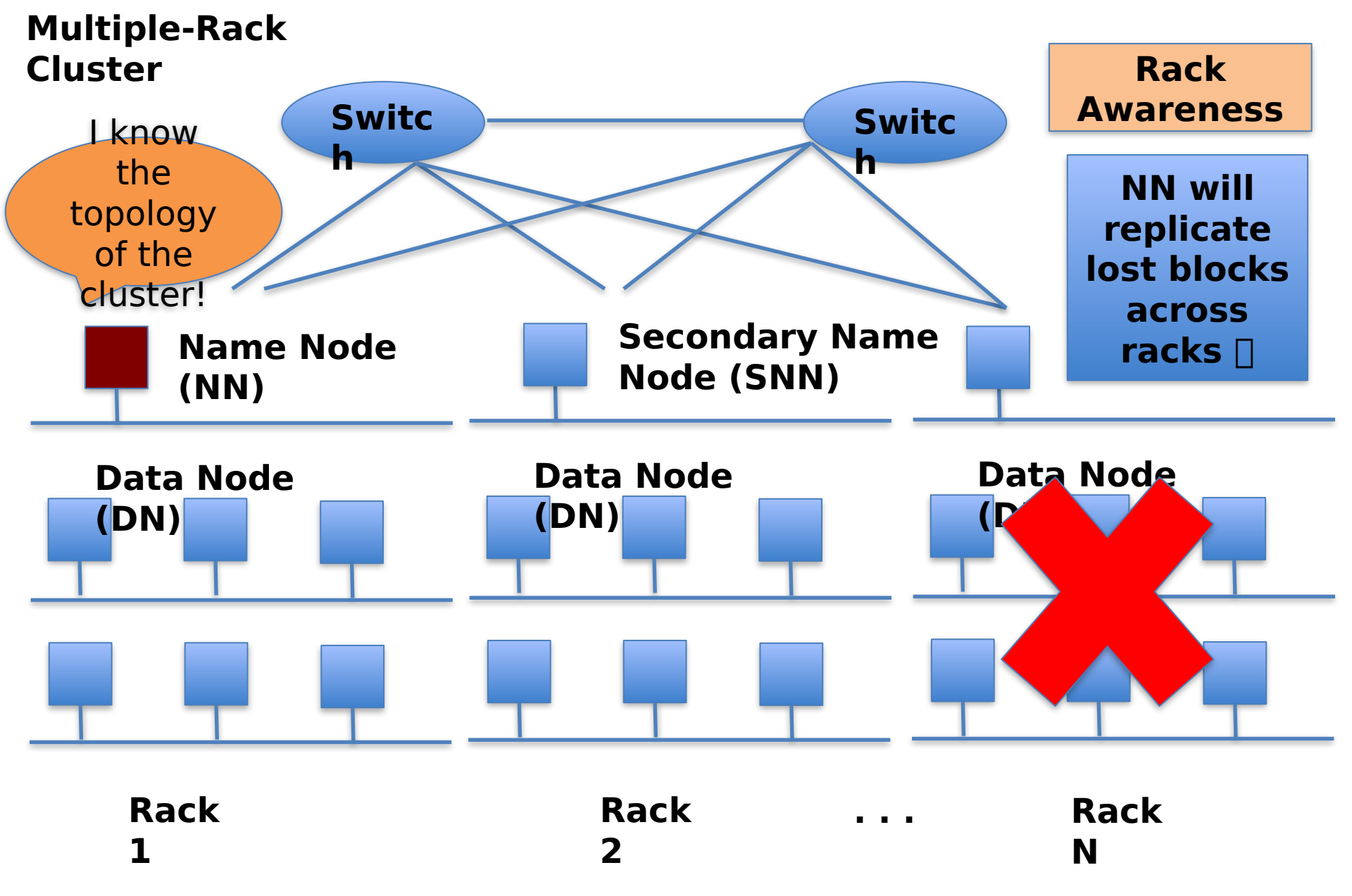
HDFS Architecture: Master-Slave



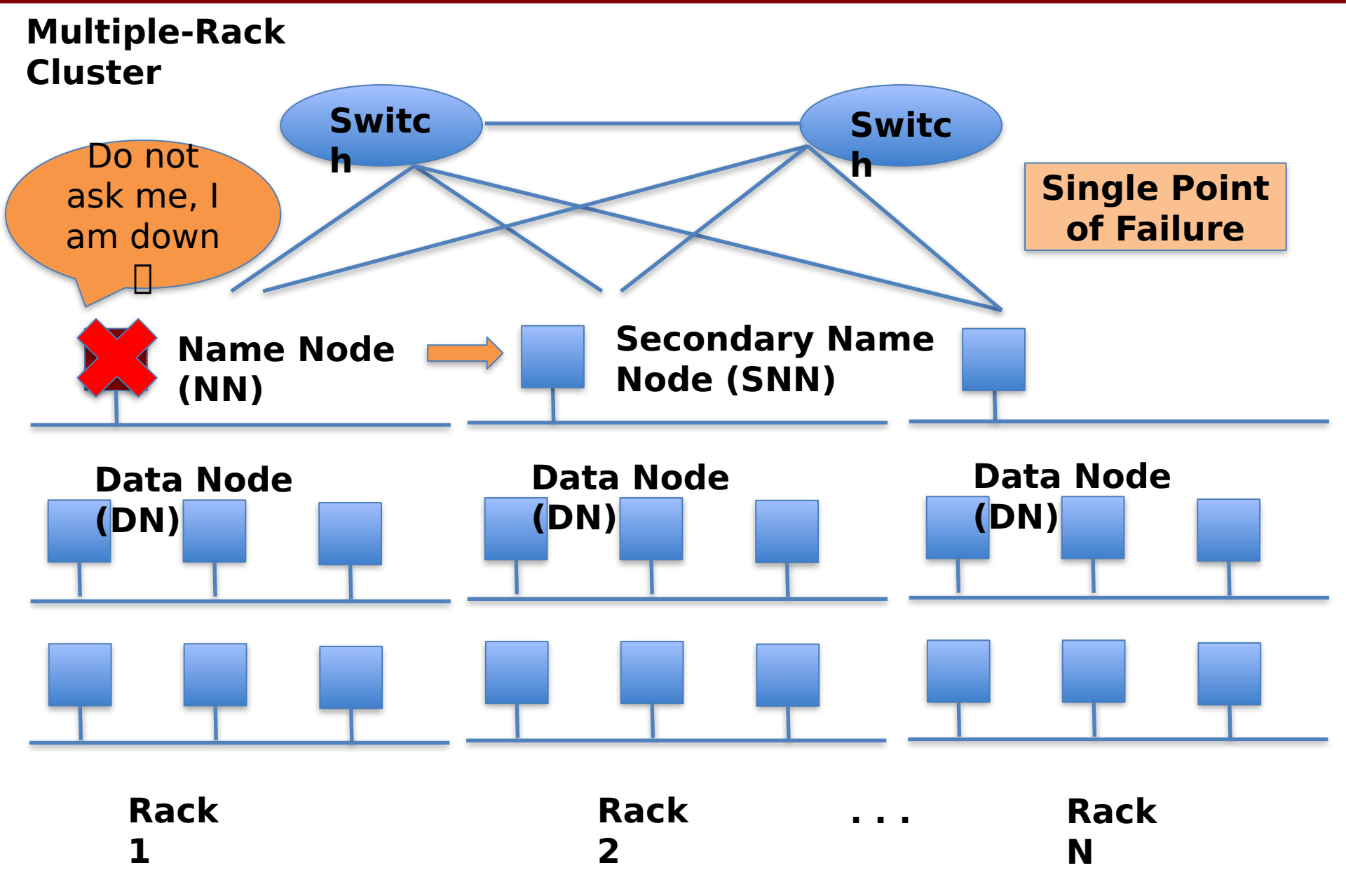
HDFS Architecture: Master-Slave



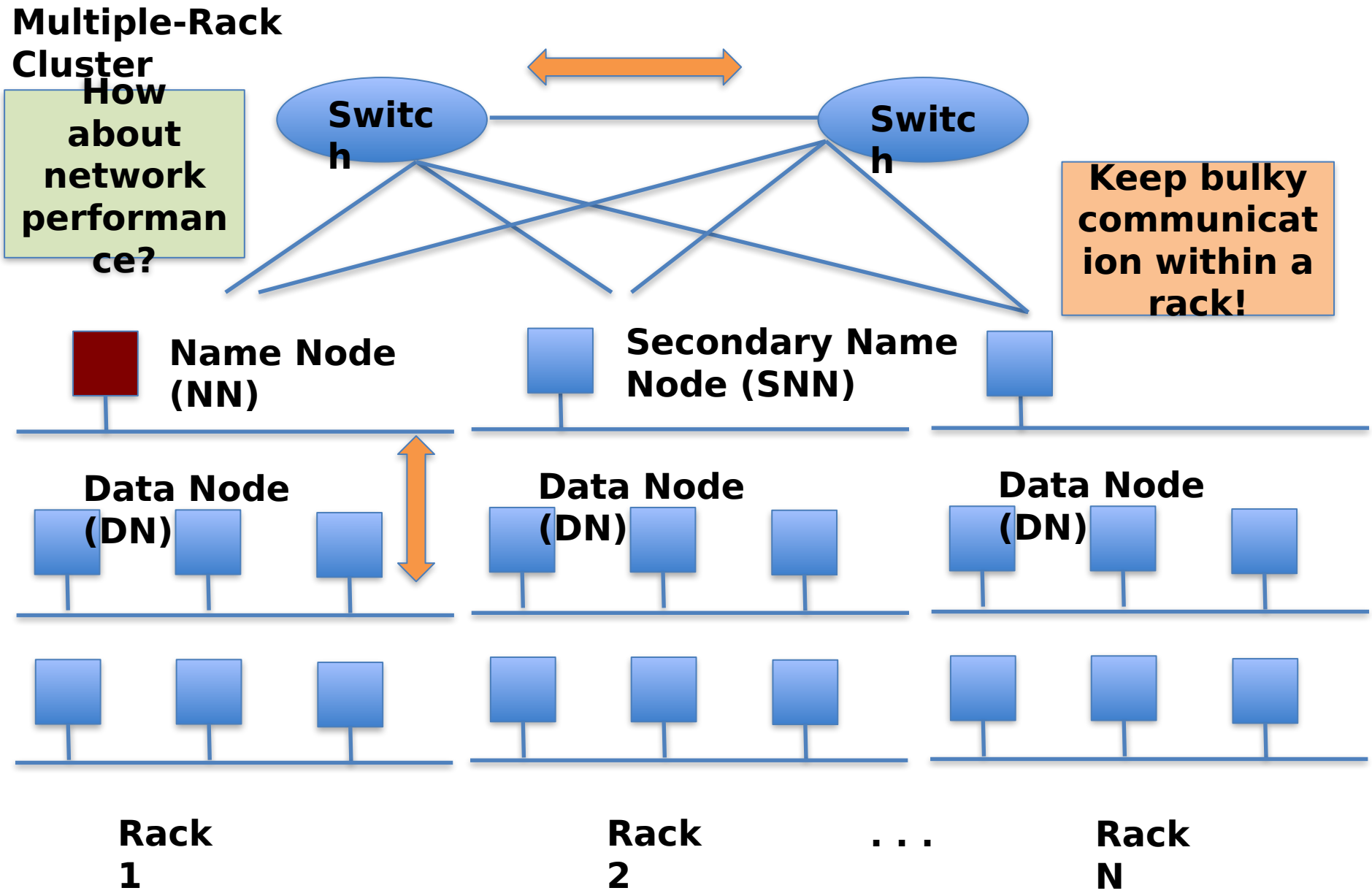
HDFS Architecture: Master-Slave



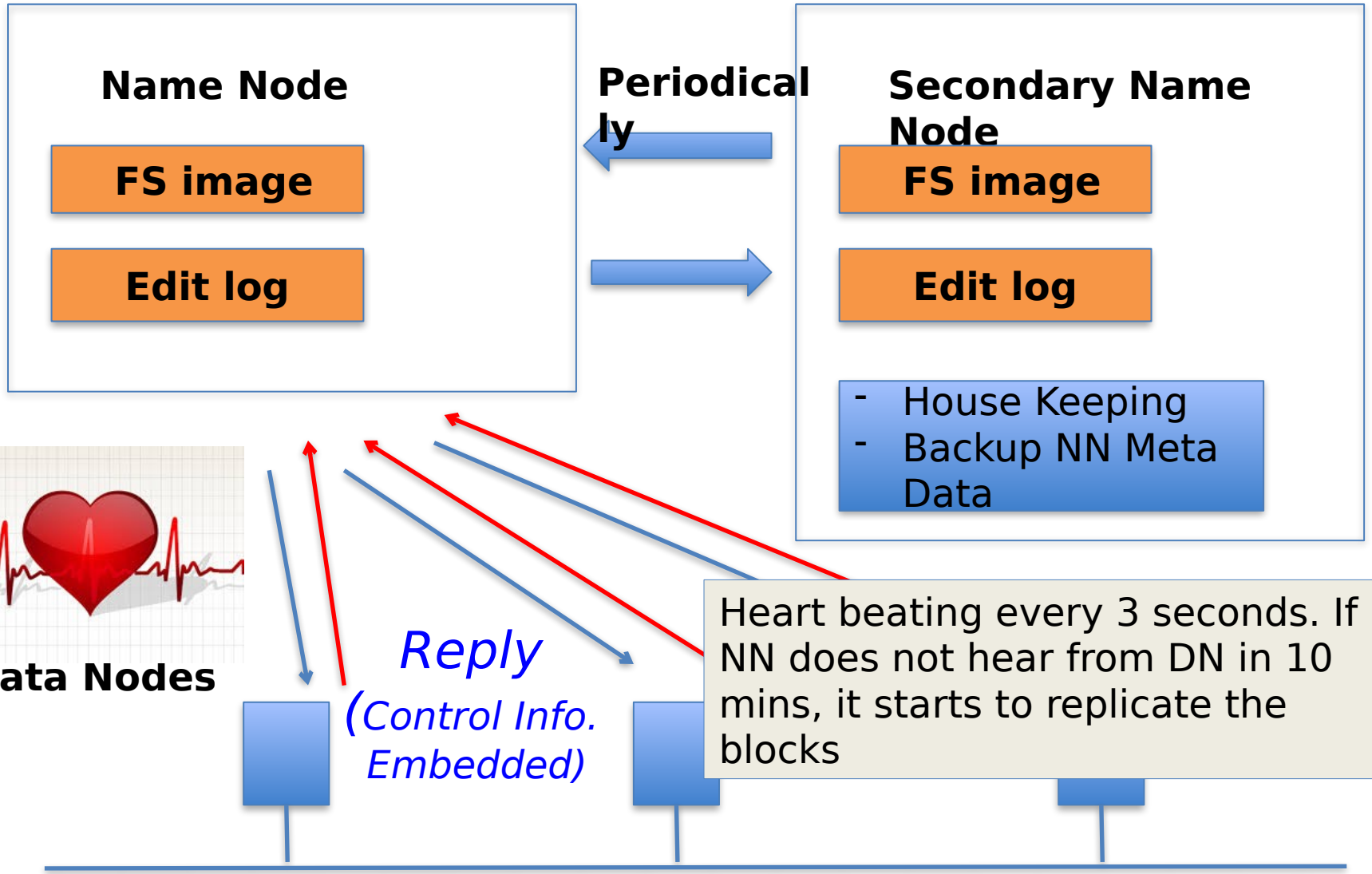
HDFS Architecture: Master-Slave



HDFS Architecture: Master-Slave



HDFS Architecture: Master-Slave



HDFS Inside: Blocks

- Q: Why do we need the abstraction "Blocks" in addition to "Files"?
- Reasons:
  - Files can be larger than a single disk
  - Block is of fixed size, easy to manage and manipulate
  - Easy to replicate and do more fine-grained load balancing