IOWA STATE UNIVERSITY

Department of Electrical and Computer Engineering

# Lecture 35:
# Midterm 2 Review

# Basic Information

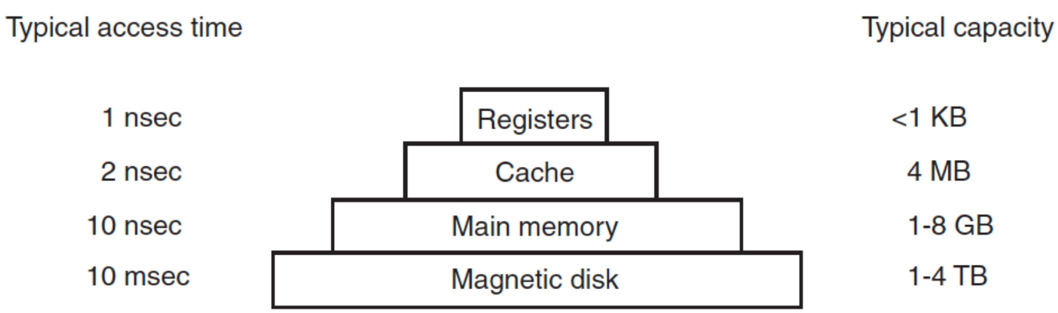
- Time
  - 09:00-09:50 am, Nov 22 (Friday)
- Location
  - Marston 2300
- Format
  - Similar to Midterm 1
  - Closed book/notes
- Scope
  - Lecture 17 to Lecture 34

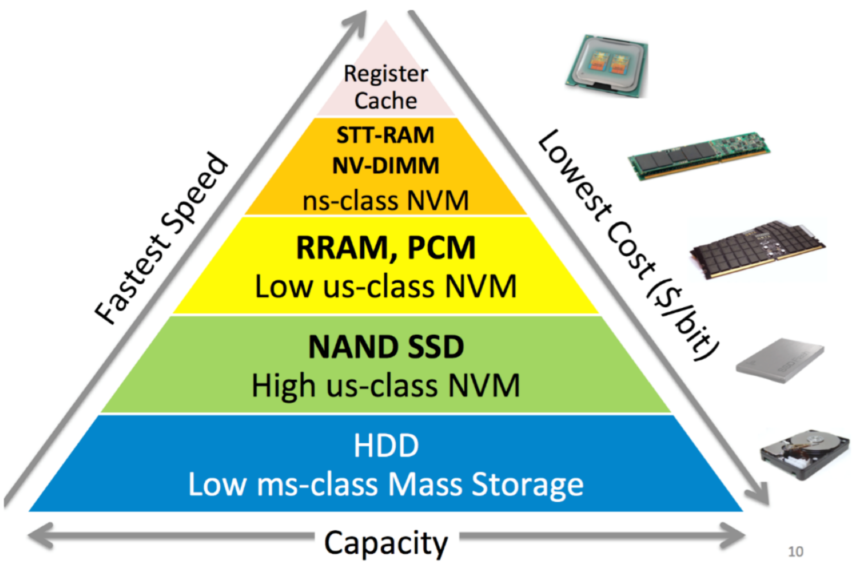
# Review

- Memory Management
  - Memory Hierarchy & Address Space
  - Free Space Management
  - Paging & TLB
  - Swapping
  - Page Replacement Algorithms
- I/O & Storage Management
  - HDD & SSD
  - File systems

# Review

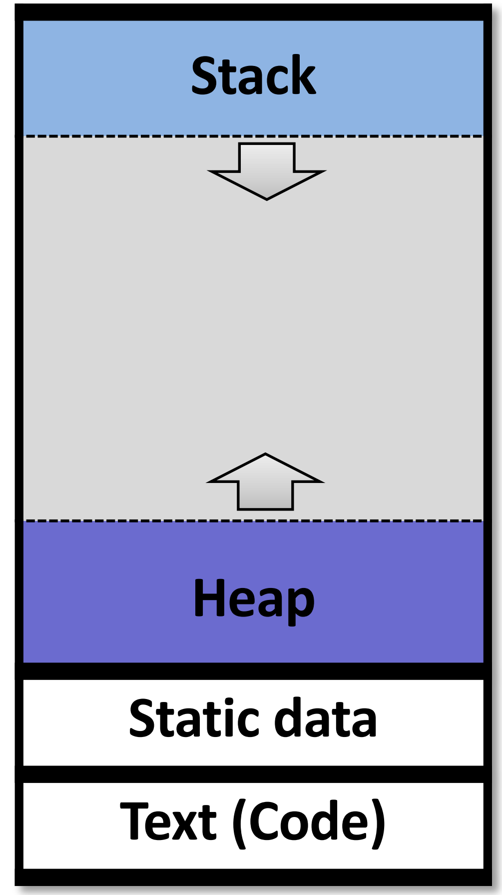
- Memory Hierarchy
  - diverse technologies with tradeoffs
    - latency, capacity, persistency, cost, …
    - non-volatile memories are revolutionizing the market!
      - A “disruptive” technology



# Review

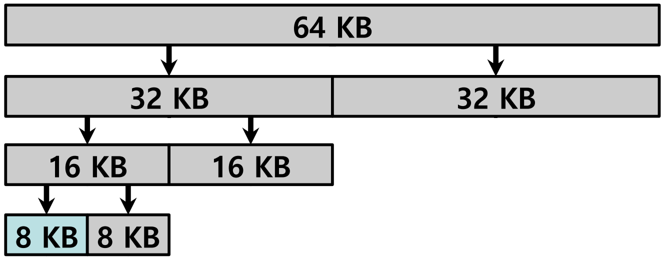
- Virtual Memory
  - An illusion that each process uses the whole memory itself
  - Improve memory efficiency, isolation & protection
- Address space
  - An abstraction of physical memory for a process
  - the set of all virtual addresses visible to a program




Address Space

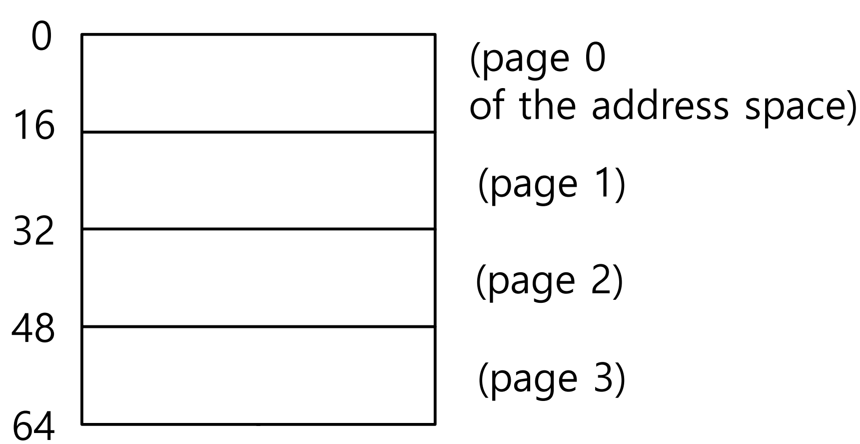
# Review

- Free-Space Management
  - Bitmap
  - Linked list
  - Segregated List
    - Keeping free chunks of popular sizes in separate lists
    - **Slab allocator**
  - Buddy Allocation
    - divides free space by two until a block that is big enough to accommodate the request is found
      - can suffer from **internal fragmentation**.
      - makes **coalescing** simple.

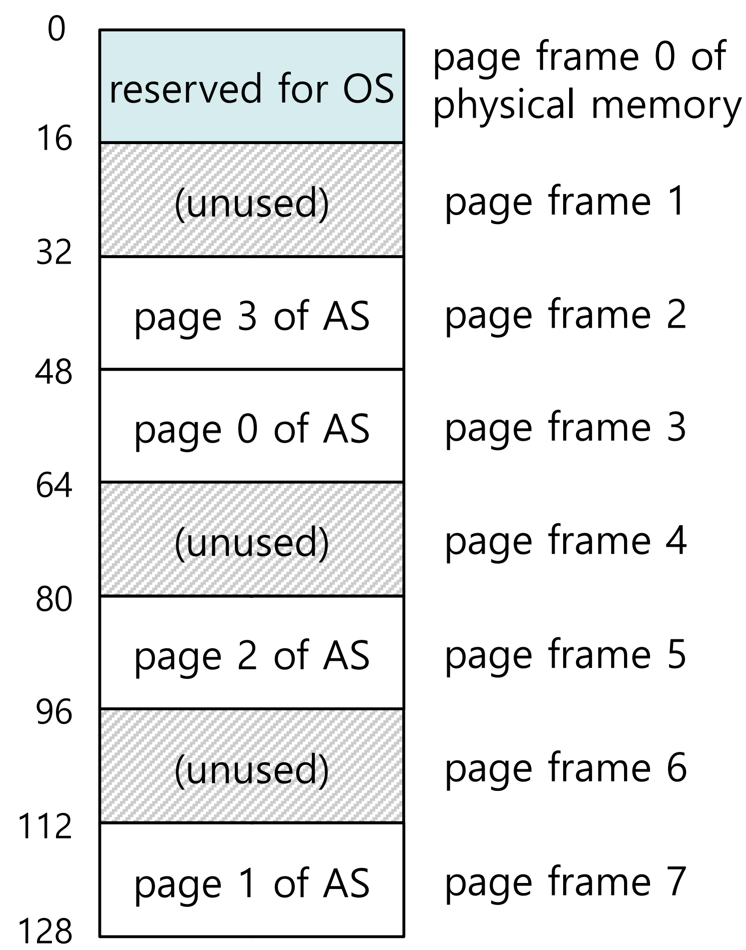


# Review

- Paging
  - split up address space into fixed-size units called ***pages***
    - physical memory is also split into fixed-size units called ***page frames***
    - Flexibility & simplicity



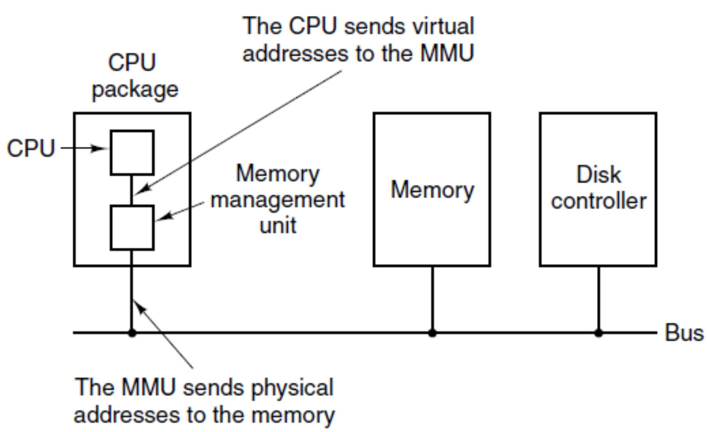

**64-Byte Address Space**

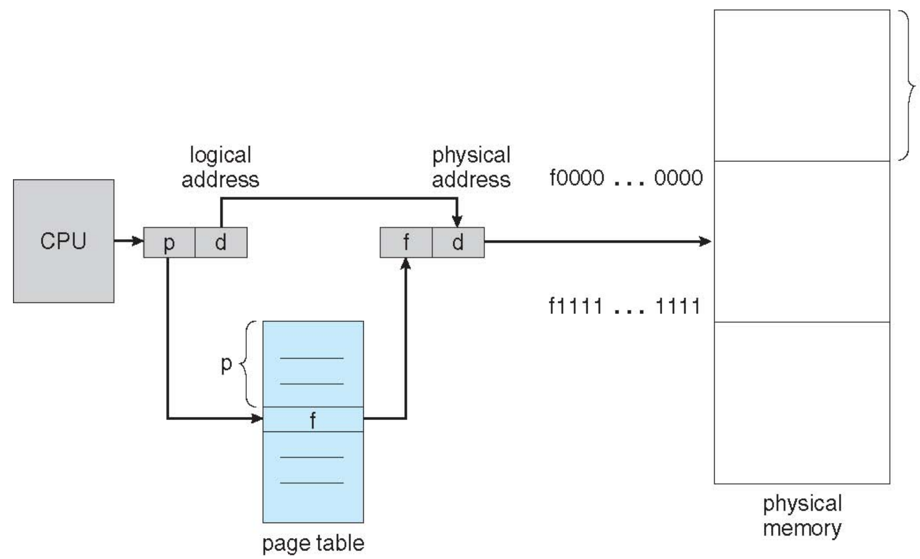



**128-Byte Physical Memory**

# Review

- Address Translation via **Page Table**
  - Each virtual address is divided into two parts:
    - VPN: virtual page number (p)
      - used as an index into the **page table**
    - Offset: offset within the page (d)
  - Hardware involved: MMU



# Review

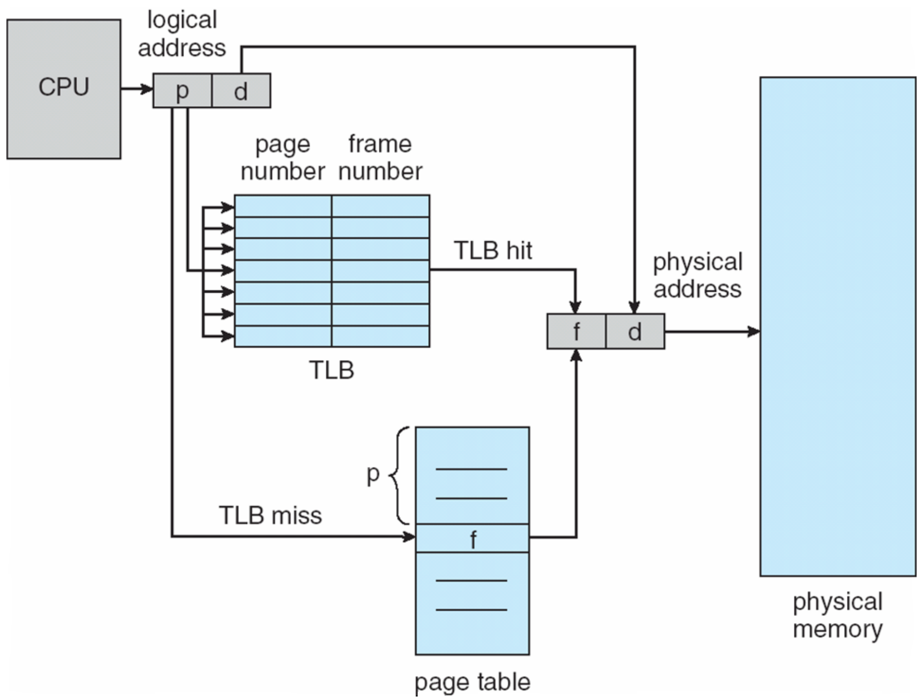
- Page Fault
  - Requested page not in the physical memory
    - MMU reports a page fault to CPU
    - CPU gives control to the OS (page fault handler routine)
    - OS fetches a page from the disk
      - May needs to evict an existing page from memory
    - Instruction is restarted
- Potential Issues of Paging
  - Time
    - for every memory access, one additional access is needed
  - Space
    - A page table can get awfully large
    - Each process needs to have a page table

# Review

- Space Overhead of Page Tables
  - A (linear) page table can be large
    - Example (revisit):
      - 32-bit address space (4GB) with 4KB pages
      - 12 bits for offset within a page (4K=2^12)
      - 20 bits for VPN (32 – 12 = 20)
      - $4MB$= 2^20 $entries$ $*$4 $Bytes\ per\ page\ table\ entry$
  - Each process needs to have a page table
    - 100 processes needs 4MB * 100 = 400MB

# Review

- Translation Lookaside Buffer (TLB)
  - Hardware cache for speeding up paging
  - VPN, PFN, ASID, etc
  - Address translation with TLB & page table

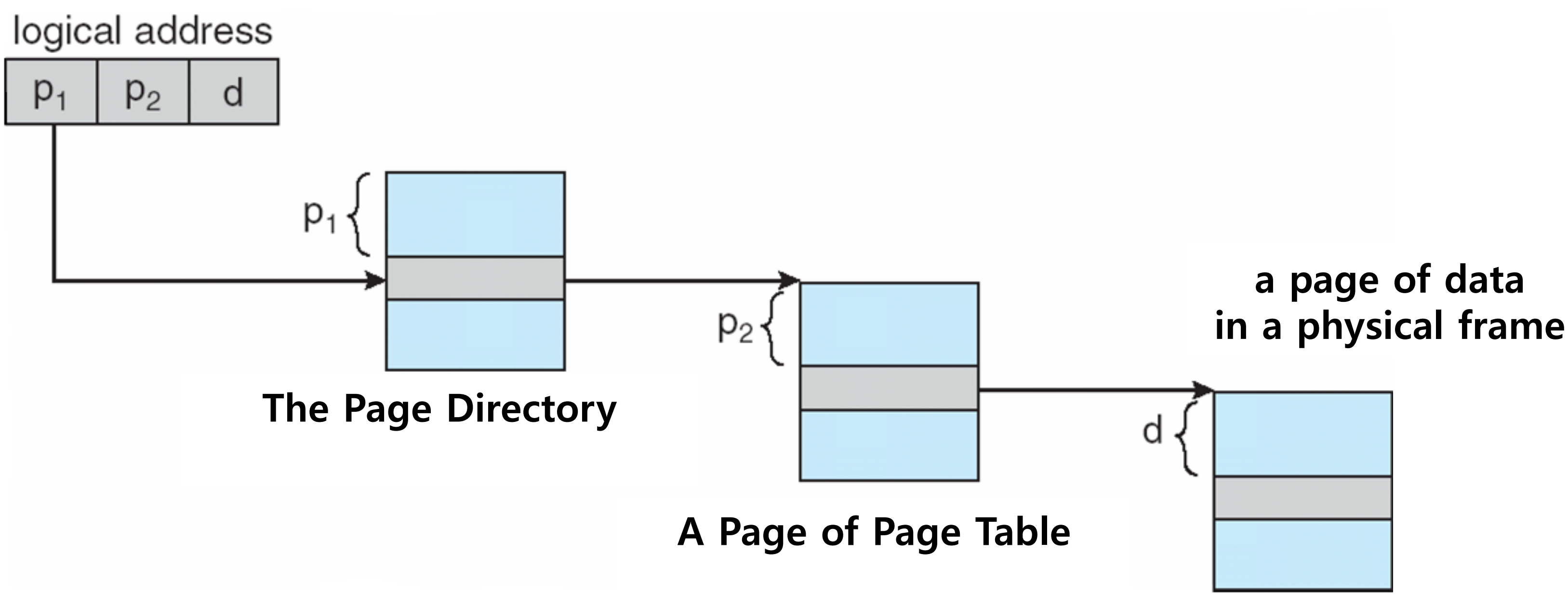


# Review

- TLB may improve performance greatly
  - Effective Access Time (EAT)
    - effective time for accessing memory with TLB
    - TLB Hit ratio = $\alpha$
      - percentage of times that a page number is found in the TLB
      - TLB hit: one memory access
      - TLB miss: two memory accesses
    - Consider $\alpha$ = 80%, 100ns for each memory access
      - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
    - Consider a more realistic hit ratio $\alpha$ = 99%; still 100ns for each memory access
      - EAT = 0.99 x 100 + 0.01 x 200 = 101ns
  - Temporal & spatial locality

# Review

- Multi-Level Page Tables
  - Paging the page table itself
    - e.g., a two-level page table
      - the page directory index (p1) is used to identify a page directory entry (PDE) in the page directory
      - the page table index (p2) is used to identify a page table entry



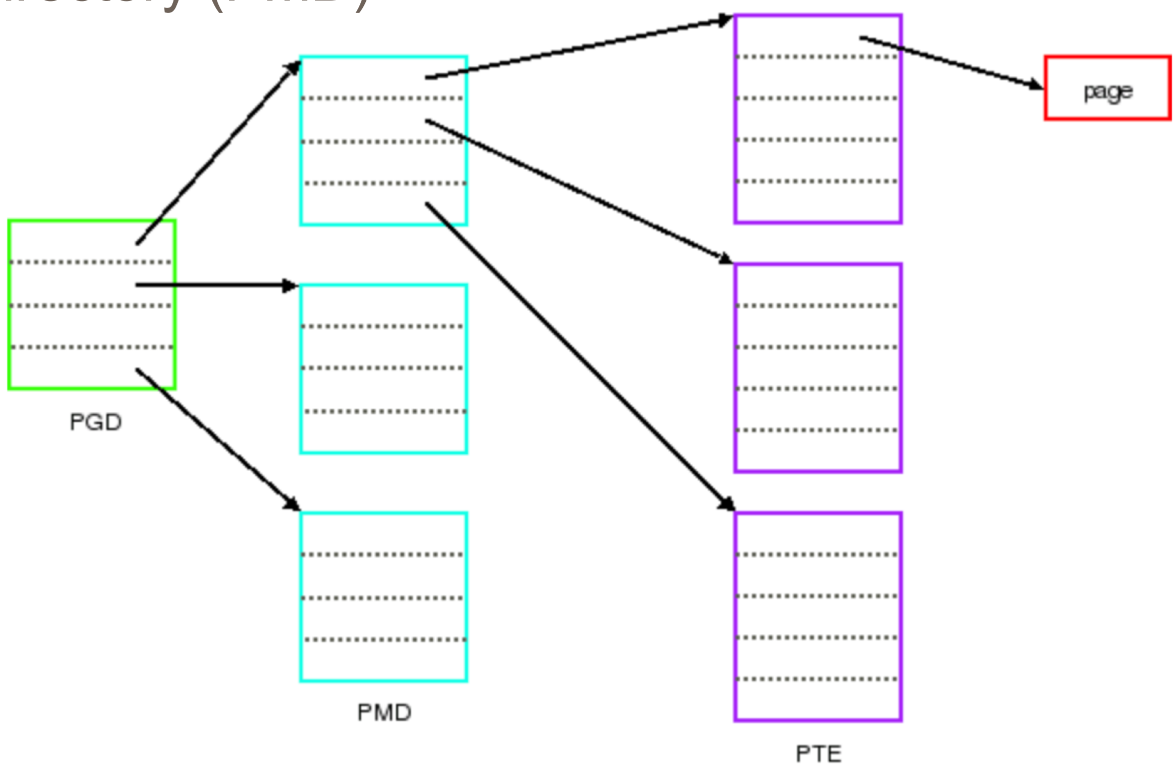
# Review

- Multi-Level Page Tables (cont')
  - E.g., assume a 64-bit address space
    - Assume page size is 4 KB ($2^{12}$)
      - then a single-level page table has $2^{52}$ entries
    - In a two-level page table
      - a page (4KB) of page table has $2^{10}$ 4B entries
        - 10 bits for indexing into the page of page table
      - the page directory has $2^{42}$ 4B entries

| $p_1$ | $p_2$ | $d$ |
|---|---|---|
| 42 | 10 | 12 |

# Review

- Multi-Level Page Tables (cont')
  - E.g., a three-level page table
    - page global directory (PGD)
    - page middle directory (PMD)
    - page table

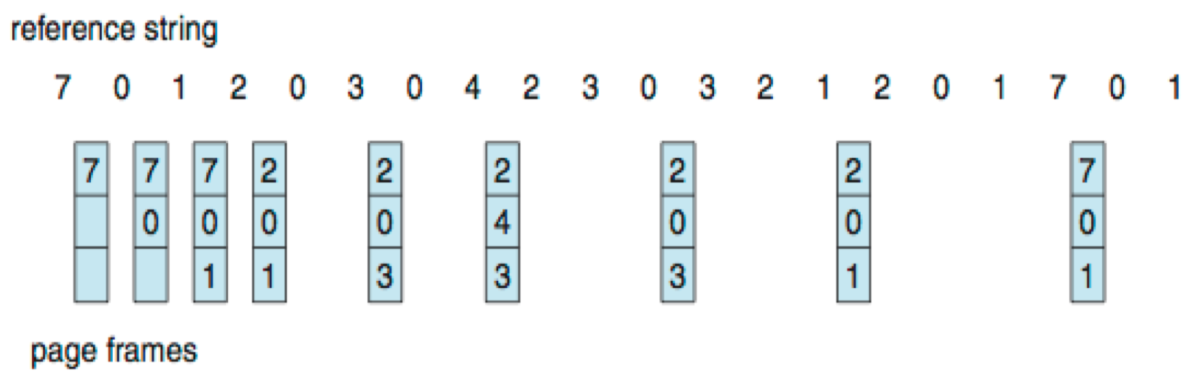


# Review

- Swapping
  - stash away portions of address space that currently aren't in great demand
  - the unpopular pages are placed in the **swap space** on disk
- Page Replacement Algorithms
  - When free physical memory is low, page replacement algorithms decide which page to evict
    - the physical memory serves as a cache of the swap space
      - a cache miss leads to a page fault
    - the goal is to minimize the number of cache misses/page faults

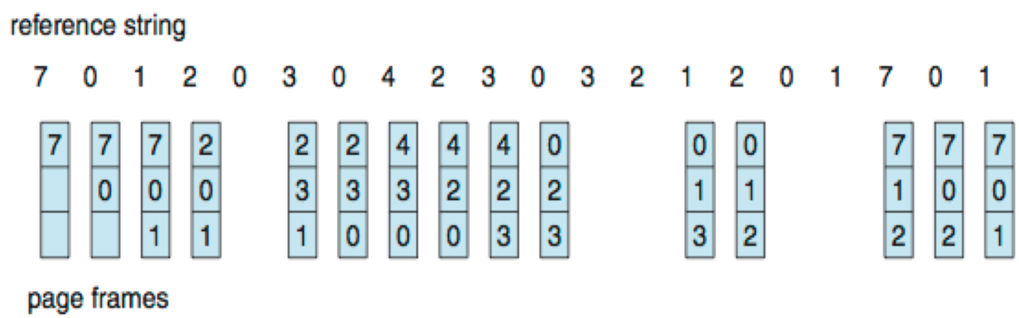
# Review

- The Optimal Algorithm
  - Evict the page that will be accessed furthest in the future
  - Example
    - 20 references in sequence
    - 3 page frames
    - # of page fault:  9



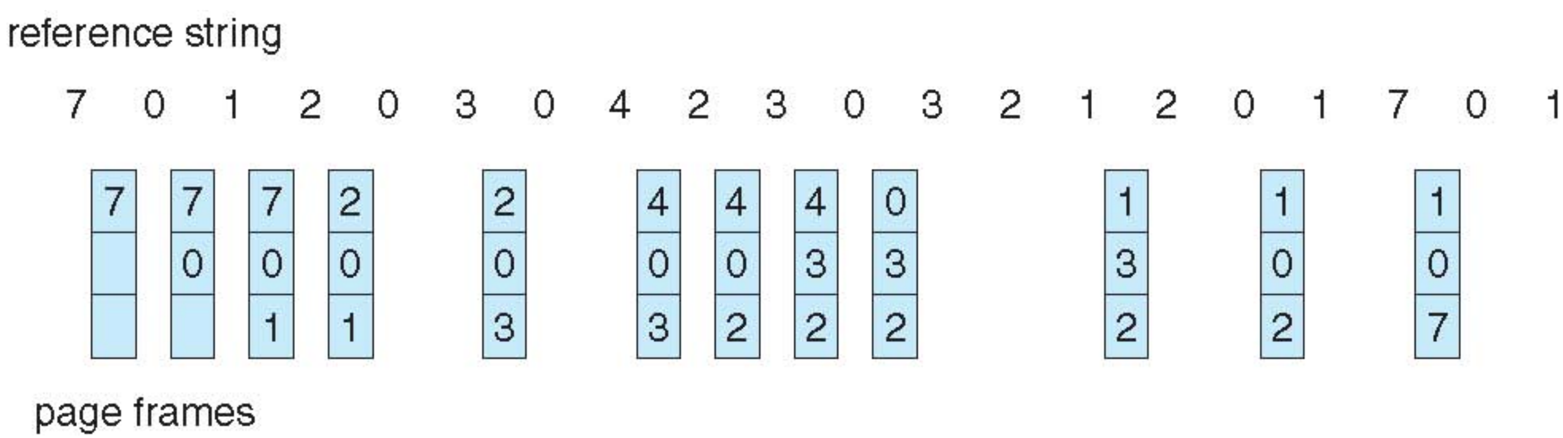
# Review

- First In First Out (FIFO) Algorithm
  - Pages were recorded in a FIFO queue; the page on the head of the queue (the “**First-in**” page) is evicted first
  - Example
    - 20 references in sequence
    - 3 page frames
    - # of page fault: 15



# Review

- Least-Recently-Used (LRU) Algorithm
  - Evict the least-recently-used page
  - Example
    - 20 references in sequence
    - 3 page frames
    - # of page fault: 12



# Review

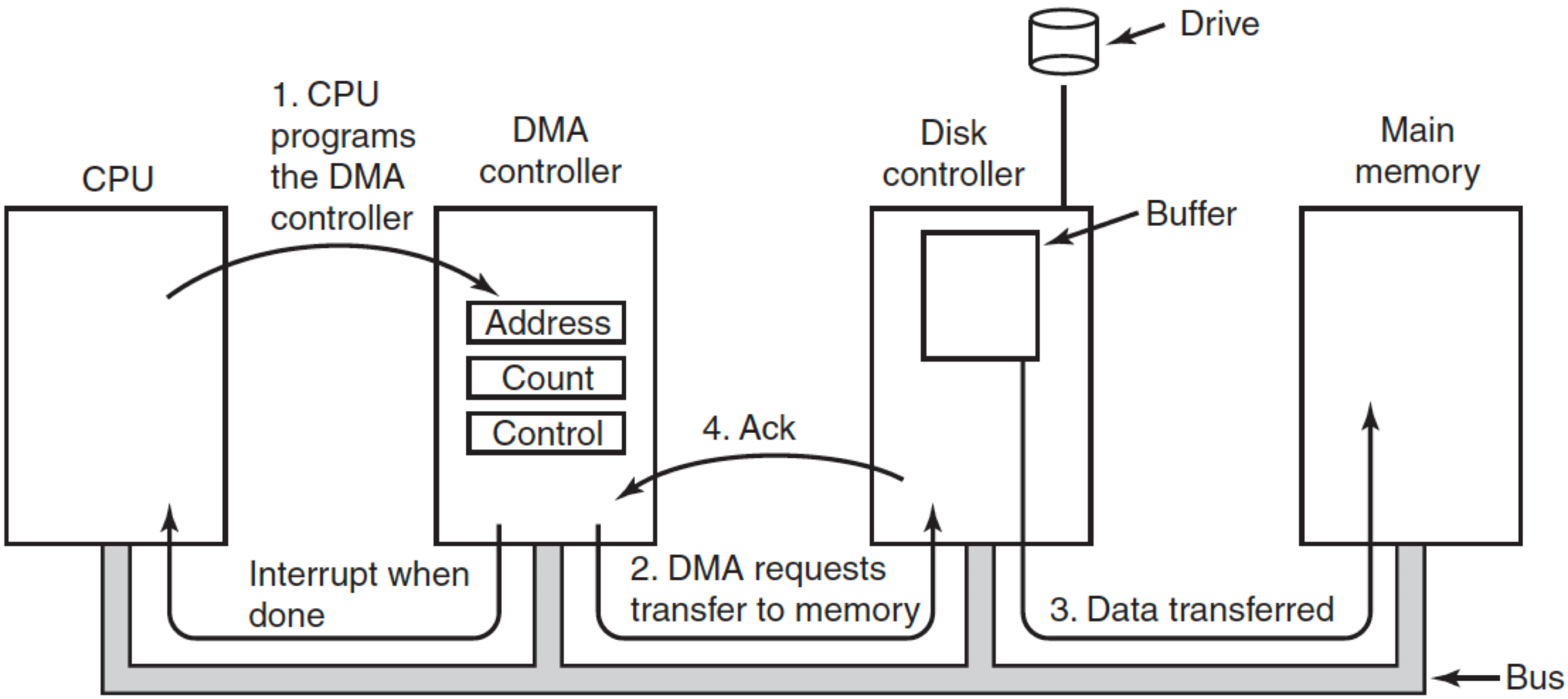
- ~~Memory Management~~
  - ~~Memory Hierarchy & Address Space~~
  - ~~Free Space Management~~
  - ~~Paging & TLB~~
  - ~~Swapping~~
  - ~~Page Replacement Algorithms~~
- I/O & Storage Management
  - HDD & SSD
  - File systems

# Review

- I/O Devices
  - Two basic types
    - Block devices
      - stores information in fixed-size blocks, each one with its own address
      - All transfers are in units of one or more entire (consecutive) blocks
        - can read or write each block independently of all the other ones
      - E.g., Hard disks, CDROM, USB
    - Character devices
      - delivers or accepts a stream of characters (bytes), without any block structure
      - not addressable and does not have any seek operation
      - E.g., printer, network interface card (NIC)

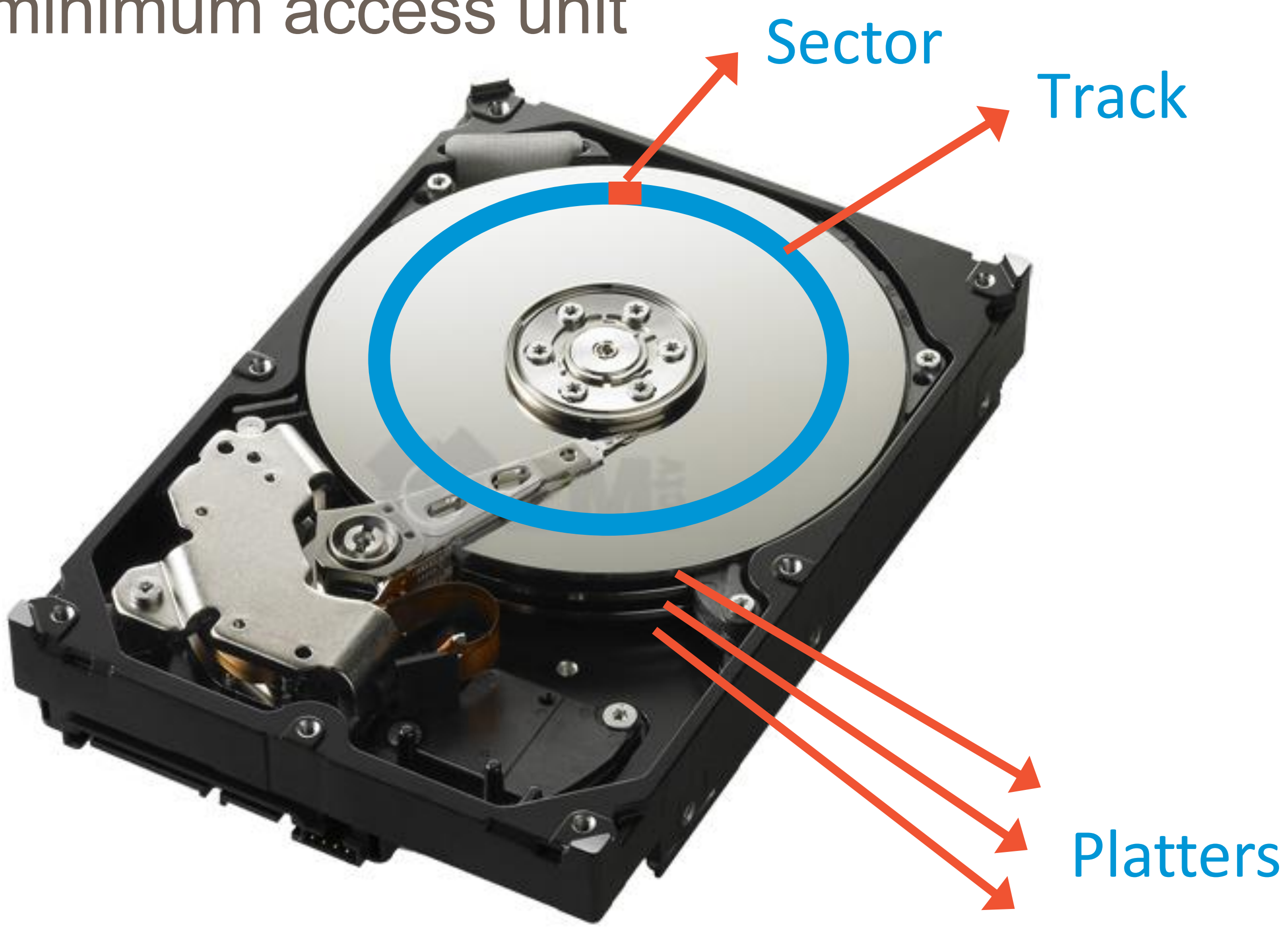
# Review

- Direct Memory Access (DMA)
  - transfer data between memory & I/O device without involving CPU



# Review

- Hard Disk Drive (HDD)
  - a sector is the minimum access unit
    - e.g., 512B



# Review

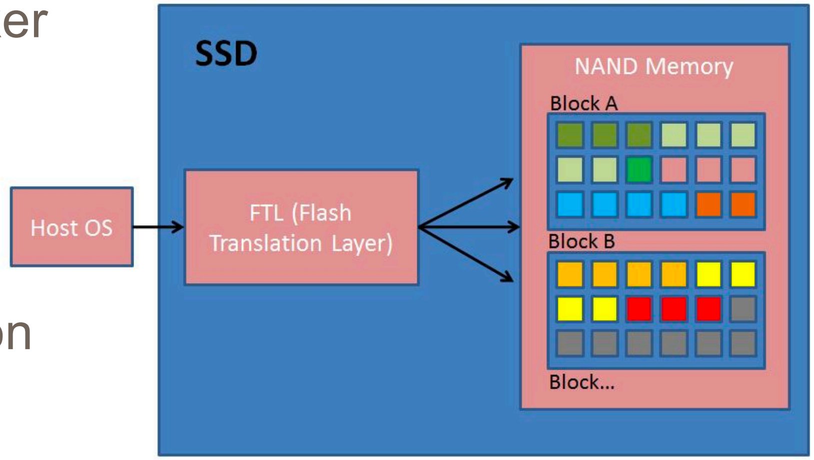
- HDD I/O Time & I/O Rate
  - I/O time ($T_{I/O}$) includes three parts
    - **Seek**
    - **rotational delay**
    - **Transfer**

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

  - I/O rate ($R_{I/O}$): $$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$$
  - Favor sequential workloads

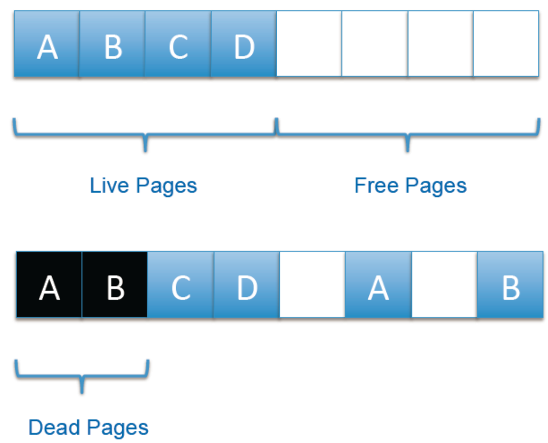
# Review

- Solid State Drives (SSDs)
  - Flash Memory
    - **SLC vs MLC**
      - MLC is used in consumer marker
    - **NOR** vs **NAND**
      - NAND is used in SSDs
    - **Block**
      - minimum unit of **erase** operation
      - contain multiple pages
    - **Page**
      - minimum unit of **program** operation
    - each cell can only stand a limited number of program/erasure cycles (**P/E cycles**)



# Review

- Solid State Drives (SSDs)
  - Flash Translation Layer (FTL)
    - Logical block mapping
      - maps logical addresses to physical addresses
      - maintains a mapping table
      - out-of-space update (append-only)
    - Garbage Collection
      - re-cycle invalid pages
      - source of I/O instability
    - Wear leveling
      - let the flash cells be erased/programmed about the same number of times

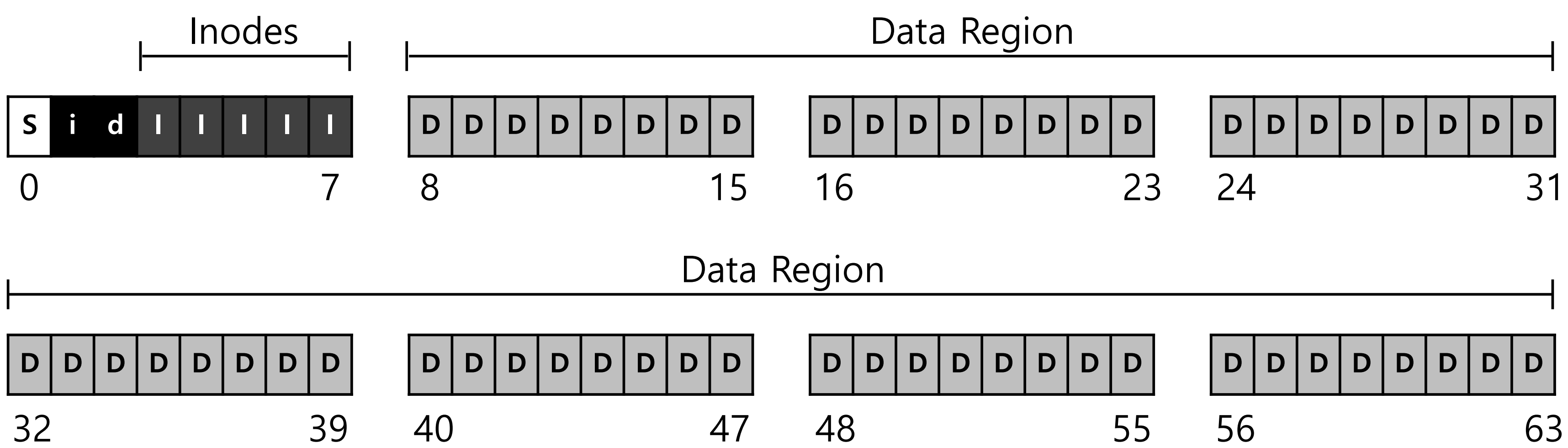


# Review

- File Systems
  - File: contains user data
    - A **file system** (FS) is responsible for managing and storing files persistently on disk
      - data structures
      - implementations of file operations
    - Each file has a unique, low-level name called **inode number** in the file system
      - each file has a corresponding inode data stucture storing the metadata
      - inode number is used to find the inode in the inode table
  - Directory: contains a list of (user-readable name, low-level name) pairs.
    - each entry refers to either *files* or other *directories*

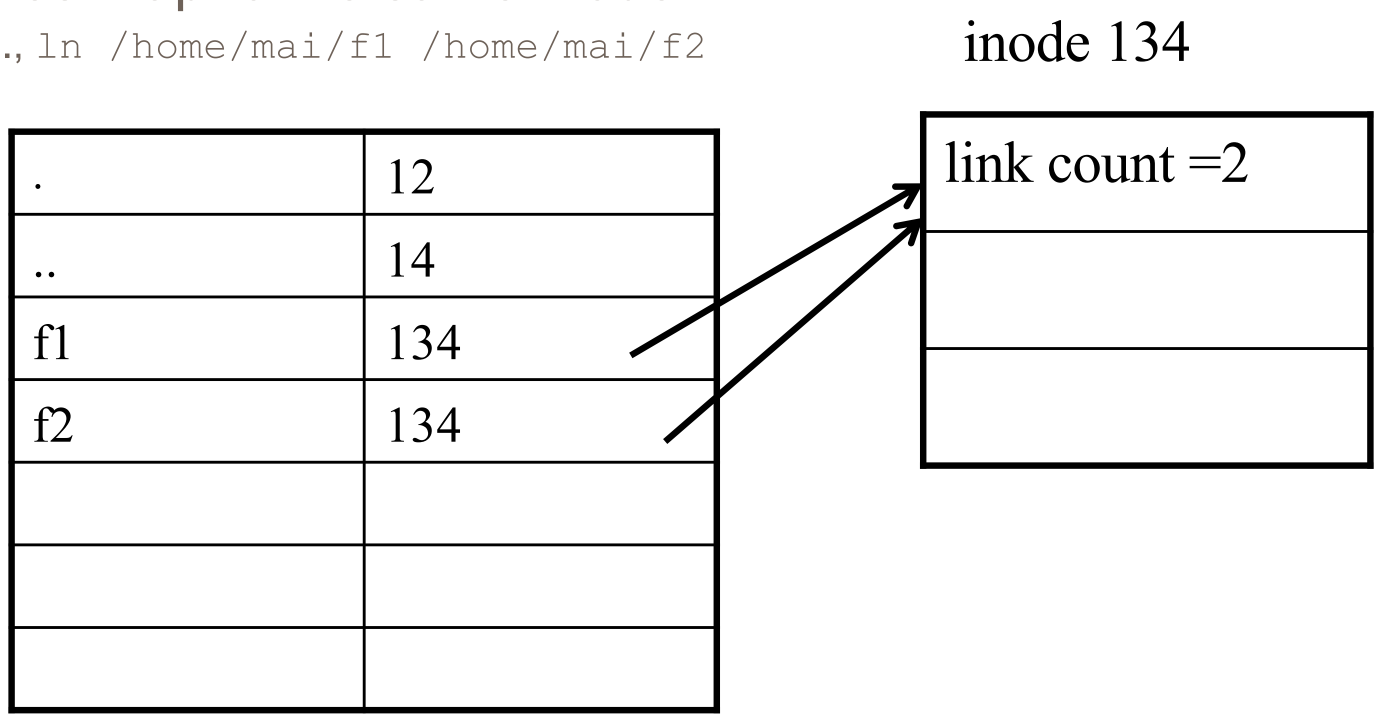
# Review

- File Systems
  - Basic layout
    - data region: user data
    - metadata region: inodes, bitmaps, superblock



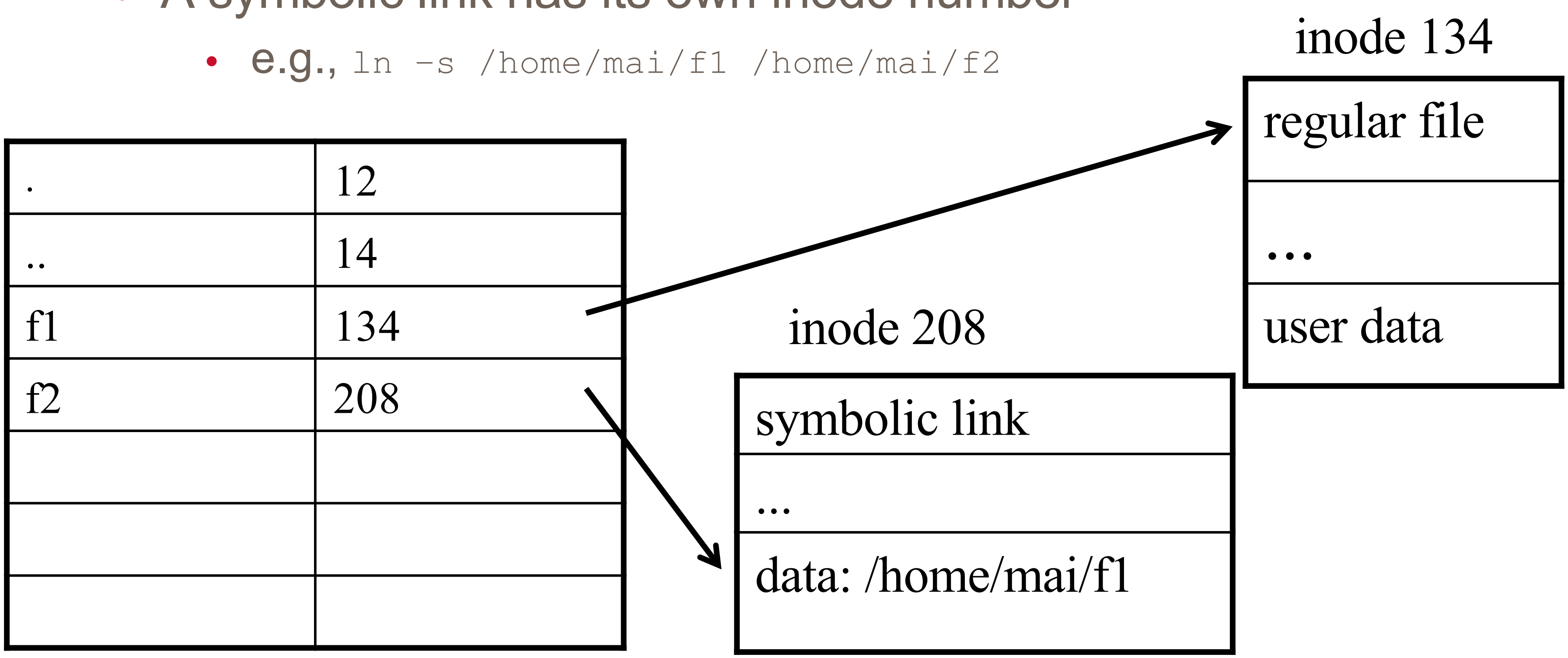
# Review

- File Systems
  - Hard link
    - Both files map to the same inode
      - e.g., `ln /home/mai/f1 /home/mai/f2`



# Review

- File Systems
  - Symbolic link
    - A symbolic link has its own inode number
      - e.g., `ln -s /home/mai/f1 /home/mai/f2`



# Review

- File Systems
  - Timeline of reading a file (/foo/bar) from disk

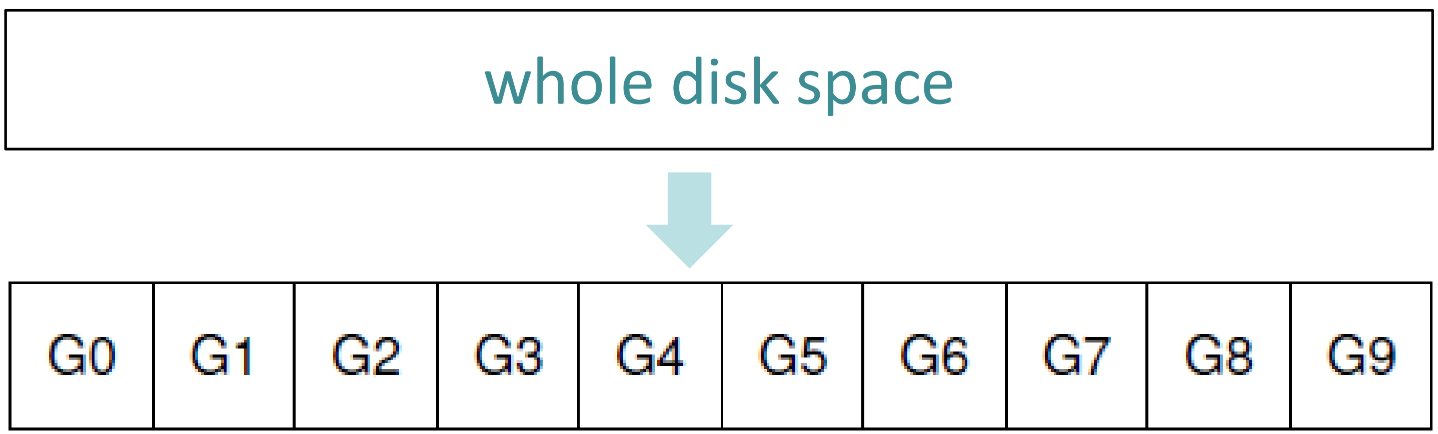
| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[2] |
|---|---|---|---|---|---|---|---|---|---|---|
| **open(bar)** | | | read | | | | | | | |
| | | | | | | read | | | | |
| | | | | read | | | | | | |
| | | | | | | | read | | | |
| | | | | | read | | | | | |
| **read()** | | | | | read | | | | | |
| | | | | | | | | read | | |
| | | | | | write | | | | | |
| **read()** | | | | | read | | | | | |
| | | | | | | | | | read | |
| | | | | | write | | | | | |
| **read()** | | | | | read | | | | | |
| | | | | | | | | | | read |
| | | | | | write | | | | | |

# Review

- File Systems
  - Caching & Buffering
    - Reading and writing files are expensive, incurring many I/Os
    - FSes use system memory (DRAM) to cache reads and buffer writes
      - **page cache** in Linux
      - FS can optimize the writes in memory, e.g.:
        - batch some updates into a smaller set of I/Os
        - avoiding unnecessary I/O (e.g., overwritten in memory)
  - Applications may force flushing dirty data to disk by calling `fsync()`

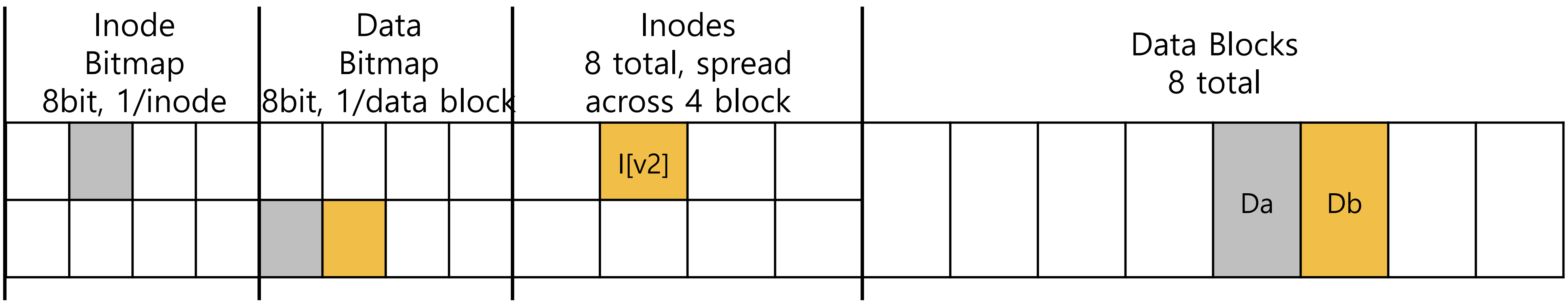
# Review

- File Systems
  - The Fast File System (FFS, ~1984)
    - Key insight: disk awareness
      - data structures and allocation polices match the internals of disks
    - Divide the disk into cylinder groups (block groups)
      - place related stuff in the same group, avoid long seek



# Review

- File Systems
  - Crash Consistency Problem
    - An user operation may generate multiple low-level writes that need to be committed atomically
    - Failure events may interrupt the writes and lead to inconsistency or corruption of FS



# Review

- File Systems
  - Two common techniques for data protection
    - Journaling
      - Also called Write-Ahead-Logging (WAL)
      - Basic Idea
        - Do not write to the main FS data structures directly
        - Write to a “journal” data structure first
        - Update the main FS data structures only after all relevant writes are safely stored in the journal
    - FSCK
      - file system checker
      - scan FS metadata, identify and fix inconsistencies between metadata structures
      - cannot fix all issues