OS for Database systems
File System and Networking

Seehwan Yoo

Dankook University

Disclaimer: Some slides are borrowed from UC. Berkeley's 2014 OS and system programming.

목차

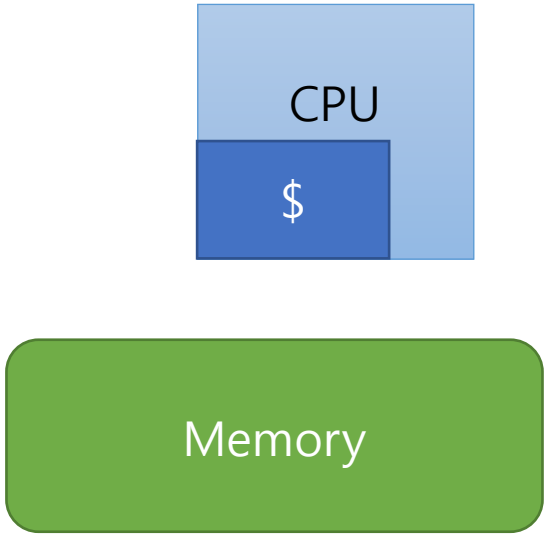
- 강의 개요
- OS 이론 강의 (8/29~9/2)
  - OS 개론, 프로세스. + mini 실습
  - 병렬성. + mini 실습
  - 메모리, IPC. + mini 실습
  - 파일시스템, 네트워킹. + mini 실습
  - 클라우드 컴퓨팅 및 시스템 관리
- OS 실습 강의 (9/19~9/23)
  - 리눅스 개발 환경 구축 및 쉘 프로그래밍 (1d)
  - 다중쓰레드 자료구조 실습 프로젝트 (2d)
  - AWS 실습 및 WordPress 기반 DB 실습 프로젝트 (2d)

Contents

- Caching in OS
- File System
  - Directory structure
  - inode structure
- Networking
  - TCP/IP stack

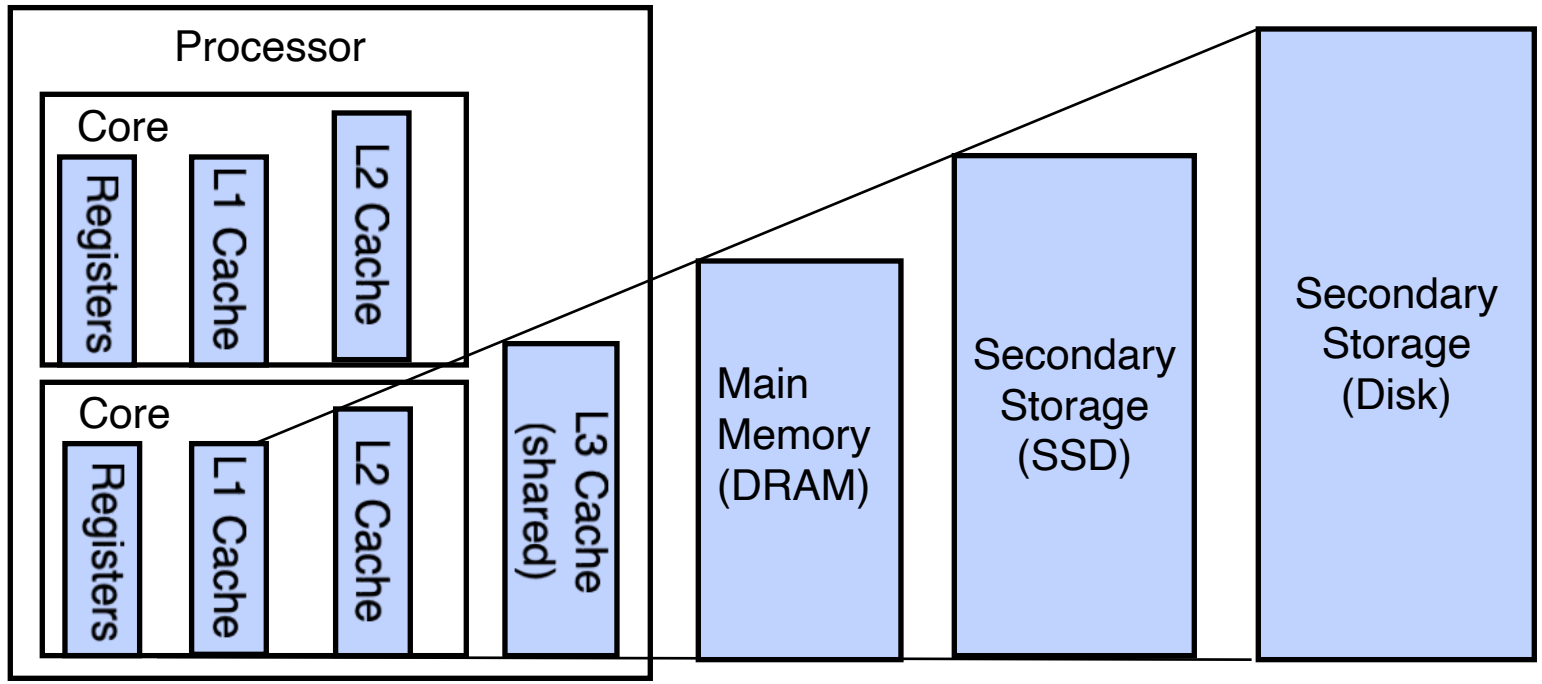
Cache, where CA meets OS

- Cache for performance
  - locate faster access memory, nearby cpu
  - larger memory takes 10x~100x times to access memory
  - e.g.) DRAM access time: 1ms, $ access time: 20us
- Cache hit/miss
  - hit: data is in the $
  - hit ratio 90%, average memory access time?
    - 90% * 20us + 10% * (1ms+20us) = 120us



Memory hierarchy with multiple caches

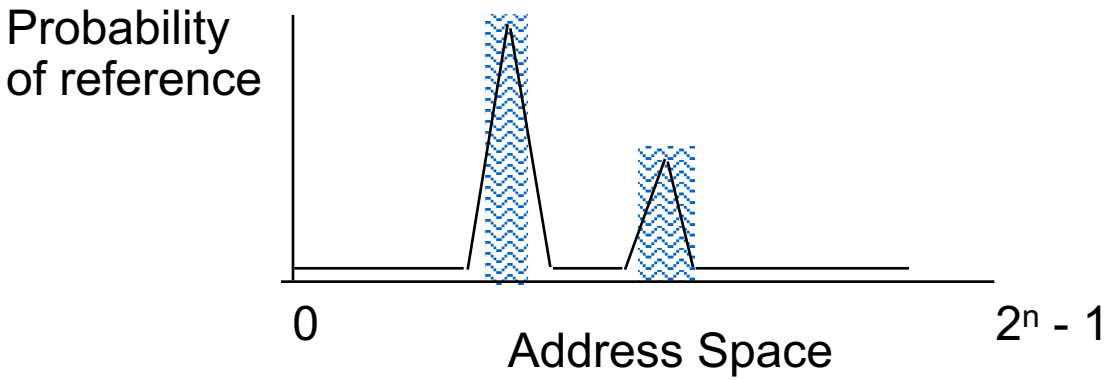
- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology

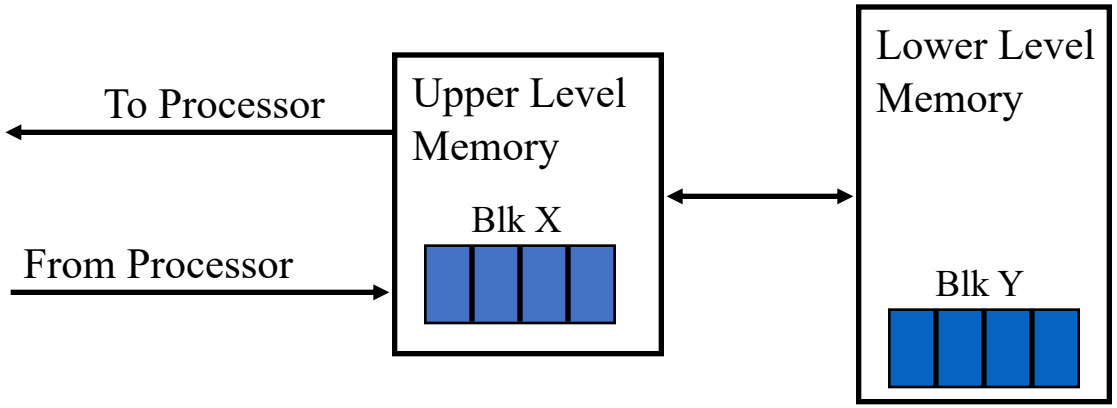


Speed (ns):	0.3	1	3	10-30	100	100,000 (0.1 ms)	10,000,000 (10 ms)
Size (bytes):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

Why Does Caching Work? Locality!

- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels

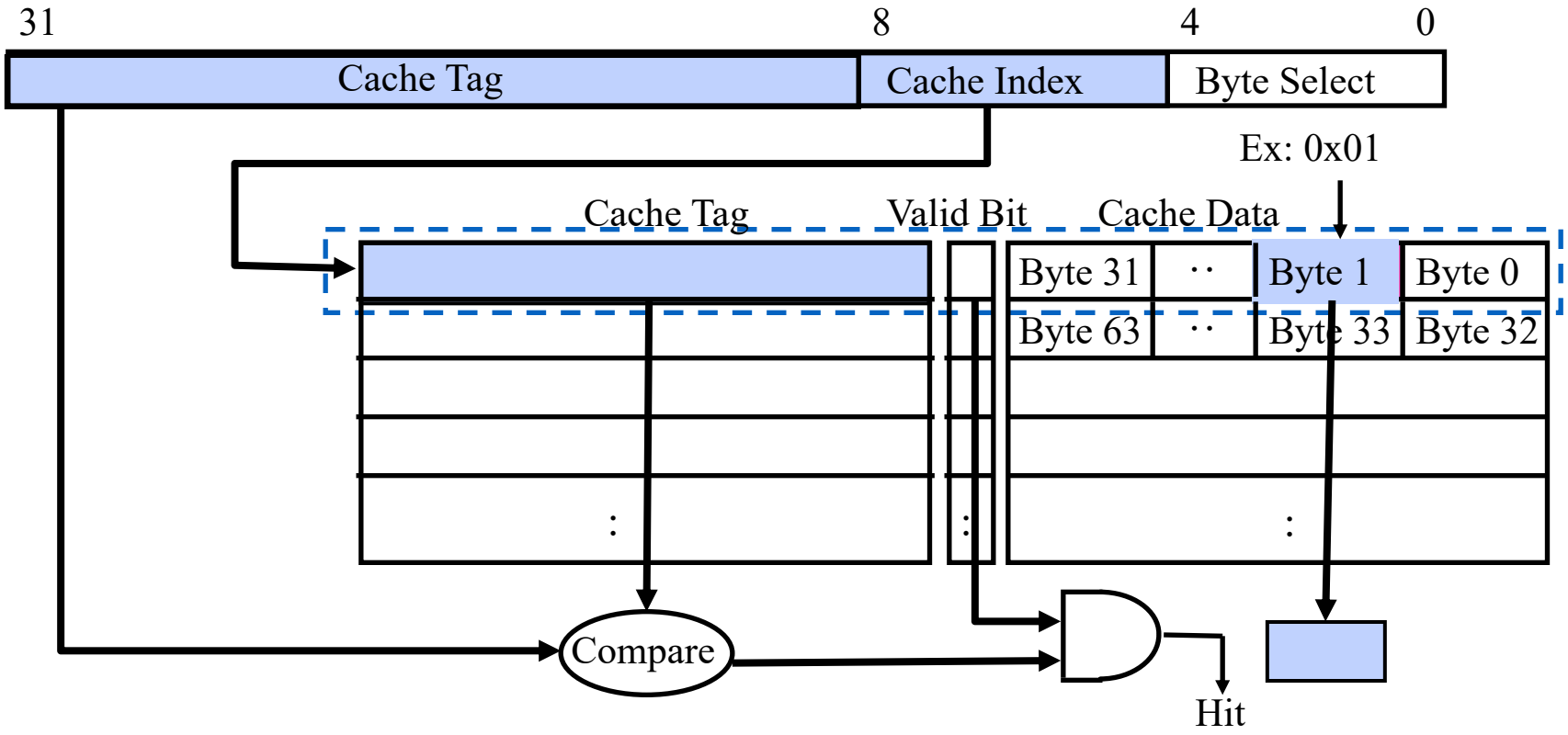




Design issues for caches

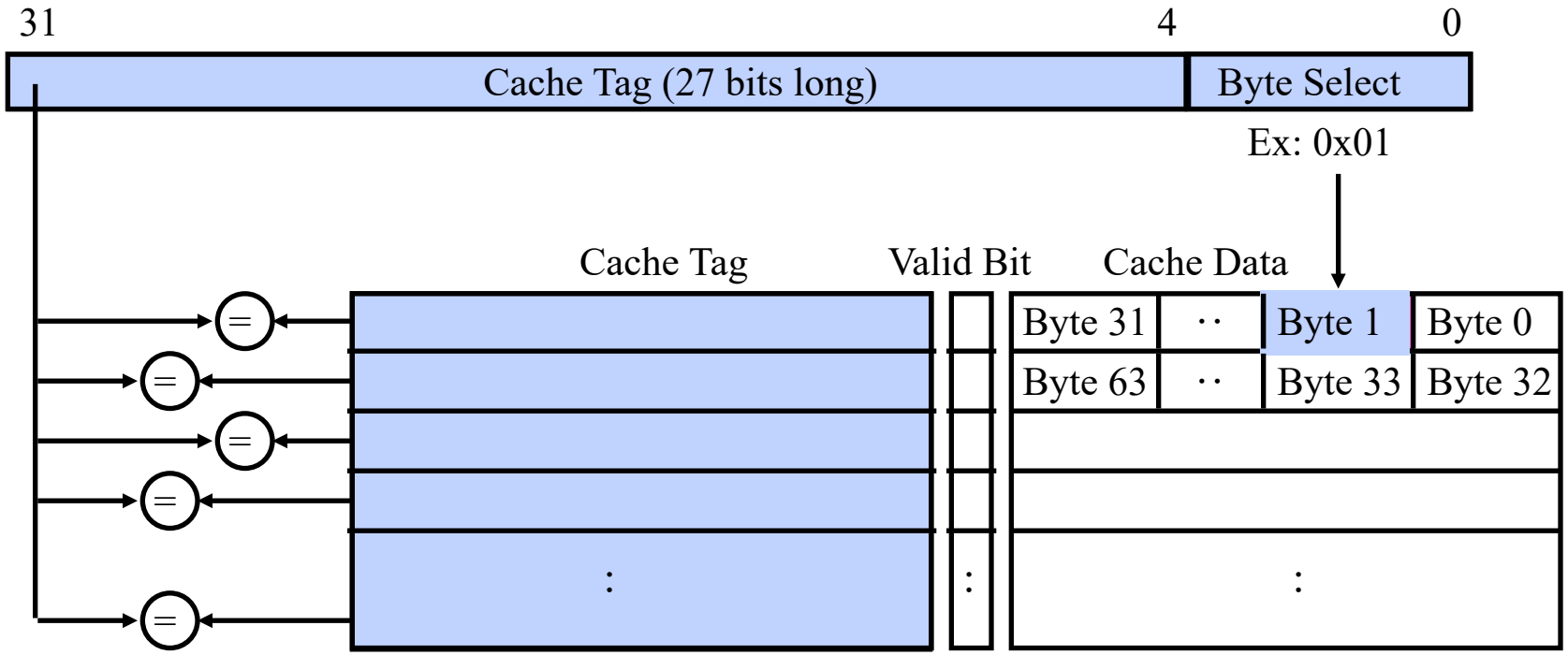
- In Computer Architecture we are focused on cache design as a transparent memory accelerator
  - reduce average MAT (latency), increase BW
- implemented directly in hardware
- Issues:
  - cache size
  - block size
  - associativity (direct mapped, set assoc, fully assoc)
  - placement, replacement
  - number of levels of caches
- trade-offs among all of these

Direct Mapped Cache



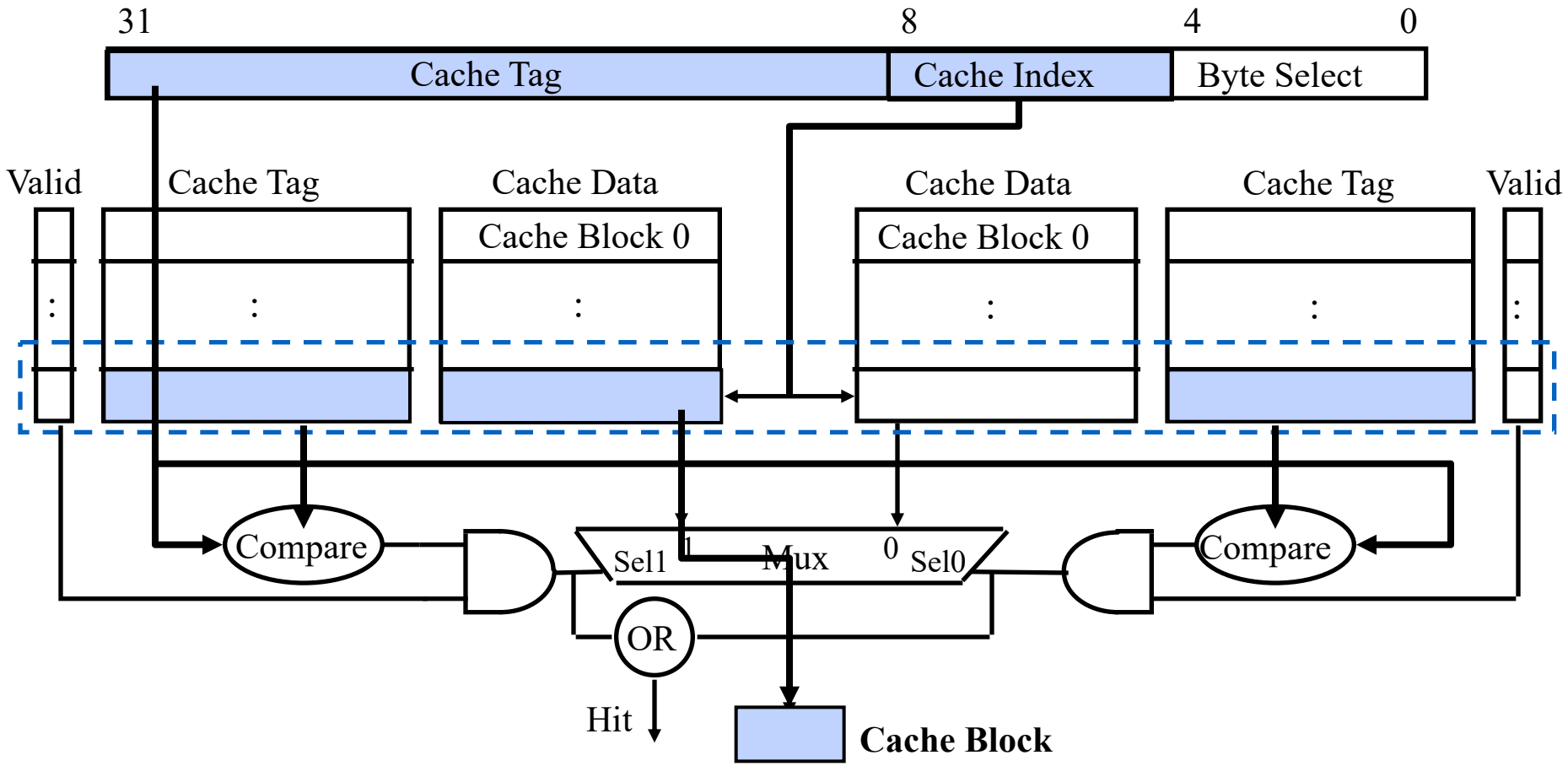
- Cache index selects a cache block
- "Byte select" selects byte within cache block
  - Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same "cache index" shares the same cache entry
  - Conflict misses

Fully Associative Cache



- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block

Set Associative Cache



- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result

Sources of Cache Misses

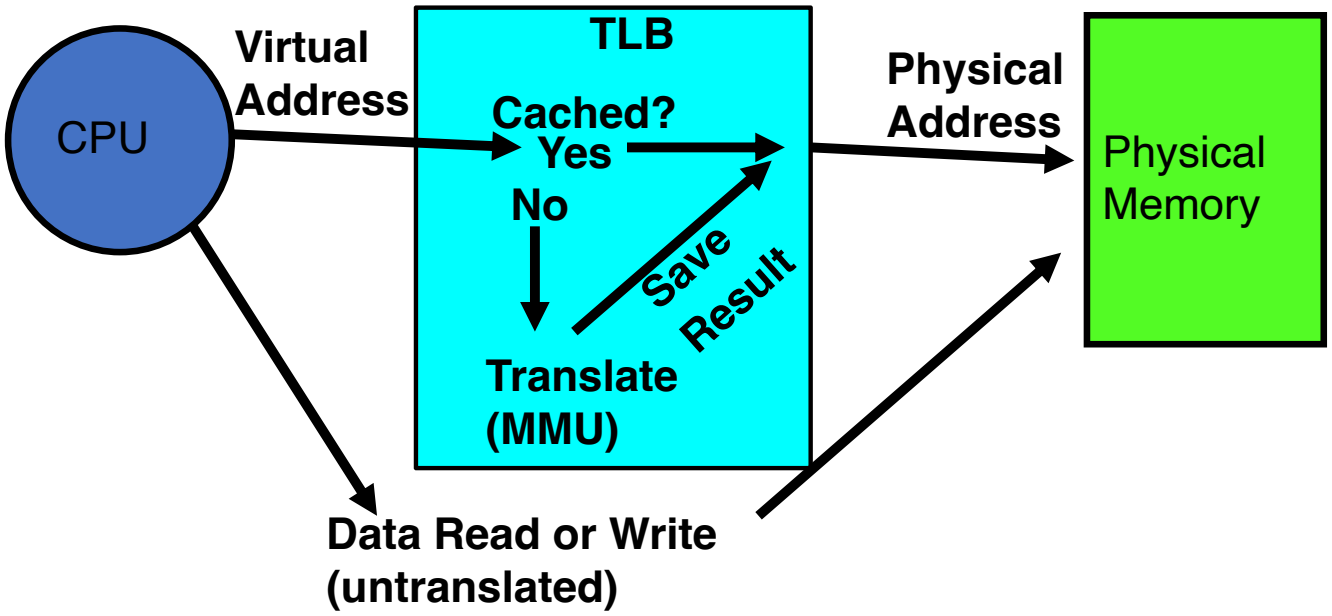
- Compulsory (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- Two others:
  - Coherence (Invalidation): other process (e.g., I/O) updates memory
  - Policy: Due to non-optimal replacement policy

Cache Design Issues

- Organization
  - cache size, block size
  - 1-way, n-way, associative
- Write Policy
  - write-through, write-back
- Replacement policy
  - given n-way associativity, which of the n gets replaced
  - FIFO, Random, LRU, Clock
- Coherence Policy (multi-processor)
  - write-invalidate, write-update

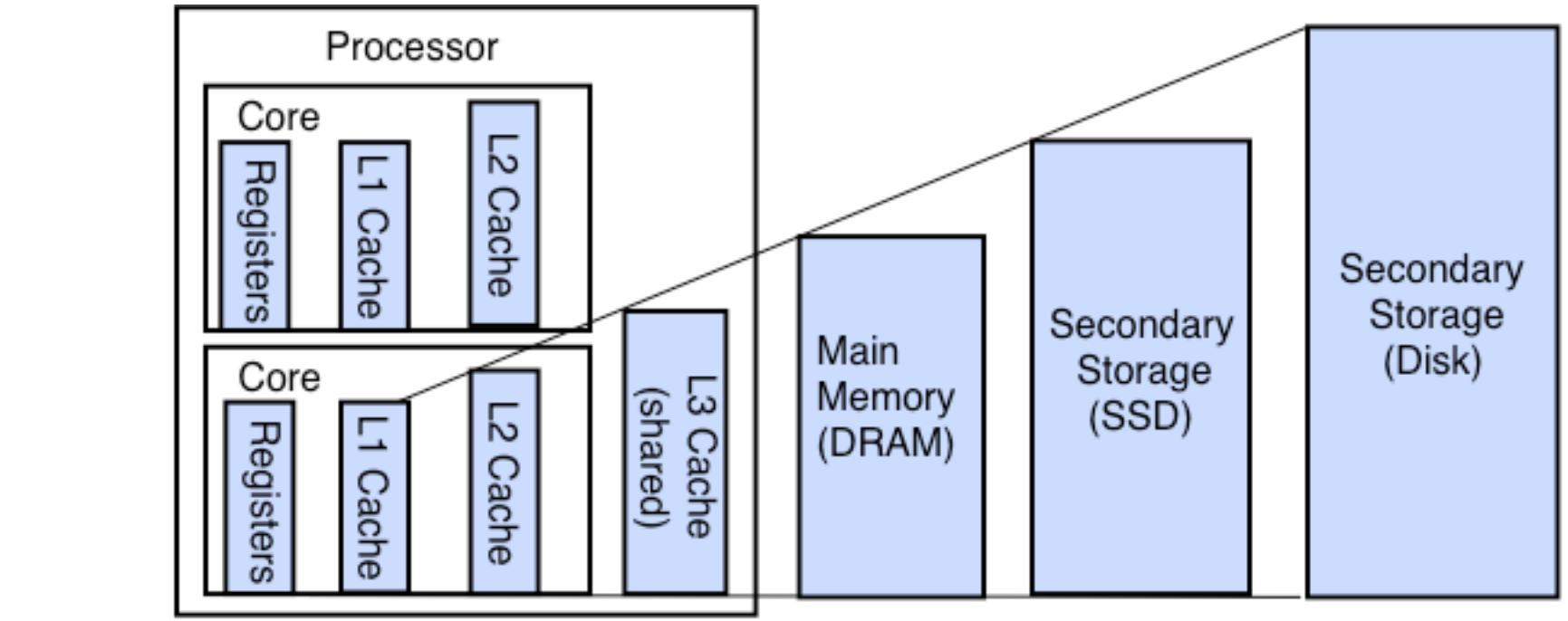
Caching Applied to Address Translation

- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but lots
- Each TLB entry is for a whole page of blocks !!!



Where does caching arise in Operating Systems ?

Hardware Design Trade-offs



Speed (ns):	0.3	1	3	10-30	100	100,000 (0.1 ms)	10,000,000 (10 ms)
Size (bytes):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

Where does caching arise in Operating Systems ?

- Direct use of caching techniques
  - paged virtual memory (mem as cache for disk)
  - TLB (cache of PTEs)
  - file systems (cache disk blocks in memory)
  - DNS (cache hostname => IP address translations)
  - Web proxies (cache recently accessed pages)
- Which pages to keep in memory?

Where does caching arise in Operating Systems ?

- Indirect - dealing with cache effects
- Process scheduling
  - which and how many processes are active ?
  - large memory footprints versus small ones ?
  - priorities ?
- Impact of thread scheduling on cache performance
  - rapid interleaving of threads (small quantum) may degrade cache performance
    - increase ave MAT !!!
- Designing operating system data structures for cache performance

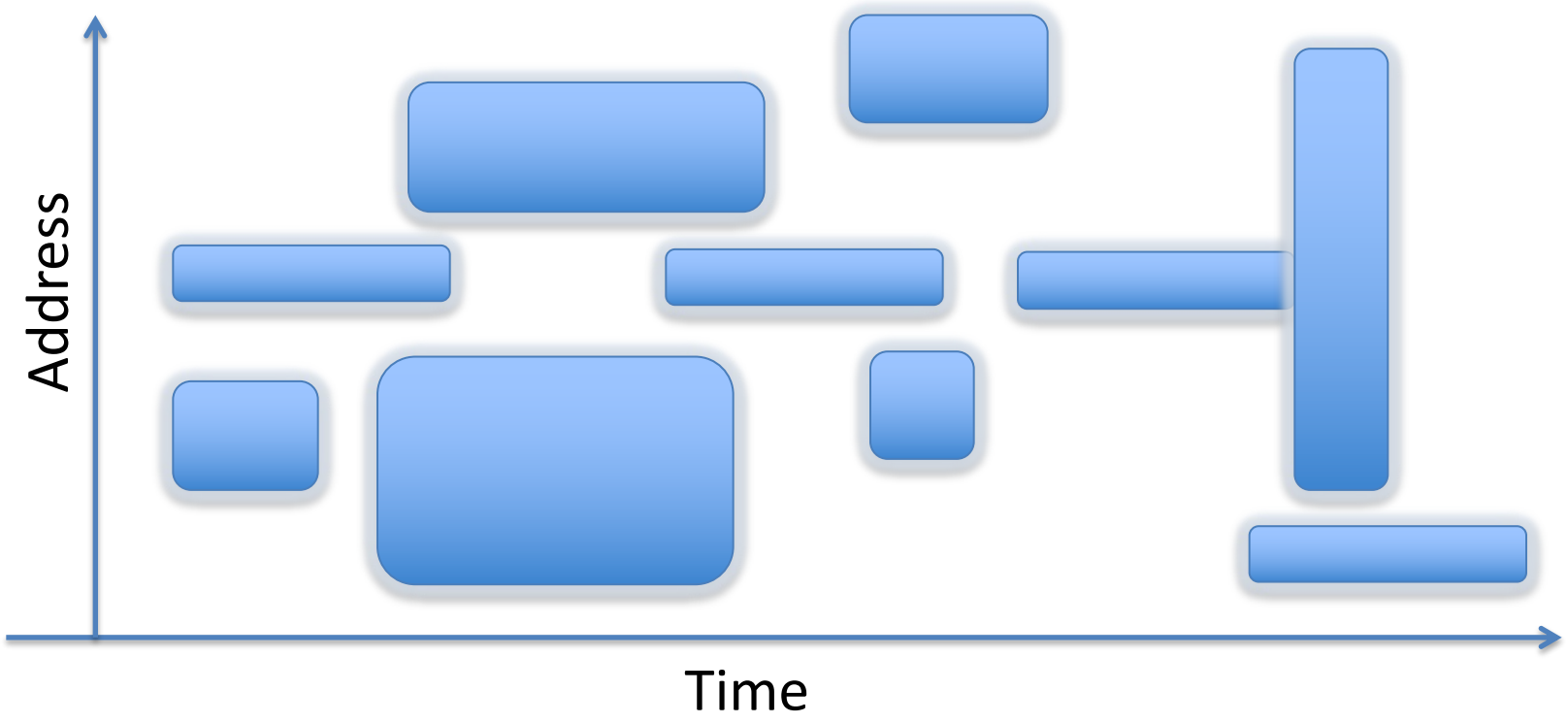
Where does caching arise in Operating Systems ?

- Maintaining the correctness of various caches

- TLB consistent with PT across context switches ?
- Across updates to the PT ?
- Shared pages mapped into VAS of multiple processes ?

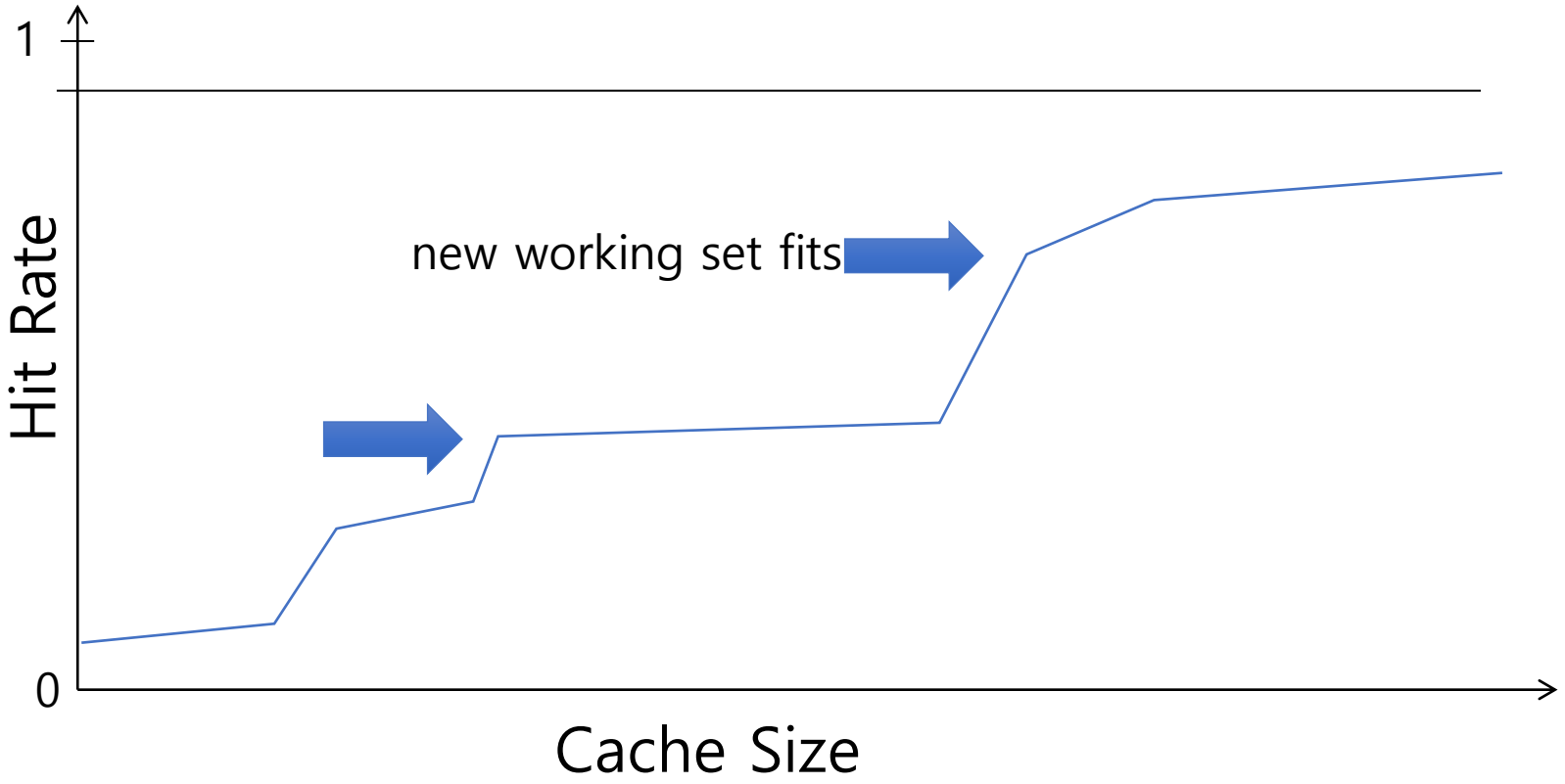
Working Set Model

- As a program executes it transitions through a sequence of "working sets" consisting of varying sized subsets of the address space



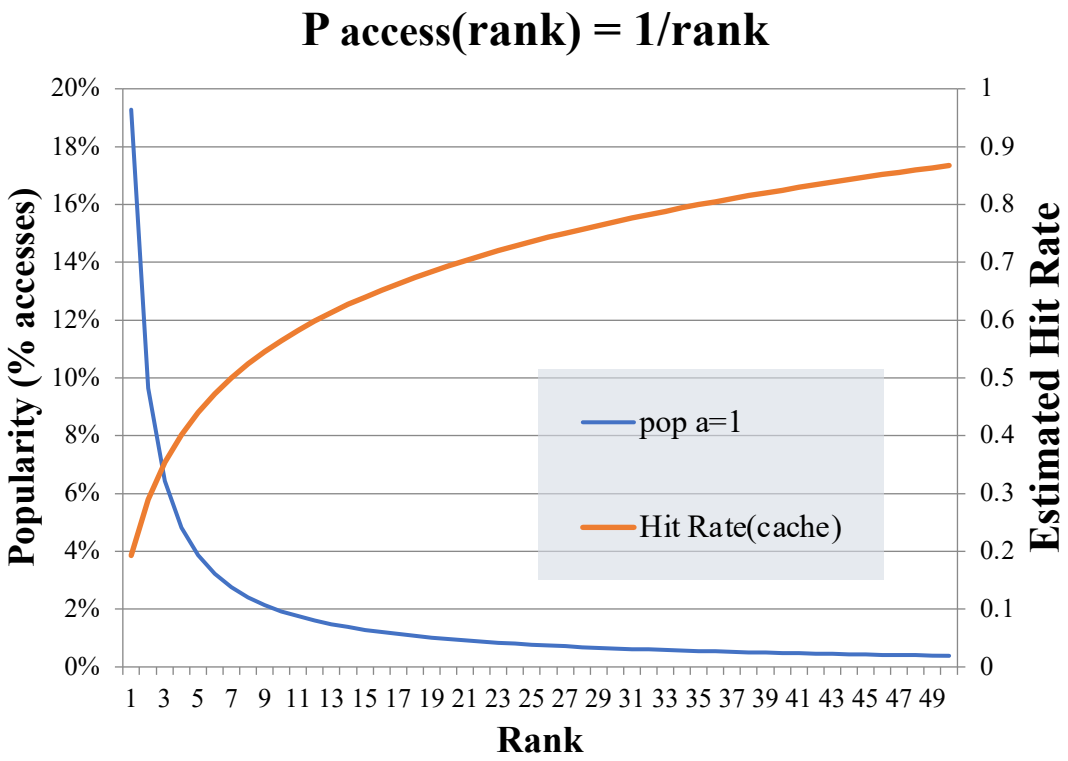
Cache Behavior under WS model

- Amortized by fraction of time the WS is active
- Transitions from one WS to the next
- Capacity, Conflict, Compulsory misses
- Applicable to memory caches and pages. Others ?



Another model of Locality: Zipf

- Likelihood of accessing item of rank r is $\alpha 1/r^a$
- Although rare to access items below the top few, there are so many that it yields a “heavy tailed” distribution.
- Substantial value from even a tiny cache
- Substantial misses from even a very large one



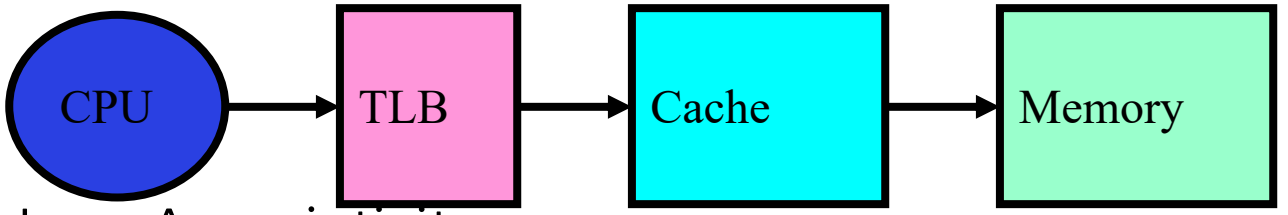
What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - If PTE valid, hardware fills TLB and processor never knows
    - If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - If PTE valid, fills TLB and returns from fault
    - If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things

What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - What if switching frequently between processes?
  - Include ProcessID in TLB
    - This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - Otherwise, might think that page is still in memory!

What TLB organization makes sense?



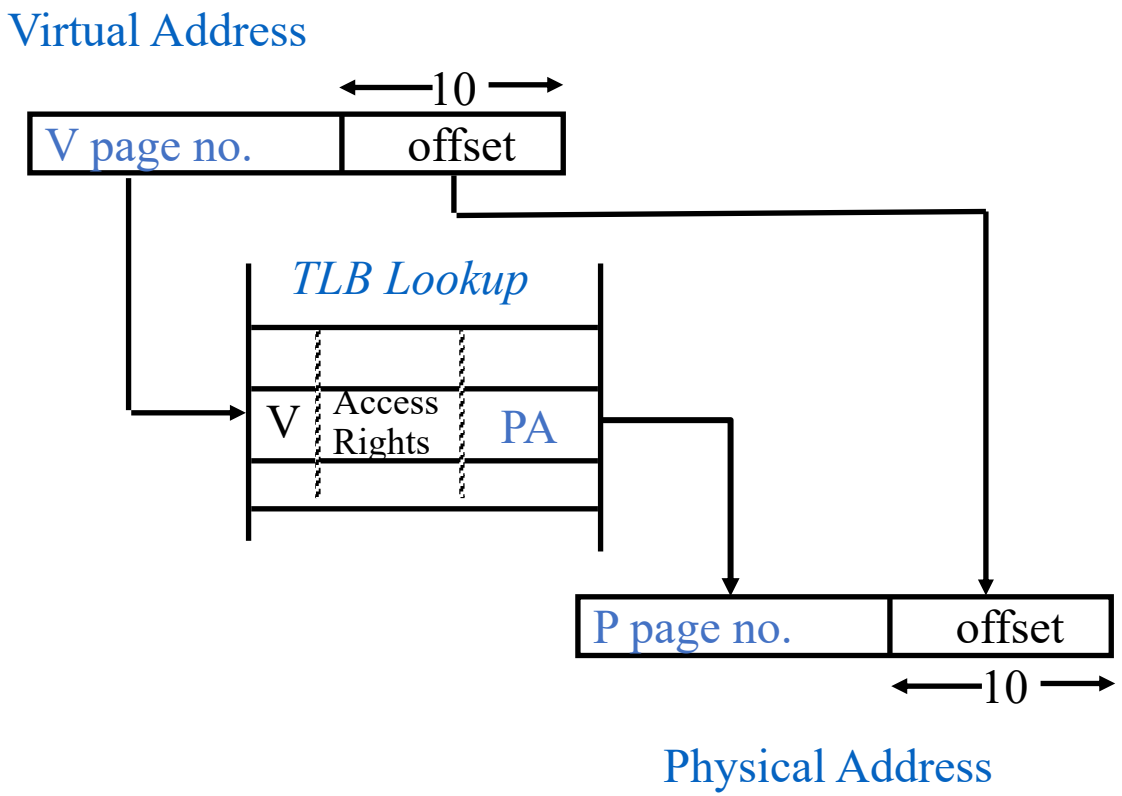
- Needs to be really fast
  - Critical path of memory access
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - First page of code, data, stack may map to same entry
    - Need 3-way associativity at least?
  - What if use high order bits as index?
    - TLB mostly unused for small programs

TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a “TLB Slice”
- When does TLB lookup occur relative to memory cache access?
  - Before memory cache lookup?
  - In parallel with memory cache lookup?

Reducing translation time further

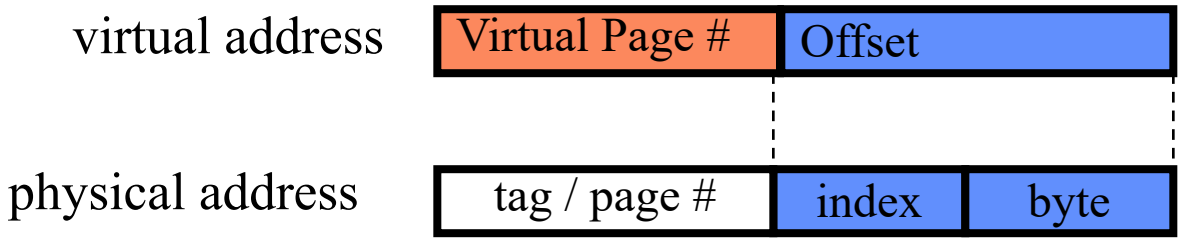
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

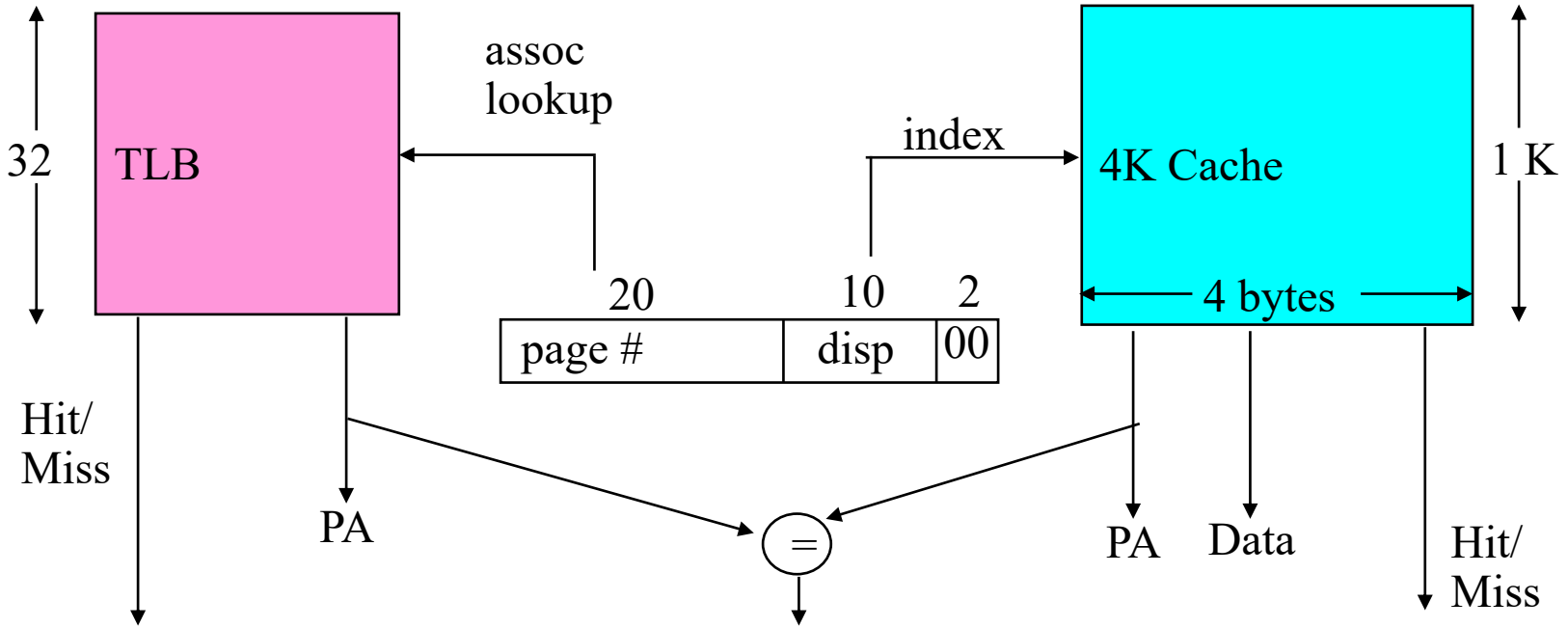
Overlapping TLB & Cache Access (1/2)

- Main idea:
  - Offset in virtual address exactly covers the “cache index” and “byte select”
  - Thus can select the cached byte(s) in parallel to perform address translation

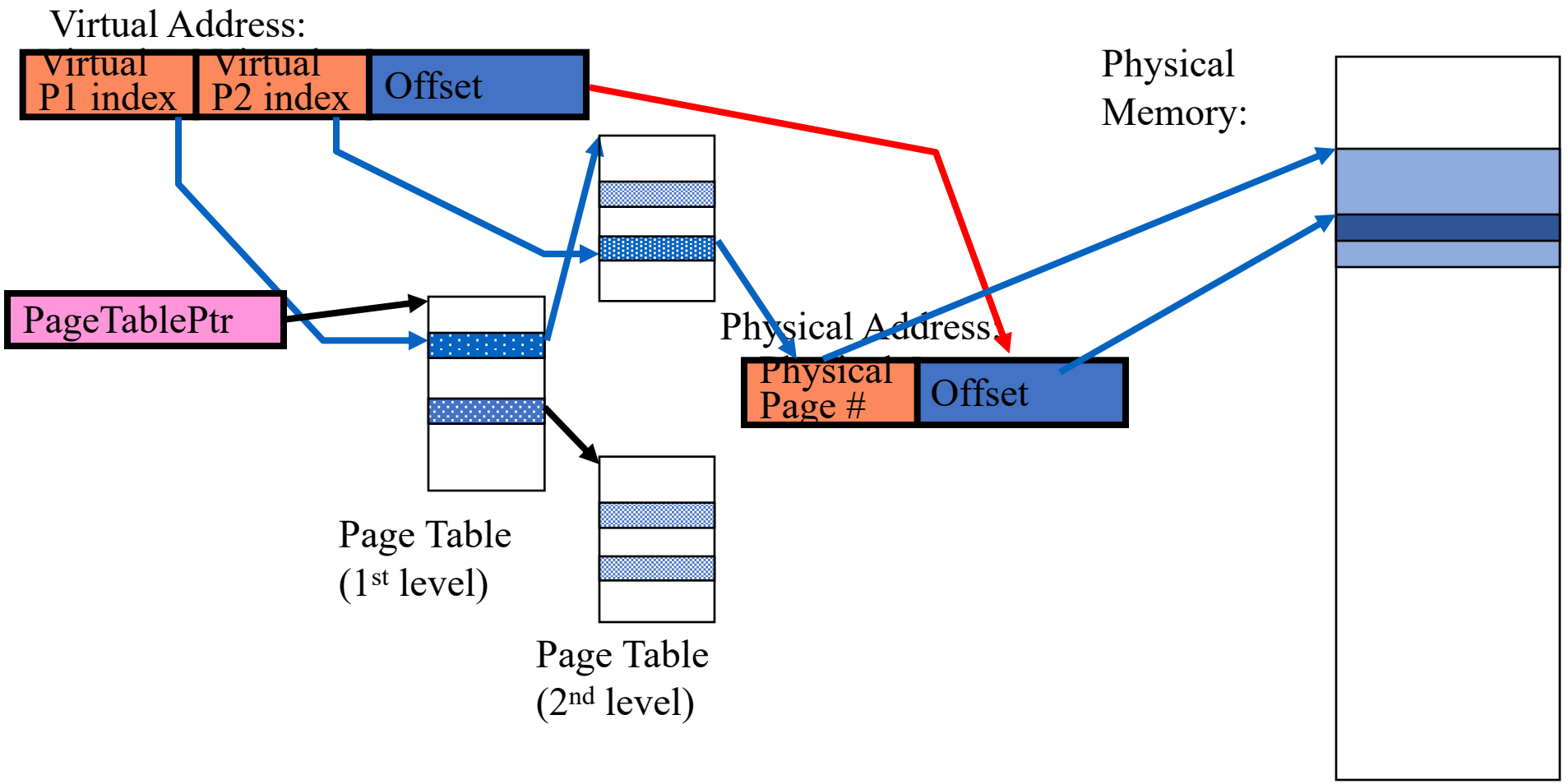


Overlapping TLB & Cache Access (1/2)

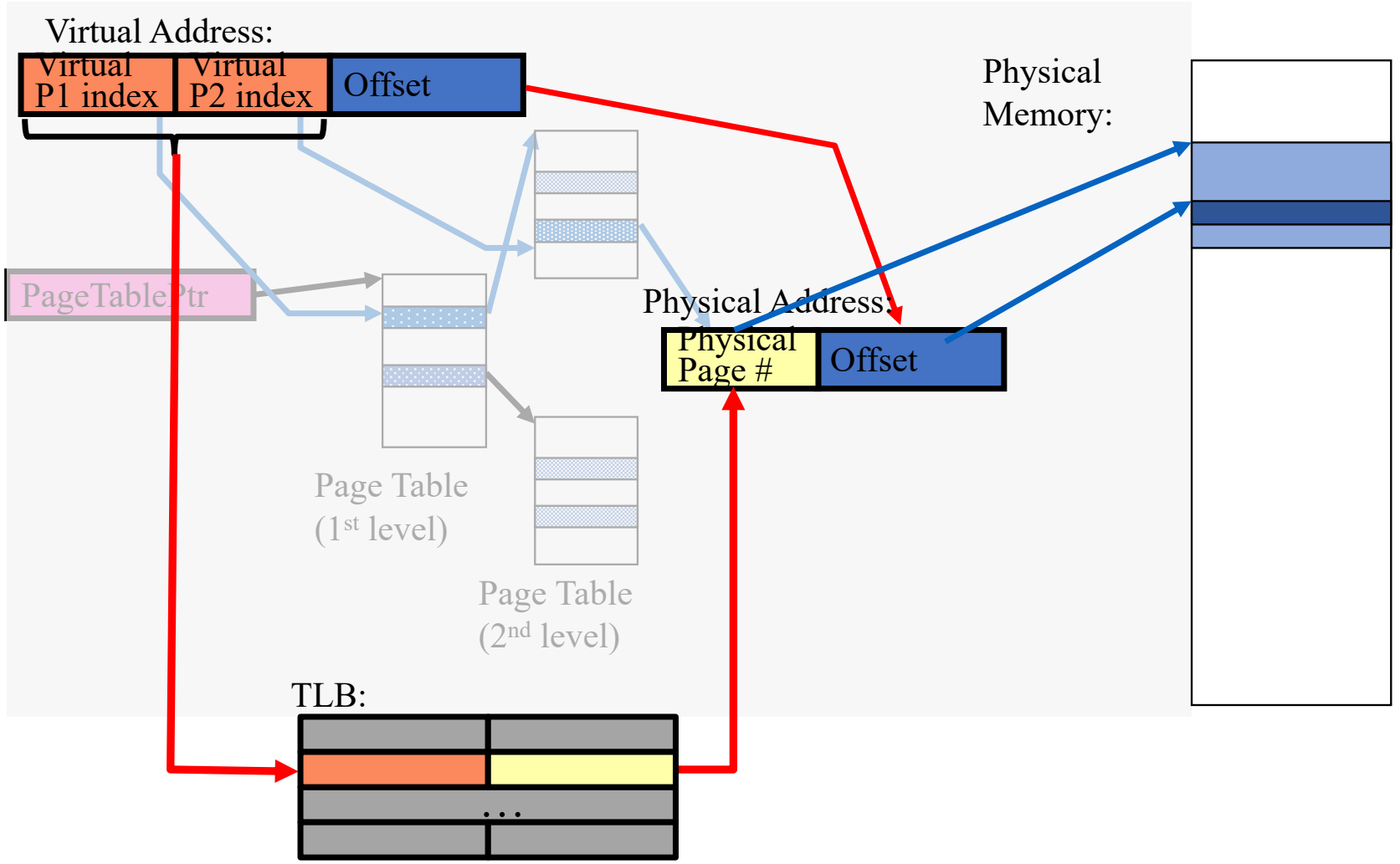
- Here is how this might work with a 4K cache:
- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else.



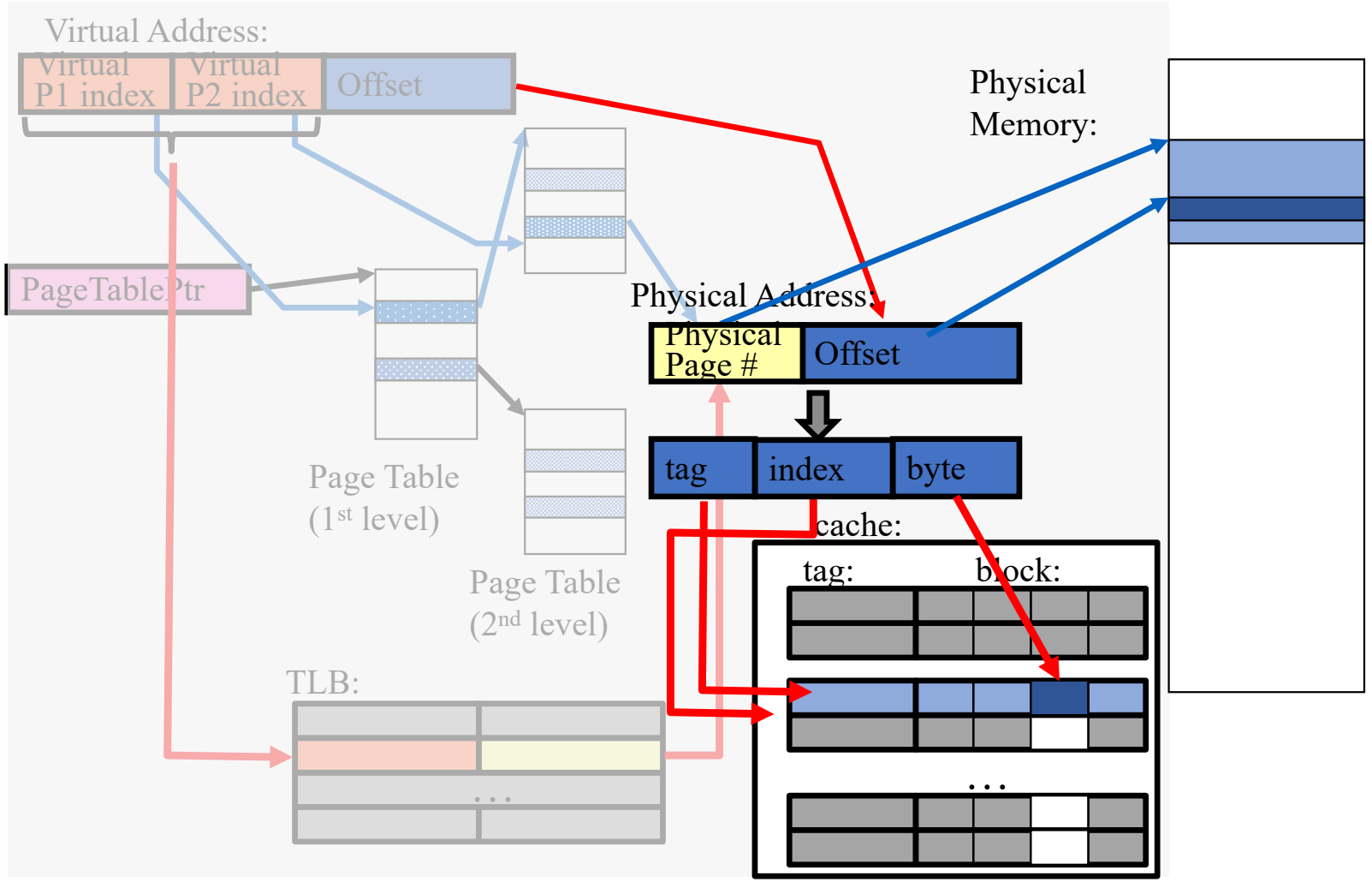
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: Cache



Some more on Caching in OS

- Buffer cache
- in-storage data (in the medium) can be cached in
  - in-storage memory: HDD, SSD has DRAM within the device!
  - in-main memory: main memory can cache, reducing I/O
  - in-CPU: on the CPU cache
  - we will revisit this in later chapter!
- DMA
  - mem-device transfer without CPU intervention
  - mem-cpu-device transfer could use CPU cache
  - DMA works with physical address (bus address)
  - DMA-transferred data is not stored on CPU cache
  - If your data is on the CPU cache, and DMA overwrites it?
    - you see stale data; page table entry has cacheable flag
    - If it is not set, the page address is not cacheable (e.g. MMIO region)

TLB and Cache size

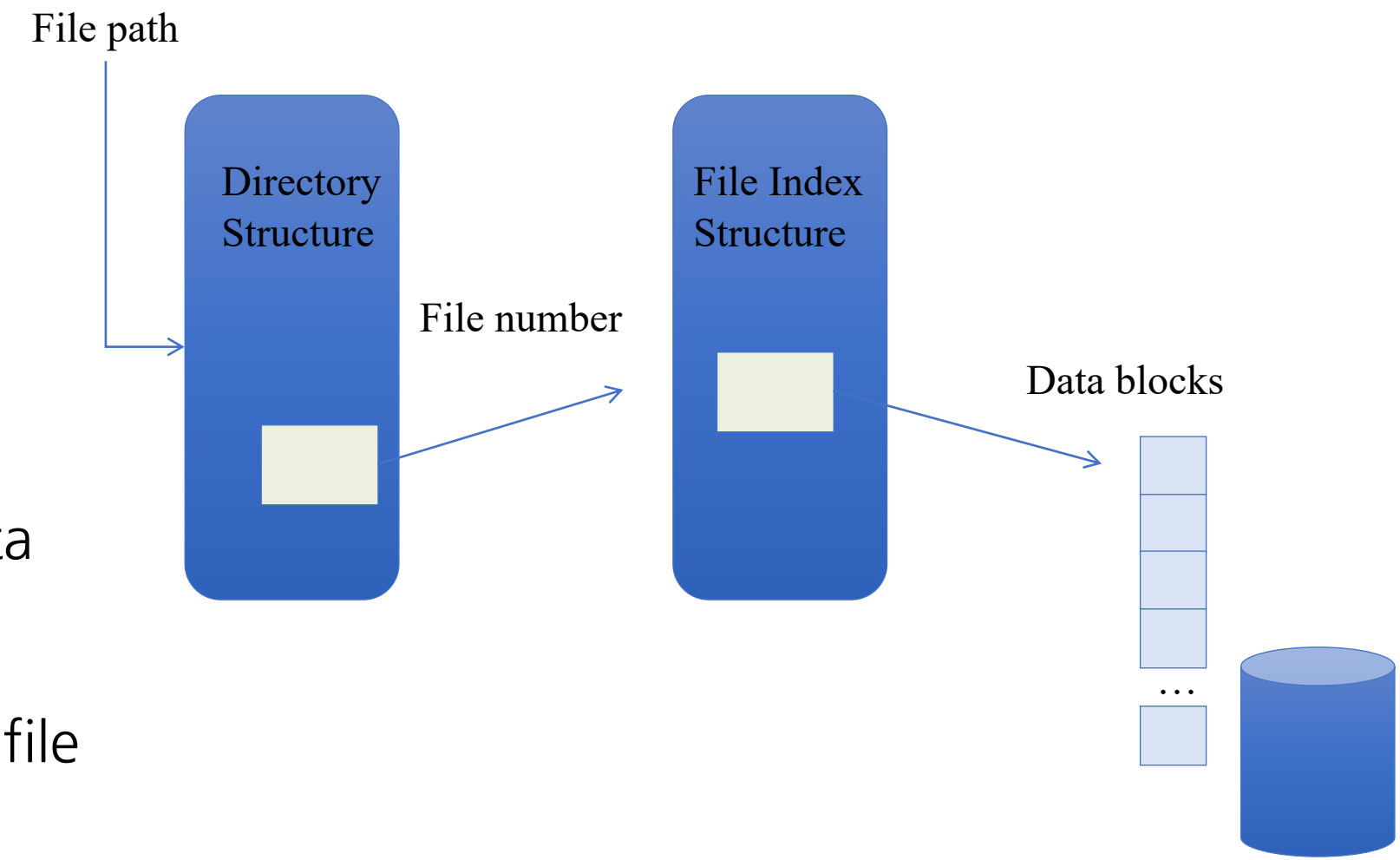
- CPUs have TLB and Cache
  - TLB size: only a few KB
  - Cache size: tens of MB
- why they are so different in size?
  - single TLB entry (8Bytes) can cover up to 4KB memory region
  - However, single cacheline (64Bytes) can cover 64 Bytes
  - To cover 4KB memory region (footprint),
    - you may need single TLB entry, and 4KB data store in cache

File System Design

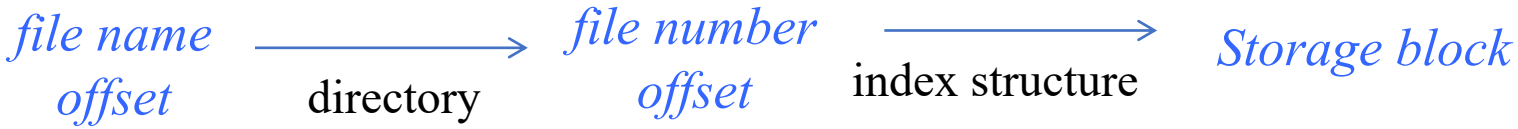
Components of a File System

- Directory structure
- File structure

- Data
- Meta-data
  - Data for data
  - Data that describes data
  - Property
  - E.g.) filename, size
    are not the data in the file



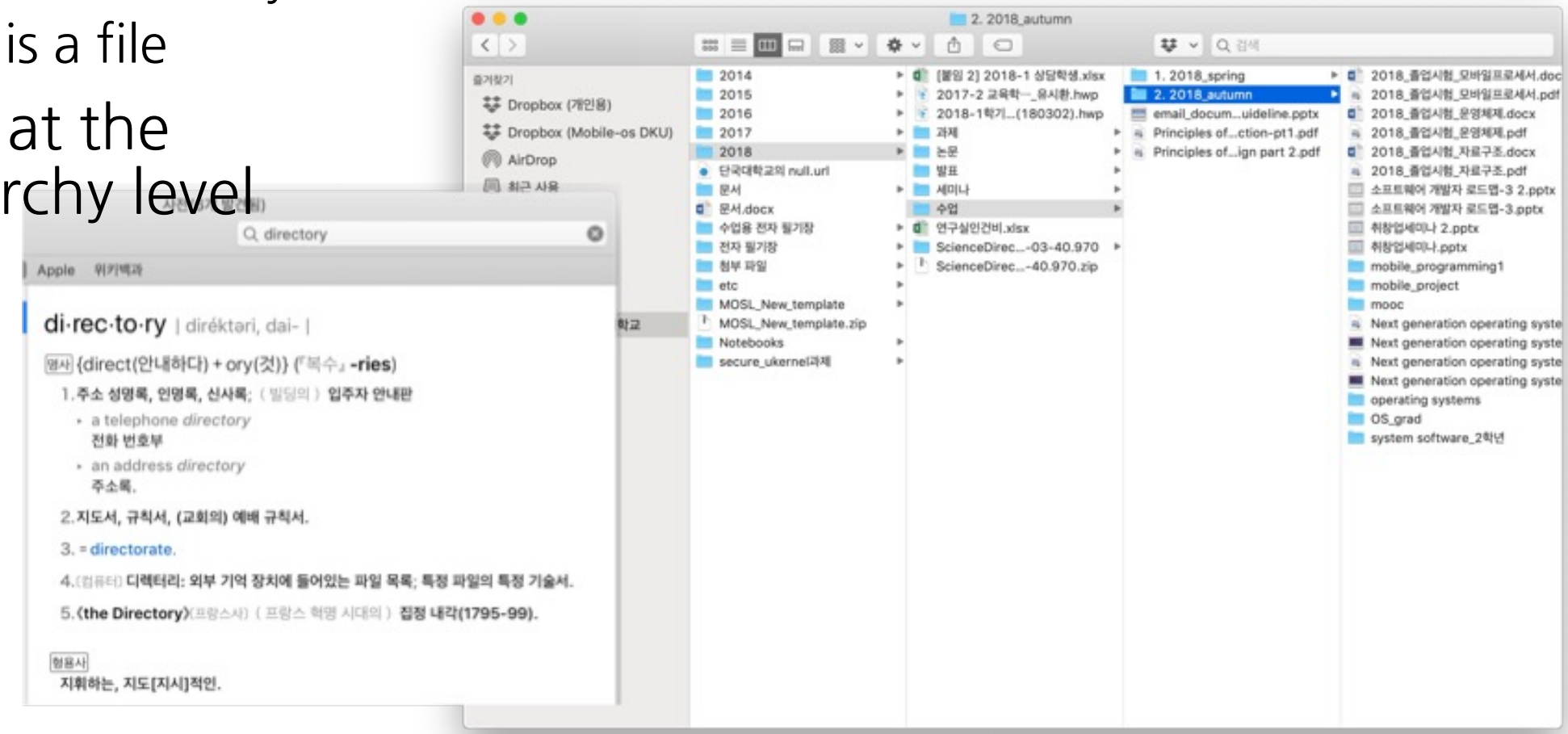
Components of a file system



- Open performs *name resolution*
  - Translates *pathname* into a "*file number*"
    - Used as an "index" to locate the blocks
  - Creates a file descriptor in PCB within kernel
  - Returns a "handle" (another int) to user process
- Read, Write, Seek, and Sync operate on handle
  - Mapped to descriptor and to blocks

Directories

- Hierarchical structure of files
  - Files are in a directory
  - Directory is a file
- List of files at the same hierarchy level

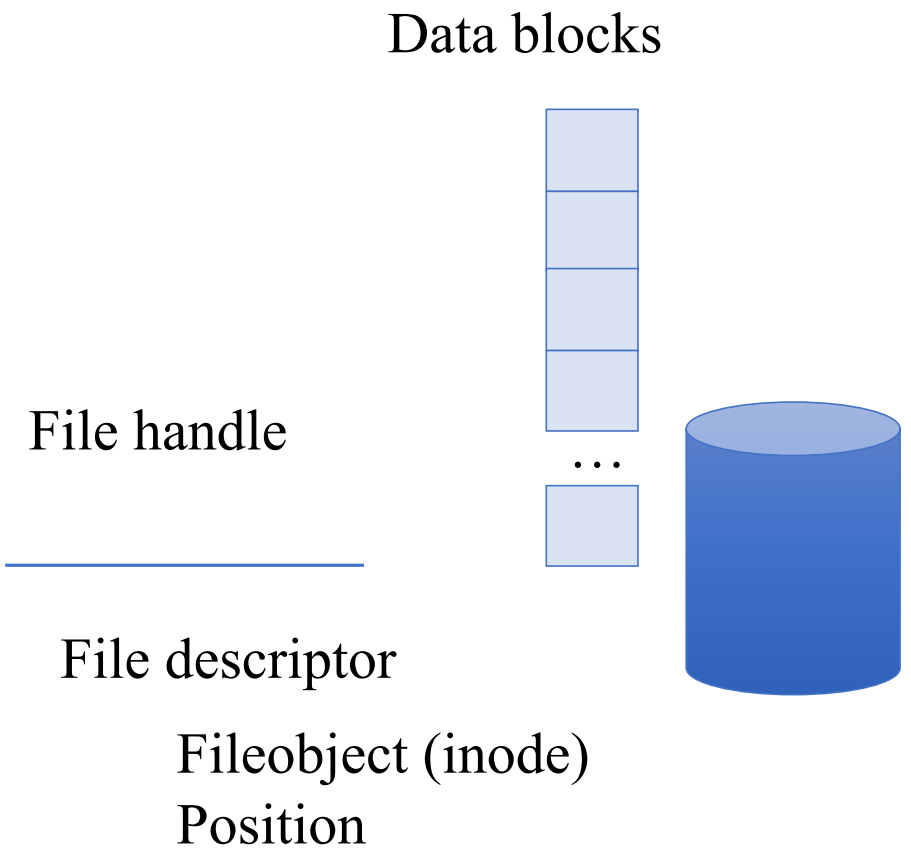


Directory

- Basically a hierarchical structure
- Each directory entry is a collection of
  - Files
  - Directories
    - A link to another entries
- Each has a name and attributes
  - Files have names, sizes, created dates, etc.
- Links (hard links) make it a DAG, not just a tree
  - Softlinks (aliases) are another name for an entry

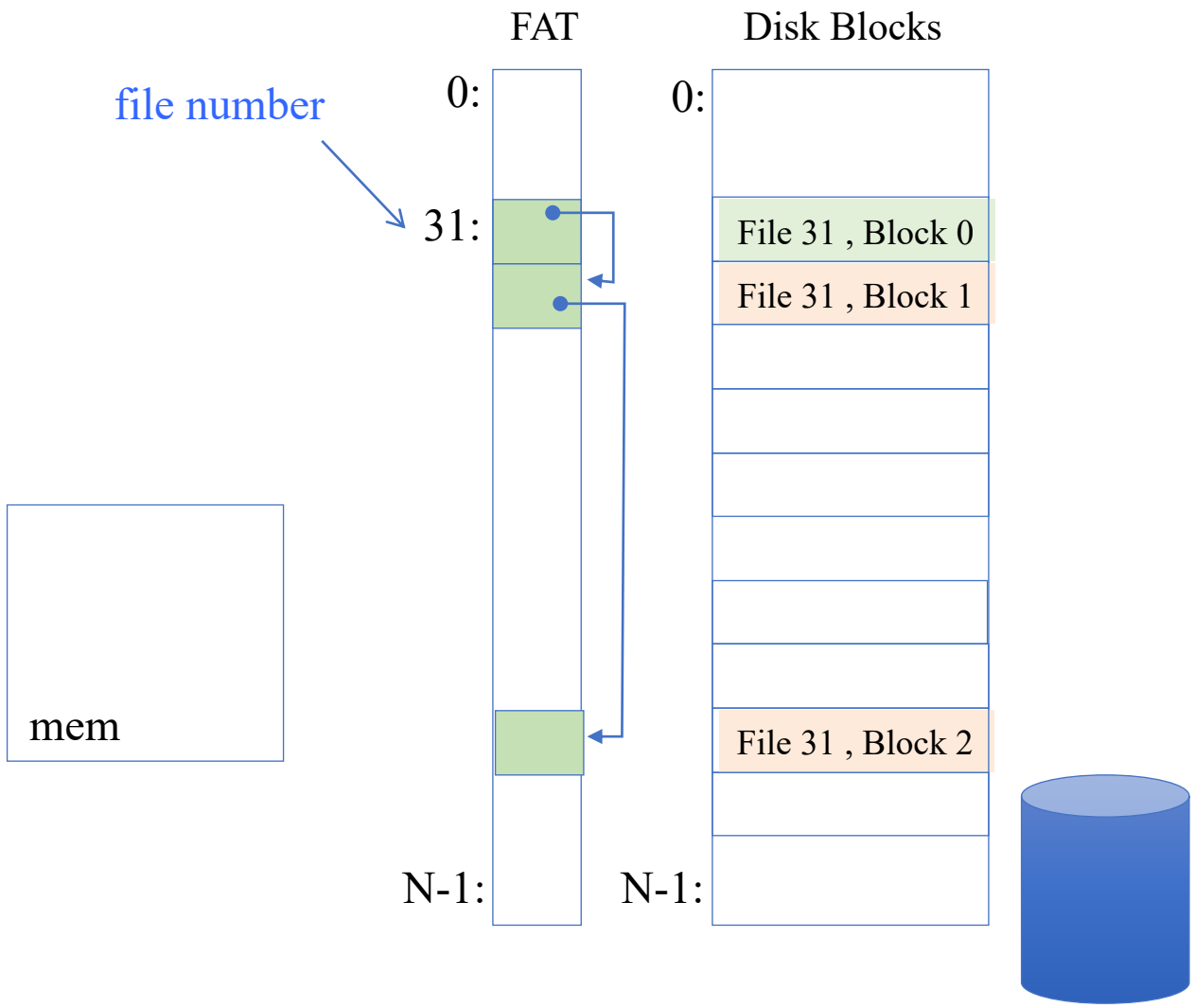
File

- Named permanent storage
- Contains
  - Data
    - Blocks on disk somewhere
  - Metadata (Attributes)
    - Owner, size, last opened, …
    - Access rights
      - R, W, X
      - Owner, Group, Other (in Unix systems)
      - Access control list in Windows system



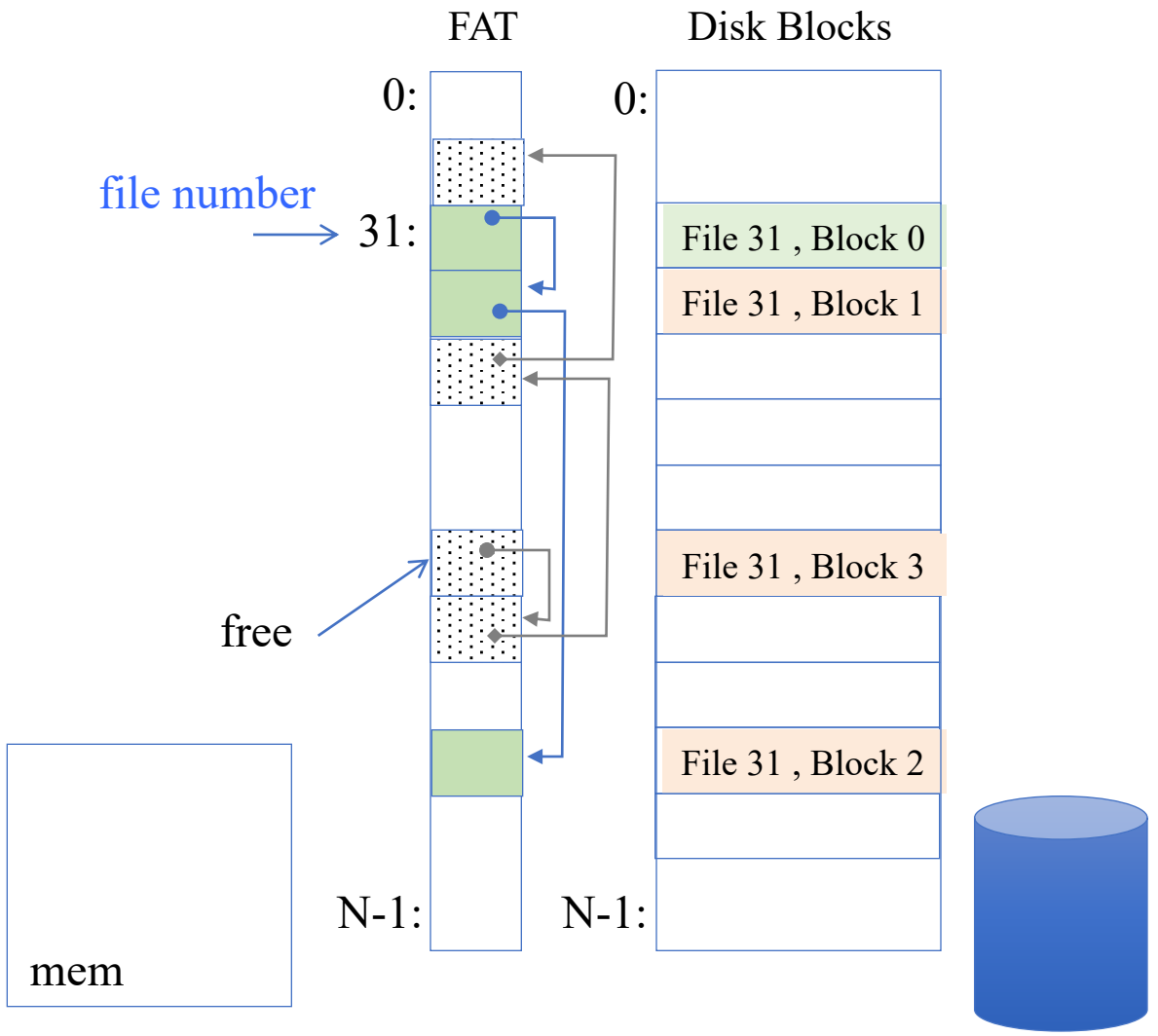
FAT (File Allocation Table)

- Assume (for now) we have a way to translate a path to a “file number”
  - i.e., a directory structure
- Disk Storage is a collection of Blocks
  - Just hold file data
- Example: file_read 31, < 2, x >
- Index into FAT with file number
- Follow linked list to block
- Read the block from disk into mem



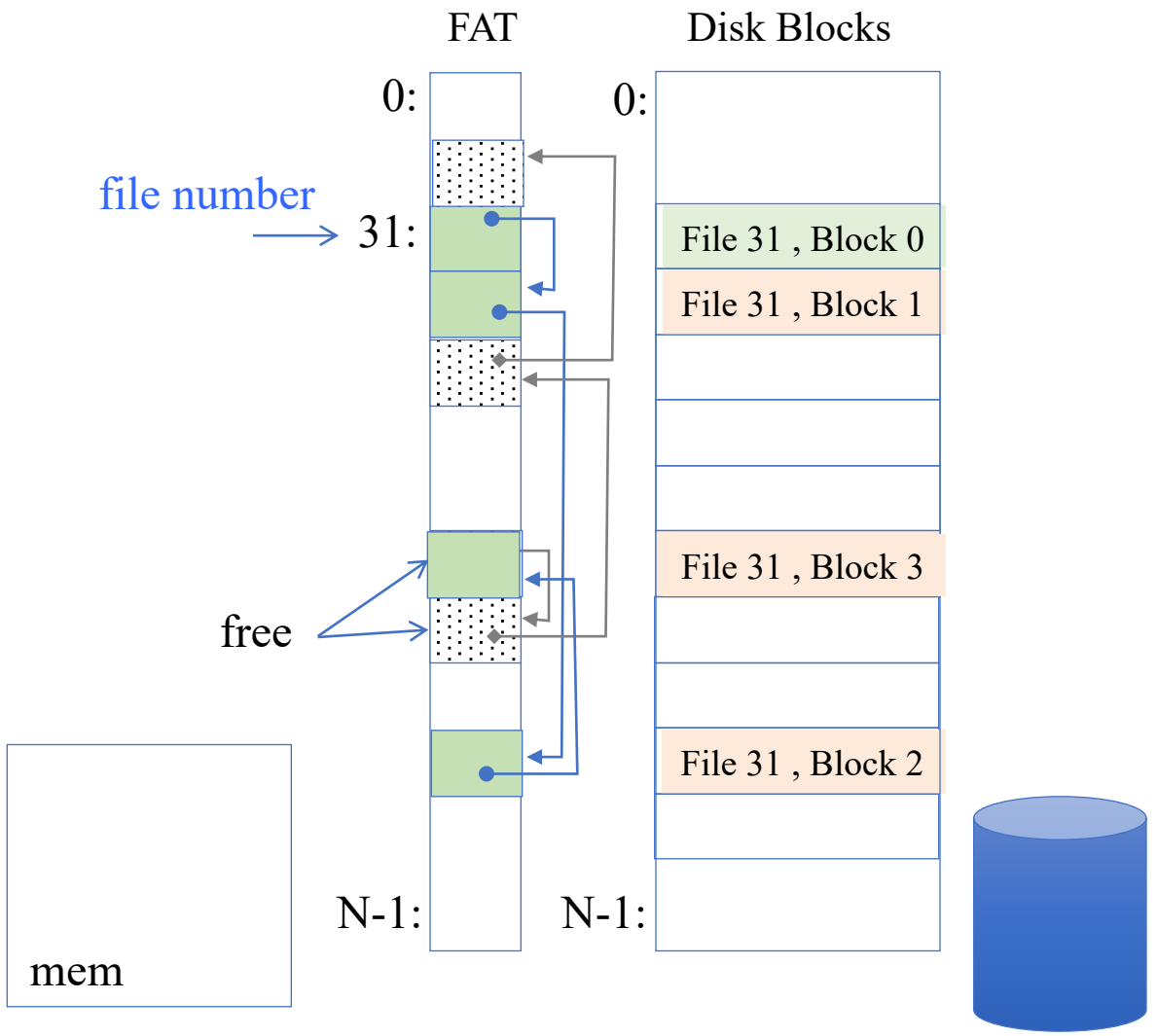
FAT (File Allocation Table)

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- File offset (o = B:x)
- Follow list to get block #
- Unused blocks ⇔ FAT free list



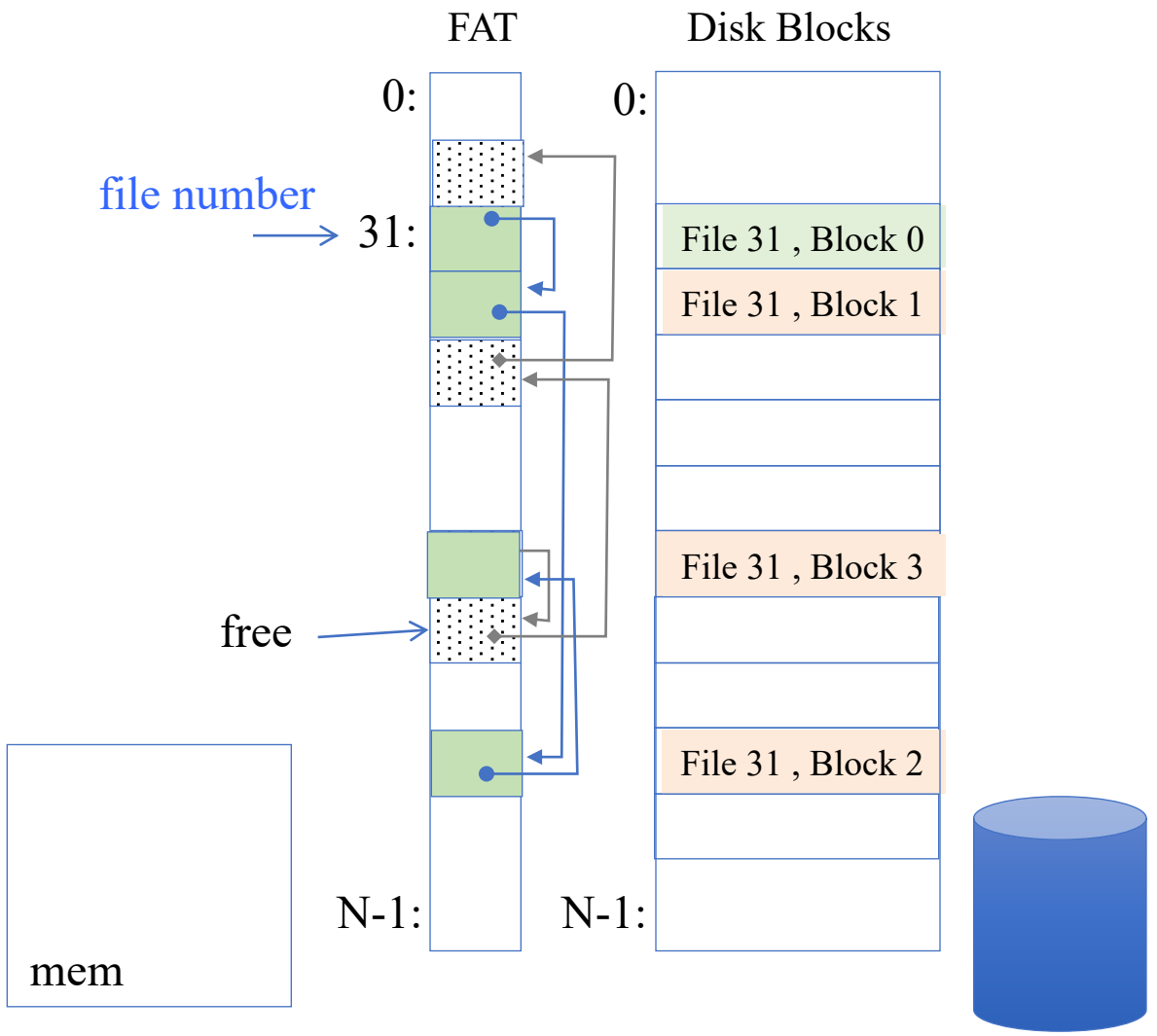
FAT (File Allocation Table)

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- File offset (o = B:x)
- Follow list to get block #
- Unused blocks ⇔ FAT free list
- Example: file_write(51, <3, y>)



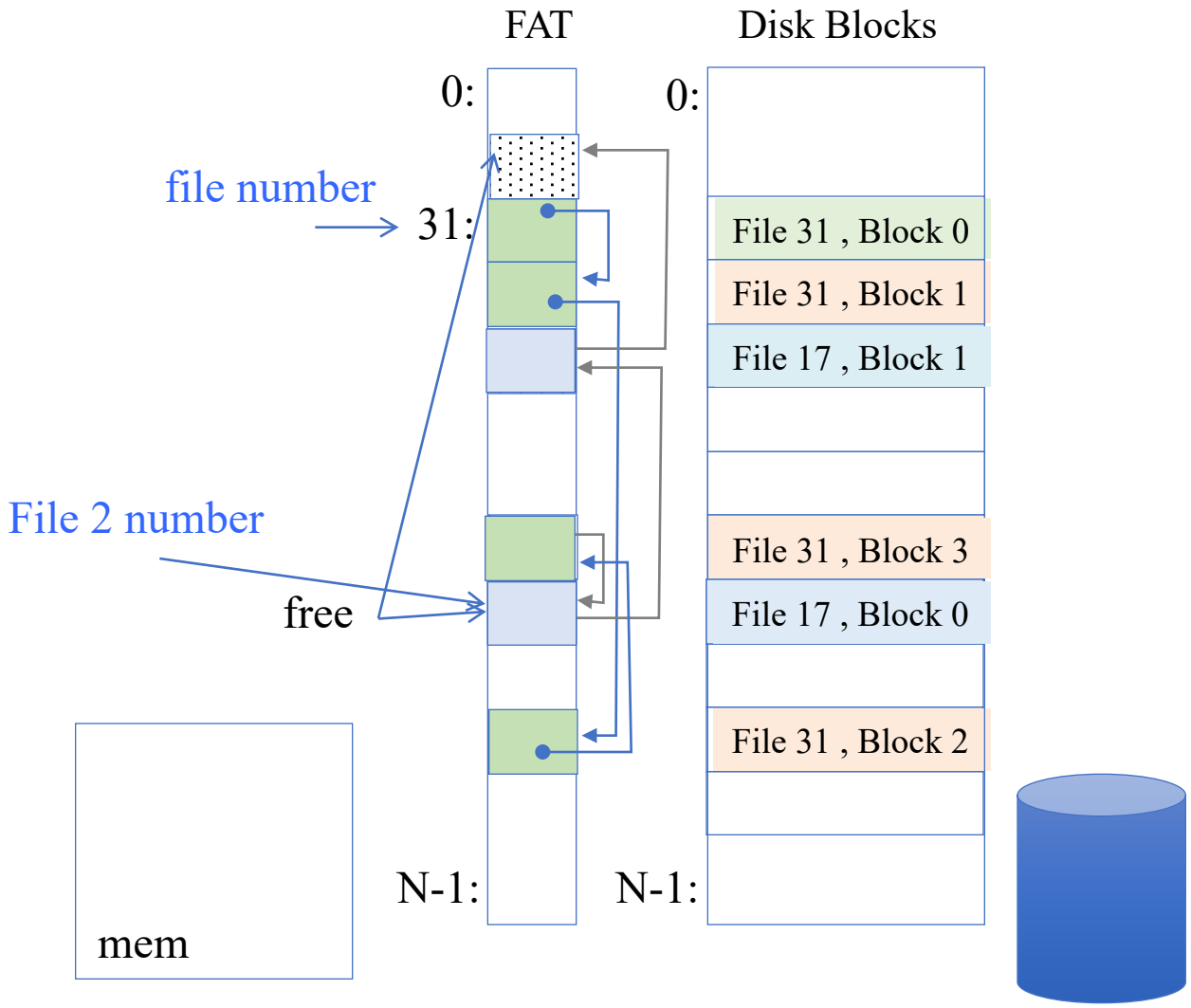
FAT (File Allocation Table)

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- File offset (o = B:x)
- Follow list to get block #
- Unused blocks ⇔ FAT free list
- Grow file by allocating free blocks and linking them in



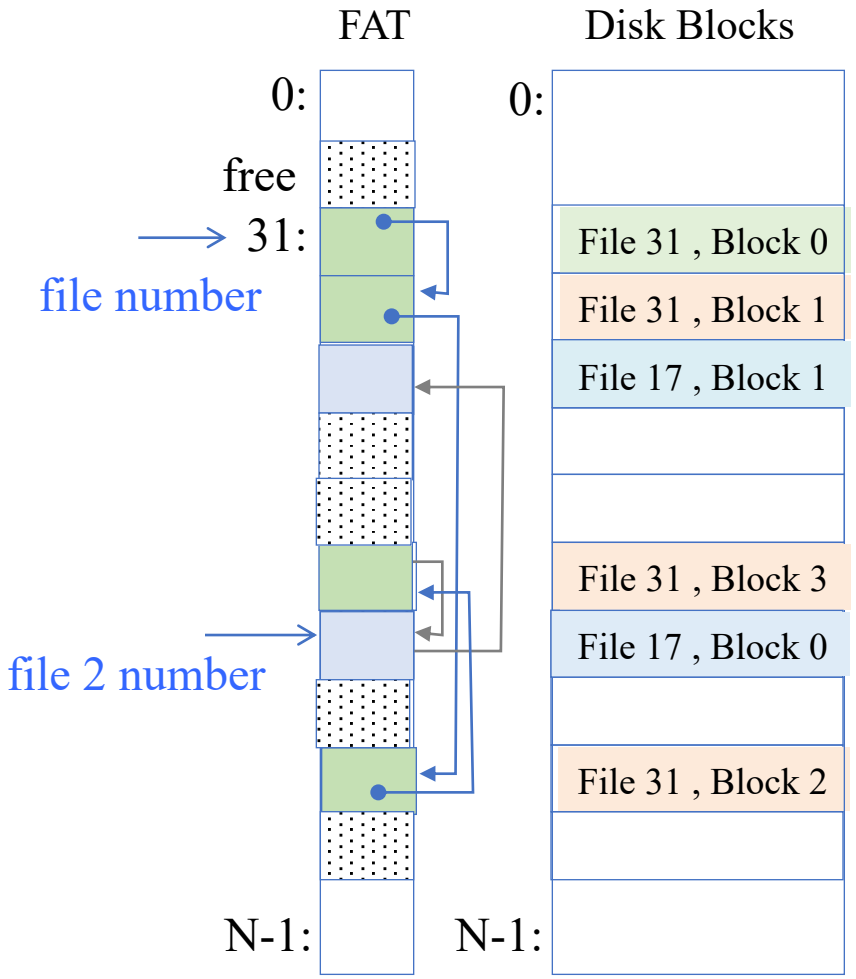
FAT (File Allocation Table)

- File is collection of disk blocks
- FAT is linked list 1-1 with blocks
- File Number is index of root of block list for the file
- Grow file by allocating free blocks and linking them in
- Example Create file, write, write



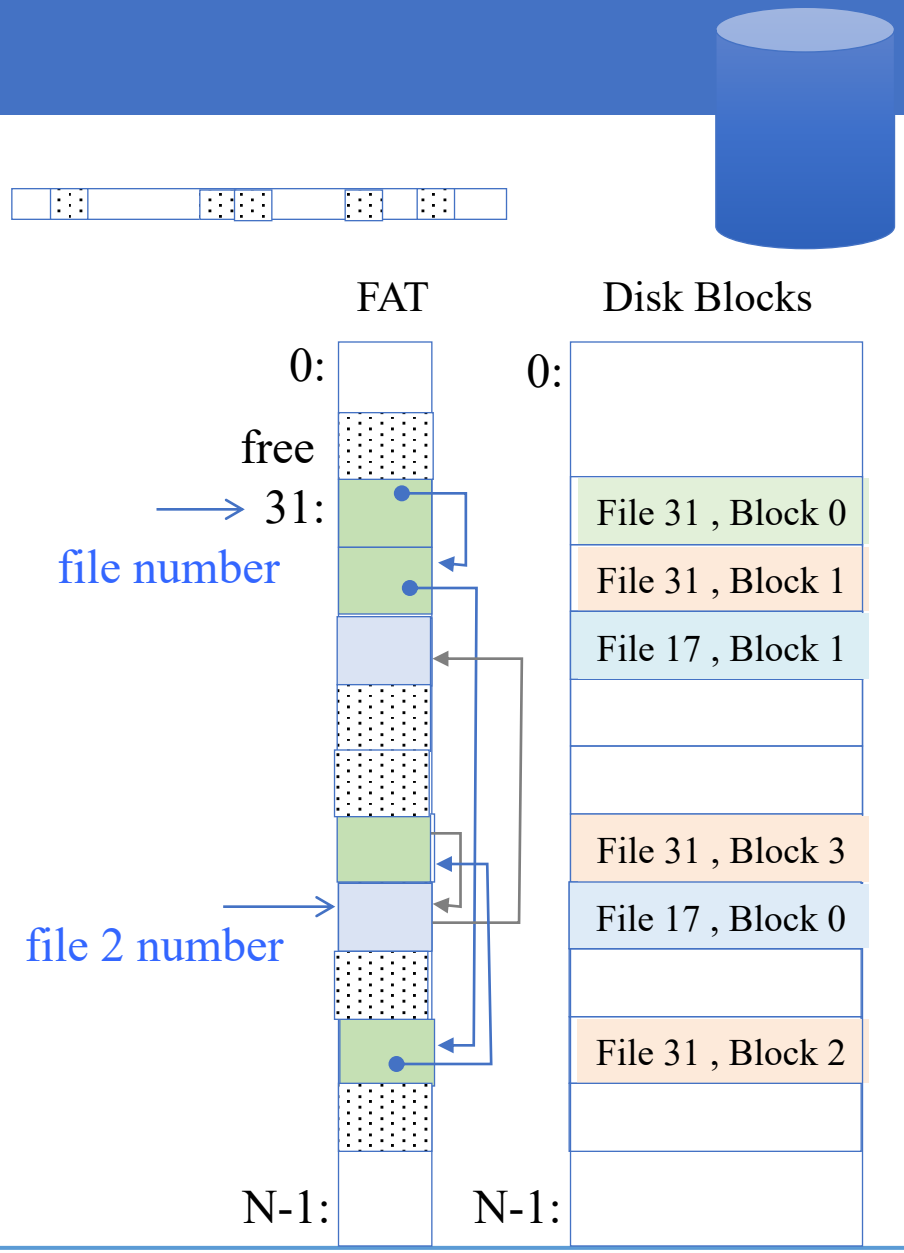
FAT Assessment

- Used in DOS, Windows, thumb drives, …
- Where is FAT stored ?
- On Disk, restore on boot, copy in memory
- What happens when you format a disk?
- Zero the blocks, link up the FAT free
- Simple

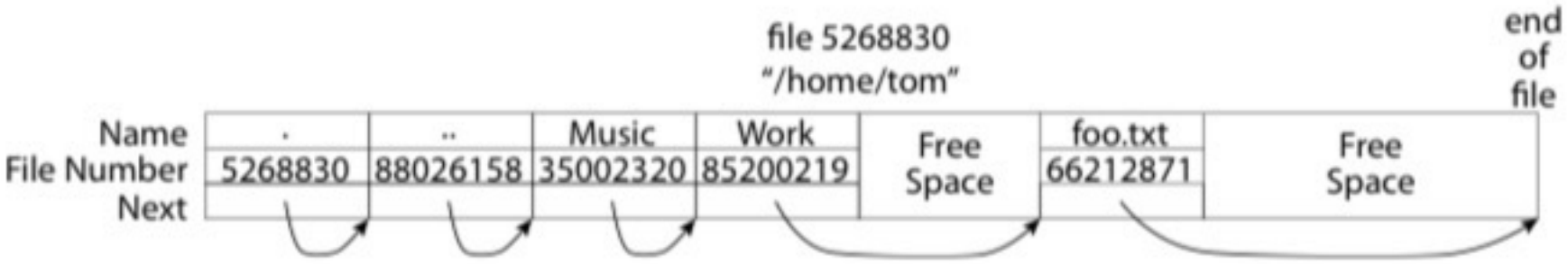


FAT Assessment

- Time to find block (large files) ??
- Free list usually just a bit vector
- Next fit algorithm
- Block layout for file ???
- Sequential Access ???
- Random Access ???
- Fragmentation ???
- Small files ???
- Big files ???



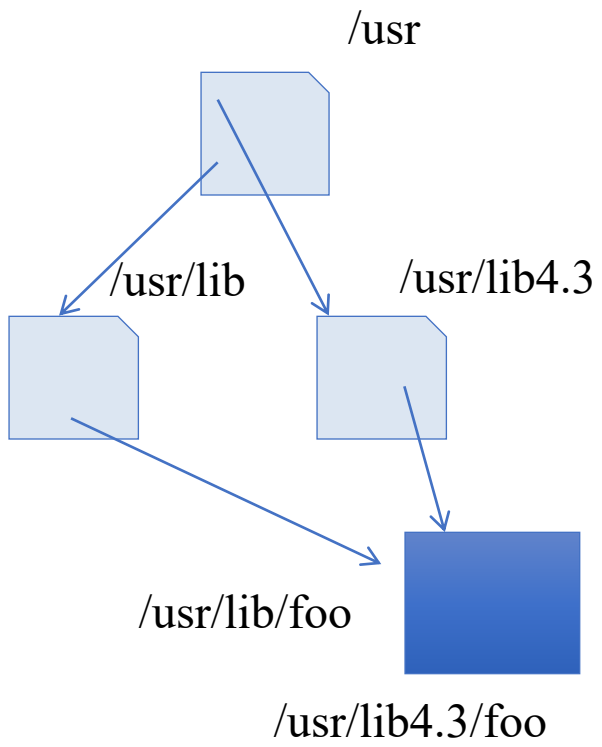
What about the Directory?



- Essentially a file containing <file_name: file_number> mappings
- Free space for new entries
- In FAT: attributes kept in directory (!!!)
- Each directory a linked list of entries
- Where do you find root directory ("/")

Bit more on directories

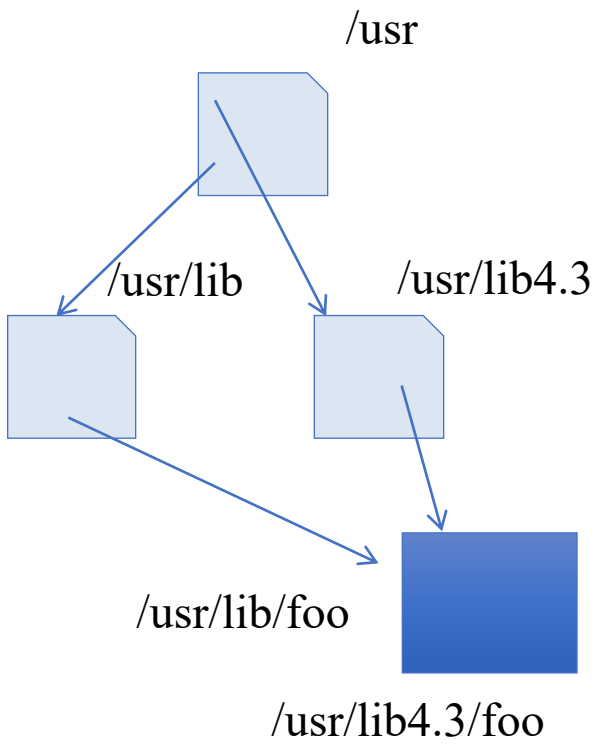
- Stored in files, can be read, but don't
  - System calls to access directories
  - Open / Creat traverse the structure
  - mkdir /rmdir add/remove entries
  - Link / Unlink
    - Link existing file to a directory
      - Not in FAT !
    - Forms a DAG



- libc support
  - DIR * opendir (const char *dirname)
  - struct dirent * readdir (DIR *dirstream)
  - int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)

When can a file be deleted ?

- Maintain reference count of links to the file.
- Delete after the last reference is gone.



Big FAT security holes

- FAT has no access rights
- FAT has no header in the file blocks
- Just gives and index into the FAT
  - (file number = block number)

Unix File System

Characteristics of Files

- Most files are small
- Most of the space is occupied by the rare big ones

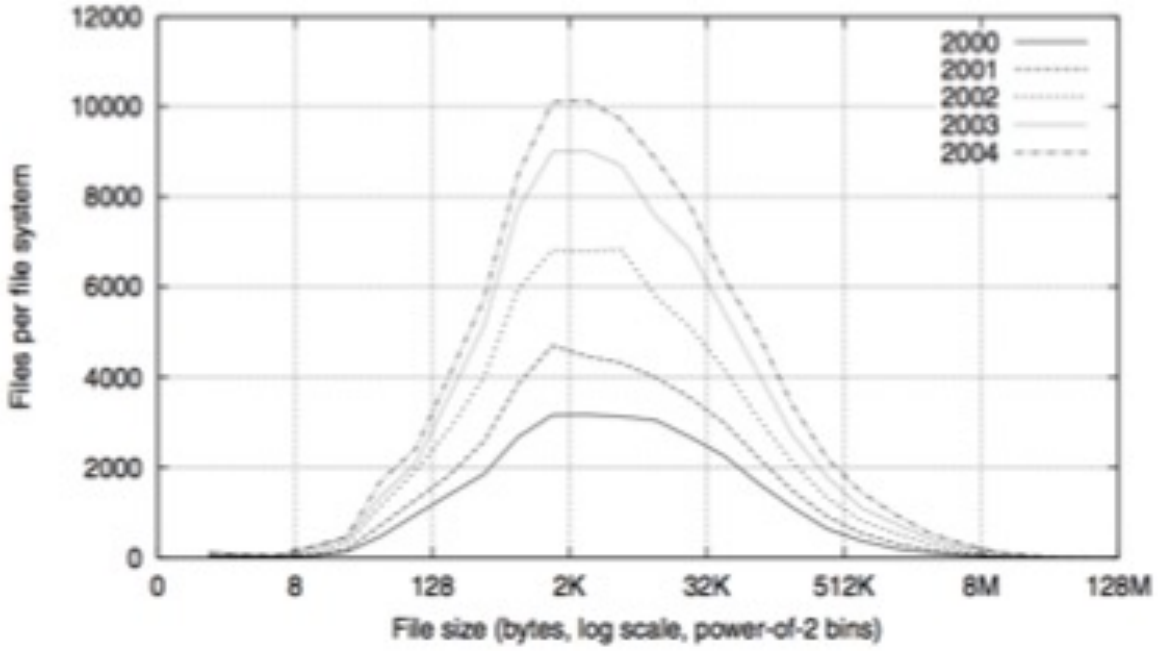


Fig. 2. Histograms of files by size.

A Five-Year Study of File-System Metadata

NITIN AGRAWAL
University of Wisconsin, Madison
and
WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH
Microsoft Research

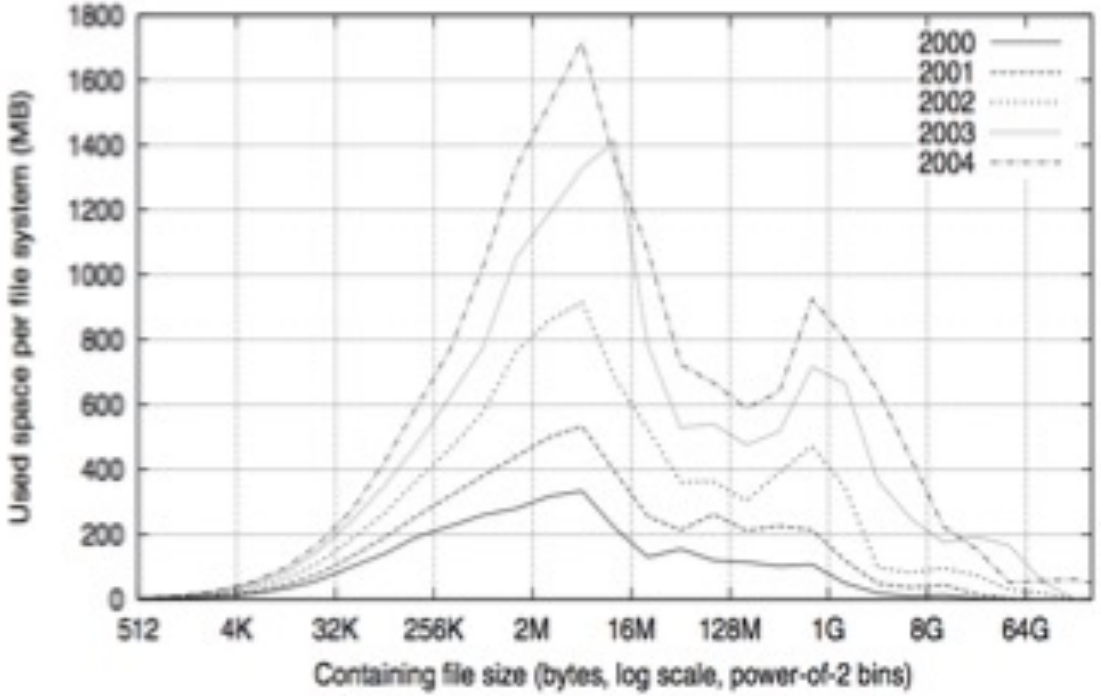
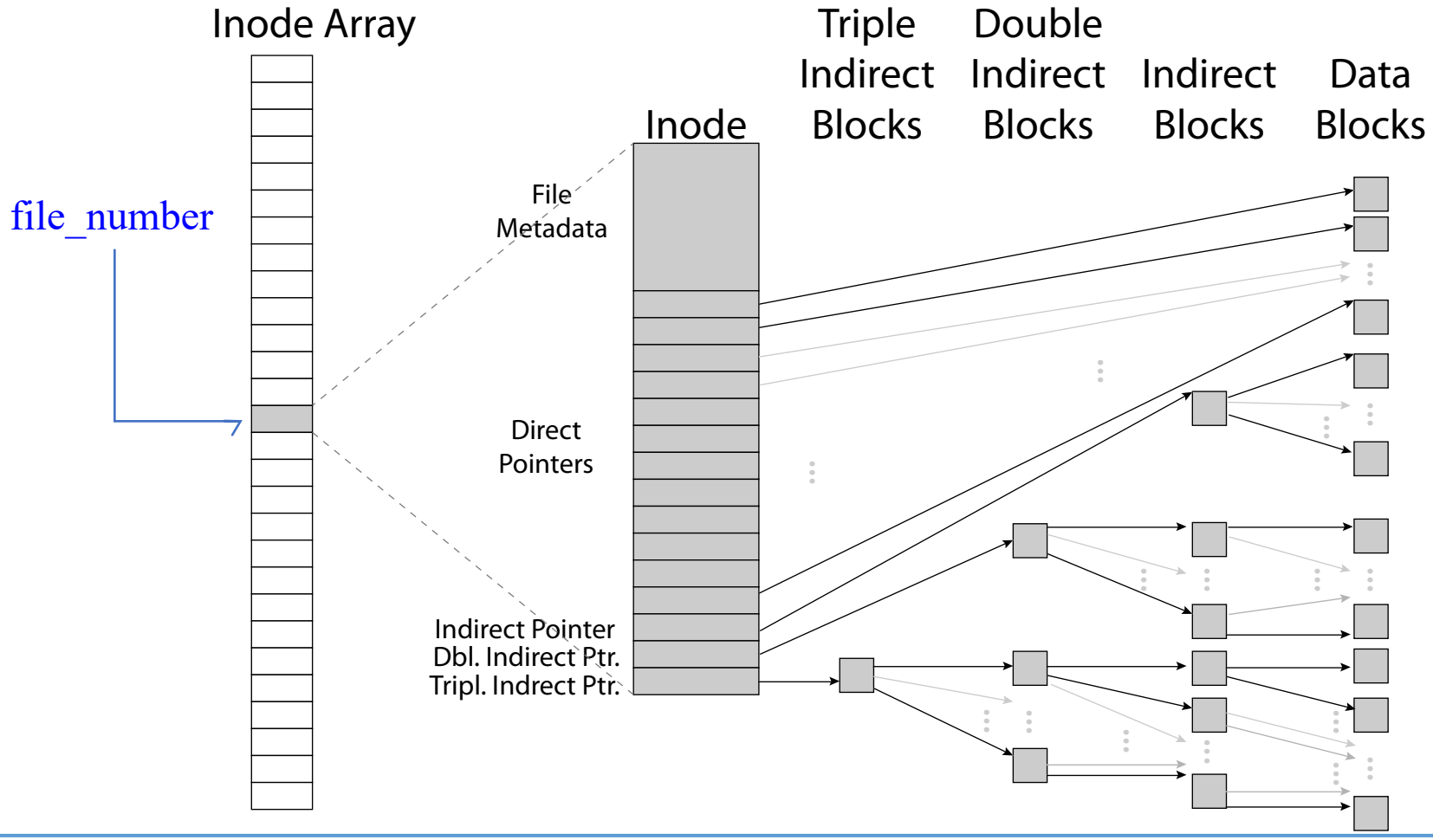


Fig. 4. Histograms of bytes by containing file size.

So what about a “real” file system

- Meet the inode



Unix File System

- File Number is index into inode arrays
- Multi-level index structure
  - Great for little to large
  - Asymmetric tree with fixed sized blocks
- Metadata associated with the file
  - Rather than in the directory that points to it
- Locality Heuristics
  - Block group placement
  - Reserve space
- Scalable directory structure

An “almost real” file system

- Pintos: src/filesys/file.c, inode.c

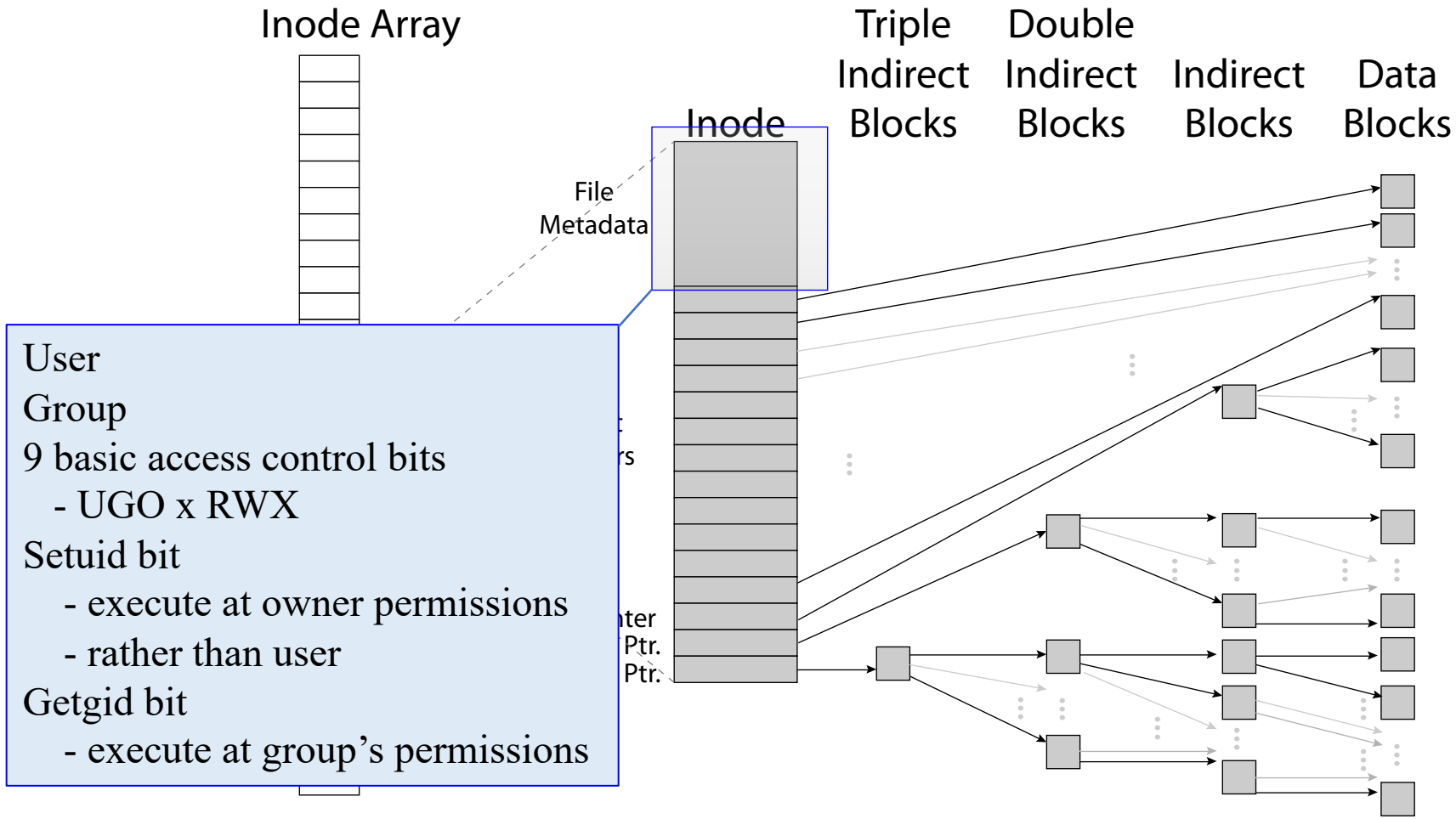
```
/* An open file. */
struct file
  {
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;            /* Has file_deny_write() been called? */
  };
```

```
/* In-memory inode. */
struct inode
  {
    struct list_elem elem;              /* Element in inode list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
    bool removed;                       /* True if deleted, false otherwise. */
    int deny_write_cnt;                 /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;             /* Inode content. */
  };
```

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
  };
```

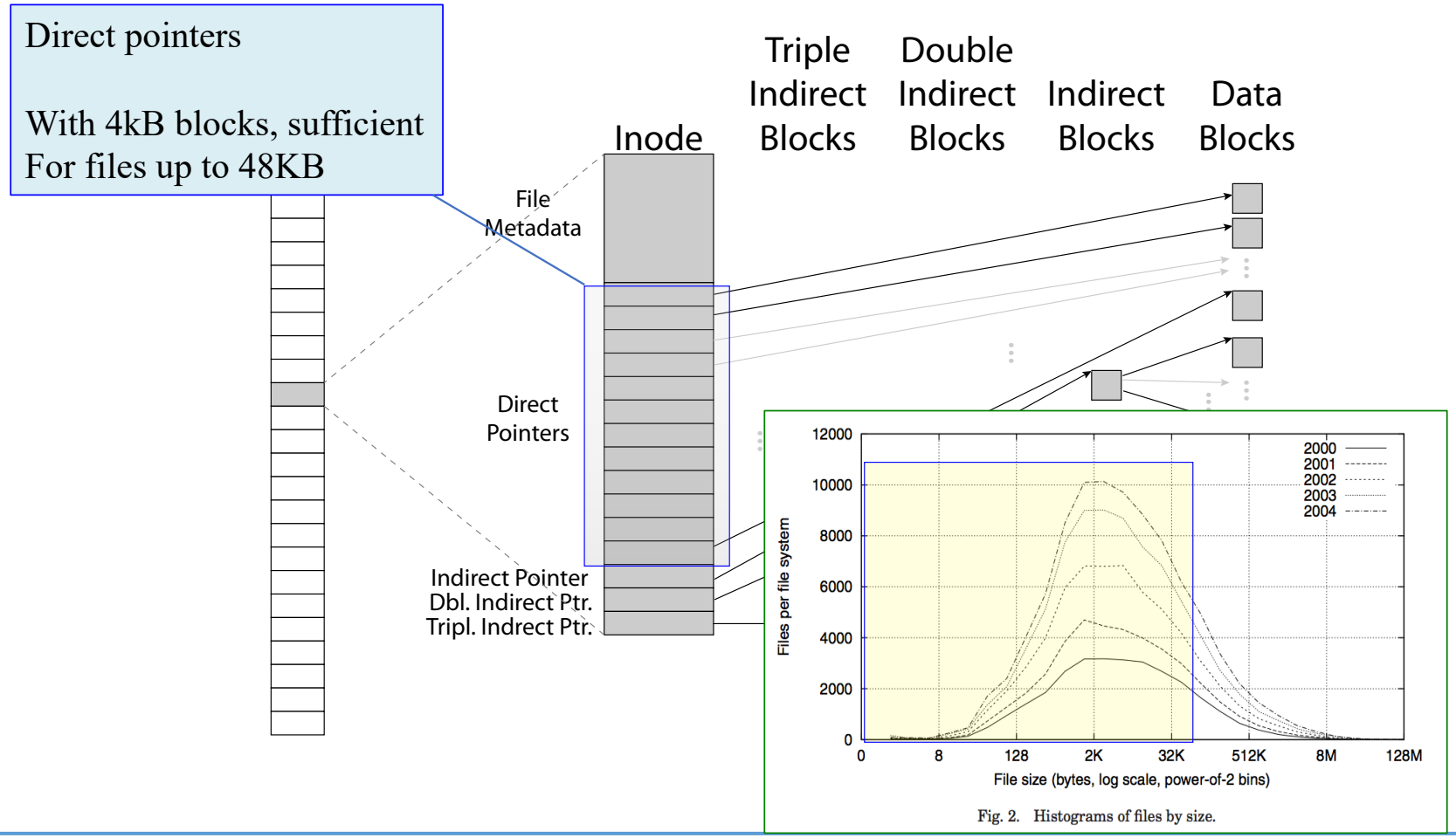
UFS: File Attributes

- Inode has metadata



UFS: Data Storage

- Small files: 12 pointers direct to data blocks



UFS: Data Storage

- Large files: 1,2,3 level indirect pointers

Indirect pointers
- point to a disk block containing only pointers
- eg. 4 kB blocks => 1024 pointers
  => 4 MB @ level 2
  => 4 GB @ level 3
  => 4 TB @ level 4

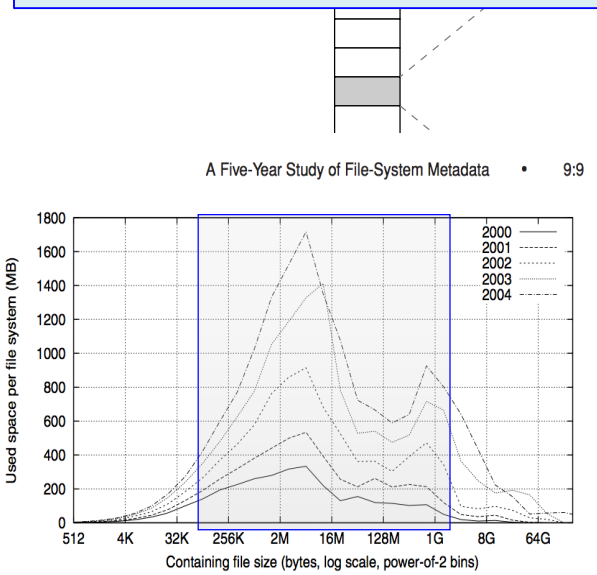
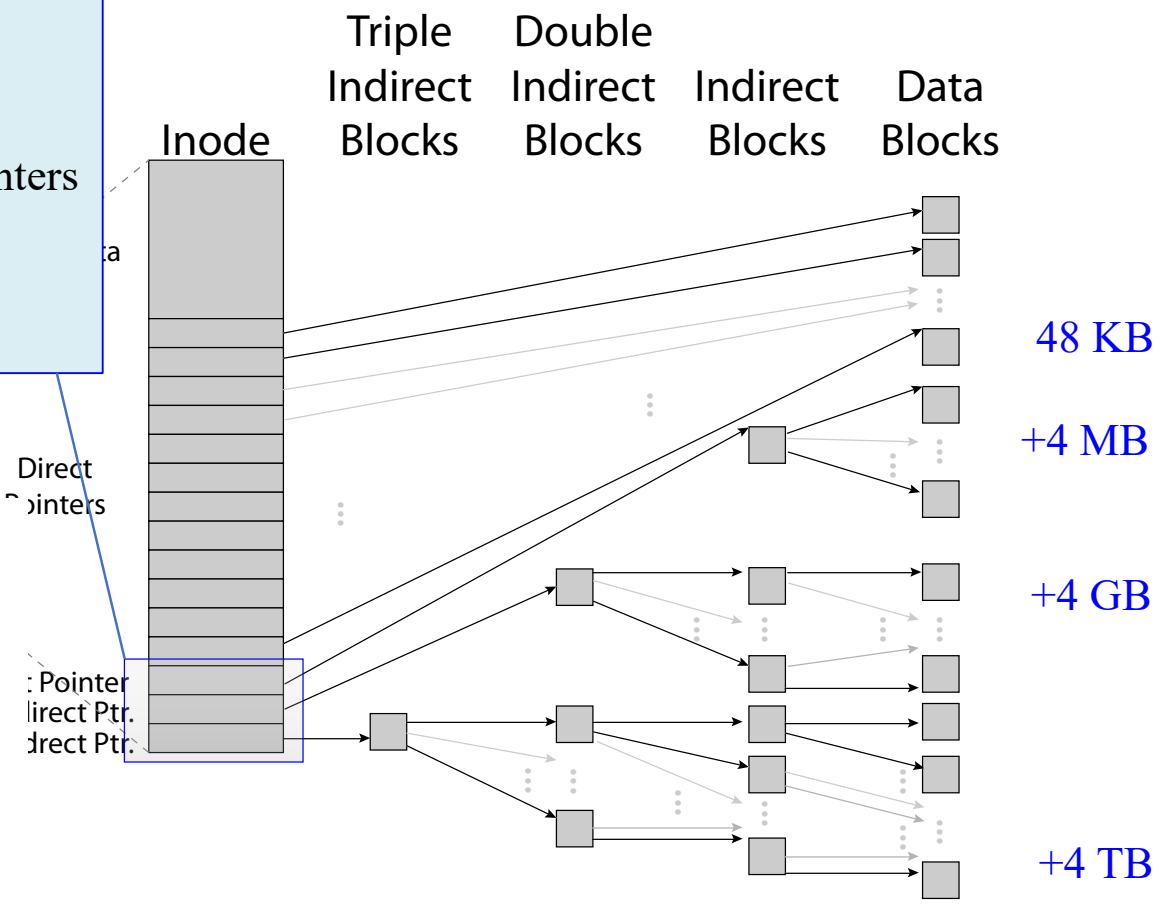


Fig. 4. Histograms of bytes by containing file size.



Freespace Management

- Bit vector with a bit per storage block
- Stored at a fixed location within the file system

Where are inodes stored?

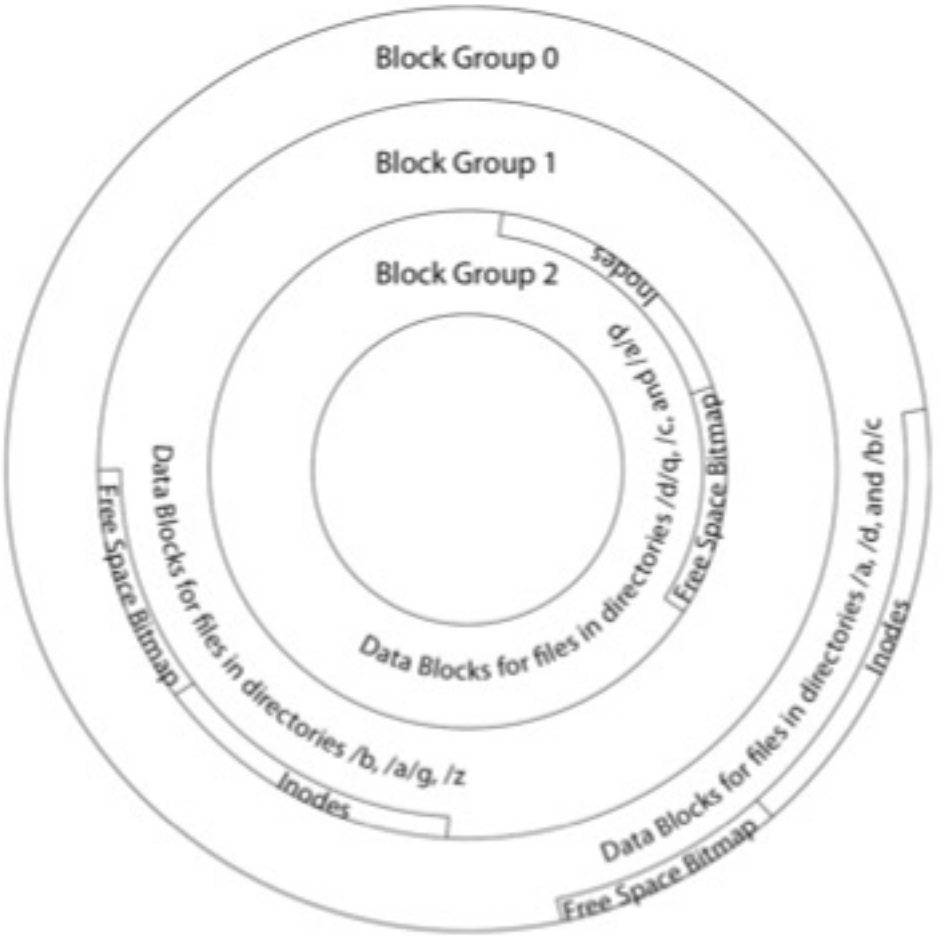
- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored anywhere near the data blocks.
  - To read a small file, seek to get header, seek back to data.
  - Fixed size, set when disk is formatted.
  - At formatting time, a fixed number of inodes were created (They were each given a unique nu mber, called an "inumber")

Where are inodes stored?

- Later versions of UNIX moved the header information to be closer to t he data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the f ile
    (makes an `ls` of that directory run fast).
  - Pros:
    - UNIX BSD 4.2 puts a portion of the file header array on each of many cylinders. For small directorie s, can fit all data, file headers, etc. in same cylinder ⇒ no seeks!
    - File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched fro m disk at same time
    - Reliability: whatever happens to the disk, you can find many of the files (even if directories disconne cted)
  - Part of the Fast File System (FFS)
    - General optimization to avoid seeks

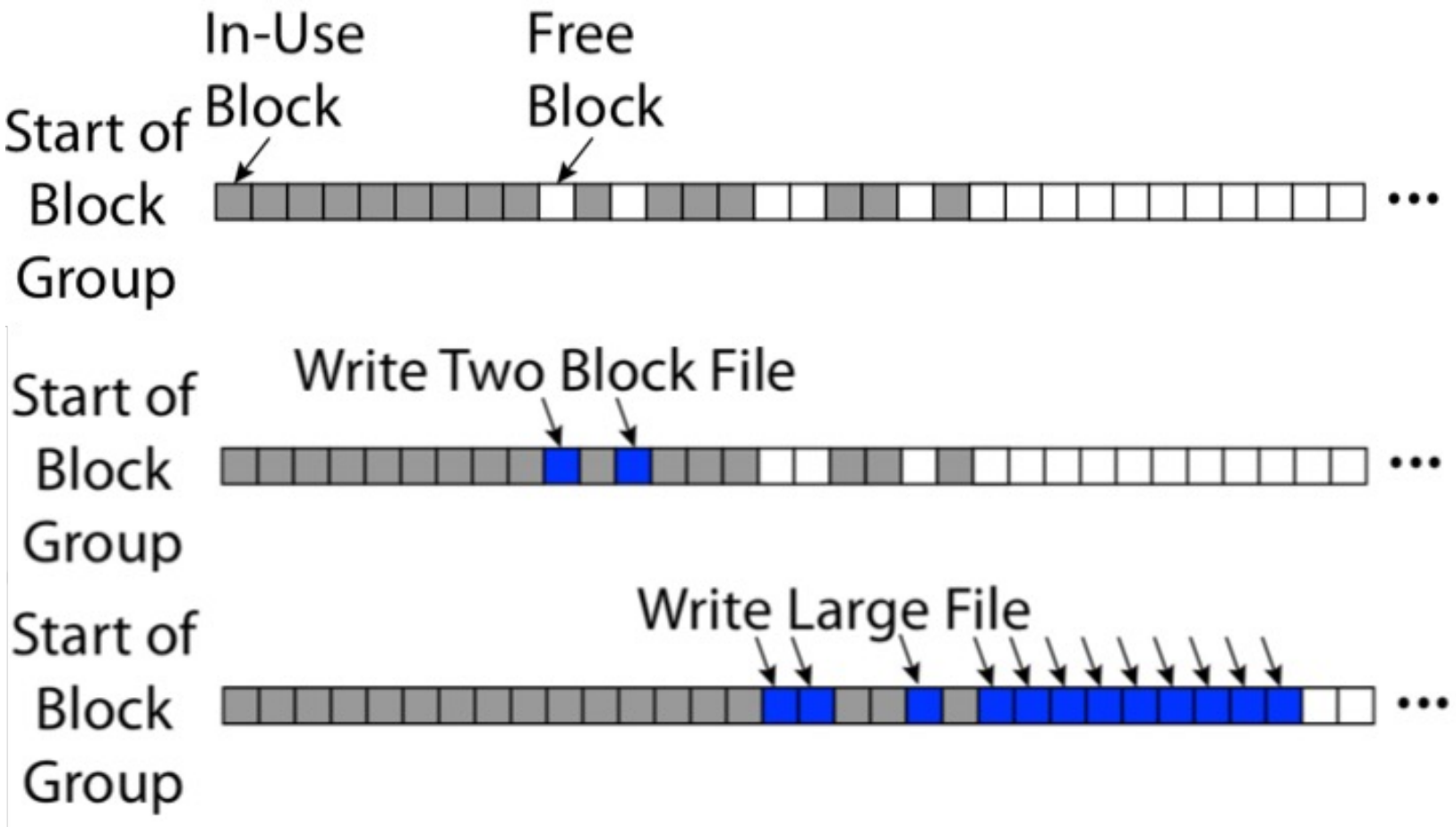
FastFileSystem considers Locality: Block Groups

- File system volume is divided into a set of bloc k groups
  - Close set of tracks
- File data blocks, metadata, and free space are interleaved within block group
  - Avoid huge seeks between user data and system str ucture
- Put directory and its files in common block gro up
- First-Free allocation of new file block
  - Few little holes at start, big sequential runs at end o f group
  - Avoids fragmentation
  - Sequential layout for big
- Reserve space in the BG



FFS First Fit Block Allocation

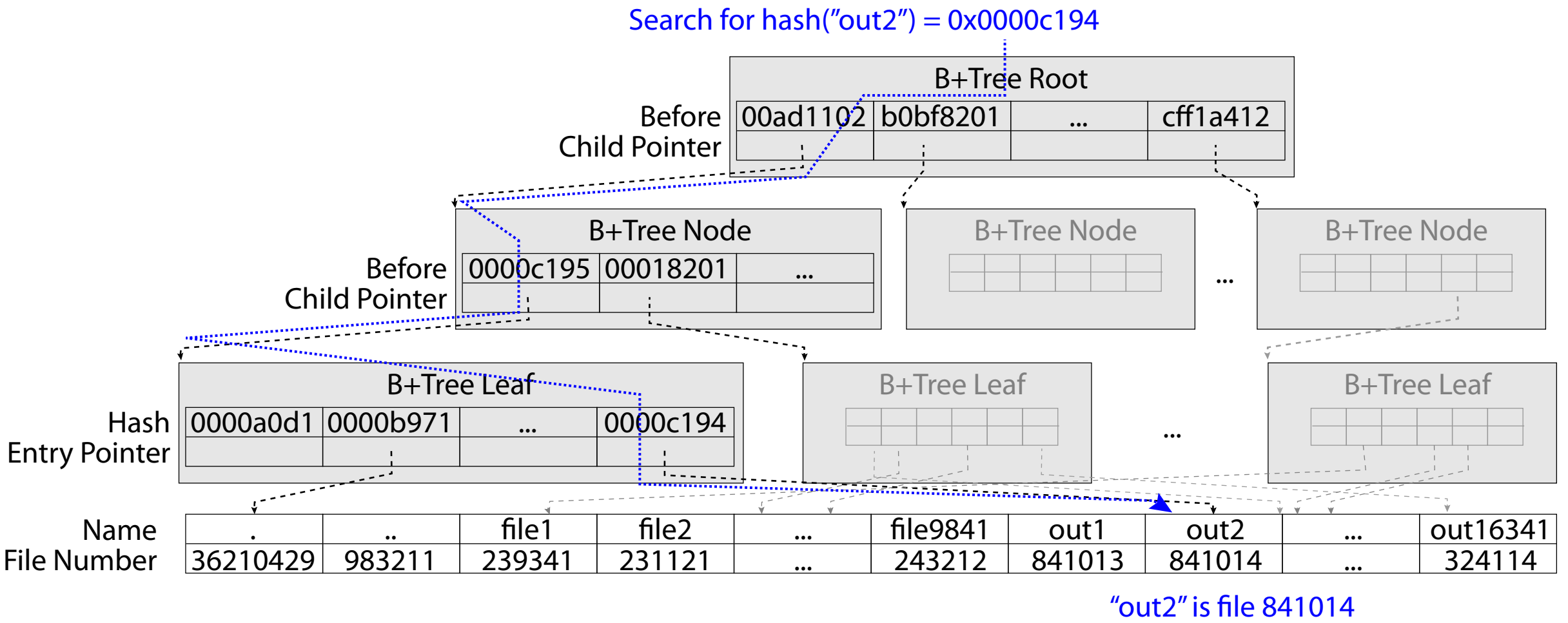
- Fills in the small ho les at the start of block group
- Avoids fragmentati on, leaves contiguous free space at end



FFS

- Pros
  - Efficient storage for both small and large files
  - Locality for both small and large files
  - Locality for metadata and data
- Cons
  - Inefficient for tiny files (a 1 byte file requires both an inode and a data bloc k)
  - Inefficient encoding when file is mostly contiguous on disk (no equivalent to superpages)
  - Need to reserve 10-20% of free space to prevent fragmentation

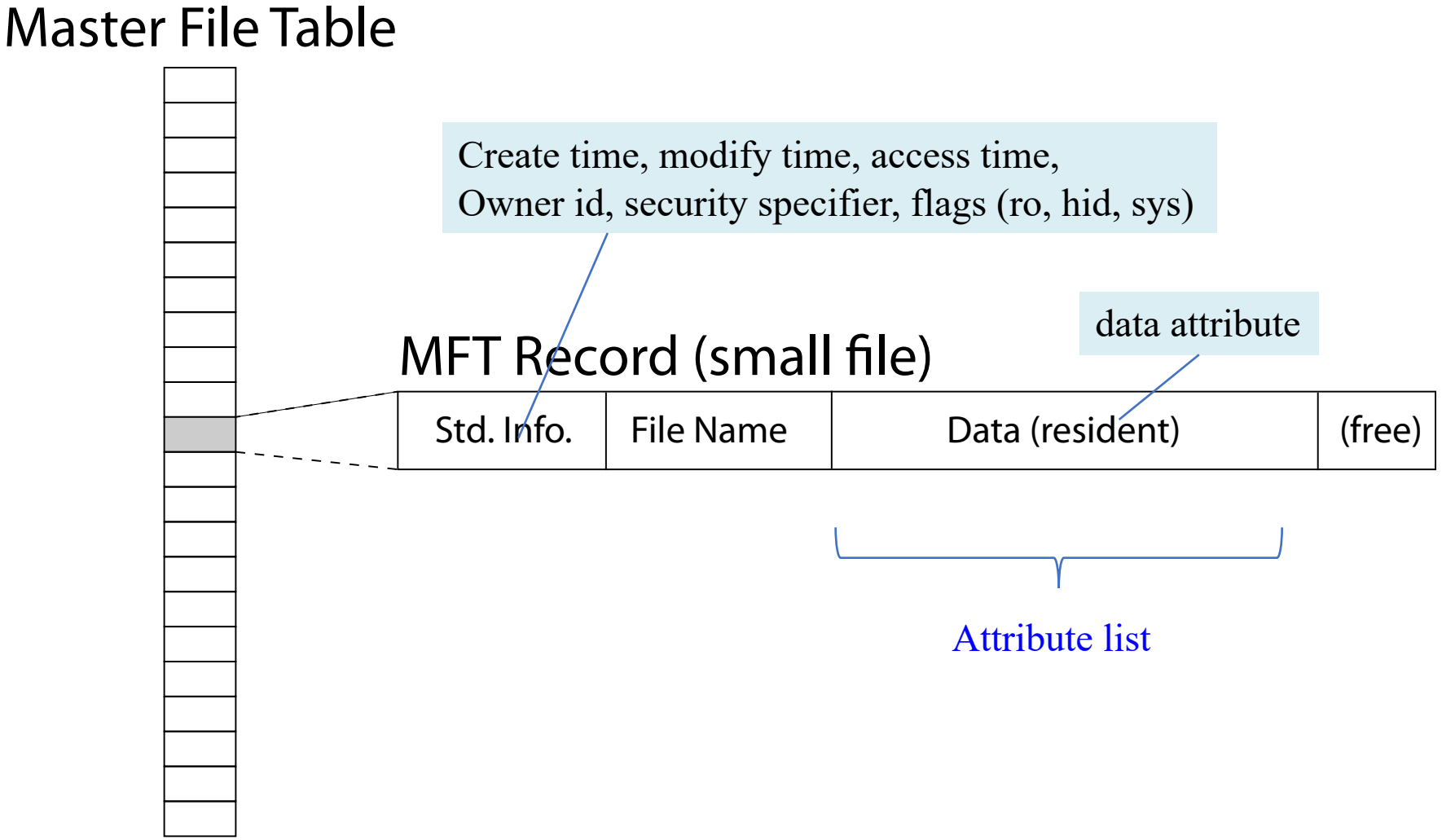
Large Directories: B-Trees



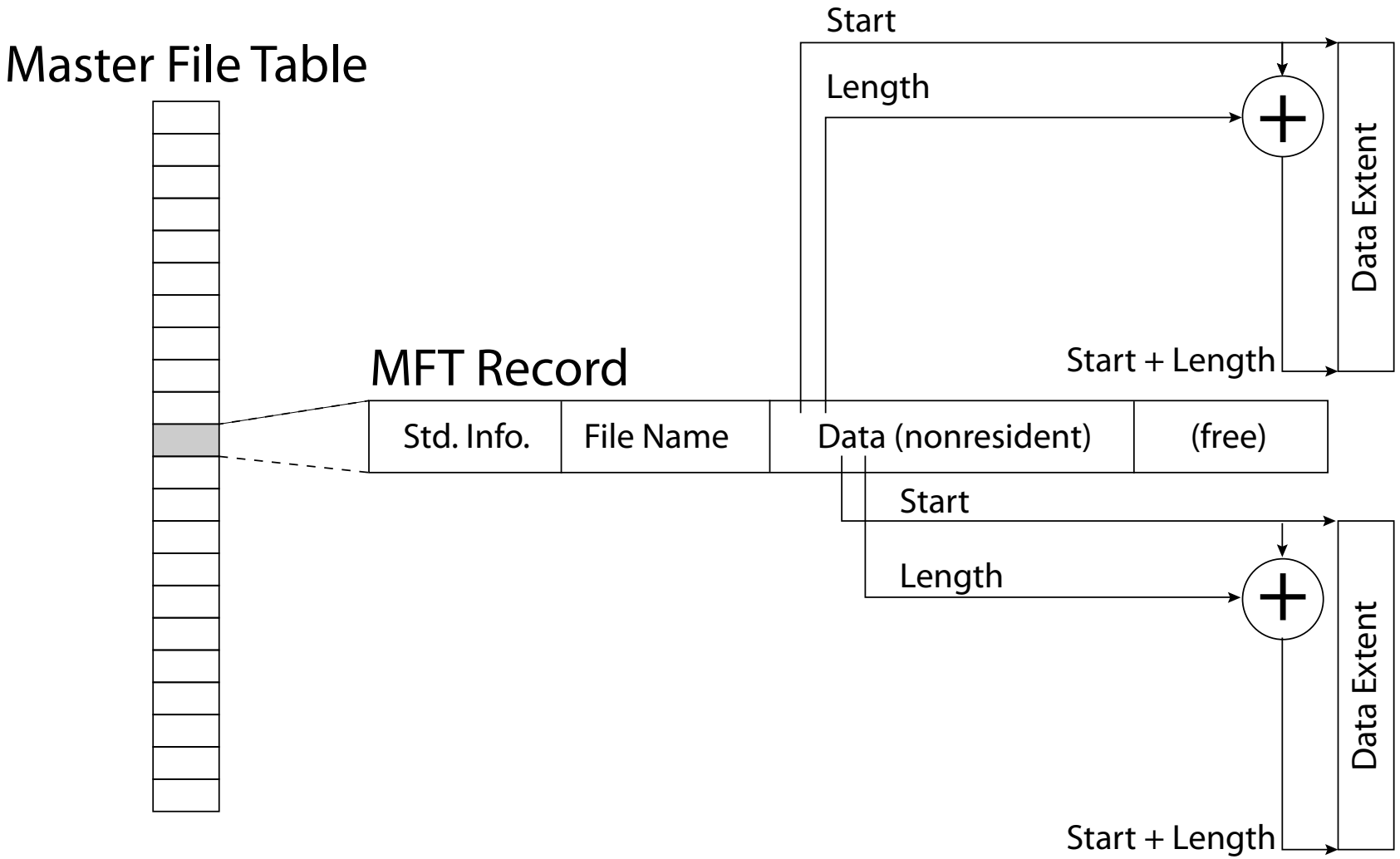
NTFS

- Master File Table
  - Flexible 1KB storage for metadata and data
  - Variable-sized attribute records (data or metadata)
  - Extend with variable depth tree (non-resident)
- Extents - variable length contiguous regions
  - Block pointers cover runs of blocks
  - Similar approach in linux (ext4)
  - File create can provide hint as to size of file
- Journalling for reliability
  - Discussed next week

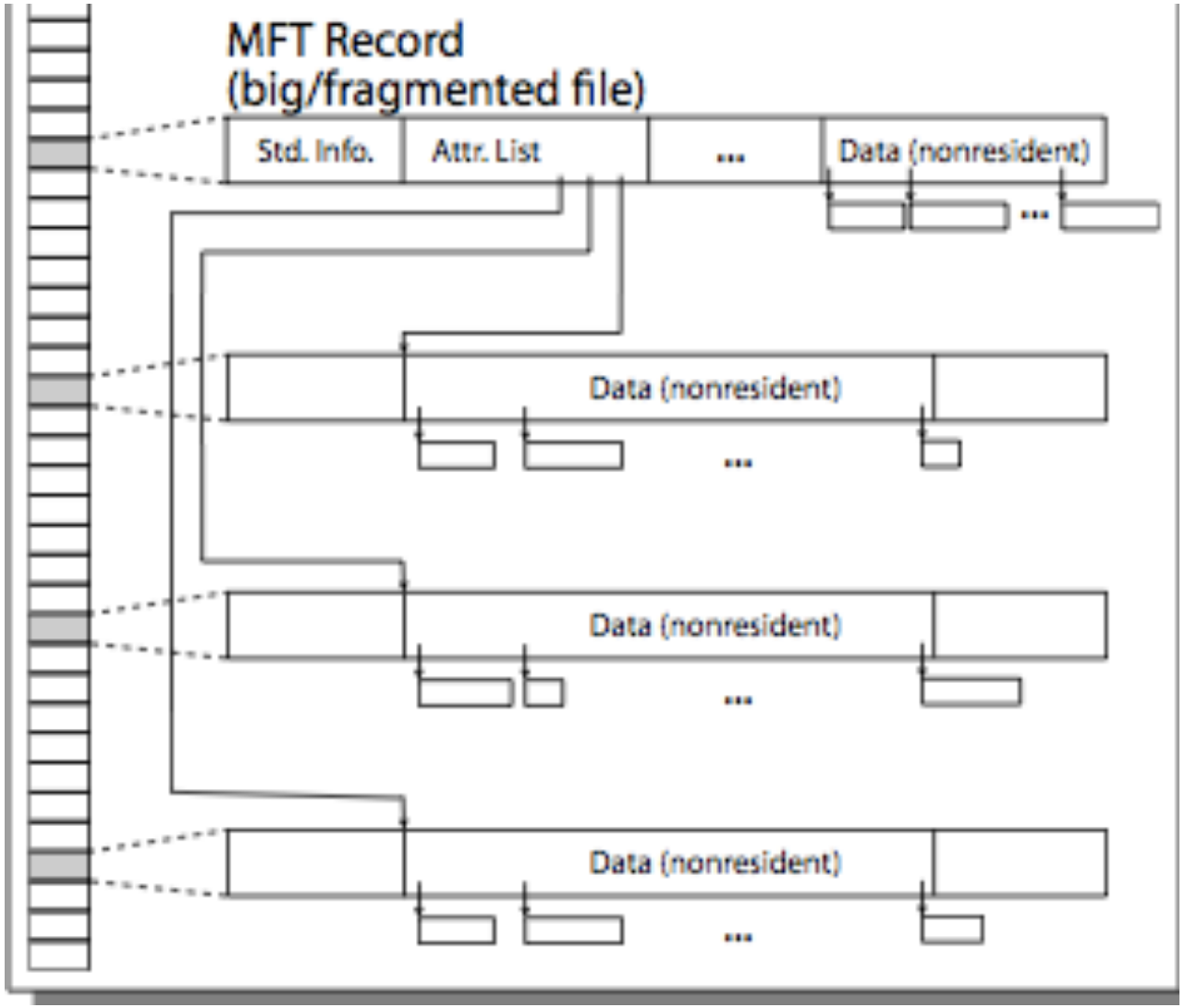
NTFS Small File



NTFS Medium File

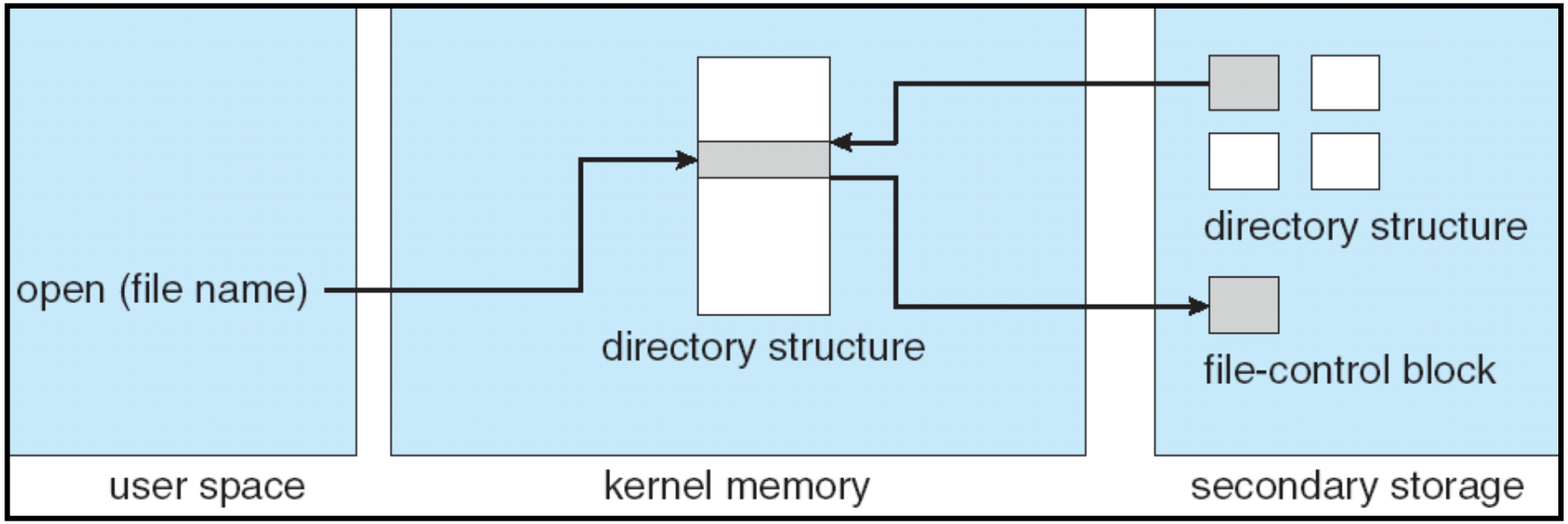


NTFS Multiple Indirect Blocks



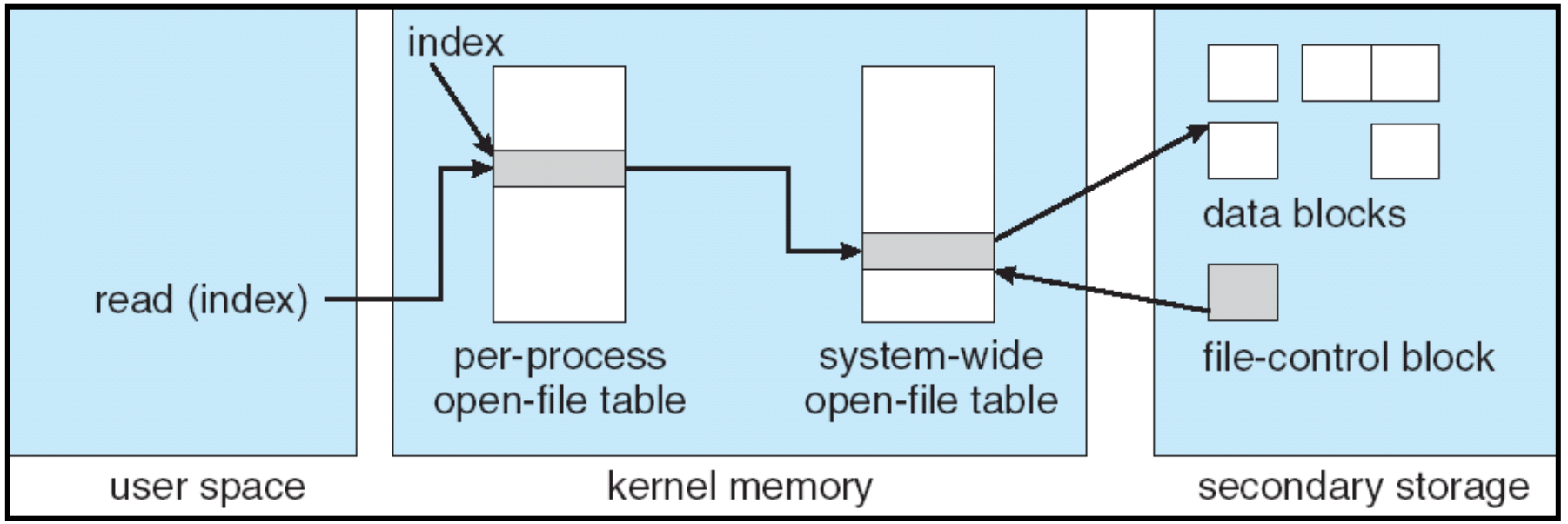
In-Memory File System Structures

- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called “file handle”) in open-file table



In-Memory File System Structures

- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes



Towards Copy-on-Write

- *Files* are for durable storage AND flexible process-independent, pro tected namespace
- Files grow incrementally as written
  - Update-in-place file systems start with a basic chunk and append (possibly l arger) chunks as file grows
  - Transition from random access to large sequential
- *Disks trends:* huge and cheap, high startup
- *Design / Memory trends:* cache everything
  - Reads satisfied from cache, buffer multiple writes and do them all together
- Application trends: make multiple related changes to a file and co mmit all or nothing

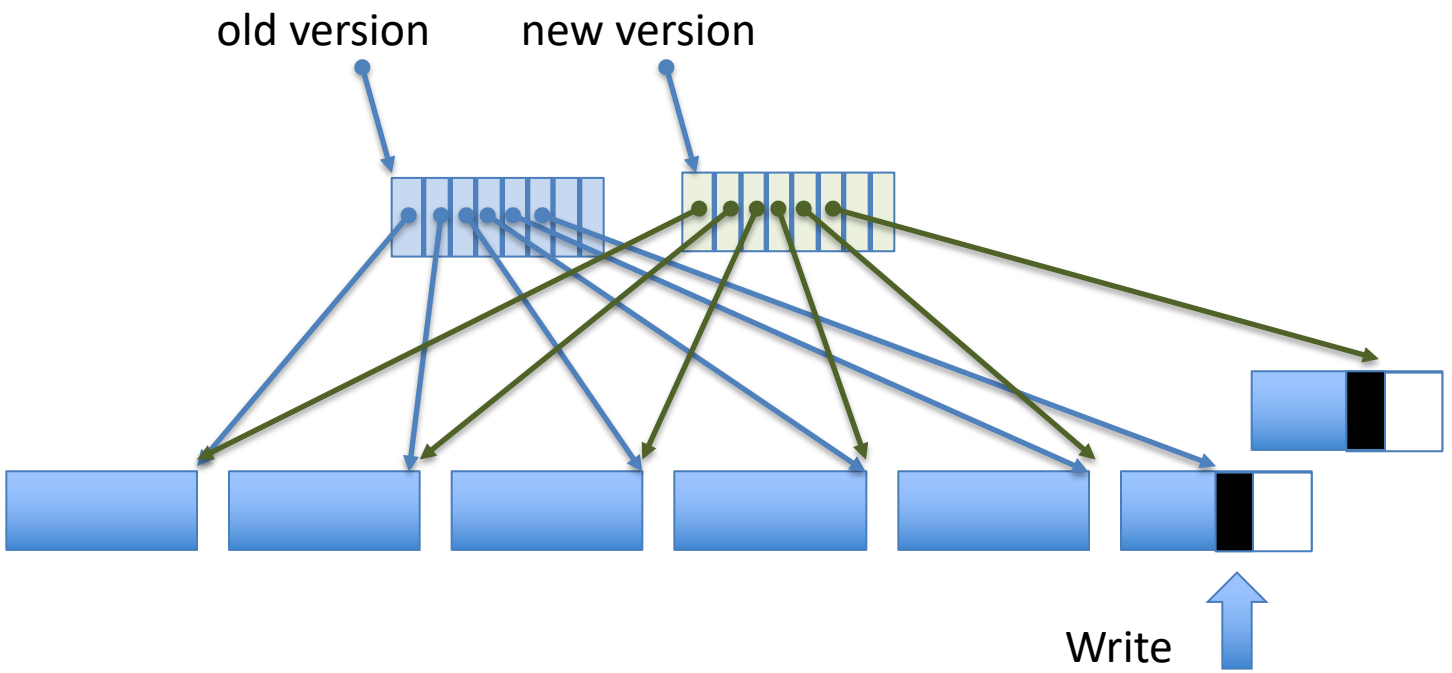
Emulating COW @ user level

- Transform file **foo** to a new version
- Open/Create a new file **foo.v**
  - where v is the version #
- Do all the updates based on the old **foo**
  - Reading from **foo** and writing to **foo.v**
  - Including copying over any unchanged parts
- Update the link
  - ln -f foo foo.v

- Does it work?

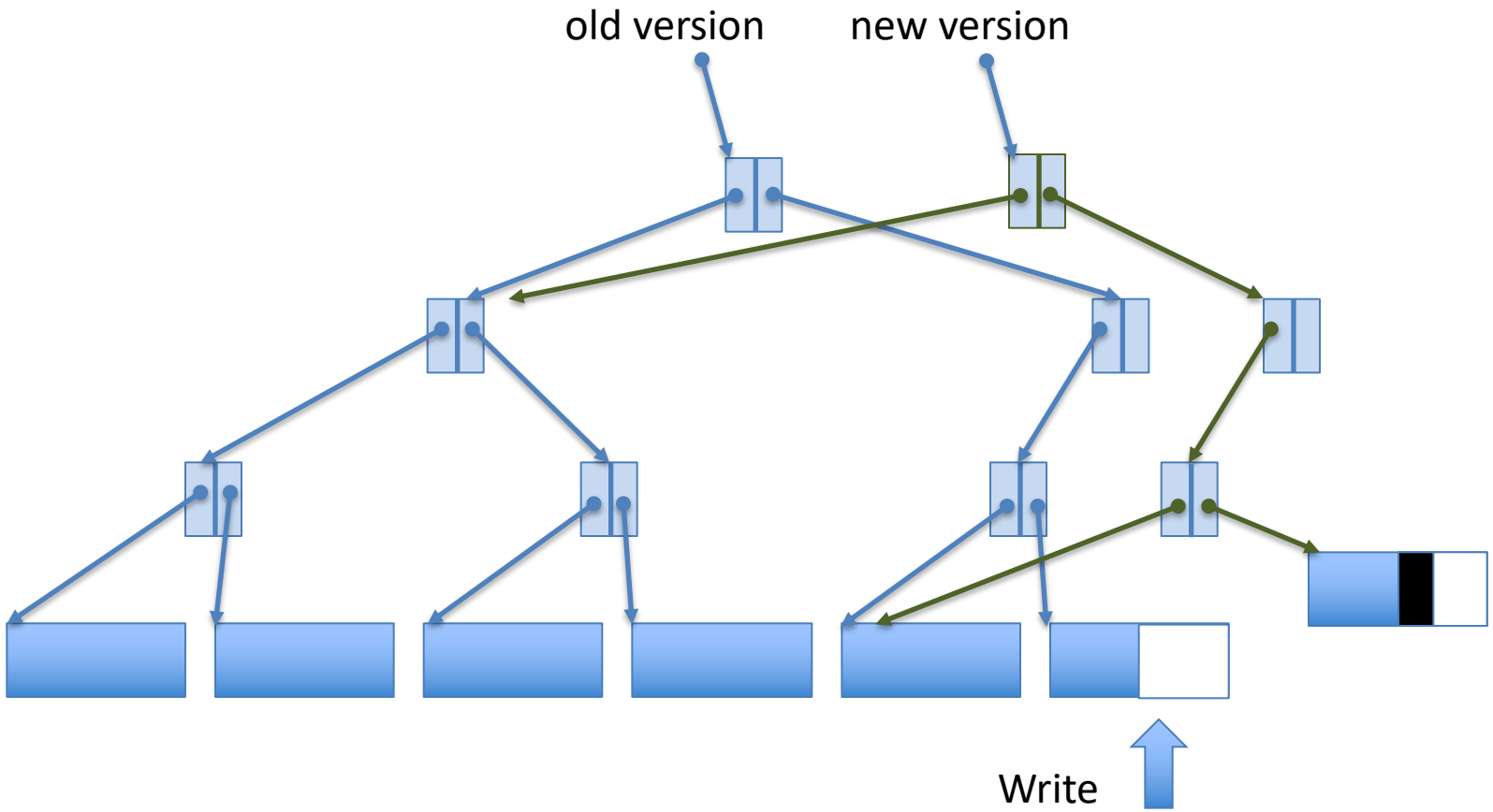
Creating a New Version

- If file represented as a tree of blocks, just need to update the leadi ng fringe



Creating a New Version

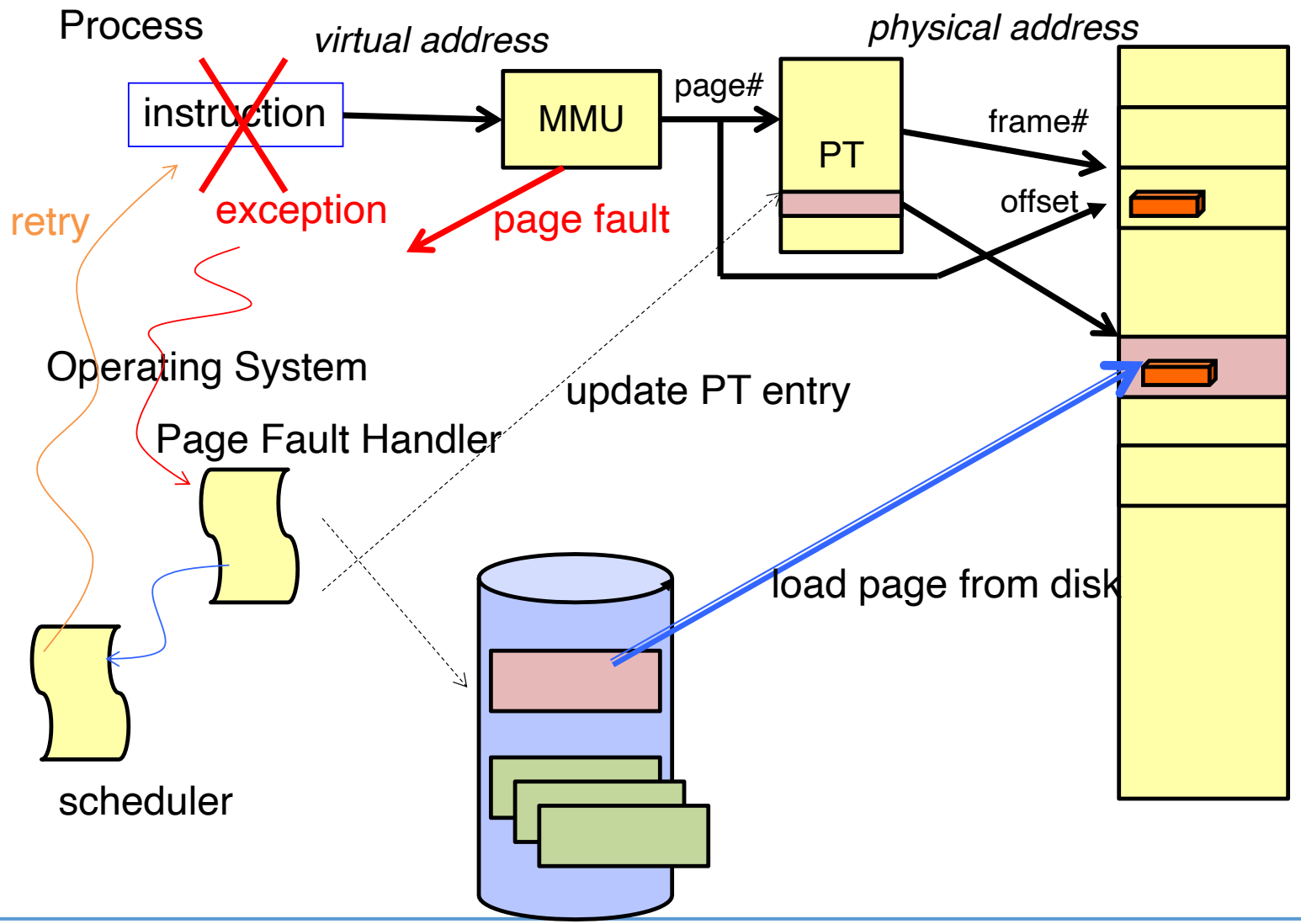
- If file represented as a tree of blocks, just need to update the leadi ng fringe



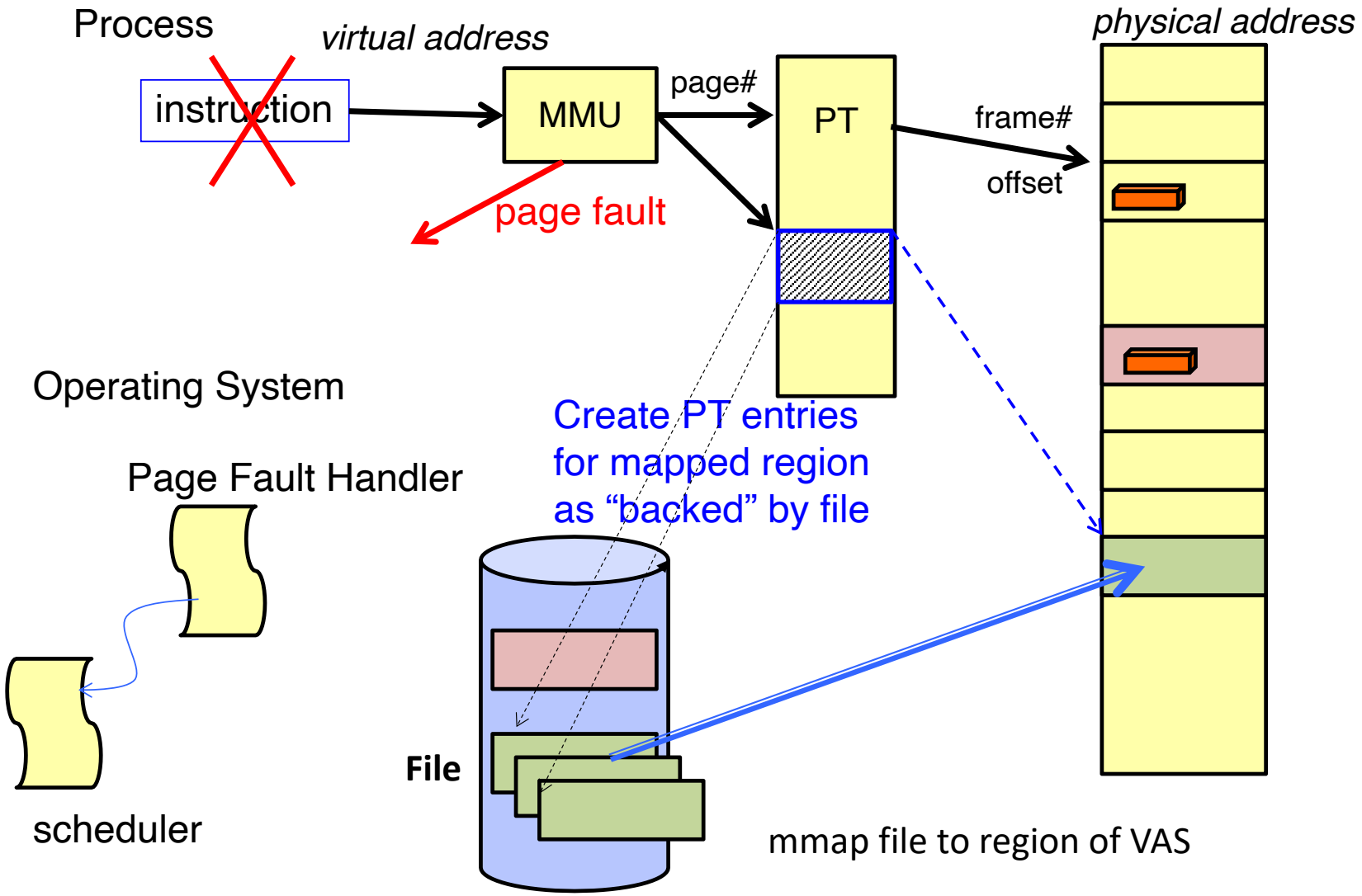
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to regions of a file
  - This involves multiple copies into caches in memory, plus system calls
- What if we could “map” the file directly into an empty region of our address space
  - Implicitly “page it in” when we read it
  - Write it and “eventually” page it out
- Executable file is treated this way when we exec the process !!

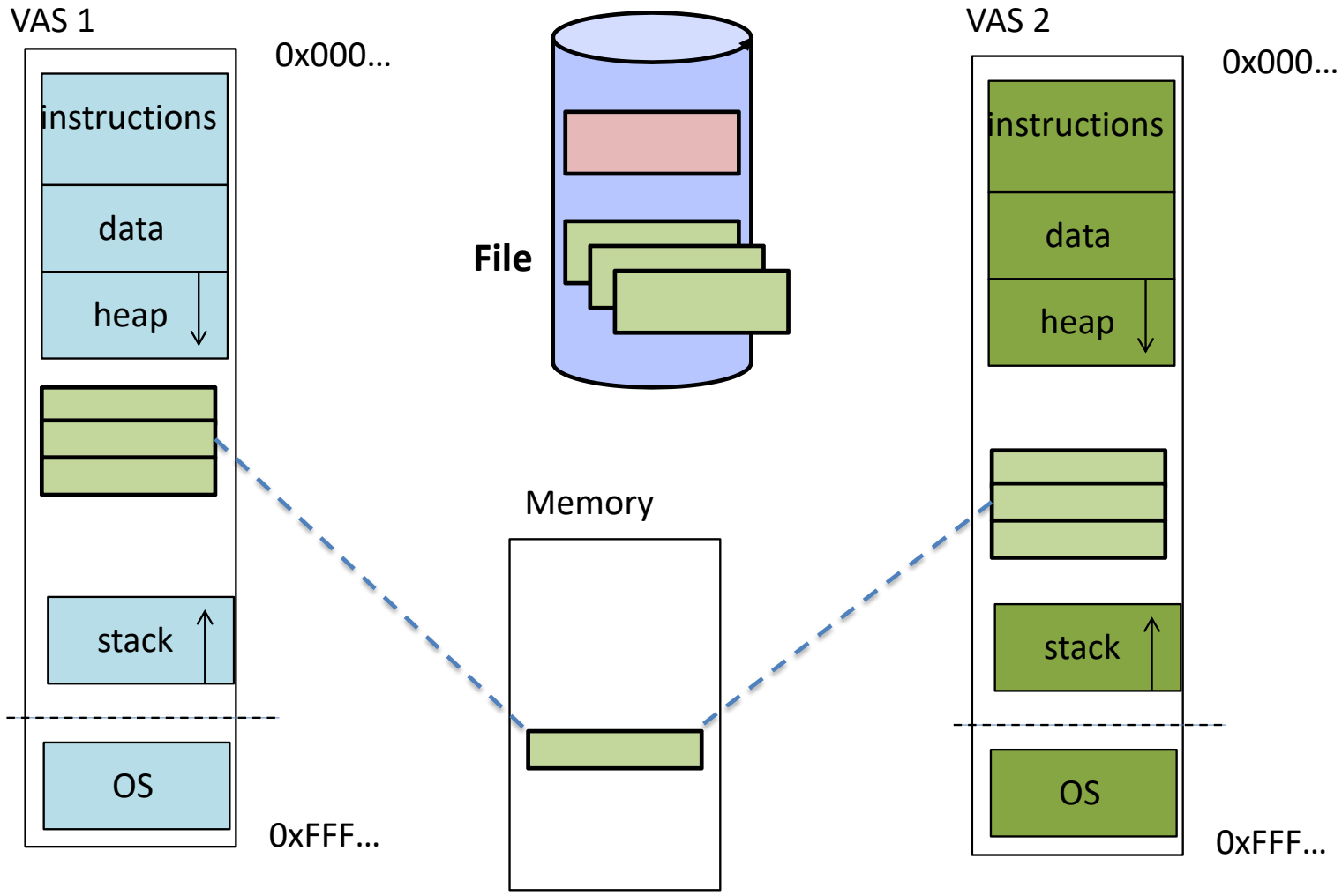
Recall: Who does what when ?



Using Paging to mmap files



Sharing through Mapped Files



Log-Structured File system

quick review: File Systems

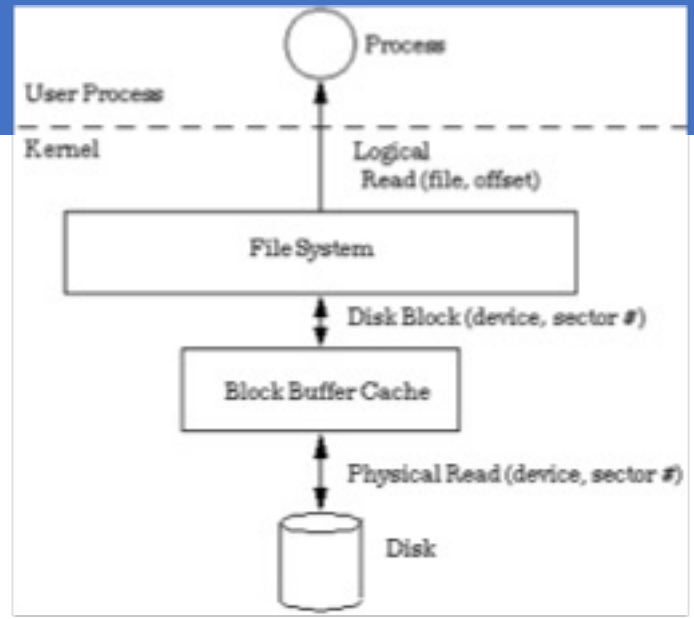
- A storage management / user interface to stored data
- Metadata of a file: i-node structure
  - File inside OS == i-node
- Considers efficient data access for small / large data
  - Direct blocks, indirect/doubly/triply indirect blocks
- Considers security for access control
  - Permission and credential information
- They have to be resident in disk, and memory
  - Durable after the power-off

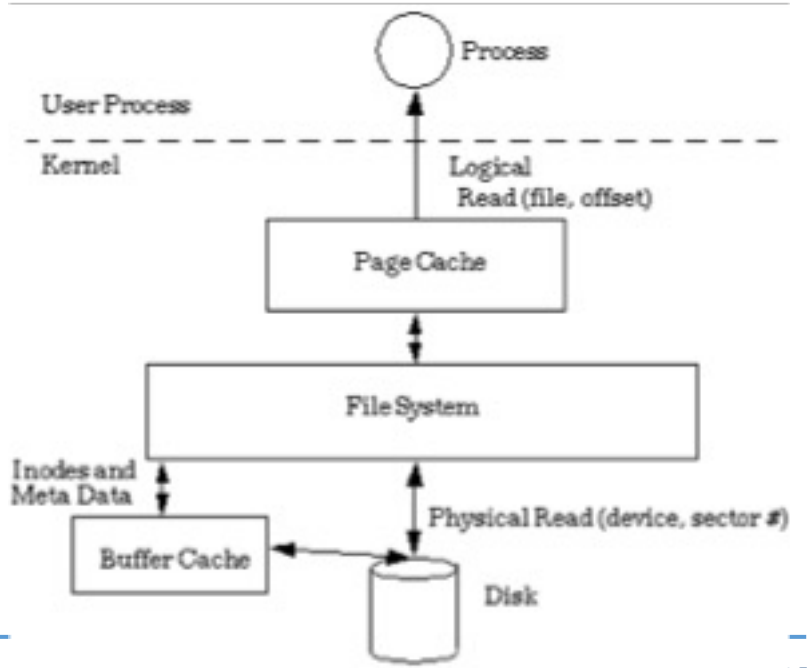
Cache & filesystem

- Huge gap in speed & scale between memory & disk
  - Latency / throughput
- Memory
  - DDR4 2133 case: 266Mhz, 2133MT/s ~ 17066MB/s
- Disk access: several milliseconds for seek time, rotational delay
  - Bandwidth: 120MB/s
- SSD: No seek time, no rotational delay, sub milliseconds
  - Controller delay & device's (NAND flash) physical operation time
  - Bandwidth: up to 500MB/s
- Leads to caching

Buffer cache & page cache

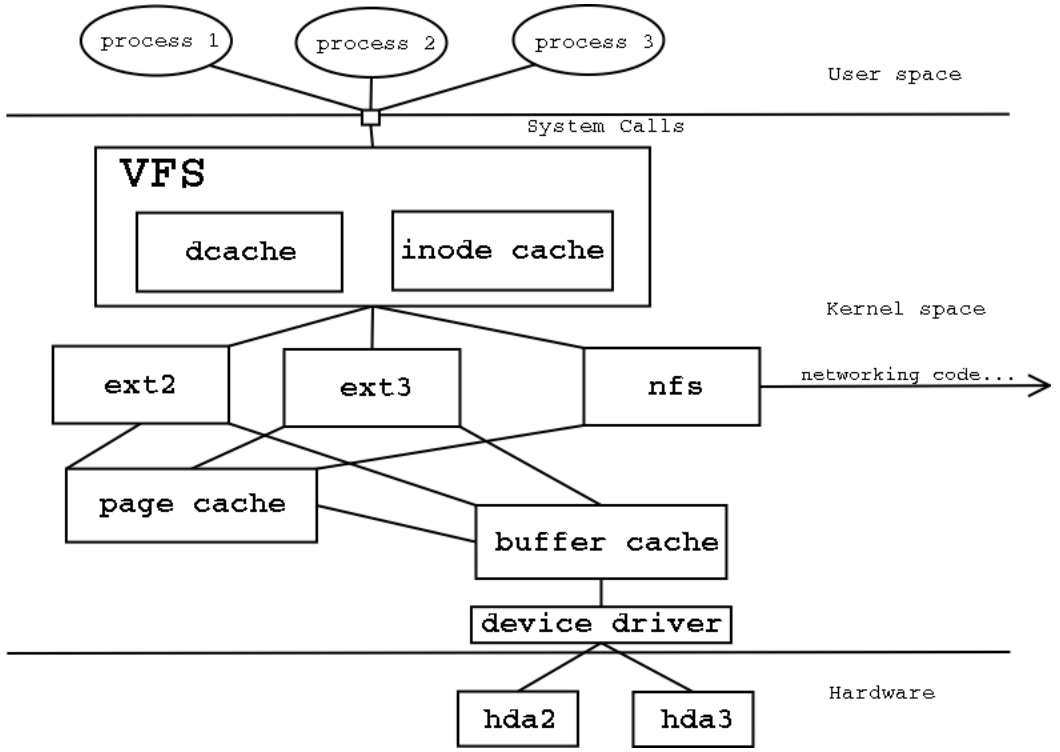
- If we have available space in memory, why not cache data in memory, rather than storing on disk?
  - Buffer cache!
  - Caches disk blocks, for faster access
  - Has to be synchronized with disk blocks
  - Device driver fetch data block from disk, place it on buffer cache
- What's page cache?
  - Cache user's file blocks
  - Enable faster user access





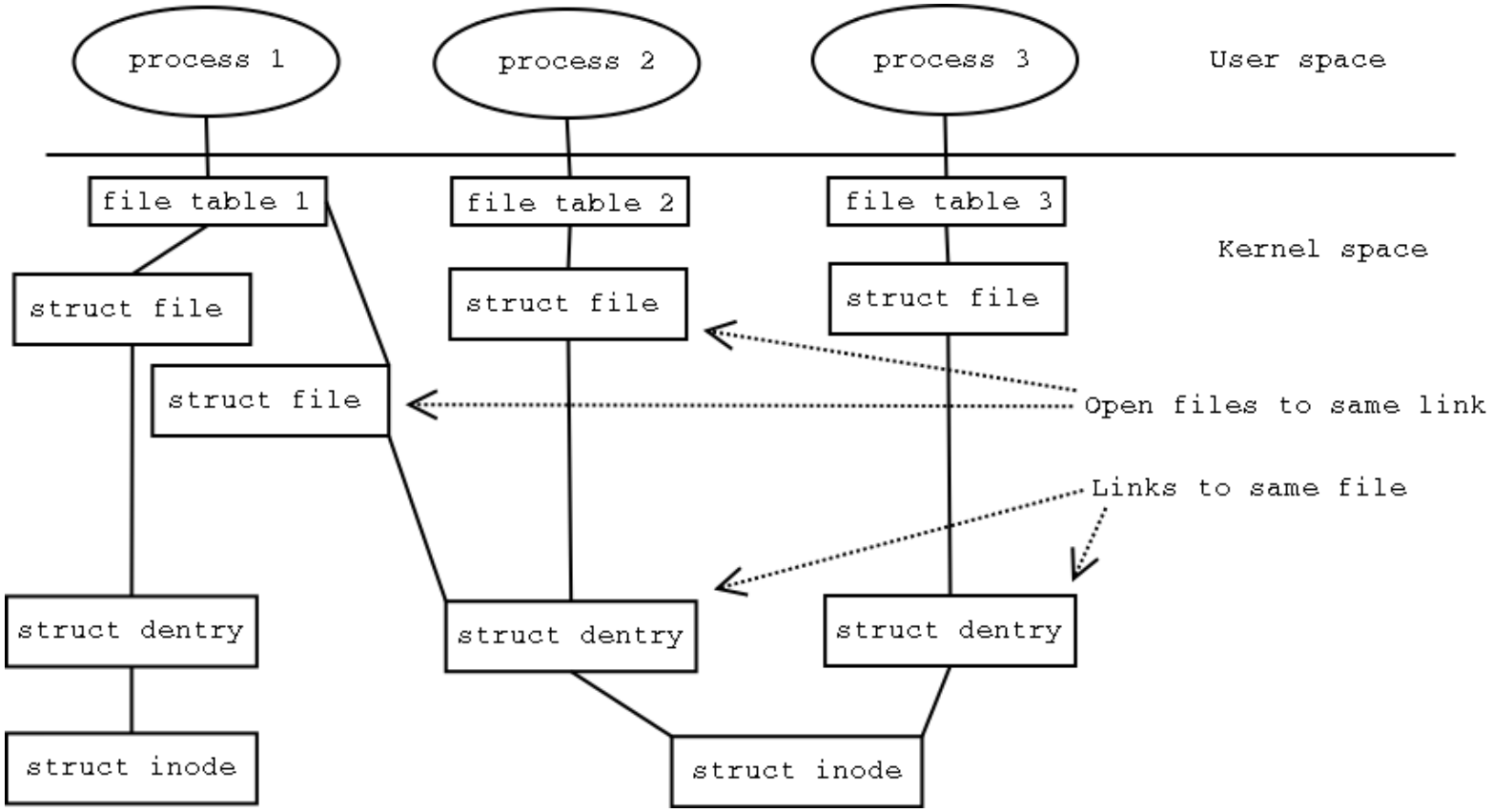
Virtual file system layer

- To support multiple filesystems, keeping unified user interface
  - Make a huge function pointers structure
  - Each file system implements the same interface;
  - Users are unable to distinguish file system's operation
- Linux VFS layer
  - Includes separate inode cache, direnty cache, page cache, and buffer cache



A travel to real FS implementation

- http://haifux.org/lectures/119/linux-2.4-vfs/linux-2.4-vfs.html



Issues in cache

- How to find data from cache?
  - It is a partial set of entire disk, so we cannot find data with index (block number)
- Dirty blocks management
  - Write-through cache vs. write-back cache
  - Is it okay to store data only on page cache/buffer cache?
  - No! → you may lose data if you turn off your computer
    - When the data goes back to disk? / When is good to move your data to disk?
    - Write permanent data as soon as possible vs. Delay the write as long as possible
- What if you have data in buffer cache, and turn off the computer?
  - Lose data!
  - How about metadata?

Durability issues – consistency in disk and memory

- All data in filesystem has to be written back to disk
  - Disk operation is way slower than memory operations…
- About files,
  - What if we write metadata in the disk, but lost data in the buffer cache?
  - What if we write data in the disk, but lost metadata in the buffer cache?
  - What about indirect/doubly indirect blocks update?
- About directory files,
  - What if we update directory entry in the disk, but lost update for the i-node?
  - What if we update i-node entry in the disk, but lost update for the directory entry?
- About free disk blocks,
  - What if we update free blocks map in the disk, but lost update for i-node?
  - What if we update i-node in the disk, but lost update for free disk blocks?

More Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, …
  - With remapping, single update to physical disk block can require multiple (even lower level) updates
- At a physical level, operations complete one at a time
  - Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

Reliability Approach #1: Careful Ordering

- Sequence operations in a specific order
  - Careful design to allow sequence to be interrupted safely
- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed

- Approach taken in FAT, FFS (fsck), and many app-level recovery schemes
  (e.g., Word)

FFS: Create a File

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks
- Update directory with file name → file number
- Update modify time for directory

Recovery:

- Scan inode table
- If any unlinked files (not in any directory), delete
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

Time proportional to size of disk

Application Level

Normal operation:

- Write name of each open file to app folder
- Write changes to backup file
- Rename backup file to be file (atomic operation provided by file system)
- Delete list in app folder on clean shutdown

Recovery:

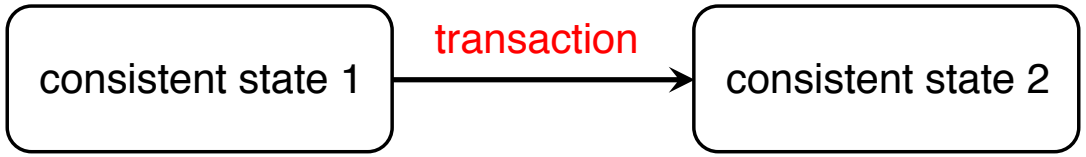
- On startup, see if any files were left open
- If so, look for backup file
- If so, ask user to compare versions

Reliability Approach #2: Copy on Write-based File System Layout

- To update file system, write a new version of the file system containing the update
  - Never update in place
  - Reuse existing unchanged disk blocks
- Seems expensive! But
  - Updates can be batched
  - Almost all disk writes can occur in parallel
- Approach taken in network file server appliances (WAFL, ZFS)

Transactional processing!

- Transaction
  - An atomic sequence of actions (reads/writes) on a storage system (or database)
  - That takes it from one consistent state to another



  - Occur all at once (conduct operation completely)
    or lose all (abort all and go back to old state)
  - That's what happened in git,
    Make some changes and commit!
    - Commit is the unit of change
    - Before you commit, the changes are in transient state
  - If some bad thing happens, roll back to the initial state

The ACID properties of Transactions

- ***Atomicity***: all actions in the transaction happen, or none happen
- ***Consistency***: transactions maintain data integrity, e.g.,
  - Balance cannot be negative
  - Cannot reschedule meeting on February 30
- ***Isolation***: execution of one transaction is isolated from that of all others; no problems from concurrency
- ***Durability***: if a transaction commits, its effects persist despite crashes

Transactional File Systems

- Journaling File System
  - Applies updates to system metadata using transactions (using logs, etc.)
  - Updates to non-directory files (i.e., user stuff) is done in place (without log s)
  - Ex: NTFS, Apple HFS+, Linux XFS, JFS, ext3, ext4
- Logging File System (Log-structured file systems)
  - *All* updates to disk are done in transactions
  - M. Rosenblum, J. Ousterhout, The Design and Implementation of a Log-structured File Sys tem, ACM Trans. of Computer Systems 1992

Logging File Systems

- Instead of modifying data structures on disk directly, write changes to a journal/log
  - Intention list: set of changes we intend to make
  - Log/Journal is **append-only**
- Once changes are in the log, it is safe to apply changes to data structures on disk
  - Recovery can read log to see what changes were intended
  - Can take our time making the changes
    - As long as new requests consult the log first
- Once changes are copied, safe to remove log
- But, …
  - If the last atomic action is not done … poof … all gone

THE atomic action

- Write a sector on disk

Redo Logging

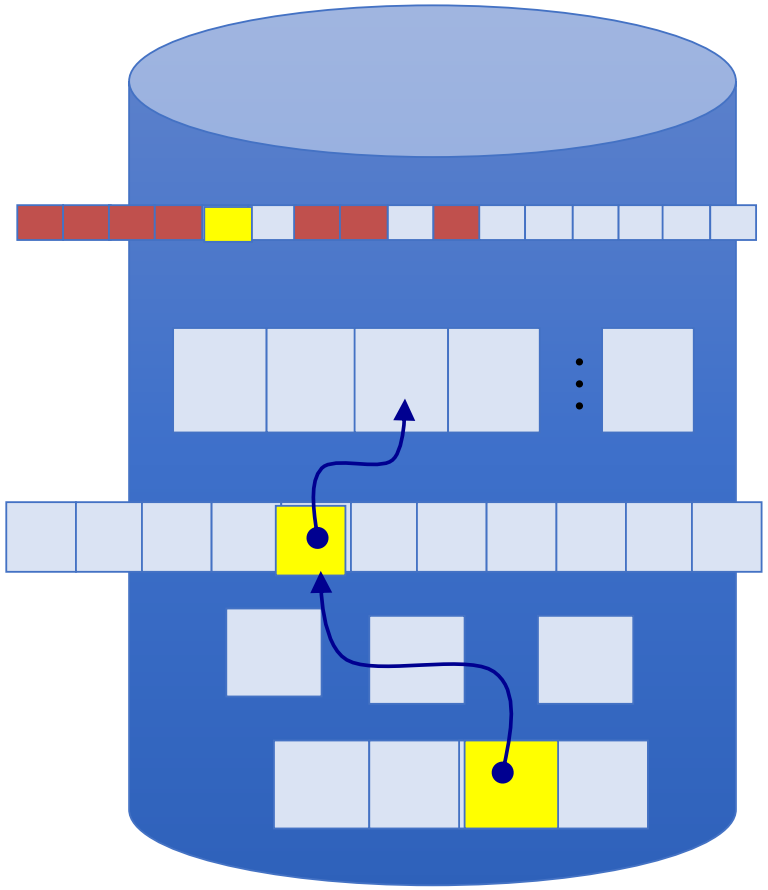
- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transact ion durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log

- Recovery
  - Read log
  - Redo any operations for committe d transactions
  - Garbage collect log

Example: Creating a file

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

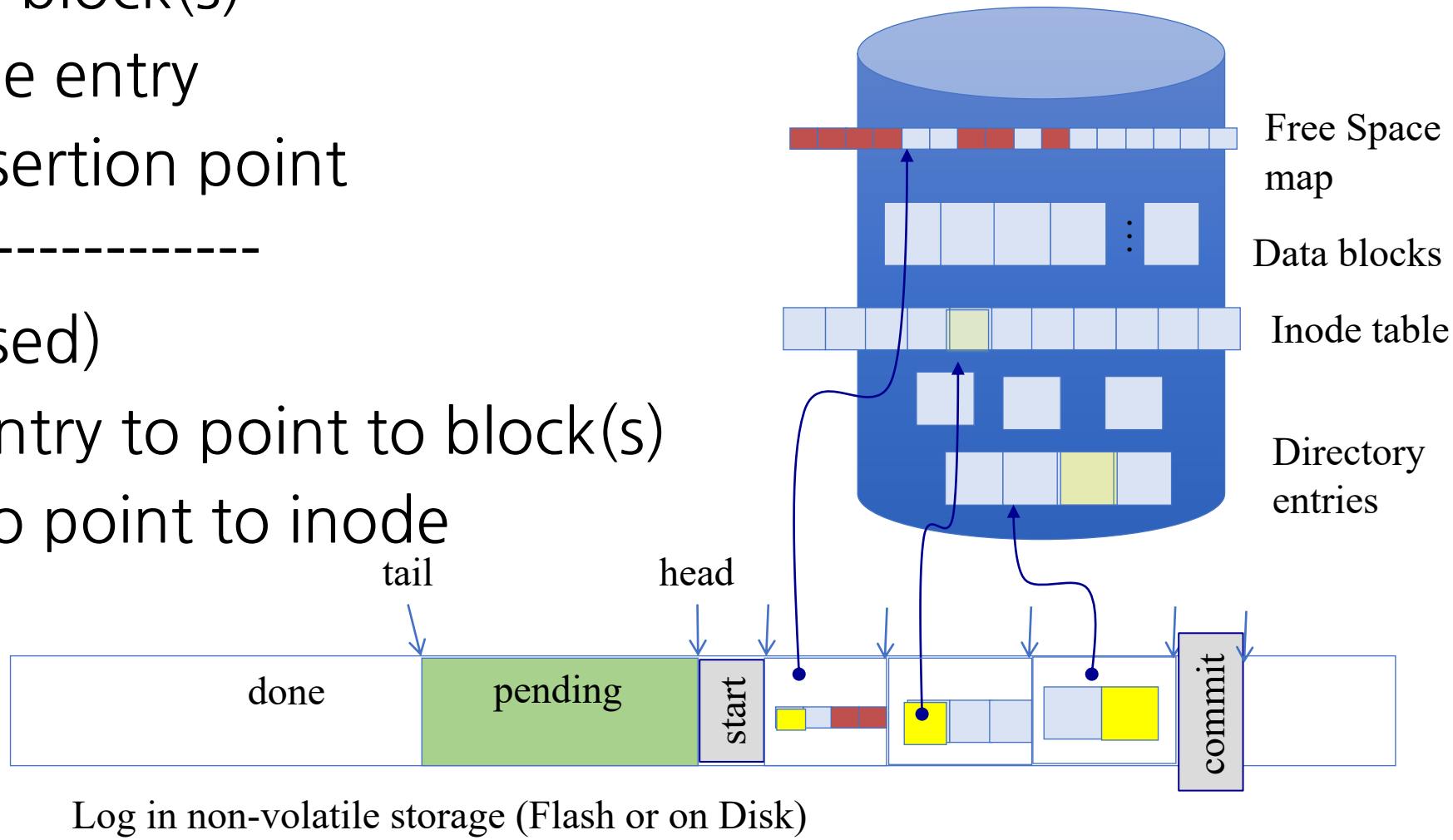
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Ex: Creating a file (as a transaction)

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

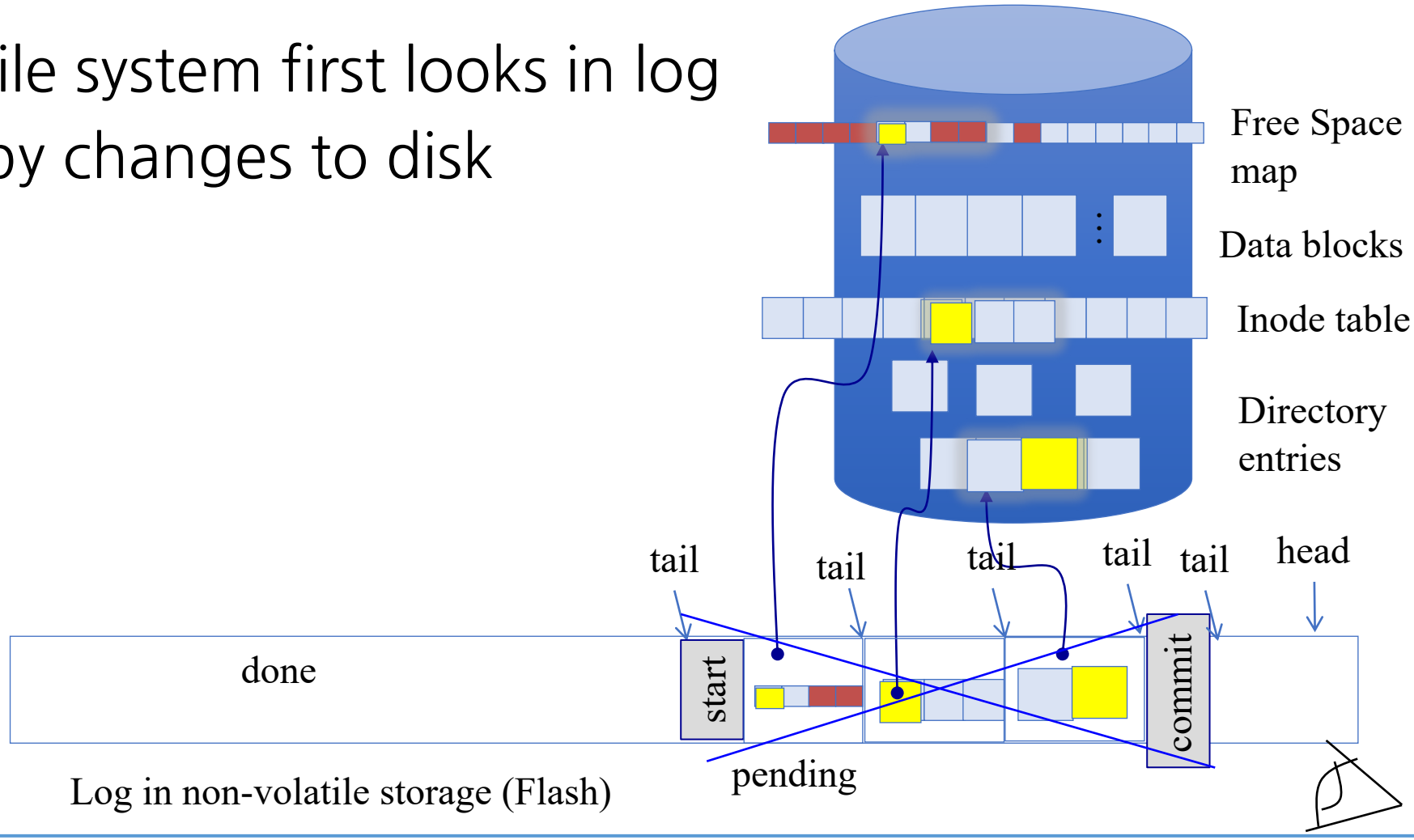
- Write map (used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Log in non-volatile storage (Flash or on Disk)

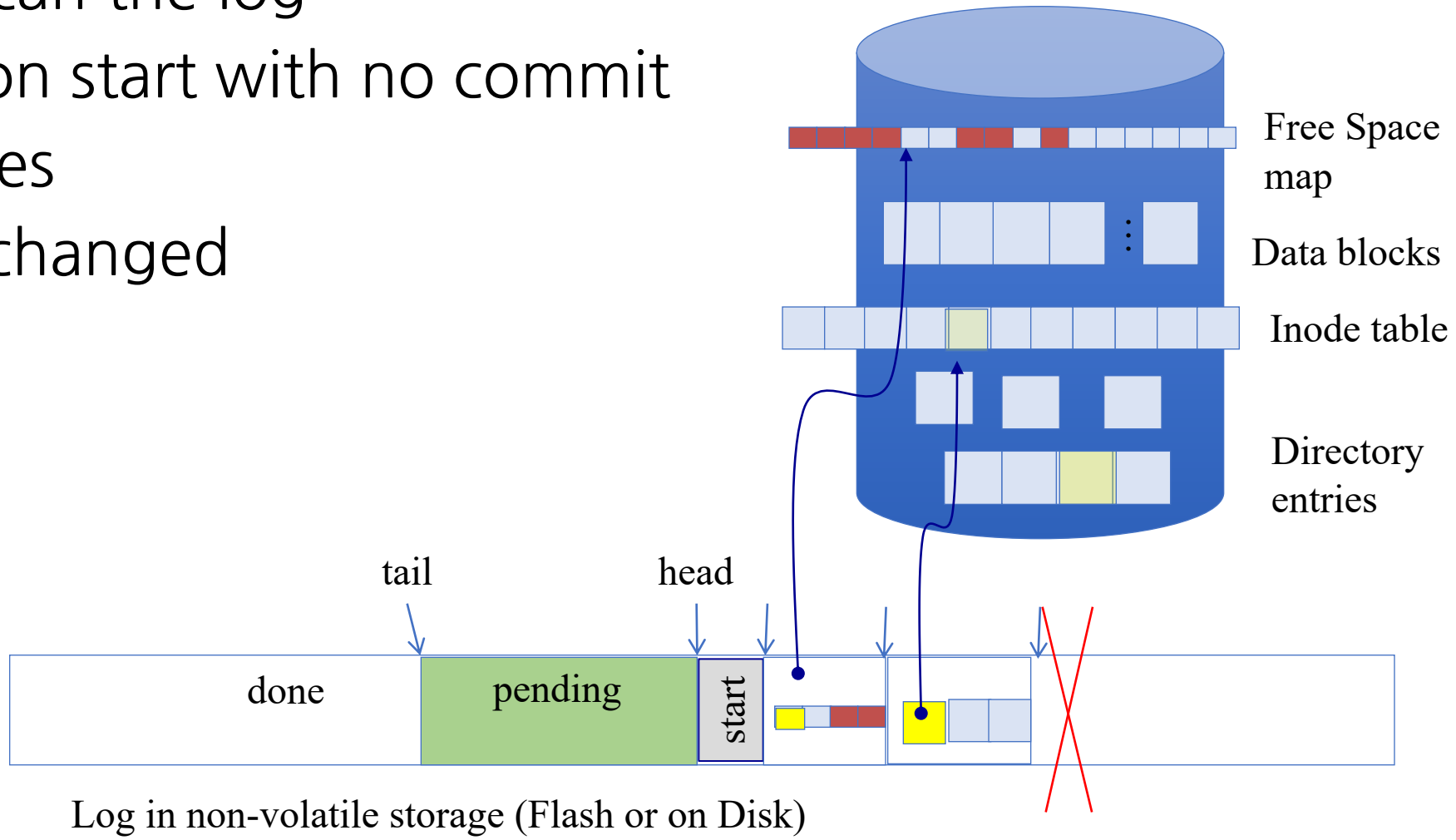
ReDo log

- After Commit
- All access to file system first looks in log
- Eventually copy changes to disk



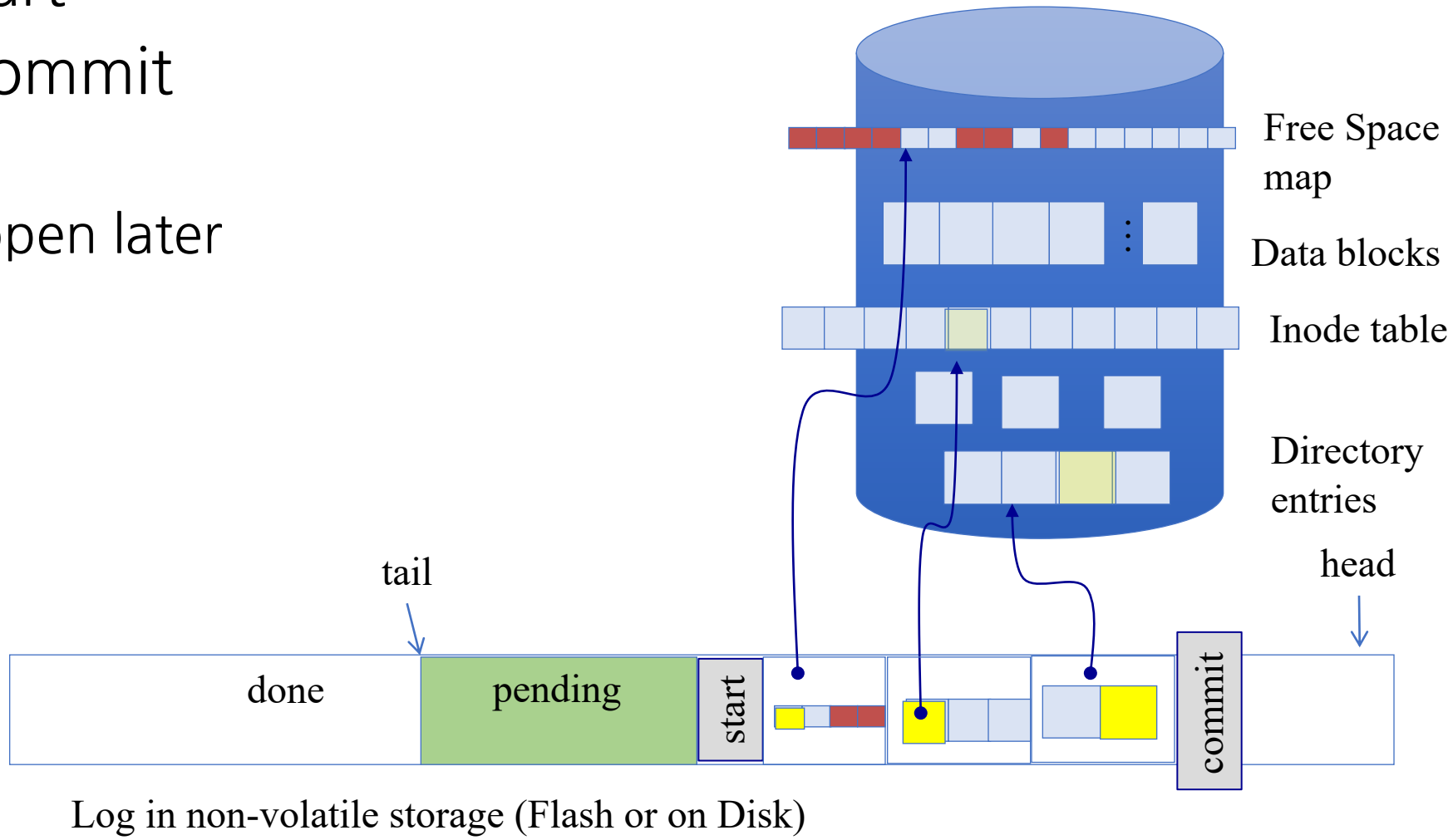
Crash during logging - Recover

- Upon recovery scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



Recovery After Commit

- Scan log, find start
- Find matching commit
- Redo it as usual
  - Or just let it happen later



Questions on Log-structured file system transactions

- What if had already started writing back the transaction ?
  - *Idempotent* - the result does not change if the operation is repeat several ti mes.
  - Just write them again during recovery
- What if the uncommitted transaction was discarded on recovery?
  - Do it again from scratch
  - Nothing on disk was changed
- What if we crash again during recovery?
  - Idempotent
  - Just redo whatever part of the log hasn't been garbage collected

Redo Logging

- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transact ion
    durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log

- Recovery
  - Read log
  - Redo any operations for committe d transactions
  - Ignore uncommitted ones
  - Garbage collect log

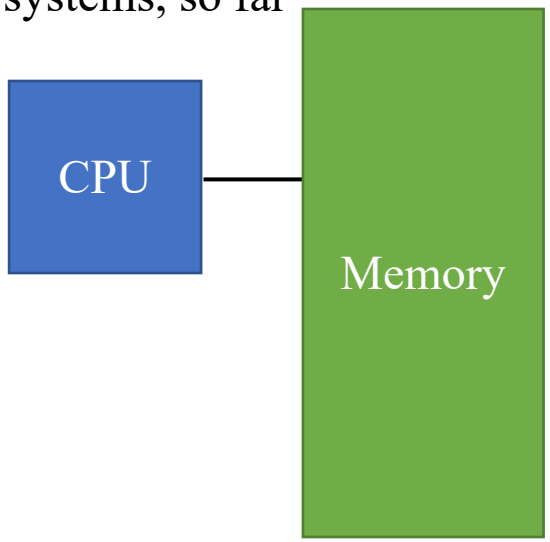
Log-structured file system

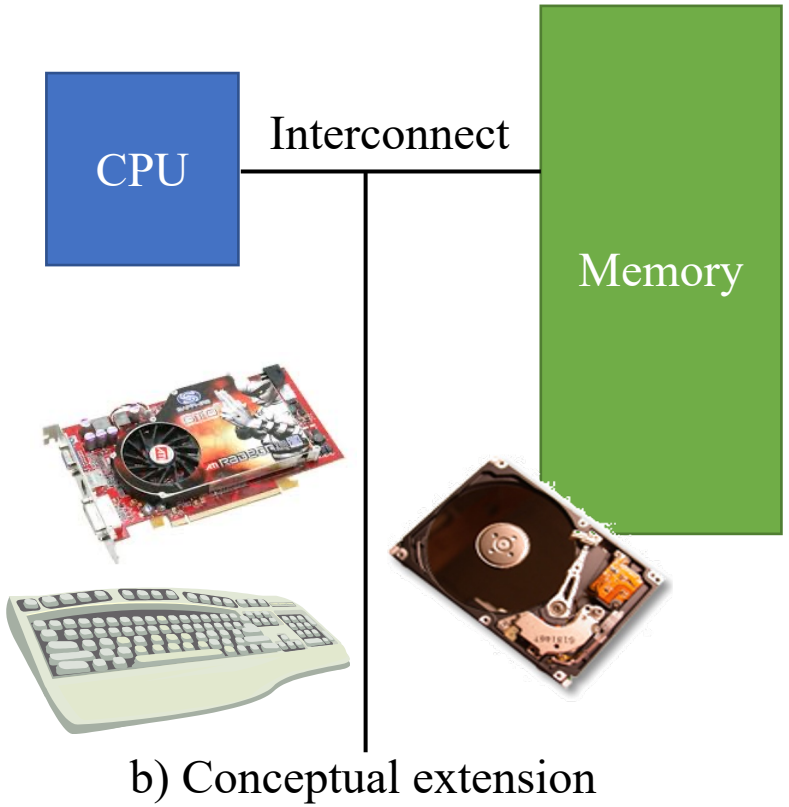
- Filesystem operation with transaction
- Transaction usages
  - Reliable application systems (e.g. database)
  - Logging/commit/rollback
- Make a system more reliable than its subsystems
  - Atomicity to support critical region's update
    - 2 Phase-Locking
  - Consistency when multiple users access
    - Make replicas / log / rollback when it fails
  - Isolate failure from the rest of the system
  - Durablity to preserve data after commit
  - Into the distributed systems

I/O devices and Networking

A real world computer system with I/O devices

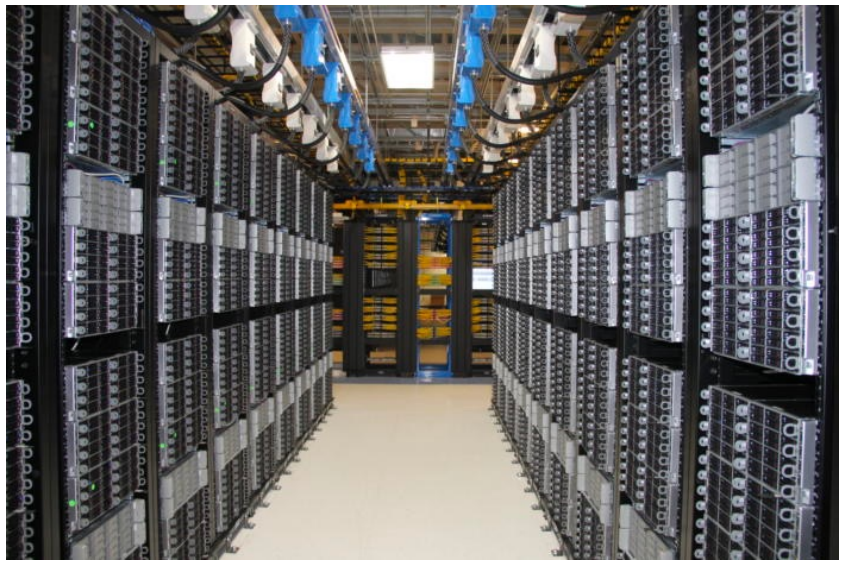
a) Your view of computer systems, so far

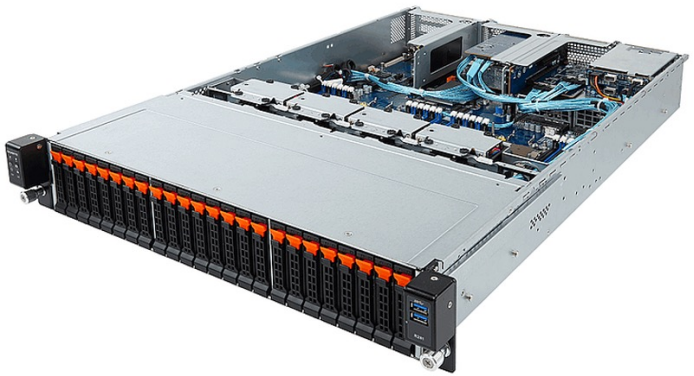




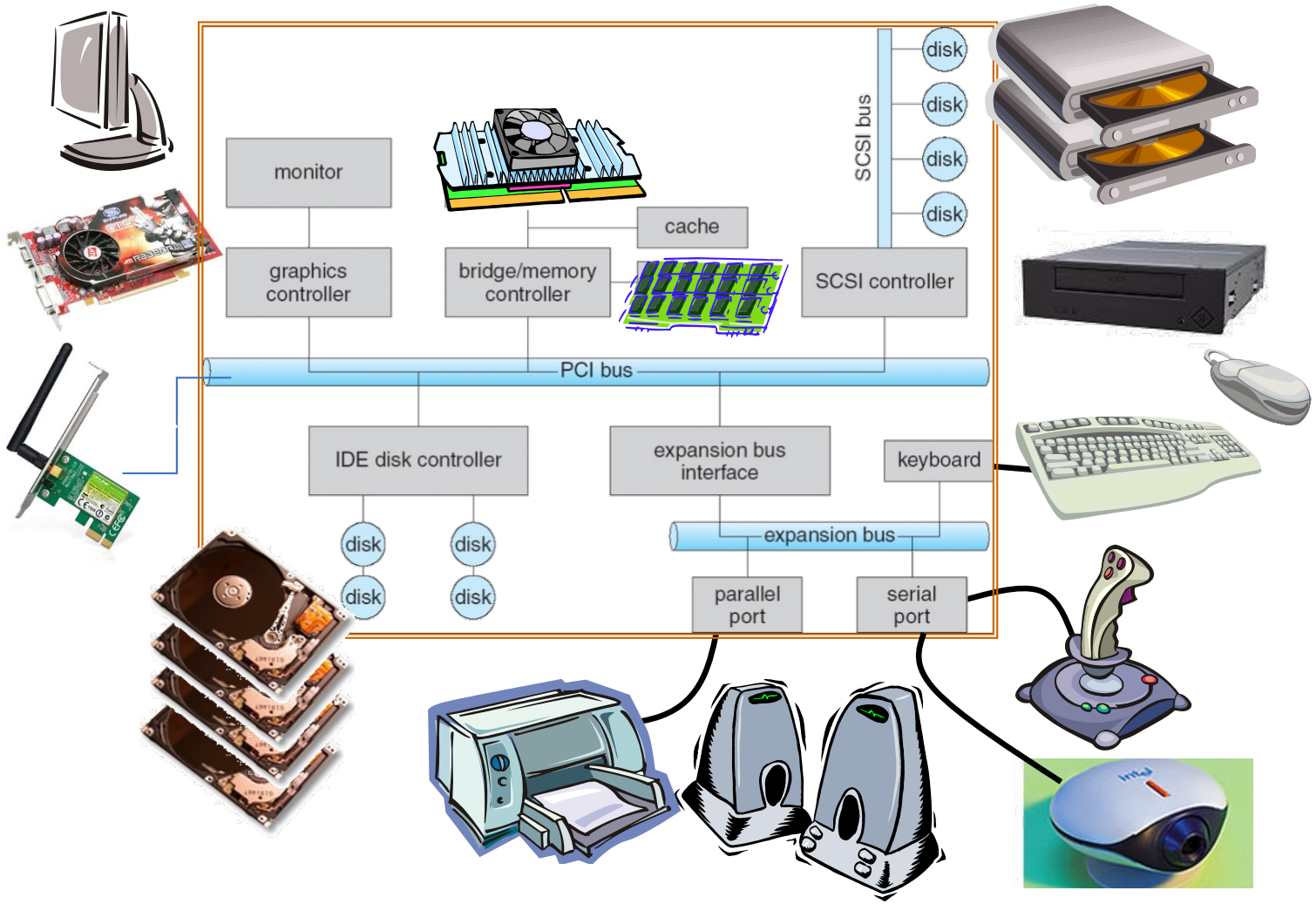
b) Conceptual extension

c) Real-life version



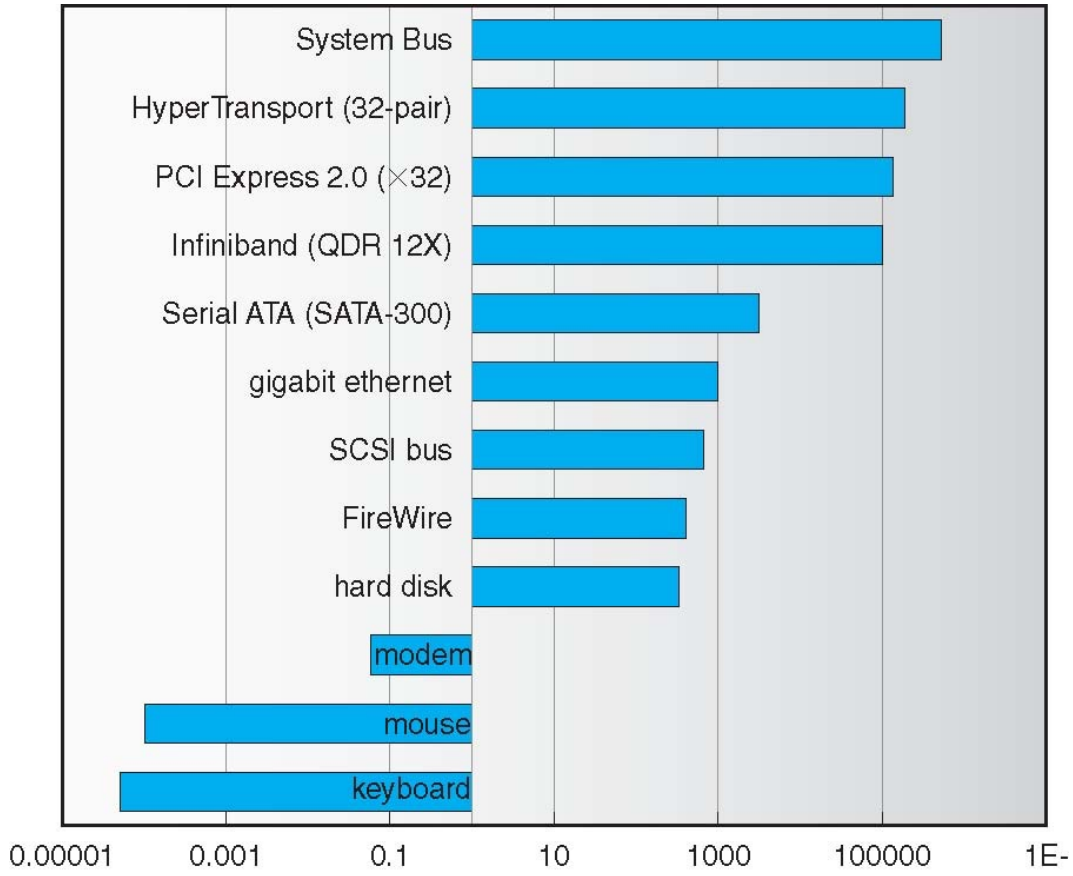


~~Modern~~ Computer Systems & I/O Devices



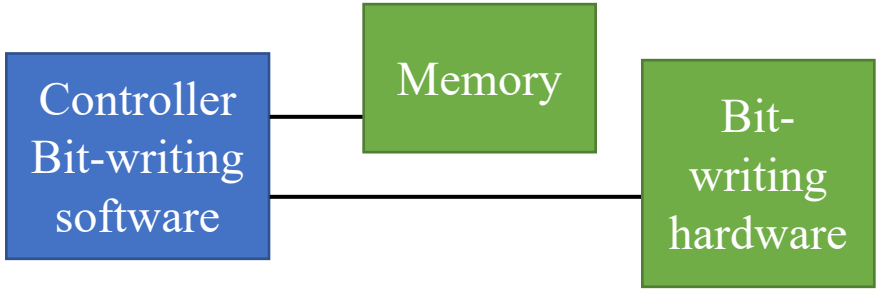
Example Device-Transfer Rates in Mb/s (Sun Enterprise 6000)

- Device Rates vary over 12 orders of m agnitude !!
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fa st devices!
  - Better not waste time waiting for slow de vices
- Rethink the fundamentals
  - When there is a change / difference in th at scale



I/O devices are small computers!

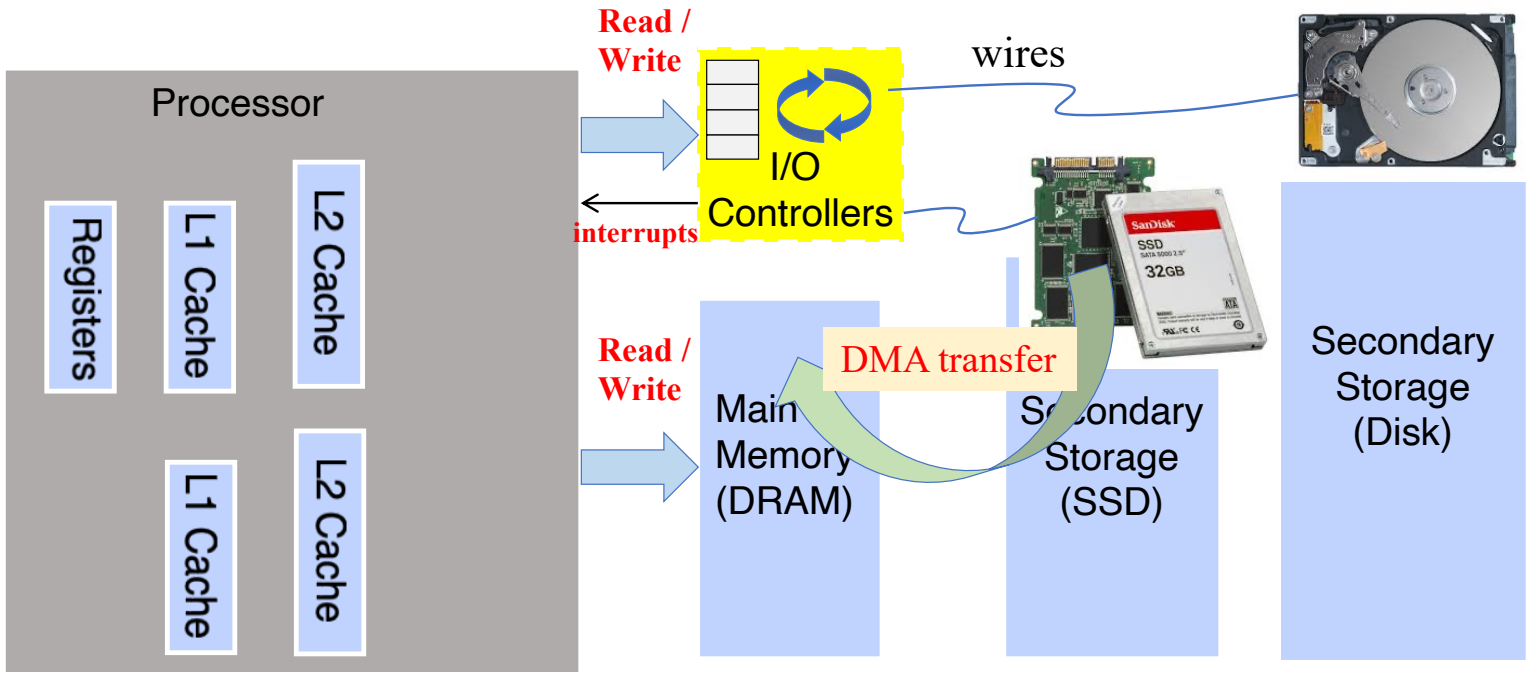
- Some complicated I/O devices
  - NIC (network interface card)
  - SSD (solid state drive)
  - HDD (hard disk drive)



- They have special CPU that operates with dedicated hardware
  - Controller: some form of processing core
    - Access hardware, instead of calculating numeric values
      - write value on wire, write value on disk block, read value from disk block
  - Buffer: some form of memory
    - Has some data from I/O device and users

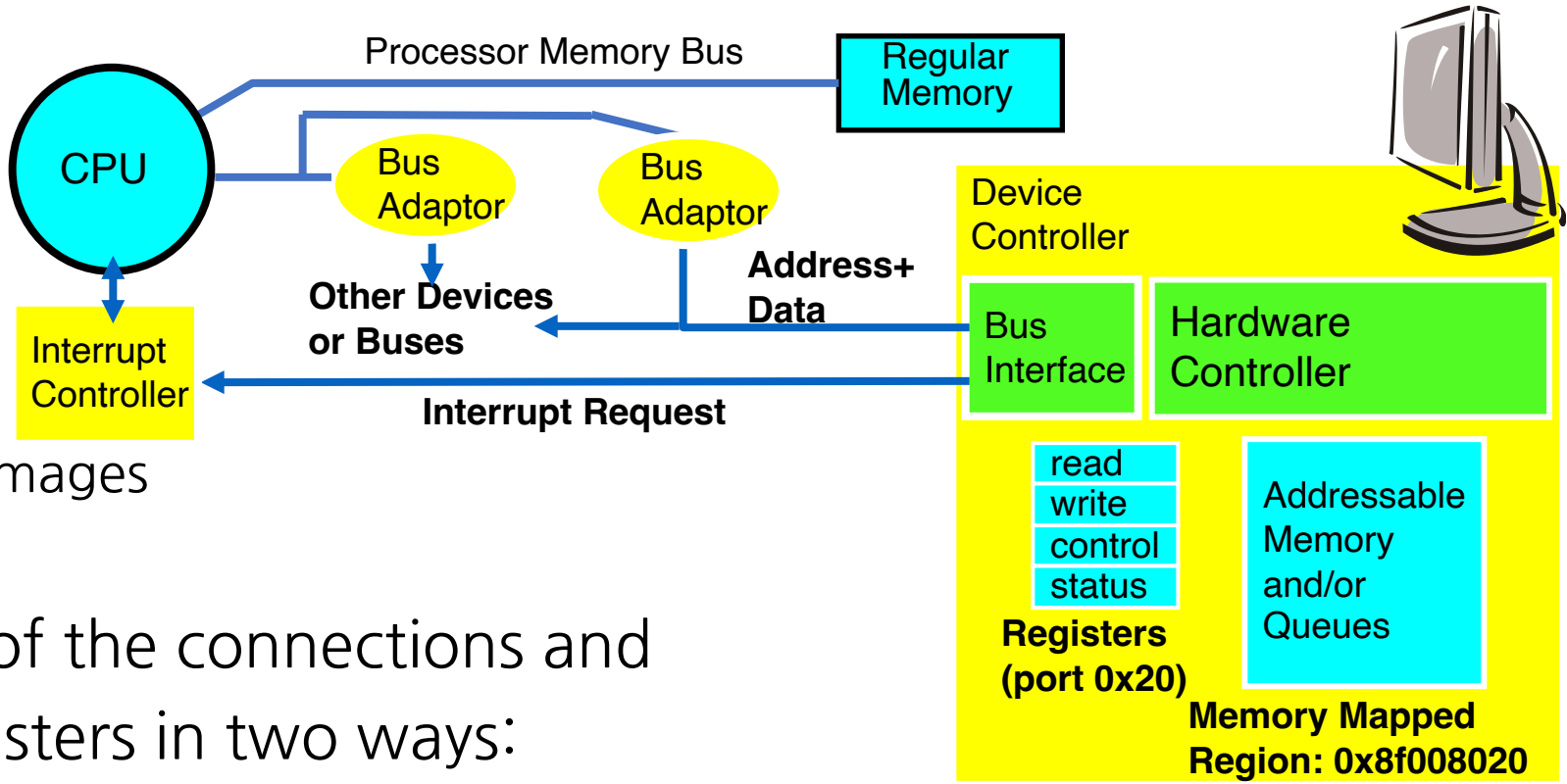
I/O devices operate asynchronous to the CPU execution

- I/O devices you recognize are supported by I/O Controllers
- Processors accesses them by reading and writing IO registers as if t hey were memory
  - Write commands and arguments, read status and results



How Does the Processor Talk to Devices?

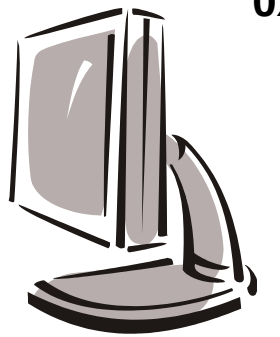
- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images

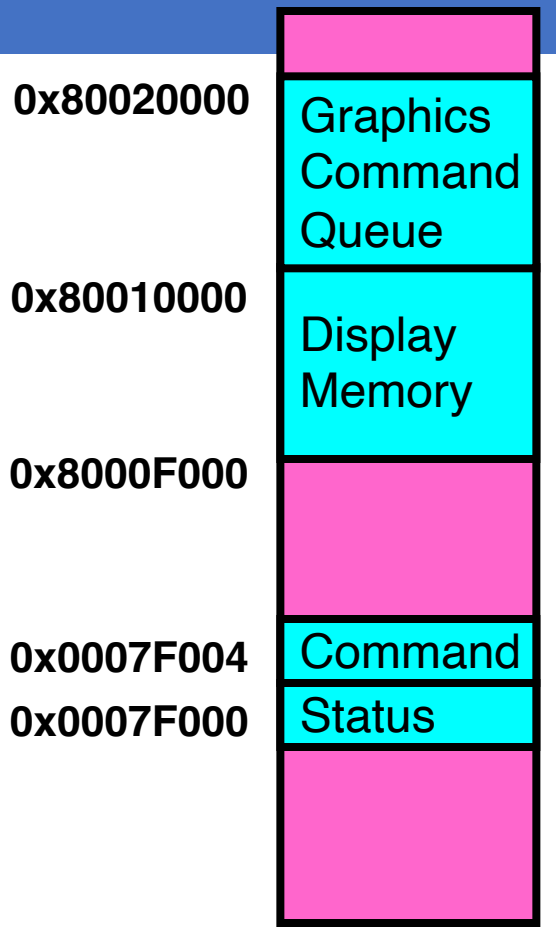


- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions (e.g., Intel's 0x21,AL)
  - Memory mapped I/O: load/store instructions
    - Registers/memory appear in physical address space
    - I/O accomplished with load and store instructions

Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - Say enter a set of triangles that describe some scene
    - Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - Say render the above scene
    - Addr: 0x0007F004
- Can protect with address translation

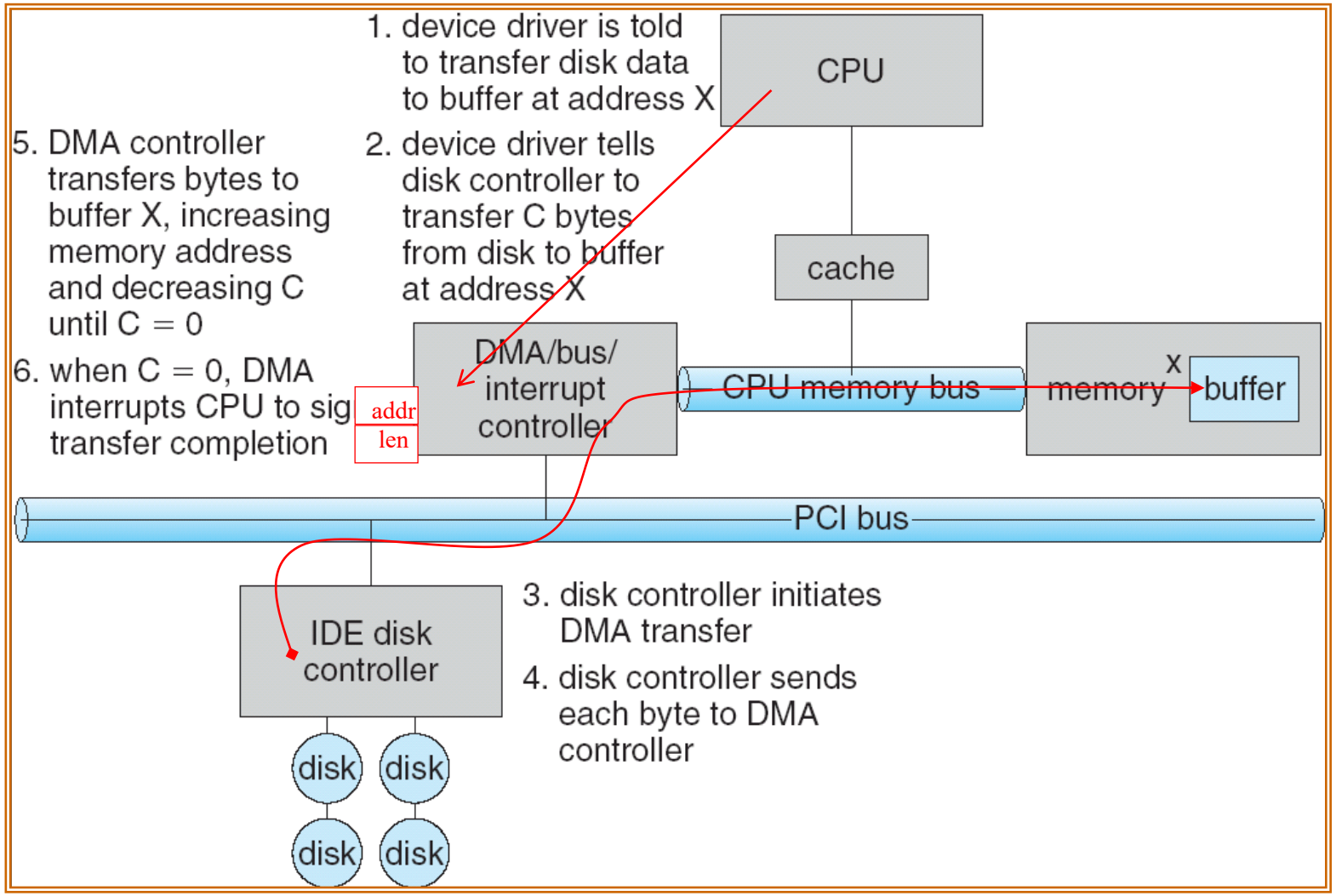




Physical Address Space

Transferring Data To/From Controller

- Programmed I/O:
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data blocks to/from memory directly
- Sample interaction with DMA controller



I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt:
  - Device generates an interrupt whenever it needs service
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- Polling:
  - OS periodically checks a device-specific status register
    - I/O device puts completion information in status register
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance - High-bandwidth network adapter:
    - Interrupt for first incoming packet
    - Poll for following packets until hardware queues are empty

What is the Role of I/O?

- Without I/O, computers are useless (disembodied brains?)
  - But… thousands of devices, each slightly different
  - How can we standardize the interfaces to these devices?
- Devices are unreliable: media failures and transmission errors
  - How can we make them reliable???
- Devices are unpredictable and/or slow
  - How can we manage them if we don't know what they will do or how they will perform?

Operational Parameters for I/O

- *Data granularity:* Byte vs. Block
  - Some devices provide single byte at a time (*e.g.,* keyboard)
  - Others provide whole blocks (*e.g.,* disks, networks, etc.)
- *Access pattern*: Sequential vs. Random
  - Some devices must be accessed sequentially (*e.g.,* tape)
  - Others can be accessed "randomly" (*e.g.,* disk, cd, etc.)
  - Fixed overhead to start sequential transfer (more later)
- *Transfer Notification:* Polling vs. Interrupts
  - Some devices require continual monitoring
  - Others generate interrupts when they need service
- *Transfer Mechanism:* Programmed IO and DMA

The Goal of the I/O Subsystem

- Provide uniform interfaces, despite wide range of different devices
  - This code works on many different devices:

```
FILE *fd = fopen("/dev/something","rw");
for (int i = 0; i < 10; i++) {
    fprintf(fd, "Count %d₩n",i);
}
close(fd);
```

  - Why?
  - Because code that controls devices ("device driver") implements standard interface
  - Standard user library raise the standard driver / subsystem interface
- We will try to get a flavor for what is involved in actually controlling devices
  - Can only scratch surface!

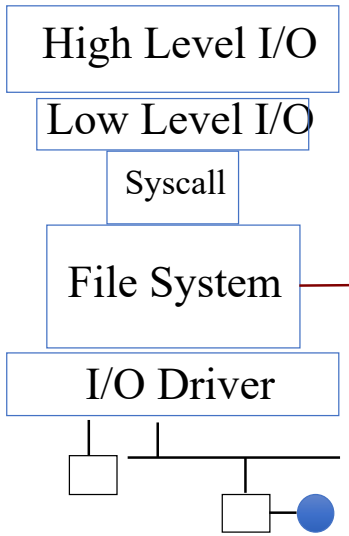
Want Standard Interfaces to Devices

- Block Devices: *e.g.,* disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Driver Commands include open(), read(), write(), seek()
  - Raw I/O or file-system access
  - Memory-mapped file access possible (discussed later … VAS!)
- Character/Byte Devices: *e.g.,* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include get(), put()
  - Libraries layered on top allow line editing
- Network Devices: *e.g.,* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - Separates network protocol from network operation
    - Includes select() functionality

I/O & Storage Layers

Operations, Entities and Interface

Application / Service



streams

handles

registers

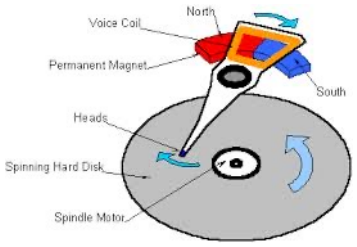
file_open, file_read, … on **struct file** * & void *

descriptors

open, read, … on **???** (blocks, chars, nets)

Commands and Data Transfers

Disks, Flash, Controllers, DMA

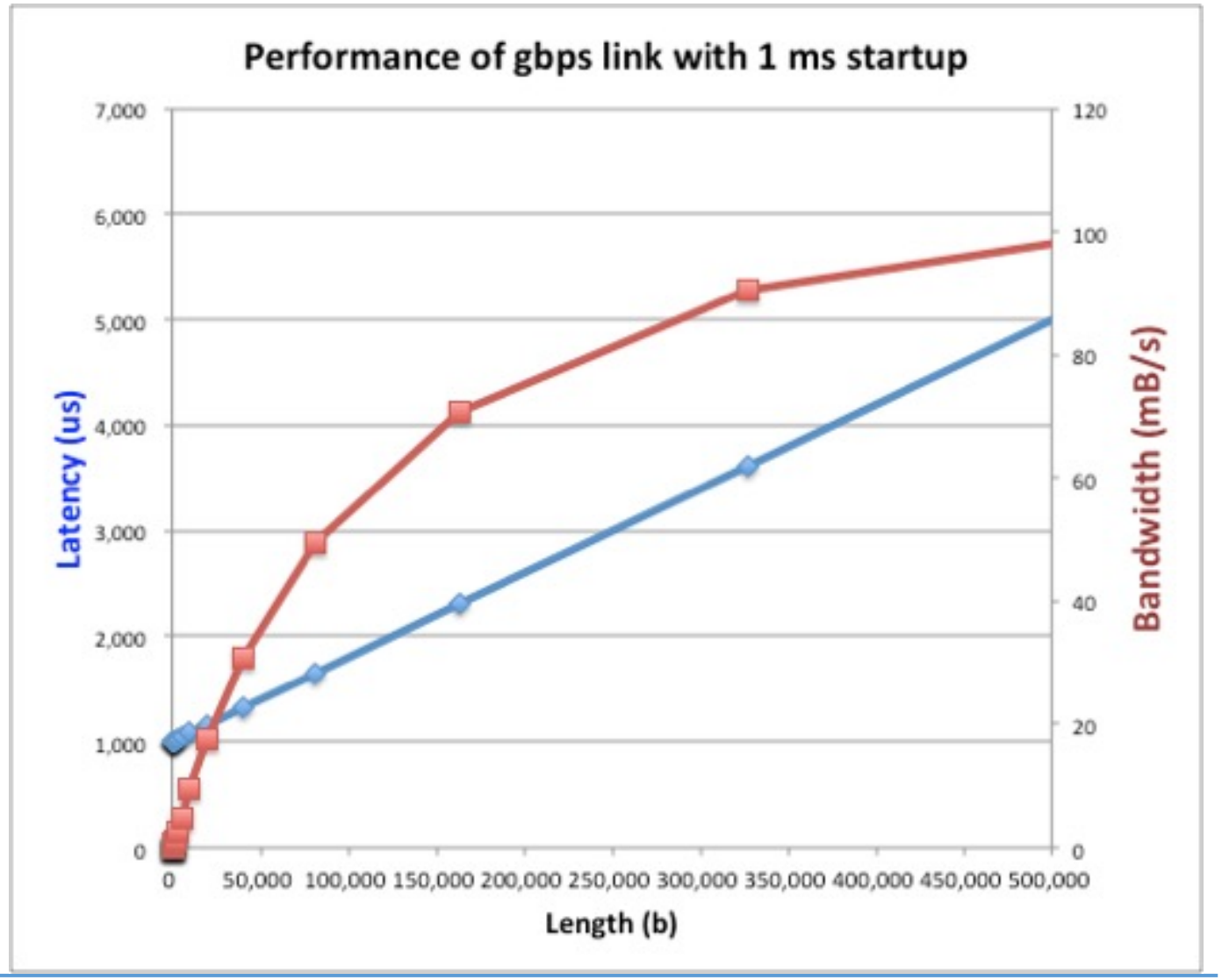


Basic Performance Concepts

- *Response Time* or *Latency*: Time to perform an operation (s)
- *Bandwidth* or *Throughput*: Rate at which operations are performed (op/s)
  - Files: mB/s, Networks: mb/s, Arithmetic: GFLOP/s
- *Start up* or "Overhead": time to initiate an operation
- Most I/O operations are *roughly* linear
  - Latency (n) = Ovhd + n/Bandwdth*
    - Bandwdth* is the bottleneck bandwidth

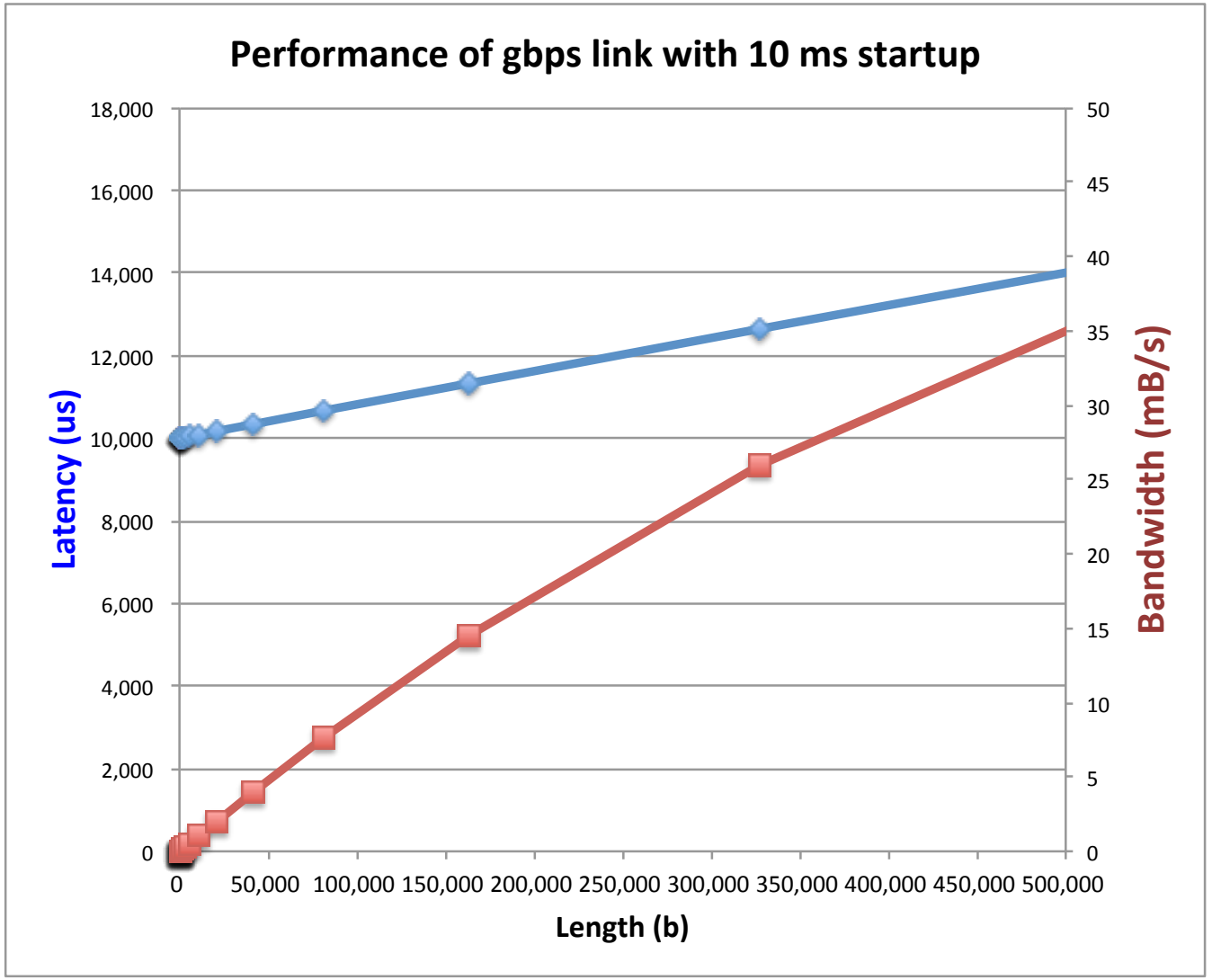
Example (fast network)

- Consider a 1Gbps link (125 mB/s)
- With a startup cost S = 1 ms



Example: at 10 ms startup (disk)

- Seagate desktop HDD (3TB)
- https://www.seagate.com/www-content/product-content/barracuda-fam/desktop-hdd/barracuda-7200-14/en-us/docs/100686584v.pdf
- Max I/O data xfer rate 600 MB/s
- Average data rate R/W 156MB/s
- Average seek time R: 8.5 ms, W: 9.5 ms
- Cache buffer 64MB



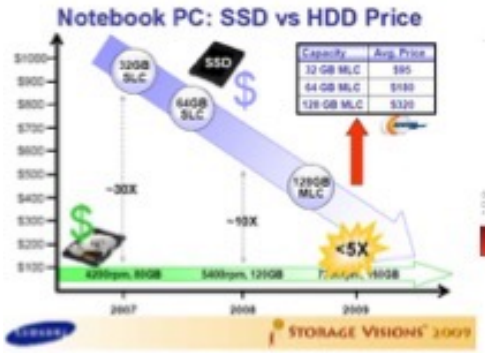
What determines *peak* BW for I/O ?

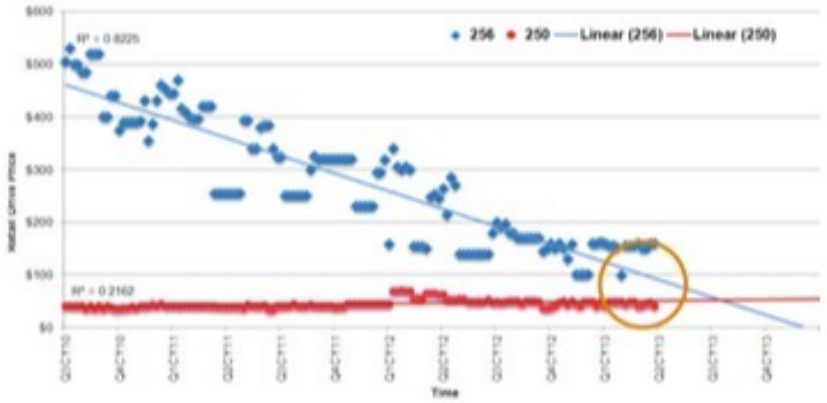
- Bus Speed
  - PCI-X: 1064 MB/s = 133 MHz x 64 bit (per lane)
    - 16GT/s per lane for PCIe v4, Up to 16 lanes or 256GT/s (~31.51GB/s)
  - IDE: Up to 133MB/s
  - ULTRA WIDE SCSI: 40 MB/s
  - Serial Attached SCSI & Serial ATA & IEEE 1394 (firewire) : 1.6 Gbps full duplex (200 MB/s)
    - 6Gbit/s for SATA3
  - USB 1.5 - 12 mb/s
    - 10Gbps for USB3.1 Gen2
- Device Transfer Bandwidth
  - Rotational speed of disk
  - Write / Read rate of nand flash
  - Signaling rate of network link
- Whatever is the bottleneck in the path

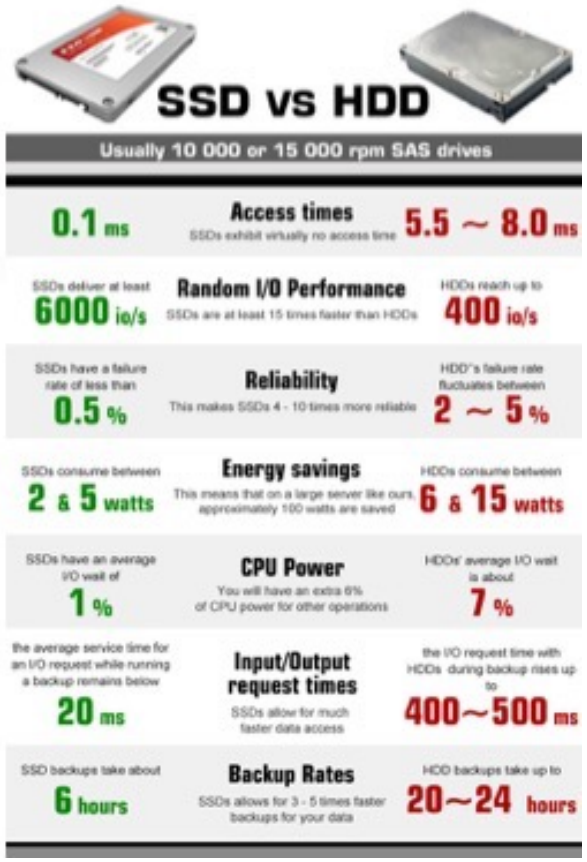
Storage Devices

- Magnetic disks
  - Storage that rarely becomes corrupted
  - Large capacity at low cost
  - Block level random access
  - Slow performance for random access
  - Better performance for streaming access
- Flash memory
  - Storage that rarely becomes corrupted
  - Capacity at intermediate cost (50x disk ???)
  - Block level random access
  - Good performance for reads; worse for random writes
  - Erasure requirement in large blocks
  - Wear patterns

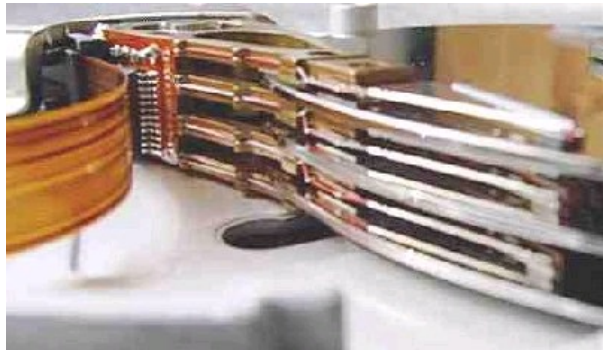
Are we in an inflection point?







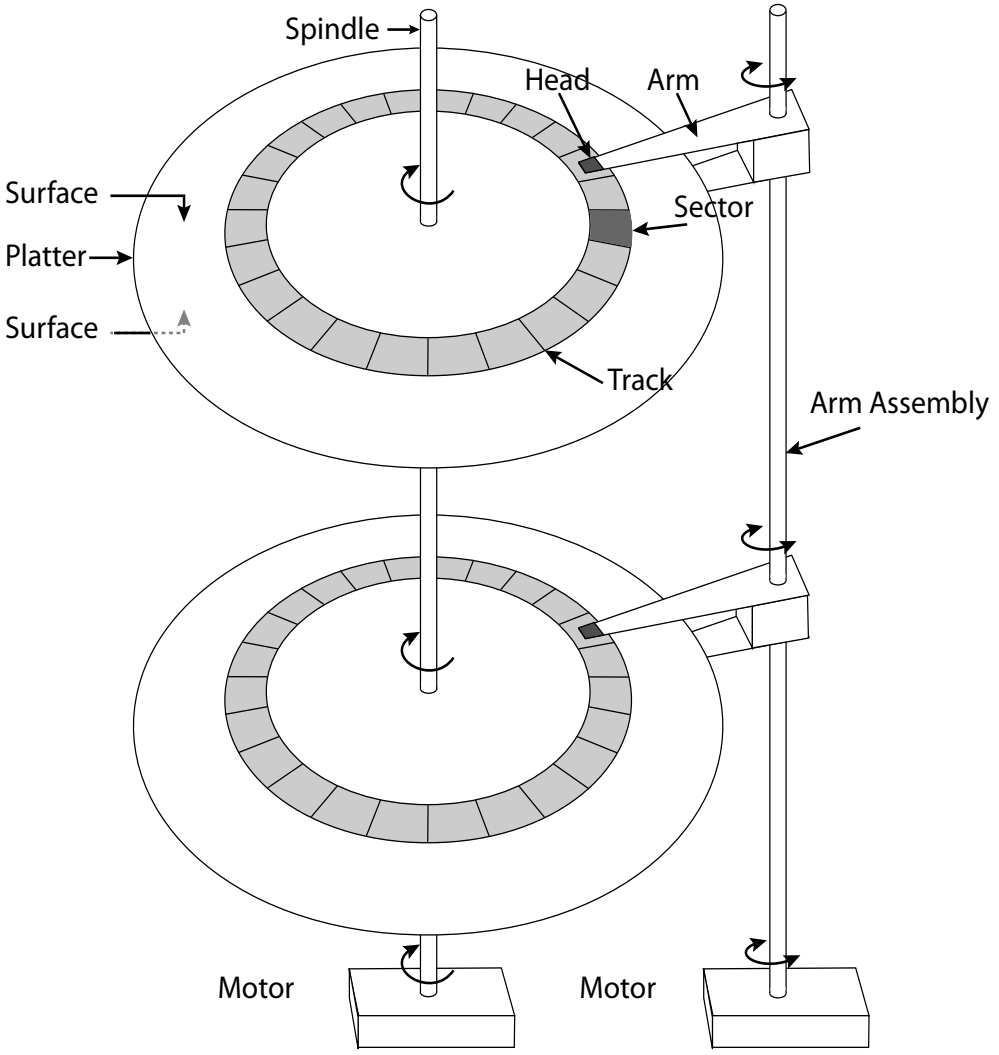
Hard Disk Drives (HDDs)



IBM Personal Computer/AT (1986)
- 30 MB hard disk - $500
- 30-40ms seek time
- 0.7-1 MB/s (est.)

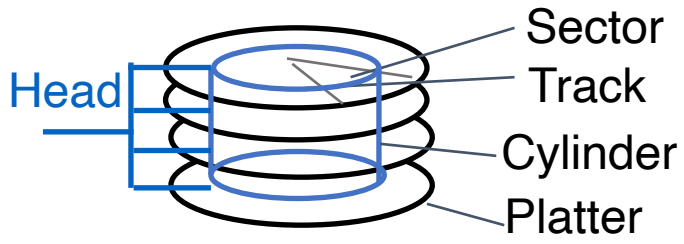
The Amazing Magnetic Disk

- Unit of Transfer: Sector
  - Ring of sectors form a track
  - Stack of tracks form a cylinder
  - Heads position on cylinders
- Disk Tracks ~ 1 micron wide
  - Wavelength of light is ~ 0.5 micron
  - Resolution of human eye: 50 microns
  - 100K on a typical 2.5” disk
- Separated by unused guard regions
  - Reduces likelihood neighboring tracks are corrupted during writes (still a small non-zero chance)
- Track length varies across disk
  - Outside: More sectors per track, higher bandwidth
  - Disk is organized into regions of tracks with same # of sectors/track
  - Only outer half of radius is used
    - Most of the disk area in the outer regions of the disk

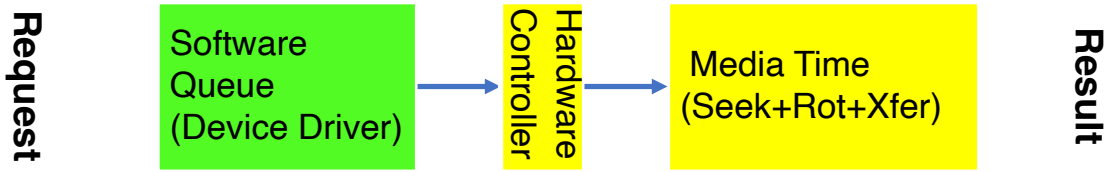


Magnetic Disk Characteristic

- Cylinder: all the tracks under the head at a given point on all surfaces
- Read/write: three-stage process:
  - **Seek time**: position the head/arm over the proper track (into proper cylinder)
  - **Rotational latency**: wait for the desired sector to rotate under the read/write head
  - **Transfer time**: transfer a block of bits (sector) under the read-write head



- Disk Latency = Queuing Time + Controller time + Seek Time + Rotation Time + Xfer Time



- Highest Bandwidth:
  - Transfer large group of blocks sequentially from one track

Typical Numbers for Magnetic Disk

Parameter	Info / Range
Average seek time	**Typically 5-10 milliseconds**. Depending on reference locality, actual cost may be 25-33% of this number.
Average rotational latency	Most laptop/desktop disks rotate at 3600-7200 RPM (16-8 ms/rotation). Server disks up to 15,000 RPM. Average latency is halfway around disk yielding corresponding times of **8-4 milliseconds**
Controller time	Depends on controller hardware
Transfer time	**Typically 50 to 100 MB/s**. Depends on: • Transfer size (usually a sector): 512B – 1KB per sector • Rotation speed: 3600 RPM to 15000 RPM • Recording density: bits per inch on a track • Diameter: ranges from 1 in to 5.25 in
Cost	Drops by a factor of two every 1.5 years (or even faster). **$0.03-0.07/GB in 2013**

Intelligence in the controller

- Sectors contain sophisticated error correcting codes
  - Disk head magnet has a field wider than track
  - Hide corruptions due to neighboring track writes
- Sector sparing
  - Remap bad sectors transparently to spare sectors on the same surface
- Slip sparing
  - Remap all sectors (when there is a bad sector) to preserve sequential behav ior
- Track skewing
  - Sector numbers offset from one track to the next, to allow for disk head m ovement for sequential ops
- …

Question 1.

- How long to complete 500 random disk reads, in FIFO order?

- How long to complete 500 random disk reads, in FIFO order?
  - Seek: average 10.5 msec
  - Rotation: average 4.15 msec
  - Transfer: 5-10 usec
- 500 * (10.5 + 4.15 + 0.01)/1000 = 7.3 seconds

Question 2.

- How long to complete 500 sequential disk reads?

- How long to complete 500 sequential disk reads?
  - Seek Time: 10.5 ms (to reach first sector)
  - Rotation Time: 4.15 ms (to reach first sector)
  - Transfer Time: (outer track)
    500 sectors * 512 bytes / 128MB/sec = 2ms
- Total: 10.5 + 4.15 + 2 = 16.7 ms
  - Might need an extra head or track switch (+1ms)
  - Track buffer may allow some sectors to be read off disk out of order (-2ms)

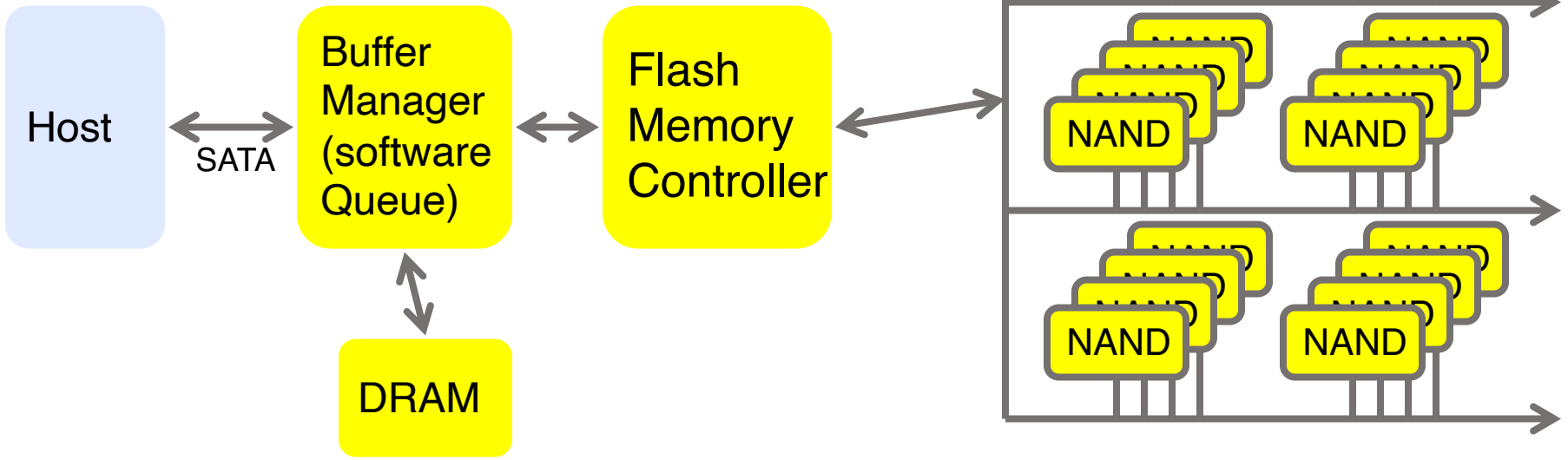
Solid State Disks (SSDs)

- 1995 – Replace rotating magnetic media with non-volatile memory (battery backe d DRAM)
- 2009 – Use NAND Multi-Level Cell (2-bit/cell) flash memory
  - Sector (4 KB page) addressable, but stores 4-64 “pages” per memory block
- No moving parts (no rotate/seek motors)
  - Eliminates seek and rotational delay (0.1-0.2ms access time)
  - Very low power and lightweight
  - Limited “write cycles”
- Rapid advance in capacity and cost since

SSD Architecture – Reads

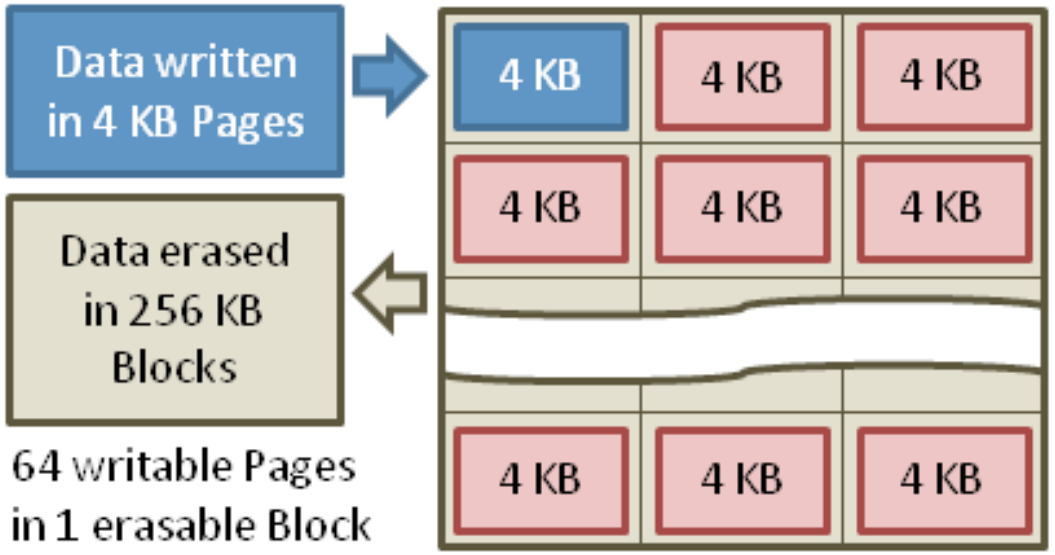
Read 4 KB Page: ~25 usec

- No seek or rotational latency
- Transfer time: transfer a 4KB page
  - SATA: 300-600MB/s =>
    ~4 x10^3 b / 400 x 10^6 bps => 10 us
- Latency = Queuing Time + Controller time + Xfer Time
- Highest Bandwidth: Sequential OR Random reads



SSD Architecture – Writes (I)

- Writing data is complex!
  (~200µs – 1.7ms)
  - Can only write empty pages in a block
  - Erasing a block takes ~1.5ms
  - Controller maintains pool of empty blocks by coalescing used pages (read, erase, write), also reserves some % of capacity
- Rule of thumb: writes 10x reads, era sure 10x writes



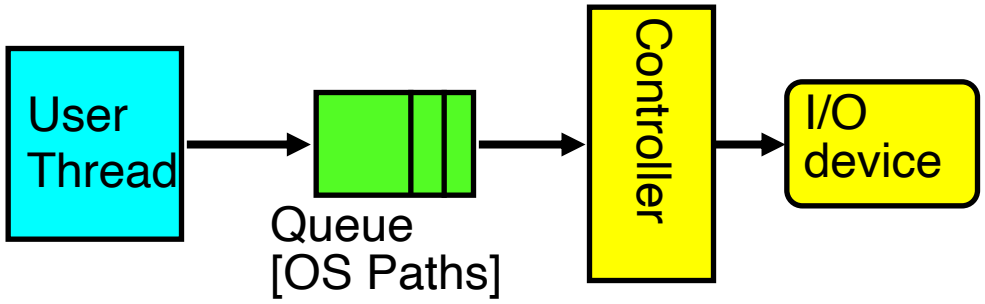
https://en.wikipedia.org/wiki/Solid-state_drive

SSD Summary

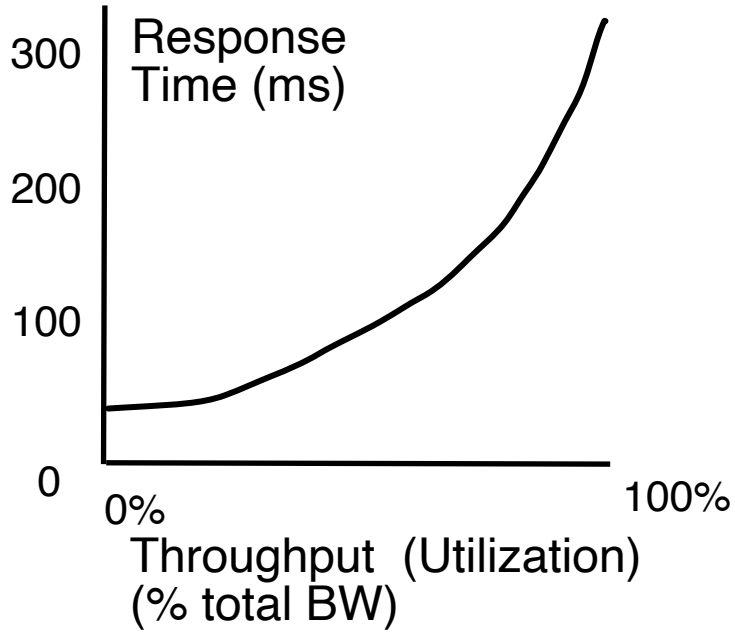
- Pros (vs. hard disk drives):
  - Low latency, high throughput (eliminate seek/rotational delay)
  - No moving parts:
    - Very light weight, low power, silent, very shock insensitive
  - Read at memory speeds (limited by controller and I/O bus)
- Cons
  - Small storage (0.1-0.5x disk), expensive (20x disk ???)
    - Hybrid alternative: combine small SSD with large HDD
  - Asymmetric block write performance: read pg/erase/write pg
    - Controller garbage collection (GC) algorithms have major effect on performance
  - Limited drive lifetime
    - 1-10K writes/page for MLC NAND
    - Avg failure rate is 6 years, life expectancy is 9–11 years
- These are changing rapidly

I/O and System Performance

I/O Performance

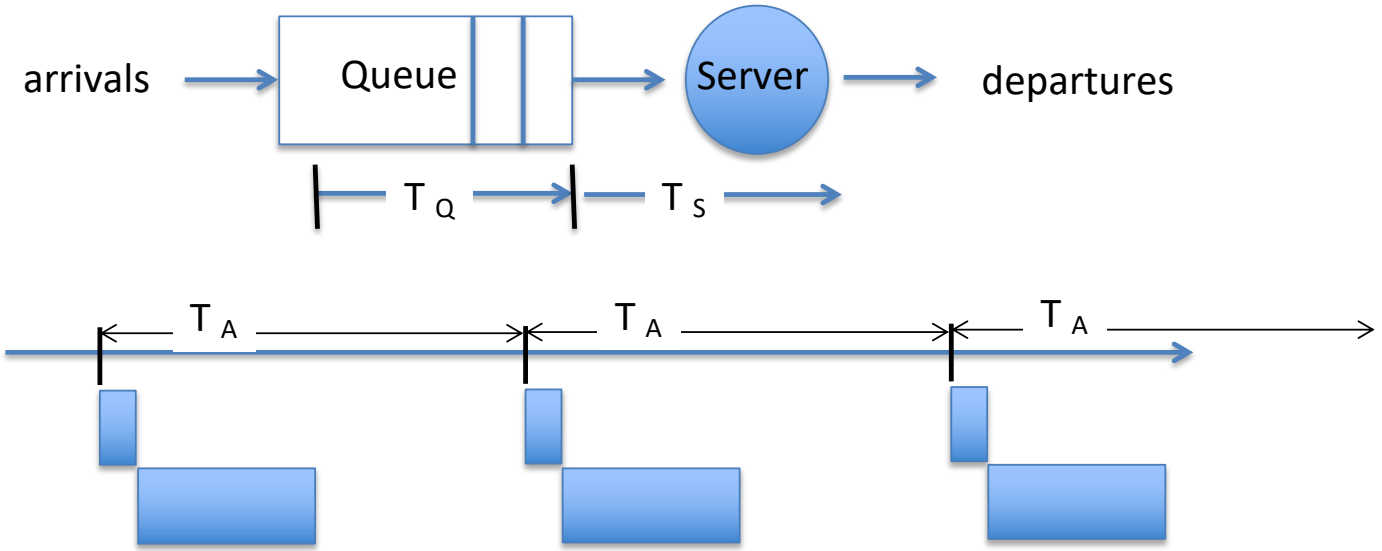


Response Time = Queue + I/O device service time



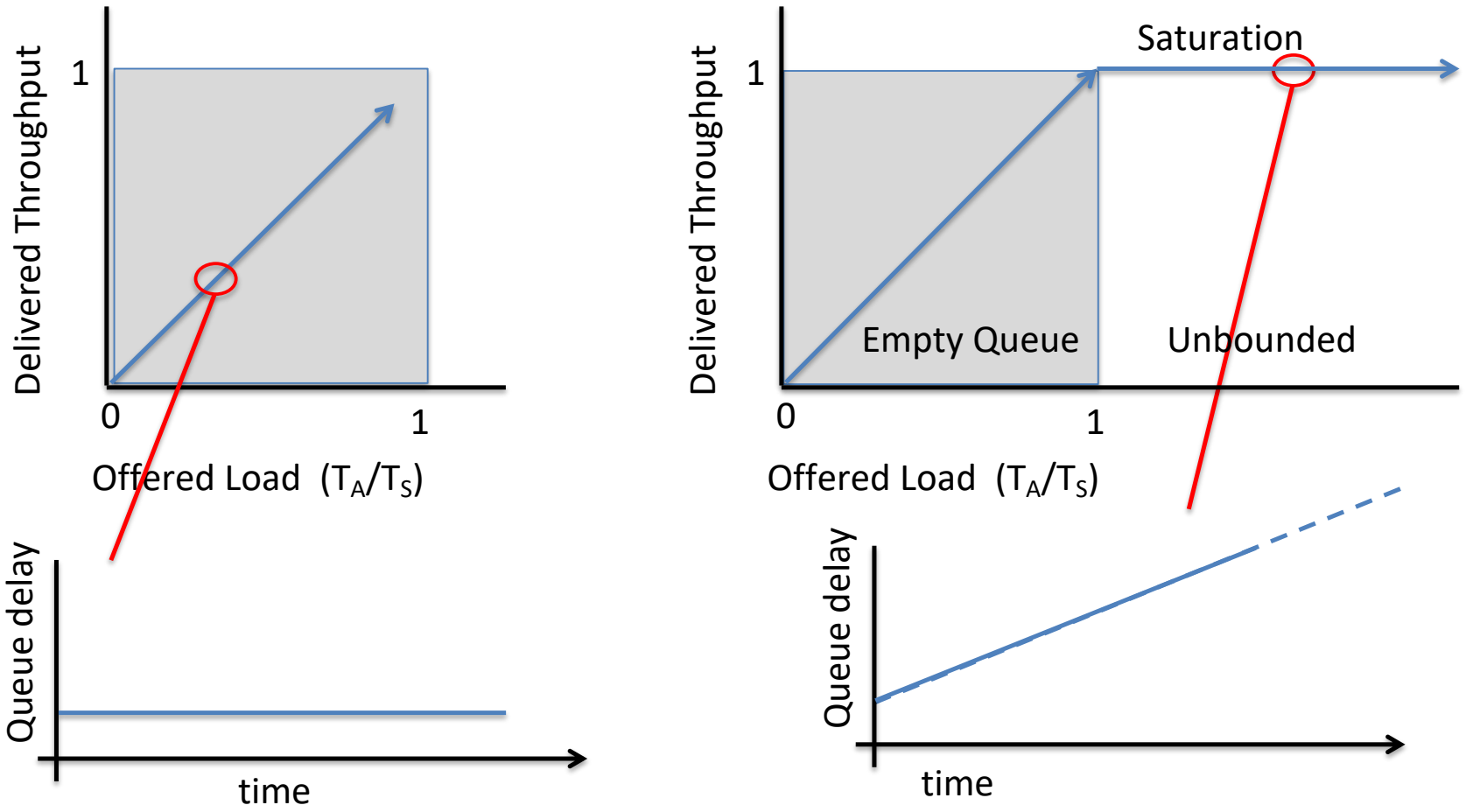
- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Effective BW per op = transfer size / response time
    - EffBW(n) = n / (S + n/B) = B / (1 + SB/n)
  - Contributing factors to latency:
    - Software paths (can be loosely modeled by a queue)
    - Hardware controller
    - I/O device service time
- Queuing behavior:
  - Can lead to big increases of latency as utilization increases
  - Solutions?

A Simple Deterministic World



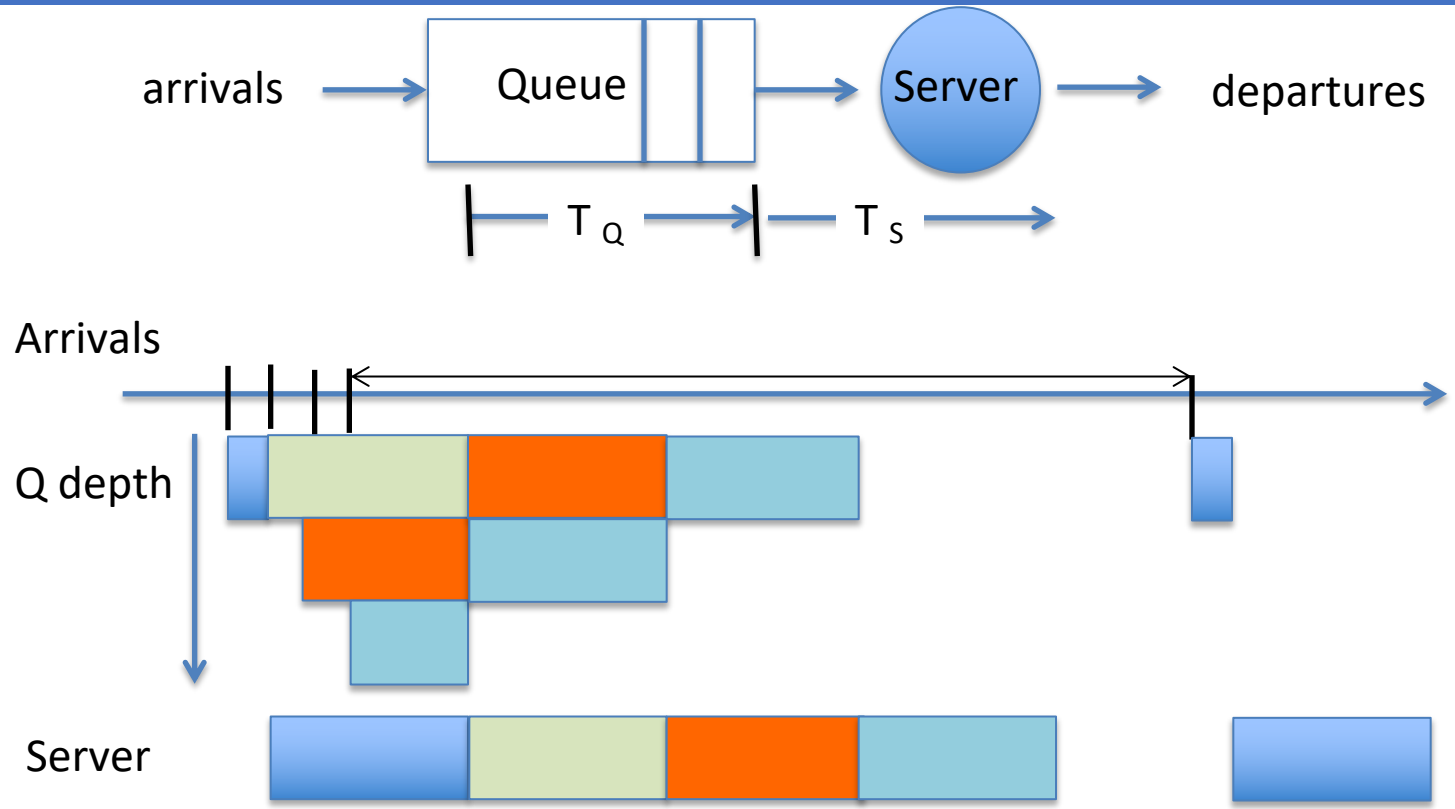
- Assume requests arrive at regular intervals, take a fixed time to process, with plenty o f time between …
- Service rate ($\mu = 1/T_S$) - operations per sec
- Arrival rate: ($\lambda = 1/T_A$) - requests per second
- Utilization: $U = \lambda/\mu$, where $\lambda < \mu$
- Average rate is the complete story

An Ideal Linear World



- What does the queue wait time look like?
  - Grows unbounded at a rate ~ (T_s/T_A) till request rate subsides

A Bursty World

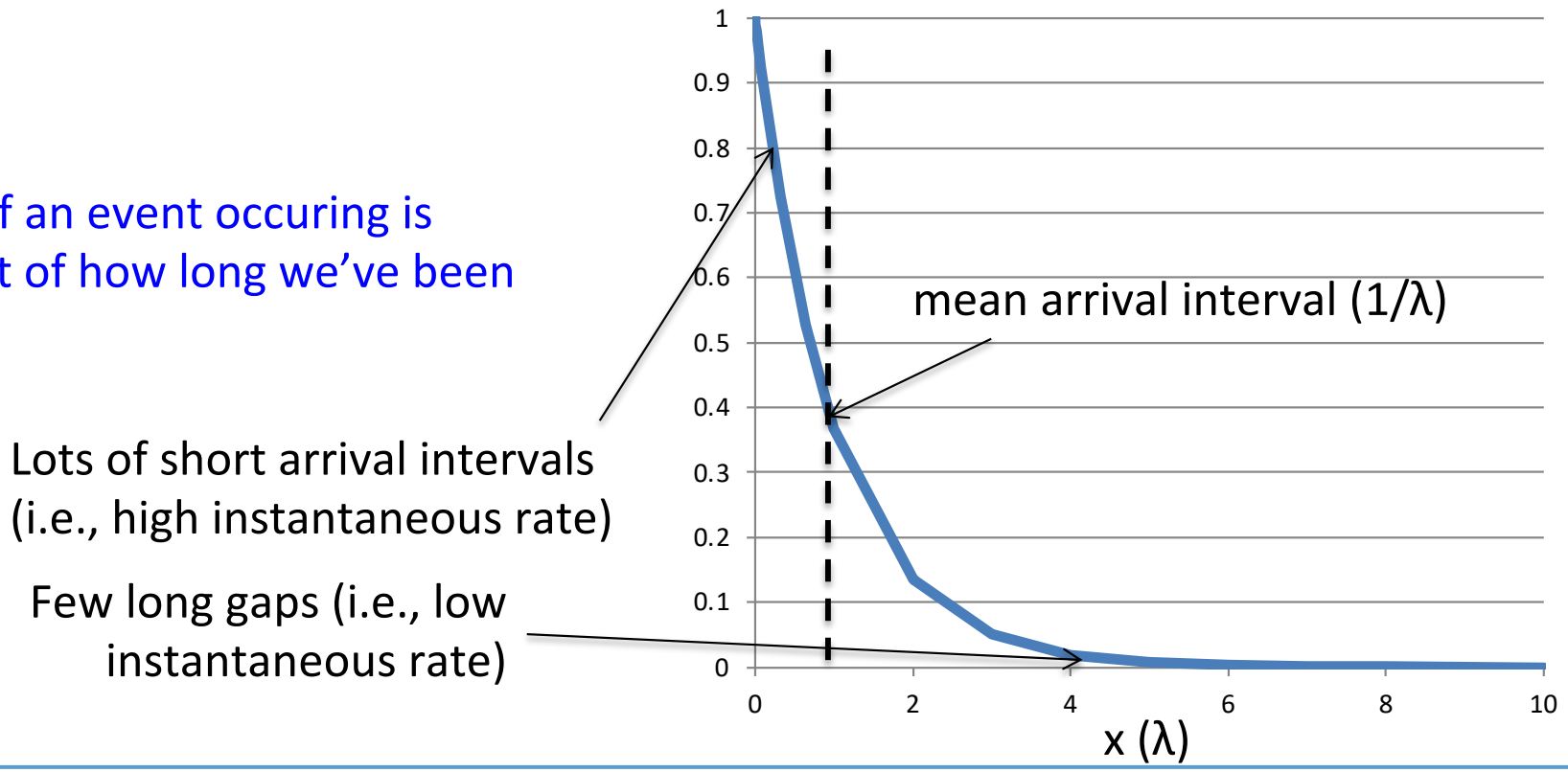


- Requests arrive in a burst, must queue up till served
- Same average arrival time, but almost all of the requests experience large queue delays
- Even though average utilization is low

So how do we model the burstiness?

- Elegant mathematical framework if you start with *exponential distri bution*
  - Probability density function of a continuous random variable with a mean o f 1/λ
  - $f(x) = \lambda e^{-\lambda x}$
  - *"Memoryless"*

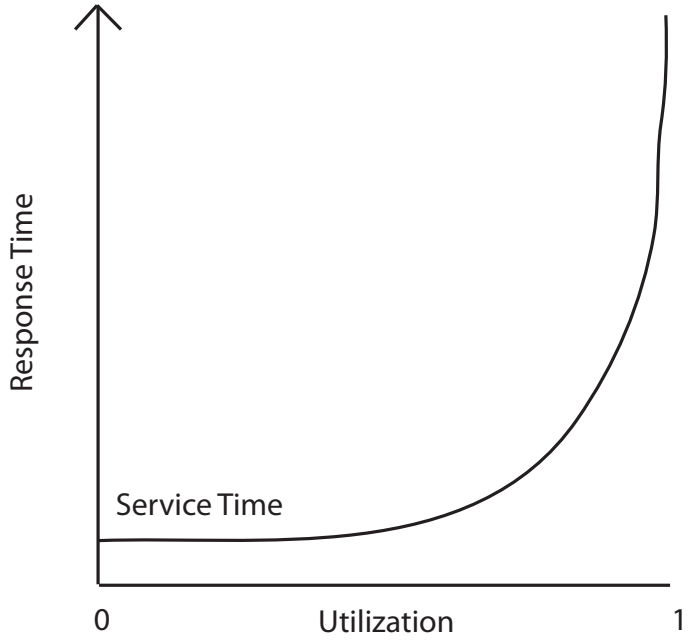
Likelihood of an event occuring is independent of how long we've been waiting



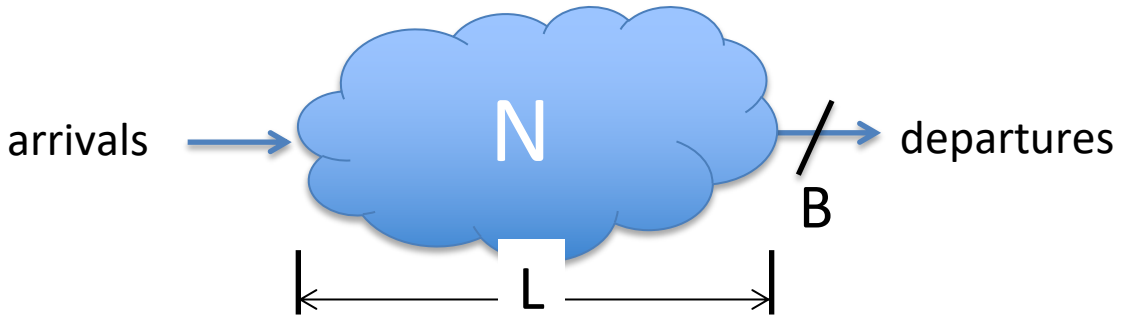
How Long should we expect to wait?

- RespTime = ServTime * 1/(1-U)
  - Better if gaussian (spread around the mean)
- Variance in R = S/(1-U)^2

Response Time vs. Utilization



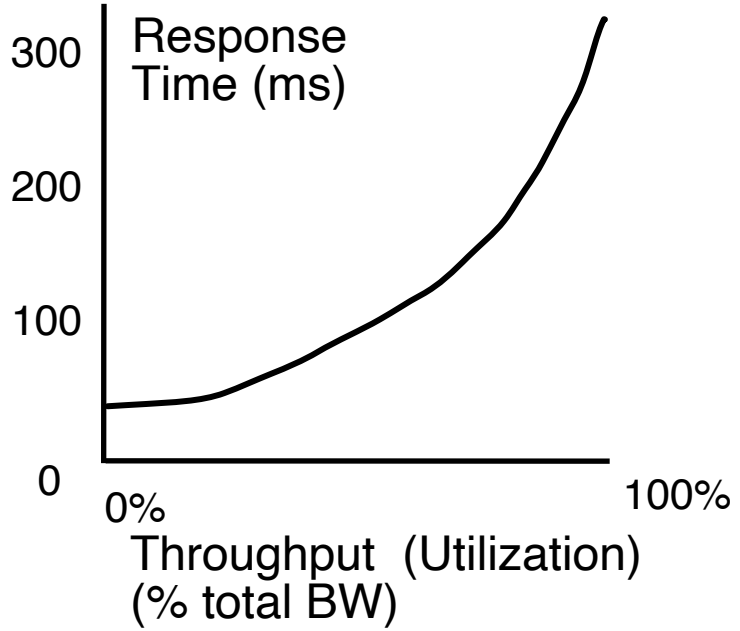
Little's Law

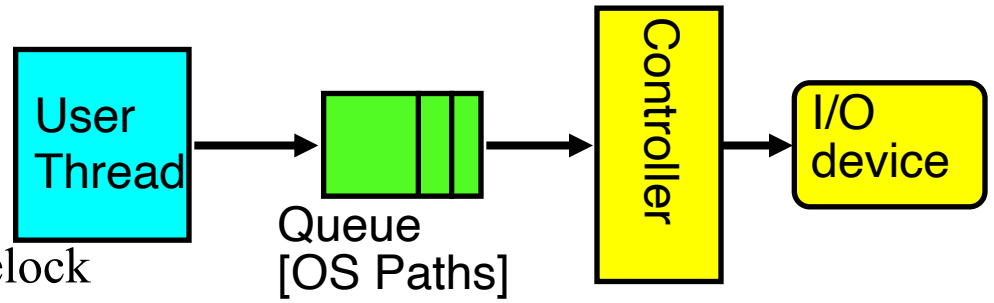


- In any *stable* system
  - Average arrival rate = Average departure rate
- the average number of tasks in the system (N) is equal to the throu ghput (B) times the response time (L)
- N (ops) = B (ops/s) x L (s)
- Regardless of structure, bursts of requests, variation in service
  - instantaneous variations, but it washes out in the average
  - Overall requests match departures

I/O Performance

- Solutions?
  - Make everything faster ☺
  - More Decoupled (Parallelism) systems
    - multiple independent buses or controllers
  - Optimize the bottleneck to increase service rate
    - Use the queue to optimize the service
  - Do other useful work while waiting
- Queues absorb bursts and smooth the flow
- Admissions control (finite queues)
  - Limits delays, but may introduce unfairness and livelock





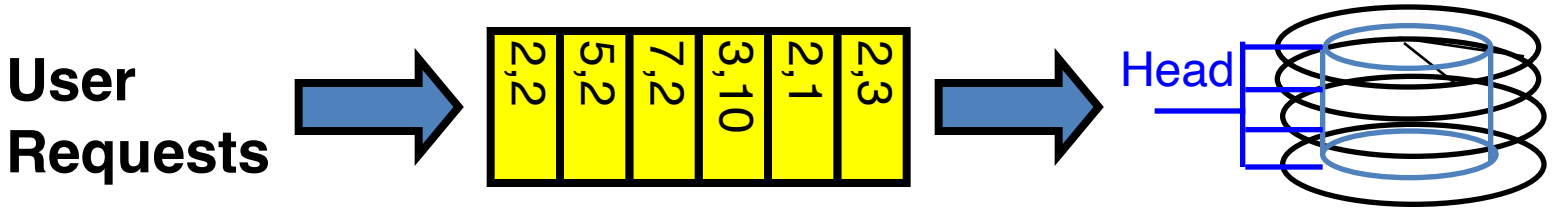
Response Time = Queue + I/O device service time

Ex: Disk Performance Examples

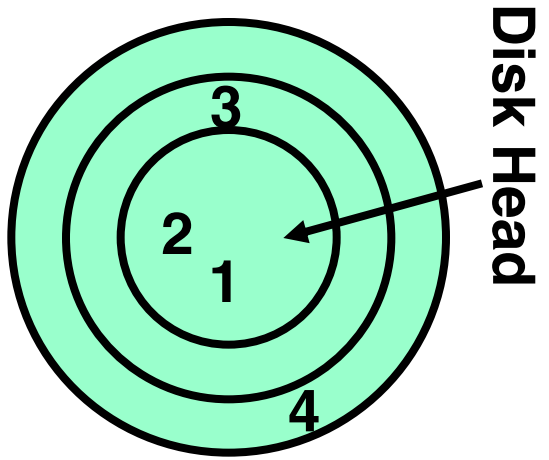
- Assumptions:
  - Ignoring queuing and controller times for now
  - Avg seek time of 5ms,
  - 7200RPM ⇒ Time for one rotation: 60000ms/7200 ~= 8ms
  - Transfer rate of 4MByte/s, sector size of 1 KByte
- Read sector from random place on disk:
  - Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  - Approx 10ms to fetch/put data: **100 KByte/sec**
- Read sector from random place in same cylinder:
  - Rot. Delay (4ms) + Transfer (0.25ms)
  - Approx 5ms to fetch/put data: **200 KByte/sec**
- Read next sector on same track:
  - Transfer (0.25ms): **4 MByte/sec**
- Key to using disk effectively (especially for file systems) is to *minimize seek and rotational delays*

Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queu ed requests?
  - Request denoted by (track, sector)



- Scheduling algorithms:
  - First In First Out (FIFO)
  - Shortest Seek Time First
  - SCAN
  - C-SCAN
- In our examples we ignore the sector
  - Consider only track #

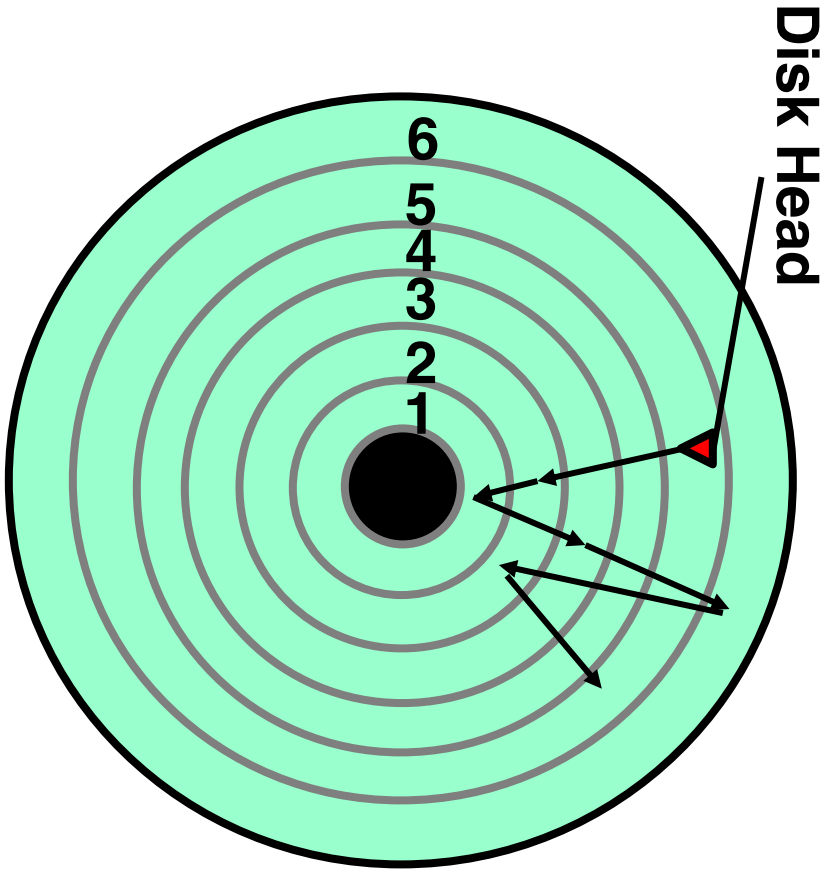


FIFO: First In First Out

- Schedule requests in the order the y arrive in the queue

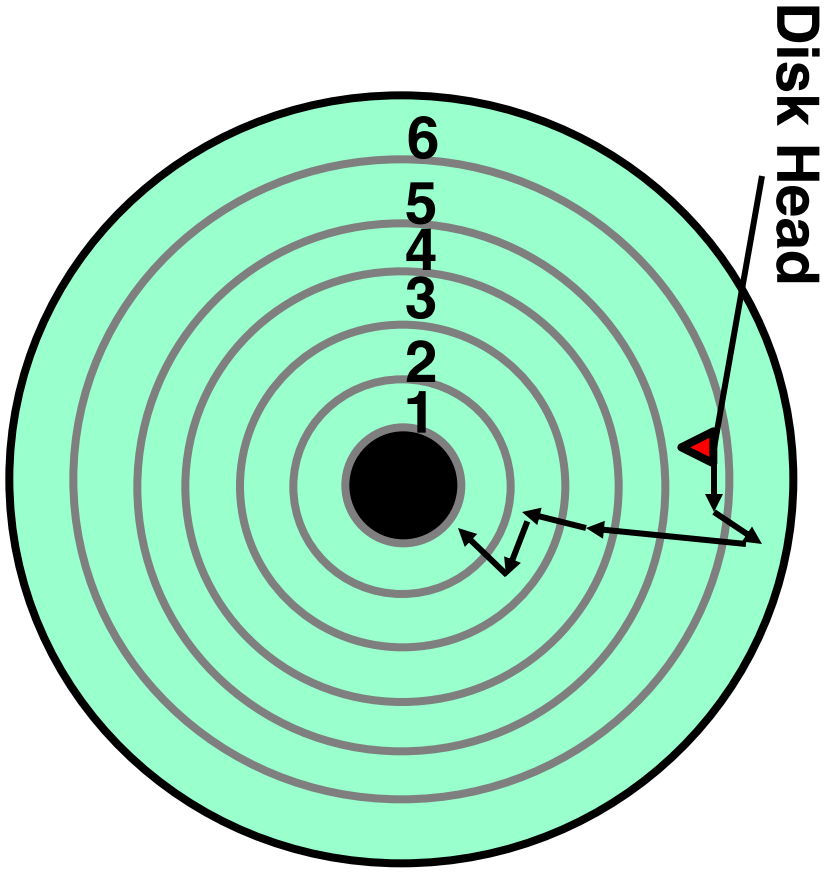
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Scheduling order: 2, 1, 3, 6, 2, 5
  - 16 tracks, 6 seeks

- Pros: Fair among requesters
- Cons: Order of arrival may be to r andom spots on the disk ⇒ Very l ong seeks



SSTF: Shortest Seek Time First

- Pick the request that's closest to the head on the disk
  - Although called SSTF, include rotational delay in calculation, as rotation can be as long as seek
- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Scheduling order: 5, 6, 3, 2, 2, 1
  - 6 tracks, 4 seeks
- Pros: reduce seeks
- Cons: may lead to starvation
  - Greedy. Not optimal

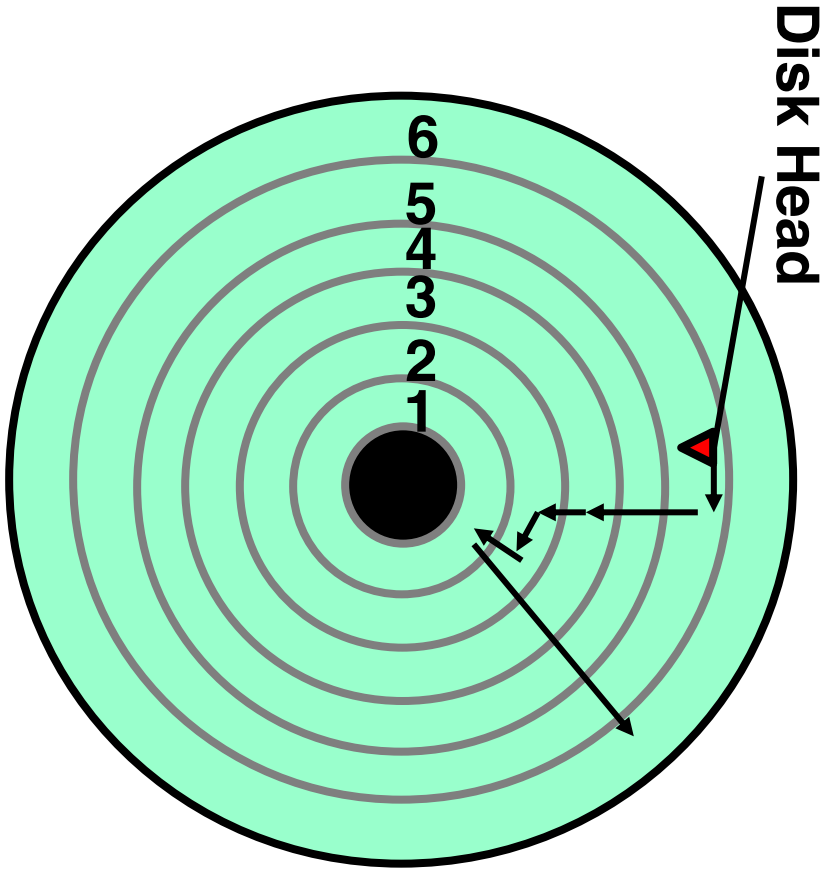


SCAN

- Implements an Elevator Algorithm: take the closest request in the direction of travel

- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head is moving towards center
  - Scheduling order: 5, 3, 2, 2, 1, 6
  - 8 tracks, 4 seeks

- Pros:
  - No starvation
  - Low seek
- Cons: favors middle tracks
  - May spend time on sparse tracks while dense requests elsewhere

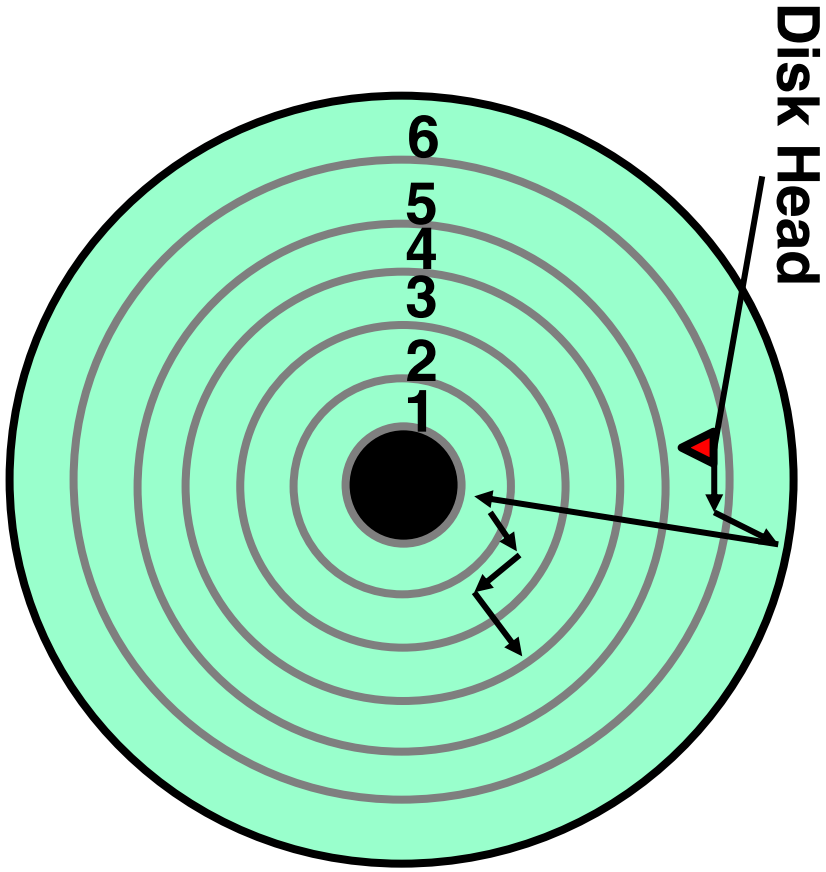


C-SCAN

- Like SCAN but only serves request in only one direction

- Example:
  - Request queue: 2, 1, 3, 6, 2, 5
  - Head only serves request on its way from center towards edge
  - Scheduling order: 5, 6, 1, 2, 2, 3
  - 8 tracks, 5 seeks

- Pros:
  - Fairer than SCAN
  - Accumulate work in remote region then go get it
- Cons: longer seeks on the way back
- Optimization: dither to pickup nearby requests as you go



When is the disk performance highest

- When there are big sequential reads, or
- When there is so much work to do that they can be piggy backed (c-scan)

- OK, to be inefficient when things are mostly idle
- Bursts are both a threat and an opportunity
- <your idea for optimization goes here>
  - Waste space for speed?

How do we hide I/O latency?

- Blocking Interface: "Wait"
  - When request data (*e.g.*, read() system call), put process to sleep until data is ready
  - When write data (*e.g.*, write() system call), put process to sleep until device is ready fo r data
- Non-blocking Interface: "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred to kernel
  - Read may return nothing, write may write nothing
- Asynchronous Interface: "Tell Me Later"
  - When requesting data, take pointer to user's buffer, return immediately; later kernel fil ls buffer and notifies user
  - When sending data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

Kernel vs User-level I/O

- Both are popular/practical for different reasons:
  - Kernel-level drivers for critical devices that must keep running, e.g. display drivers.
    - Programming is a major effort, correct operation of the rest of the kernel depends on correct driver operation.
  - User-level drivers for devices that are non-threatening, e.g USB devices in Linux (libu sb).
    - Provide higher-level primitives to the programmer, avoid every driver doing low-level I/O register t weaking.
    - The multitude of USB devices can be supported by Less-Than-Wizard programmers.
    - New drivers don't have to be compiled for each version of the OS, and loaded into the kernel.

Kernel vs User-level Programming Styles

- Kernel-level drivers
  - Have a much more limited set of resources available:
    - Only a fraction of libc routines typically available.
    - Memory allocation (e.g. Linux kmalloc) much more limited in capacity and required to be physicall y contiguous.
    - Should avoid blocking calls.
    - Can use asynchrony with other kernel functions but tricky with user code.
- User-level drivers
  - Similar to other application programs but:
    - Will be called often – should do its work fast, or postpone it – or do it in the background.
    - Can use threads, blocking operations (usually much simpler) or non-blocking or asynchronous.

Some performance guidance

- Concurrency and Parallelism
  - Make I/O work asynchronous to CPU jobs
  - Make your cpu cores busy
  - Thread-level parallelism
- Avoid copy as much as possible
  - share page with control
- batch jobs for reduce per-transaction overhead
  - trade throughput with latency, if you want
- Consider cache and memory hierarchy
- Compiler and Symbolic execution
- ref_google Performance engineering for software systems

Mini-lab.