# TCSS 422: OPERATING SYSTEMS

## HDDs, RAID, File Systems Final Exam Review

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

June 4, 2020 TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

## FALL 2020 - TCSS 562 SOFTWARE ENG. FOR CLOUD COMPUTING

- This is a "cloud computing" course
- Previous year's course: http://faculty.washington.edu/wlloyd/courses/tcss562
- Course introduces major cloud computing delivery models: Infrastructure-as-a-Service (IaaS), Platform (PaaS), Functions (FaaS), Container (CaaS), Software (SaaS)
- Course features a software development project where we build and evaluate software entirely for the cloud
- Fall 2019 focus: developing serverless software: e.g. data processing pipelines
- Fall 2020 focus: serverless/cloud-native software, containerization, cloud services

| June 4, 2020 | TCSS422: Operating Systems [Spring 2020]<br>School of Engineering and Technology, University of Washington - Tacoma | L19.2 |
|---|---|---|

## TCSS 562 – CLOUD COMPUTING - 2

- Class does not have prerequisites
- TCSS 422 provides good foundation – we use Linux
- If interested in enrolling, contact by email
- Can take 1 x 500-level class, counts as 400-level elective
  - SAVINGS: able to take graduate course and only pay undergraduate tuition
- DOUBLE-DIP !!
  - Class taken in last quarter of undergrad can be used twice
    - Once as a undergraduate elective towards graduation
    - Once as a graduate elective towards the Masters in Computer Science & Systems (MSCSS) degree

June 4, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L19.3

## NEXT YEAR - TCSS 498/499 (ANY QUARTER) UNDERGRADUATE READIN/RESEARCH IN CSS

- Independent study in "cloud computing"
- Work to collaboratively draft a proposal and submit to Dr. Chinn, CSS Chair for Approval
- Focus on variety of topics related to cloud/distributed systems
- Variable credits from 1 to 5
- Involves participation in weekly research group meeting
  - Spring 2020: currently Wednesday at 12:30p
- Usually 1 or 2 one-on-one or small group meeting during week
- Contact by email if interested
- Identify preferred quarter(s)
- Number of credits
- Can take up to 10 credits TCSS 498/499 - CSS elective credits

June 4, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L19.4

## COURSE EVALUATION: TCSS 422 B SPRING 2020

- Please complete the course evaluation survey at:
- TCSS 422 B - Computer Operating Systems:
- https://uwt.iasystem.org/survey/106940
- New this quarter:
- Assignment 2– available in Java or C
- Tutorial 2– parallel prime number generation
- Assignment 3– Kernel Module programming- tutorial format
- Course entirely online & recorded
- Paperless daily feedback surveys
- Quizzes with multiple attempts
- Problem solutions w/ document cam
- Open book, note, and internet midterm and final exam
- Slide revisions & refactoring for 100% online delivery



## OBJECTIVES – 6/4

- Questions from 6/2
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Assignment 3 – on Linux kernel programming – offered in "tutorial" format – due Sat June 13
- Quiz 4 – Page Tables – optional – provides practice problems – to be posted
- Chapter 37: Hard Disk Drives
- Chapter 38: RAID (Redundant array of inexpensive disks)- *very brief*
- Chapter 39/40: File Systems – *very brief*
- Practice Final Exam Questions – Today – 2nd hour

## MATERIAL / PACE

- Please classify your perspective on material covered in today's class (35 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 6.44 (↓ *from 7.3*)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 5.56 (↓ from 5.83)

| June 4, 2020 | TCSS422: Operating Systems [Spring 2020]<br>School of Engineering and Technology, University of Washington - Tacoma | L19.7 |
|---|---|---|

## FEEDBACK FROM 6/2

-

| June 4, 2020 | TCSS422: Operating Systems [Spring 2020]<br>School of Engineering and Technology, University of Washington - Tacoma | L19.8 |
|---|---|---|

## OBJECTIVES – 6/4

- Questions from 6/2
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Assignment 3 – on Linux kernel programming – offered in "tutorial" format – due Sat June 13
- Quiz 4 – Page Tables – optional – provides practice problems – to be posted
- Chapter 37: Hard Disk Drives
- Chapter 38: RAID (Redundant array of inexpensive disks)- *very brief*
- Chapter 39/40: File Systems – *very brief*
- Practice Final Exam Questions – Today – 2nd hour



## OBJECTIVES – 6/4

- Questions from 6/2
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Assignment 3 – on Linux kernel programming – offered in "tutorial" format – due Sat June 13
- Quiz 4 – Page Tables – optional – provides practice problems – to be posted
- Chapter 37: Hard Disk Drives
- Chapter 38: RAID (Redundant array of inexpensive disks)- *very brief*
- Chapter 39/40: File Systems – *very brief*
- Practice Final Exam Questions – Today – 2nd hour

# OBJECTIVES – 6/4

- Questions from 6/2
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Assignment 3 – on Linux kernel programming – offered in “tutorial” format – due Sat June 13
- Quiz 4 – Page Tables – optional – provides practice problems – to be posted
- Chapter 37: Hard Disk Drives
- Chapter 38: RAID (Redundant array of inexpensive disks)- *very brief*
- Chapter 39/40: File Systems – *very brief*
- Practice Final Exam Questions – Today – 2nd hour



# CH. 37: HARD DISK DRIVES

# OBJECTIVES – 6/4

- Questions from 6/2
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Assignment 3 – on Linux kernel programming – offered in "tutorial" format – due Sat June 13
- Quiz 4 – Page Tables – optional – provides practice problems – to be posted
- Chapter 37: Hard Disk Drives
- Chapter 38: RAID (Redundant array of inexpensive disks)- *very brief*
- Chapter 39/40: File Systems – *very brief*
- Practice Final Exam Questions – Today – 2nd hour

| June 4, 2020 | TCSS422: Operating Systems [Spring 2020]<br>School of Engineering and Technology, University of Washington - Tacoma | L19.13 |
|---|---|---|

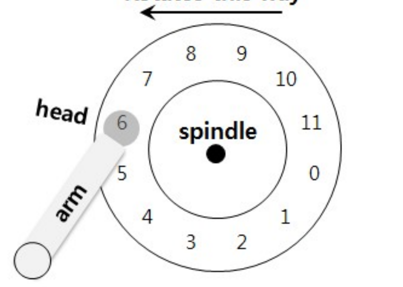
# OBJECTIVES

- Chapter 37
  - HDD Internals
  - Seek time
  - Rotational latency
  - Transfer speed
  - Capacity
  - Scheduling algorithms

| June 4, 2020 | TCSS422: Operating Systems [Spring 2020]<br>School of Engineering and Technology, University of Washington - Tacoma | L19.14 |
|---|---|---|

# EXAMPLE: SIMPLE DISK DRIVE

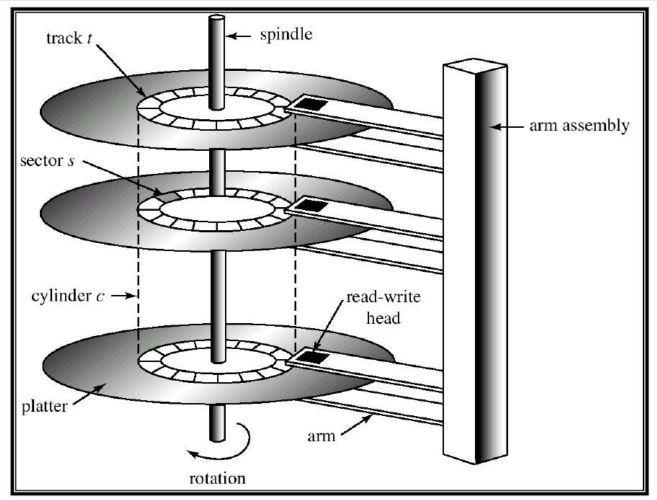
- Single track disk
- Head: one per surface of drive
- Arm: moves heads across surface of platters




A Single Track Plus A Head



# HARD DISK STRUCTURE

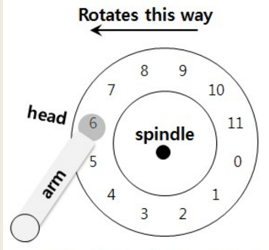


# SINGLE-TRACK LATENCY: THE ROTATIONAL DELAY

- **Rotational latency** ($T_{rotation}$): time to rotate to desired sector
- Average $T_{rotation}$ is ~ about half the time of a full rotation
- How to calculate $T_{rotation}$ from rpm

1. Calculate time for 1 rotation based on rpm
   > Convert rpm to rps
2. Divide by two (*average rotational latency*)

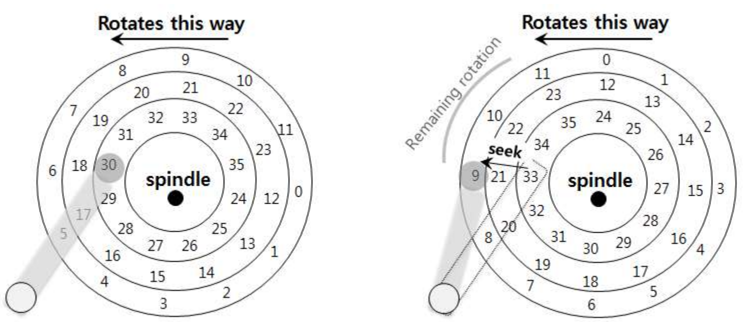
- 7200rpm = 8.33ms per rotation /2= ~4.166ms
- 10000rpm = 6ms per rotation /2= ~3ms
- 15000rpm = 4ms per rotation /2= ~2ms



A Single Track Plus A Head



# SEEK TIME



Three Tracks Plus A Head (Right: With Seek)
(e.g., read to sector 11)

- **Seek time** ($T_{seek}$): time to move disk arm to proper track
- Most time consuming HDD operation

# FOUR PHASES OF SEEK

- Acceleration → coasting → deceleration →settling
- **Acceleration**: the arm gets moving
- **Coasting**: arm moving at full speed
- **Deceleration**: arm slow down
- **Settling**: Head is carefully positioned over track
  - Settling time is often high, from .5 to 2ms

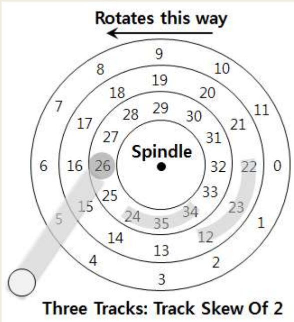


# HDD I/O

- Data transfer
  - Final phase of I/O: time to read or write to disk surface
- Complete I/O cycle:
  1. Seek (accelerate, coast, decelerate, settle)
  2. Wait on rotational latency (*until track aligns*)
  3. Data transfer

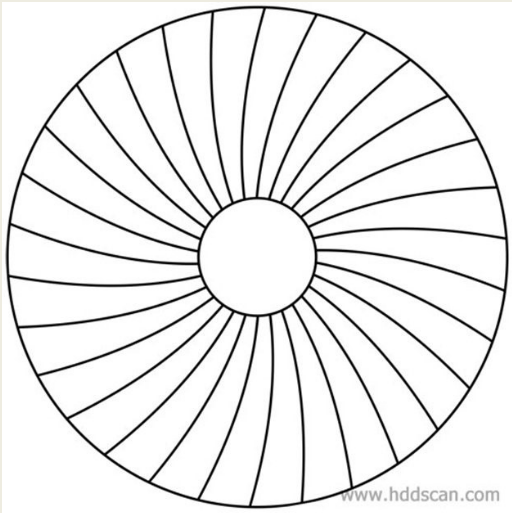
# TRACK SKEW

- Sectors are offset across tracks to allow time for head to reposition for sequential reads
- Without track skew, when head is repositioned sector would have already been passed






# TRACK SKEW - 2



# HDD CACHE

- Buffer to support caching reads and writes
- Improves drive response time
- Up to 256 MB, slowly have been growing
- Two styles
  - Writeback cache
    - Report write complete immediately when data is transferred to HDD cache
    - Dangerous if power is lost
  - Writethrough cache
    - Reports write complete only when write is physically completed on disk



# TRANSFER SPEED

- Can calculate I/O transfer speed with:
- I/O Time: $T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$
- $T_{transfer}$ = $DATA_{size}$ x $Rate_{I/O}$
- Rate of I/O: $R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$

# EXAMPLE: I/O SPEED

- Compare two disks:

1. Random workload: 4KB (random read on HDD)
2. Sequential workload: 100MB (contiguous sectors)

> Calculate $T_{rotation}$ from rpm (rpm→rps, time for 1 rotation / 2)

| | Cheetah 15K.5 | Barracuda |
|---|---|---|
| Capacity | 300 GB | 1 TB |
| RPM | 15,000 | 7,200 |
| Average Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |
| Platters | 4 | 4 |
| Cache | 16 MB | 16/32 MB |
| Connects Via | SCSI | SATA |

Disk Drive Specs: SCSI Versus SATA



# EXAMPLE: I/O SPEED

1. Random workload: 4KB (random read on HDD)
2. Sequential workload: 100MB (contiguous sectors)

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

$$T_{transfer} = Data_{size} \ x \ Rate_{I/O}$$

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$$

| | | Cheetah 15K.5 | Barracuda |
|---|---|---|---|
| | $T_{seek}$ | 4 ms | 9 ms |
| | $T_{rotation}$ | 2 ms | 4.2 ms |
| 4 KB Random | $T_{transfer}$ | 30 microsecs | 38 microsecs |
| | $T_{I/O}$ | 6 ms | 13.2 ms |
| | $R_{I/O}$ | 0.66 MB/s | 0.31 MB/s |
| 100 MB Sequential | $T_{transfer}$ | 800 ms | 950 ms |
| | $T_{I/O}$ | 806 ms | 963.2 ms |
| | $R_{I/O}$ | 125 MB/s | 105 MB/s |

Disk Drive Performance: SCSI Versus SATA

There is a huge gap in drive throughput between random and sequential workloads

# MODERN HDD SPECS

- See sample HDD configurations here:
  - Up to 20 TB
- https://www.westerndigital.com/products/data-center-drives#hard-disk-hdd

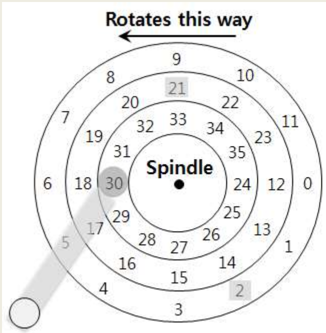


# DISK SCHEDULING

- Disk scheduler: determine how to order I/O requests
- Multiple levels of scheduling: OS and HW
- OS: provides ordering
- HW: further optimizes using intricate details of physical HDD implementation and state

# DISK SCHEDULING ALGORITHMS - 1

- Disk scheduling: which I/O request to schedule next
- **Shortest Seek Time First (SSTF)**
  - Order queue of I/O requests by nearest track




SSTF: Scheduling Request 21 and 2

Issue the request to 21 → issue the request to 2



# SSTF ISSUES

- Problem 1: HDD abstraction
- Drive geometry not available to OS. Nearest-block-first is a comparable alternate algorithm.
- Problem 2: Starvation
- Steady stream of requests for local tracks may prevent arm from traversing to other side of platter
  - Keeps head local to a few tracks

# DISK SCHEDULING ALGORITHMS - 2

- **SCAN (SWEEP)**
- Perform single repeated passes back and forth across disk
- Issue: if request arrives for a recently visited track it will not be revisited until a full cycle completes

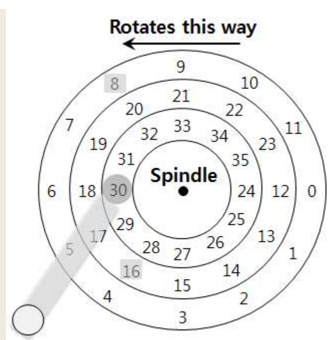
- **F-SCAN**
- Freeze incoming requests by adding to queue during scan
- Cache arriving requests until later
- Delays help avoid starvation by postponing servicing nearby newly arriving requests vs. requests at edge of sweep
- Provides better fairness

- **Elevator (C-SCAN)** – circular scan
- Sweep only one direction (e.g. outer to inner) and repeat
- SCAN favors middle tracks vs. outer tracks with 2-way sweep

| June 4, 2020 | TCSS422: Operating Systems [Spring 2020]<br>School of Engineering and Technology, University of Washington - Tacoma | L19.31 |
|---|---|---|

# DISK SCHEDULING ALGORITHMS - 3

- **Shortest Positioning Time First**
- Select next sector to read based on which sector can be reached first
  - Use when : $T_{seek} = T_{rotation}$
- Next read depends on current position
  - which track?
  - which sector?




SSTF: Sometimes Not Good Enough

On modern drives, both seek and rotation are roughly equivalent:
**Thus, SPTF (Shortest Positioning Time First) is useful.**

| June 4, 2020 | TCSS422: Operating Systems [Spring 2020]<br>School of Engineering and Technology, University of Washington - Tacoma | L19.32 |
|---|---|---|

## OPTIMIZATION: I/O MERGING

- Group temporary adjacent requests
- Reduce overhead
- Read (memory blocks): 33 8 34
- How long we should wait for I/O ?
- When do we know we have waited too long?

June 4, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L19.33

## CH. 38: RAID

June 4, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L19.34

# OBJECTIVES – 6/4

- Questions from 6/2
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Assignment 3 – on Linux kernel programming – offered in "tutorial" format – due Sat June 13
- Quiz 4 – Page Tables – optional – provides practice problems – to be posted
- Chapter 37: Hard Disk Drives
- Chapter 38: RAID (Redundant array of inexpensive disks)- *very brief*
- Chapter 39/40: File Systems – *very brief*
- Practice Final Exam Questions – Today – 2nd hour



# RAID

- Redundant array of inexpensive disks (RAID)
- Grouping together of multiple disks
- Provides the illusion of one **GIANT** disk
- For performance improvements:
  - **STRIPING**: create big disk by spreading data across several
  - Data reads/writes are automatically distributed to physically different device
  - **MIRRORING:** duplicate disks: read transactions are distributed across disks and can run in parallel
  - 2 disks: each handles 50% of the reads – doubles throughput
- For redundancy / fault tolerance:
  - Mirroring: duplicates data in case of drive failure

## RAID - 2

- Good system administrators will always say:

**RAID is not a backup!**



## RAID LEVEL 0 - STRIPING

- RAID Level 0: Simplest form
- Stripe blocks of data across disks in a round-robin fashion
- Excellent performance and capacity

- Capacity
  - Capacity is equal to the sum of all disks
- Performance
  - R/W are distributed in round-robin fashion across all disks
- Reliability
  - No redundancy

# RAID LEVEL 1 - MIRRORING

- RAID 1 tolerates HDD failure
- Two copies of each block across disks
- RAID 1 Example with 4 disks, each data block saved twice:

| Disk 0 | Disk 1 | Disk 2 | Disk 3 |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 2 | 2 | 3 | 3 |
| 4 | 4 | 5 | 5 |
| 6 | 6 | 7 | 7 |

Simple RAID-1: Mirroring (Keep two physical copies)

- Can suffer the loss of two disks
- Just not two even or odd numbered disks !!



# RAID 1 - EVALUATION

- Capacity: RAID 1 is expensive
  - The useful capacity is n/2
- Reliability: RAID-1 does well
  - Can tolerate the loss of disk(s)
  - Up to n/2 disk failures tolerated depending on which disk fails
- Performance: RAID-1 is slow at writing
  - Must wait for writes to complete to all disk(s)

# RAID 5 – PARITY DISK

- Raid 5 – trades off space requirement for redundancy
- In a 5-disk array, you can only recover from the loss of 1 HDD
- 5 disk RAID 5: Capacity is 80% of 5 disks
- Writes rotate across disks, distributing a parity data
- To rebuild data blocks you only need data from 4 disks
  - Any drive can fail, as long just one drive fails
  - To recover need:
    3 blocks + 1 parity block
    -or-
    4 blocks

| Disk 0 | Disk 1 | Disk 2 | Disk 3 | Disk 4 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | P0 |
| 5 | 6 | 7 | P1 | 4 |
| 10 | 11 | P2 | 8 | 9 |
| 15 | P3 | 12 | 13 | 14 |
| P4 | 16 | 17 | 18 | 19 |

RAID-5 With Rotated Parity



# RAID 5 – EVALUATION

- Capacity: Useful capacity is (n-1) disks
  - A HDD must be dedicated as a parity disk
- Performance
  - Writes are very slow: roughly = n/4
  - Reads are equivalent to a single disk
- Reliability
  - In RAID 5, a disk may fail, and the RAID keeps running
  - Rebuilds are slow !!!
  - Depending on disk size 8-24 hours is not unheard of
- RAID 6: Adds a second parity disk for increased resilience

# RAID COMPARISON

RAID Level Comparison

| Features | RAID 0 | RAID 1 | RAID 1E | RAID 5 | RAID 5EE | RAID 6 | RAID 10 |
|---|---|---|---|---|---|---|---|
| Minimum # Drives | 2 | 2 | 3 | 3 | 4 | 4 | 4 |
| Data Protection | No Protection | Single-drive failure | Single-drive failure | Single-drive failure | Single-drive failure | Two-drive failure | Up to one disk failure in each sub-array |
| Read Performance | High | High | High | High | High | High | High |
| Write Performance | High | Medium | Medium | Low | Low | Low | Medium |
| Read Performance (degraded) | N/A | Medium | High | Low | Low | Low | High |
| Write Performance (degraded) | N/A | High | High | Low | Low | Low | High |
| Capacity Utilization | 100% | 50% | 50% | 67% - 94% | 50% - 88% | 50% - 88% | 50% |
| Typical Applications | High end workstations, data logging, real-time rendering, very transitory data | Operating system, transaction databases | Operating system, transaction databases | Data warehousing, web serving, archiving | Data warehousing, web serving, archiving | Data archive, backup to disk, high availability solutions, servers with large capacity requirements | Fast databases, application servers |



# CH. 39: FILESYSTEMS

# OBJECTIVES – 6/4

- Questions from 6/2
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Assignment 3 – on Linux kernel programming – offered in "tutorial" format – due Sat June 13
- Quiz 4 – Page Tables – optional – provides practice problems – to be posted
- Chapter 37: Hard Disk Drives
- Chapter 38: RAID (Redundant array of inexpensive disks)- *very brief*
- Chapter 39/40: File Systems – *very brief*
- Practice Final Exam Questions – Today – 2nd hour



# FILE SYSTEMS

- Implemented by the OS as pure software
- Provide:
  - Data structures: to describe disk content
  - Arrays of blocks, index-nodes, trees
  - Access methods: provides mapping for OS calls open(), read(), write(), etc.
  - Which structures are read? written? For each call?
  - How efficiently does the structure support file operations?
- Many available file systems (A-Z)

# FILE SYSTEMS: A TO Z

- Numerous file systems abound (A-Z)
- ADFA, AdvFS, AFS, AFS, AosFS, AthFS, BFS, BFS, Btrfs, CFS, CMDFS, CP/M, DDFS, DTFS, DOS 3.x, EAFS, EDS, ext, etx2, etx3, ext4, ext3cow, FAT, VFAT, FATX, FFS, Fossil, Files-11, Felx, HFS, HPFS, HTFS, IceFS, ISO 9660, JFS, JXFS, Lisa FS, LFS, MFS, Minix FS, NILFS, NTFS, NetWare FS, OneFS, OFS, OS-9, PFS, ProDOS, Qnx5fs, Qnx6fs, ReFS, ReiserFS, Reiser4, Reliance, Reliance Nitro, RFS, S51K, SkyFS, SFS, Soup (Apple), SpadFS, STL, TRFS, Tux3, UDF, UFS, UFS2, VxFS, VLIR, WAFL, XFS, FS, ZFS



# FILE SYSTEM ORGANIZATION

- Disk is divided into blocks
- Block size supported by most HDDs is 512 bytes
- Typical FS block size is 4 KB
- An instance of a file system is typically called a **partition**
- A single physical disk can have multiple partitions (file systems)

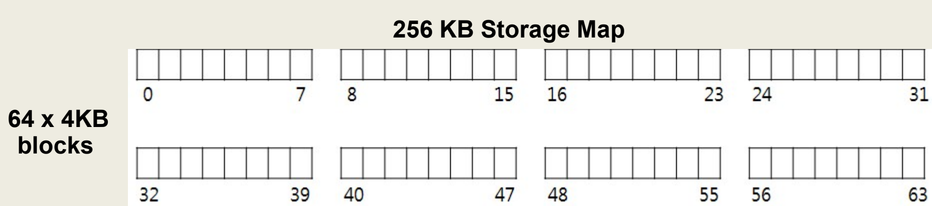
# FILE SYSTEM STORAGE

- File system is stored using blocks on the disk
- Location considered a "reserved" region of the disk
- Corruption of the reserved region can destroy the file tables causing data on the disk to by unaddressable
- File system keeps track of:
  - Which blocks comprise a file
  - Where the blocks reside (are they contiguous?)
  - The size of files
  - The owner of files
  - File permissions (e.g. R/W/X)



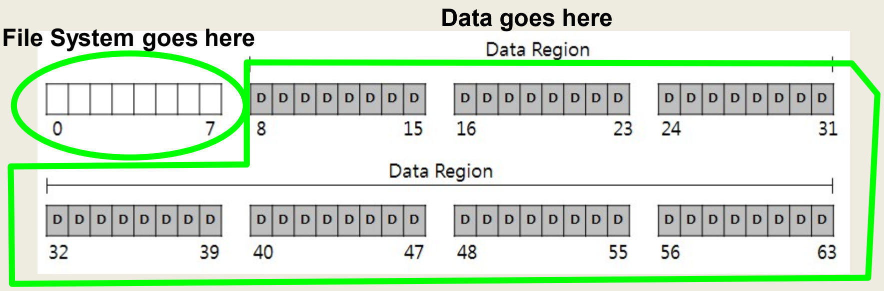
# FILE SYSTEM EXAMPLE

- Consider a 64 block disk (w/ 4096B block size) a.k.a. a 256 KB disk
- Legacy low density 5-¼" floppies had 160KB single side, 360KB double sided capacity →

# FILE SYSTEM LOCATION

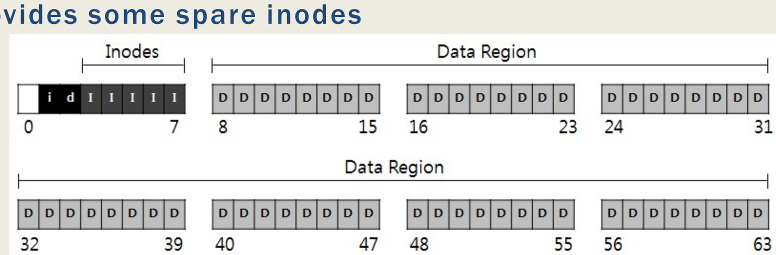
- Reserved region is at the front on a 64-block disk partition
- 8 x 4KB blocks for the file system
  - File sys contains: superblock, i-node bitmap, data bitmap, i-nodes
  - Each file is tracked using an “index-node” (inode)
- 56 x 4 KB blocks for data





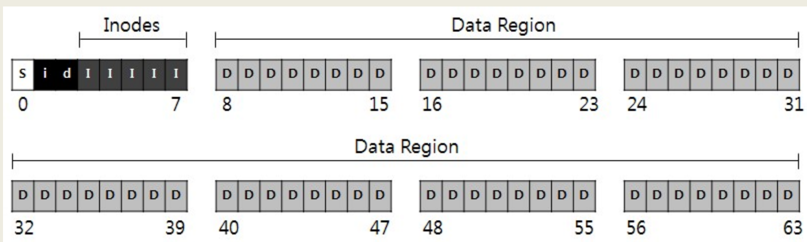

# FILE SYSTEM EXAMPLE - 2

- Consider 256kb disk, with 56 free data blocks
- i-node size 256 bytes each
- 4KB block can contain 16 inodes
- Need to track up to 56 files (1 for each 4kb data block)
- Need minimum of 4 x 4KB blocks required for inodes (64)
- EXAMPLE here reserves 5 x 4KB blocks for files
- Provides some spare inodes



# SUPERBLOCK

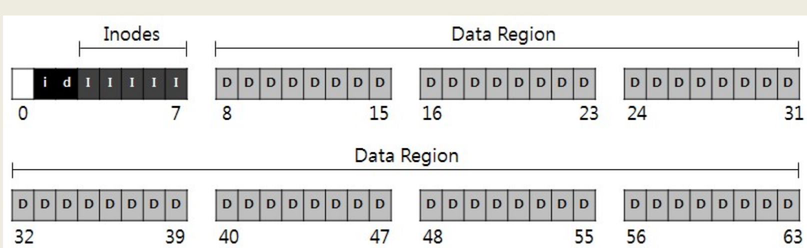
- First 4KB block of disk
- Contains information about the file system: “S” block below
  - How many inodes?
  - How many data blocks?
  - Location of inode table
  - File system identity code(s)




June 4, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L20.53

# FILE SYSTEM - FREE LIST

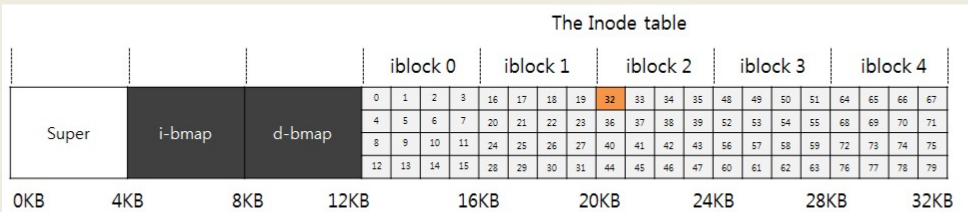
- File system maintains list of free inodes and data blocks
  - Example stores free list using bitmaps
  - Bitmaps: array of bits that track if inode or data block is in use (0/1)
  - Inode bitmap (“i” block below): 80 bits for inode table
  - Data bitmap (“d” block below): 56 bits for data blocks




June 4, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L20.54

# INODE EXAMPLE

- Every inode has an inode number (index value)
- Based on inode number, disk location can be calculated
- Example: inode number=32
- Offset into inode region = 32
- Size of inode=256 bytes
- Inode number x inode size = 8192
- Inode location = inode start addr + (inode no. x inode size)





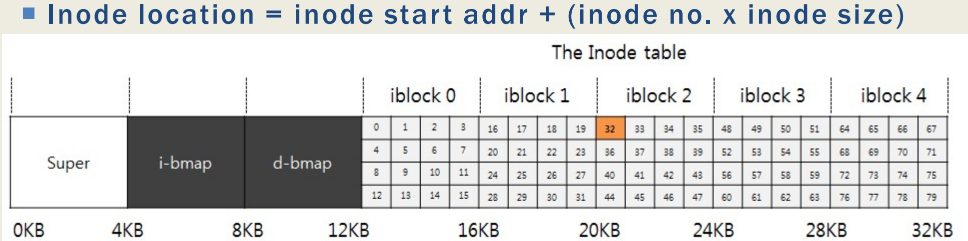

# INODE EXAMPLE

- Every inode has an inode number (index value)
- Based on i ulated
- Example: i
- Offset into
- Size of ino
- Inode num
- Inode location = inode start addr + (inode no. x inode size)

**What is the inode location?**

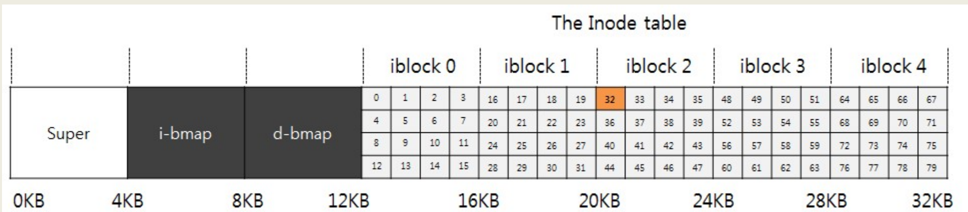
**12KB + (32 x 256)**

**12KB + 8KB = 20 KB**



# INODE EXAMPLE - 2

- Inodes store all information about a file:
- File type (e.g. directory, file, other)
- Size, and the number of blocks allocated to a file on disk
- R/W/X permissions
- Time information






# INODES – EXT2 LINUX FS

| Size | Name | What is this inode field for? |
|---|---|---|
| 2 | mode | can this file be read/written/executed? |
| 2 | uid | who owns this file? |
| 4 | size | how many bytes are in this file? |
| 4 | time | what time was this file last accessed? |
| 4 | ctime | what time was this file created? |
| 4 | mtime | what time was this file last modified? |
| 4 | dtime | what time was this inode deleted? |
| 4 | gid | which group does this file belong to? |
| 2 | links_count | how many hard links are there to this file? |
| 2 | blocks | how many blocks have been allocated to this file? |
| 4 | flags | how should ext2 use this inode? |
| 4 | osd1 | an OS-dependent field |
| 60 | block | a set of disk pointers (15 total) |
| 4 | generation | file version (used by NFS) |
| 4 | file_acl | a new permissions model beyond mode bits |
| 4 | dir_acl | called access control lists |
| 4 | faddr | an unsupported field |
| 12 | i_osd2 | another OS-dependent field |

# MULTI-LEVEL INDEXING

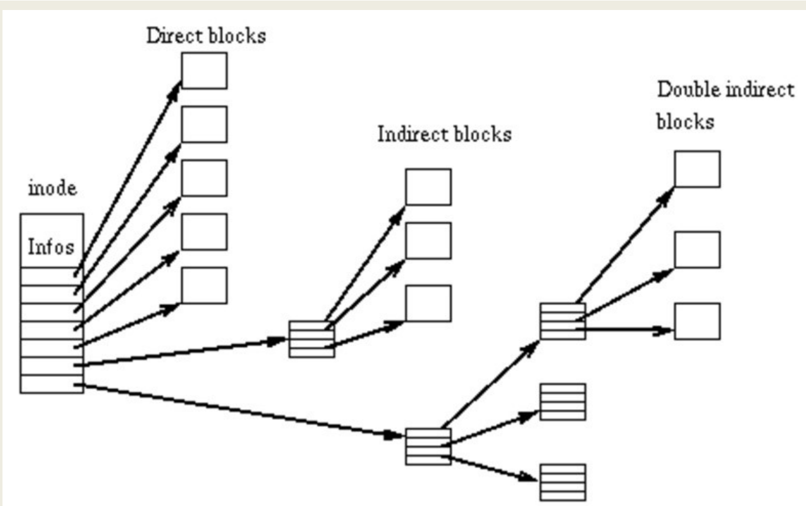
- Inodes use a multi-level index
- First level: includes 12-direct block pointers
- Second level: provides 1 indirect block pointer
- Points to a Block: contains 1,024 x 4 byte block pointers

- **Indirect pointer:**
- 12 direct block pointers, 1 indirect block ptr to a block
- Maximum file size:
- (12 + 1,024 entries) * 4KB = (1,036 x 4KB) = 4,144 KB

- Need more space?
  - Can have double and triple indirect block pointers



# MULTI-LEVEL INDEX - 2



# MULTI-LEVEL INDEX - 3

- **Double indirect pointer**
- First level: include 12-direct block pointers
- Second level: include 1 indirect block pointer
- Third level: include 1 indirect block pointer
- Maximum file size: *(entries x 4KB block size)*
- 12 + 1,024 + (1,024 x 1,024) * 4KB
- 1,049,612 x 4KB = 4,198,448 KB
  - ~ 4GB
- **Triple indirect pointer**
- Adds another level & indirect block pointer
- Maximum file size: *(entries x 4KB block size)*
- 12 + 1,024 + ($1024^2$) + ($1024^3$) * 4KB = ~ 4TB



# EXTENTS

- Extents have a pointer with a stored length
- Each file has multiple extents
- A single extent would require contiguous file allocation
- In contrast to block pointers:
- Extents conserve space better than multi-level indexes, but are less agile at representing file allocations scattered across the disk
  - Multi-level indexes excel for files w/ blocks scattered across the disk
  - Don't care if storage is contiguous because each block has a pointer
- File indexing presents a *space vs. flexibility* tradeoff
  - Extents (space efficient, rigid), multi-level indexes (better for tracking files w/ fragmented blocks)

# FILE INDEXING

- Multi-level indexing
  - ext2, ext3
  - *can you have a disk >4TB w/ ext3 ??*
- Extents
  - ext4 (default Ubuntu 16.04), XFS (default CentOS 7)
  - NTFS, Btrfs (b-tree fs)
- Exhaustive file systems feature comparison
  - https://en.wikipedia.org/wiki/Comparison_of_file_systems

| June 4, 2020 | TCSS422: Operating Systems [Spring 2020]<br>School of Engineering and Technology, University of Washington - Tacoma | L20.63 |
|---|---|---|

# TCSS 422 WILL RETURN AT ~2:35PM

June 4, 2020 TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L15.64

# OBJECTIVES – 6/4

- Questions from 6/2
- Tutorial 2 (pthreads, locks, conditions) – due Thurs June 4
- Assignment 3 – on Linux kernel programming – offered in "tutorial" format – due Sat June 13
- Quiz 4 – Page Tables – optional – provides practice problems – to be posted
- Chapter 37: Hard Disk Drives
- Chapter 38: RAID (Redundant array of inexpensive disks)- *very brief*
- Chapter 39/40: File Systems – *very brief*
- Practice Final Exam Questions – Today – 2nd hour



# QUESTION 1 – BASE AND BOUNDS

- A computer system uses a simple base/bounds register pair to virtualize address spaces. For each traces fill in the missing values of virtual addresses, physical addresses, base, and/or bounds registers. In some cases, it is not possible to provide an exact value. If so, specify a range (e.g. greater than 100), or value that is not a single number.

**Scenario 1**

| Virtual Address | Physical Address | | |
|---|---|---|---|
| 100 | 600 | | |
| 300 | 800 | Base? | ________ |
| 699 | 1199 | | |
| 700 | [fault] | Bounds? | ________ |

# Q1 - 2

**Scenario 2**

| Virtual Address | Physical Address | | |
|---|---|---|---|
| 300 | 1500 | Base? | ____________ |
| 1600 | 2800 | | |
| 1801 | __________? | Bounds? | ____________ |
| 2801 | 4001 | | |

**Scenario 3**

| Virtual Address | Physical Address | | |
|---|---|---|---|
| ____________ | 1000 | Base? | 1000________ |
| ____________ | 1100 | | |
| ____________ | 2999 | Bounds? | 2000________ |
| ____________ | [fault] | | |



# QUESTION 2 – SINGLE-LEVEL PAGE TABLE

- Consider a computer with 4 GB ($2^{32}$) of physical memory, where the page size is 4 KB ($2^{12}$). For simplicity assume than 1GB=1000MB, 1MB=1000KB, 1KB=1000 bytes
- (a) How many pages must be tracked by a single-level page if the computer has 4GB ($2^{32}$) of physical memory and the page table size is 4 KB ($2^{12}$)?
- (b) How many bits are required for the virtual page number (VPN) to address any page within this 4GB ($2^{32}$) memory space?
- (c) Assuming that the smallest addressable unit of memory within a page is a byte (8-bits), how many bits are required for the offset to refer to any byte in the 4 KB page?
- (d) Assuming each page table entry (PTE) requires 4 bytes of memory, how much memory is required to store the page table for one process (in MB)?

## Q2 - 2

- (e) Using this memory requirement, how many processes would fill the memory with page table data on a 4GB computer?



## QUESTION 3 - TWO-LEVEL PAGE TABLE

- Consider a computer with 1 GB ($2^{30}$) of physical memory, where the page size is 1024 bytes (1KB) ($2^{10}$). We would like to index memory pages using a two level page table consisting of a page directory which refers to page tables which are created on demand to index the entire memory space.
- For simplicity assume than 1GB=1000MB, 1MB=1000KB, 1KB=1000 bytes
- (a) For a two-level page table, divide the VPN in half. How many bits are required for the page directory index (PDI) in a two-level scheme?
- (b) How many bits are required for the page table index (PTI)?
- (c) How many bits are required for an offset to address any byte in the 1 KB page?

# Q3 - 2

- (d) Assuming each page table entry (PTE) requires 4 bytes of memory, how many extra bits are available for status bits?
- (e) HelloWorld.c consists of 4 memory pages. One code page, one heap page, one data segment page, and one stack segment page. How large is the two-level page table in bytes with the structure described above that could index the all 4 memory pages of HelloWorld.c?
  *Hint: There should be 2 tables, a page directory, and a page table.*
- *(f) Assuming the same page table as for HelloWorld.c, using the exact same two-level page table, how large in bytes could the program grow to before needing to expand the page table?*



# QUESTION 4 – CACHE TRACING

- Consider a 3-element cache with the cache arrival sequences below.
- Determine the number of cache hits and cache misses using each of the following cache replacement policies:

**A. Optimal policy**

Arrival sequence:

5 3 7 5 3 1 0 7 1 6 4 3 2 1 3

<u>Working Cache</u>
Cache 1:

Cache 2:

Cache 3:

# Hits: ______________

# Misses: ______________

# Q4 - 2

**B. FIFO policy**

Arrival sequence:

5 3 7 5 3 1 0 7 1 6 4 3 2 1 3

Working Cache
Cache 1:

Cache 2:

Cache 3:

# Hits: _______________

# Misses: _______________

**C. LRU policy**

Arrival sequence:

5 3 7 5 3 1 0 7 1 6 4 3 2 1 3

Working Cache
Cache 1:

Cache 2:

Cache 3:

# Hits: _______________

# Misses: _______________



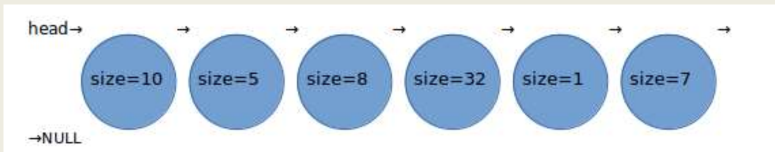
# QUESTION 5 – FREE SPACE MANAGEMENT

- Free space management involves capturing a description of the computer's free memory using a data structure, storing this data structure in memory, and OS support to rapidly use this structure to determine an appropriate location for new memory allocations. An efficient implementation is very important when scaling up the number of operations the OS is required to perform.
- Consider the use of a linked list for a free space list where each node is represented by placing the following structure in the header of the memory chunk:

```
typedef struct __node_t
{
    int size;
    struct __node_t *next;
} node_t;
```

# Q5 - 2

- Consider the following free space list:




- (a) Consider the **next fit** allocation strategy. For this free list above, how many comparison operations must be performed to identify a free chunk of **30-bytes** ?
- (b) After the last free space identification, the chunk is split and the remaining free space is returned to the free space list. Now, consider the **next fit** allocation strategy. After finding a free space for the previous request, how many comparisons are required to identify a free chunk of 10-bytes?



# Q5 - 3

- Now, after the last free space identification the chunk is split and the remaining free space is returned to the free space list. Now consider each of the following free space allocation strategies. How many comparisons are required on the updated free space list to find a free chunk of 2 bytes using:
- (c) best fit?
- (d) worst fit?
- (e) first fit?

# QUESTIONS

# EXTRA SLIDES

# COMMON FILE CHARACTERISTICS

| | |
|---|---|
| **Most files are small** | Roughly 2K is the most common size |
| **Average file size is growing** | Almost 200K is the average |
| **Most bytes are stored in large files** | A few big files use most of the space |
| **File systems contains lots of files** | Almost 100K on average |
| **File systems are roughly half full** | Even as disks grow, file system remain -50% full |
| **Directories are typically small** | Many have few entries; most have 20 or fewer |

**File System Measurement Summary**



# DIRECTORIES

- Directory contains file name and i number (index)
- Extra files for the **parent dir** and **pwd**
- Can store dirs as linear list, often stored in inodes
- XFS uses B-trees to eliminate sequential search of filenames for duplicates when creating a new file

| inum | reclen | strlen | name |
|---|---|---|---|
| 5 | 4 | 2 | . |
| 2 | 4 | 3 | .. |
| 12 | 4 | 4 | foo |
| 13 | 4 | 4 | bar |
| 24 | 8 | 7 | foobar |

**on-disk for dir**

# FILE I/O - READ

- Consider reading a file called "/foo/bar"
- Traverse starting at root "/" (inumber = 2) to find file
- Read each inode to dereference file block location on disk



# FILE I/O – READ OPERATIONS

- 3 block file: 11 reads, 3 writes (last access time)

| | 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 | | | | | |

**File Read Timeline (Time Increasing Downward)**

# FILE I/O - WRITE

- At least Five I/Os to update an existing file
  - one to read the data bitmap
  - one to write the bitmap (to reflect its new state to disk)
  - two more to read and then write the inode
  - one to write the actual block itself.



# FILE I/O – WRITE - 2

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[2] |
|---|---|---|---|---|---|---|---|---|---|---|
| create (/foo/bar) | | read<br>write | read | read<br>write | read<br>write | read | read<br>write | | | |
| write() | read<br>write | | | | read<br>write | | | write | | |
| write() | read<br>write | | | | read<br>write | | | | write | |
| write() | read<br>write | | | | read<br>write | | | | | write |

**File Creation Timeline (Time Increasing Downward)**

# FREE SPACE MANAGEMENT

- Free Lists
  - Linked list of free blocks
  - Head node tracks first free block, each subsequent block is linked with a pointer
  - Bitmaps
    - Bit-wise arrays of free blocks
  - B-trees (XFS)
    - Represents free list in a more compact form, with better search performance
- Free list design impacts efficiency of finding free blocks



# CACHING READS AND WRITES

- Two approaches to cache allocation
- Static partitioning
  - Allocate a fixed size cache at system boot time
  - For example: dedicate 10% of memory for disk R/W cache
- Dynamic partitioning
  - Linux has a unified page cache
    - Pages are cached to a unified page cache for multiple purposes
    - Memory virtualization pages
    - Inodes, disk pages

# FILE CACHING

- Subsequent file opens to a cache file can eliminate reads
- Benefits of write caching
  - Batch updates together to reduce HDD requests
  - Writes can be scheduled intelligently in the future
  - Some writes can be avoided altogether
  - For example: short lived tmp files
- Typical write buffering is from 5 to 30 seconds
- Risk of data loss
  - Fsync(): force synchronization to disk
  - Some apps such as database use to ensure immediate writes



WJL1

# WILL RETURN IN A FEW MINUTES

**Slide 88**

---

**WJL1** Wes J. Lloyd, 5/30/2020