EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

Recap: Accounting for storage device characteristics

- How to account for device heterogeneity?
  - Use device drivers to offer abstraction of sequence of disk blocks
- How to account for slow seek times?
  - Caching
  - Scheduling of I/O requests
  - Store related items together on disk

File systems

- File system: a data structure which ensures that data persists across …
  - Power outages
  - Machine crashes/reboots
  - Process creation/exit
- How to enable persistence across these events?
  - Use persistent storage medium
  - Write data carefully
  - Avoid use of addresses that change across processes

Interface to file system

- Create file
- Delete file
- Read <file, offset>
- Write <file, offset>
- Other (e.g., list files in a directory)

- Alternate interface: SQL → Database

File system workloads

- Optimize data structure for the common case

- Some general rules of thumb
  - Most file accesses are reads

  - Most programs access files sequentially and entirely

  - Most files are small, but most bytes belong to large files

File abstraction

- Reality: One (or a few) disks to store data
  - Each is an array of (logical) blocks
- Abstraction: Numerous storage objects (files)
  - Each is an array of bytes
- Challenges:
  - How to name files?
  - How to find and organize files?

How to store a file?

- Need to store metadata
  - File size
  - Owner/Permissions
  - Time of creation/last access
- Need to store pointer to data
  - Pointer must be independent of process
  - Use logical block number to point to data on disk
- Store a file header
  - inode in Unix, Master File Table record in NTFS
  - Structure that describes file and allows you to find data

Contiguous allocation

- File = array of blocks ("extent")
  - Reserve space in advance
  - If file grows, move it to a larger free area
  - File header contains starting location of file and size

- Pros and cons?

  + Fast sequential access

  + Easy random access

  - Wastes space; external fragmentation

  - Difficult to grow file

Indexed files

- File = array of block pointers
  - Just like page table

File block #	Disk block #
0	18
1	50
2	8
3	15

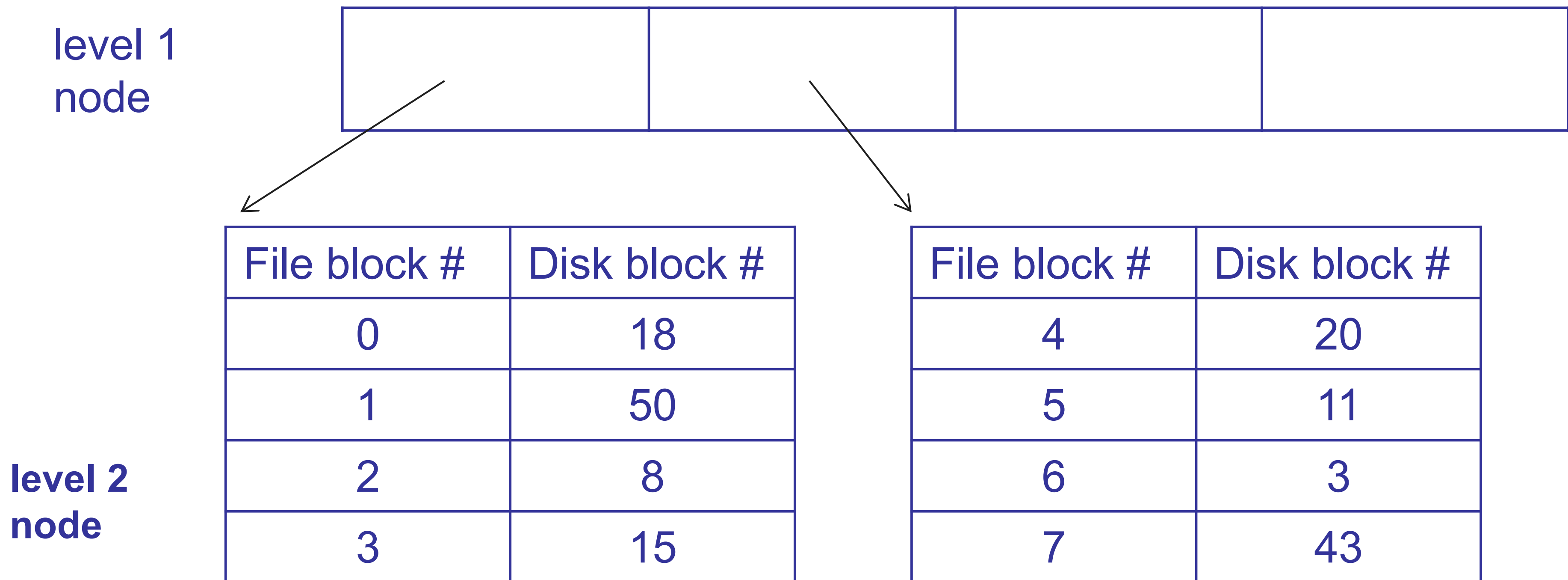
- Pros and cons?
  - + Easy to grow file
  - + Easy random access
  - - But potentially slow for sequential access
- How to speed up sequential access?
  - To grow file, allocate new block close to previous block
- Consequence of allowing for large files?
  - Trade-off between page table size and block size

Project 3

- Think of questions regarding project
- Pose your questions to a neighbor not in your group

Multi-level indexed files

- File = tree of block pointers



File block #	Disk block #
0	18
1	50
2	8
3	15

File block #	Disk block #
4	20
5	11
6	3
7	43

- Pros?
  - Files can easily grow
  - Allows large file, but small files don't waste header space

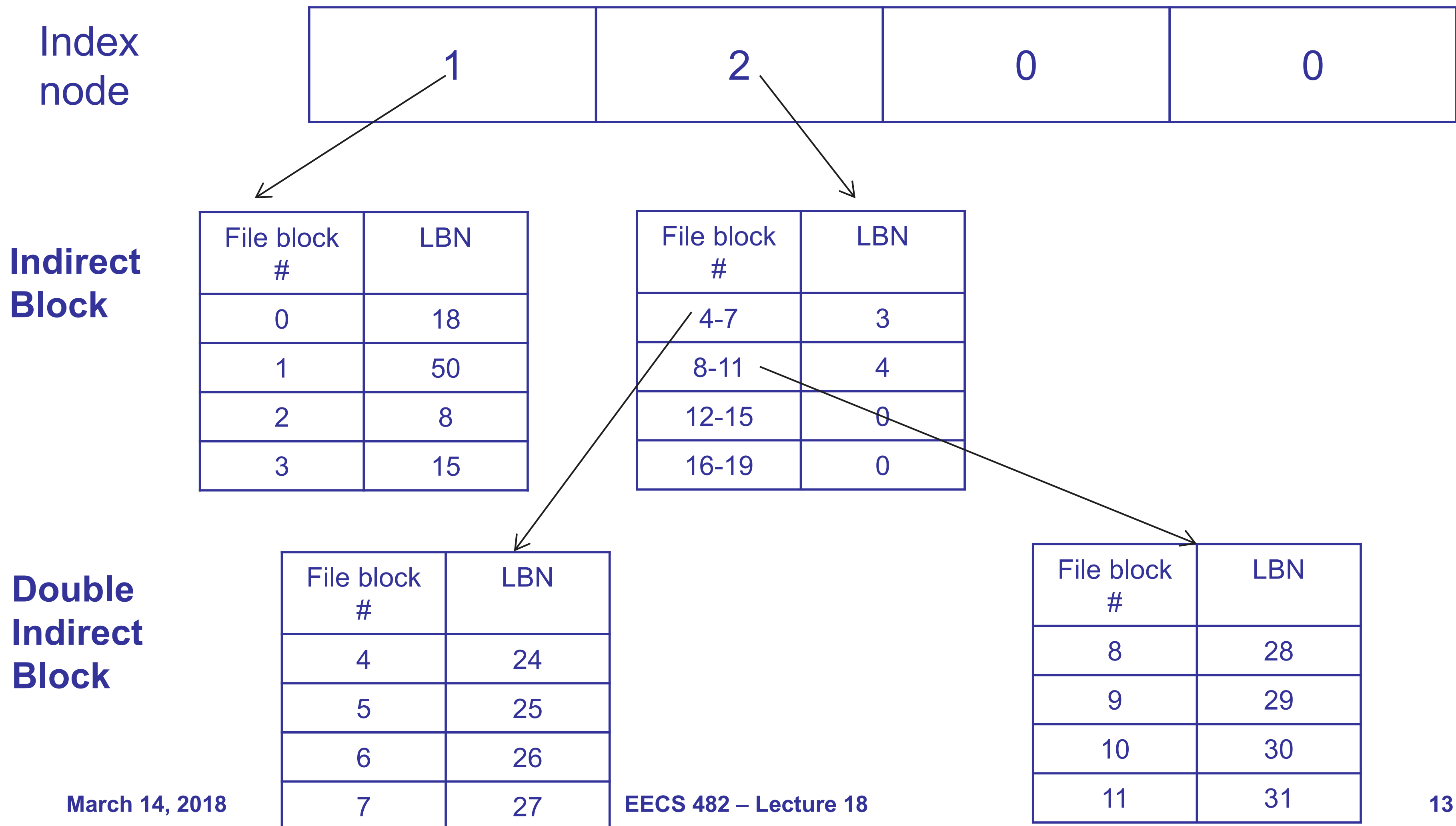
Multi-level indexed files

- Downsides?
  - Could have lots of seeks for sequential access

  → Bad performance, especially for large files

- How to fix?
  - Caching
  - Non-uniform depth

Non-Uniform depth



Representing files

- Can have other dynamic ways of allocating file
  - Must ensure that location of file header does not change as file grows

- Example: Header is head of linked list
  - \+ Easy to append
  - \- Slow sequential access
  - \- Really slow random access

Naming and directories

- How to specify file to be accessed?
  - File name, click on icon, or describe contents
- File name is usually hierarchical
  - E.g., /home/harshavm/482/notes
  - Allows users to group related files into one folder
  - Allows easy searching, e.g., “ls /home/harshavm/482”
- Must translate file name to disk block # of header
  - What data structure to use to store mapping?
  - Tree of directories
  - Why not a hash table?

Directories

- Directory: mapping information for a set of files
  - Name of file → file header's disk block # for that file
  - Once, array of (name, file header's disk block #) entries
  - Modern file systems: hash table or B-tree

- Directories and files are largely equivalent
  - Same storage structure
  - Directory entry points to inode for file or directory

Directory Example

/ directory

Name	LBN
“home”	100
“usr”	35
“tmp”	43
“foo.txt”	254

home directory

Name	LBN
“harshavm”	23
“barisk”	99
	0
	0

harshavm directory

Name	LBN
“482.txt”	44
	0
“src”	55
“foo.txt”	33

- Any differences in allowing application to update file versus directory?

Example: /home/harshavm/482/notes

1. Read file header for / (root directory)
   - Contains pointers to data blocks of / directory
2. Read data blocks of /
   - Contains list of the files and directories in /. For each entry, contains mapping from name → header's disk block #
   - One of those entries is "home"
3. Read file header for /home
4. Read data blocks for /home
5. Read file header for /home/harshavm
6. Read data blocks for /home/harshavm
7. Read file header for /home/harshavm/482
8. Read data blocks for /home/harshavm/482
9. Read file header for /home/harshavm/482/notes
10. Read first data block for /home/harshavm/482/notes

Eliminated by caching file header for current working directory