FILE SYSTEMS

CS124 – Operating Systems

Spring 2024, Lecture 21

Persistent Storage

- All programs require some form of persistent storage that lasts beyond the lifetime of an individual process
  - Most obvious reason: need a way to store programs!
  - Other examples: configuration, input data files, output data files, documents, files shared among programs, etc.
- Also need storage to last through a process- or system-crash
  - e.g. banking information, reservation systems, document editors
  - Or, just for backing up system information
- Computers often include some form of persistent storage
  - Very common: solid-state drives (SSDs), hard disks (HDDs), USB flash drives, flash memory cards, other flash storage
  - Less common, but still widely used: tape drives, optical drives

Large Data-Sets

- Sometimes programs must manipulate data-sets that are much larger than the system memory size
  - i.e. the size of physical memory, or system's virtual address space
- Frequently want to allow multiple processes to access and manipulate the same large data-set concurrently
  - Allow multiple processes to read the same data concurrently
  - Allow multiple processes to write different parts of the same data concurrently
- Typically, devices used to store these large data-sets are much slower than computer's main memory
  - Reading many bytes is roughly as costly as reading one byte…
  - Divide storage device into fixed-size blocks; allow system software to read/write individual blocks of data

Managing Persistent Storage

- User applications usually don’t want to deal with reading/writing blocks of data
  - …especially when different devices have different block sizes, different read/write/erase characteristics, etc.
- Similarly, applications usually don’t want to deal with:
  - Remembering unusable areas of storage device (e.g. bad blocks)
  - Remembering where various data-sets start on the device
  - Data-sets that may not be stored contiguously on the device
- Operating systems present a **file** abstraction to programs
- Files are logical units of information created by processes
  - A contiguous linear sequence of bytes accessed via a relative offset from the start of the file
- The OS’ **file system** manages this file abstraction

File Contents

- General-purpose operating systems usually don't constrain file contents to follow any particular format
  - A process can interpret the file's contents however it wants to
  - OS may distinguish between text and binary files, but usually apps impose this distinction
- Some files may be constrained to follow a specific format, e.g. executable program binaries or shared libraries
- Older systems would have constraints on file formats, due to the characteristics of their storage devices!
  - e.g. card readers could only store 80-character-wide text files, and output was written to 132-character printers
- Purpose-built operating systems may also constrain files to follow a specific format (files of records, images, etc.)

Referring to Files

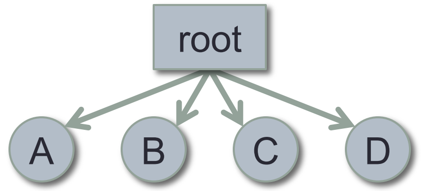
- Files are referred to by a text **name** (a.k.a. “filename”)
- Often, files also specify an **extension** indicating the kind of file, i.e. how to interpret the file’s contents
  - The extension is usually the portion of the filename following last period “.” in the filename
  - The portion before the extension is often called the base name
  - Term “filename” may refer to base name, or name and extension
- The specific constraints on filenames vary from OS to OS
  - e.g. MS-DOS only allows 8 characters for the filename and 3 characters for the extension
  - Some OSes respect the capitalization of a filename; e.g. most UNIXes treat **`FOO.txt`** and **`foo.txt`** as different files
  - Others treat these as referring to the same file (e.g. MS-DOS, and macOS by default)

Organizing Files

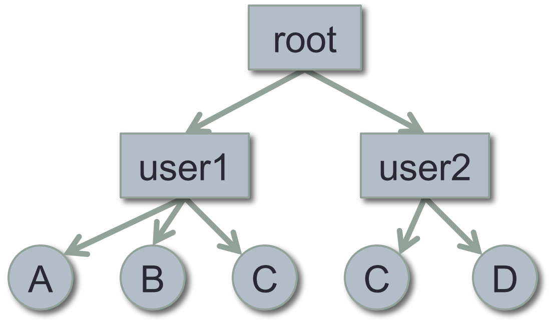
- Once a system has more than a few files, it becomes very helpful to be able to organize them
- Operating systems provide **directories** or **folders** that contain files
  - Within a specific directory, every file must have a unique name
  - Two different directories can contain files with the same name
- File systems usually have at least one directory: the **root directory**
  - The starting point for finding any file in the file system
  - If the file system doesn't have a root directory, programs must know how to find files on the storage device by themselves
- Depending on the complexity of the file system, OSes will support varying levels of complexity for directory structure

Directory Structures

- Simple operating systems frequently support a very simple directory structure
  - e.g. simple phones, digital cameras, …
- A **single-level directory** maintains all files in a single root directory
  - Used when the system won't have many files to manage
- Files within a directory must be uniquely named…

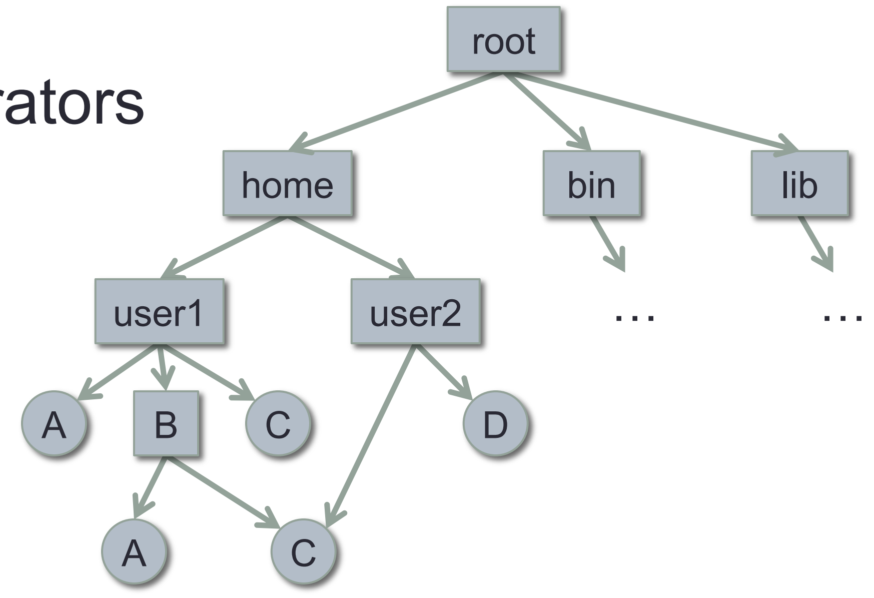


- Multiuser systems need at least a **two-level directory** structure, so that each user can have their own files
  - First level is called a **master file directory**
  - Second level is called a **user file directory**



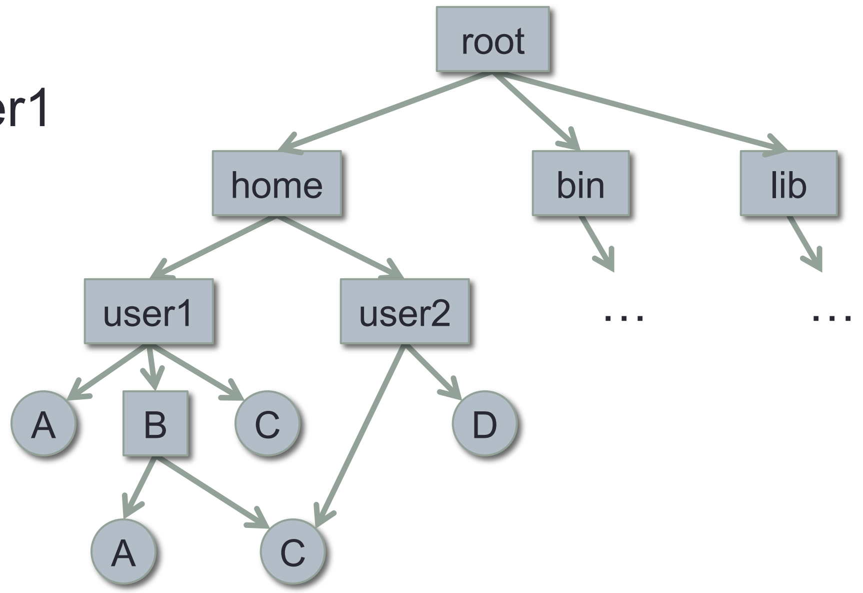
Directory Structures (2)

- General-purpose operating systems frequently support a directory structure that forms a graph
  - Top level directory is still the root
  - Top-level subdirectories group files based on their purpose
- Files are referenced by specifying the path to the file from the root directory
  - e.g. /root/home/user2/D
- Different operating systems use various path separators
  - Windows uses “\”
  - UNIX variants use “/”
  - MULTICS used “>”
  - Older MacOS used “:”, MacOS X (a.k.a. macOS) switched to “/”



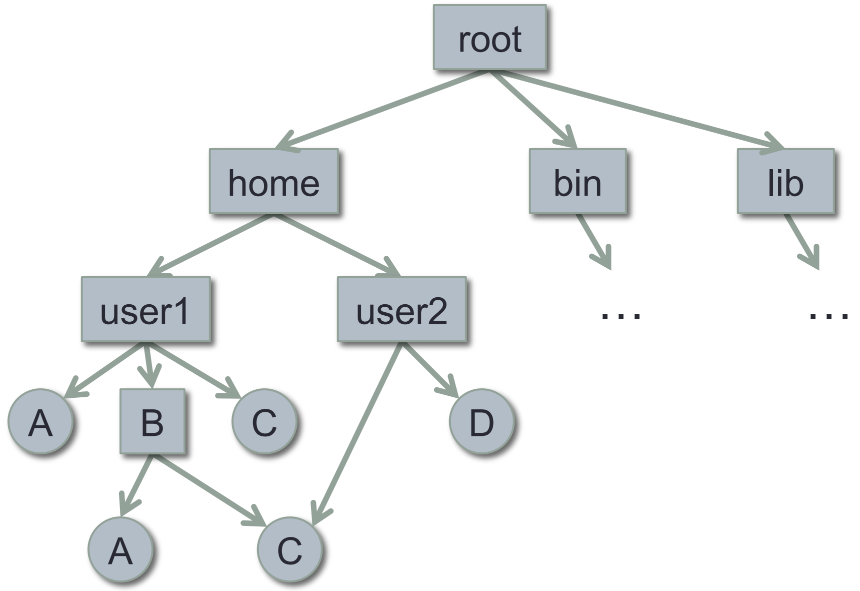
Directory Structures (3)

- Every process has a "current directory"
  - e.g. when a user logs in to the system, their current directory is their home directory
  - e.g. when user1 logs in, their shell's current directory is /home/user1
- Relative paths are resolved using the current directory
  - Can refer to current directory with ".", or parent with ".."
- Allows e.g. users to share files
  - Example: user1 is logged in, current directory is /home/user1
  - user1 can access user2's file D with path "../user2/D"



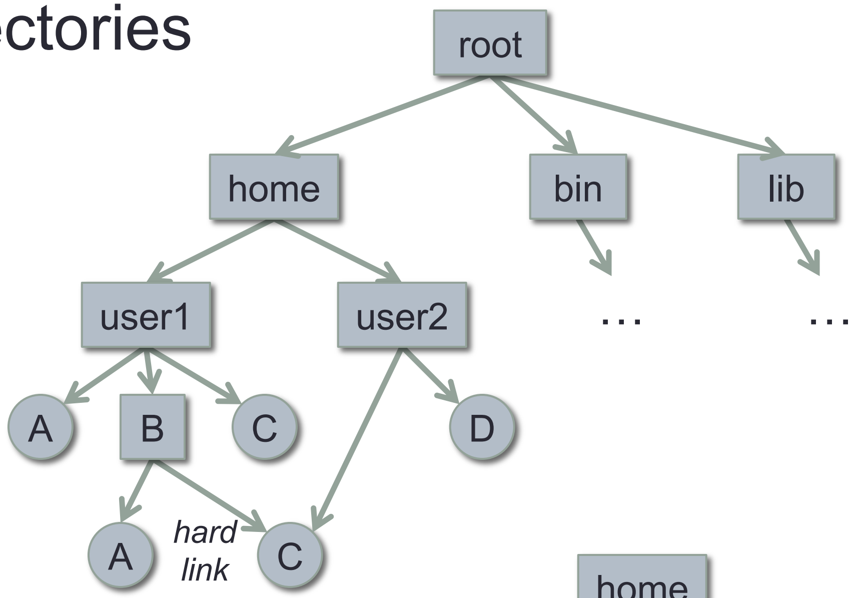
Directory Structures (4)

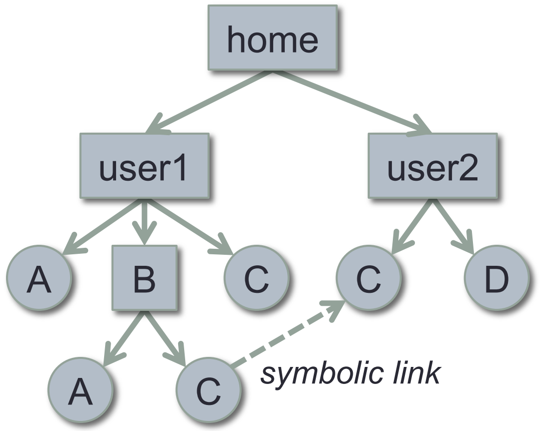
- Don't always want to share files with other users…
- Similarly, don't want system files being deleted or edited by just anybody
- Files include **metadata** that holds additional details about the file
- A small list of examples:
  - The user that created the file
  - When the file was created or last modified
  - Access permissions for the file
  - The icon associated with the file
  - The application used to open the file



Directory Structures (5)

- Often, files can be shared between different directories
- A **link** is a "pointer" to a file that resides in another directory
- **Hard links** are when the directory entries themselves reference a specific file
  - OS uses a reference-count to tell when a file is no longer used, and the space may be reclaimed
- **Symbolic links** are specified as a file in a different directory
- Not all file systems support hard links
  - e.g. FAT file system doesn't support hard links





File Storage

- Files are presented as an unstructured sequence of bytes
  - Programs are free to read and write sequences of varying sizes
  - Programs are free to impose whatever meaning on a file's contents
- The file system exposes various operations on files, e.g.
  - Create a file at a specific path (can specify permissions, etc.)
  - Delete a file at a specific path
  - Rename a file
- The OS maintains a "current position" for open files
  - When bytes are read or written, the current position is used
  - The position is updated by the read or write operation
  - Programs can also seek to a specific position within a file
- If multiple processes have a given file open, each process has its own "current position" in the file

File Access Patterns

- Programs exhibit two major access patterns when interacting with files
- **Sequential access** is when a file's contents are read in order, block by block
- **Direct access** (or **relative access**) is when a program seeks to the location of a specific piece of data
  - e.g. to read or write that piece of data
  - A program may seek relative to the current position, or relative to the start of the file, or relative to the end of the file
- Different filesystem layouts have different strengths
  - Some are great for sequential access, e.g. because they reduce disk seeks and other access overhead
  - Some are terrible for direct access, e.g. because they don't provide an easy way to map a logical position to a block of storage

File Layout: Contiguous Allocation

- Most persistent storage devices are large – can hold many files at once
- Storage devices are also accessed by blocks
- The file system must keep track of which blocks hold the contents of each file, and the order of blocks in the file
- Easiest approach for file layout: **contiguous allocation**
  - Each file occupies a contiguous region of the storage device
- Directory structure is very simple: each entry must simply record where file starts, and how many blocks in the file
- Indexing into the file is also easy:
  - To find block corresponding to the current file position, divide file position by block size, then add in the starting block number

File Layout: Contiguous Allocation (2)

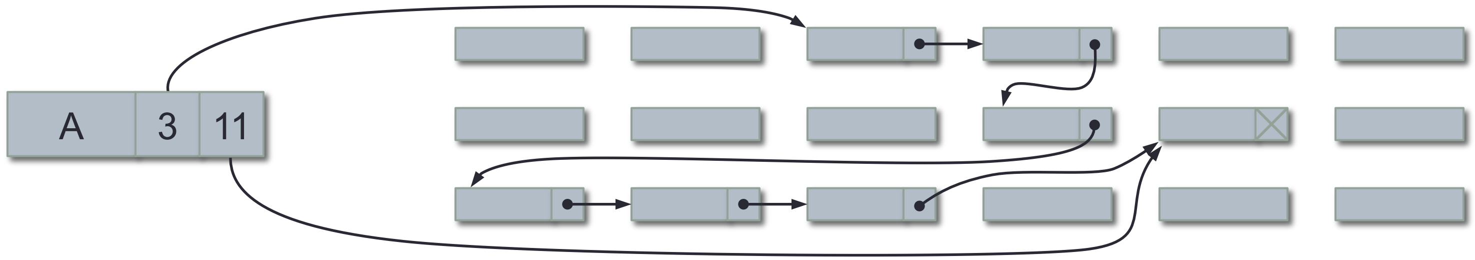
- Contiguous allocation suffers from external fragmentation
  - After many file creations and deletions, disk space becomes fragmented
- Can **compact** the free space on the device by copying all files to another device, then copying them back
  - Same technique as relocation register and segmentation approaches to virtual memory
- Frequently, the device must be taken offline before it can be compacted
  - Can’t allow programs to access/manipulate the device’s contents while it’s being compacted
- Another major issue with contiguous allocation: programs must often extend the size of a given file
  - e.g. write results to a data file, or messages to a log file

File Layout: Contiguous Allocation (3)

- Contiguous allocation can be modified to provide **extents**: a contiguous region of space on the storage device
  - An extent is usually comprised of many blocks
- A file consists of one or more extents chained together
- Reduces issues of external fragmentation since a single file can occupy multiple regions of the disk
  - But, external fragmentation still become a serious issue over time
- Can also suffer from **data fragmentation**: a file is broken into many parts and spread all over the storage device
- CDs, DVDs and tapes all use contiguous allocation
- Many file systems support extents: NTFS (Windows), HFS/HFS+ (older Mac), APFS (current Mac), ext4, btrfs, etc.
  - Some of these require extents to be enabled before they are used

File Layout: Linked Allocation

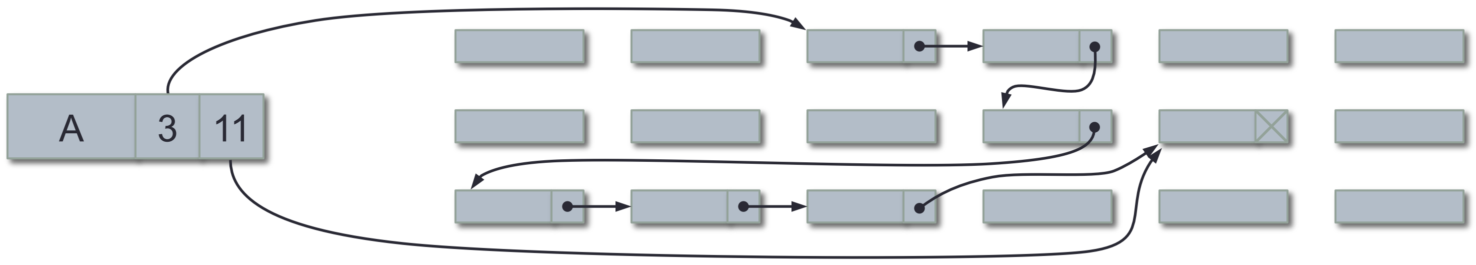
- In **linked allocation**, files are comprised of a sequence of blocks that are linked together
- Directory entries point to first and last block in each file
- Each block stores a pointer to the next block in the file



- This approach is really only good for sequential access
- Can’t easily find which block of a file corresponds to a given logical position within the file
  - Must read through file’s blocks to identify the block corresponding to a given position

File Layout: Linked Allocation (2)

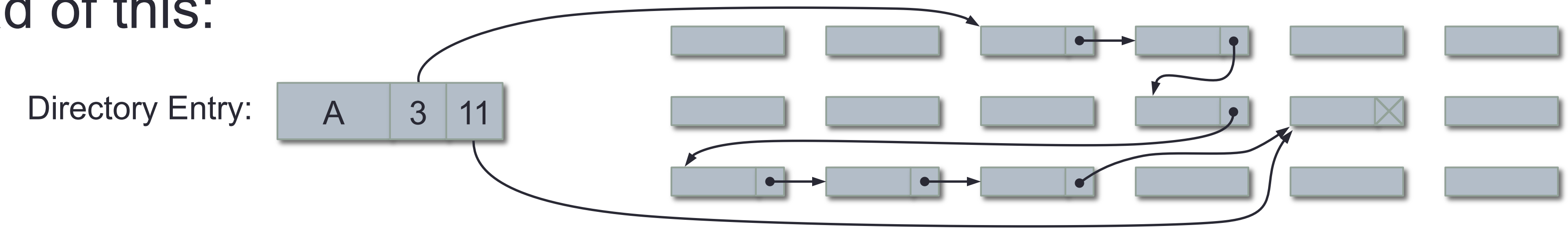
- Compaction isn't necessary because storage is always allocated in units of blocks…



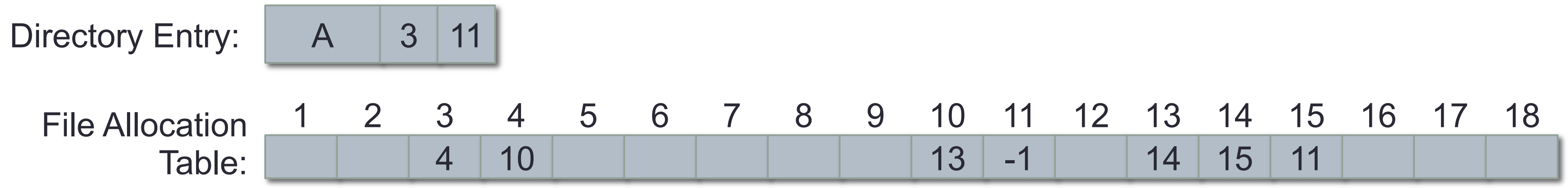
- …but internal fragmentation becomes an issue, especially for files that are much smaller than the block size
- Similarly, data fragmentation can be a very serious issue
- A small amount of space is lost within each block due to a "next block" pointer
  - Blocks are usually a power of 2 in size; programmers like to work with buffers that are a power of 2 in size (for best cache usage)
  - Can easily have reads that inadvertently span multiple blocks

File Layout: Linked Allocation (3)

- Instead of storing the sequence of blocks in the blocks, move this into a separate **file-allocation table** (FAT)
  - A part of the file system is specifically devoted to storing the FAT
- Instead of this:



- Record the block sequence in a separate table elsewhere on the disk
  - Each block in the file is wholly used for storing the file's data

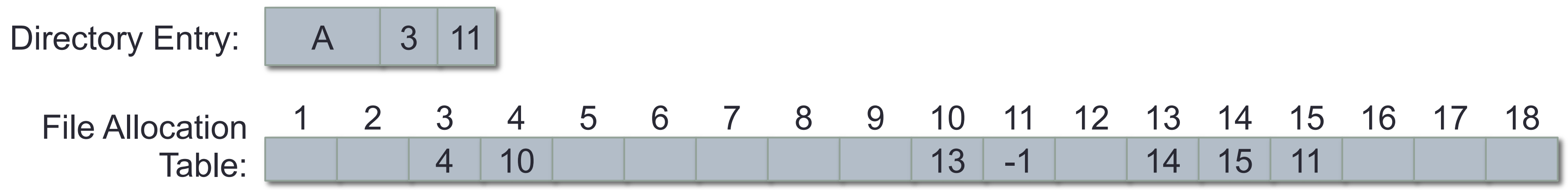


Directory Entry:	A	3	11

File Allocation Table:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
			4	10						13	-1		14	15	11			

File Layout: Linked Allocation (4)

- The file-allocation table tends to be a limited, fixed size
  - Can load the entire FAT into memory
  - Makes it faster to identify the block corresponding to a specific logical offset within a file



Directory Entry:

A	3	11

File Allocation Table:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
		4	10						13	-1		14	15	11			

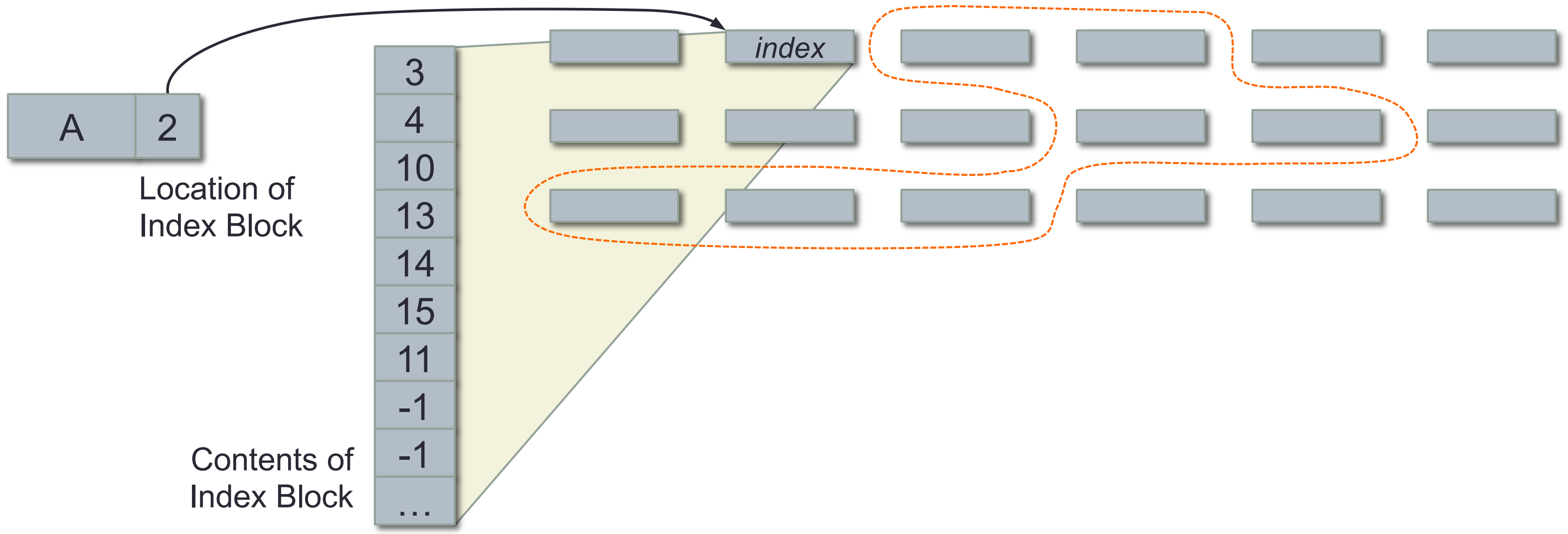
- As storage devices grow in size, run into two problems
- Problem 1: Sometimes, the set of FAT entries can't address the number of blocks the device actually has
- Example: Original FAT system had 8 bits per table entry
  - Only 256 blocks can participate in files!
  - Subsequent FAT formats devoted more bits to each table entry, e.g. FAT16 has 16 bits per entry, FAT32 has 32 bits per entry

File Layout: Linked Allocation (5)

- To solve problem of large disks using FAT file systems, files are allocated in **clusters**, not in blocks
  - Every cluster contains a fixed number of blocks, e.g. one cluster might be 16 blocks
- Causes problem 2: FAT file systems have severe internal fragmentation issues storing small files on large devices
- Clusters can be as large as 32KiB or even 64KiB in size
- Example: a disk with sectors that are 512 bytes in size
  - A cluster is 32 sectors, or 16KiB
  - The FAT filesystem can only hand out space in 16KiB chunks!
- If a 3KiB file is created, 13KiB of space is wasted
- If a 100-byte file created, nearly the entire cluster is empty

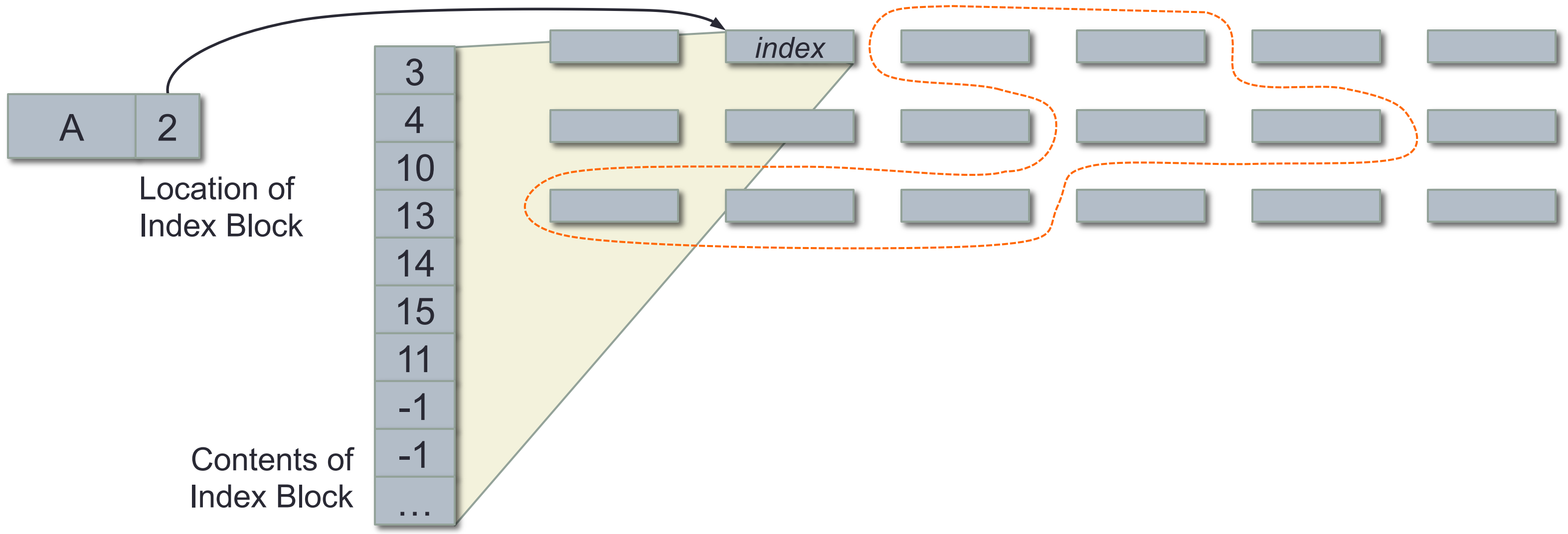
File Layout: Indexed Allocation

- **Indexed allocation** achieves the benefits of linked allocation while also being very fast for direct access
- Files include indexing information to allow for fast access
    - Each file effectively has its own file-allocation table optimized for both sequential and direct access
    - This information is usually stored separate from the file's contents, so that programs can assume that blocks are entirely used by data



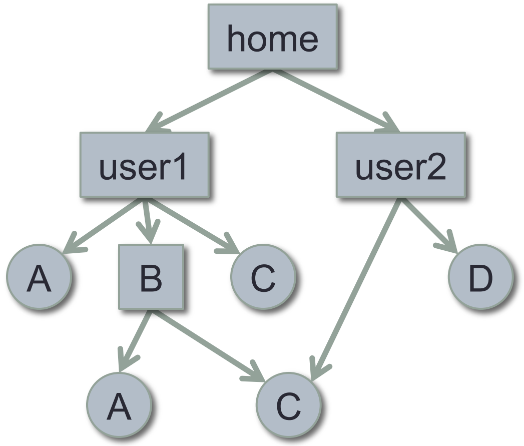
File Layout: Indexed Allocation (2)

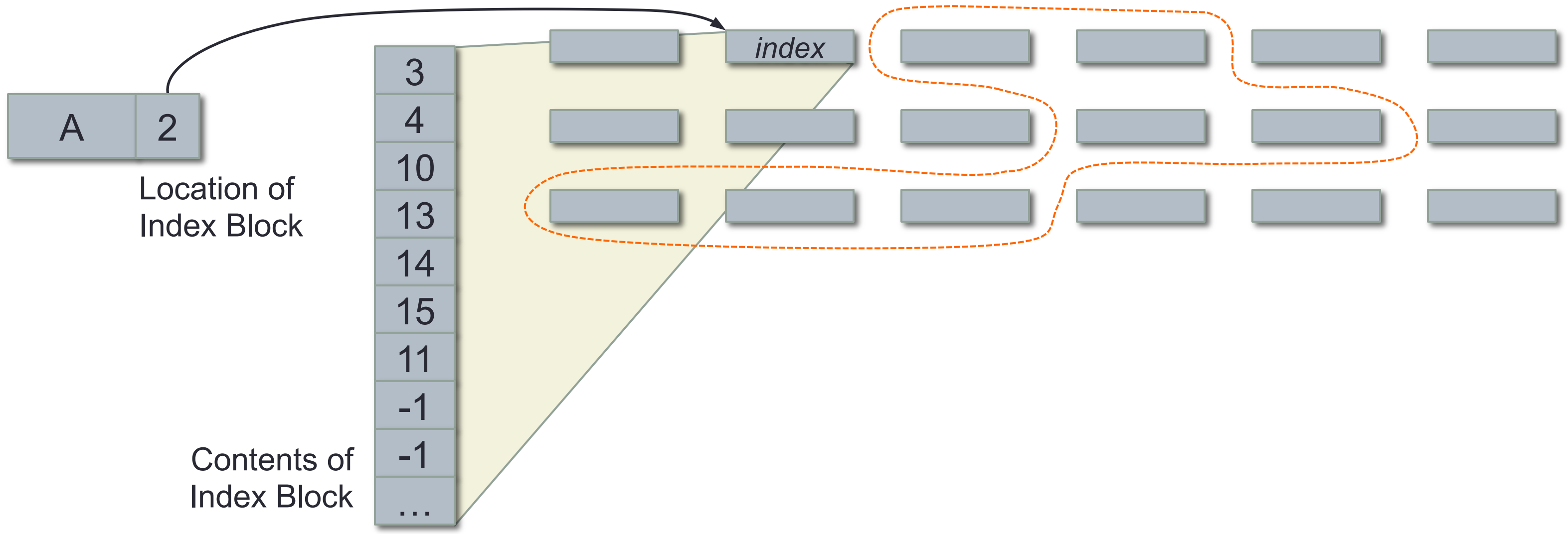
- Both direct and sequential access are very fast
- Very easy to translate a logical file position into the corresponding disk block
  - Position in index = logical position / block size
  - Use value in index to load the corresponding block into memory



File Layout: Indexed Allocation (3)

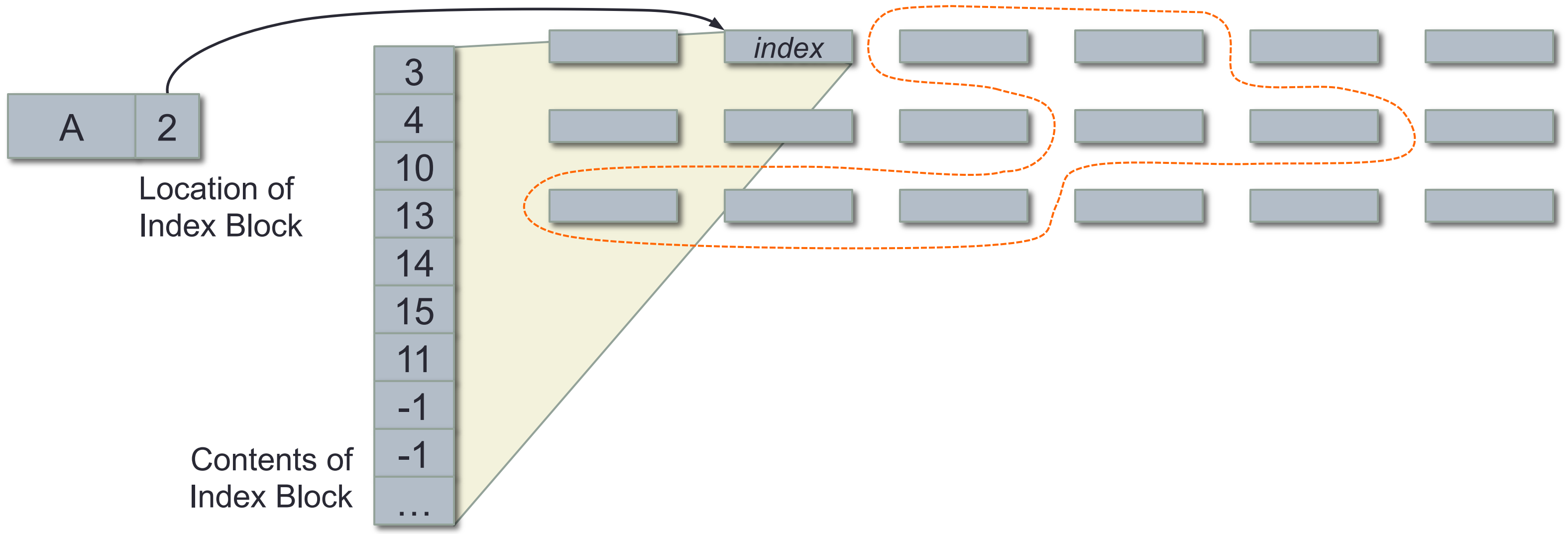
- Index block can also store file metadata
- Recall: many filesystems support hard linking of a file from multiple paths
- If metadata is stored in the directory instead of with the file, metadata must be duplicated, could get out of sync, etc.
  - Indexed allocation can avoid this issue!





File Layout: Indexed Allocation (4)

- Obvious overhead from indexed allocation is the <u>index</u>
  - Tends to be greater overhead than e.g. linked allocation
- Difficult to balance concerns for small and large files
  - Don’t want small files to waste space with a mostly-empty index…
  - Don’t want large files to incur a lot of work from navigating many small index blocks…
- Index space tend to be allocated in units of storage blocks



File Layout: Indexed Allocation (5)

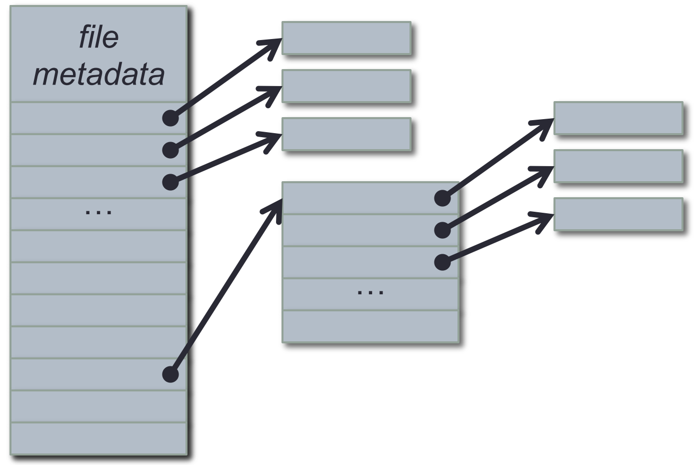
- Option 1: a linked sequence of index blocks
- Each index block has an array of file-block pointers
- Last pointer in index block is either "end of index" value, or a pointer to the next index block
- Good for smaller files
- Example: storage blocks of 512B; 32-bit index entries
  - 512 bytes / 4 bytes = maximum of 128 entries
- Index block might store 100 or more entries (extra space for storing file metadata)
  - 100 entries per index block × 512 byte blocks = ~50KB file size for a single index block
- Usually want to use virtual page size as block size instead
  - Max of 1024 entries per 4KiB page
  - If index entries refer to 4KiB blocks, a single index block can be used for up to 4MB files before requiring a second index block

File Layout: Indexed Allocation (6)

- <u>Option 2</u>: a multilevel index structure
- An index page can reference other index pages, or it can reference data blocks in the file itself (but not both)
- Depth of indexing structure can be adjusted based on the file's size
- As before, a single-level index can index up to ~4MB file sizes
- Above that size, a two-level index can be used:
  - Leaf pages in index can each index up to ~4MB regions of the file
  - Each entry in the root of the index corresponds to ~4MB of the file
  - A two-level index can be used for up to a ~4GB file
  - A three-level index can be used for up to a ~4TB file
  - etc.
- Index can be navigated very efficiently for direct access

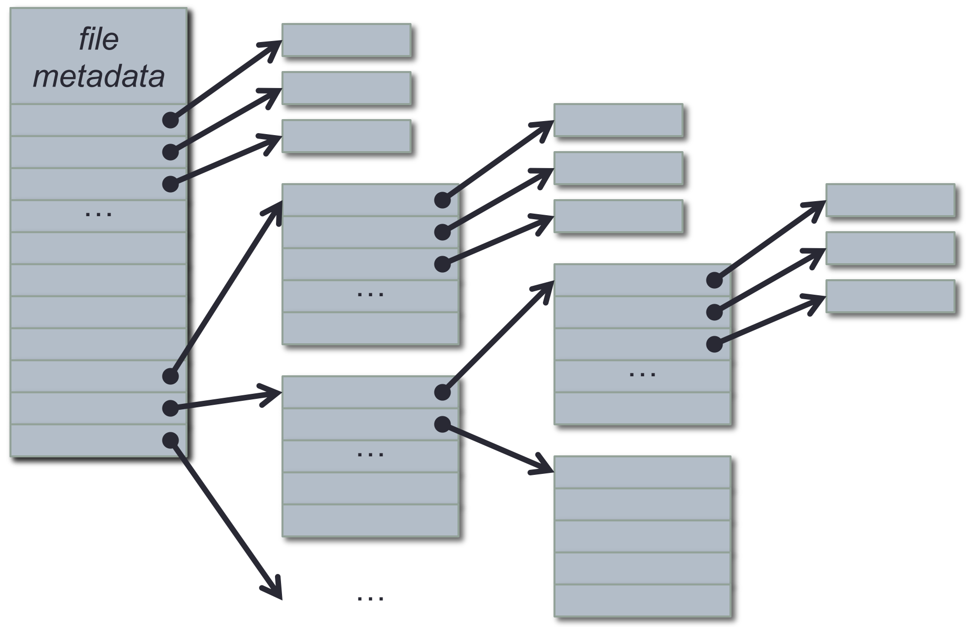
File Layout: Indexed Allocation (7)

- Option 3: hybrid approach that blends other approaches
- Example: UNIX Ext2 file system
- Root index node (**i-node**) holds file metadata
- Root index also holds pointers to the first 12 disk blocks
  - Small files (e.g. up to ~50KB) only require a single index block
  - Called **direct blocks**
- If this is insufficient, one of the index pointers is used for **single indirect blocks**
  - One additional index block is introduced to the structure, like linked organization
  - Extends file size up to e.g. multiple-MB files



File Layout: Indexed Allocation (8)

- For even larger files, the next index pointer is used for **double indirect blocks**
- These blocks are accessed via a two-level index hierarchy
  - Allows for very large files, up into multiple GB in size
- If this is insufficient, the last root-index pointer is used for **triple indirect blocks**
- These blocks use a three-level index hierarchy
  - Allows file sizes up into TB
- A size limit is imposed…
  - More recent extensions to this filesystem format allow for larger files (e.g. extents)



Next Time

- More details on file systems