Chapters 10, 11 and 12: File Allocation

CSCI 3753 Operating Systems

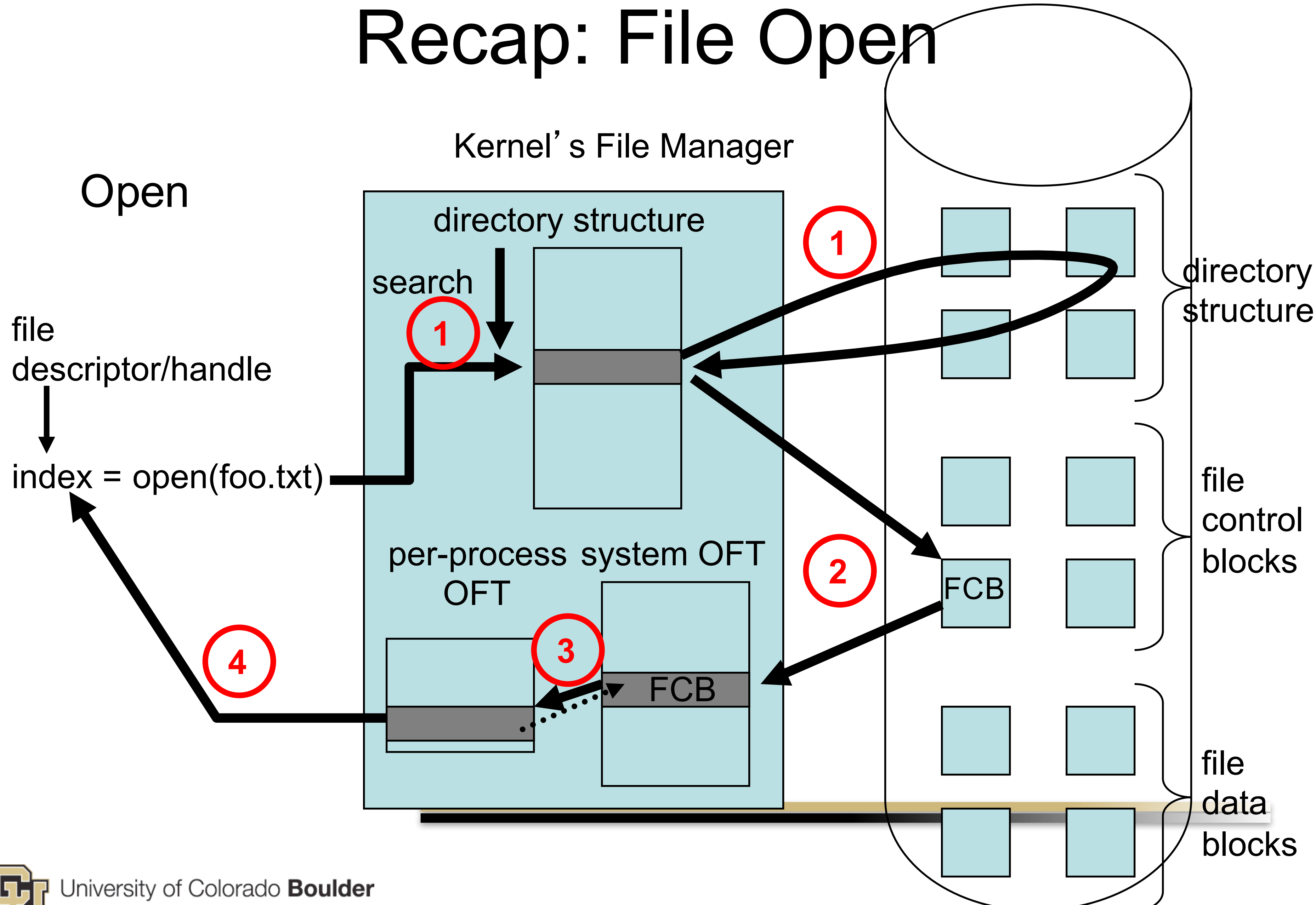
Prof. Rick Han

University of Colorado Boulder

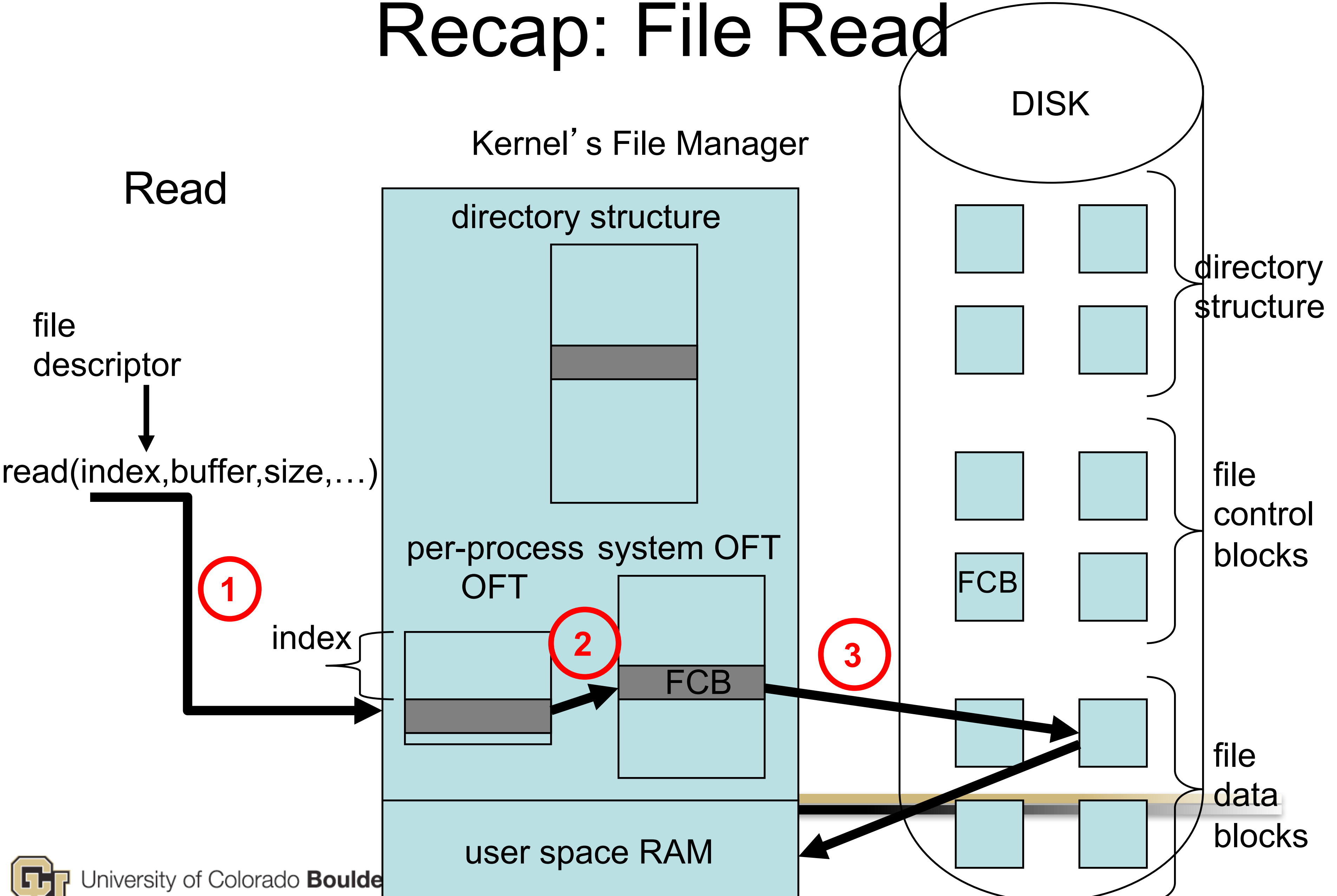
Announcements

- Entrepreneurial Capstone, I will be recommending to some students 1-on-1 in the coming weeks:
  - Great chance to work on something you feel passionately about
  - http://custartupcapstone.github.io/

Recap: File Open



Recap: File Read



Recap: File System Close

- on a close(),
  1. remove the entry from the per-process OFT
  2. decrement the open file counter for this file in the system OFT
  3. if counter = 0, then write back to disk any metadata changes to the FCB, e.g. its modification date
     - Note: there may be a temporary inconsistency between the FCB stored in memory and the FCB on disk – designers of file systems need to be aware of this. A similar inconsistency occurred for modified memory-mapped file data in RAM that had not yet been written to disk.

Directory Implementation

- Implement Directory as a Linear List
  - Searching for a file is slow because each directory requires a linear search
  - Also slow because creating and deleting a file requires a search through the linear list
  - Could keep a sorted list in memory

Directory Implementation

- Implement Directory as a Hash Table
  - Hash the file name, and in each directory search only the short linked list corresponding to that hash value
  - Greatly reduces search time
  - Linux ext3 and ext4 file systems use a variant called HTree, a hashed B-tree for fast lookup

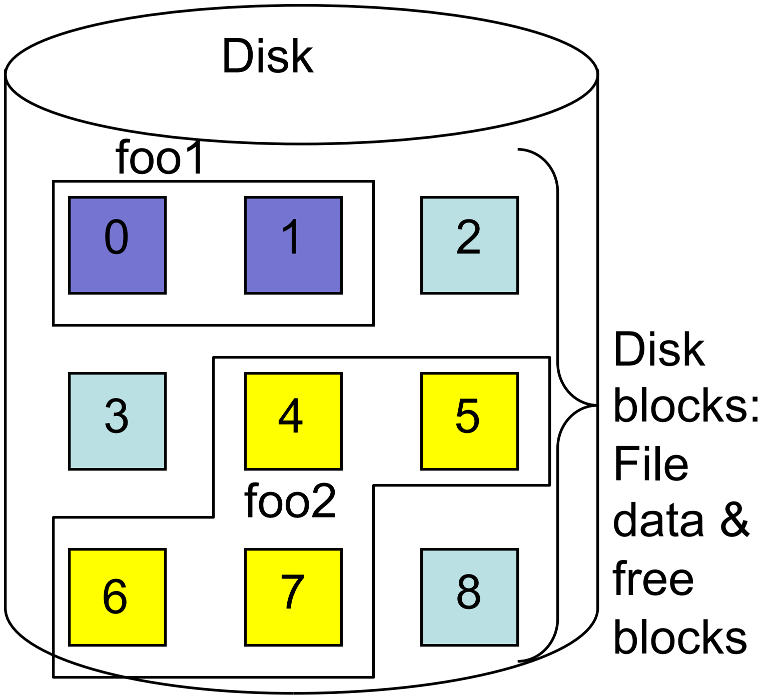
File Allocation

- File allocation concerns how file data is stored or laid out on disk
  – for now, we divide up disk into equally sized blocks
- Approaches:
  1. Contiguous file allocation
     - a file is laid out contiguously, i.e. if a file is n blocks long, then a starting address b is selected and the file is allocated blocks b, b+1, b+2, ..., b+n-1

Contiguous File Allocation

File headers

file	start	length
foo1	0	2
foo2	4	4



- Advantage: fast performance (low seek times because the blocks are all allocated near each other on disk)

Contiguous File Allocation

- Disadvantages:
  - Problem 1: external fragmentation (same problem as trying to contiguously fit processes into RAM)
    - same solutions apply: first fit, best fit, etc.
    - also compact memory/defragment disk
    - can be performed in the background late at night, etc.
  - Problem 2: May not know size of file in advance
    - allocate a larger size than estimated
    - if file exceeds allocation, have to copy file to a larger free “hole” between allocated files

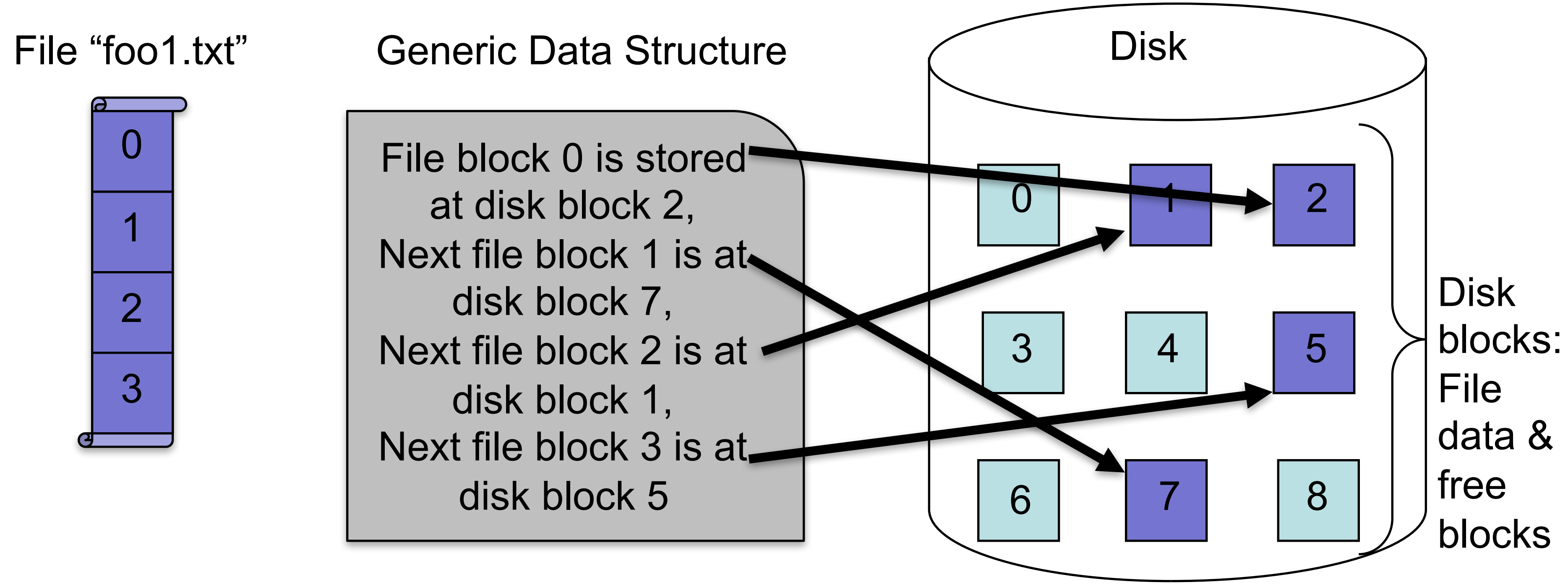
Contiguous File Allocation

- Disadvantages:
  - Problem 3: Over-allocation of a “slow growth” file
    - A file may eventually need 1 million bytes of space
    - But initially, the file doesn’t need much, and it may be growing at a very slow rate, e.g. 1 byte/sec
    - So for much of the lifetime of the file, allocating 1 MB wastes allocation
    - This is a “slow growth” problem.

General File Allocation

- Page table solved external fragmentation problem for process allocation
- Apply a similar concept to file allocation
  – Divide disk into fixed-sized blocks, just as main memory was divided into fixed-sized physical frames
  – Allow a file's data blocks to be spread across any collection of disk blocks, not necessarily contiguous
  – *Need a data structure to keep track of what block of a file is stored on which block in disk*

General File Allocation



- Generic data structure can be:
  - A Linked list and variants
  - Indexed allocation (somewhat resembles a page table) and variants

General File Allocation

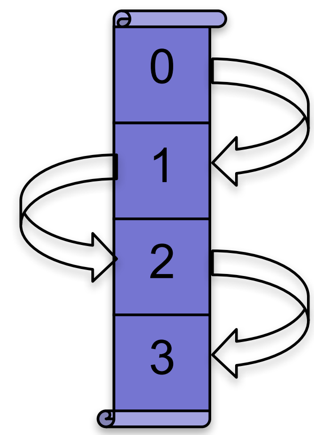
- General approach to file allocation solves:
  - External fragmentation problem
  - Problem of not knowing file size in advance and having to over-estimate
    - Allocate exactly number of disk blocks needed
    - As more disk blocks are needed, easy to allocate exactly the additional number of disk blocks needed from pool of free/unallocated disk blocks
    - and these disk blocks can be anywhere on disk, not necessarily contiguous
  - Slow growth problem: only allocate exactly as many blocks as a slow growth file needs

General File Allocation

- Examples:
  - UNIX FS (UFS, = Berkeley FFS) uses 8 KB blocks.
  - Linux’ file system ext2fs uses default 1 KB blocks (though 2 and 4 KB supported (and much larger))

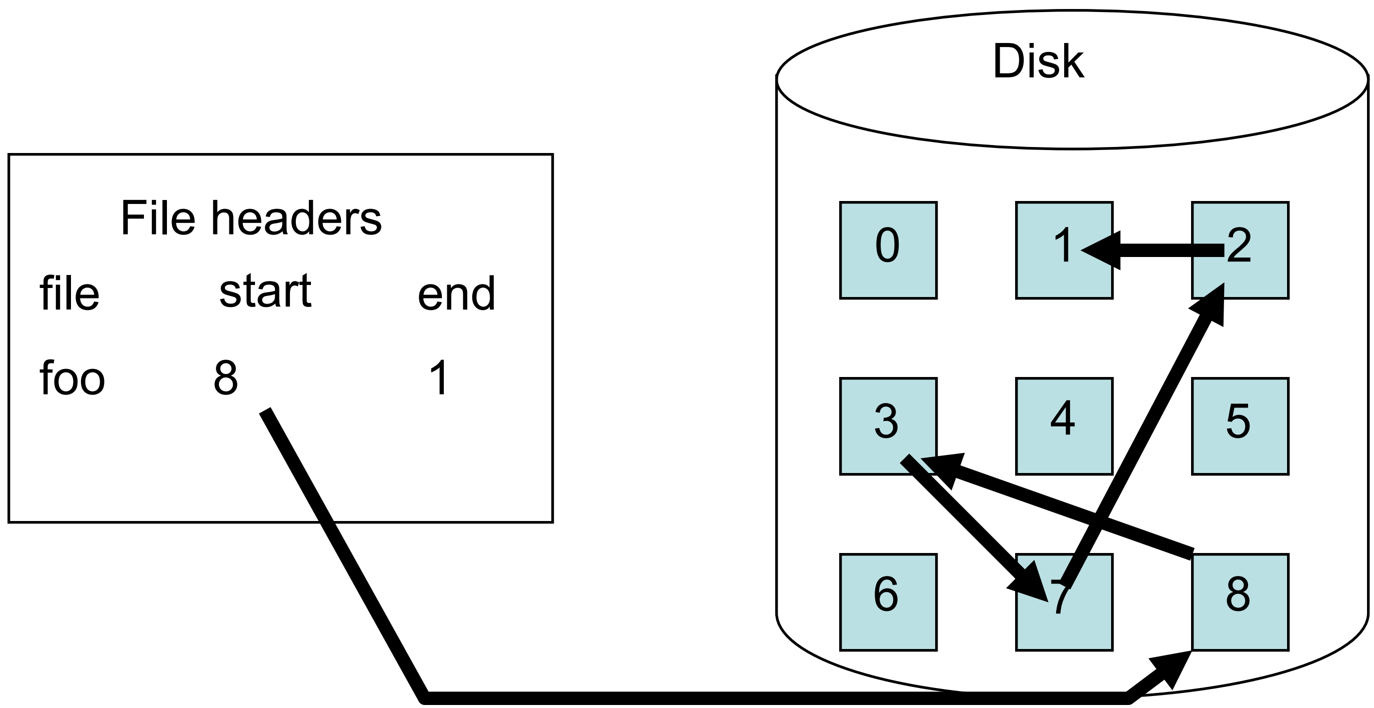
Approach #2: Linked File Allocation

- Linked Allocation
  - each file is a linked list of disk blocks
  - to add to a file, just modify the linked list either in the middle or at the tail, depending on where you wish to add a block
  - to read from a file, traverse linked list until reaching the desired data block



Linked File Allocation

- Linked Allocation
  - each file is a linked list of disk blocks



Linked File Allocation

- Advantages:
  - solves problems of contiguous allocation
    - no external fragmentation
    - don’t need to know size of a file a priori
    - slow growth is not a problem
  - Minimal bookkeeping overhead in file header – just a pointer to start of file on disk
    - Compromise is that all the pointer overhead is stored in each disk block
  - Good for sequential read/write data access
  - Easy to insert data into middle of linked list

Linked File Allocation

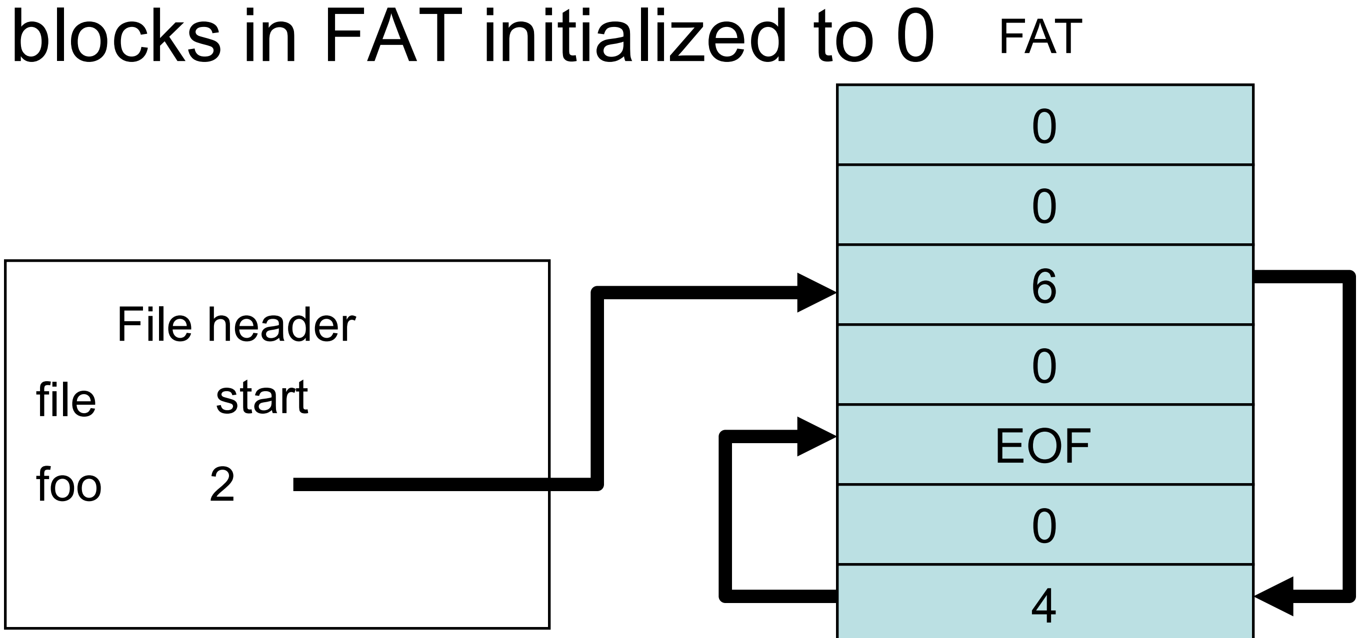
- Problems:
  - performance of random (direct) data access is extremely slow for reads/writes
    - because you have to traverse the linked list until indexing into the correct disk block
  - Space is required for pointers on disk in every disk block
  - reliability is fragile
    - if one pointer is in error or corrupted, then lose the rest of the file after that pointer

Approach #3: File Allocation Table (FAT)

- the File Allocation Table (FAT) is an important variation of linked lists
  - Don’t embed the pointers of the linked list with the file data blocks themselves
  - Instead, separate the pointers out and put them in a special table – the file allocation table (FAT)
  - The FAT is located at a section of disk at the beginning of a volume

File Allocation Table

- entries in the FAT point to other entries in the FAT as a linked list, but their values are interpreted as the disk block number
- unused blocks in FAT initialized to 0



File Allocation Table

- the linked list for a file is terminated by a special end-of-file EOF value
- allocating a new block is simple - find the first 0-valued block
- Advantage: random Reads/Writes faster than pure linked list
  - the pointers are all colocated in the FAT near each other at the beginning of disk volume - low disk seek time
  - still have to traverse the linked list though to find location of data – this is still a slow operation

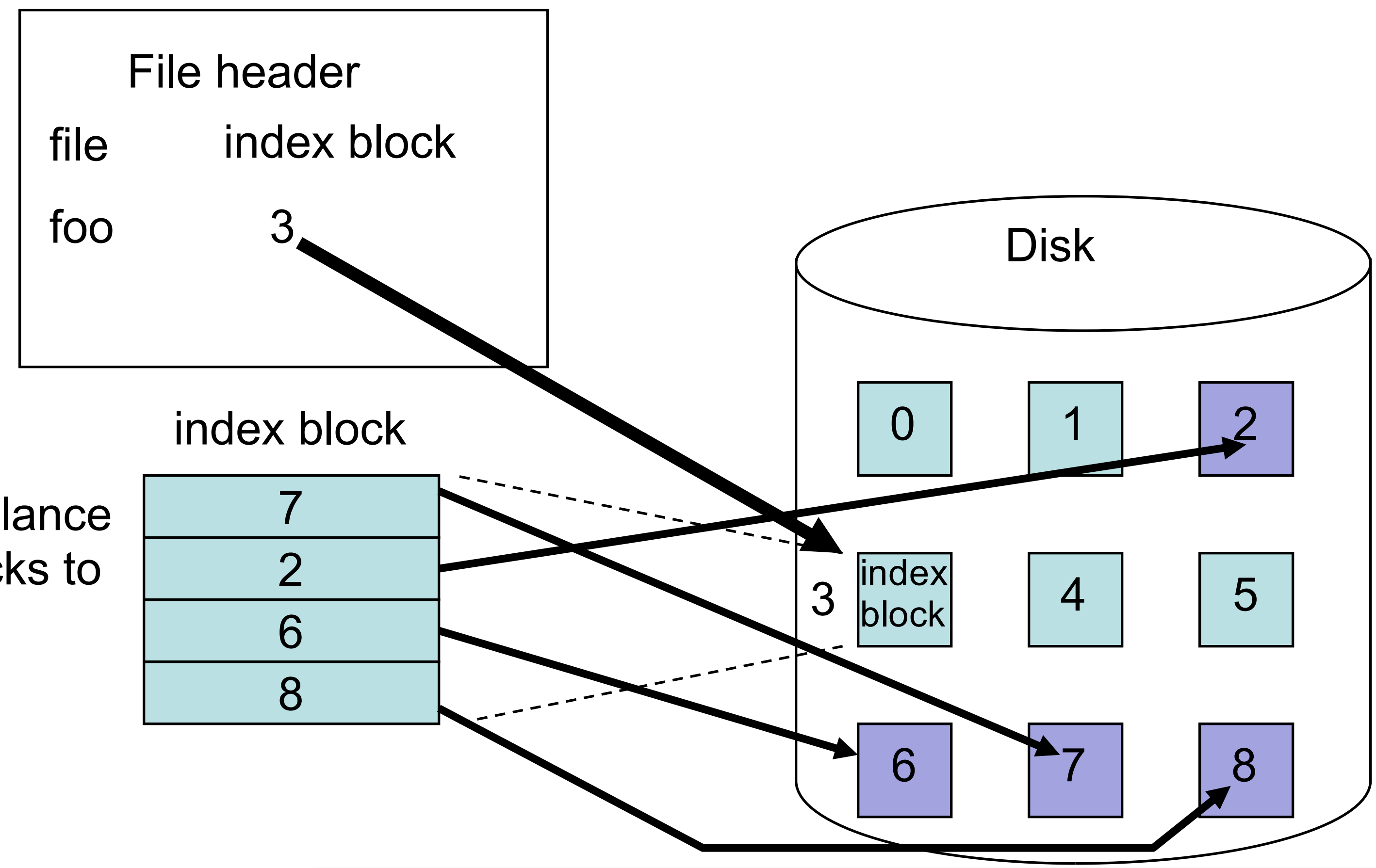
File Allocation Table

- Note resemblance of FATs to inverted page tables (IPTs), absent the pointers
- FAT file systems used in MS-DOS and Win95/98
  - Bill Gates designed/coded original FAT file system
  - replaced by NTFS (basis of Windows file systems from WinNT through Windows Vista/7)
  - Variants include FAT16, FAT32, etc. FAT16 and FAT32 refer to the size of the address used in the FAT.

Approach #4: Indexed Allocation

- conceptually, collect all pointers into a list or table called an *index block*
  - the index j into the list or index block retrieves a pointer to the j'th block on disk
  - Looks kind of like a page table, except it's extensible
- unlike the FAT, the index block can be stored in any block on disk, not just in a special section at the beginning of disk
- unlike the FAT, the index is just a linear list of pointers

Indexed Allocation



note resemblance of index blocks to page tables!

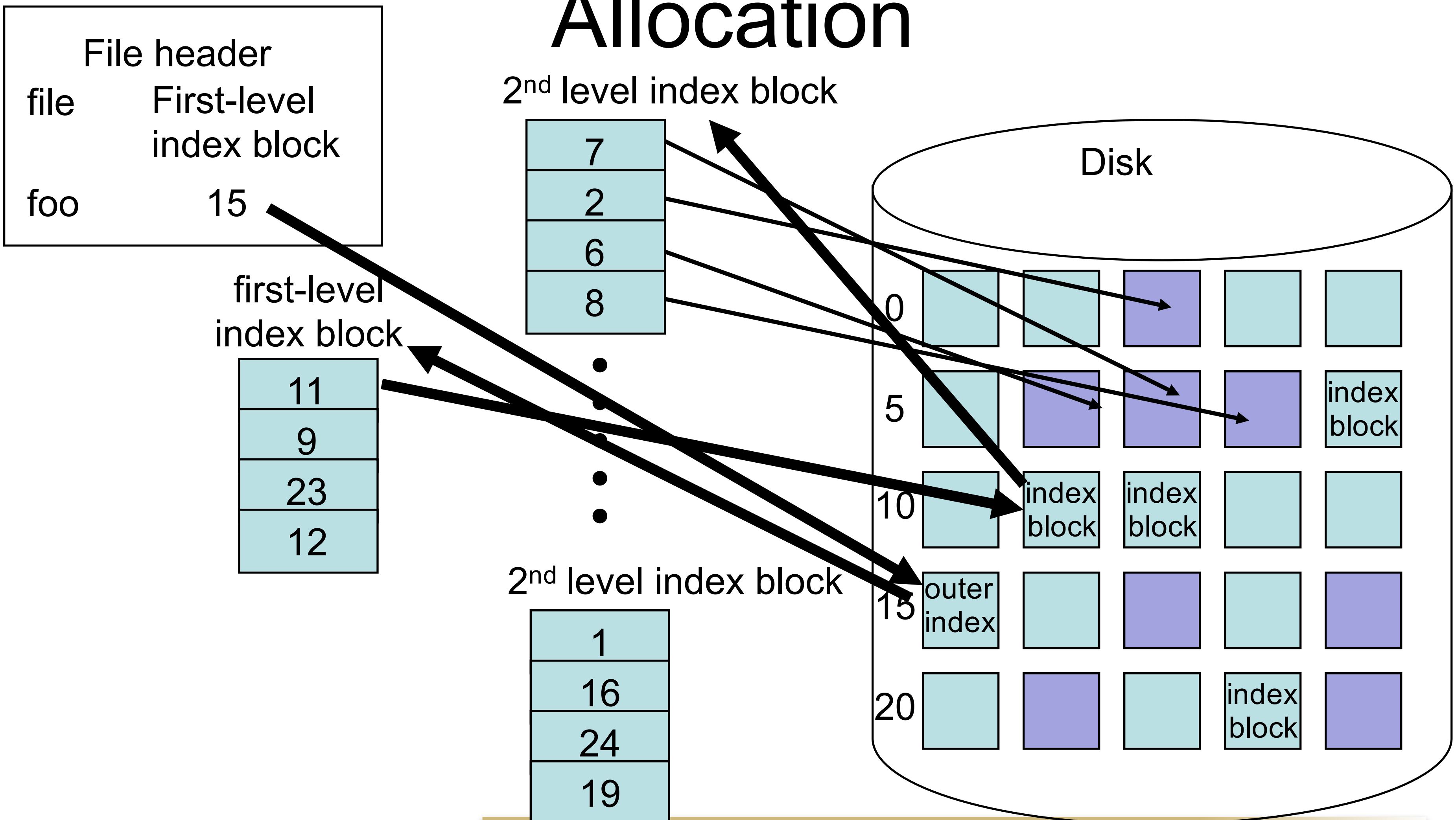
Indexed Allocation

- Solves many problems of contiguous and linked list allocation:
  - no external fragmentation
  - size of file not required a priori
  - slow growth is efficiently supported
  - don't have to traverse linked list for random/direct reads/writes
    - just index quickly into the index block

Indexed Allocation

- Problem: how big should the index block be?
  - if the index block is too large, then there are many wasted/empty entries for small files
  - if the index block is too small, then there are not enough entries for large files
- Solutions:
  - link together index blocks
  - *multilevel index* (like hierarchical page tables!)
    - indexing into the first-level index block provides a pointer to second-level index blocks. Indexing into the second-level index block (using a different offset) retrieves a pointer to ~~the actual file block on disk~~

Approach #5: Multilevel Indexed Allocation



Multilevel Indexed Allocation

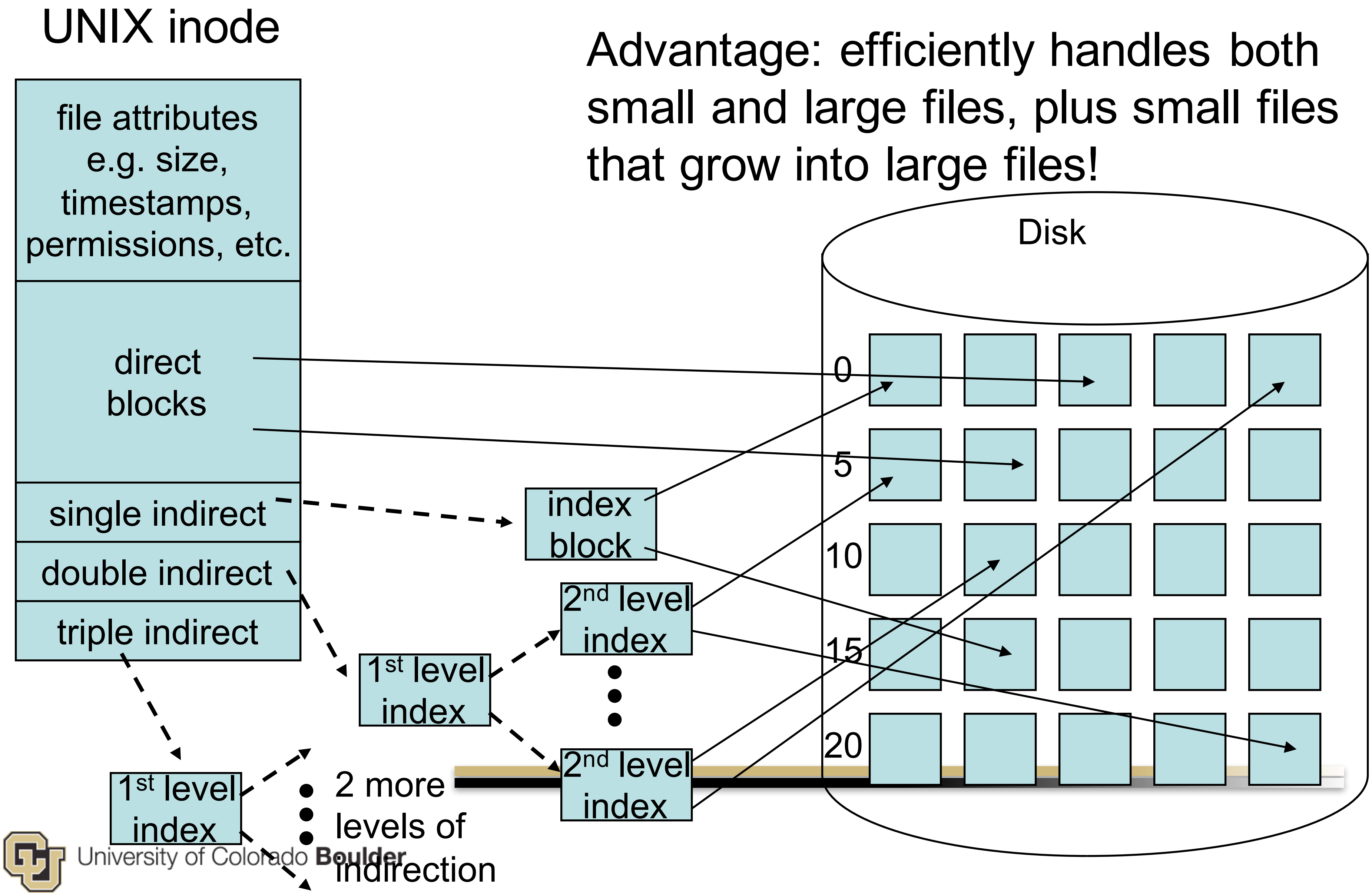
- Don’t have to allocate unused second-level index blocks!
- example: two levels of index blocks, 1024 pointer entries/block => 1 million addressable data blocks. If each block is 4 KB, then the largest file size is 4 GB.
- Problem: with multi-level indexing, accessing small files takes just as long as large files
  - have to go through the same # of levels of indexing, hence same # of disk operations

Approach #6: UNIX Multilevel Indexed Allocation

- UNIX (and Linux ext2fs, ext3fs, etc.) uses this variation of multi-level indexing to accommodate large and small files
  - Suppose there are 15 entries in the index block
  - the first 12 entries are pointers to direct blocks of file data on disk
  - the 13th pointer points to a singly indirect block, which is an index block pointing to disk blocks
  - the 14th pointer points a *doubly indirect block* (2 levels of index blocks)
  - the 15th pointer points to *a triply indirect block* (3 levels of index blocks)

UNIX Multilevel Indexed Allocation

Advantage: efficiently handles both small and large files, plus small files that grow into large files!



UNIX Multilevel Indexed Allocation

- for small files, this approach only uses a small index block of 15 entries, so there is very little wasted memory
- for large files, the indirect pointers allow expansion of the index block to span a large number of disk blocks
- UNIX stores this hybrid index block with the file inode

Comparing File Allocation with Process Allocation

- In both cases, mapping an entity to storage
  - Process address space allocated frames in RAM via page tables
  - File data is allocated to disk/flash
- Differences:
  - Address spaces are fixed in size and known in advance, while files tend to grow/contract over time – files need a mapping/allocation system that is more flexible than page tables, which can't grow
  - Address spaces can be sparse and mostly unused, while file data is all "used"

Comparing File Allocation with Process Allocation

- Similarities:
  - Note a FAT looks very much like an IPT, except the FAT has pointers to the next block.
    - I don't know why an IPT isn't modified to support pointers – perhaps too slow for RAM
  - Indexed allocation looks very much like a page table, except there's room for growth