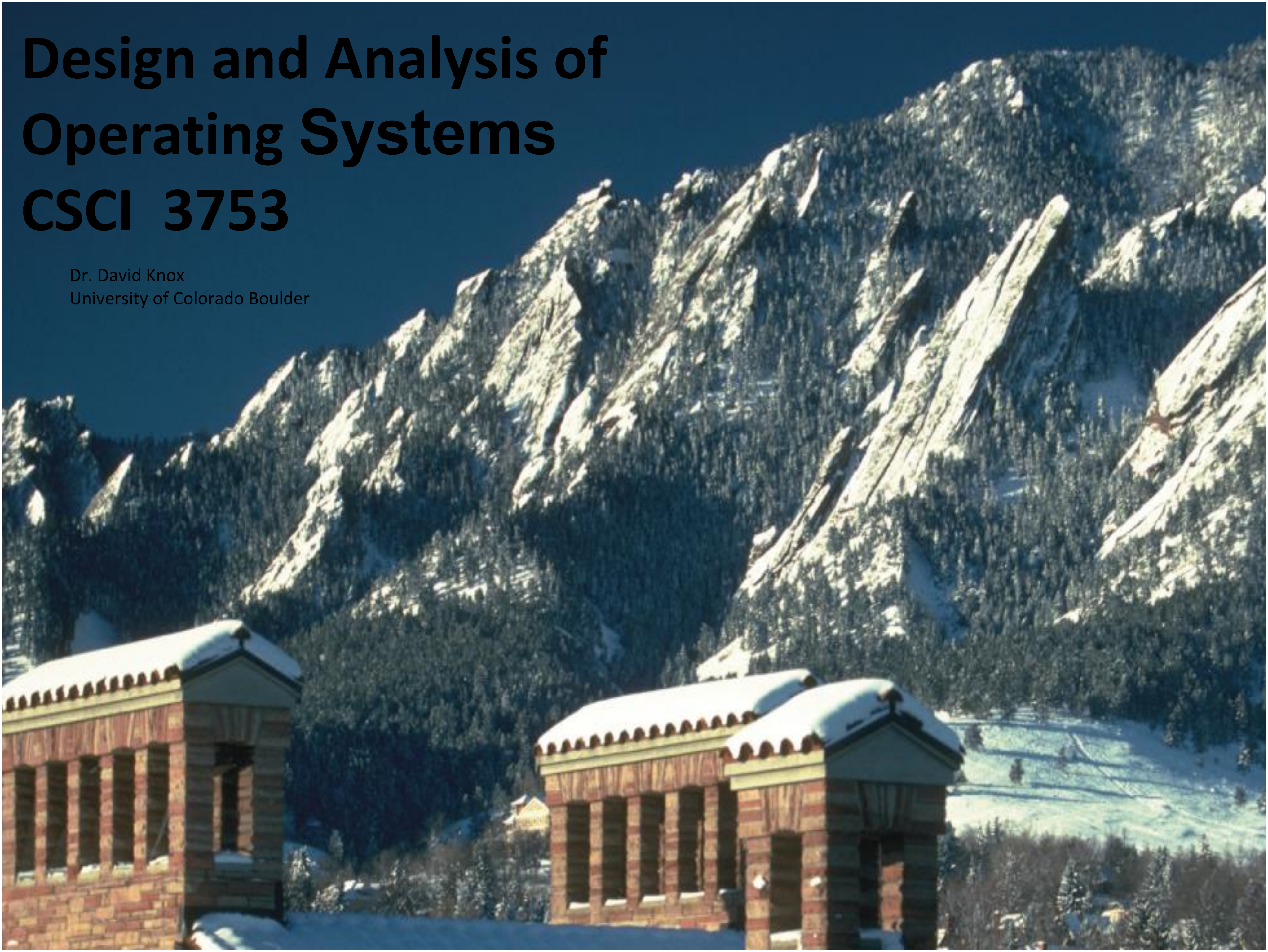
Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox
University of Colorado Boulder



Design and Analysis of Operating Systems CSCI 3753

File Allocation

Dr. David Knox

University of Colorado Boulder

File Allocation

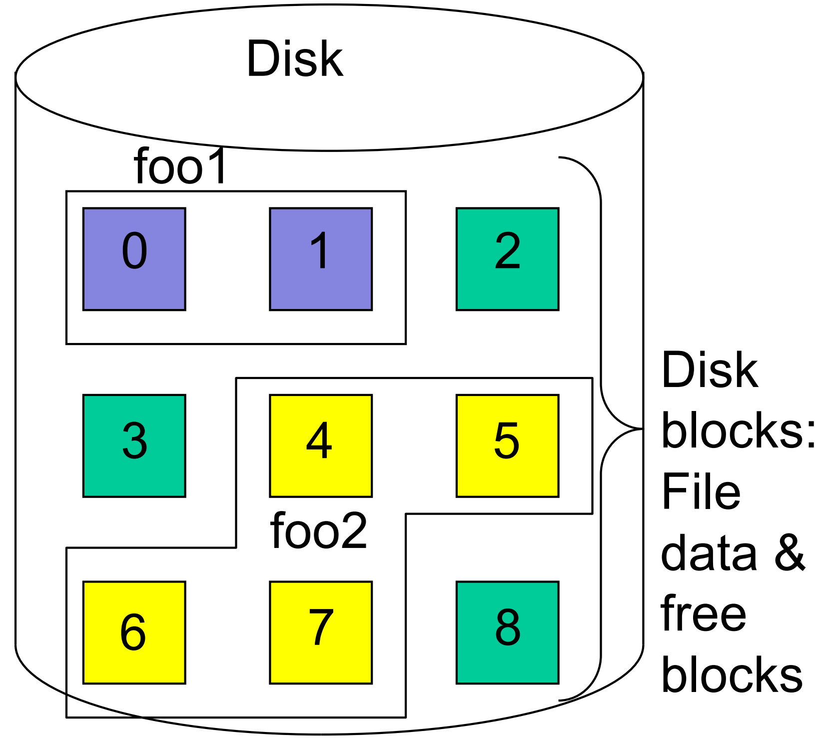
File Allocation

- **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
  2. Linked Allocation
     - each file is a linked list of disk blocks
  3. 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
  4. Indexed Allocation
     - 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

Approach #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)**

Approach #1: Contiguous File Allocation

- **Disadvantages:**
  - **Problem 1:** external fragmentation (observed same problem fitting into RAM)
    - same solutions apply: first fit, best fit, etc.
    - can also compact memory/defragment disk
    - schedule to 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”
  - **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
      (may be growing at a very slow rate, e.g. 1 byte/sec)
    - For much of the lifetime of the file, allocating 1 MB wastes allocation

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

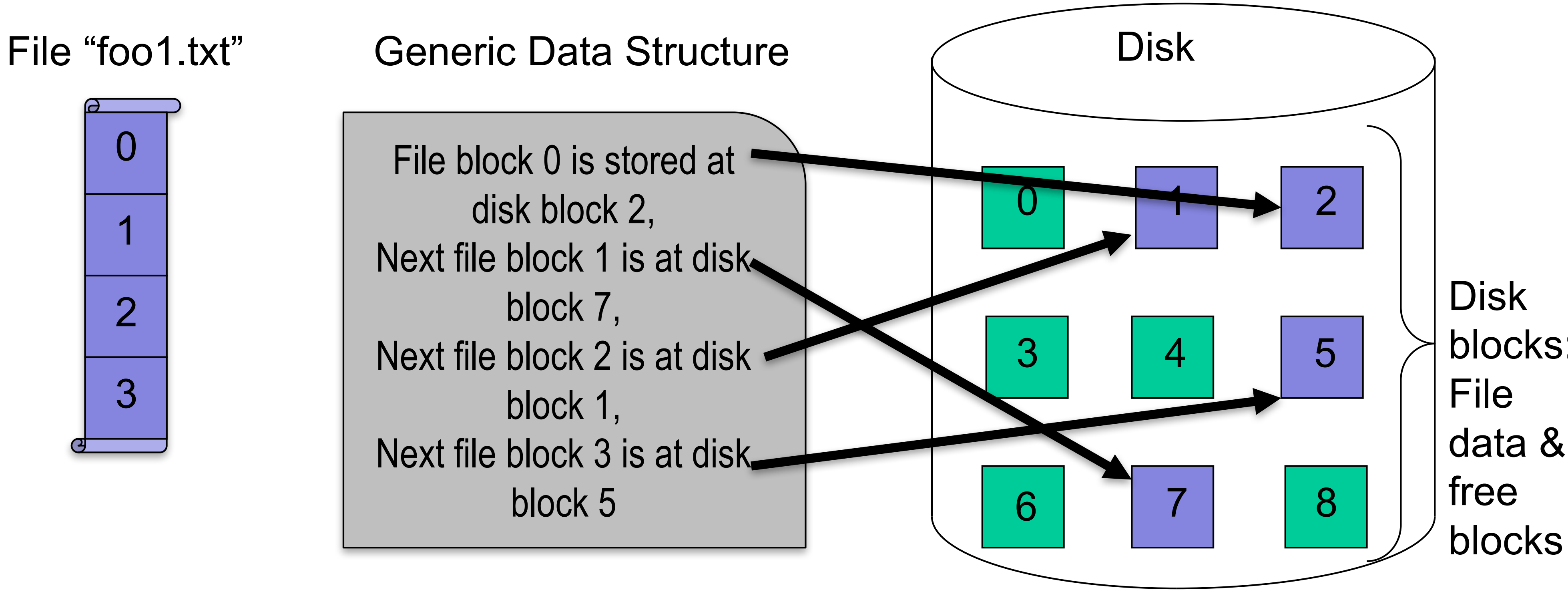
- **Page table solved external fragmentation problem for process allocation**

- **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

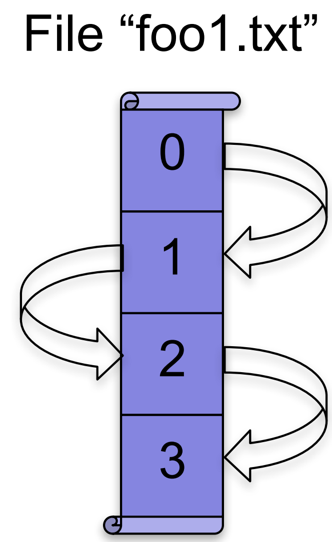
  - Slow growth problem: only allocate exactly as many blocks as a slow growth file needs

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))

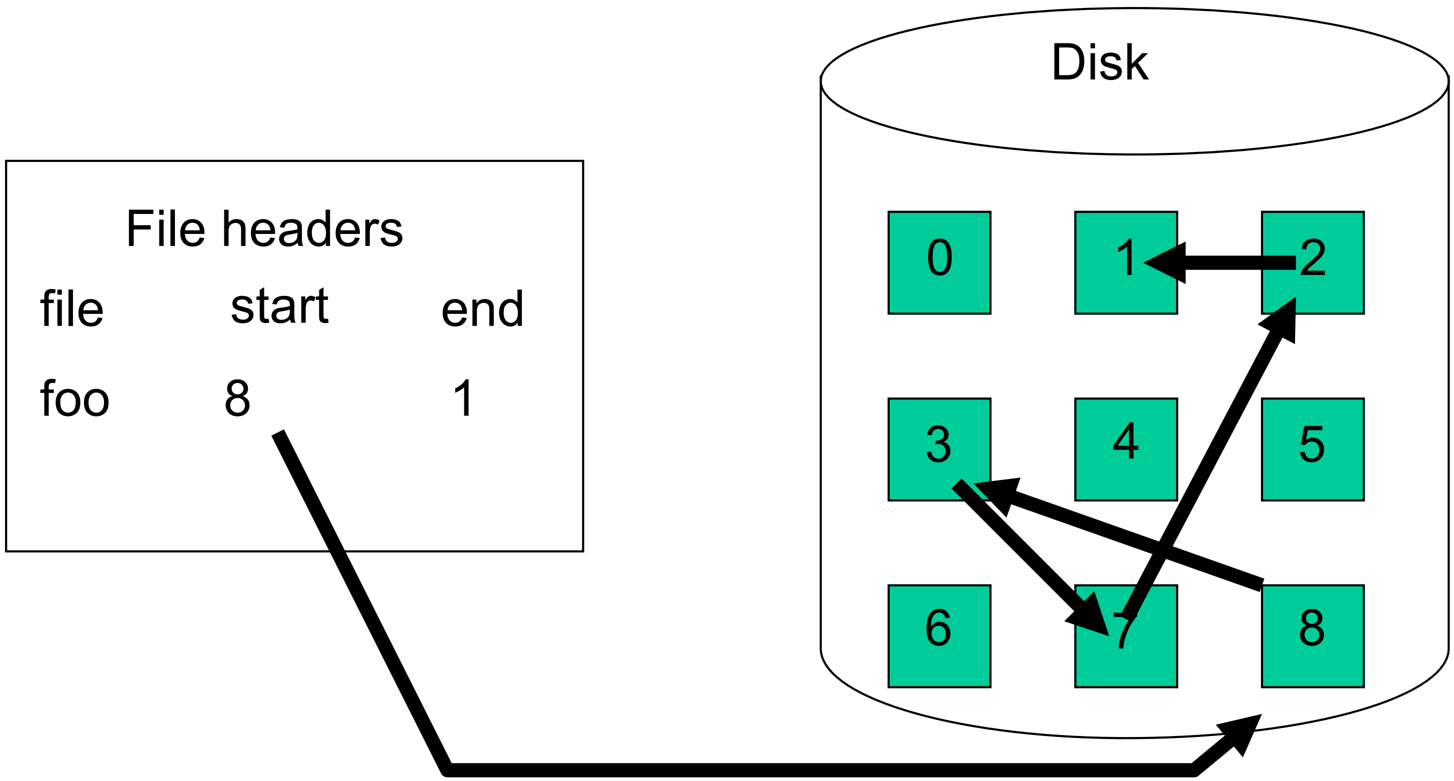
Approach #2: Linked File Allocation

- **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
  - 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

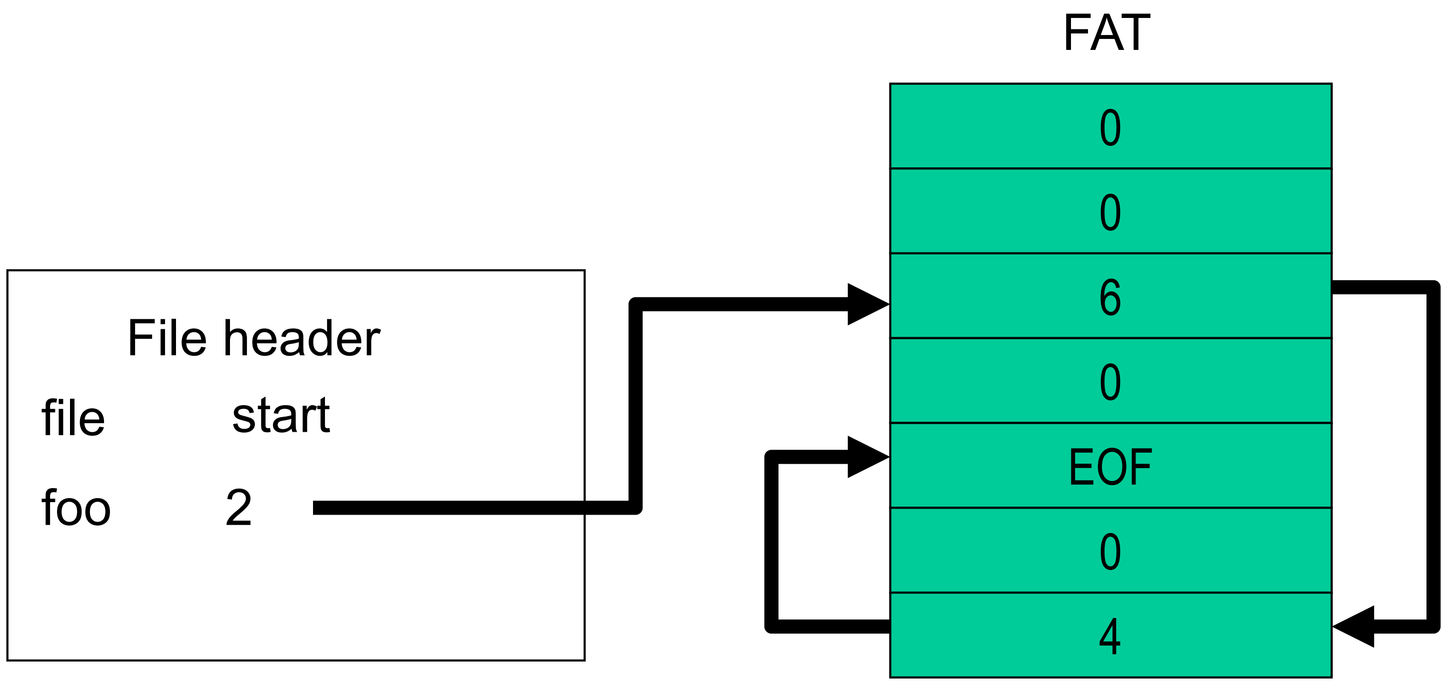
- **Problems:**
  - performance of random (direct) data access is extremely slow for reads/writes
    - because you have to traverse the linked list until indexing into the correct disk block
  - Space is required for pointers on disk in every disk block
  - reliability is fragile
    - if one pointer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

Approach #3: File Allocation Table (FAT)

- **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

- **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.

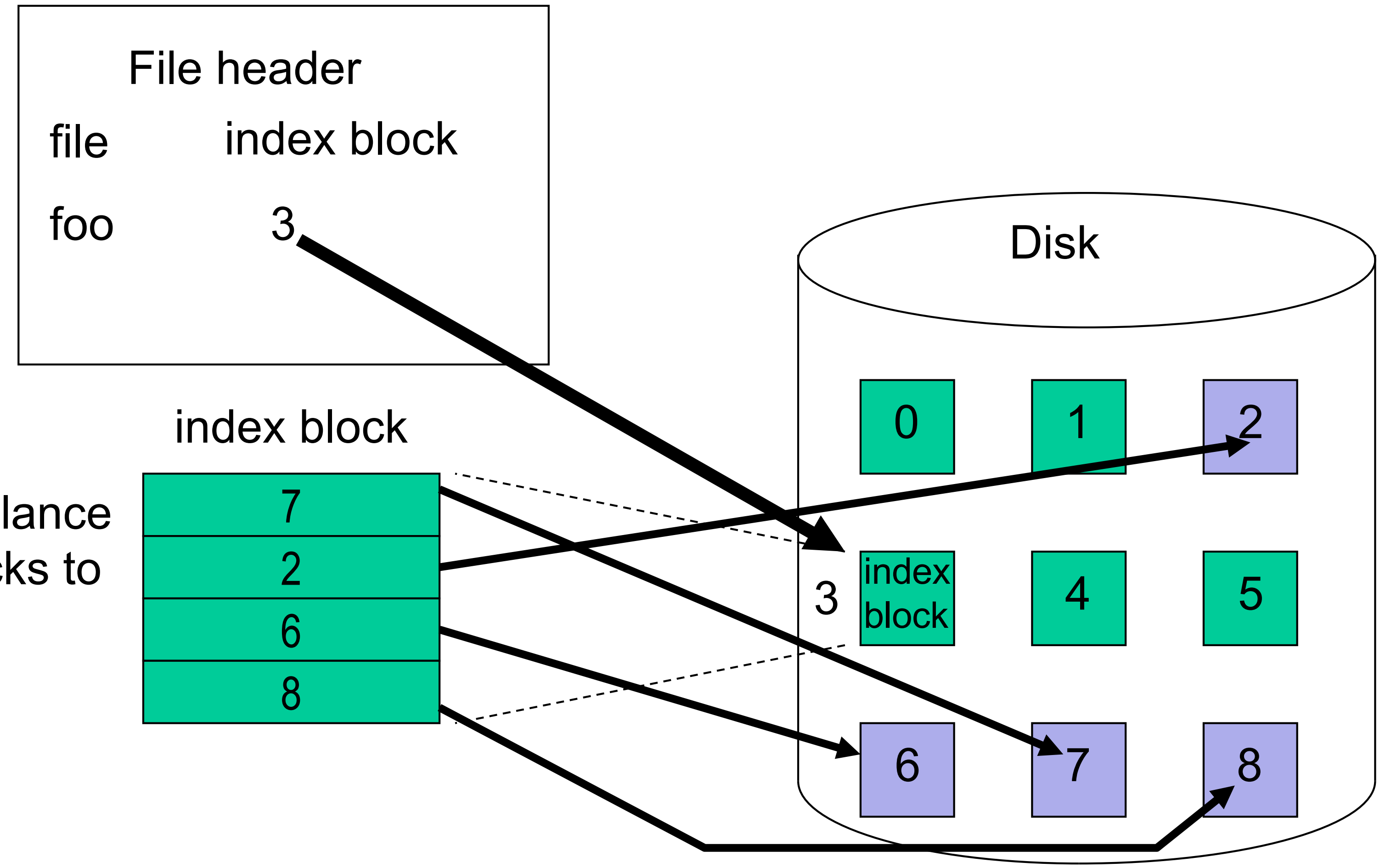
File Allocation Table

- **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 to find location of data – this is a slow operation**

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
  - the index is just a linear list of pointers

Indexed Allocation



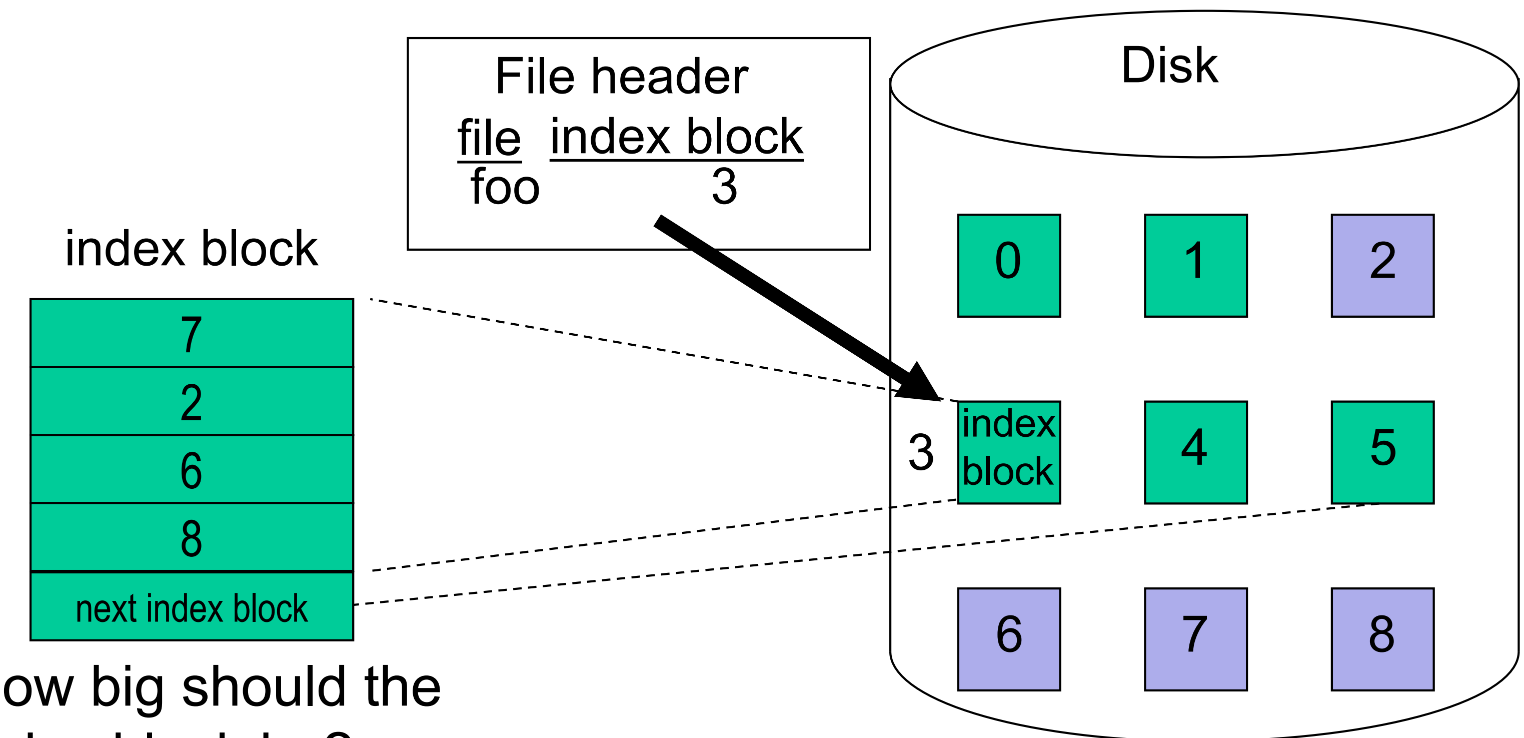
note resemblance of index blocks to page tables!

Indexed Allocation

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

Indexed Allocation

- **Solutions:**
  1. Link together index blocks –
     - each index block has link to next index block
  2. *Multilevel index* (like hierarchical page tables!)
     - First level is list of all index blocks for file
     - Second level is list of all data blocks in that section of the file



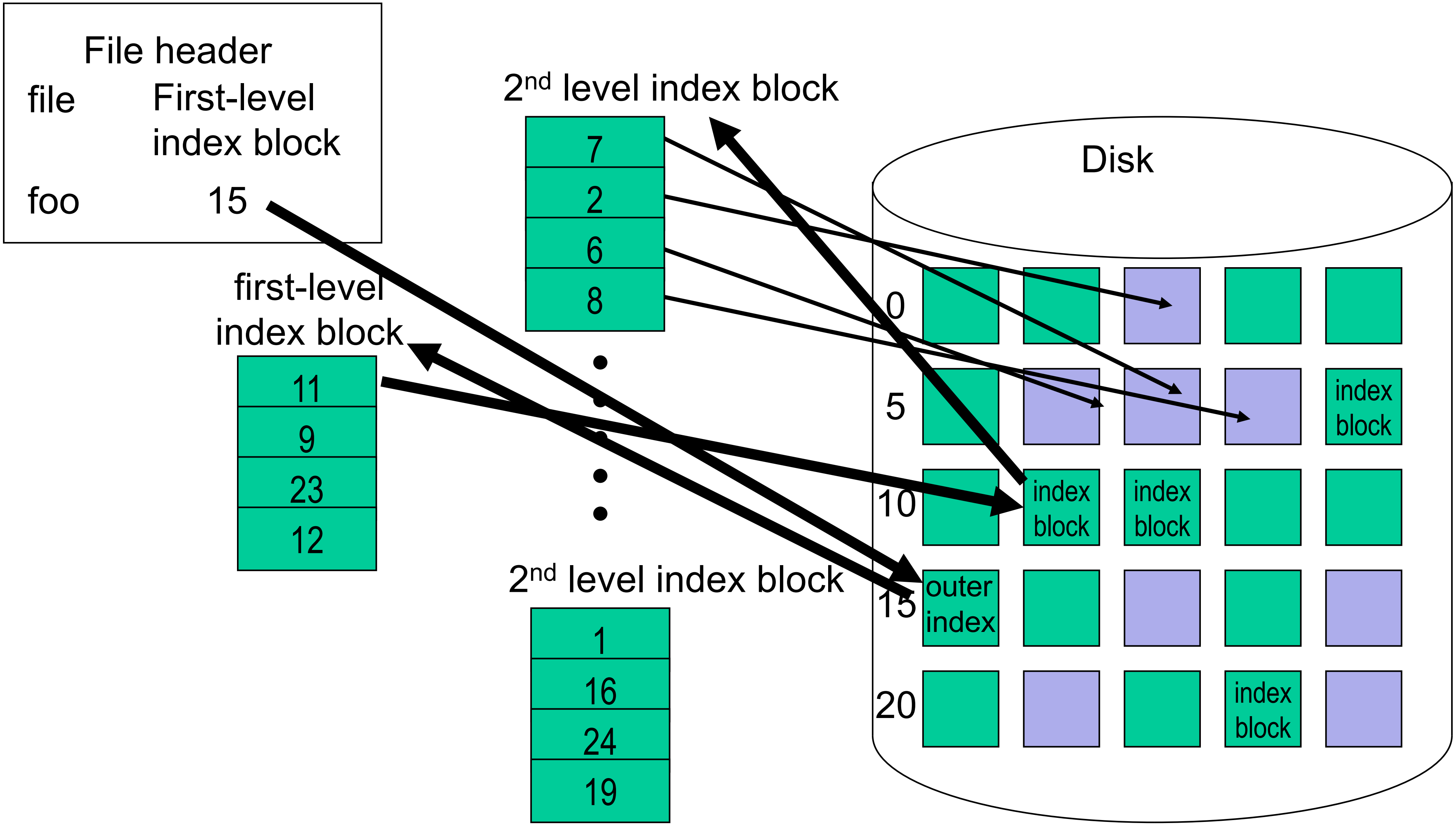
How big should the index block be?

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!**
- **Maximum file size?**

  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.

- **Problems 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
  - accessing the data of a 100 byte file requires at leas 4 block reads

Approach #6: UNIX Multilevel Indexed Allocation

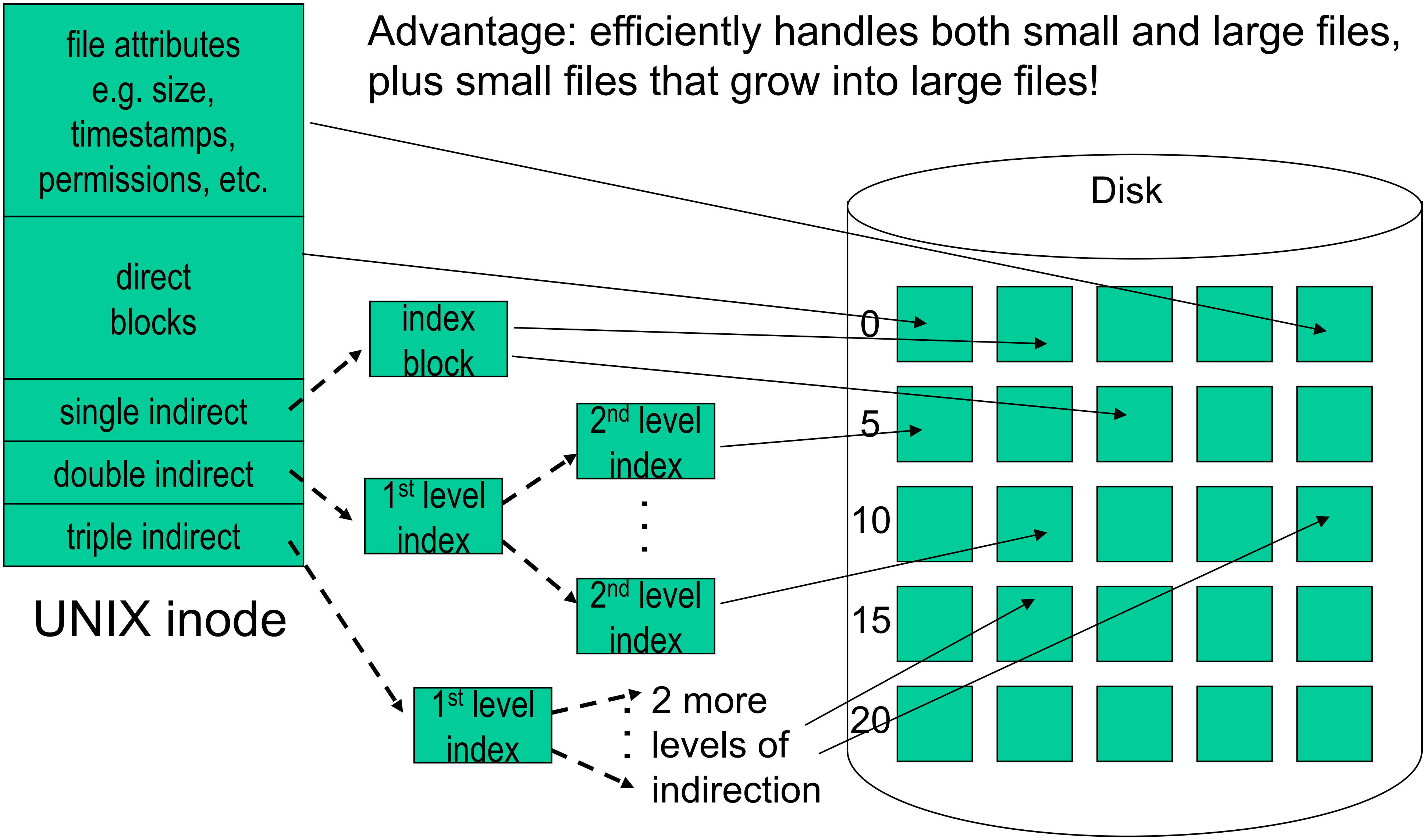
- **UNIX (and Linux ext2fs, ext3fs, etc.) uses this variation of multi-level indexing to accommodate large and small files**
  - Suppose there are 15 entries in the index block
  - the first 12 entries are pointers to direct blocks of file data on disk
  - the 13th pointer points to a singly indirect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

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

UNIX Multilevel Indexed Allocation

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



UNIX Multilevel Indexed Allocation

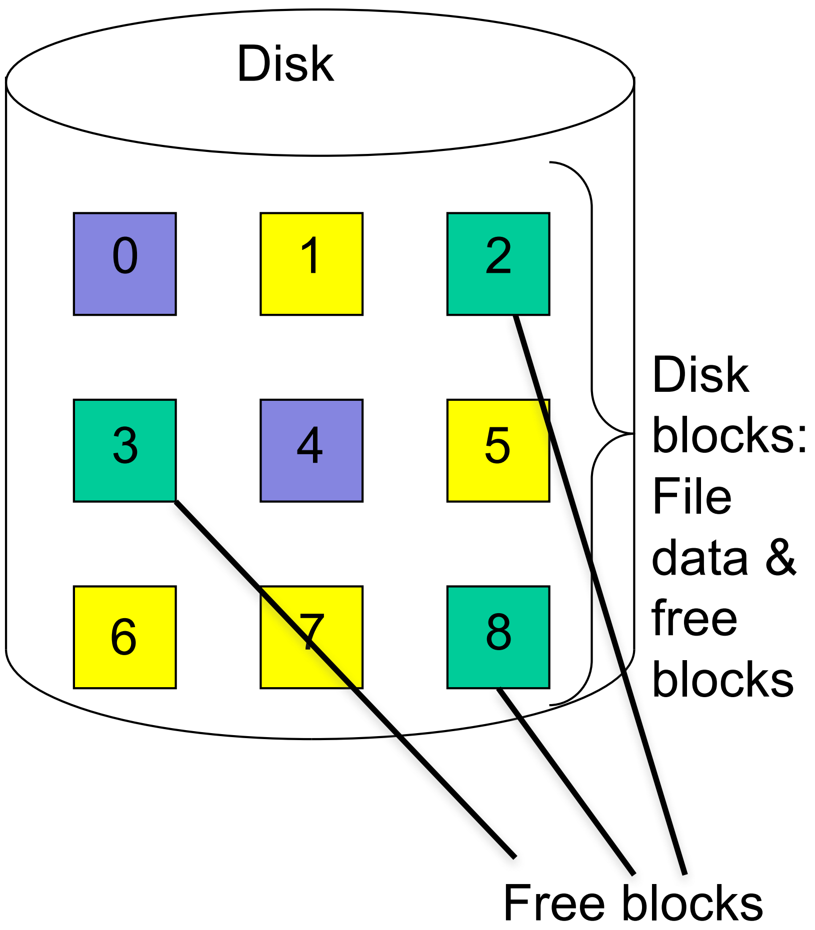
- **for small files**
  - only uses a small index block of 15 entries, so there is very little wasted memory
- **for large files**
  - the indirect pointers allow expansion of the index block to span a large number of disk blocks

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,
  - Files 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"

Free Space Management

- **Another aspect of managing a file system is managing free space**
  - the file system needs to keep track of what blocks of disk are free/unallocated
  - keeps a free-space "list"
  - In this example, need to keep track that disk blocks 2, 3 and 8 are free/unallocated



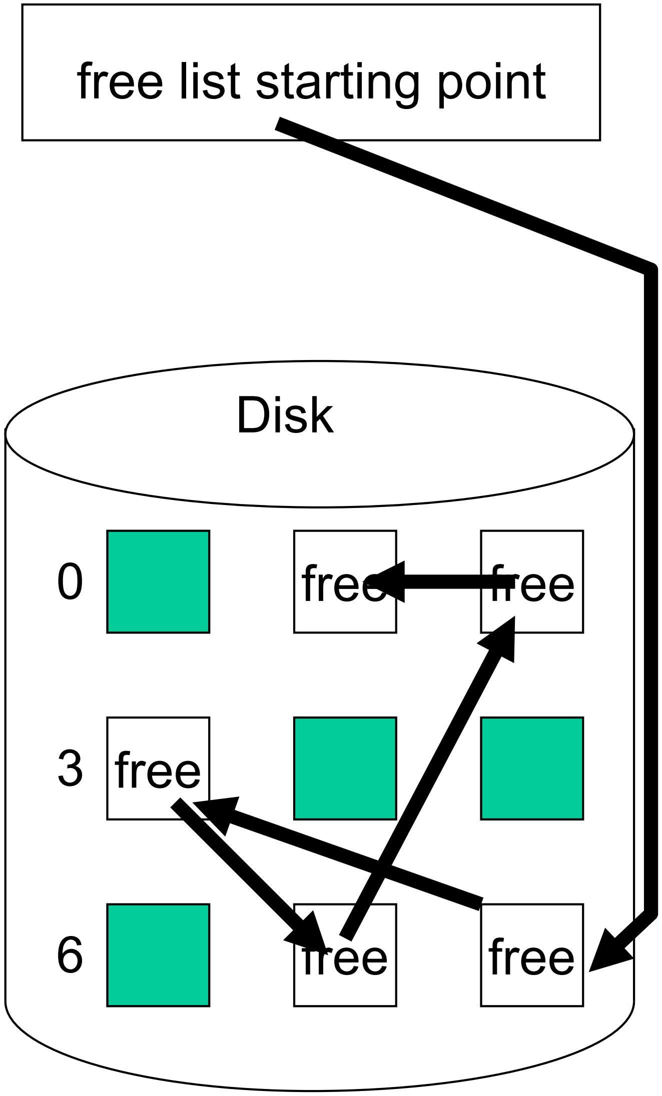
Free Space Management Approaches

1. **Bit Vector or Bit Map**

- **Each block is represented by a bit.**
- **Concatenate all such bits into an array of bits, namely a bit vector.**
  - The j'th bit indicates whether the j'th block has been allocated.
  - if bit = 1, then a block is free, else if bit = 0, then block is allocated

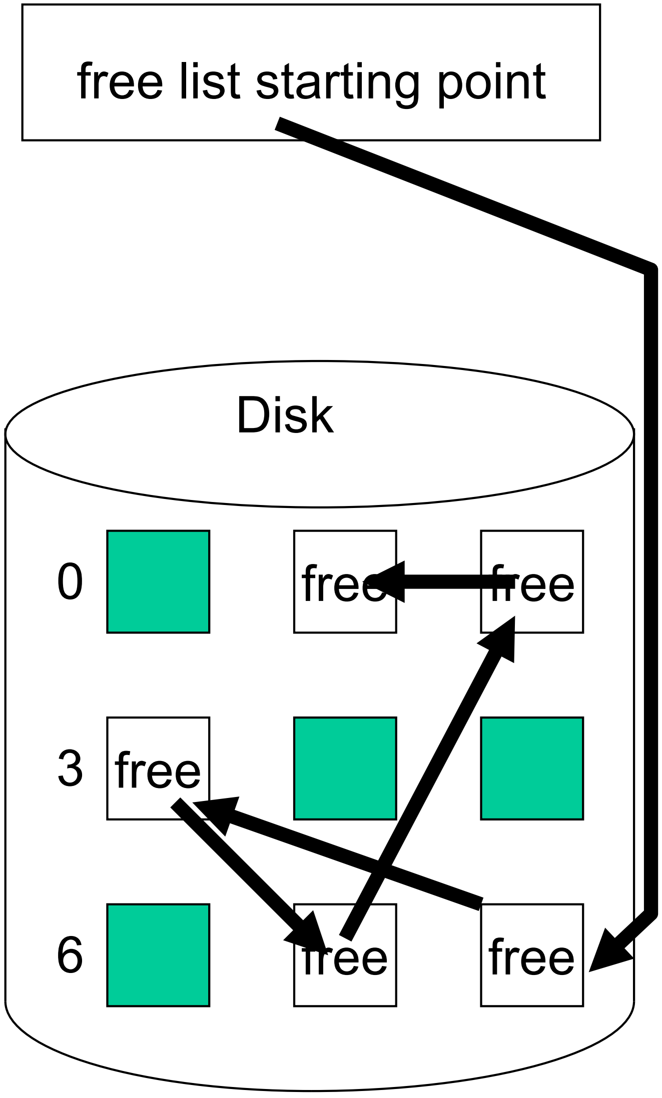
Free Space Management Approaches

2. **Linked List**
   - link together all free blocks
   - efficient - keeps track of only the free blocks.
     - bitmap has the overhead of tracking both free and allocated blocks - this is wasteful if memory is mostly allocated
   - Faster than bitmap – find 1st free block immediately



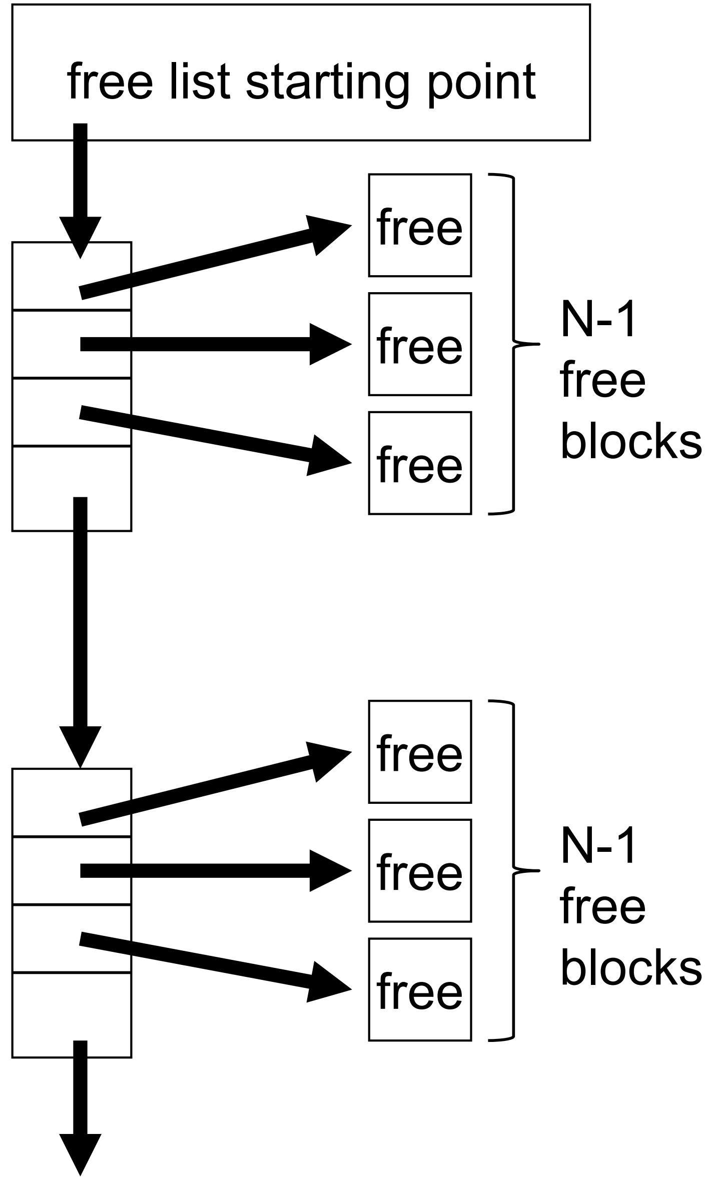
Free Space Management Approaches

- **Problem with Linked List free space management:**
  - traversing the free list is slow if you want to allocate a large number of free blocks all at once
    - hopefully this occurs infrequently



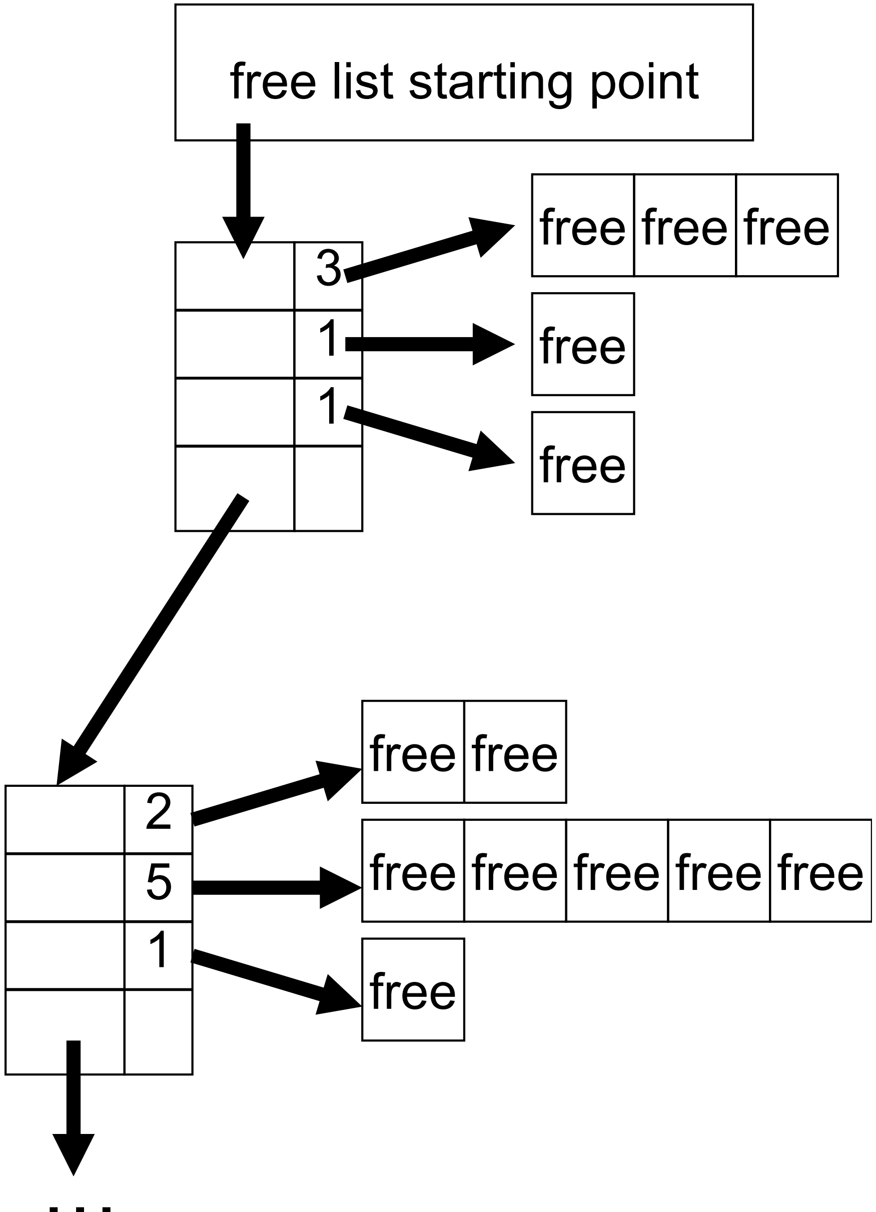
Free Space Management Approaches

3. **Grouping**
   - linked list, except store n-1 pointers to free blocks in each list block
   - the last block points to the next list block containing more free pointers
   - allows faster allocation of larger numbers of free blocks all at once



Free Space Management Approaches

4. **Counting -**
   - grouped linked list, but also add a field to each pointer entry that indicates the number of free blocks immediately after the block pointed to
   - even faster allocation of large #'s of free blocks



Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox
University of
Colorado Boulder