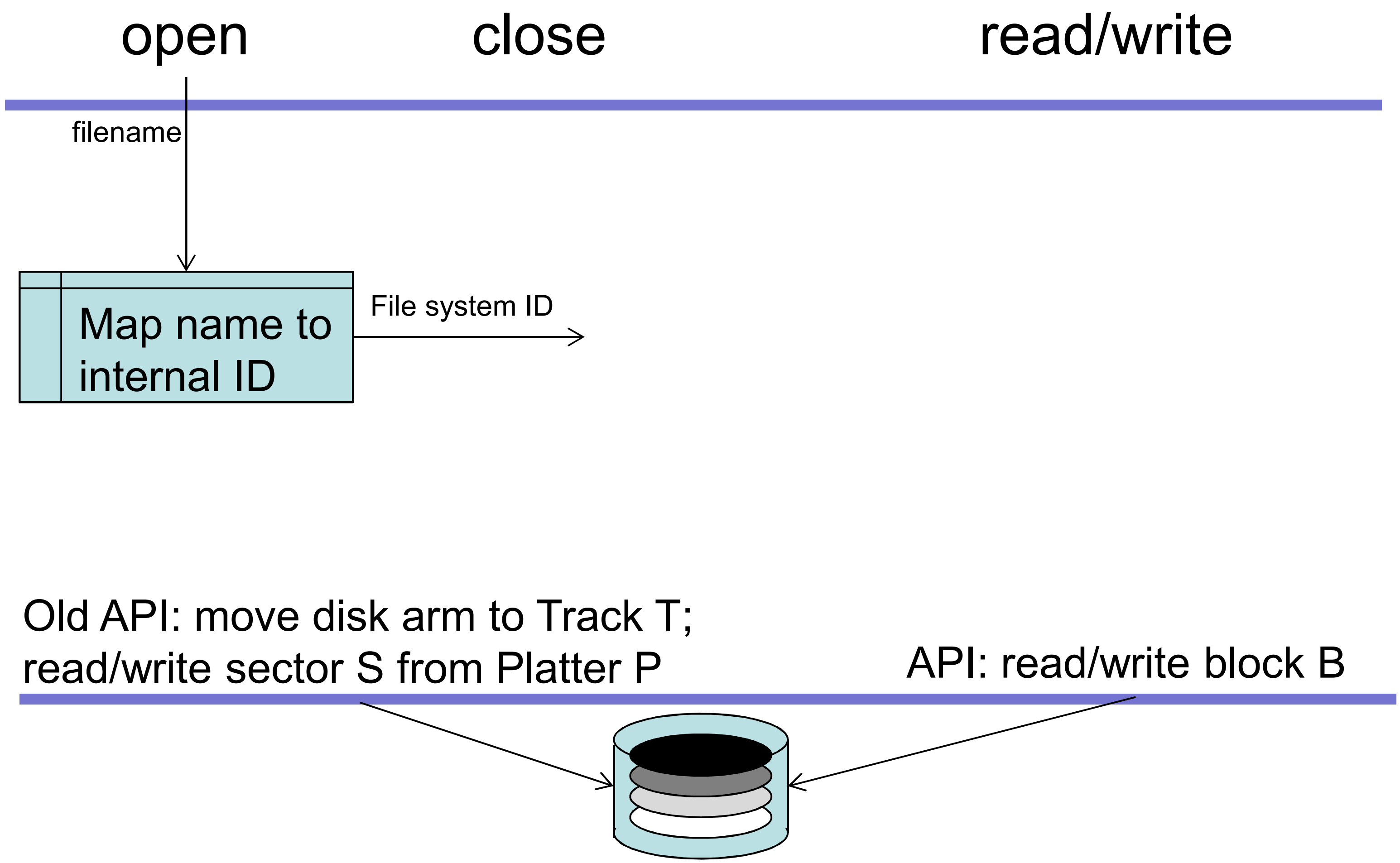
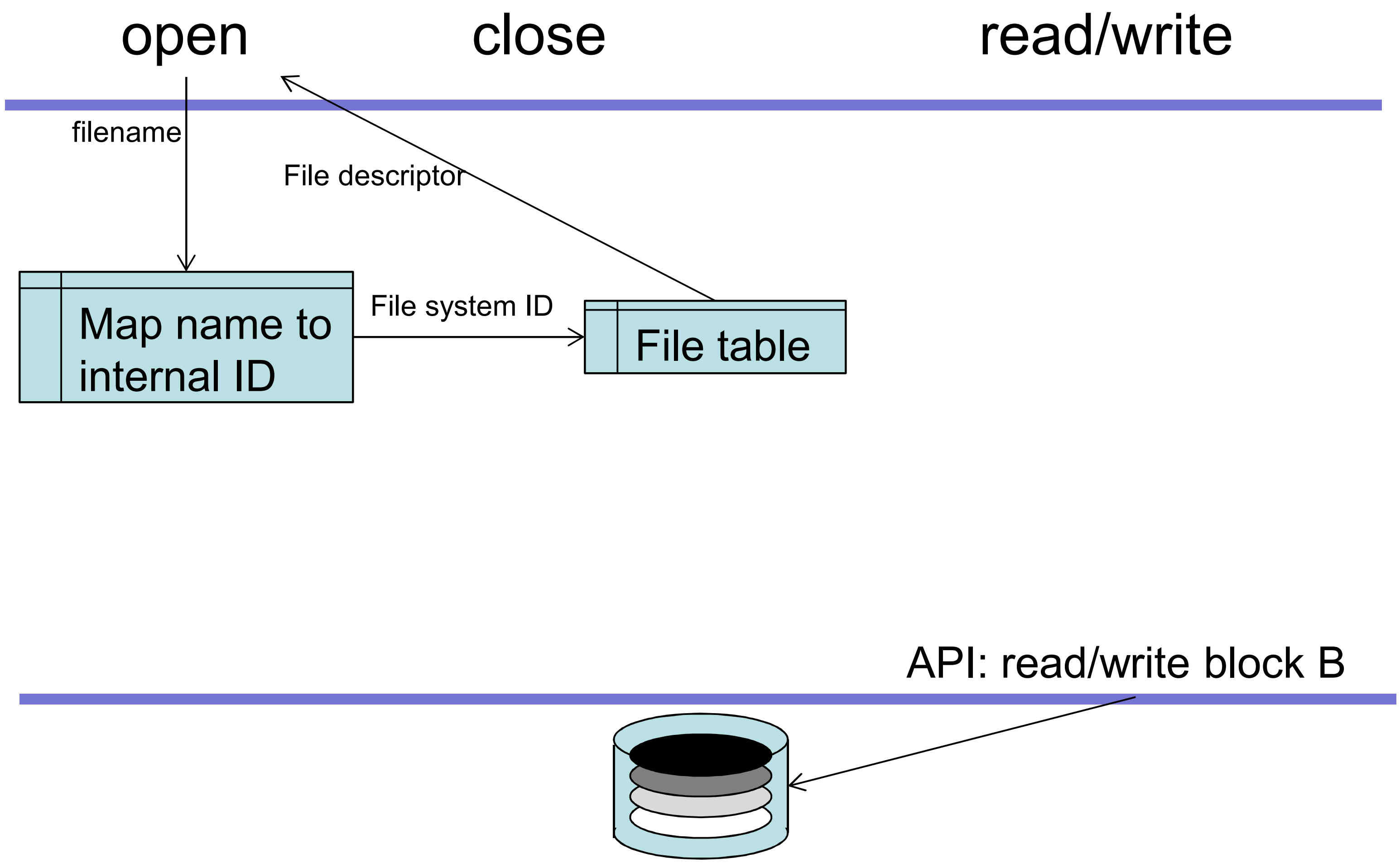
File Systems: Introduction

- Learning Objective
  - Describe the layers of software between the file system system call API and the disk.
  - Decompose those layers in a collection of independent problems.
  - Derive solutions to the key problems of:
    - File representation
    - Naming & Name Spaces
    - Disk Allocation
    - Recovery
- Topics:
  - From “open/close/read/write” to spinning media.
  - File representation
  - Naming
  - Allocation

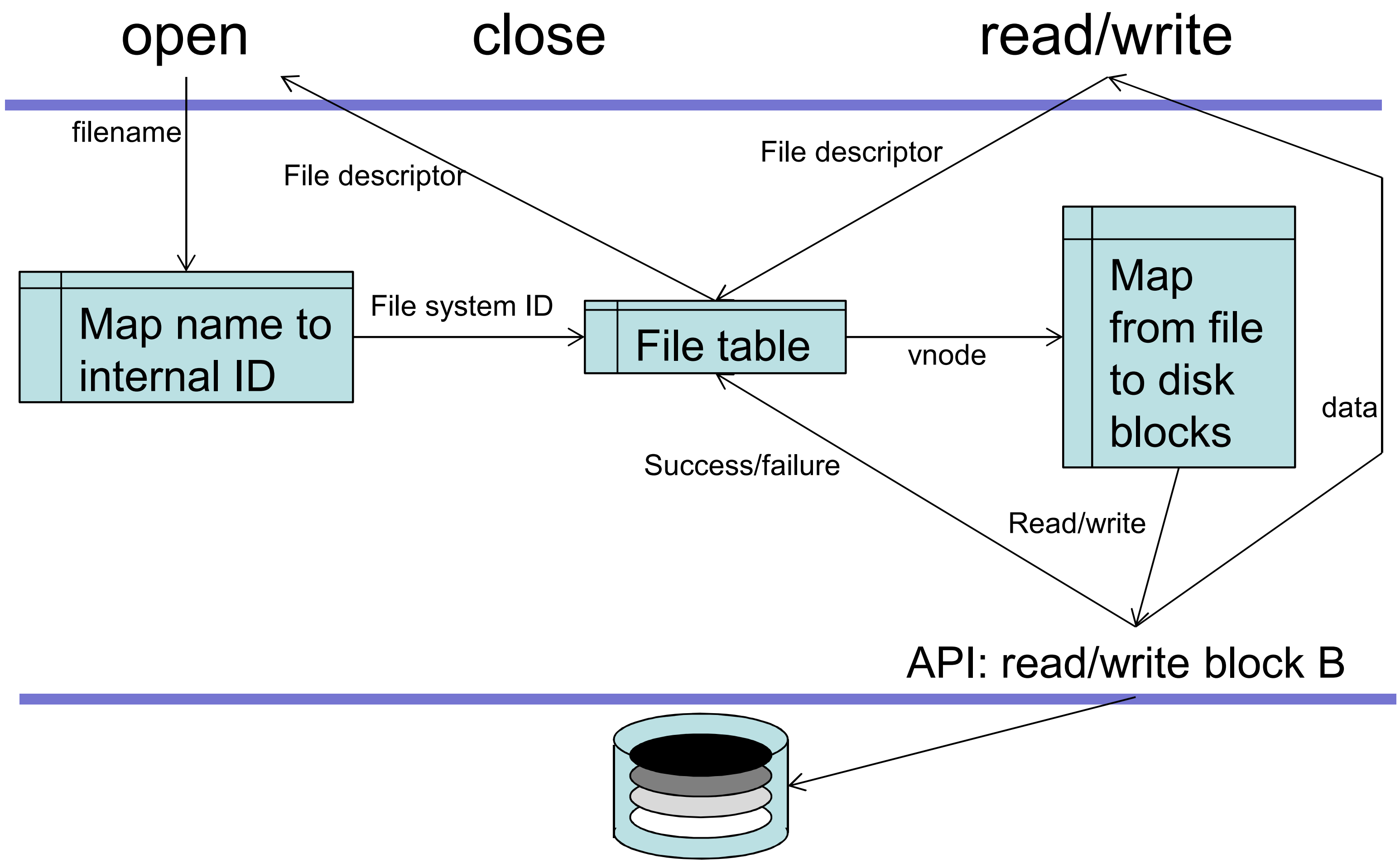
From Syscall API to Disk



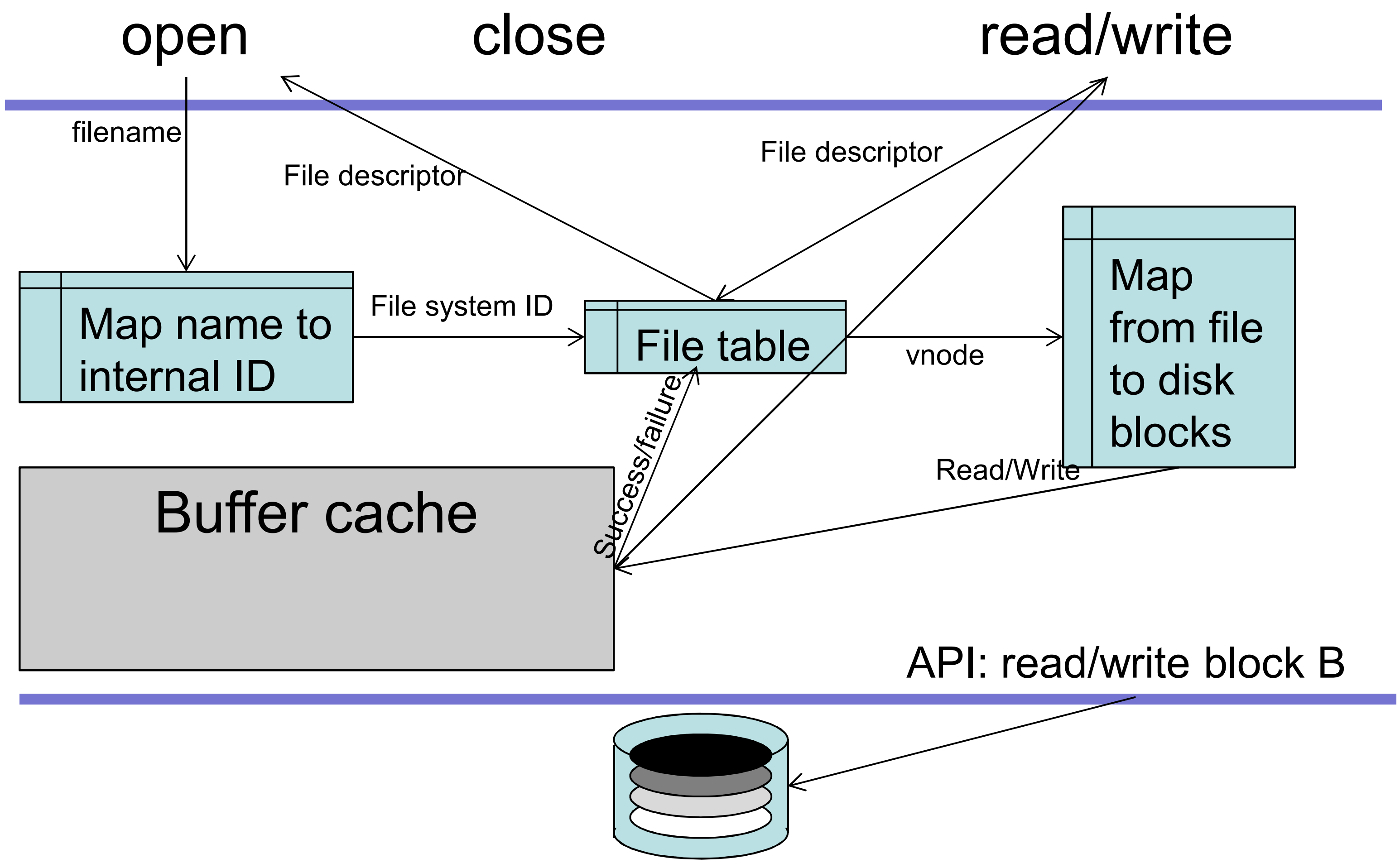
From Syscall API to Disk



From Syscall API to Disk



From Syscall API to Disk



Components of a File System

- Directory: maps names to internal IDs
- File table: keeps track of file state
- File index: maps from a file to a collection of disk blocks (in UNIX systems, this is an inode)
- Buffer cache: keeps copies of recently used blocks in memory.
- What kinds of design parameters are likely to be important?
  - Transfer sizes: how much do you move to/from disk?
  - Allocation size: in what unit to you allocate disk blocks?
  - Placement: Where do you place files on disk?

Exercise 1: File Representation

- How might you represent a file (i.e., design a file index/inode structure)?
  - Must support sequential and random access to a file.
  - Must be reasonably efficient.
  - Address the following two questions:
    1. In what size pieces will you allocate disk space to files?
    2. What metadata (data that describes the data) do you need?
  - Questions to think about:
    - Where will you store metadata?
    - What is the ratio of metadata to data for your representation?
    - What kind of *internal fragmentation* can your representation support?
    - What are the advantages/disadvantages of the approach you picked?

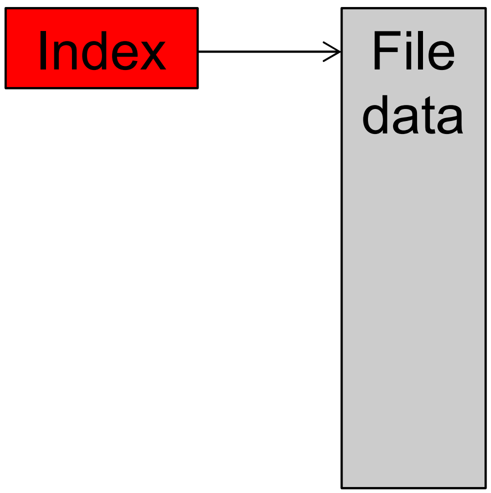
Allocation Units

- Allocation units
  - Fixed sized block
  - A small number of fixed size blocks
  - Variable sizes blocks (called extents)
- Tradeoffs:
  - Fixed size make allocation much easier!
  - Extents can represent files very efficiently
  - A few sizes sounds like a potential compromise

File Representation

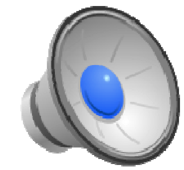
- Single extent: Metadata is a single address (and perhaps a length)
- A small (fixed) number of extents: Metadata is a few disk addresses (perhaps with length)
- File is a large number of blocks
  - Put blocks together in a linked list: metadata is an address
  - Build a large flat index: Metadata is a large array of one address per block/extent
  - Build a multi-level index (like a multi level page table)

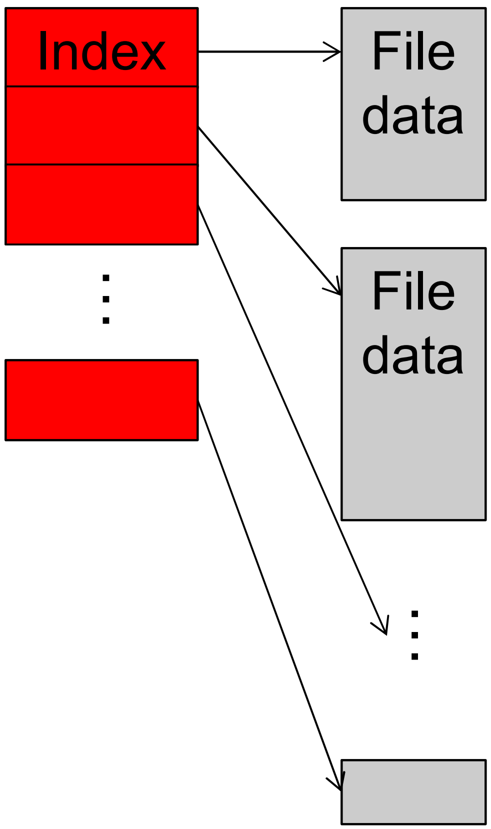
File Representation: Single Extent



- Pros:
  - Dead simple
  - Good for both sequential and random access
  - Very efficient
- Cons:
  - Inflexible – what happens if a file changes size?
  - Have to pre allocate space at create time?
  - Dynamic memory management – lots of external fragmentation

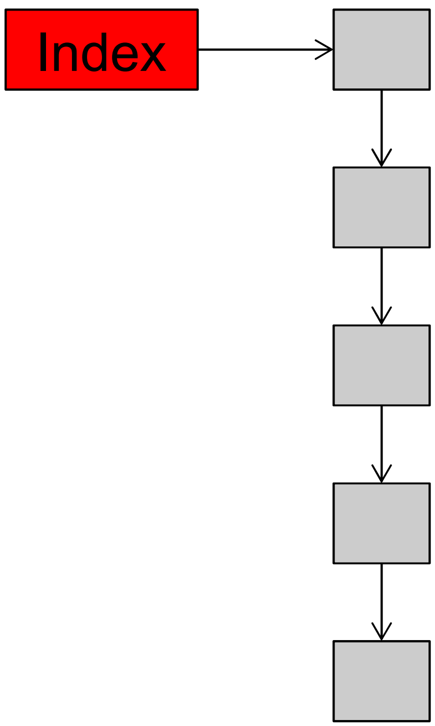
File Representation: A Few Extents





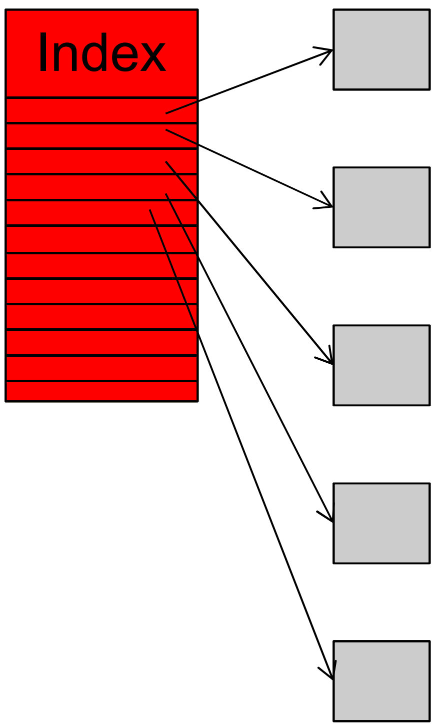
- Pros:
  - If extents are large you get good disk bandwidth
  - Both sequential and random are good (you have to do some work for random)
  - Meta data is small
- Cons:
  - Lots of design decisions
    - How big are extents?
    - How do you decide?
  - How do you grow files?
  - External fragmentation
  - Depending on answers to design, might have internal fragmentation
  - Could end up with a file too big to represent

File Representation: Linked Blocks



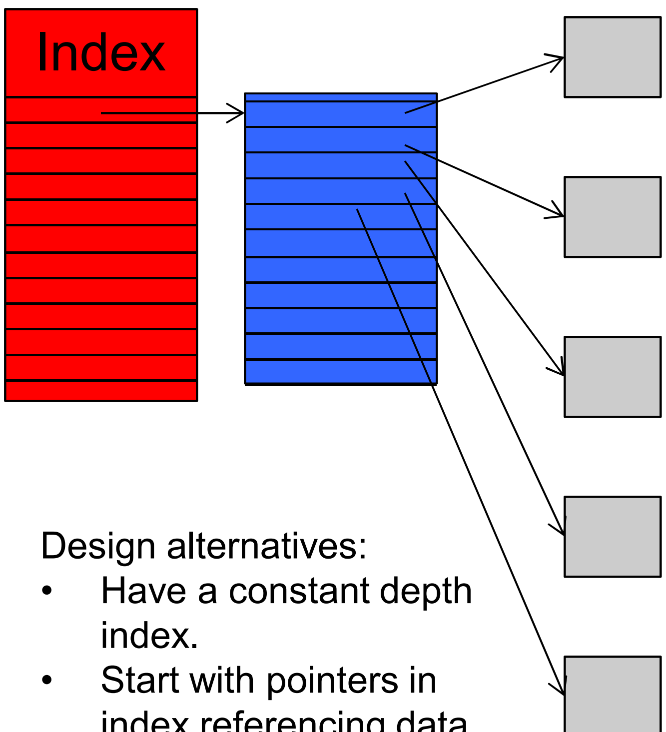
- Pros:
  - Files can be extended easily.
  - Don't have to worry about fragmentation.
  - Sequential access is easy.
- Cons:
  - Random access is virtually impossible.
  - Even sequential access requires lots of seeks.
  - Can't do read-ahead.

File Representation: Flat Index



- Pros:
  - Still no external fragmentation
  - Sequential and random access are easy.
- Cons:
  - How big do we make the index?
  - Do we pre-allocate the entire index?
  - Have we made our metadata (i.e., the inode) variable-sized?
  - We still may have a lot of seeks between blocks.

File Representation: Multi-level Index

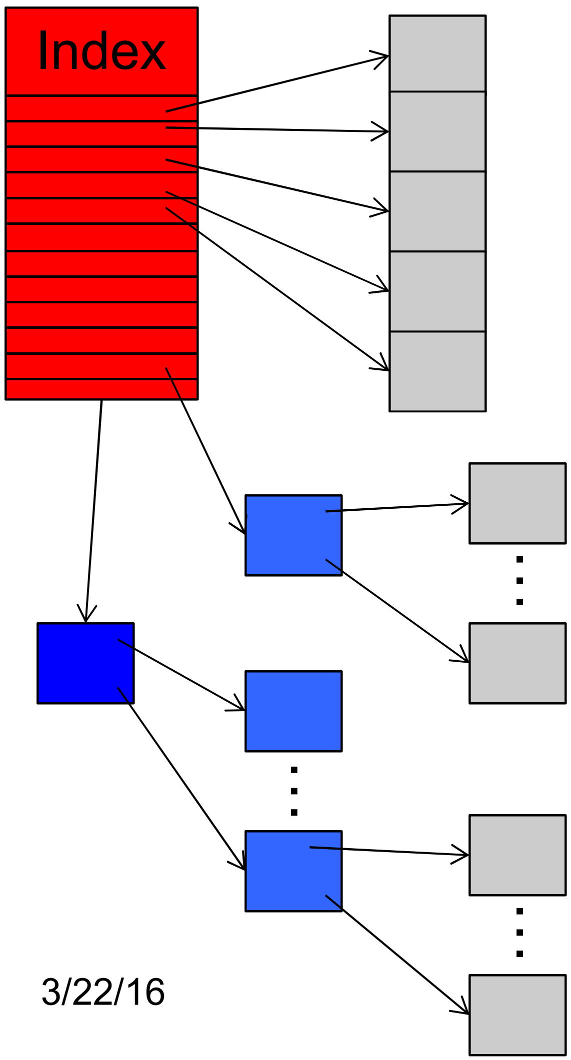


Design alternatives:

- Have a constant depth index.
- Start with pointers in index referencing data (direct pointers). When that fills, add first level of indirection and copy pointers, repeat.

- Pros:
  - Simple
  - Easy to represent really big files.
  - The changing-depth design is space-efficient for both small and large files.
- Cons:
  - The constant-depth solution is inefficient for small files.
  - Accessing large files will require multiple block accesses.
  - May require seeks between reads of adjacent blocks.

File Representation: Hybrid Index

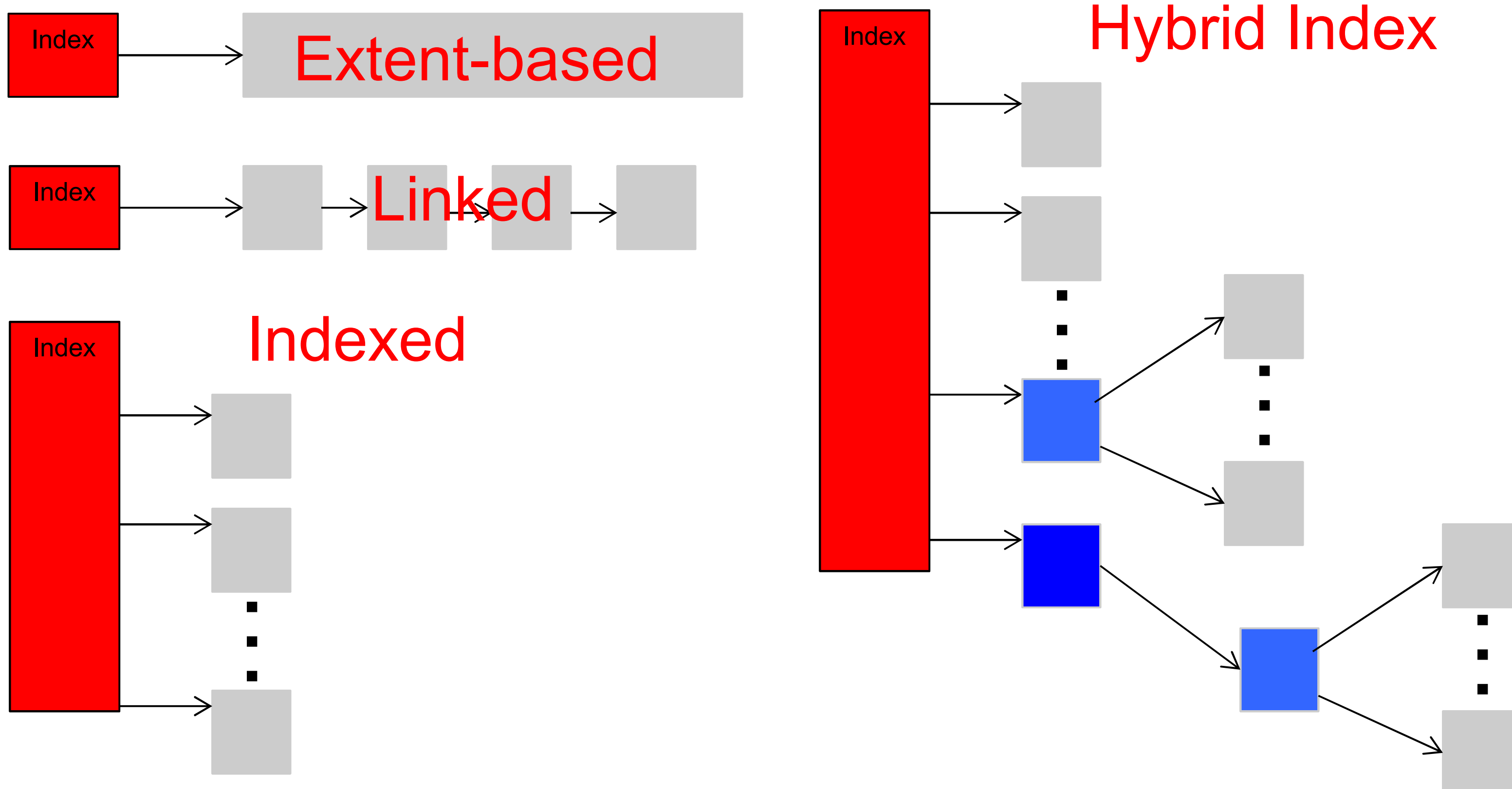


- Pros:
  - Simple
  - No wasted space for unallocated blocks.
  - Efficient for small files.
  - Although there is a maximum file size, it’s really big
- Cons:
  - Multiple block reads on large files.
  - May require seeks between reads of adjacent blocks.

Fixing the existing problems

- The buffer cache essentially solves the “multiple blocks to read for big files” problem.
- Intelligent placement and delayed allocation solve the “seek between blocks” problem.

File Structure Summary



Exercise 2: Free Space Management

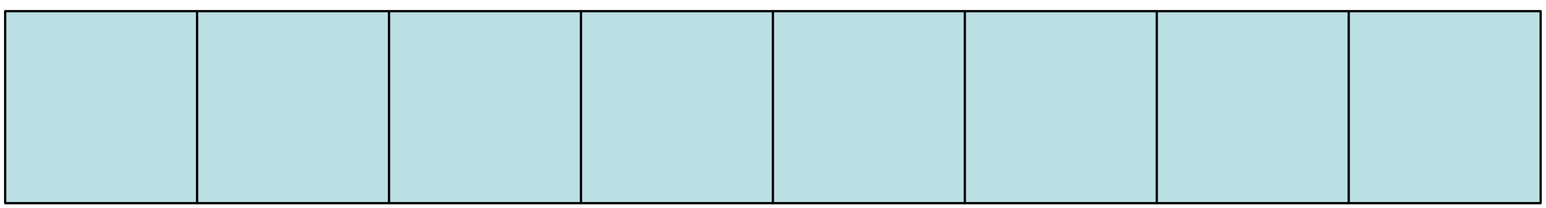
- Assume you allocate in fixed size blocks:
  - How do you keep track of free space?
  - How do you select which blocks to allocate to a particular file?
- Assume that you allocate variable size extents:
  - How do you select the extent size?
  - How do you manage free space?
  - Where do you allocate extents?

Free Space Management (1)

- There is often a tradeoff between the amount of (allocation) meta data you keep and the quality of allocation.
- Fixed size blocks:
  - Free list: link all the free pages together in a list (placing the pointer on the actual page).
    - Metadata: One pointer (excellent).
    - Ease of allocation: Pull first block off the list (excellent).
    - Ability to produce good (e.g., contiguous) allocations? Poor.
  - Bitmaps
    - Metadata: One bit per block (good)
    - Ease of allocation: Find a free bit (good)
    - Ability to produce good allocations? (good)
- How do these apply to a small number of block sizes?

Buddy Allocation

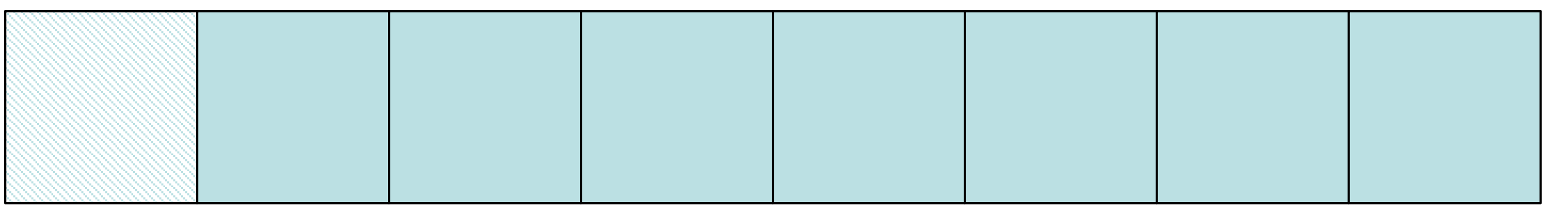
- One way to support multiple block sizes is to make all the sizes be a power-of-two multiple of a basic block size.
- Rather than assign disk blocks to different sized file system blocks haphazardly, create blocks of size 2^N by splitting a block of size 2^{N+1}



1. Disk is collection of maximum size blocks

Buddy Allocation

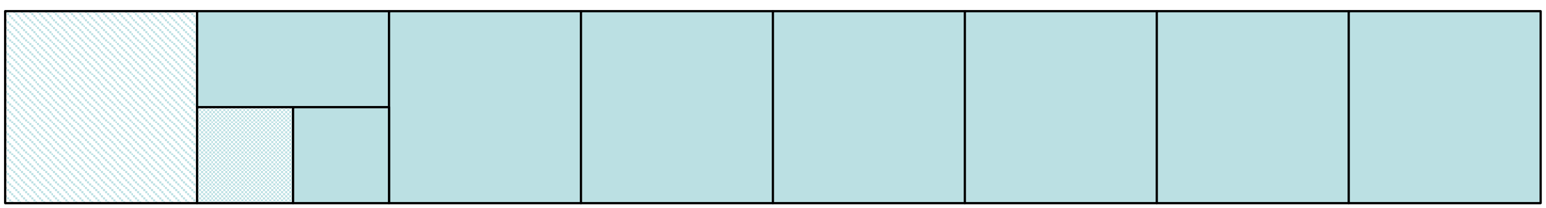
- One way to support multiple block sizes is to make all the sizes be a power-of-two multiple of a basic block size.
- Rather than assign disk blocks to different sized file system blocks haphazardly, create blocks of size 2^N by splitting a block of size 2^{N+1}



2. Allocate a large block.

Buddy Allocation

- One way to support multiple block sizes is to make all the sizes be a power-of-two multiple of a basic block size.
- Rather than assign disk blocks to different sized file system blocks haphazardly, create blocks of size 2^N by splitting a block of size 2^{N+1}



3. Allocate minimum-sized block.

Free Space Management (2)

- Extents
  - On-disk malloc (free list approach)
    - Keep free extents in lists, tagged with size
    - Or, like a slab allocator, have multiple lists with different-sized blocks
    - Metadata: one or a few pointers (excellent)
    - Ease of allocation: pretty good
    - Problems? Fragmentation (both internal and external)
  - Bitmap based: probably need to track in some primitive unit size
    - Metadata: one bit per primitive unit (good)
    - Ease of allocation: not great – need to search for contiguous chunks.