Systems and Networking – Unit I

B.Sc. in Applied Computer Science and Artificial Intelligence

2021-2022

Gabriele Tolomei

Department of Computer Science

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

File System's Logical View

File System API	File creation, manipulation, protection, etc.
OS Implementation	OS internal data structures and algorithms
Physical Implementation	Second storage structure, disk scheduling algorithms

File System's Logical View

File System API	File creation, manipulation, protection, etc.
OS Implementation	OS internal data structures and algorithms
Physical Implementation	Second storage structure, disk scheduling algorithms

File System Implementation

How do we actually lay down data on disk?

Recap: Disk Overheads

- Overhead: the time the CPU (or the DMA controller) takes to start a disk operation

Recap: Disk Overheads

- Overhead: the time the CPU (or the DMA controller) takes to start a disk operation
- Latency: the time to initiate a disk transfer of 1 byte to memory
  - Seek time → the time to position the head over the correct cylinder
  - Rotational time → the time for the correct sector to rotate under the head

Recap: Disk Overheads

- **Overhead:** the time the CPU (or the DMA controller) takes to start a disk operation
- **Latency:** the time to initiate a disk transfer of 1 byte to memory
  - Seek time → the time to position the head over the correct cylinder
  - Rotational time → the time for the correct sector to rotate under the head
- **Bandwidth:** once a transfer is initiated, the rate of the I/O transfer

File Organization on Disk

- From the OS's perspective:
  - Disk is just an array of blocks

File Organization on Disk

- From the OS's perspective:
  - Disk is just an array of blocks
- We can think of a block as a disk sector
  - In practice, a block may be a multiple of a sector (e.g., 4 sectors)
  - Typical block size ranges from 512B to 4KiB or larger

File Organization on Disk

- From the OS's perspective:
  - Disk is just an array of blocks
- We can think of a block as a disk sector
  - In practice, a block may be a multiple of a sector (e.g., 4 sectors)
  - Typical block size ranges from 512B to 4KiB or larger
- How it should work:
  - The OS requests for **`fileID 42`**, **`block 73`** (contiguous integer addressing)
  - The disk responds with the corresponding (**`head`**, **`cylinder`**, **`sector`**) triple

File Organization on Disk

- Disk Access:
  - Must be able to support both sequential and direct/random access

File Organization on Disk

- Disk Access:
  - Must be able to support both sequential and direct/random access
- File information on disk:
  - Data structure to maintain file location information

File Organization on Disk

- Disk Access:
  - Must be able to support both sequential and direct/random access
- File information on disk:
  - Data structure to maintain file location information
- File location on disk:
  - Physically deploy file on disk

On-Disk Data Structures: Overview

- A boot control block (per volume)
- Contains information about how to boot up the system
- Also known as boot block in UNIX and partition boot sector in Windows
- Generally, the first sector of the volume if there is a bootable system loaded on that volume

On-Disk Data Structures: Overview

- A volume control block (per volume)
- Contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks
- Also known as the master file table in UNIX or the superblock in Windows

On-Disk Data Structures: Overview

- A directory structure (per file system)
- Contains file names and pointers to FCBs
- UNIX uses inode numbers, and NTFS uses a master file table

On-Disk Data Structures: Overview

- The File Control Block (FCB) (per file)
- Contains details about file ownership, size, permissions, dates, etc.
- UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure

On-Disk Data Structures: File Control Block

- Per-file data structure used to describe where the file is located on disk

On-Disk Data Structures: File Control Block

- Per-file data structure used to describe where the file is located on disk
- Contains also file attributes (i.e., file metadata)

On-Disk Data Structures: File Control Block

- Per-file data structure used to describe where the file is located on disk
- Contains also file attributes (i.e., file metadata)
- Must be stored on disk as regular files

On-Disk Data Structures: File Control Block

- Per-file data structure used to describe where the file is located on disk
- Contains also file attributes (i.e., file metadata)
- Must be stored on disk as regular files
- Also known as File Descriptor (FD)

On-Disk Data Structures: File Control Block

- Per-file data structure used to describe where the file is located on disk
- Contains also file attributes (i.e., file metadata)
- Must be stored on disk as regular files
- Also known as File Descriptor (FD)
- A copy of each FCB is stored also in the OS's Global Open File Table

On-Disk Data Structures: File Control Block

- Per-file data structure used to describe where the file is located on disk
- Contains also file attributes (i.e., file metadata)
- Must be stored on disk as regular files
- Also known as File Descriptor (FD)
- A copy of each FCB is stored also in the OS's Global Open File Table
- Called inode in Linux

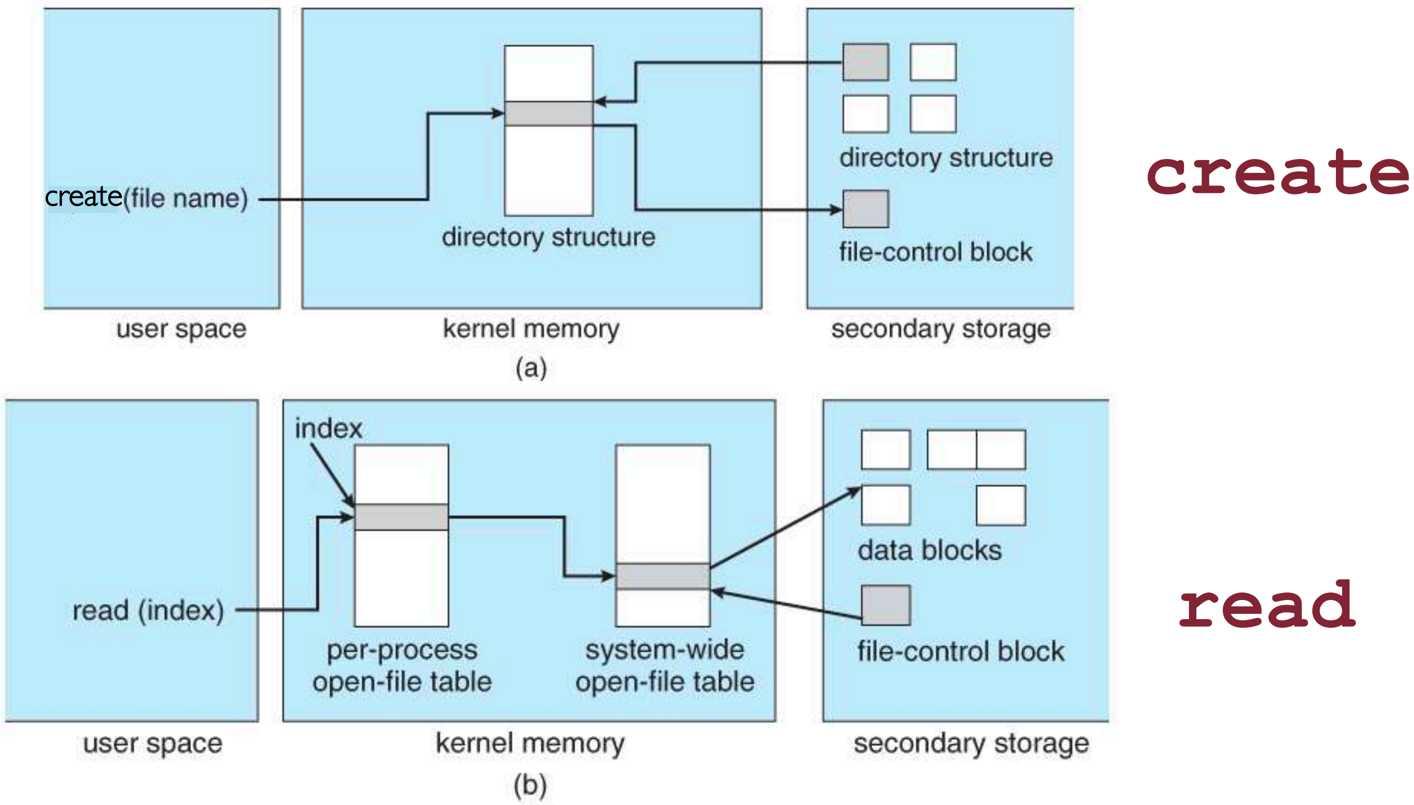
On-Disk Data Structures: File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

In-Memory Data Structures: Overview

- In-memory mount table
- In-memory directory cache of recently accessed directory information
- The global (i.e., system-wide) open file table, containing a copy of the FCB for every currently open file in the system, plus other related information
- A local (i.e., per-process) open file table, containing a pointer to the system open file table as well as some other information

Wrapping Things Up!



Directory Implementation

- Need to be fast to search, insert, and delete, with a minimum of wasted disk space

Directory Implementation

- Need to be fast to search, insert, and delete, with a minimum of wasted disk space
- Linear list: simplest yet inefficient
    - searching requires a linear scan
    - linked list makes insertion/deletion into a sorted list easier but needs to keep track of pointers (more complex structures like B-trees can be used)

Directory Implementation

- Need to be fast to search, insert, and delete, with a minimum of wasted disk space
- Linear list: simplest yet inefficient
  - searching requires a linear scan
  - linked list makes insertion/deletion into a sorted list easier but needs to keep track of pointers (more complex structures like B-trees can be used)
- Hashtable: usually implemented in addition to a linear structure

File Allocation Methods: Considerations

- Most files in a system are typically very small

File Allocation Methods: Considerations

- Most files in a system are typically very small
- The vast majority of disk space is taken up by few but very large files

File Allocation Methods: Considerations

- Most files in a system are typically very small
- The vast majority of disk space is taken up by few but very large files
- Disk I/O operations target both small and large files

File Allocation Methods: Considerations

- Most files in a system are typically very small
- The vast majority of disk space is taken up by few but very large files
- Disk I/O operations target both small and large files
- Per-file cost must be low (and large files must be handled efficiently)

Option 1: Contiguous Allocation

- Sounds familiar with how basic memory allocation is done

Option 1: Contiguous Allocation

- Sounds familiar with how basic memory allocation is done
- The OS keeps track of a list of free disk blocks

Option 1: Contiguous Allocation

- Sounds familiar with how basic memory allocation is done
- The OS keeps track of a list of free disk blocks
- When a file is created the OS allocates a sequence of free blocks

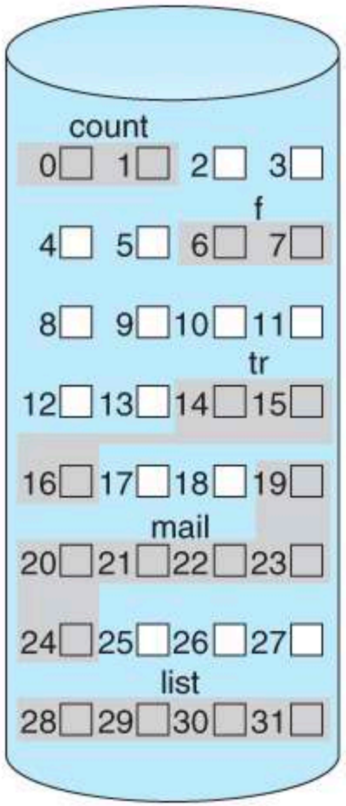
Option 1: Contiguous Allocation

- Sounds familiar with how basic memory allocation is done
- The OS keeps track of a list of free disk blocks
- When a file is created the OS allocates a sequence of free blocks
- File descriptor needs only to store the start location and size

Option 1: Contiguous Allocation

- Sounds familiar with how basic memory allocation is done
- The OS keeps track of a list of free disk blocks
- When a file is created the OS allocates a sequence of free blocks
- File descriptor needs only to store the start location and size
- Examples: IBM/360, write-once disks, early PCs

Option 1: Contiguous Allocation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation: PROs and CONs

- PROs:
  - Very simple
  - Best possible choice for sequential access (only 1 disk seek) and random access (1 disk seek + rotational time to get to the correct block)

Contiguous Allocation: PROs and CONs

- PROs:
  - Very simple
  - Best possible choice for sequential access (only 1 disk seek) and random access (1 disk seek + rotational time to get to the correct block)
- CONs:
  - Hard to change file size (may need to re-allocate it entirely to another location)
  - Fragmentation (may need to run compaction/defragmentation)

Option 2: Linked Files

- The OS keeps a linked list of free (not necessarily contiguous) blocks

Option 2: Linked Files

- The OS keeps a linked list of free (not necessarily contiguous) blocks
- The OS keeps also a linked list of where subsequent blocks are located

Option 2: Linked Files

- The OS keeps a linked list of free (not necessarily contiguous) blocks
- The OS keeps also a linked list of where subsequent blocks are located
- This frees the file to be physically located sequentially

Option 2: Linked Files

- The OS keeps a linked list of free (not necessarily contiguous) blocks
- The OS keeps also a linked list of where subsequent blocks are located
- This frees the file to be physically located sequentially
- Keep a pointer to the first block of the file in the file descriptor

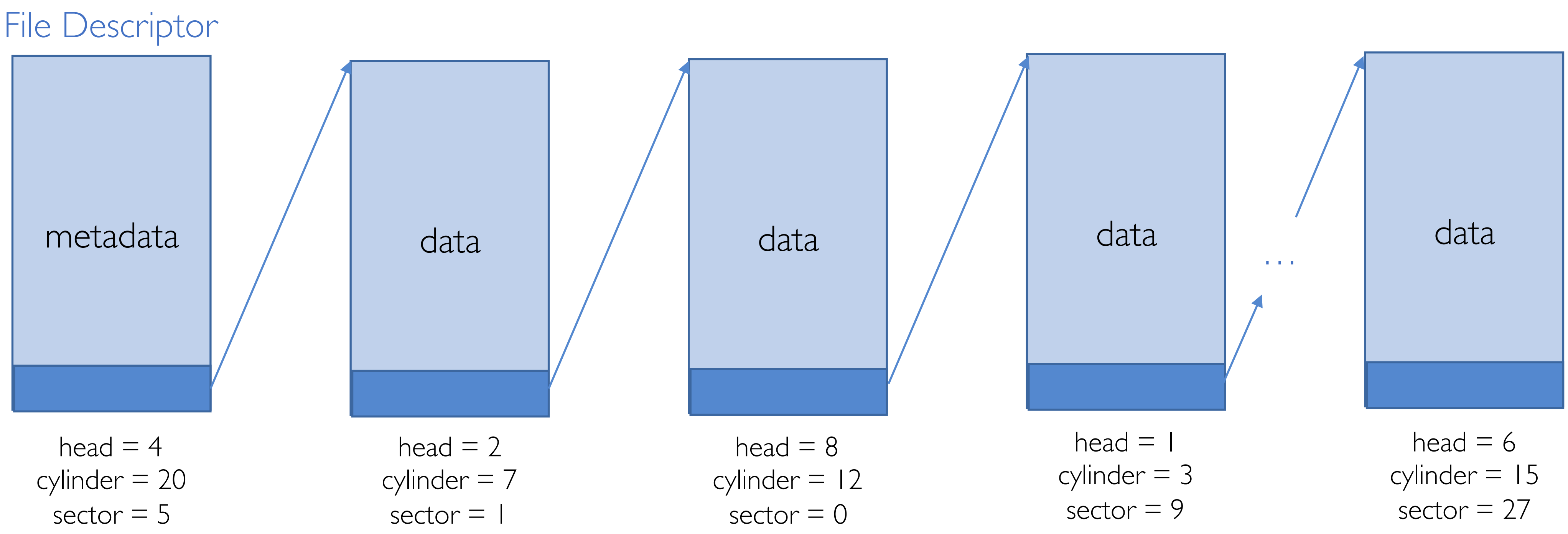
Option 2: Linked Files

- The OS keeps a linked list of free (not necessarily contiguous) blocks
- The OS keeps also a linked list of where subsequent blocks are located
- This frees the file to be physically located sequentially
- Keep a pointer to the first block of the file in the file descriptor
- Keep a pointer to the next block in each sector

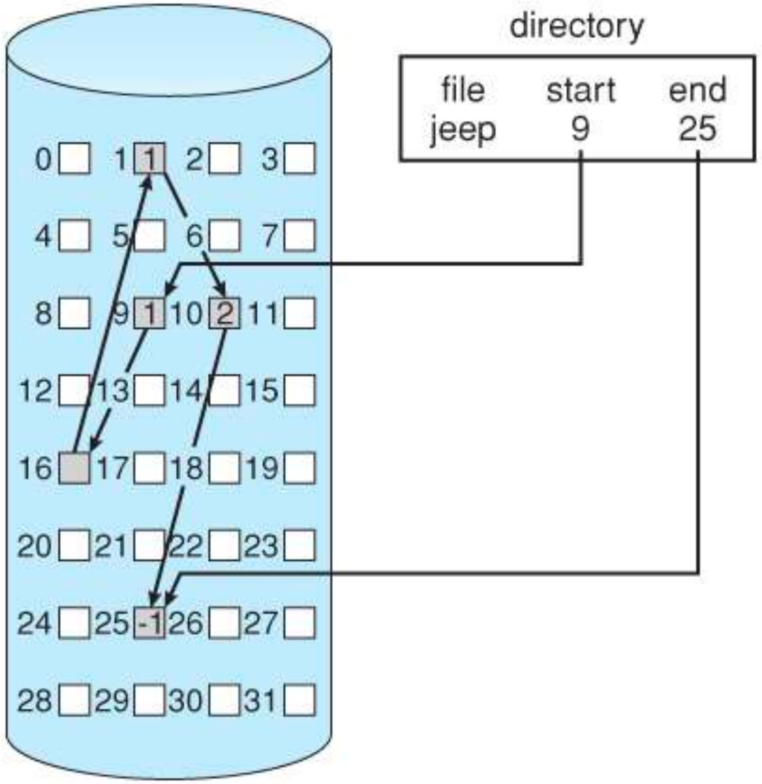
Option 2: Linked Files

- The OS keeps a linked list of free (not necessarily contiguous) blocks
- The OS keeps also a linked list of where subsequent blocks are located
- This frees the file to be physically located sequentially
- Keep a pointer to the first block of the file in the file descriptor
- Keep a pointer to the next block in each sector
- Examples: FAT, MS-DOS

Option 2: Linked Files



Option 2: Linked Files



Linked Files: PROs and CONs

- PROs:
  - No fragmentation
  - File changes is managed very easily (new blocks can be inserted in the list)

Linked Files: PROs and CONs

- PROs:
  - No fragmentation
  - File changes is managed very easily (new blocks can be inserted in the list)
- CONs:
  - Inefficient sequential access: need to traverse the whole linked list (may need *n* seeks + *n* rotational delays for *n*-block files)
  - Inefficient random access: basically, as above (of course the exact cost depends on the specific block referenced)

Option 3: Indexed Files

- The file descriptor contains a block of pointers (vs. only 1 pointer as in the linked list approach)

Option 3: Indexed Files

- The file descriptor contains a block of pointers (vs. only 1 pointer as in the linked list approach)
- The user or OS must declare the maximum length of the file when it is created

Option 3: Indexed Files

- The file descriptor contains a block of pointers (vs. only 1 pointer as in the linked list approach)
- The user or OS must declare the maximum length of the file when it is created
- OS allocates an array to hold the pointers to all the blocks when it creates the file, but allocates the blocks only on demand

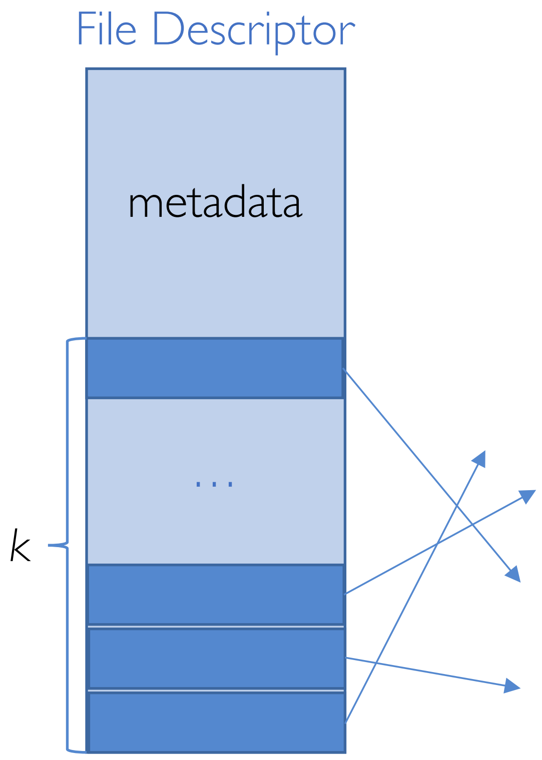
Option 3: Indexed Files

- The file descriptor contains a block of pointers (vs. only 1 pointer as in the linked list approach)
- The user or OS must declare the maximum length of the file when it is created
- OS allocates an array to hold the pointers to all the blocks when it creates the file, but allocates the blocks only on demand
- OS fills in the pointers as it allocates blocks

Option 3: Indexed Files

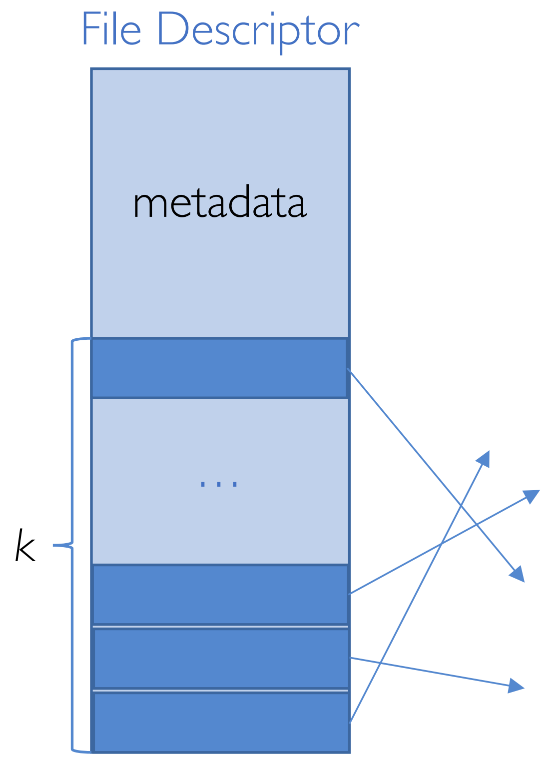
- The file descriptor contains a block of pointers (vs. only 1 pointer as in the linked list approach)
- The user or OS must declare the maximum length of the file when it is created
- OS allocates an array to hold the pointers to all the blocks when it creates the file, but allocates the blocks only on demand
- OS fills in the pointers as it allocates blocks
- Example: Nachos

Option 3: Indexed Files



The number of block pointers *k* determine the maximum file size the system can manage

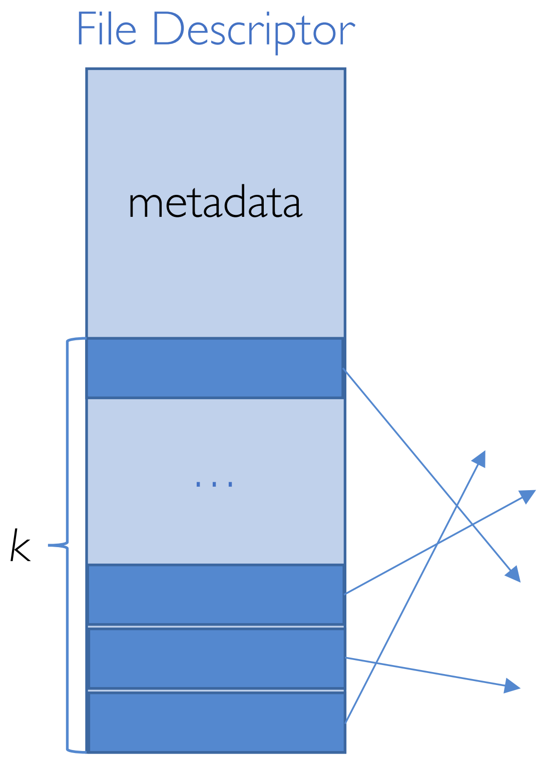
Option 3: Indexed Files



The number of block pointers *k* determine the maximum file size the system can manage

The size of the file descriptor is the same for all files

Option 3: Indexed Files

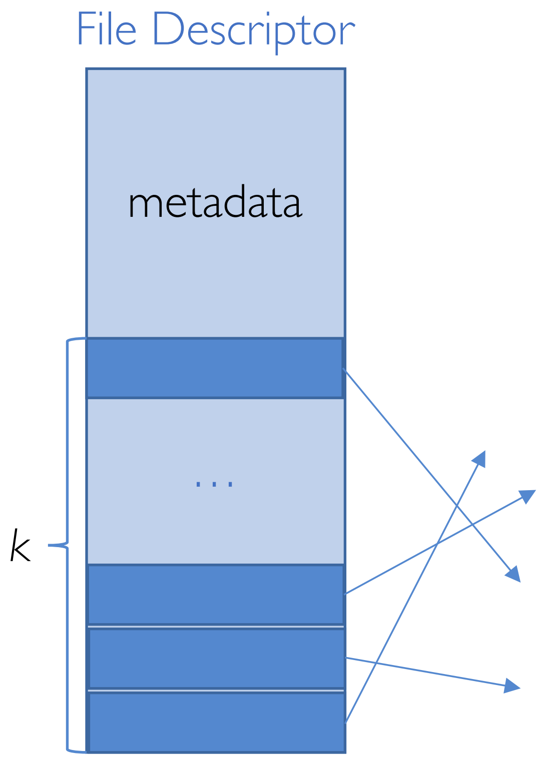


The number of block pointers *k* determine the maximum file size the system can manage

The size of the file descriptor is the same for all files

Remember: most files are small!

Option 3: Indexed Files



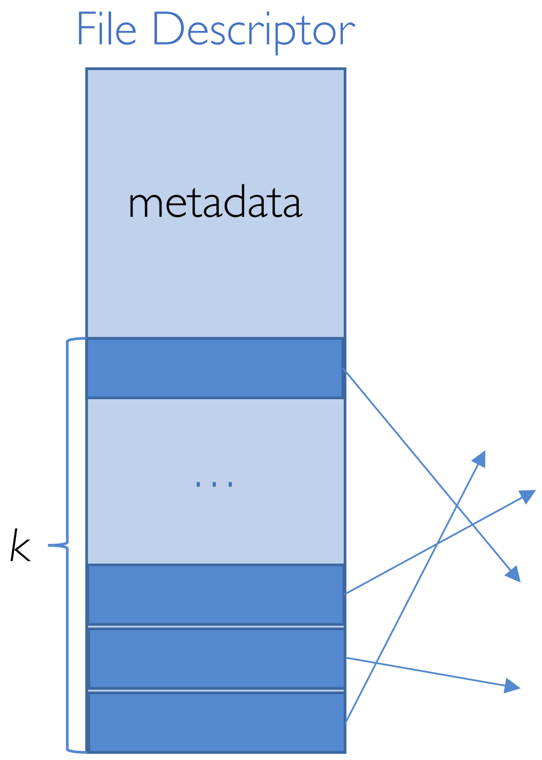
The number of block pointers *k* determine the maximum file size the system can manage

The size of the file descriptor is the same for all files

Remember: most files are small!

The larger the max file size the system is capable to work with, the larger is the space wasted on the file descriptor

Option 3: Indexed Files



The number of block pointers *k* determine the maximum file size the system can manage

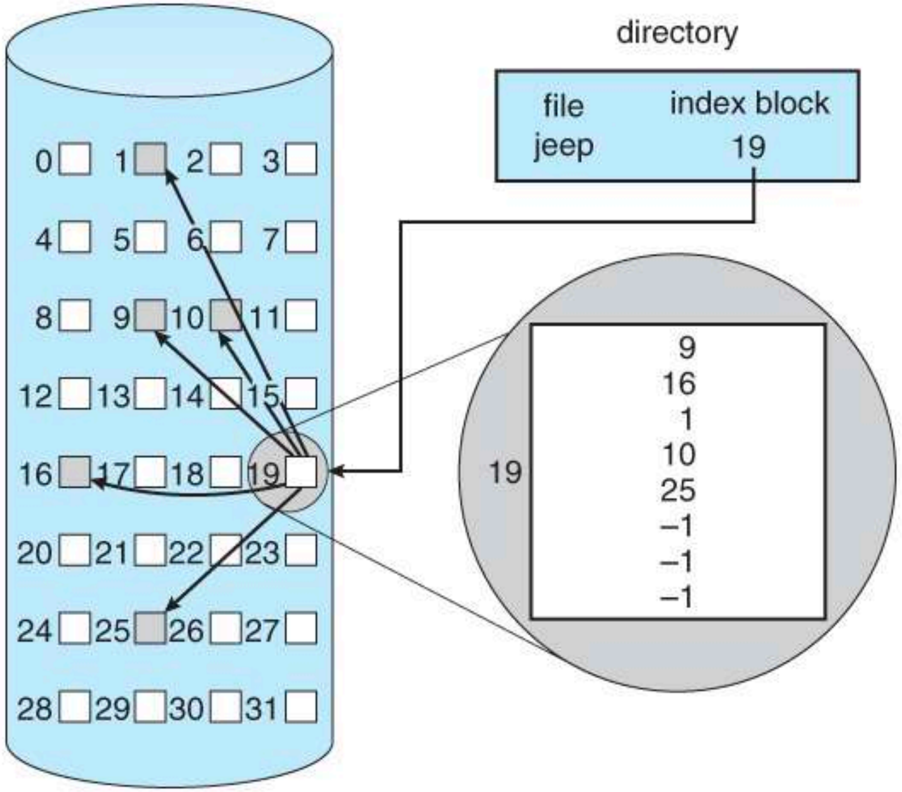
The size of the file descriptor is the same for all files

Remember: most files are small!

The larger the max file size the system is capable to work with, the larger is the space wasted on the file descriptor

Of course, only pointers to blocks are allocated on the file descriptor, not the blocks themselves!

Option 3: Indexed Files



Indexed Files: PROs and CONs

- PROs:
  - No fragmentation
  - Efficient random access: just follow the correct pointer (1 seek + 1 rotation)

Indexed Files: PROs and CONs

- PROs:
  - No fragmentation
  - Efficient random access: just follow the correct pointer (1 seek + 1 rotation)
- CONs:
  - Waste some space on the file descriptor
  - Max file size to be set upfront (things change very quickly!)
  - Inefficient sequential access: as for the linked files approach, it may need n seeks + n rotational delays for n-block files

Option 4: Multi-Level Indexed Files

- Each file descriptor contains a number of block pointers (e.g., 14)

Option 4: Multi-Level Indexed Files

- Each file descriptor contains a number of block pointers (e.g., 14)
- The first 12 of those point to data blocks

Option 4: Multi-Level Indexed Files

- Each file descriptor contains a number of block pointers (e.g., 14)
- The first 12 of those point to data blocks
- The 13th pointer points to another block of, say, 1024 block pointers
  - Each of those pointer points to a specific file data block

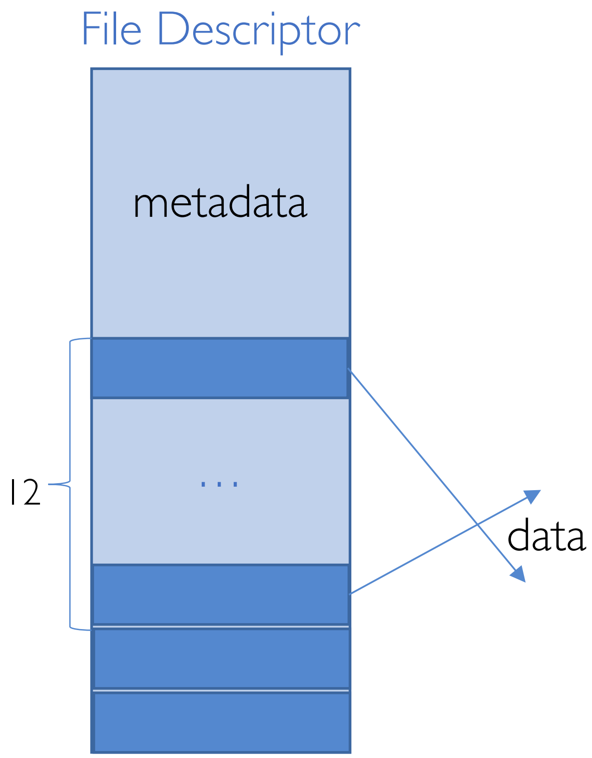
Option 4: Multi-Level Indexed Files

- Each file descriptor contains a number of block pointers (e.g., 14)
- The first 12 of those point to data blocks
- The 13th pointer points to another block of, say, 1024 block pointers
  - Each of those pointer points to a specific file data block
- The 14th pointer points to another block of, say, 1024 pointers
  - Each of those pointer points to, say, 1024 block pointers, which in turn point to file data blocks

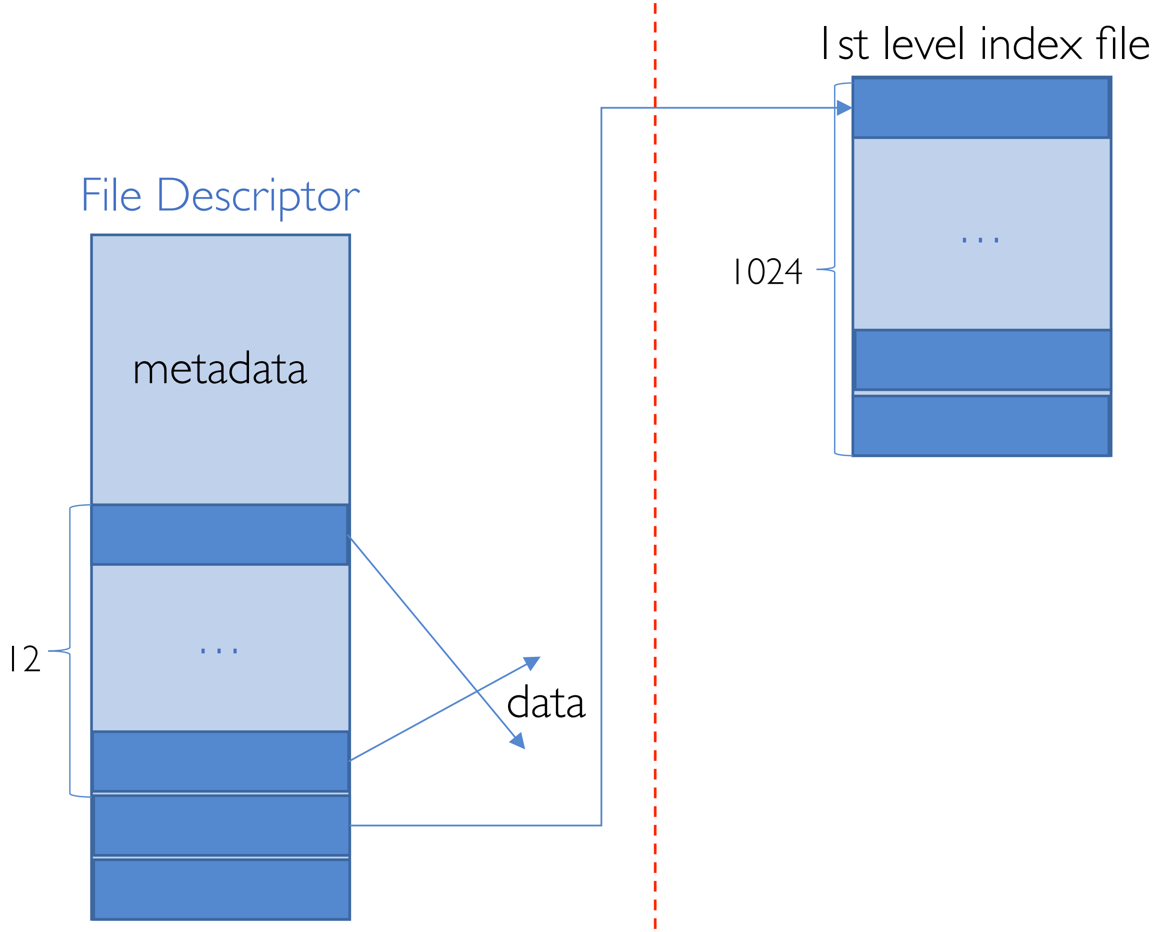
Option 4: Multi-Level Indexed Files

- Each file descriptor contains a number of block pointers (e.g., 14)
- The first 12 of those point to data blocks
- The 13th pointer points to another block of, say, 1024 block pointers
  - Each of those pointer points to a specific file data block
- The 14th pointer points to another block of, say, 1024 pointers
  - Each of those pointer points to, say, 1024 block pointers, which in turn point to file data blocks
- Example: UNIX BSD 4.3

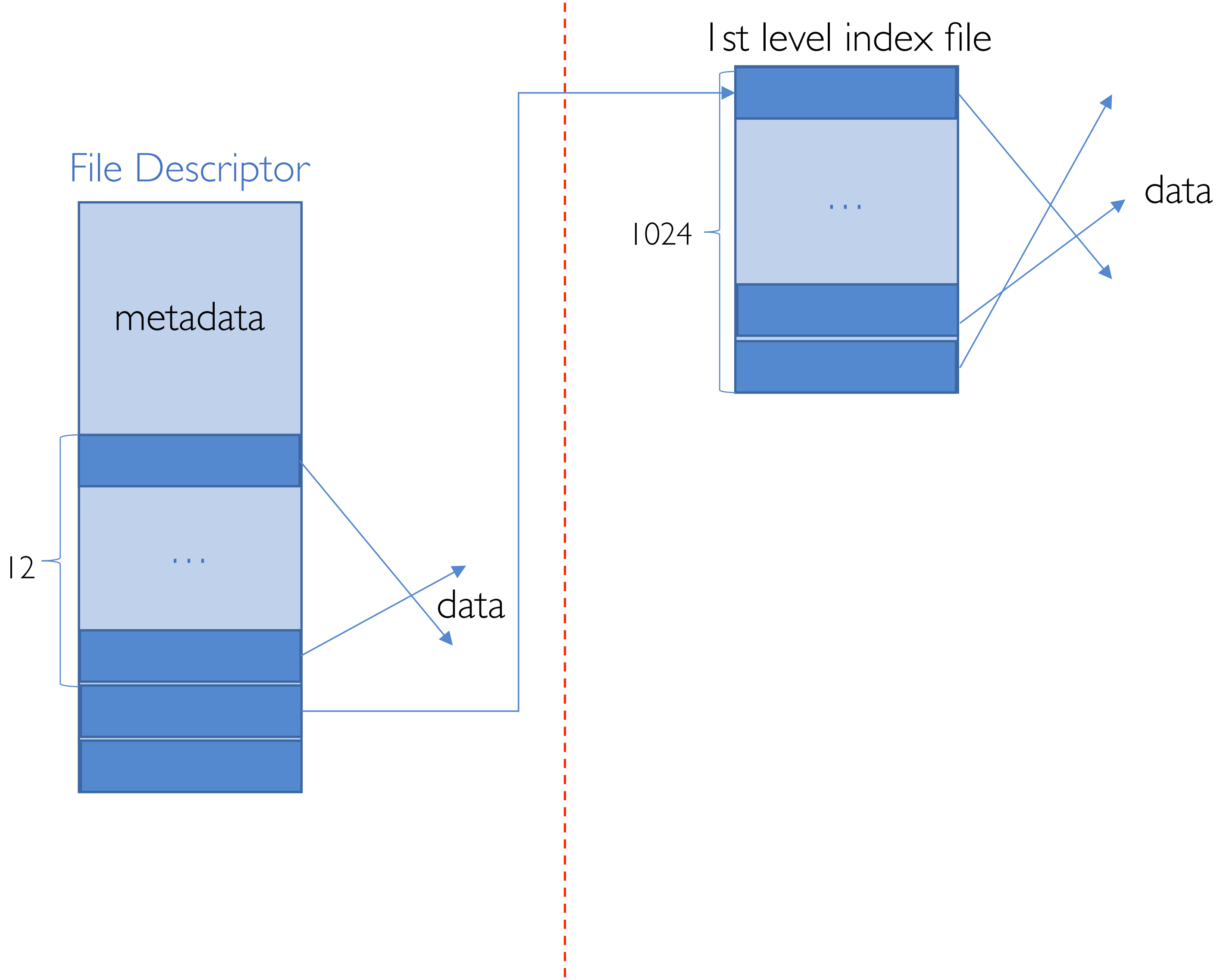
Option 4: Multi-Level Indexed Files



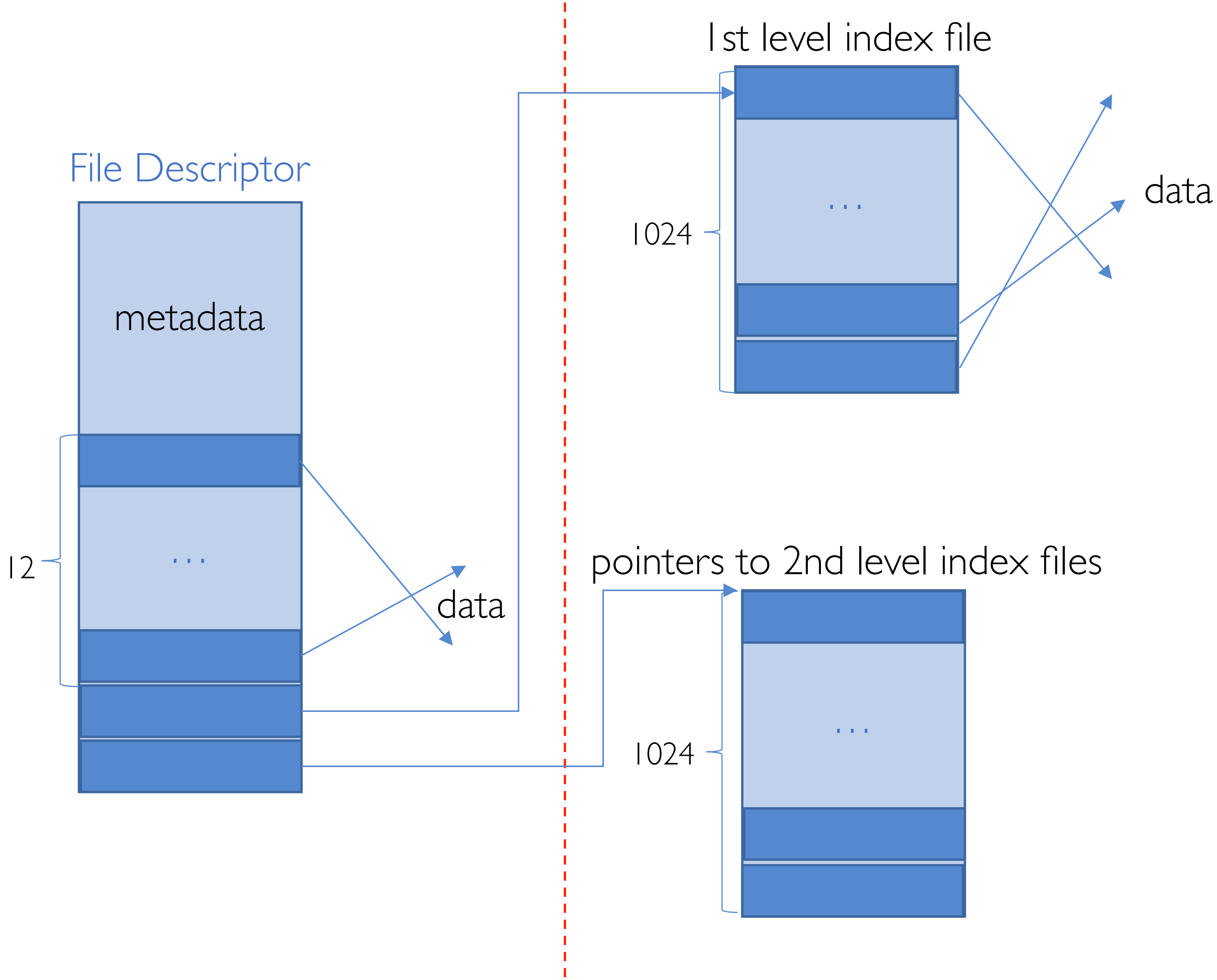
Option 4: Multi-Level Indexed Files



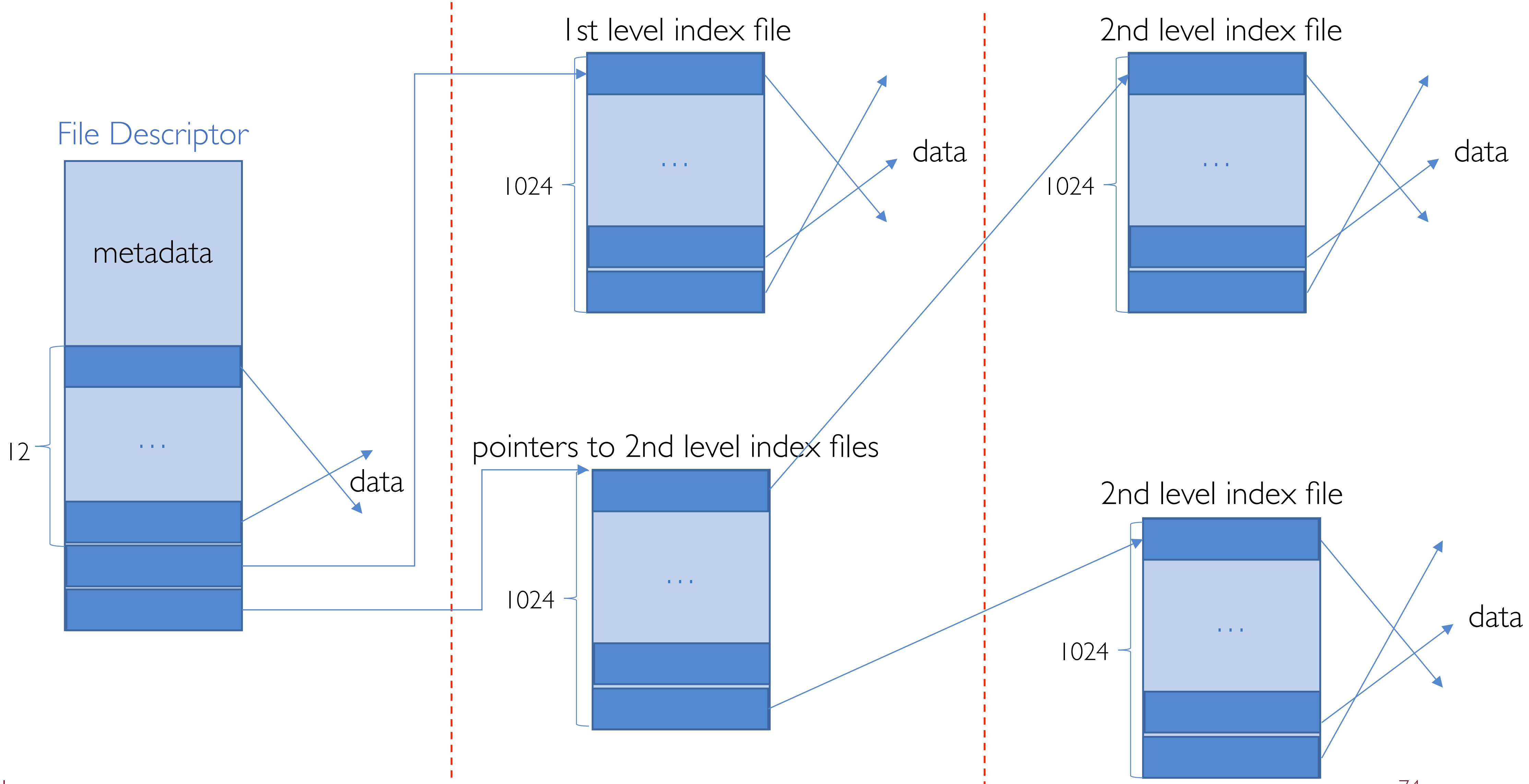
Option 4: Multi-Level Indexed Files



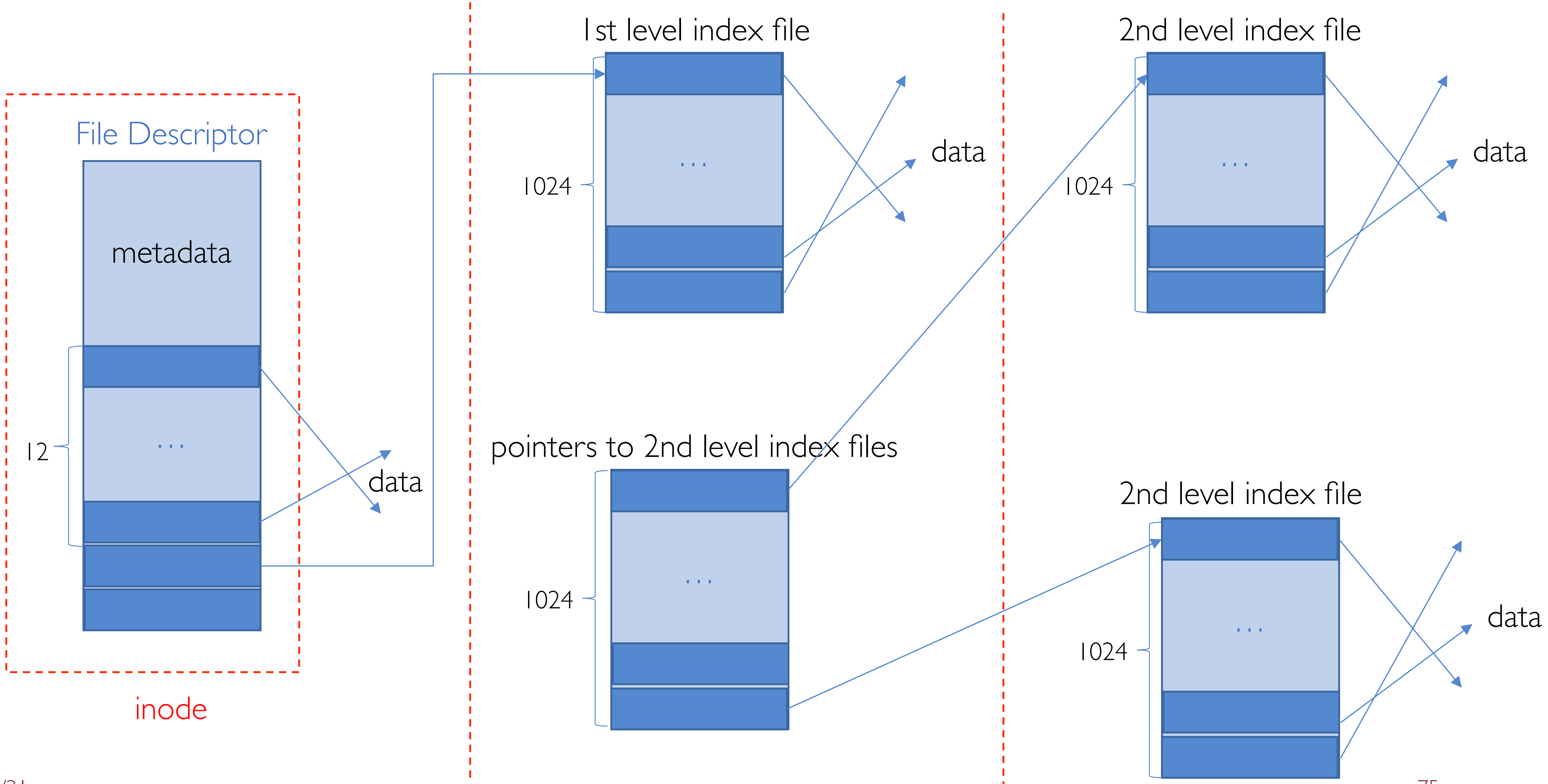
Option 4: Multi-Level Indexed Files



Option 4: Multi-Level Indexed Files



Option 4: Multi-Level Indexed Files



Multi-Level Indexed Files

What's the maximum file size with a 2-level of indirection as above?

Multi-Level Indexed Files

What's the maximum file size with a 2-level of indirection as above?

12 blocks are referenced directly from within the file descriptor

Multi-Level Indexed Files

What's the maximum file size with a 2-level of indirection as above?

12 blocks are referenced directly from within the file descriptor

1024 blocks are referenced from within the 1st level index file

Multi-Level Indexed Files

What's the maximum file size with a 2-level of indirection as above?

12 blocks are referenced directly from within the file descriptor

1024 blocks are referenced from within the 1st level index file

1024^2 blocks are referenced from within the 2nd level indices file

Multi-Level Indexed Files

What's the maximum file size with a 2-level of indirection as above?

12 blocks are referenced directly from within the file descriptor

1024 blocks are referenced from within the 1st level index file

1024^2 blocks are referenced from within the 2nd level indices file

$1024^2 + 1024 + 12 \sim 1\text{MiB}$

Multi-Level Indexed Files

What's the maximum file size with a 2-level of indirection as above?

12 blocks are referenced directly from within the file descriptor

1024 blocks are referenced from within the 1st level index file

1024^2 blocks are referenced from within the 2nd level indices file

$1024^2 + 1024 + 12 \sim 1$MiB

In general, $\sim k^m$ if k = n. of block pointers and m = n. of levels

Multi-Level Indexed Files: PROs and CONs

- PROs:
  - Simple to implement
  - Supports incremental file growth
  - No upper bound to the max file size upfront
  - Optimized for small size files

Multi-Level Indexed Files: PROs and CONs

- PROs:
  - Simple to implement
  - Supports incremental file growth
  - No upper bound to the max file size upfront
  - Optimized for small size files
- CONs:
  - Still inefficient sequential/random access yet better than linked files
  - Lots of seeks because of non-contiguous allocation

Free Space Management: Bitmap

- Need a free-space list to keep track of which disk blocks are free (just as we need for main memory)
- Need to be able to find free space quickly and release space quickly
- The bitmap has one bit for each block on the disk
- If the bit is 1 the block is free, otherwise (0) the block is allocated

Free Space Management: Bitmap

- Use a 32-bit bitmap (i.e., a typical CPU-word size)
- Can quickly determine if any block in the next 32 is free, by comparing the word to 0
- If the bitmap is 0, all the blocks are in use
- Otherwise, use bit operations to find an empty block
- Marking a block as freed is simple since the block number can be used to index into the bitmap to set a single bit

Free Space Management: Bitmap

Problem:

bitmap might become too big to be kept in memory for large disks

Free Space Management: Bitmap

Problem:

bitmap might become too big to be kept in memory for large disks

⇩

How many entries does a 2 TB disk with 512-byte sectors need?

Free Space Management: Bitmap

<u>Problem:</u>

bitmap might become too big to be kept in memory for large disks

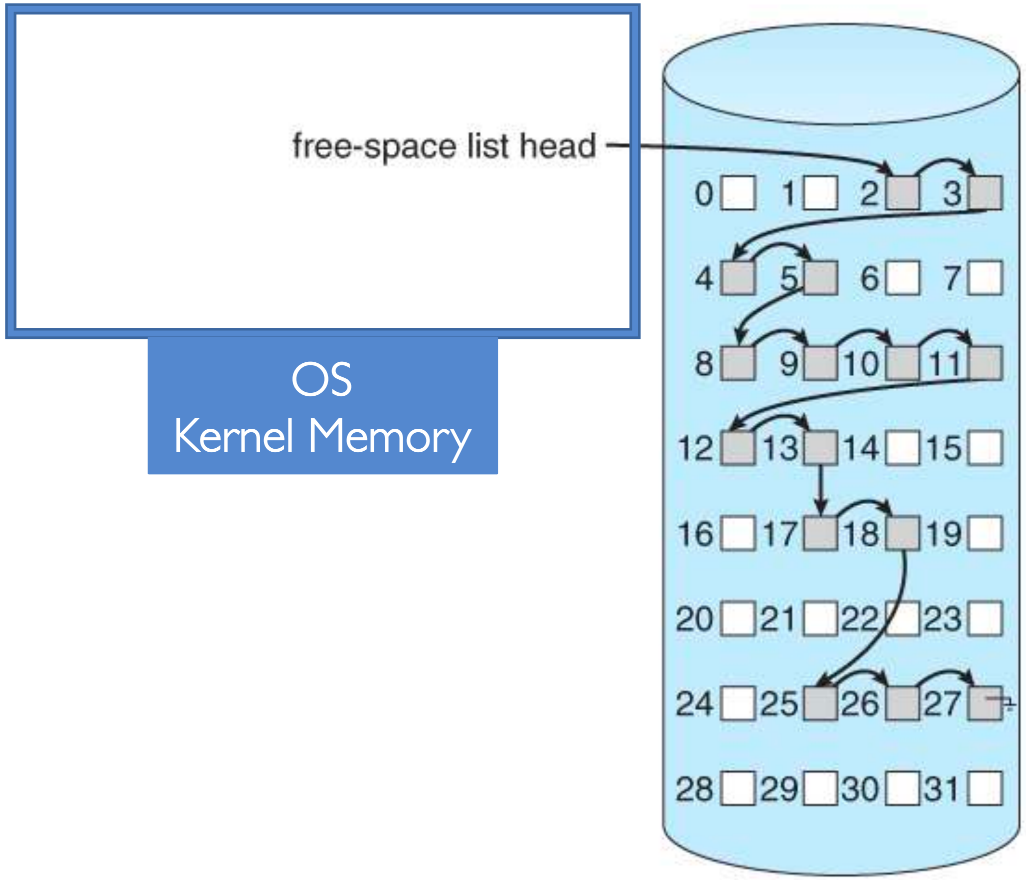
How many entries does a 2 TB disk with 512-byte sectors need?

~4,000,000,000 bitmap entries = 500,000,000 bytes = 500MB

Free Space Management: Linked List

- If most of the disk is in use, it will be expensive to find free blocks with a bitmap
- An alternative implementation is to link together the free blocks
- The head of the list is cached in kernel memory
- Each block contains a pointer to the next free block
- Allocating/Deallocating blocks by modifying pointers of this list

Free Space Management: Linked List



Summary

- Many of the concerns of file system implementation are similar to those of virtual memory implementation

Summary

- Many of the concerns of file system implementation are similar to those of virtual memory implementation
- Contiguous allocation is simple, but suffers from external fragmentation, the need for compaction, and the need to move files as they grow

Summary

- Many of the concerns of file system implementation are similar to those of virtual memory implementation
- Contiguous allocation is simple, but suffers from external fragmentation, the need for compaction, and the need to move files as they grow
- Indexed allocation is very similar to page tables
    - A table maps from logical file blocks to physical disk blocks

Summary

- Many of the concerns of file system implementation are similar to those of virtual memory implementation
- Contiguous allocation is simple, but suffers from external fragmentation, the need for compaction, and the need to move files as they grow
- Indexed allocation is very similar to page tables
  - A table maps from logical file blocks to physical disk blocks
- Free space can be managed using a bitmap or a linked list