Sistemi Operativi

Corso di Laurea in Informatica

a.a. 2020-2021

SAPIENZA
UNIVERSITÀ DI ROMA

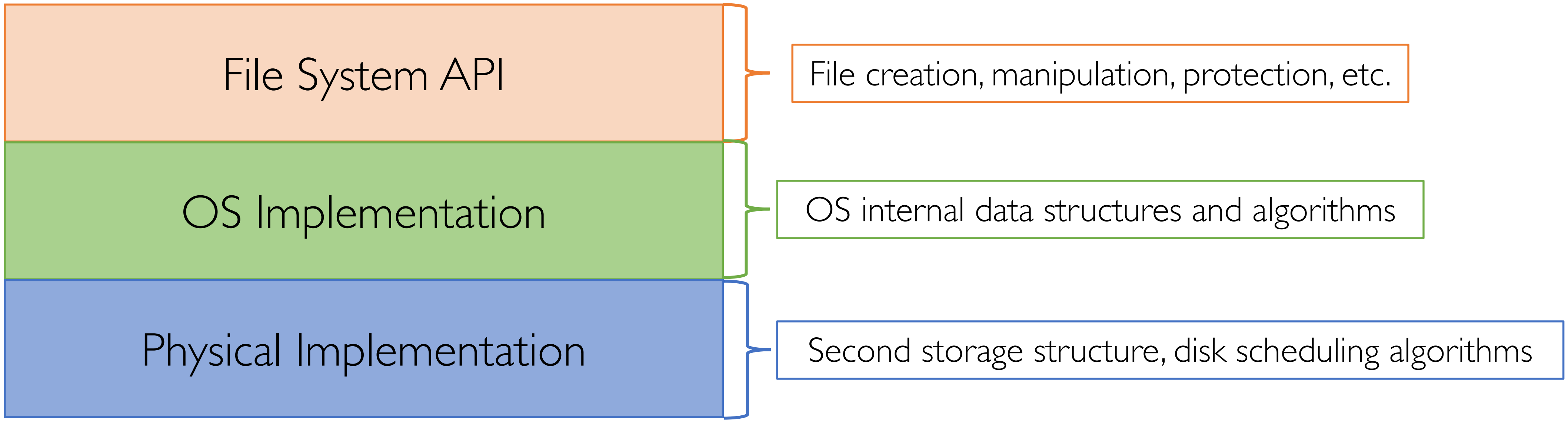
Gabriele Tolomei

Dipartimento di Informatica

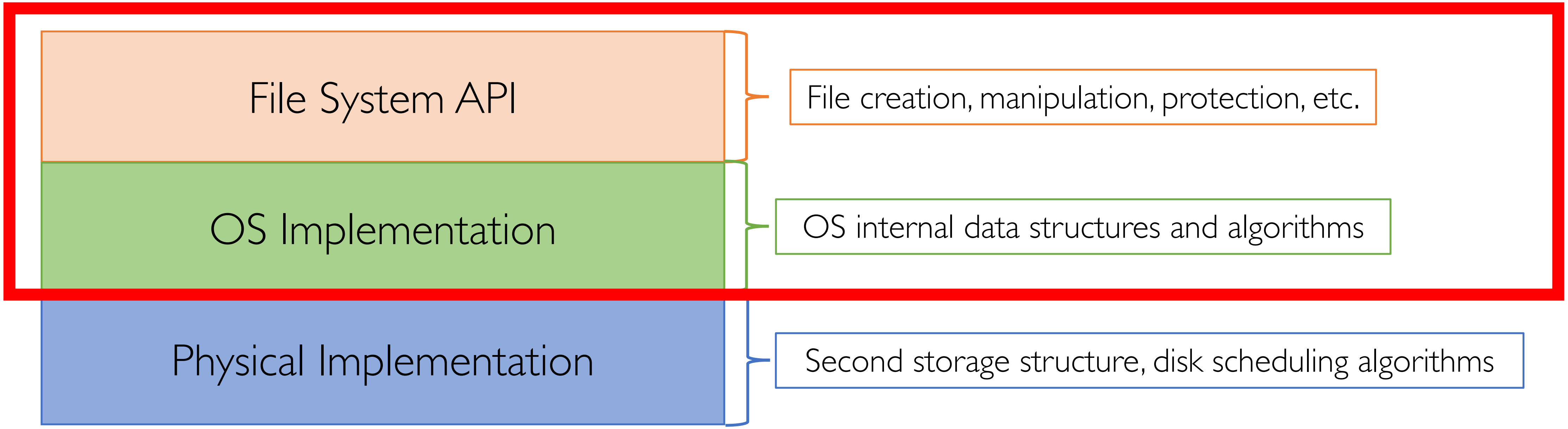
Sapienza Università di Roma

tolomei@di.uniroma1.it

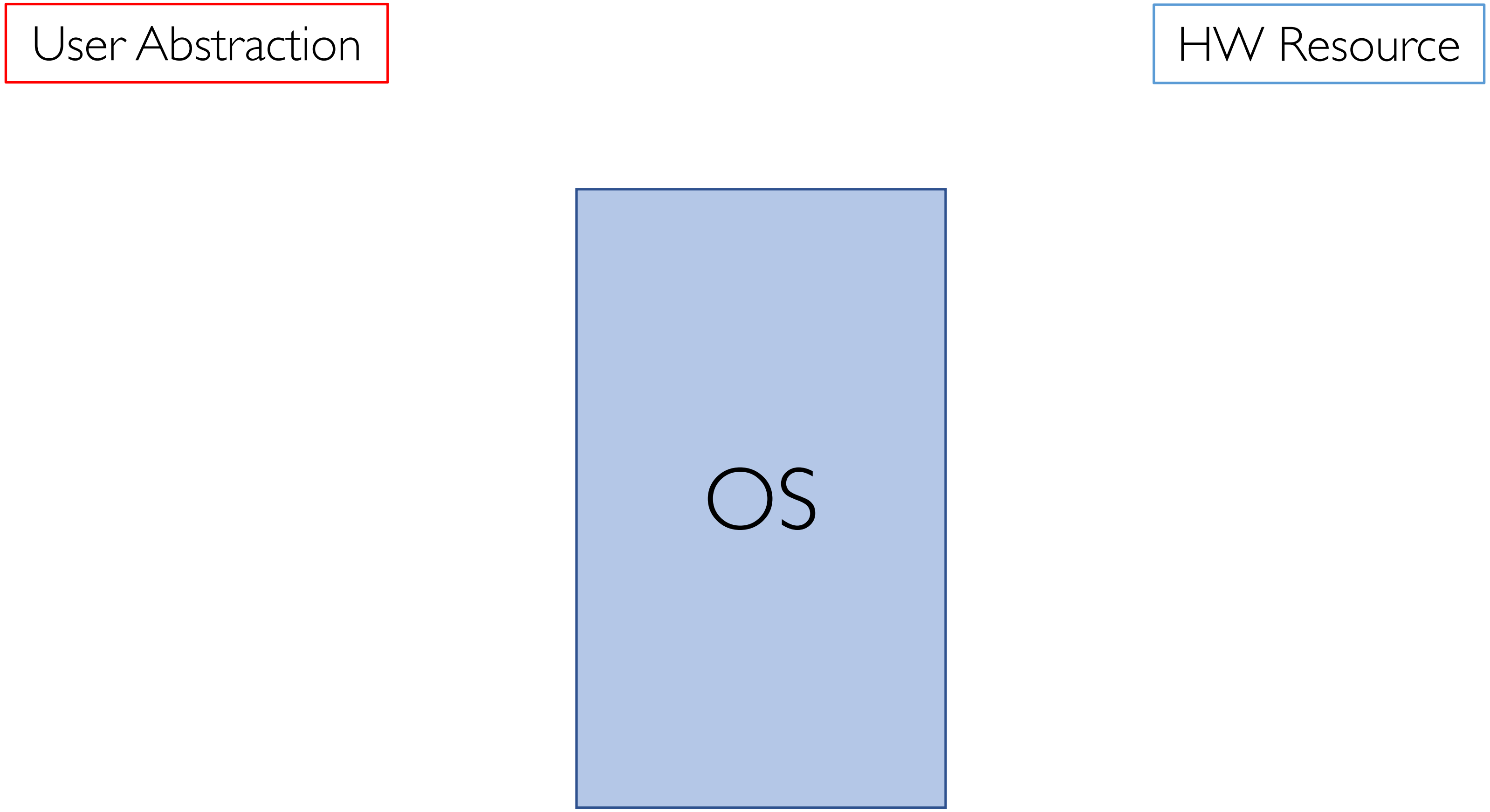
File System's Logical View



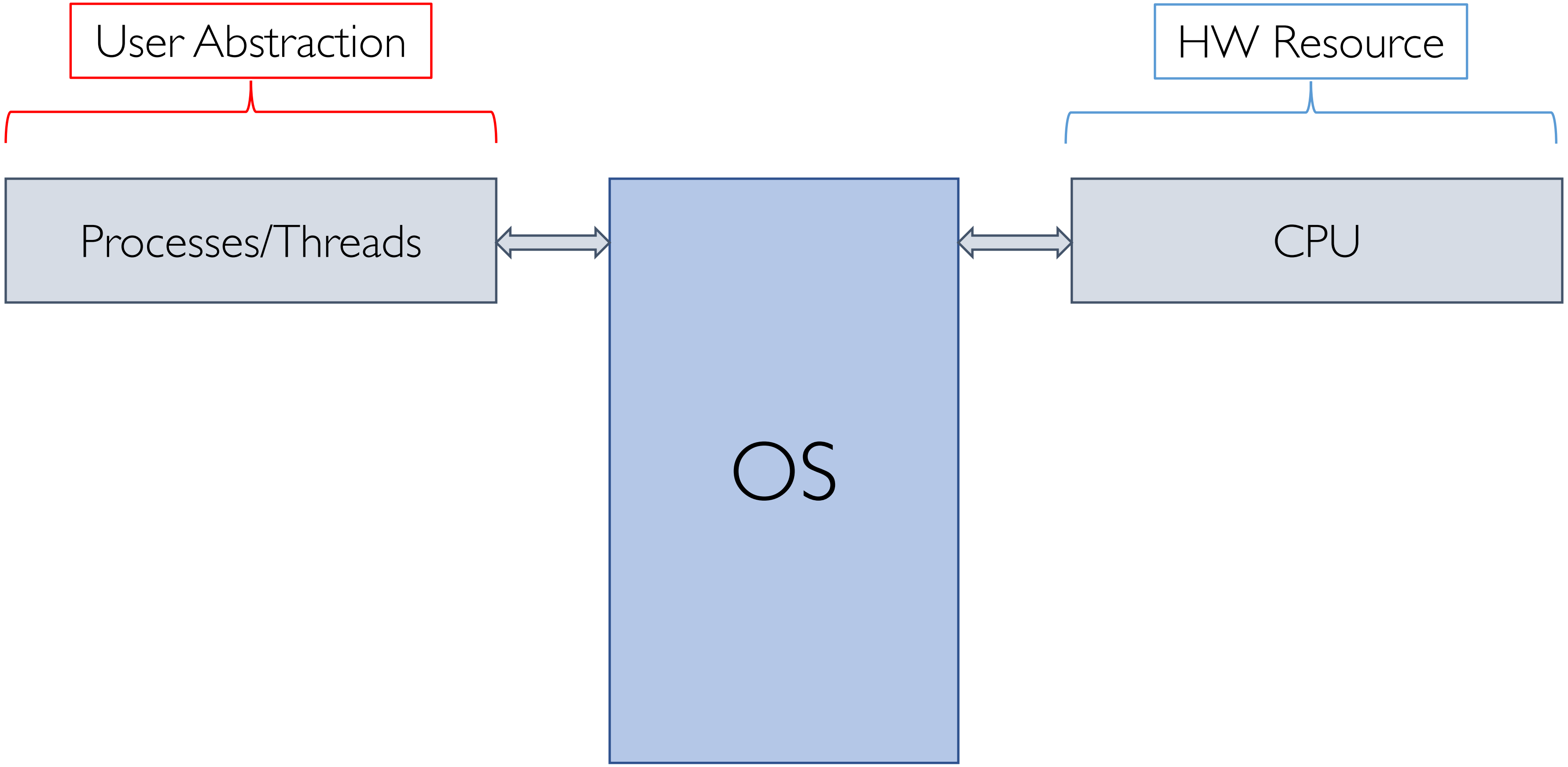
File System's Logical View



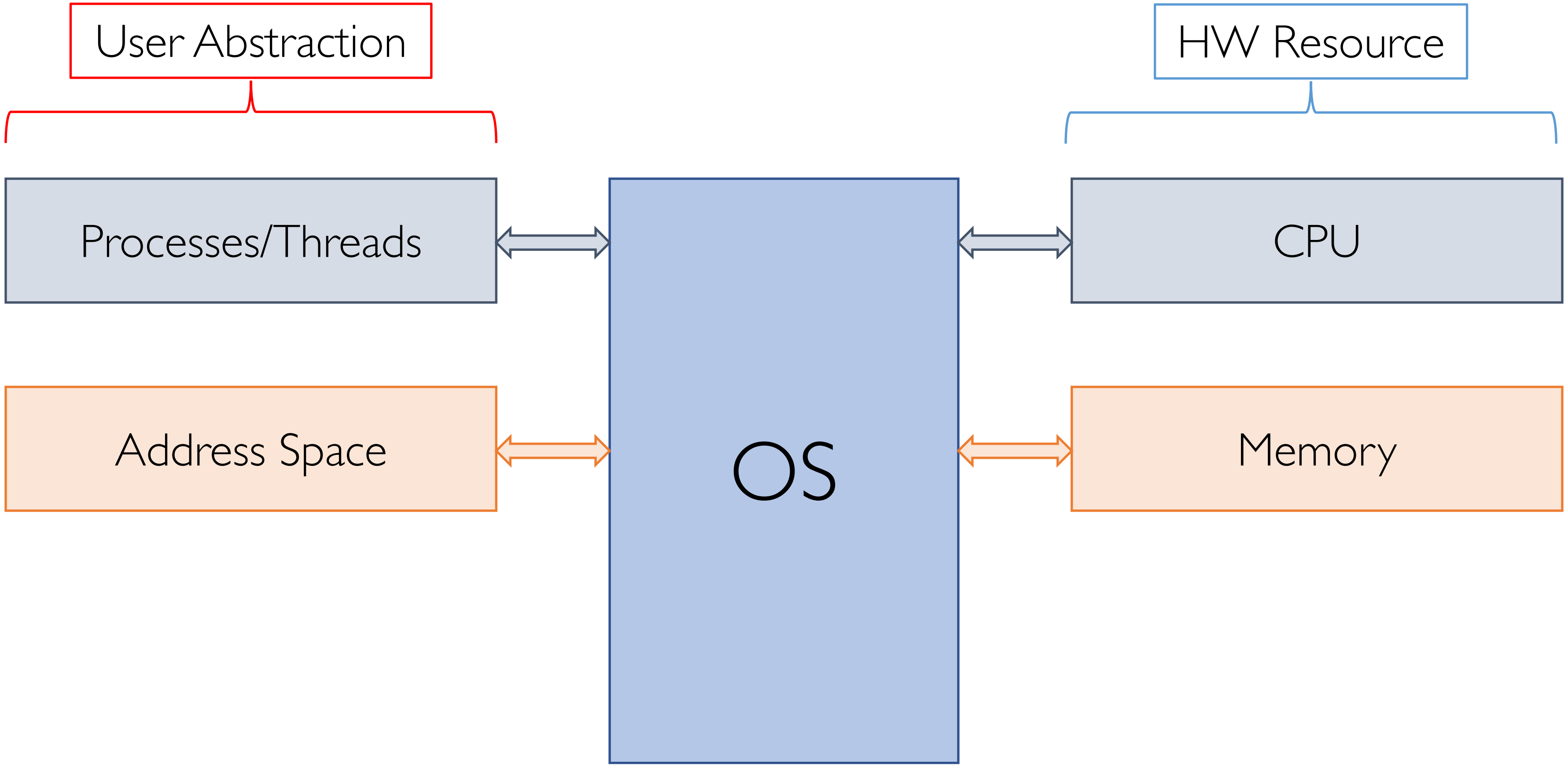
Abstraction



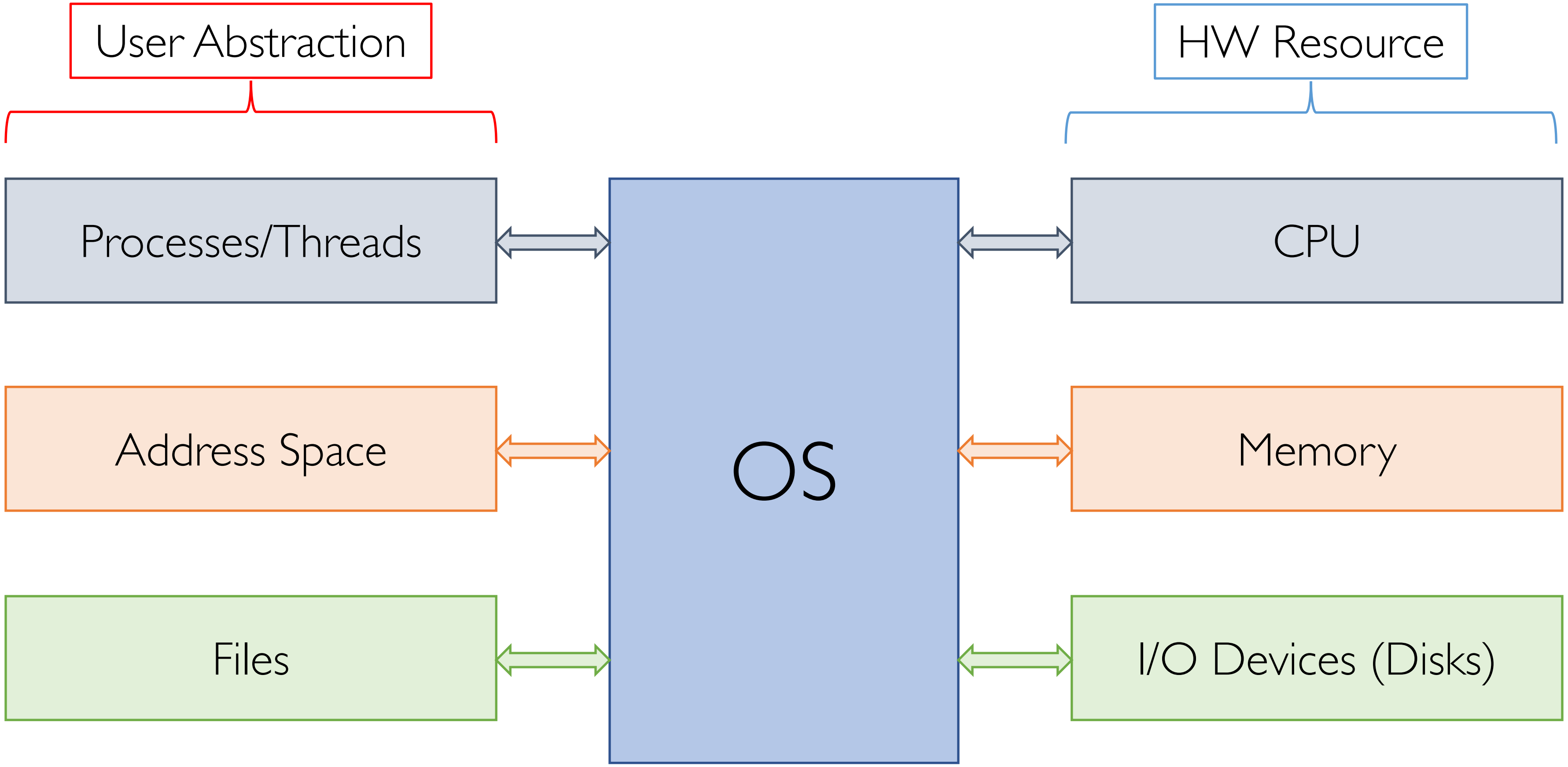
Abstraction



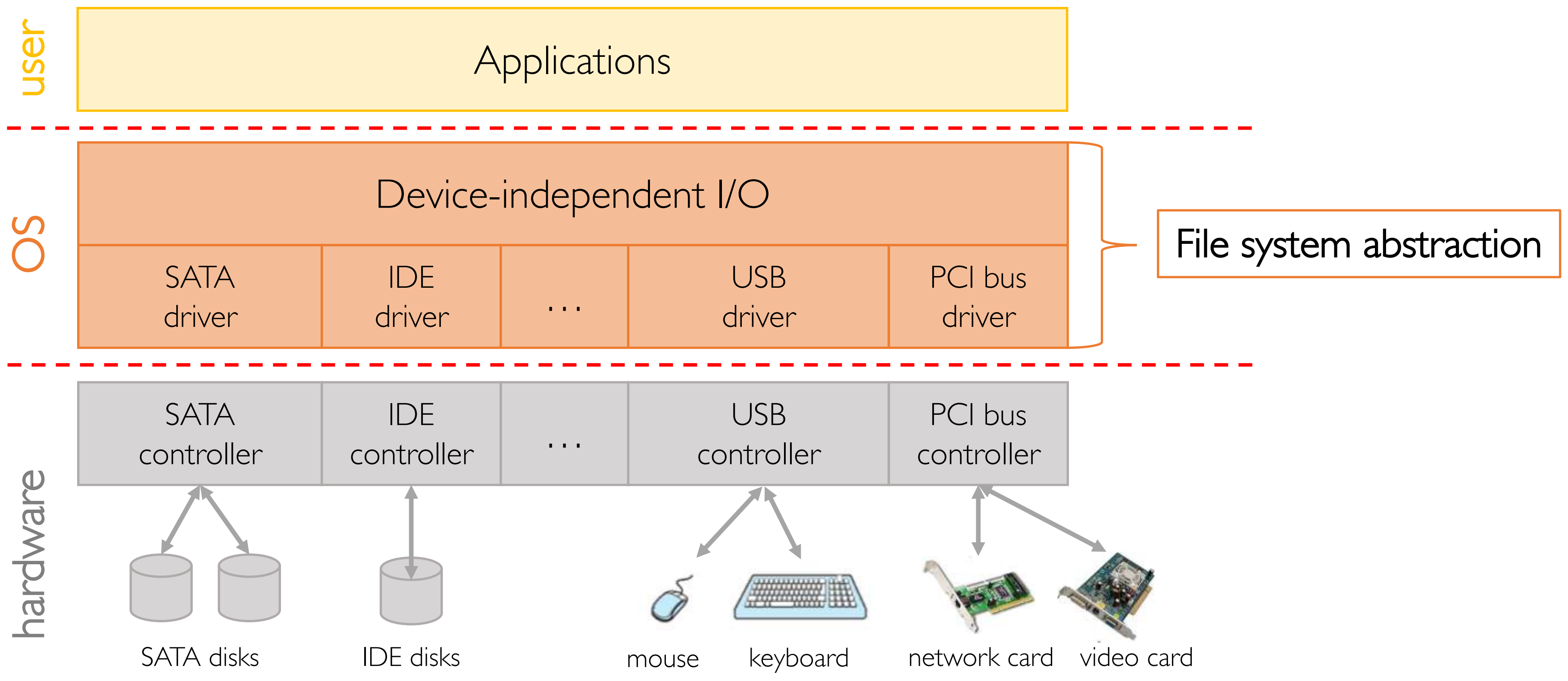
Abstraction



Abstraction



File System Abstraction



User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes

User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes
- Speed → Data must be retrieved quickly

User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes
- Speed → Data must be retrieved quickly
- Size → May want to store huge amounts of data

User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes
- Speed → Data must be retrieved quickly
- Size → May want to store huge amounts of data
- Sharing/Protection → Data can be shared where appropriate

User Requirements on Data

- Persistence → Data must stay around between jobs, reboots, crashes
- Speed → Data must be retrieved quickly
- Size → May want to store huge amounts of data
- Sharing/Protection → Data can be shared where appropriate
- Ease of Use → Data should be easily found, examined, modified, etc.

HW vs. OS Capabilities

- HW provides:
  - Persistence: Disks are non-volatile storage devices
  - Speed (somewhat): Disks enable direct/random access
  - Size: Disks keep getting bigger (order of TBs on today's laptop)

HW vs. OS Capabilities

- HW provides:
  - Persistence: Disks are non-volatile storage devices
  - Speed (somewhat): Disks enable direct/random access
  - Size: Disks keep getting bigger (order of TBs on today's laptop)
- OS provides:
  - Persistence: Redundancy mechanisms
  - Sharing/Protection: Permissions (e.g., UNIX **`rwx`** privileges)
  - Ease of Use: named files, directories, search tools (e.g., Spotlight in macOS)

What's a File?

- The abstraction used by the OS to refer to the logical unit of data on a storage device
  - Named collection of related information stored on secondary memory

What's a File?

- The abstraction used by the OS to refer to the logical unit of data on a storage device
  - Named collection of related information stored on secondary memory
- Files are mapped by the OS onto physical storage devices (e.g., disks)
  - Such devices are non-volatile (their content persist across reboots)

What's a File?

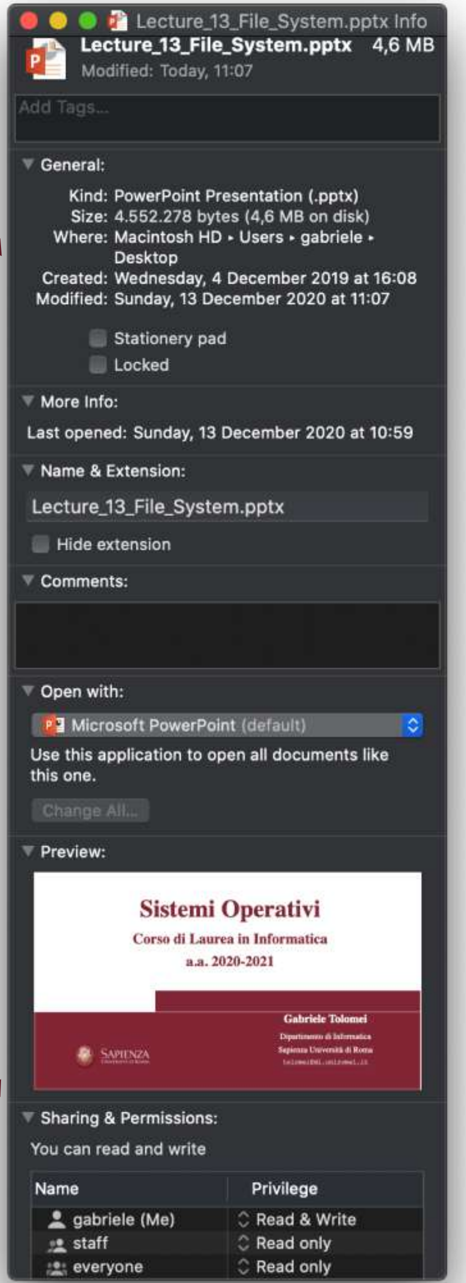
- The abstraction used by the OS to refer to the logical unit of data on a storage device
  - Named collection of related information stored on secondary memory
- Files are mapped by the OS onto physical storage devices (e.g., disks)
  - Such devices are non-volatile (their content persist across reboots)
- Files can contain programs (source, binary) or data
  - Examples: **`main.cpp`**, **`test.exe`**, **`doc.txt`**

Files: Attributes (Metadata)

- Different OSs keep track of different file attributes
- Examples:
  - Name: human-friendly identifier
  - Identifier: how the OS actually identifies the file (e.g., **`inode`** number)
  - Type: text, executable, other binary, etc.
  - Location (on the hard drive)
  - Size
  - Protection
  - Time & Date
  - User ID

Files: Attributes (Example)

All the information displayed are metadata associated with *this* file



Files: Operations (User Interface to File System)

- Operations can affect the actual file content (i.e., data) or metadata

Files: Operations (User Interface to File System)

- Operations can affect the actual file content (i.e., data) or metadata
- Data operations:
  - `create()`, `open()`, `read()`, `write()`, `seek()`, `close()`, `delete()`

Files: Operations (User Interface to File System)

- Operations can affect the actual file content (i.e., data) or metadata
- Data operations:
  - **`create()`**, **`open()`**, **`read()`**, **`write()`**, **`seek()`**, **`close()`**, **`delete()`**
- Metadata operations:
  - change owner/permissions (**`chown`**/**`chmod`**)
  - make symbolic links (**`ln`**)
  - etc.

Files: Operations (User Interface to File System)

- Operations can affect the actual file content (i.e., data) or metadata
- Data operations:
  - **`create()`**, **`open()`**, **`read()`**, **`write()`**, **`seek()`**, **`close()`**, **`delete()`**
- Metadata operations:
  - change owner/permissions (**`chown`**/**`chmod`**)
  - make symbolic links (**`ln`**)
  - etc.

Those are all system calls typically wrapped within a user library

OS (Kernel) File Data Structures

Global Open File Table

- shared by all the processes with an open file
- one entry for each open file
- multiple processes may have the same file open (counter)
- file attributes (ownership, protection, etc.)
- location of each file on disk
- pointers to location of each file on disk

OS (Kernel) File Data Structures

Global Open File Table

- shared by all the processes with an open file
- one entry for each open file
- multiple processes may have the same file open (counter)
- file attributes (ownership, protection, etc.)
- location of each file on disk
- pointers to location of each file on disk

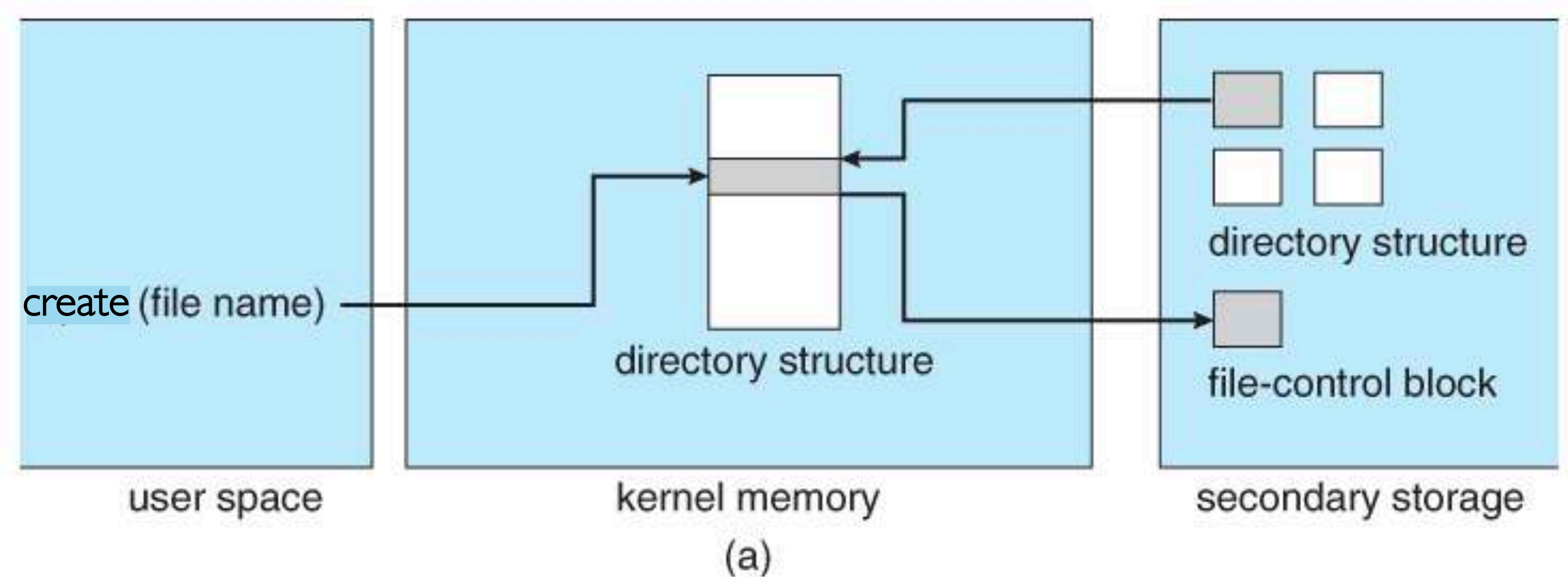
Local Per-Process File Table

- one table for each process
- for each open file of this process:
  - pointer to the entry in the global table
  - current position in the file (offset)
  - open mode (r, w, r/w)

Files Operations: `create(filename)`

- Allocate disk space, also checking disk quotas and permissions
- Create a file descriptor for the file including:
  - **`filename`**
  - location on disk
  - other attributes
- Add the file descriptor to the directory that contains the file

Files Operations: `create(filename)`



Files Operations: `create(filename)`

- Optional file attribute: file type (MS Word, executable, etc.):
  - better error detection
  - specialized default operations (e.g., double-click triggers the right application)
  - storage layout optimization
  - more complex filesystem and OS
  - less flexibility (what if we want to change the file type)
- In UNIX no file type, Windows and Mac opt for user-friendliness

Files Operations: `delete(filename)`

- Find the directory containing the file
- Free the disk blocks used by the file
- Remove the file descriptor from the directory
- Behavior dependent on hard links (more on this later)

Files Operations: `open(filename,mode)`

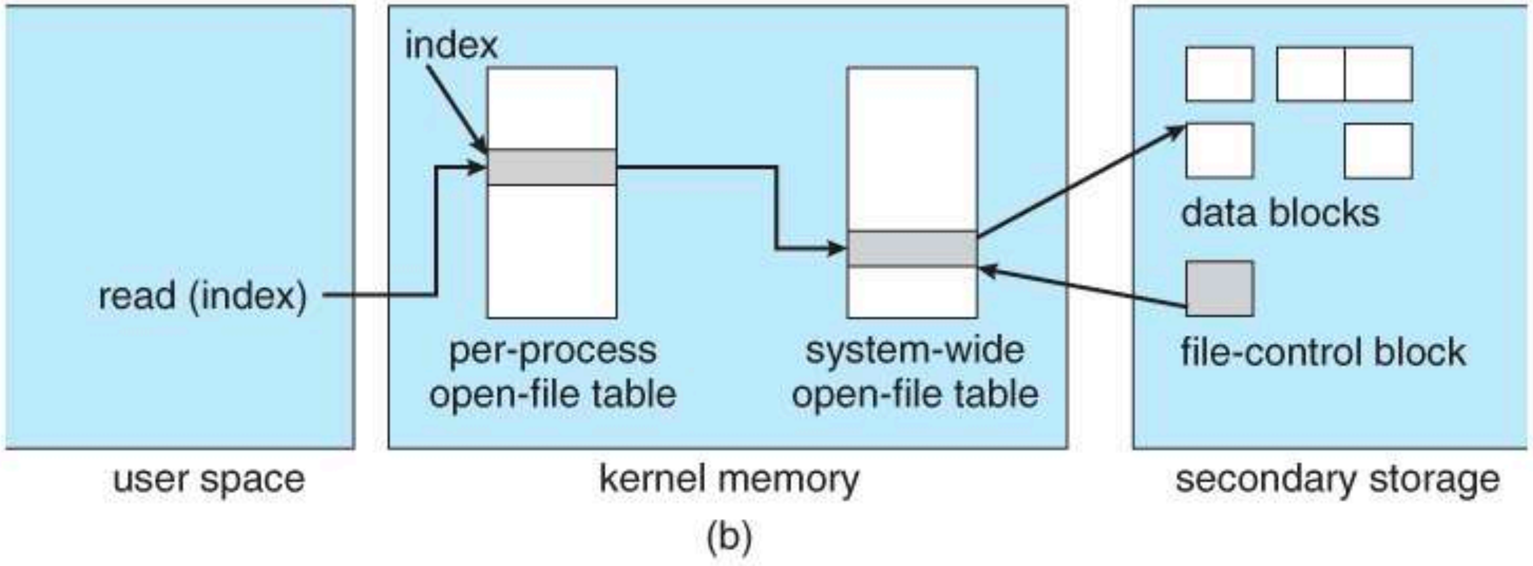
- Returns the **`fileID`** the OS associated with that **`filename`**
- Check the global open file table if the file is already open by another process, if not:
  - Find the file and copy the file descriptor into the global open file table
- Check protection of the file against the mode, if not ok abort
- Increment the open count
- Create an entry in the process' file table pointing to the entry of the global table, and initialize the file pointer to the beginning of the file

Files Operations: `close(fileID)`

- Remove the entry for the file in the process' file table
- Decrement the open count of this file on the global file table
- If the open count gets to 0 → no processes have this file open
  - The corresponding entry in the global table can be safely removed

Files Operations: `read(fileID)`

- Read a file given the index (file descriptor) returned by the **`open`** call
- In order for a file to be read, it must therefore be open!



Files Operations: Read

- 2 possible ways of reading a file:
  - random/direct access
  - sequential access

Files Operations: Read

- 2 possible ways of reading a file:
  - random/direct access
  - sequential access
- random/direct access → hard drives (or main memory)
  - Can access to a specific disk block (memory address)

Files Operations: Read

- 2 possible ways of reading a file:
  - random/direct access
  - sequential access
- random/direct access → hard drives (or main memory)
  - Can access to a specific disk block (memory address)
- sequential access → devices which do not support direct access (e.g., tape drives)
  - Need to go all the way through the desired position

Files Operations: Read (Random Access)

- **`read(fileID, from, size, bufAddress)`**
  - OS reads **`size`** bytes from file position **`from`** into **`bufAddress`**

Files Operations: Read (Random Access)

- **`read(fileID, from, size, bufAddress)`**
  - OS reads **`size`** bytes from file position **`from`** into **`bufAddress`**

```
for (i = from; i < from + size; ++i) {
    bufAddress[i - from] = fileID[i];
}
```

Files Operations: Read (Sequential Access)

- **`read(fileID, size, bufAddress)`**
  - OS reads **`size`** bytes from file current position (**`fp`**) into **`bufAddress`**, and updates the file position accordingly

Files Operations: Read (Sequential Access)

- **`read(fileID, size, bufAddress)`**
  - OS reads **`size`** bytes from file current position (**`fp`**) into **`bufAddress`**, and updates the file position accordingly

```
for (i = 0; i < size; ++i) {

    bufAddress[i] = fileID[fp + i];

}
fp += size;
```

Files Operations: Other Operations

- **`write`** → similar to **`read`** but copies from buffer to the file
- **`seek`** → just updates the file position (no need to actual I/O)
- **`mmap`** → Memory mapping a file
  - Map (a part of) the virtual address space to a file
  - Read from/write to that portion of memory implies OS reads from/writes to the corresponding location in the file (stored on disk)
  - File accesses are greatly simplified (no read/write system calls are necessary)
  - No need to copy from/to the buffer in kernel space at each operation

File Access Methods: Programmer's Perspective

- Sequential → Data is accessed in order, one byte/record at a time
  - Example: compiler reading source file

File Access Methods: Programmer's Perspective

- Sequential → Data is accessed in order, one byte/record at a time
  - Example: compiler reading source file
- Direct/Random → Data is accessed at a specific position
  - Example: text editor "goto line" feature

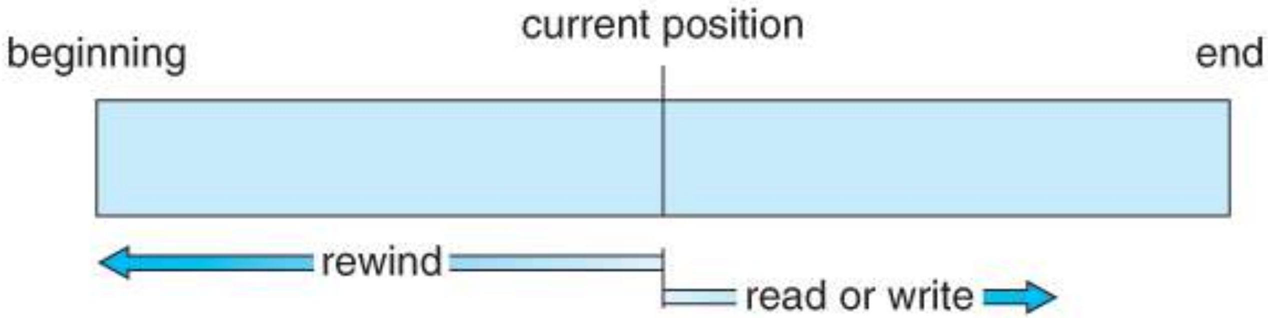
File Access Methods: Programmer's Perspective

- Sequential → Data is accessed in order, one byte/record at a time
  - Example: compiler reading source file
- Direct/Random → Data is accessed at a specific position
  - Example: text editor "goto line" feature
- Keyed/Indexed → Data is accessed based on a key
  - Example: database search

File Access Methods: OS's Perspective

Sequential

Keep a pointer to the next byte in the file, and update the pointer on each read/write operation



File Access Methods: OS's Perspective

Direct/Random

Address any block of data directly given its offset within the file

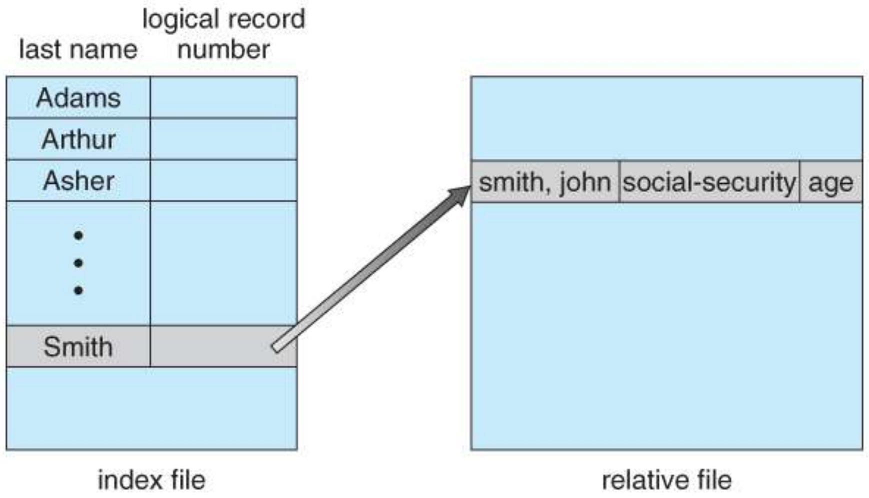
sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

simulating sequential access using direct access

File Access Methods: OS's Perspective

<u>Keyed/Indexed</u>

Address any block of data directly given a key



implemented on top of direct access

Naming and Directories

- Need a method to getting back files that are located on disk
- OS uses unique numbers to identify files
- Users would rather use human-friendly names to refer to files
- Directory → OS data structure which maps file names to descriptors

Directory: Overview

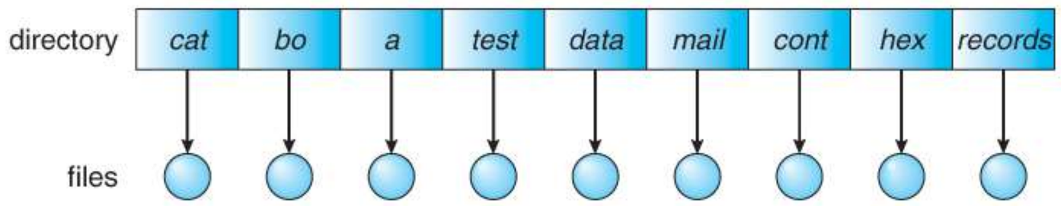
- Directory operations to be supported include:
  - Search for a file
  - Create a file (add it to the directory)
  - Delete a file (erase it from the directory)
  - List a directory (possibly ordered in different ways)
  - Rename a file (may change sorting order)
  - Traverse the file system

Directory: Naming Strategies

- Single-Level Directory
  - One name space for the entire disk
  - Every filename must be unique
  - Use a special area of disk to hold the directory
  - Directory contains (name, index) pairs
  - If one user uses a name, no one else can
  - Used by early personal computers because their disks were very small

Directory: Naming Strategies

- Single-Level Directory
  - One name space for the entire disk
  - Every filename must be unique
  - Use a special area of disk to hold the directory
  - Directory contains (name, index) pairs
  - If one user uses a name, no one else can
  - Used by early personal computers because their disks were very small

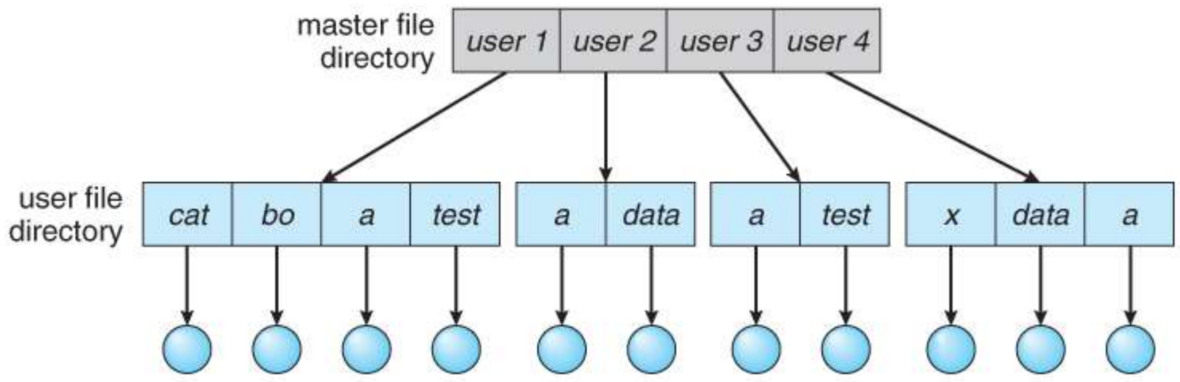


Directory: Naming Strategies

- Two-Level Directory
  - Each user gets their own directory space
  - File names only need to be unique within a given user's directory
  - A master file directory is used to keep track of each users directory
  - A separate directory is generally needed for system (executable) files

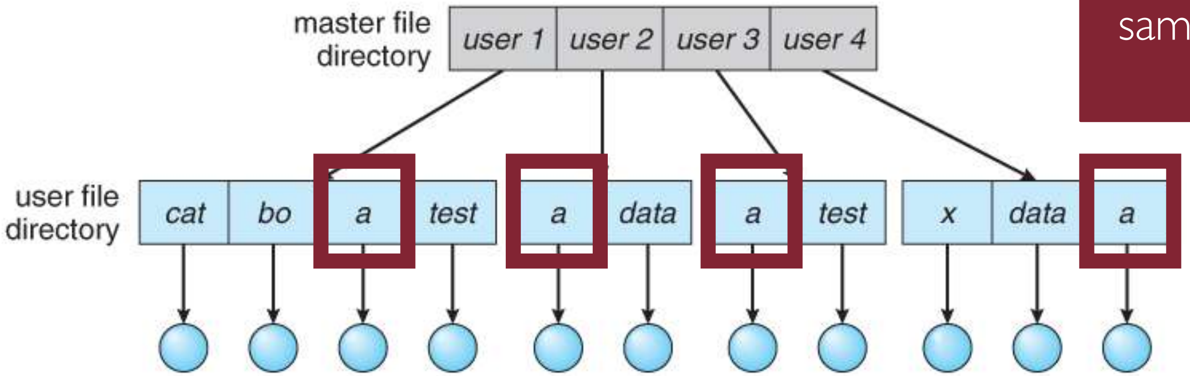
Directory: Naming Strategies

- Two-Level Directory
  - Each user gets their own directory space
  - File names only need to be unique within a given user's directory
  - A master file directory is used to keep track of each users directory
  - A separate directory is generally needed for system (executable) files



Directory: Naming Strategies

- Two-Level Directory
  - Each user gets their own directory space
  - File names only need to be unique within a given user's directory
  - A master file directory is used to keep track of each users directory
  - A separate directory is generally needed for system (executable) files

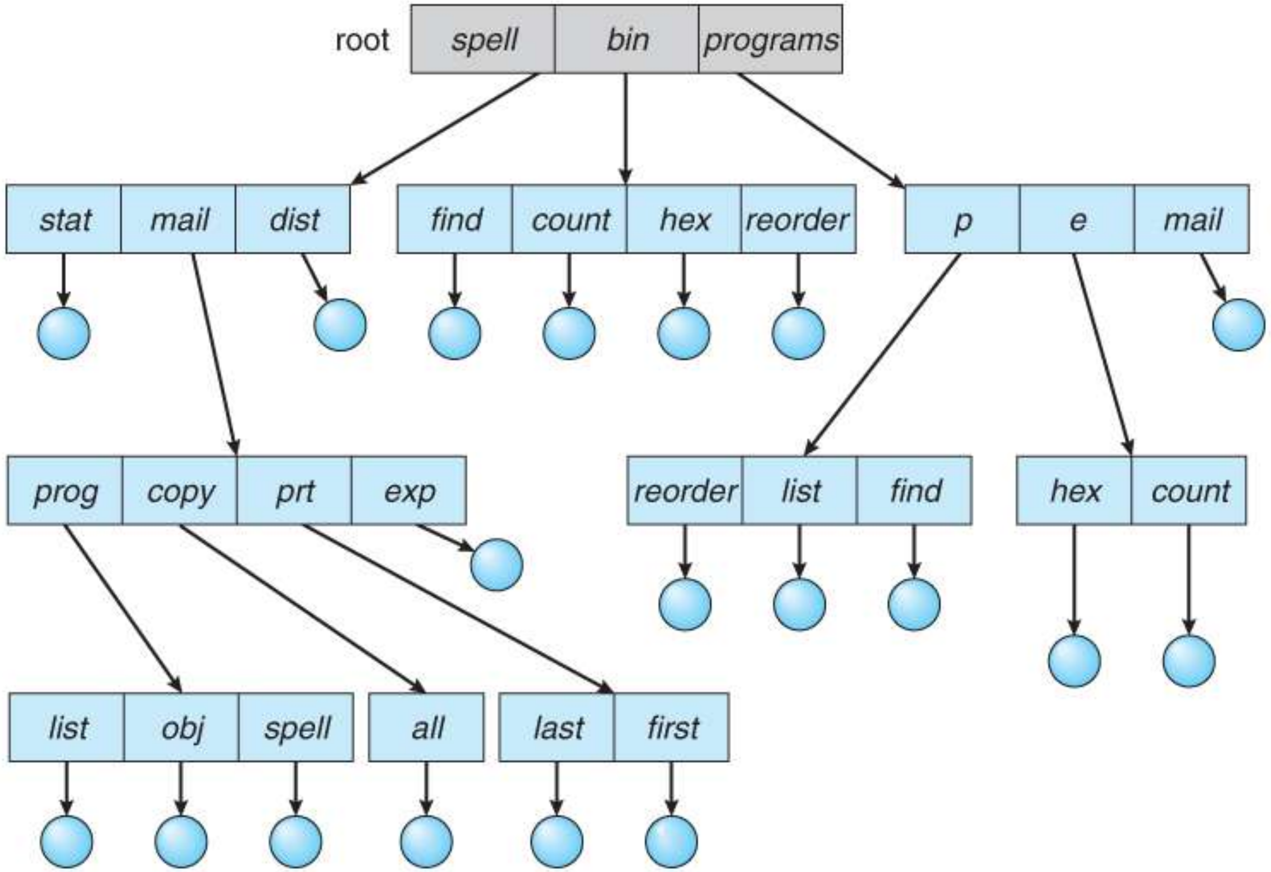


same user directories (possibly same filenames too!) yet underneath different master directories

Directory: Naming Strategies

- Multi-Level (Tree-based) Directory
  - An obvious extension to the two-tiered directory structure
  - Each user/process has the concept of a current directory from which all (relative) searches take place
  - Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory)
  - Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories
  - Used by most modern OSs (UNIX/Linux, Windows, and macOS)

Directory Tree



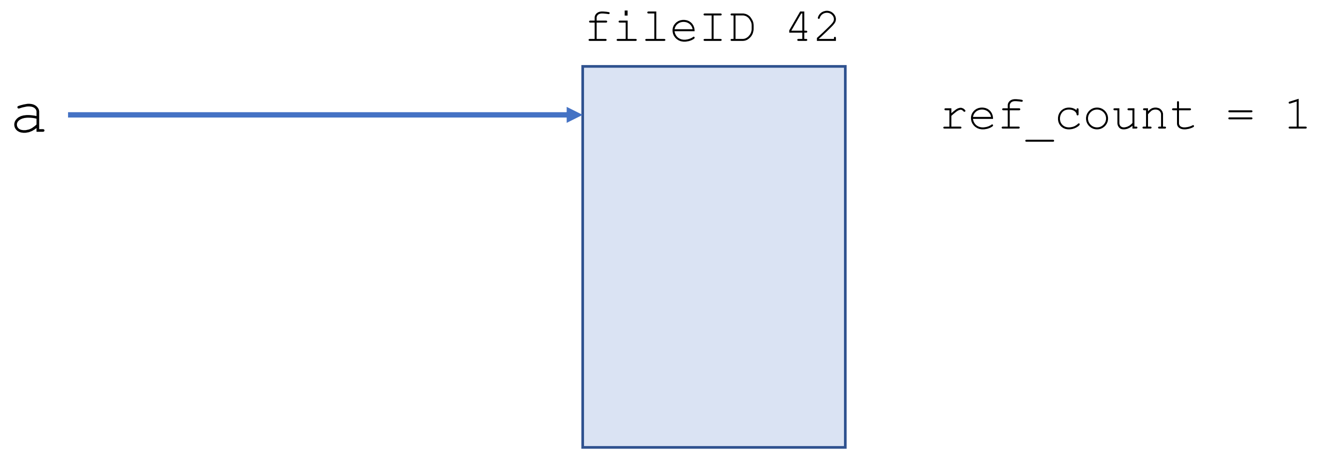
Referential Naming

- Sharing files between different user's directory trees may be complicated

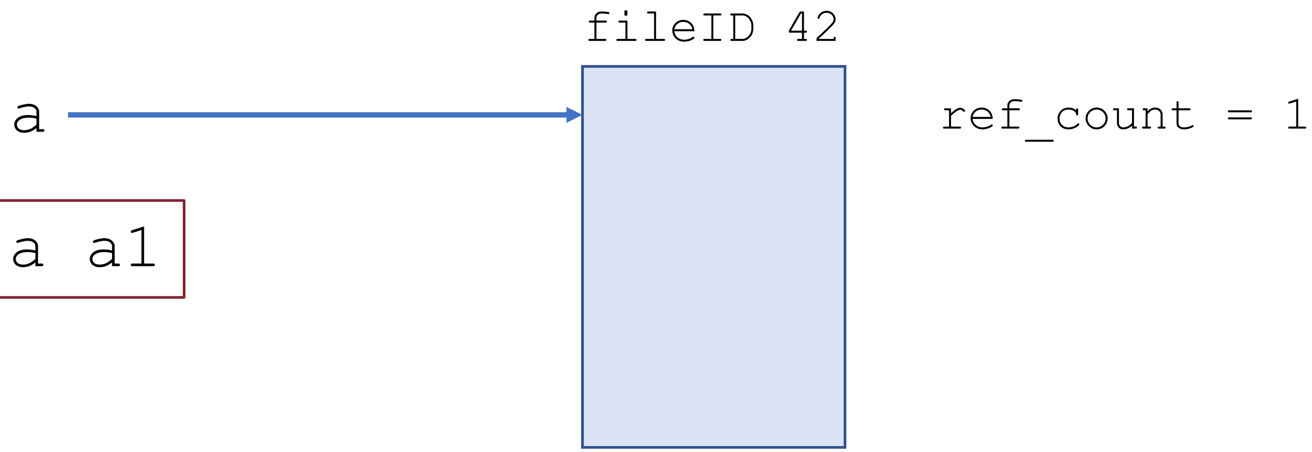
Referential Naming

- Sharing files between different user's directory trees may be complicated
- UNIX provides 2 types of links via the **`ln`** command:
  - hard link → multiple directory entries that refer to the same file
  - symbolic link → an alias to the linked file

Referential Naming: Hard Links

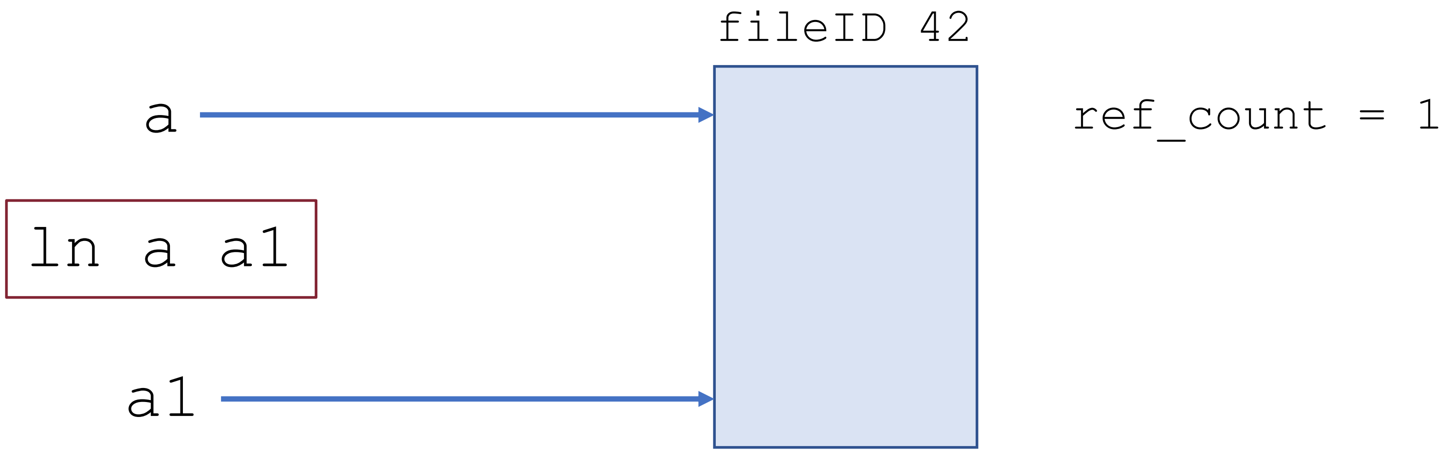


Referential Naming: Hard Links



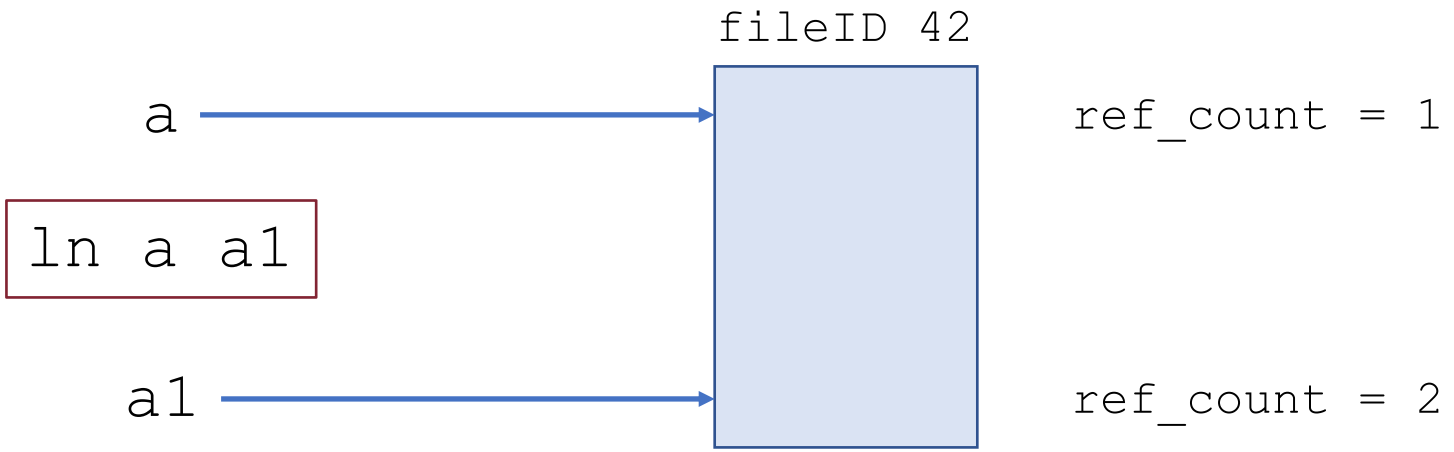

```
ln a a1
```

Referential Naming: Hard Links



Adds a second connection to a file

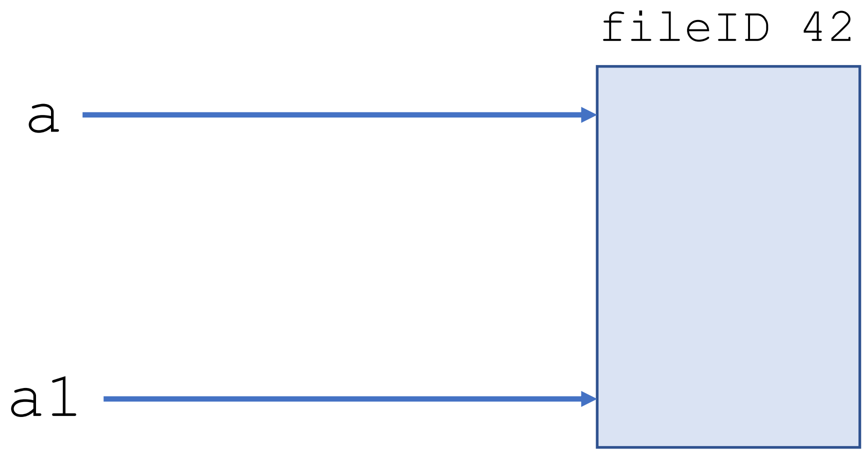
Referential Naming: Hard Links



Adds a second connection to a file

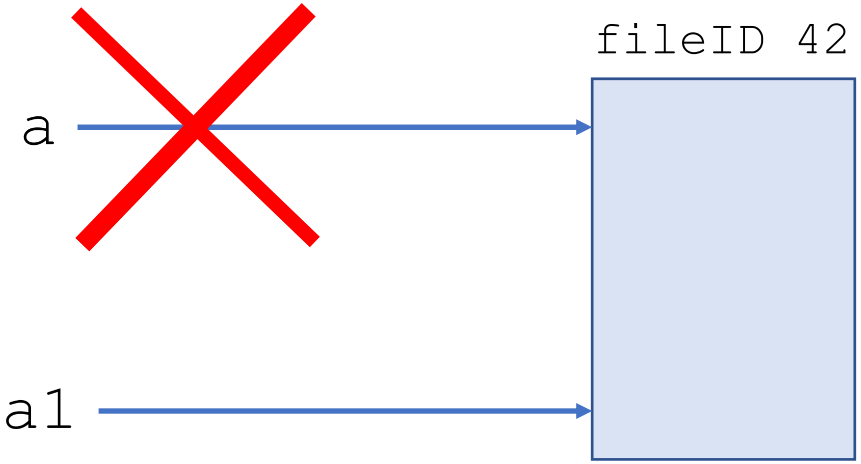
OS maintains reference counts, so it will delete a file only when the last hard link is deleted

Referential Naming: Hard Links



Change to the file using any of its hard links is reflected globally

Referential Naming: Hard Links



Removing a reference **does not** affect others!

as long as reference count > 0

Referential Naming: Hard Links

<u>Problem</u>

Hard links to directories may cause circular links which prevent the OS from claiming back disk space

Referential Naming: Hard Links

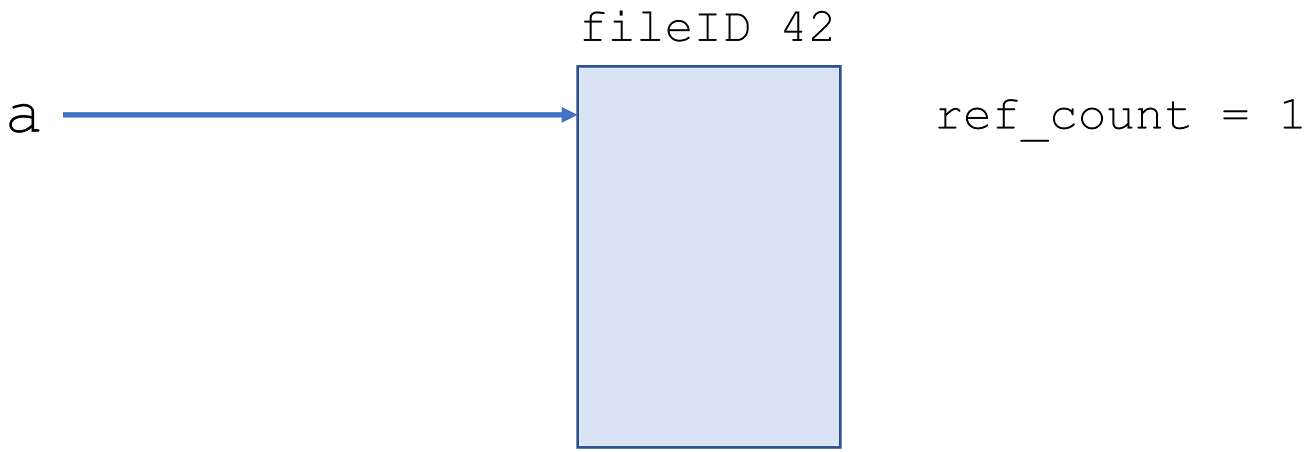
Problem

Hard links to directories may cause circular links which prevent the OS from claiming back disk space

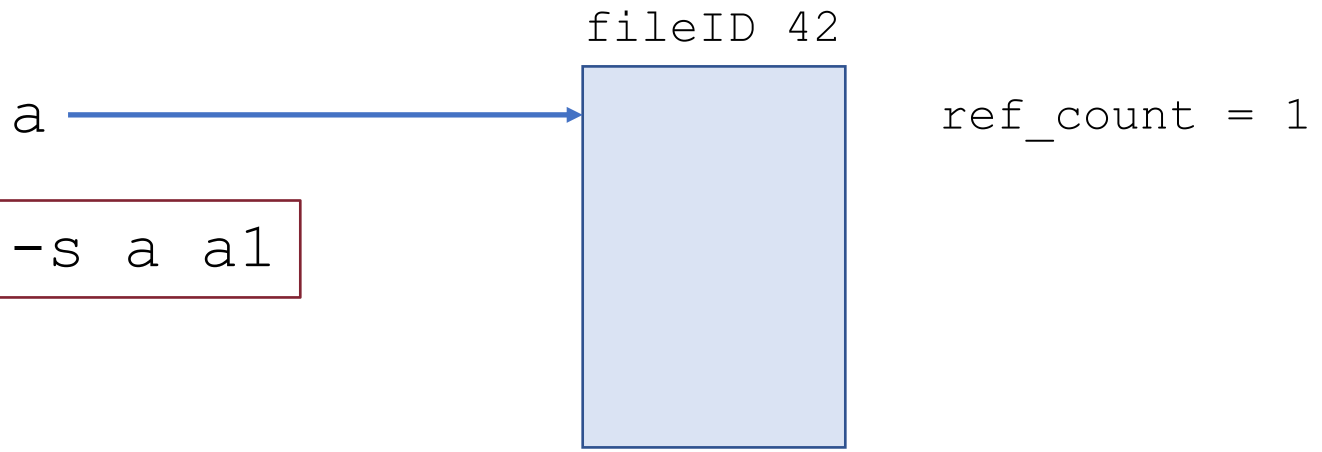
Solution

Do not allow hard links to directories at all!
Hard links to files are safe since files are leaves of the tree

Referential Naming: Soft Links

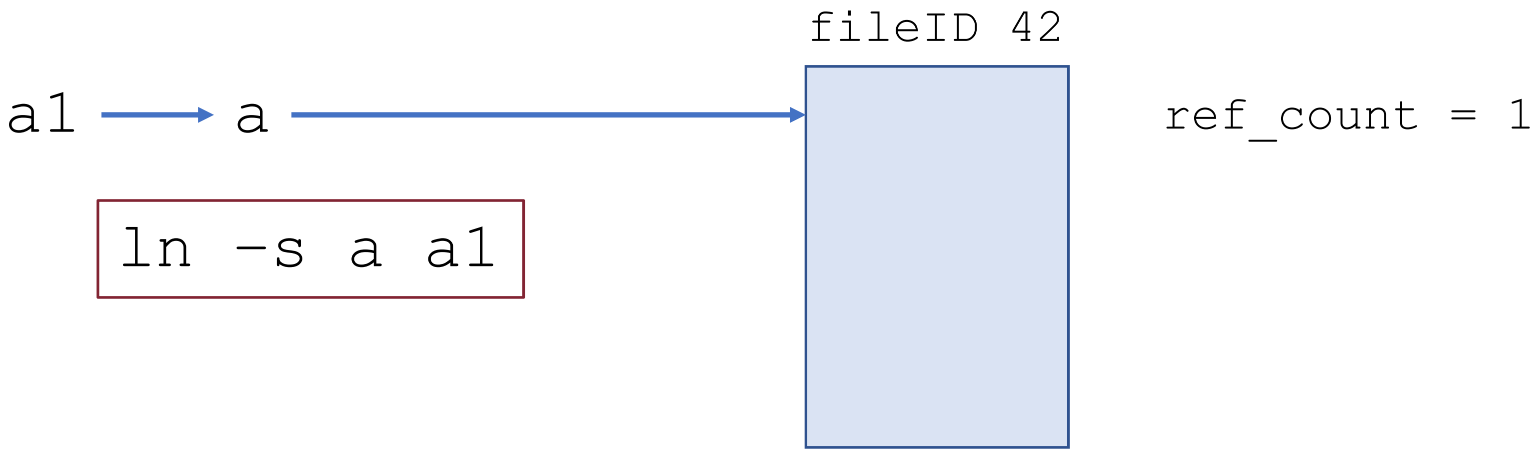


Referential Naming: Soft Links



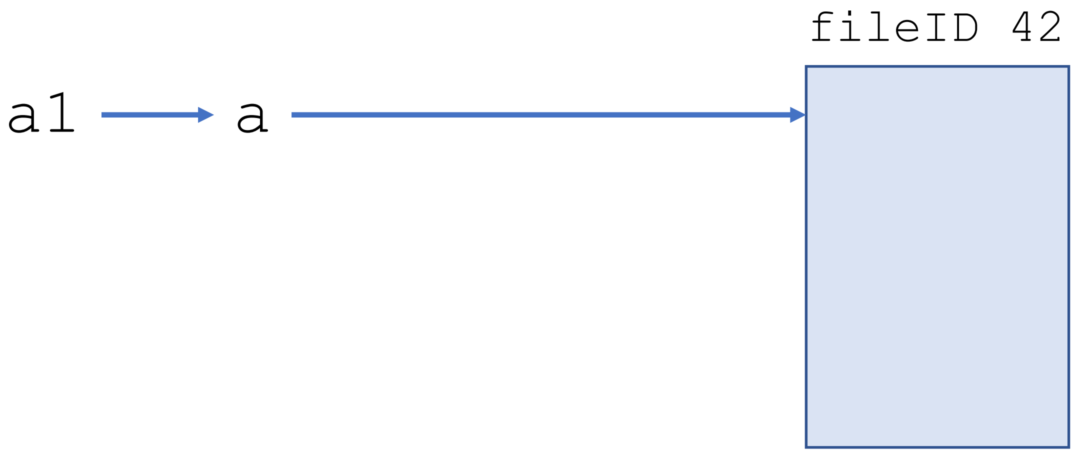

```
ln -s a a1
```

Referential Naming: Hard Links



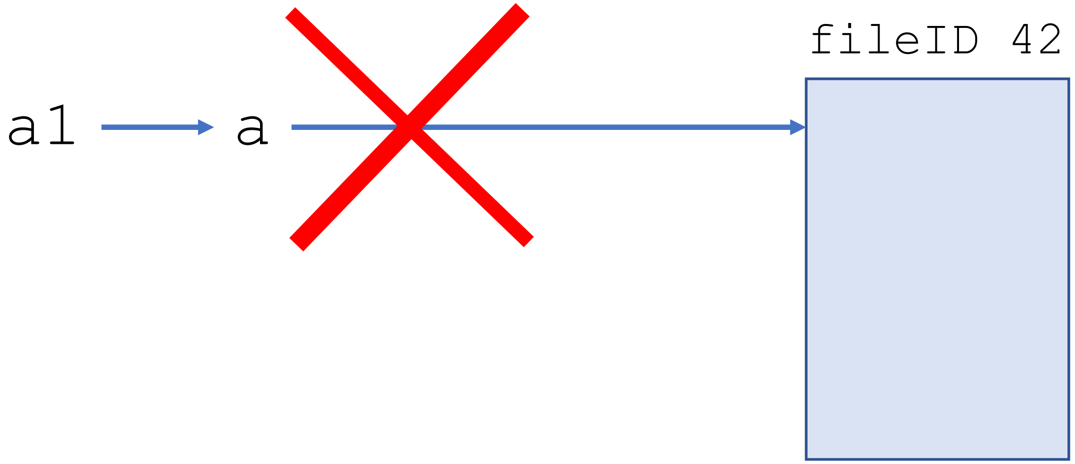
Adds a symbolic pointer to a file

Referential Naming: Soft Links



Change to the file using soft link is reflected globally

Referential Naming: Soft Links



Removing a reference affects all the symbolic links pointing to the file!

`a1` remains in the directory but its content no longer exists (dangling pointer)

File Protection

- The OS must allow users to control sharing of their files

File Protection

- The OS must allow users to control sharing of their files
- Control access to files: grant or deny access to file operations depending on protection information

File Protection

- The OS must allow users to control sharing of their files
- Control access to files: grant or deny access to file operations depending on protection information
- 2 different approaches:
  - access lists and groups (Windows NT)
  - access control bits (UNIX/Linux)

File Protection: Access Lists

- Keep an access list for each file with user name and type of access
- PRO: Highly flexible solution
- CON: Lists can become large and tedious to maintain

File Protection: Access Control Bits

- 3 categories of users: (`owner`, `group`, `world`)
- 3 types of access privileges: (`read`, `write`, `execute`)
- Keep one bit for each privilege on each category

(`111101000` = `rwxr-x---`)

- PRO: Easy to implement and maintain
- CON: Less accurate

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)

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)
- 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 information on disk:
  - Data structure to maintain file location information
- File location on disk:
  - Physically deploy file on disk

On-Disk Data Structures: File Descriptor

- 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 Control Block (FCB)
- A copy of each FCB is stored also in the OS's Global Open File Table

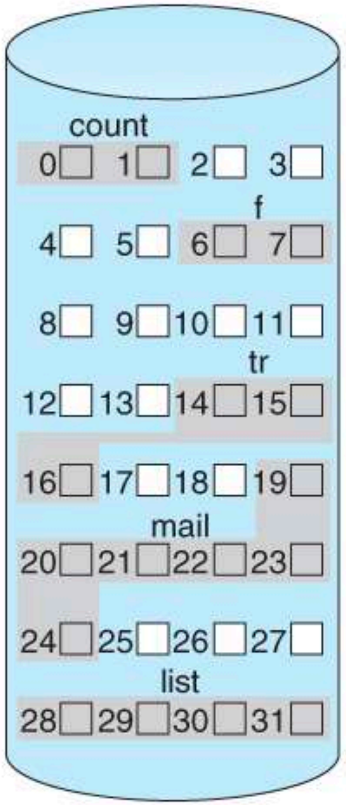
Considerations on Files

- 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
- 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)

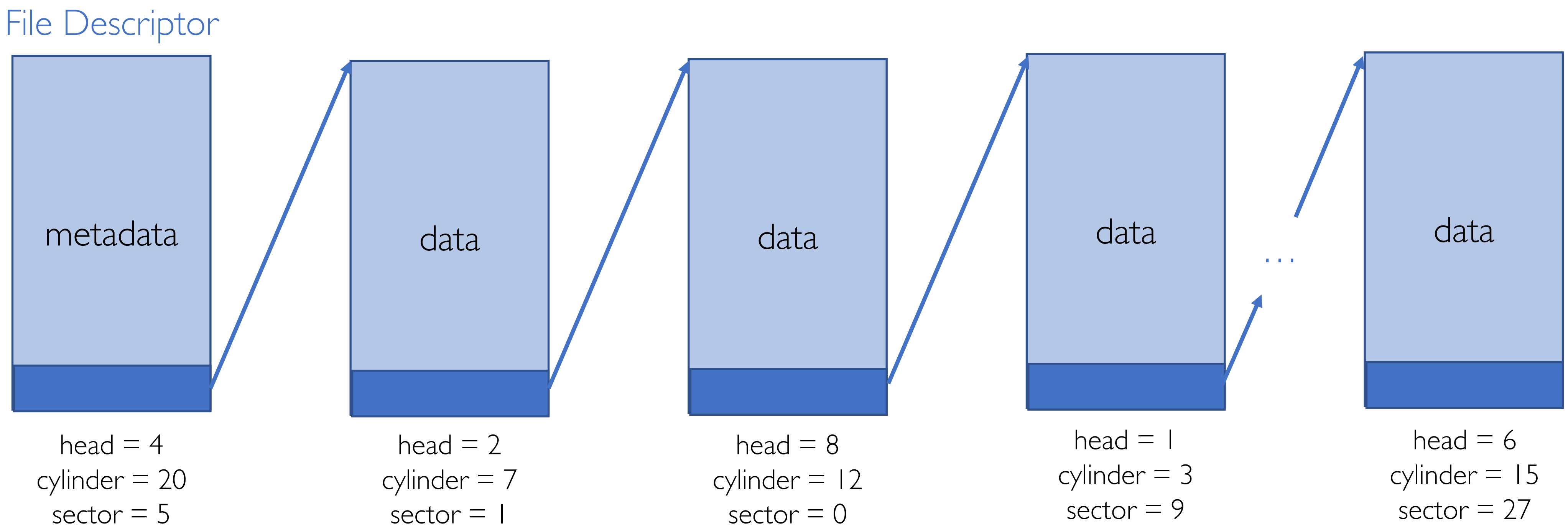
Contiguous Allocation: PROs and CONs

- 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)

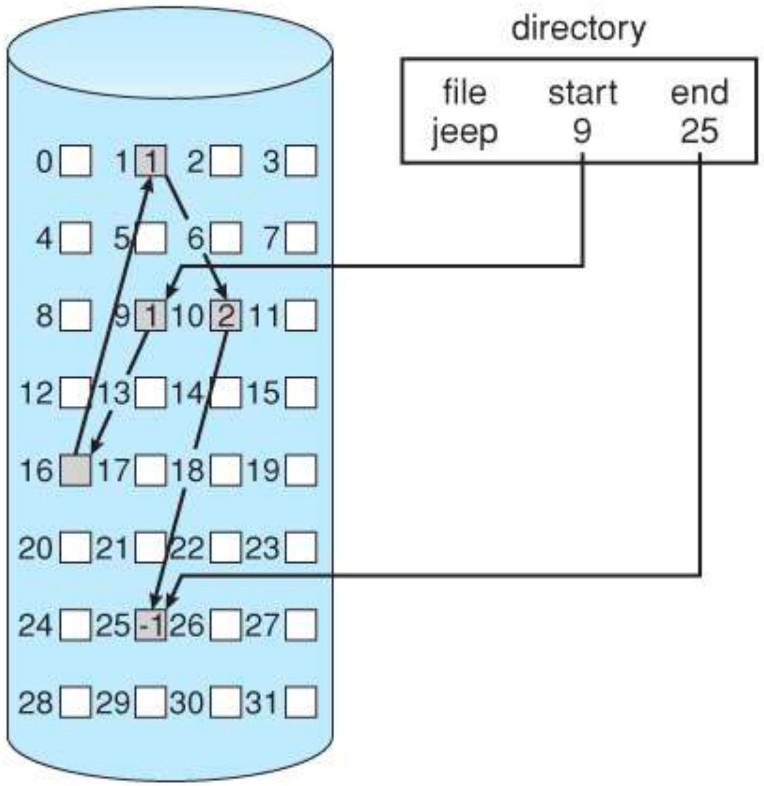
Option 2: Linked Files

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

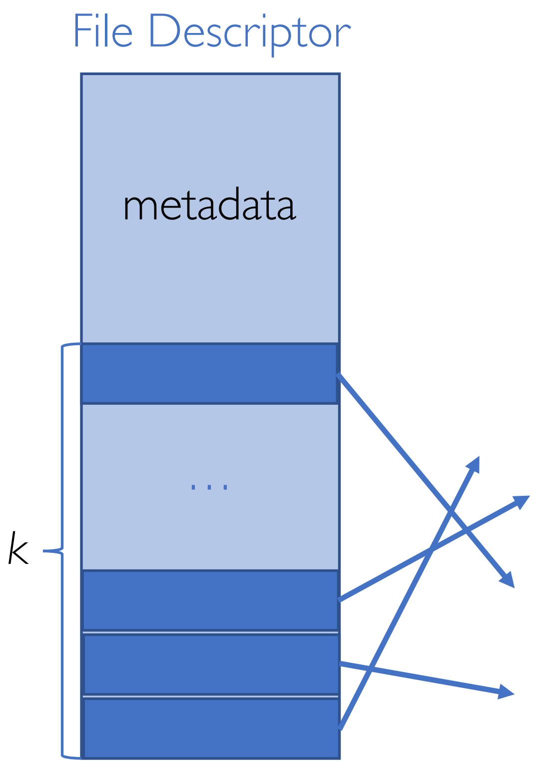
Linked Files: PROs and CONs

- 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

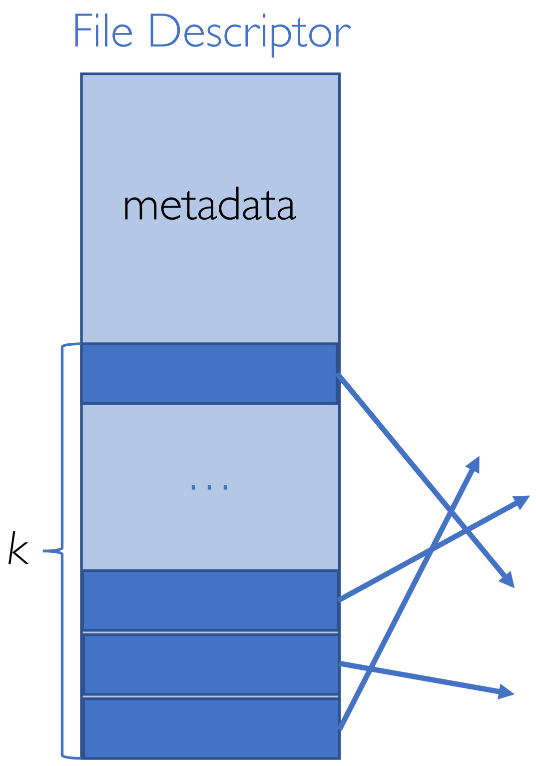
- The file descriptor contains a block of pointers (vs. only 1 pointer as in the linked list approach)
- The user or OS must declare the maximum length of the file when it is created
- OS allocates an array to hold the pointers to all the blocks when it creates the file, but allocates the blocks only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

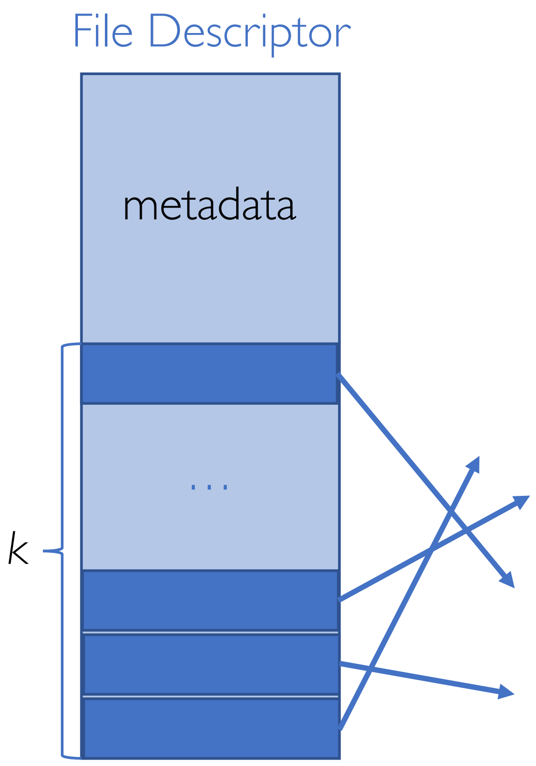
Option 3: Indexed Files



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

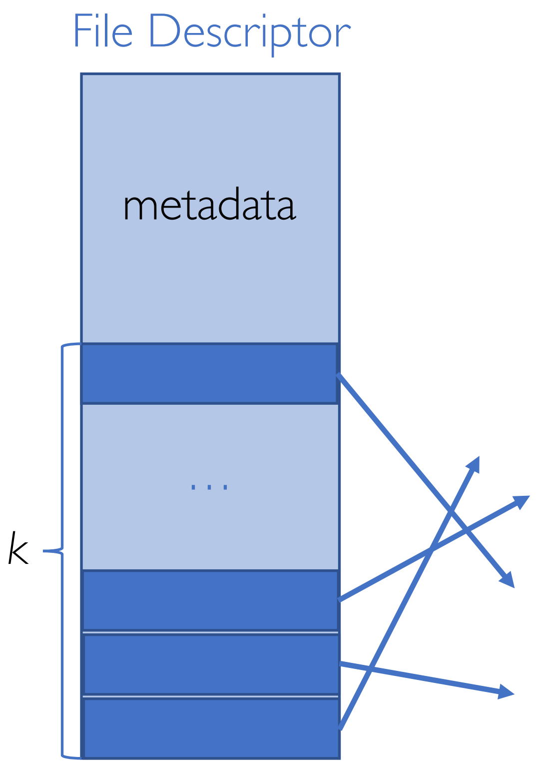


The number of block pointers *k* determine the maximum file size the system can manage

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

Remember: most files are small!

Option 3: Indexed Files



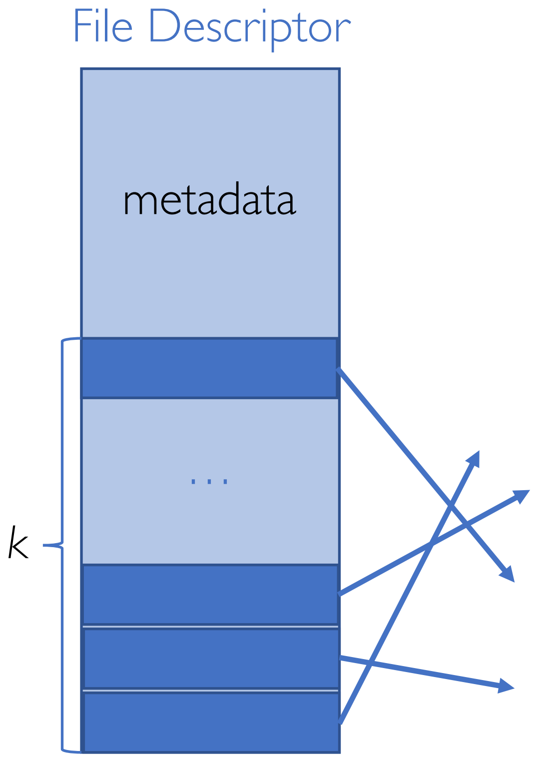
The number of block pointers k determine the maximum file size the system can manage

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

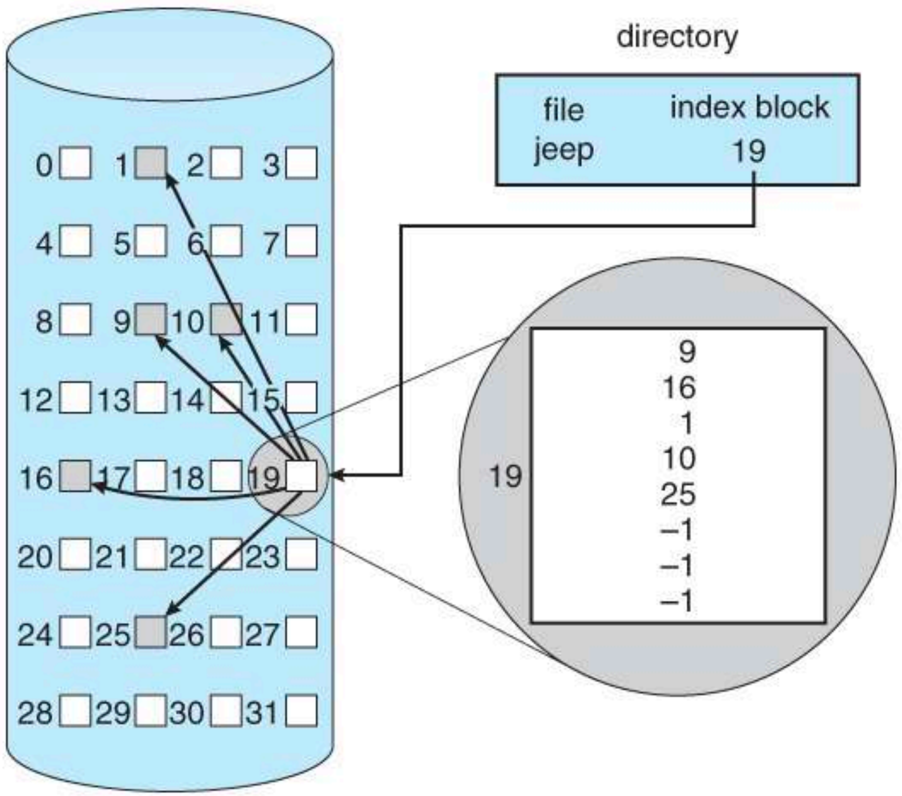
The size of the file descriptor is the same for all files

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)

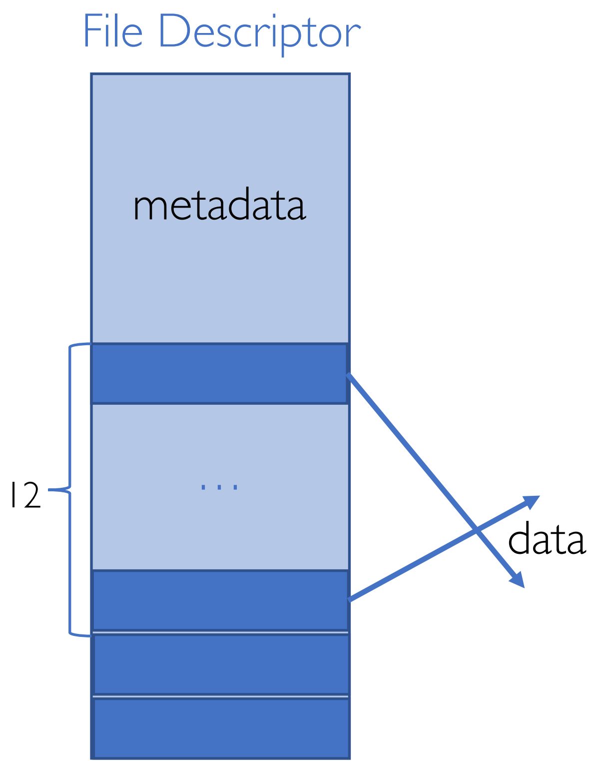
Indexed Files: PROs and CONs

- 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

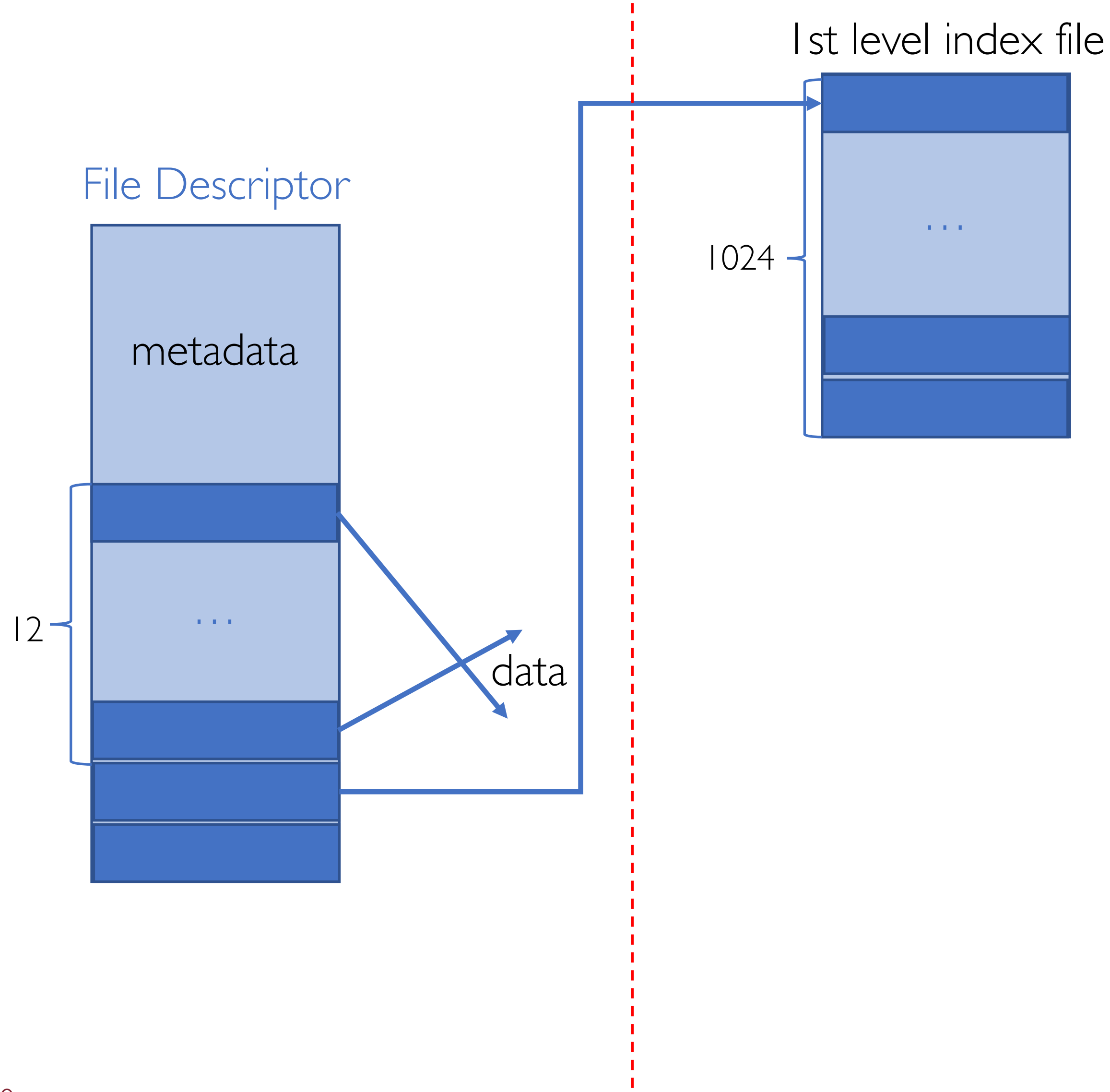
Option 4: Multi-Level Indexed Files

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

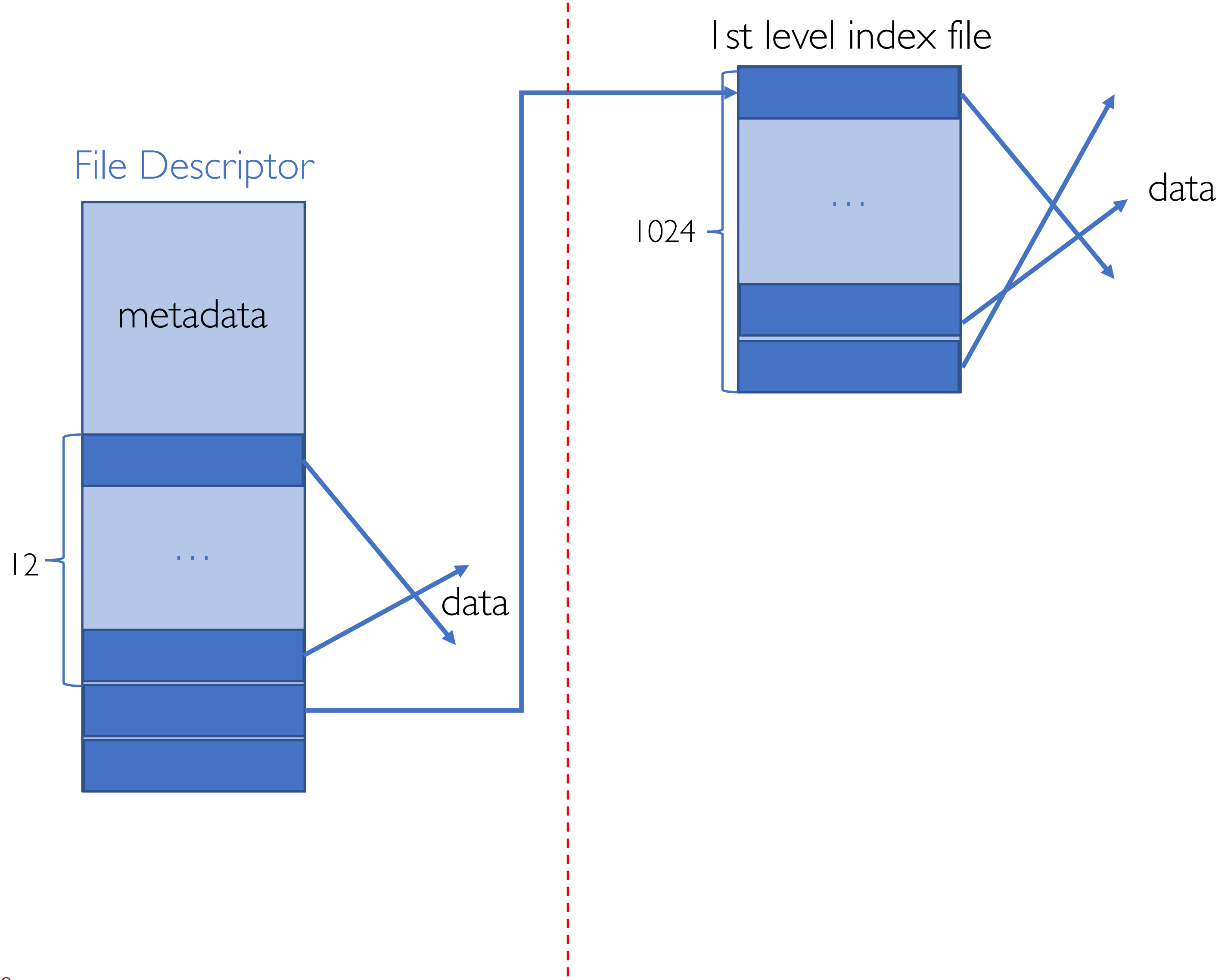
Option 4: Multi-Level Indexed Files



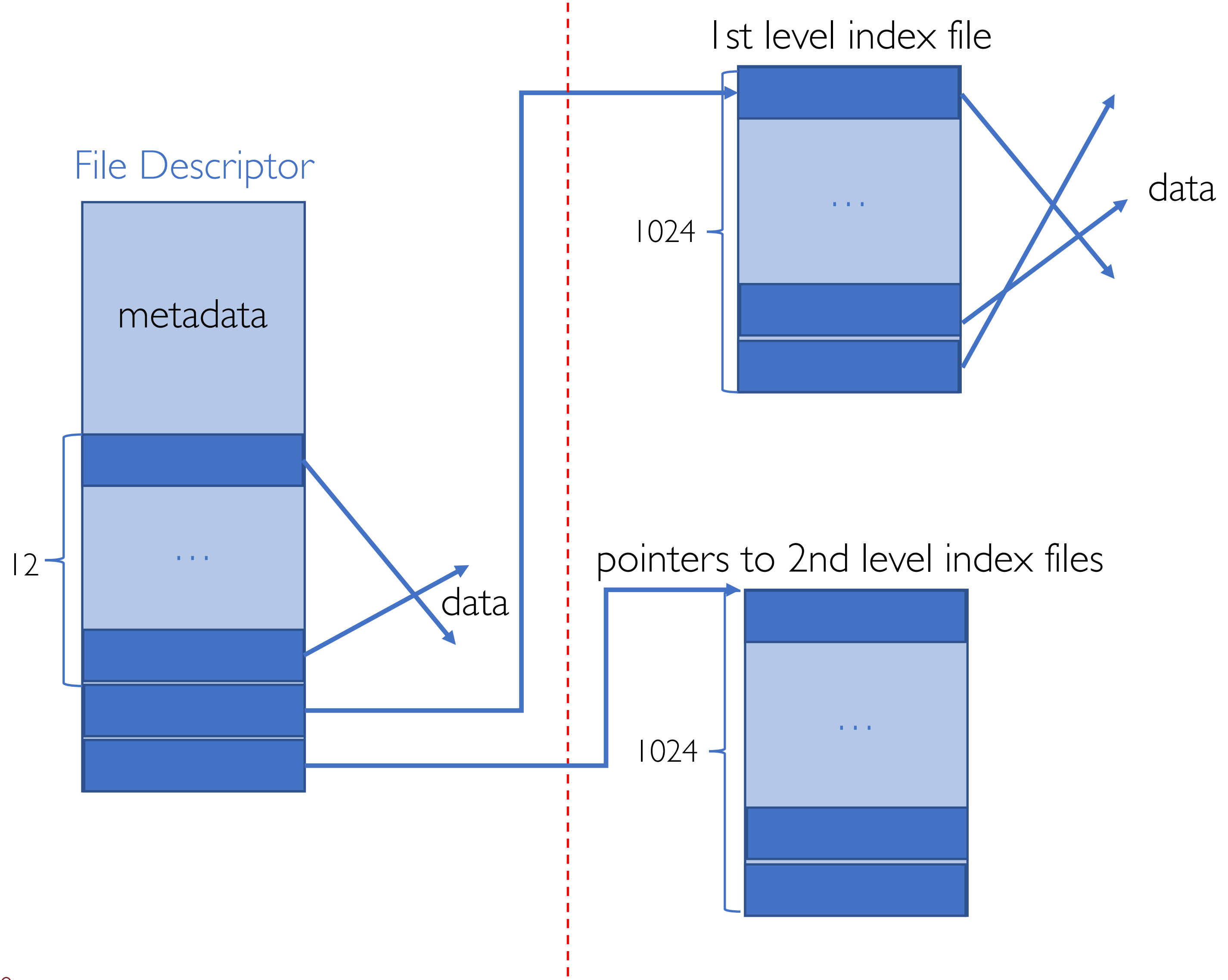
Option 4: Multi-Level Indexed Files



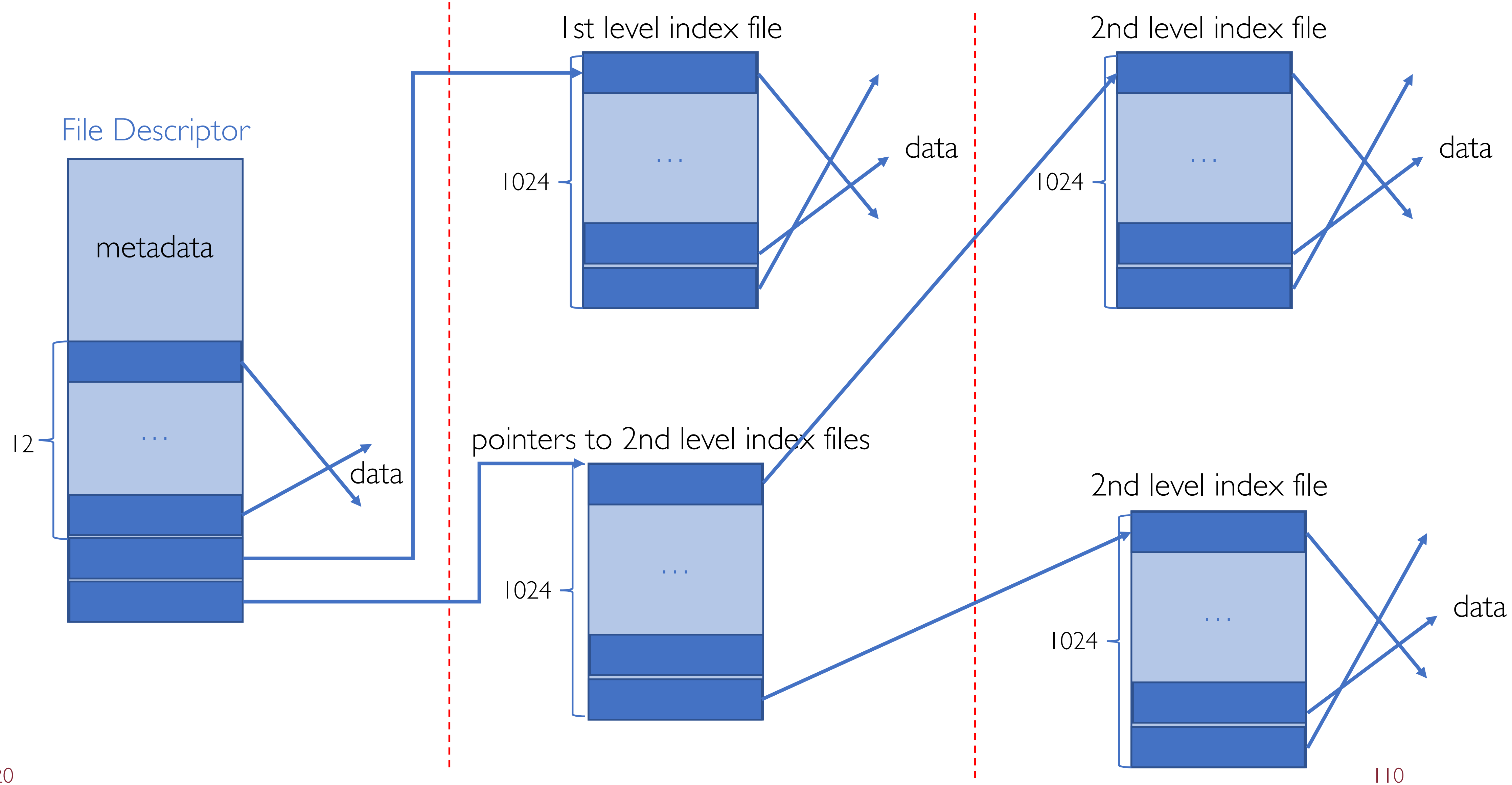
Option 4: Multi-Level Indexed Files



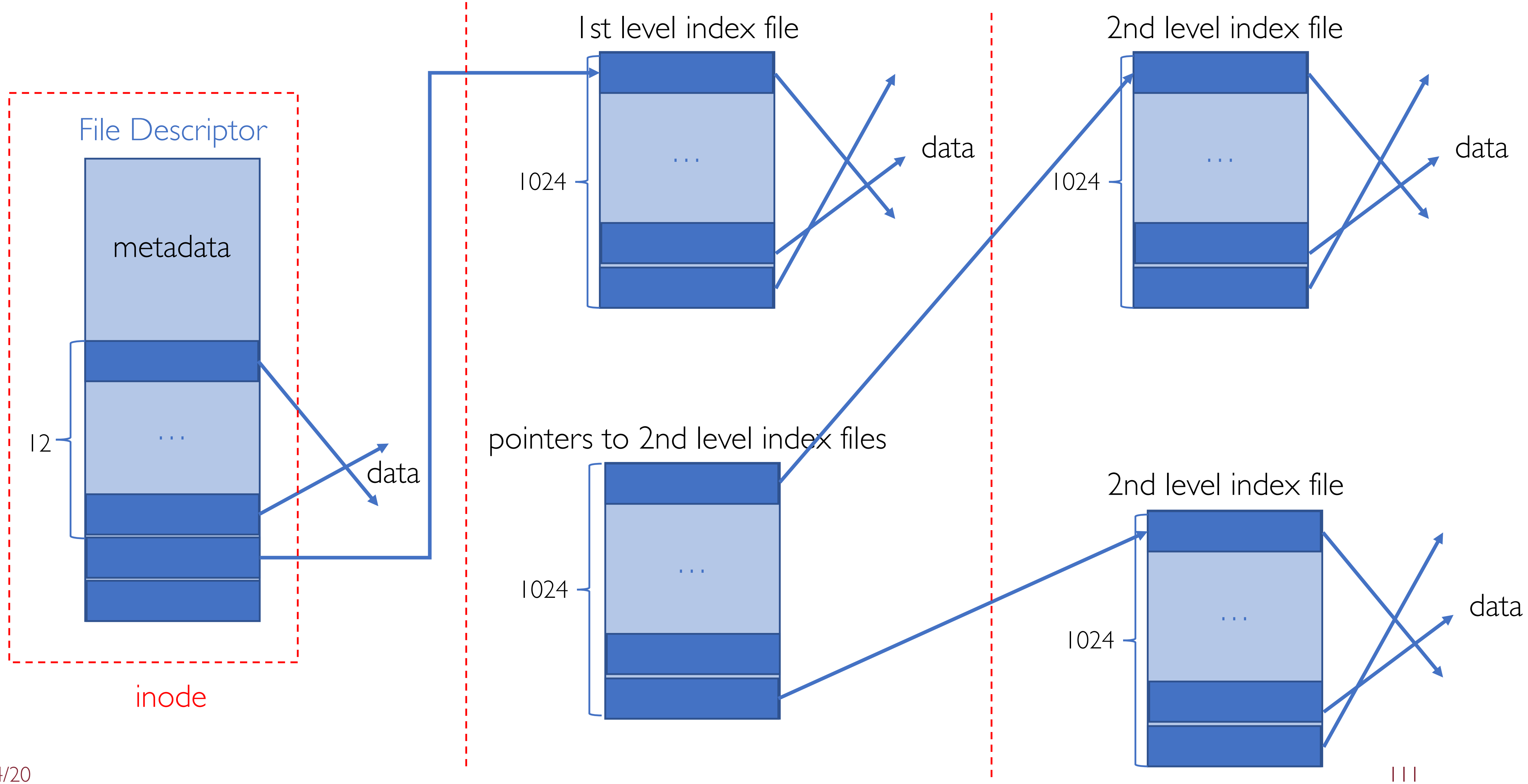
Option 4: Multi-Level Indexed Files



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

$1024^2 + 1024 + 12 \sim 1$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, $\sim k^l$ if k = n. of block pointers and l = 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 pages 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

Problem:

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

⇓

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

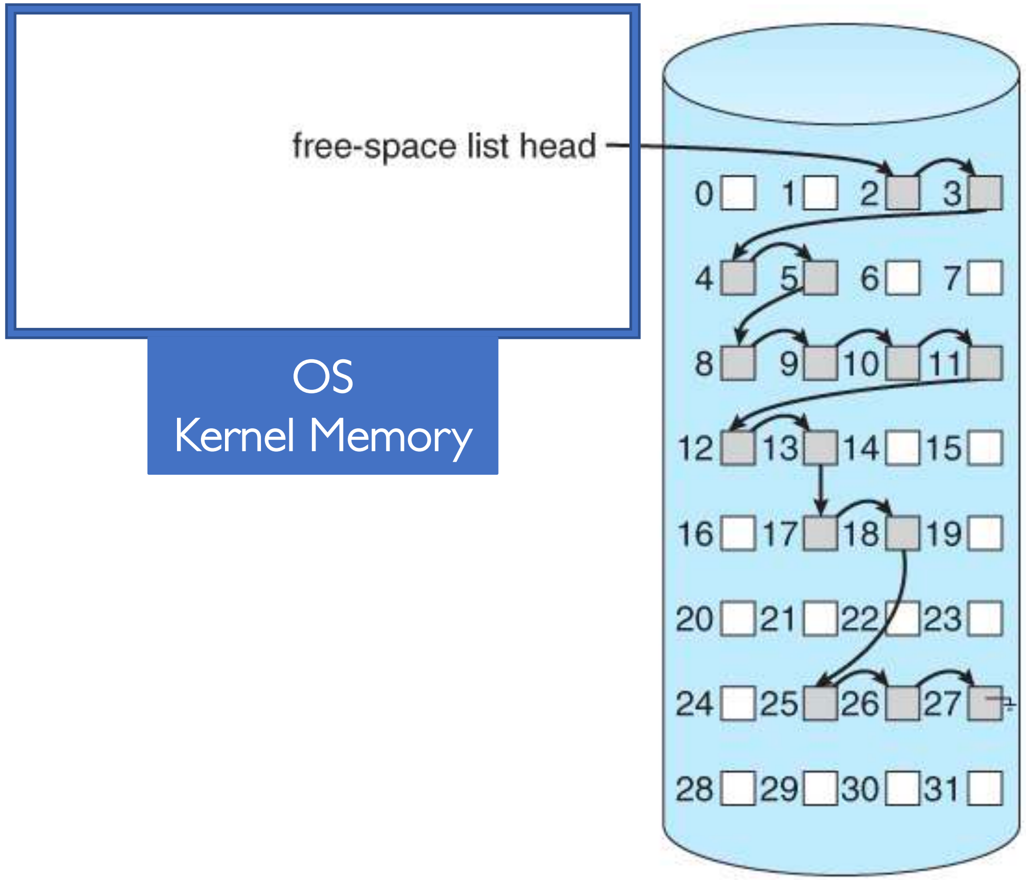
⇓

~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