Operating Systems
File Systems

David Hay
Dror Feitelson

Back to Lecture 1…

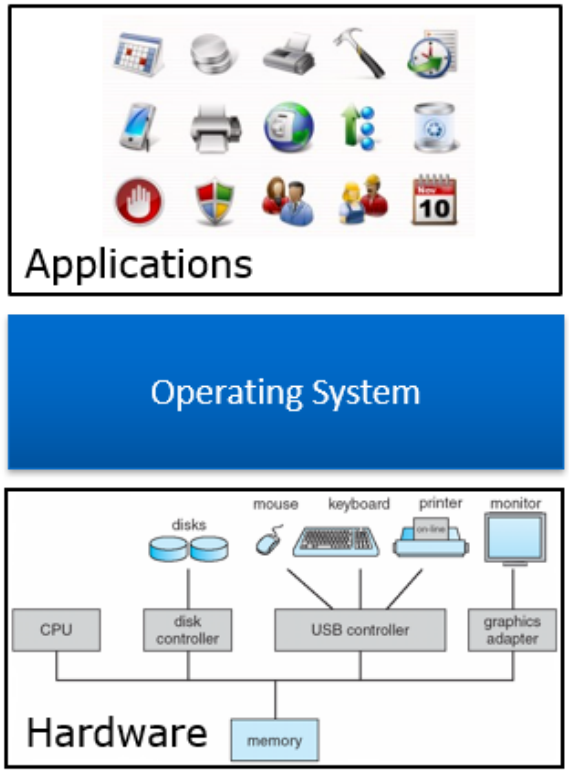
So What Does the Operating System Do?

- A **program** that provides an **environment** within which other programs can do useful work
  - Activates hardware devices for the applications
  - No useful work on its own!
  - Reacts to events (reactive program)
  - Always there to help (resident program)
- Make the computer system convenient and safe to use
  - ➲ abstraction and virtualization
- Use the computer hardware in an efficient manner
  - ➲ resource management

Back to Lecture 1…

So What Does the Operating System Do?

- A **program** that provides an **environment** within which other programs can do useful work
  - Activates hardware devices for the applications
  - No useful work on its own!
  - Reacts to events (reactive program)
  - Always there to help (resident program)
- Make the computer system convenient and safe to use
  - abstraction and virtualization
- Use the computer hardware in an efficient manner
  - resource management



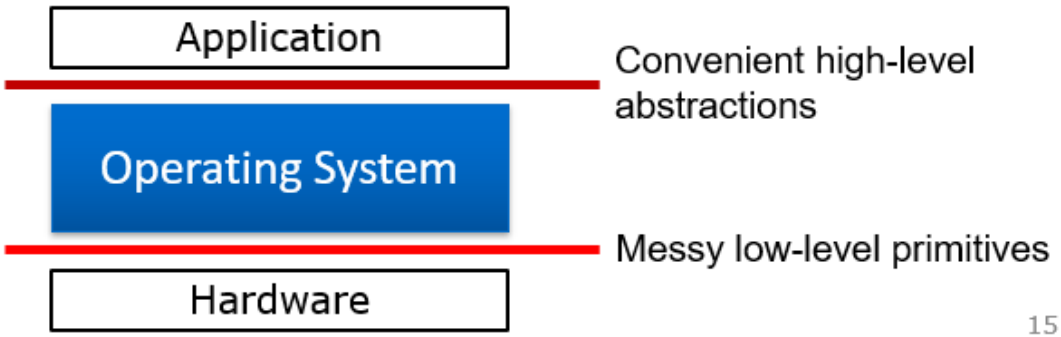
12

Back to Lecture 1…

Abstractions

The OS presents applications with an abstract machine

- More powerful: includes new abstractions that do not exist in the hardware
- Simpler: does not include all the complexities the OS has to deal with



15

Abstraction

A prominent abstraction provided by the operating system is FILES

So what is the essence of the file abstraction?

Files

File = named persistent sequential data storage

- Sequential: data within file is ordered (and may have some structure)
- Persistent: you can go for a trip around the world and when you come back it will still be there
- Named: you can find it using its name

Naming

- Names are external to the operating system
- Exist in many contexts
  - Access shared memory segments (shmget, shmat)
  - Domain name service to find web hosts
  - Well-known ports for certain services
- You need to remember the name to access the file
  - Or find it in a list (which could be long…)
- Modern alternative: search by content

Persistence

- Files are stored on non-volatile devices
  - Disks and other magnetic media
  - SSD (non-volatile memory)
  - Data is retained when computer turned off
- Survive the process which created the data
  - Typically one process creates the file, and others read the data later
  - As opposed to memory which is created and dies with the process

Side Note on Longevity

Several problems with data access after long periods of many years:

- Media deterioration
  - Papyrus in desert cave preserved 2000 years
  - Magnetic tape on shelf usable for 10-20 years?
- Media accessibility
  - No reader available for that magnetic tape
- Data format
  - What do those bits mean actually?

Structure and Meaning

- Unix: data is a sequence of bytes
  - Can represent text
  - Can represent pictures (e.g. in jpeg format)
  - Can represent video or sound
  - Can represent program source code or binary
  - Can represent numerical data
- IBM mainframes: data is a sequence of records (for databases)
- Windows NTFS: set of attribute-value pairs
  - E.g the file's icon, indication of internet source
  - Always has an unnamed data stream with contents

Filename Extensions

- Name divided into 2 parts, name and extension
- On UNIX, extensions are not enforced by OS
  - However, applications (e.g. C compiler) might insist on its extensions
    - These extensions are very useful for C
- Windows attaches meaning to extensions
  - Tries to associate applications to file extensions

file.bak -- Backup file
file.c -- C Source program
file.gif -- Compuserve Graphical Interchange Format image
file.hlp -- Help file
file.html -- Hyper Text Markup Language document
file.jpg -- Still picture encoded with JPEG standard
file.mp3 -- Music encoded in MPEG layer 3 audio format
file.mpg -- Movie encoded with MPEG standard
file.zip -- compressed archive

Typical file extensions

Separation of Concerns

Operating system

- Provide infrastructure and support
- Store and retrieve the data
- Keep metadata about the data
- Don’t limit the application

Application

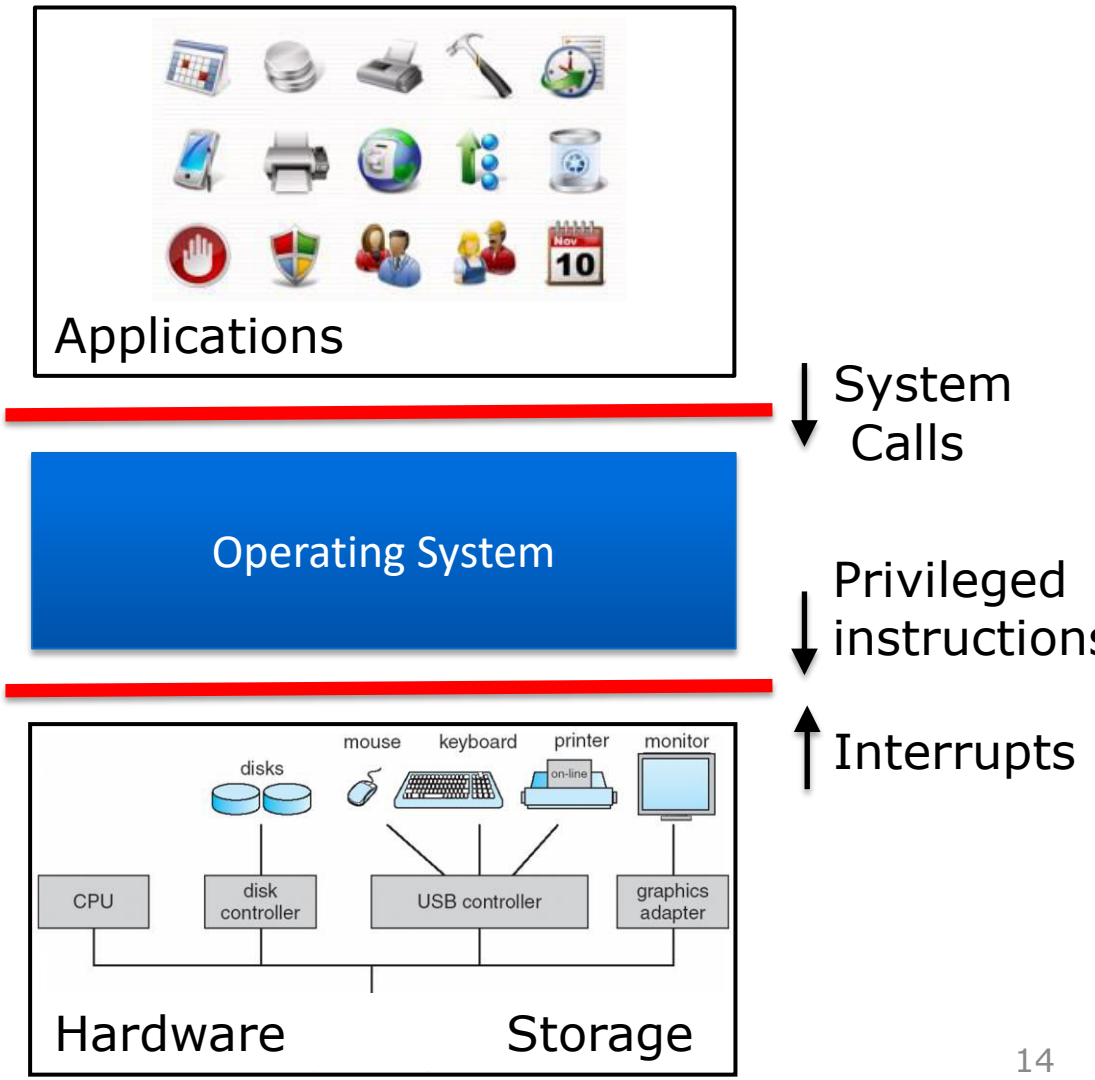
- Do interpretation and processing
- Create and use the data
- Decide on data format (how things are represented)
- Know what the data means

File System

- File system API
  - Files concept and attributes
  - Directories and naming
  - Sharing and protection
- File system implementation
  - Bookkeeping for storing data
  - Space allocation methods and management
  - Reliability and performance

File System Operations

- FS operations are **system calls**
- OS implementation uses **privileged instructions** to activate storage devices
- Devices report completion using interrupts



Basic File System Operations

Basic File System Operations

- Create a file

Basic File System Operations

- Create a file
- Delete a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file
- Close a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file
- Close a file
- Write to a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file
- Close a file
- Write to a file
- Read from a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file
- Close a file
- Write to a file
- Read from a file
- Seek to somewhere in a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file
- Close a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Truncate a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file
- Close a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Truncate a file
- Set attributes of a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file
- Close a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Truncate a file
- Set attributes of a file
- Get attributes of a file

Basic File System Operations

- Create a file
- Delete a file
- Open a file
- Close a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Truncate a file
- Set attributes of a file
- Get attributes of a file
- Rename a file

File Attributes

- File-specific info maintained by the OS (METADATA)
  - Varies a lot across different OSes
- User facing:
  - Name – needed to identify file, kept in human-readable form
  - Type – needed for systems that support different types
  - Size – current file size
  - user identification – data for protection and security
  - Access rights – controls who can read, write, execute
  - Time, date - usage monitoring, make
- For internal OS use:
  - Identifier – unique tag (number) identifies file within file system
  - Location – pointer to file location on device

FILES NAMESPACE

File Naming

- Motivation: to re-access data that was stored previously
  - You do not need to remember block, sector, …
  - We have human readable names!
    - One process creates a file and gives it a name
    - Later another process can access the file using this name
- Naming conventions are OS dependent
  - Usually names as long as 255 characters are allowed
  - Digits and special characters (spaces) are sometimes allowed
  - MS-DOS and Windows are not case sensitive, UNIX family is
- Naming administration is done separately of the data

Namespace Organization

Files are organized in a hierarchy of directories, which aim to obtain:

Namespace Organization

Files are organized in a hierarchy of directories, which aim to obtain:

- **Efficiency** – locating a file quickly
  - Avoid one list with all the files

Namespace Organization

Files are organized in a hierarchy of directories, which aim to obtain:

- **Efficiency** – locating a file quickly
  - Avoid one list with all the files
- **Separation** – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names

Namespace Organization

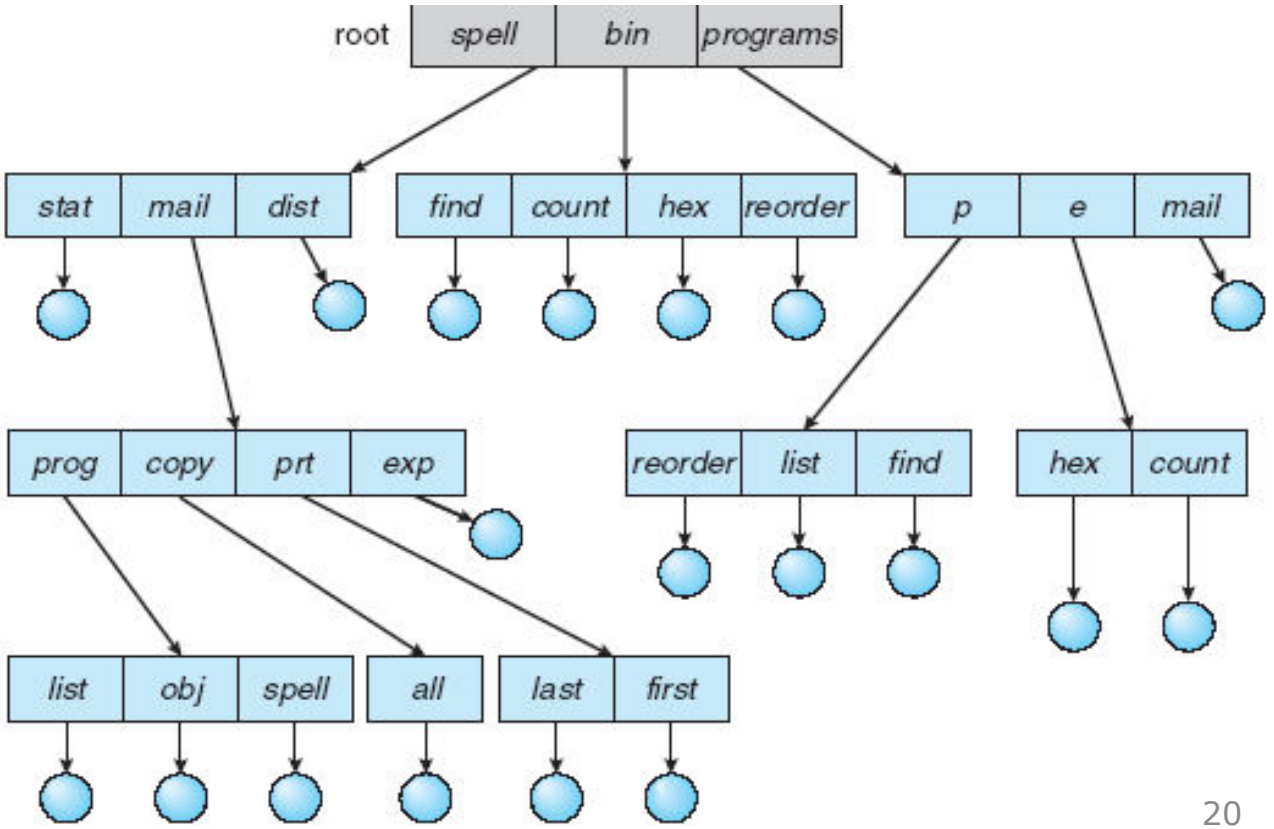
Files are organized in a hierarchy of directories, which aim to obtain:

- **Efficiency** – locating a file quickly
  - Avoid one list with all the files
- **Separation** – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- **Grouping** – logical grouping of files by properties or usage
  - e.g., all Java programs, all games, …
  - Or all the files related to ex 4 in OS

Tree-Structured Namespace

Filesystem is a tree of arbitrary height

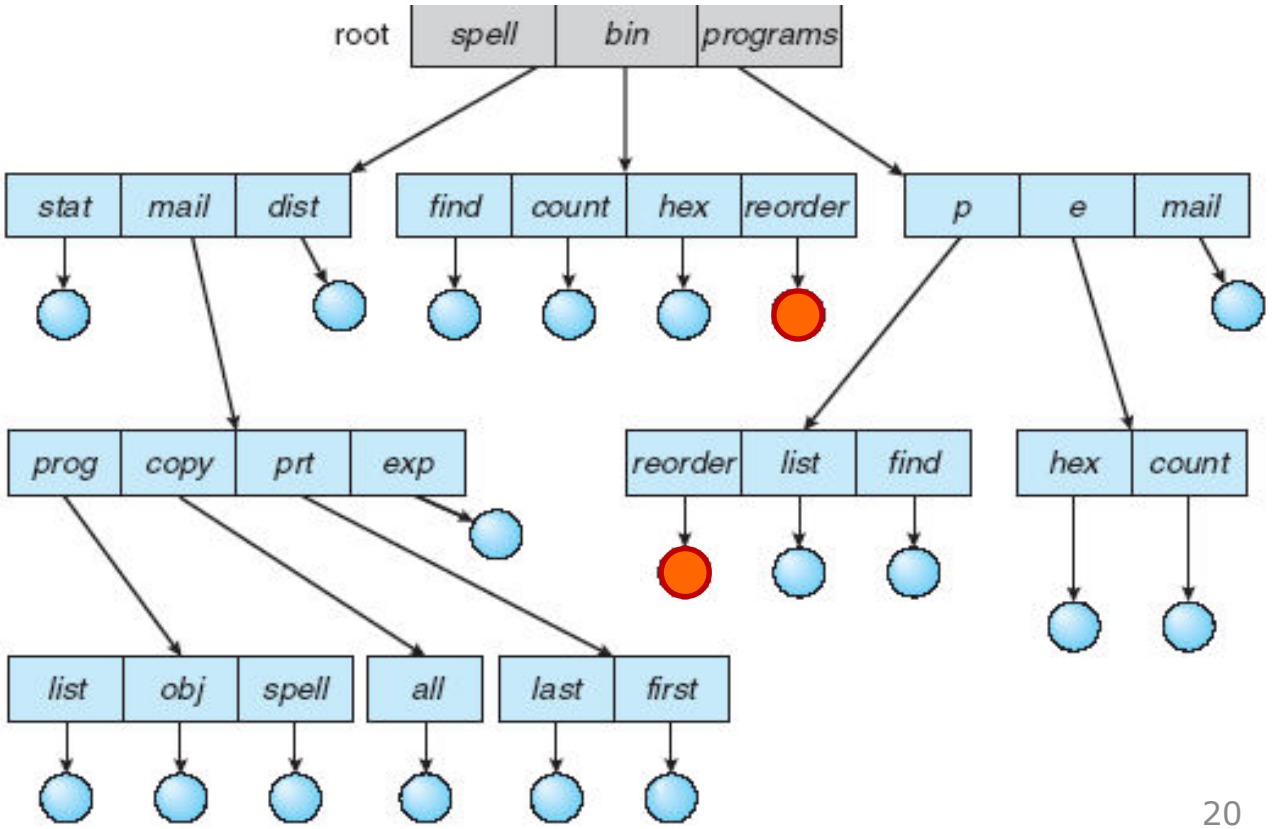
- Nodes are directories and files
  - Internal nodes are directories
- Directories contains files and sub-directories
- Names are relative to a directory
- Files are identified by their path from the root



Tree-Structured Namespace

Filesystem is a tree of arbitrary height

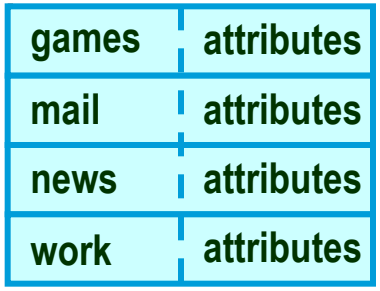
- Nodes are directories and files
  - Internal nodes are directories
- Directories contains files and sub-directories
- Names are relative to a directory
- Files are identified by their path from the root



Directories

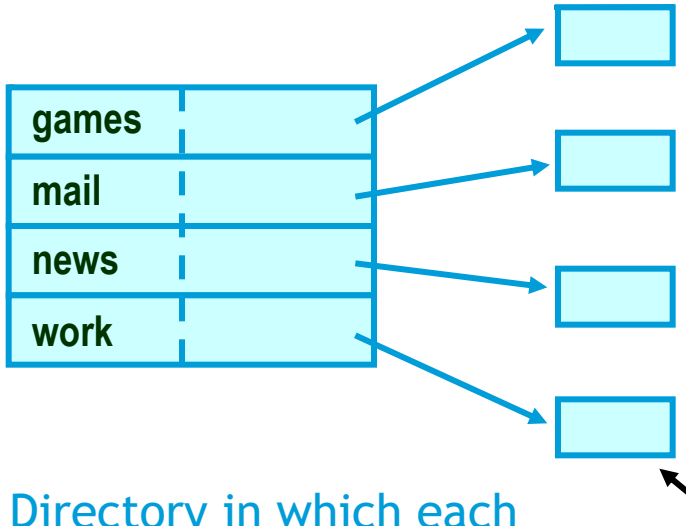
- Directories map names to file objects
  - One level of the naming hierarchy
  - This data is stored in files (so directories are themselves files)
    - A bit in the file object indicates if it is a directory
    - The OS interprets the format of the directory file data
- Ideally use efficient data structure for
  - Search for a file
  - Create a file
  - Delete a file
  - List the files in the directory
  - Rename a file

Implementing Directories



A simple directory

- fixed size entries
- disk addresses and attributes in directory entry
- example: MS-DOS



Directory in which each entry just refers to an i-node

- example: Unix

Data structure containing the attributes

Path Names

- File system has a root directory
- Processes have a working directory
  - Typically the home directory of their user
- To access a file, the process needs to specify the **path** where the file is
- Path names are either absolute or relative
  - Absolute: path of file from the root directory
  - Relative: path from the current working directory
- Path components are separated by / (unix) or \ (windows)
- Most OSes have two special entries in each directory:
  - "." for current directory ".." for parent

Gaining Access to a File

- Need to parse the whole path
- 2 disk accesses per element:

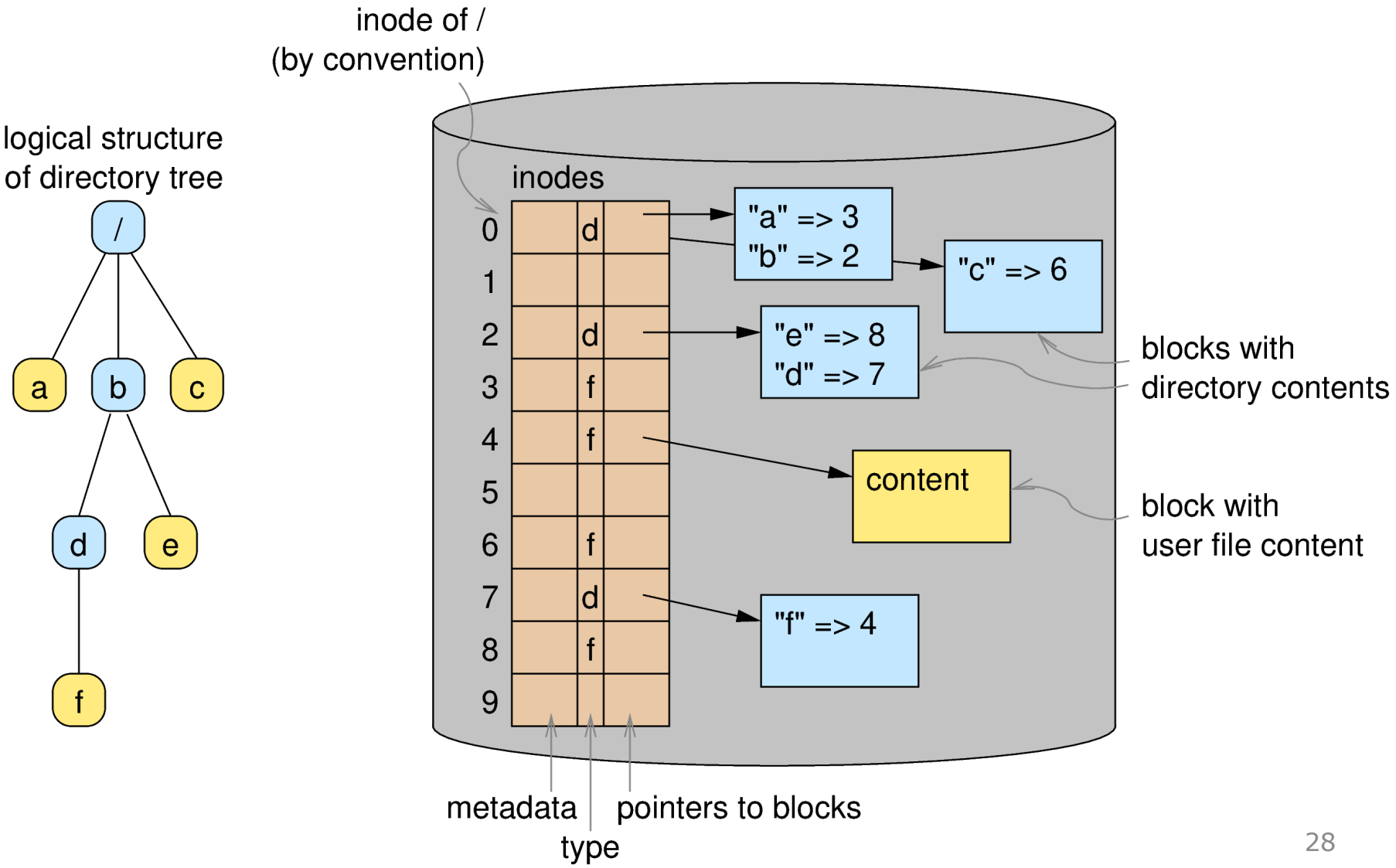
Gaining Access to a File

- Need to parse the whole path
- 2 disk accesses per element:
  - The directory object
    (to find where content is stored)

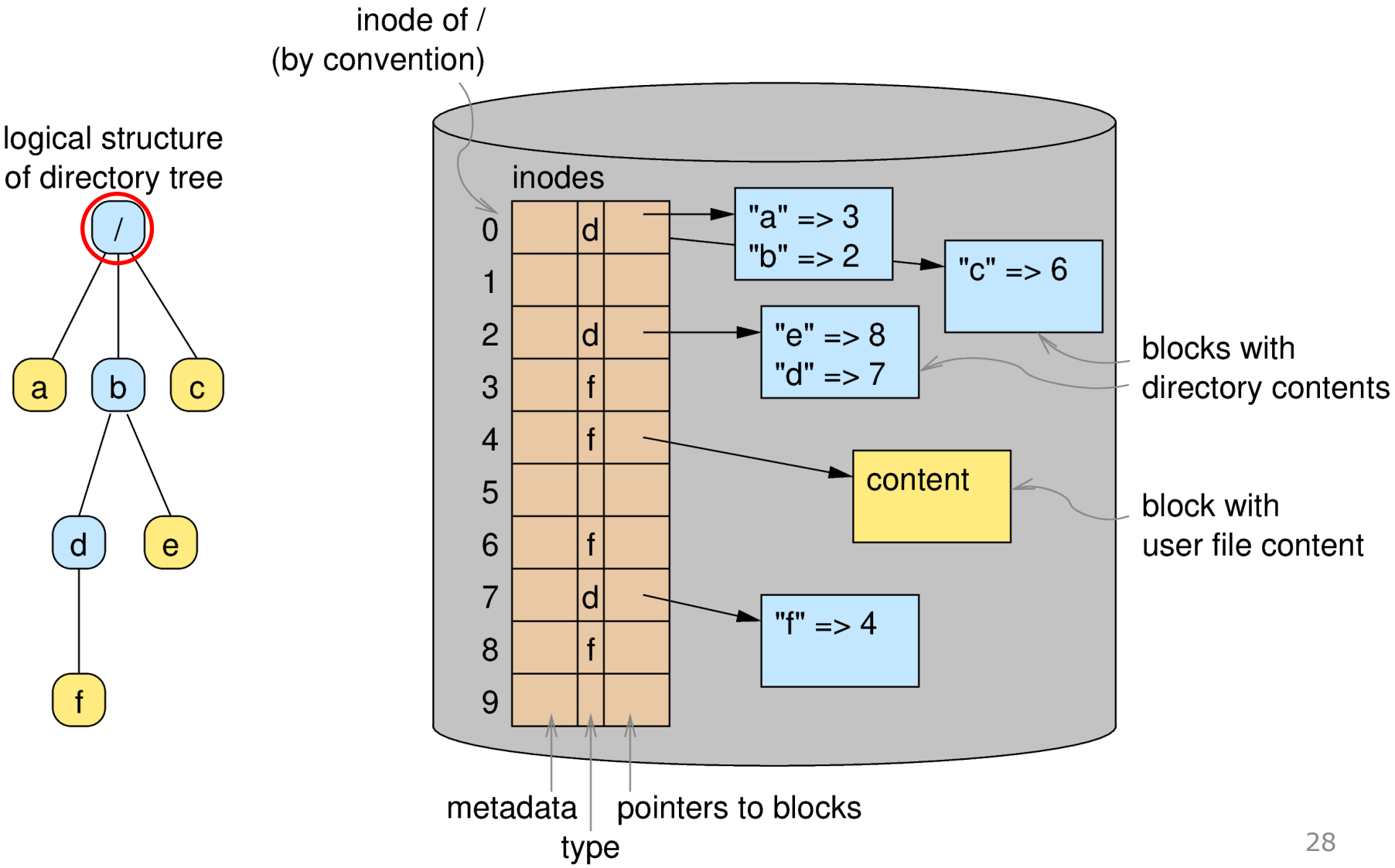
Gaining Access to a File

- Need to parse the whole path
- 2 disk accesses per element:
  - The directory object
    (to find where content is stored)
  - The directory contents
    (to find mapping of name)

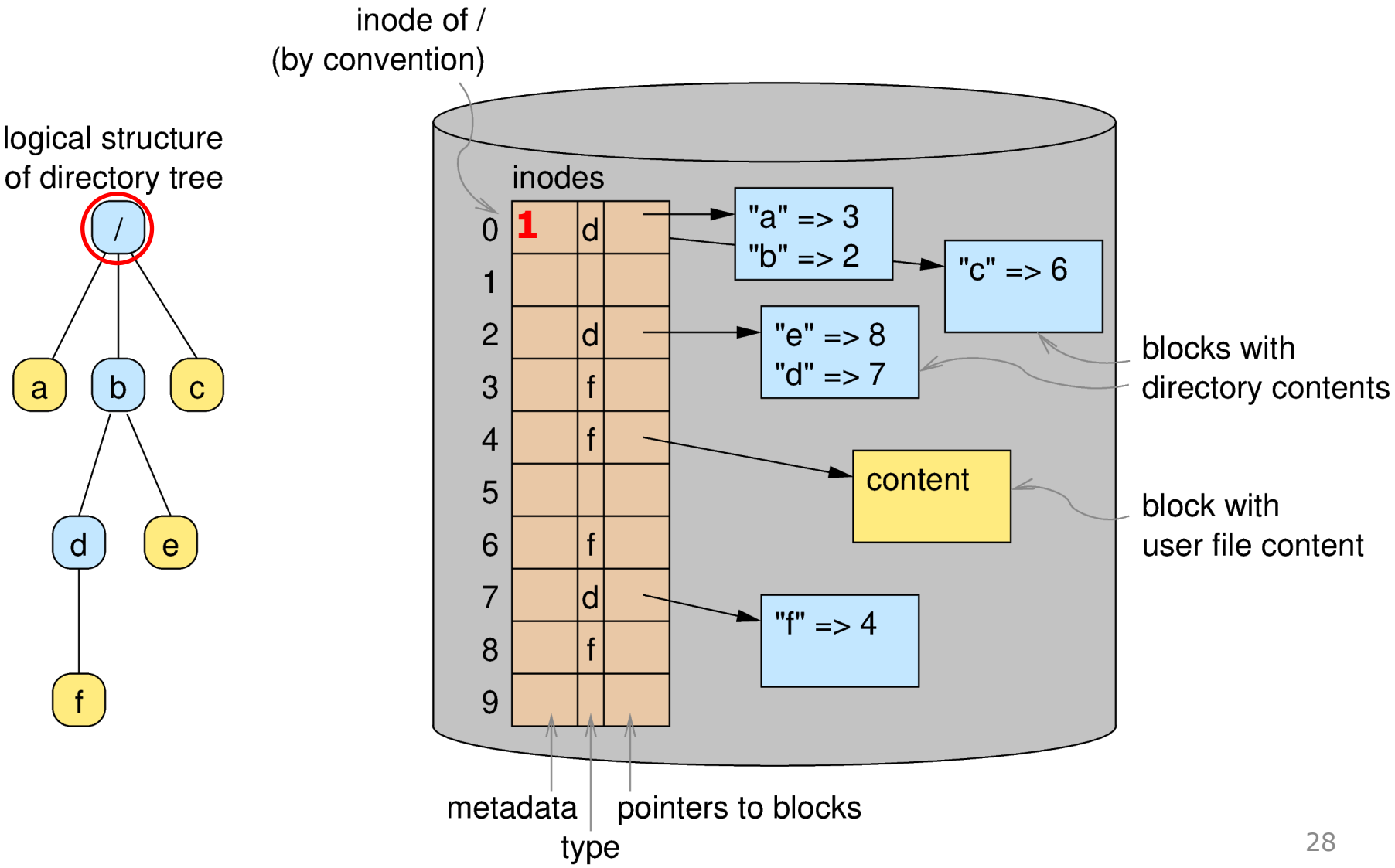
Gaining Access to a File



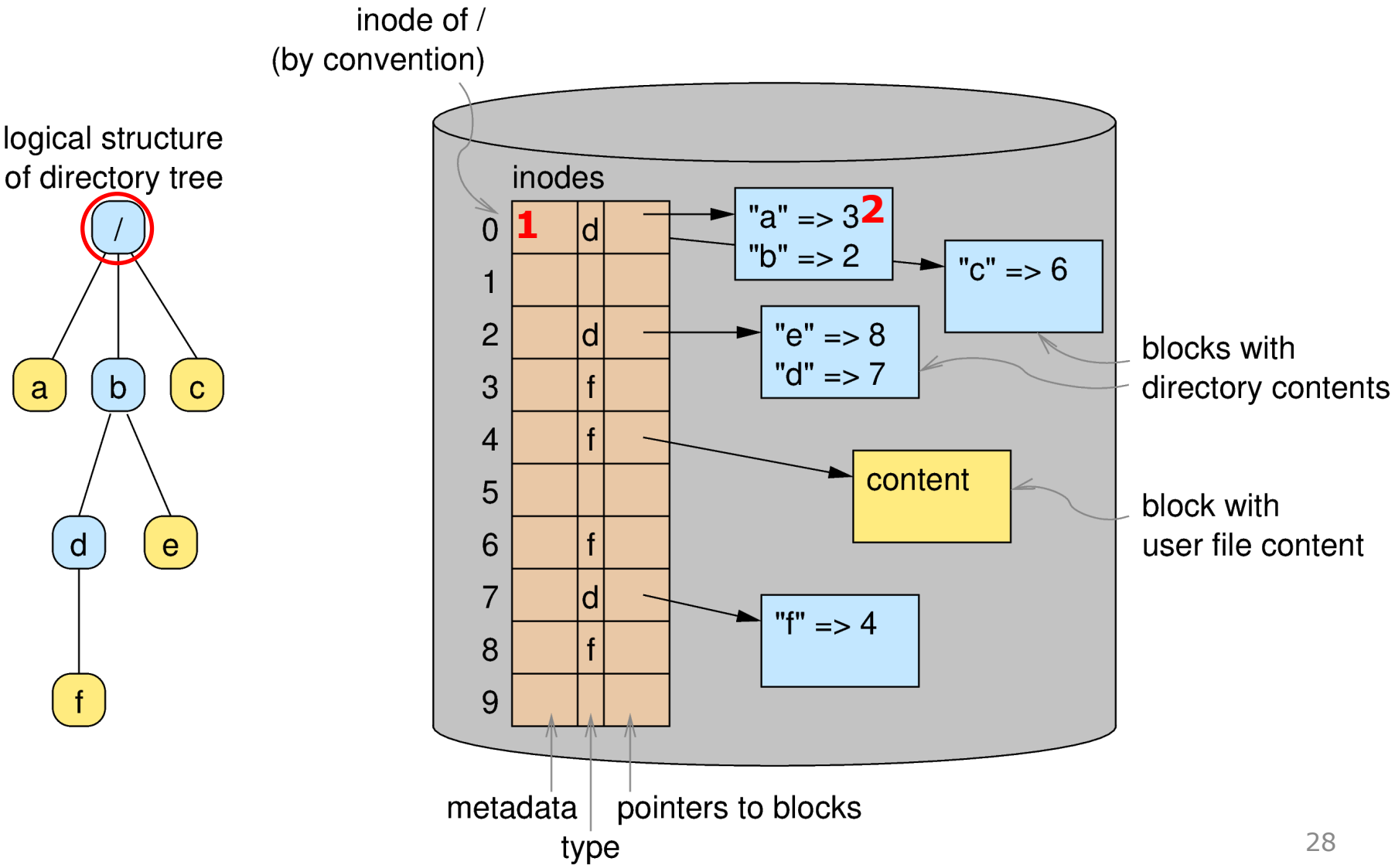
Gaining Access to a File



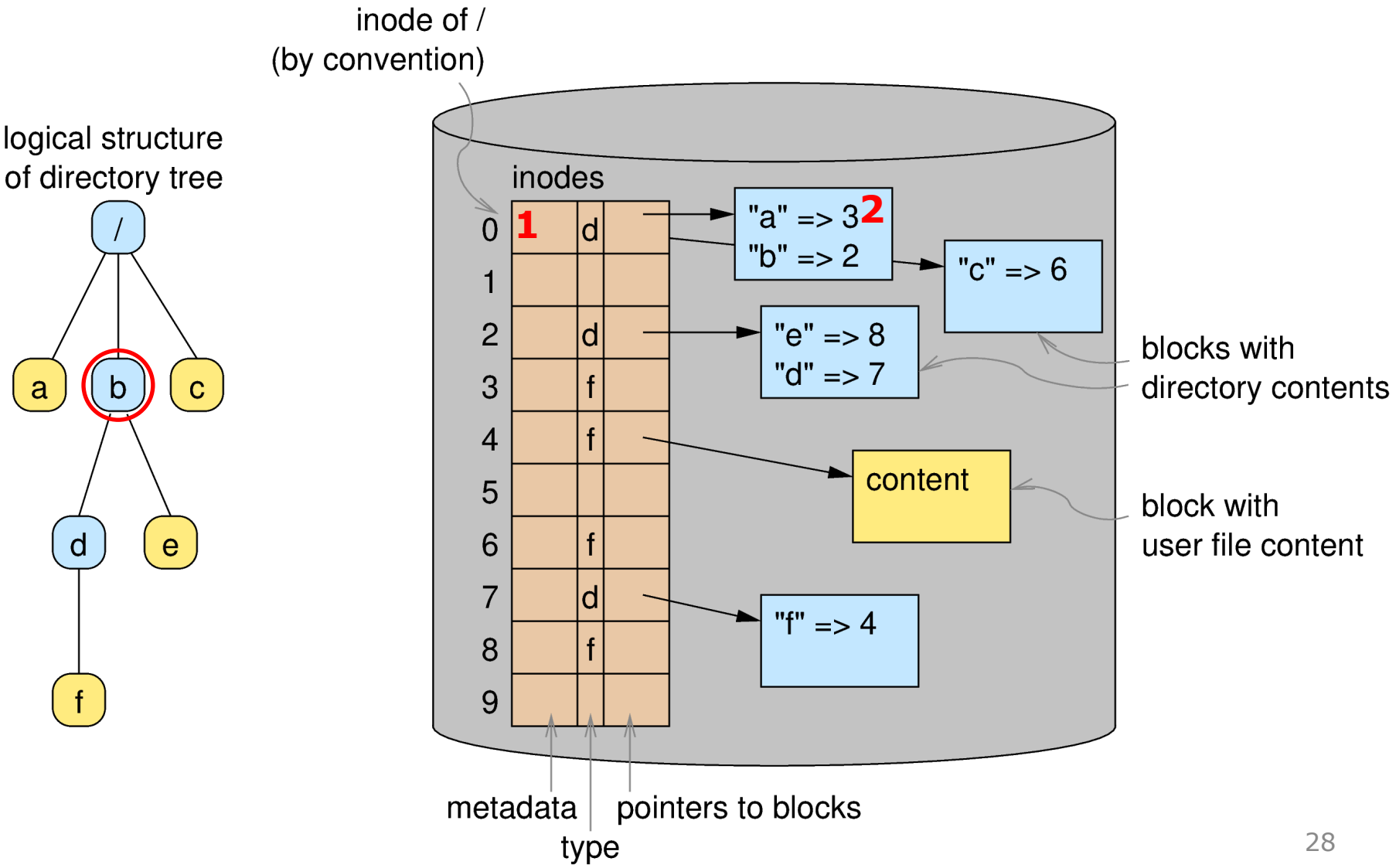
Gaining Access to a File



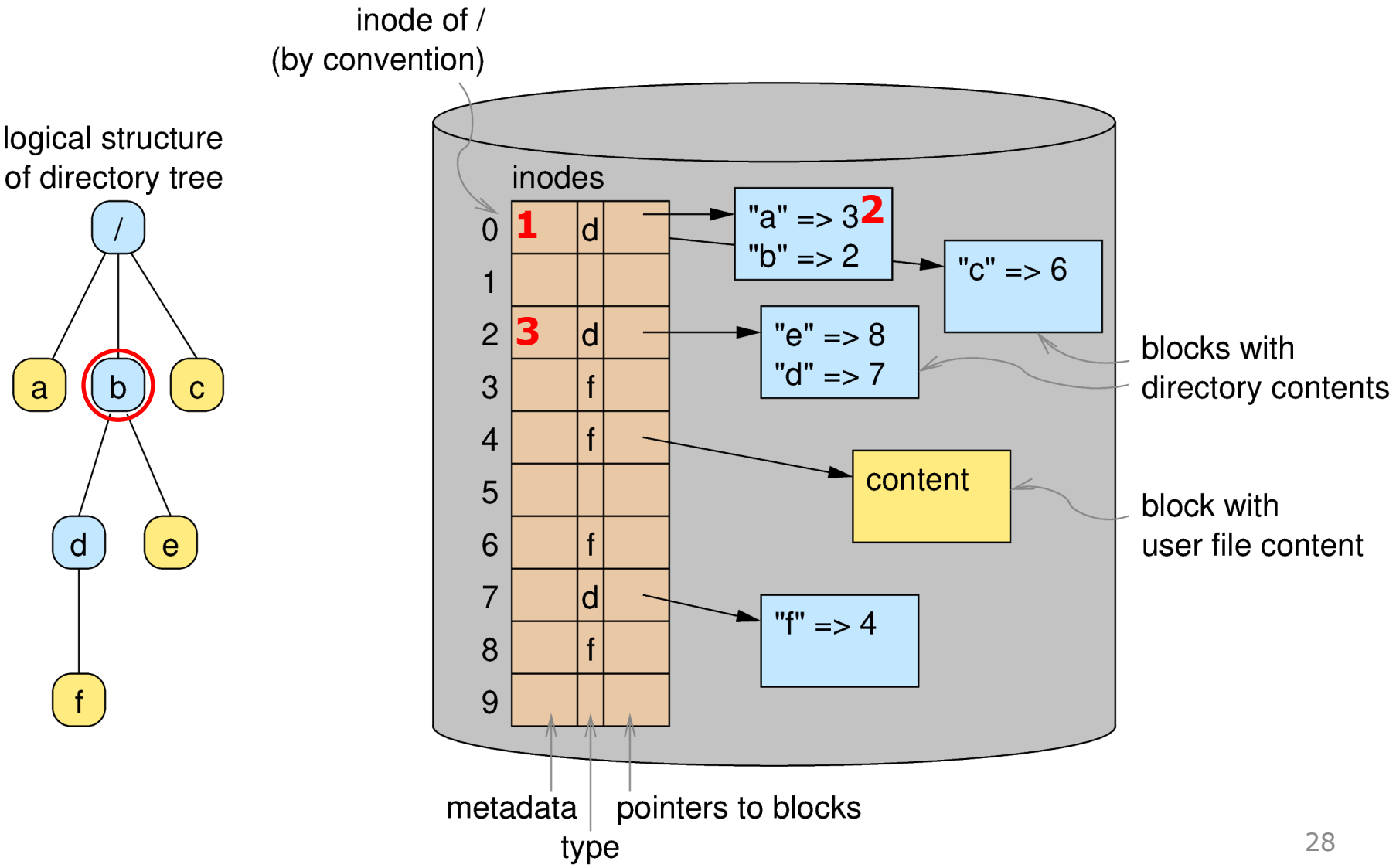
Gaining Access to a File



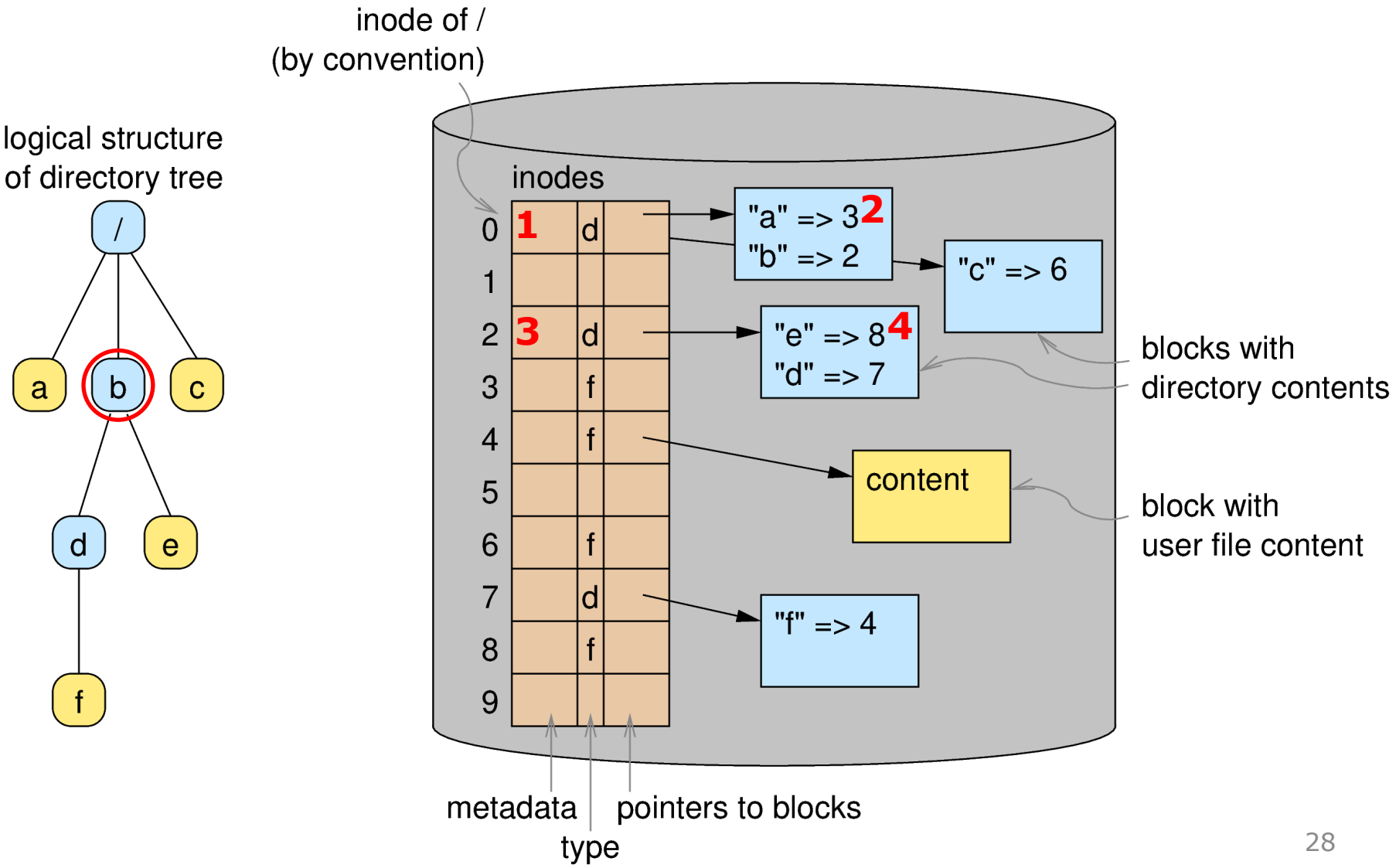
Gaining Access to a File



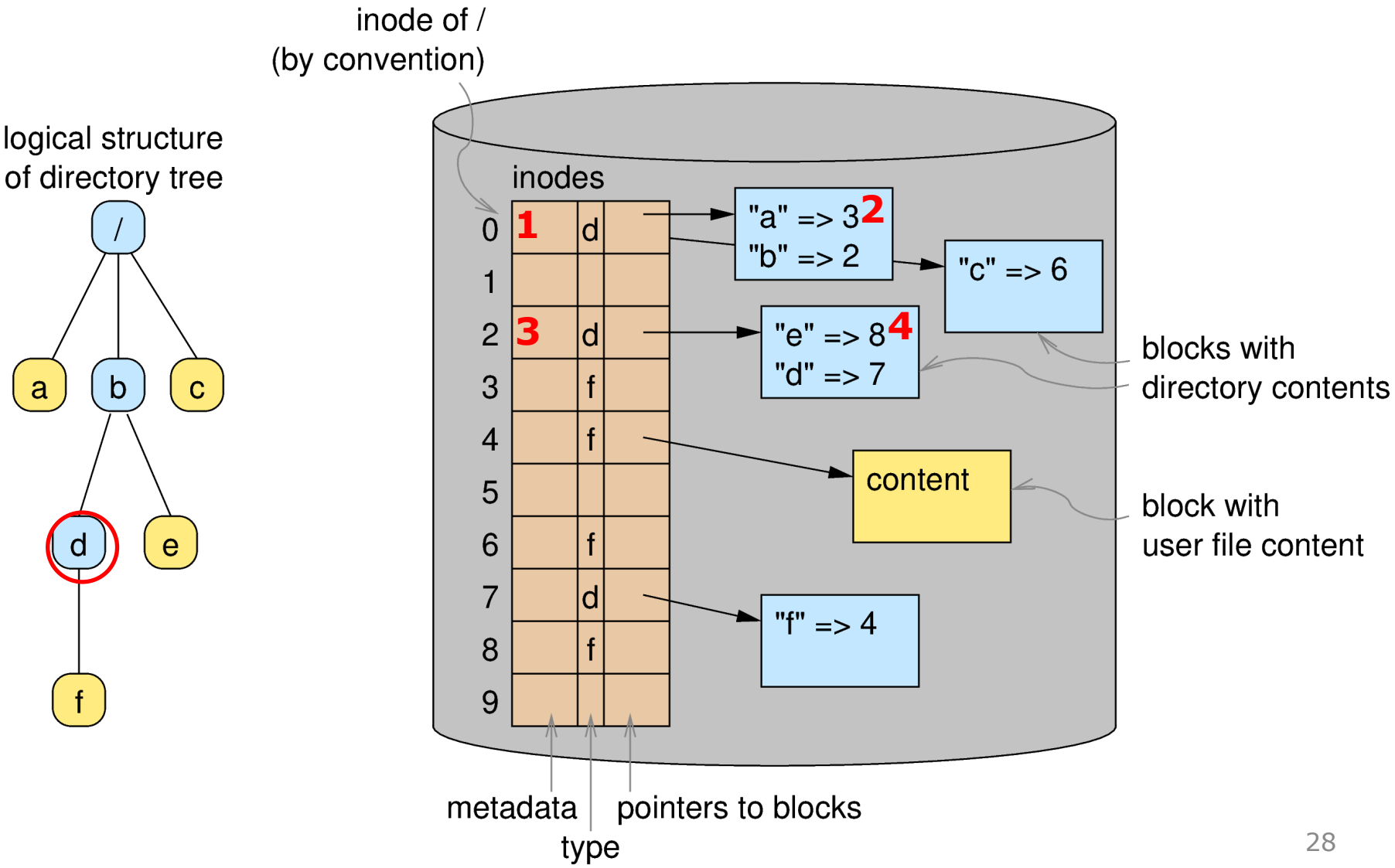
Gaining Access to a File



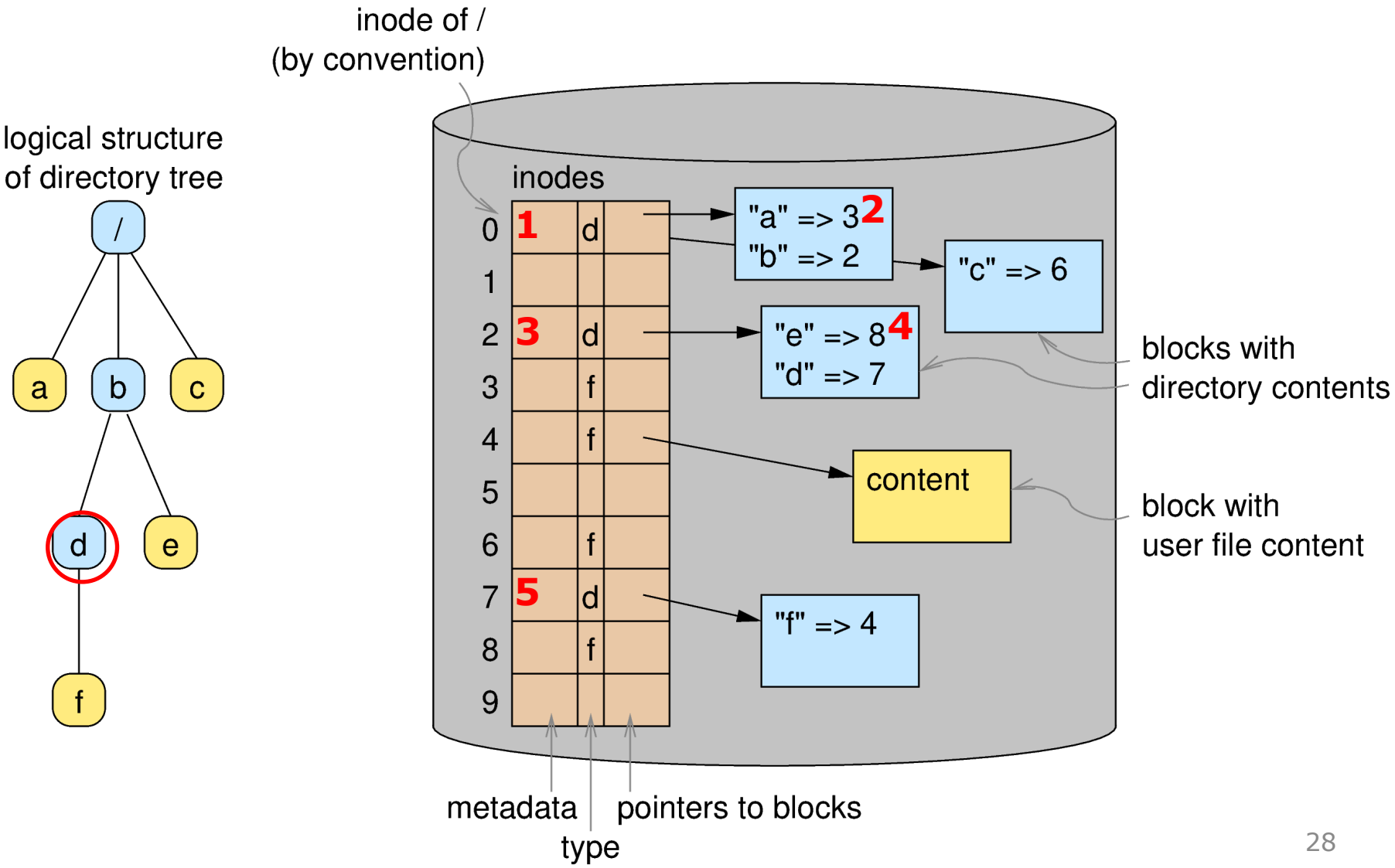
Gaining Access to a File



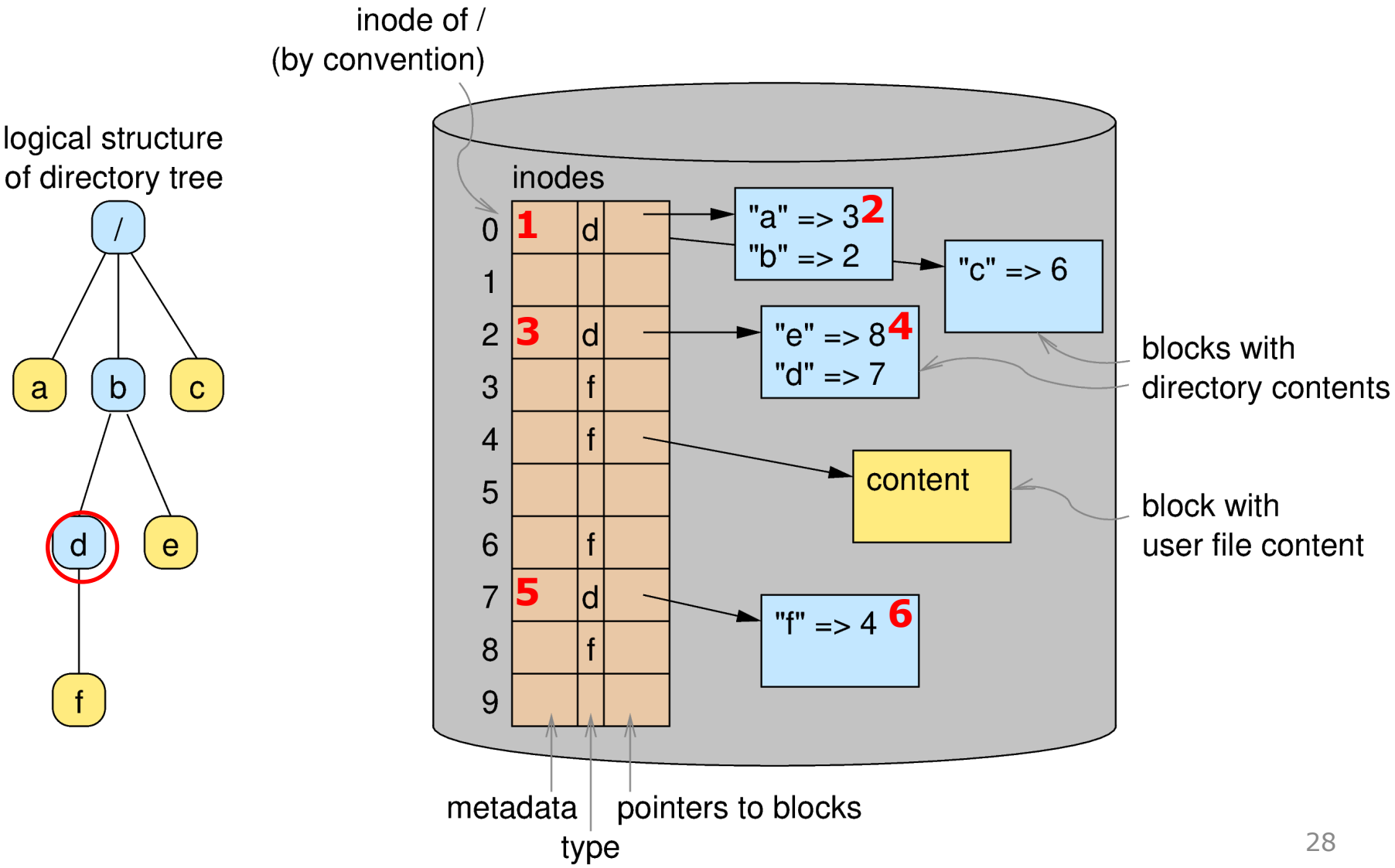
Gaining Access to a File



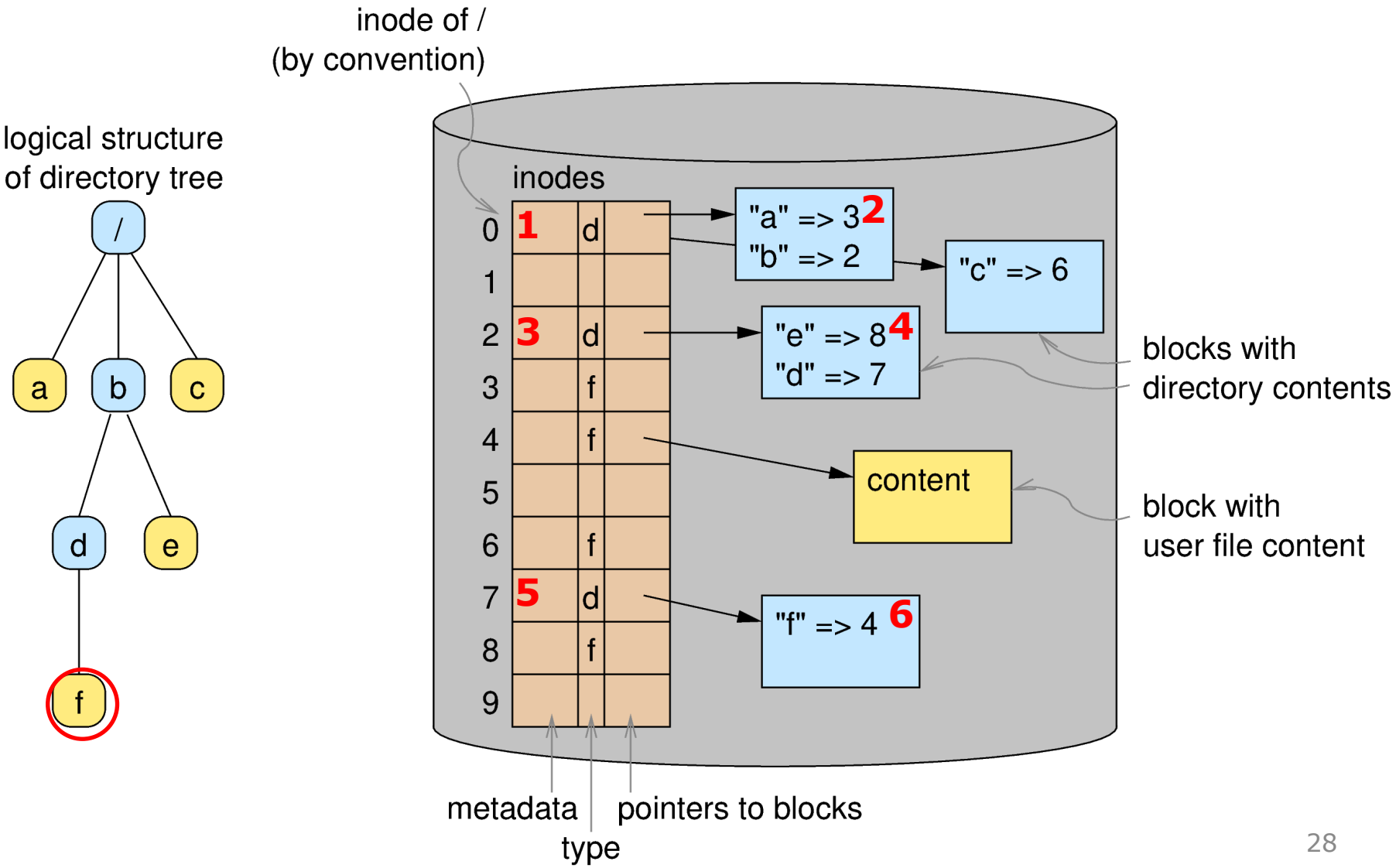
Gaining Access to a File



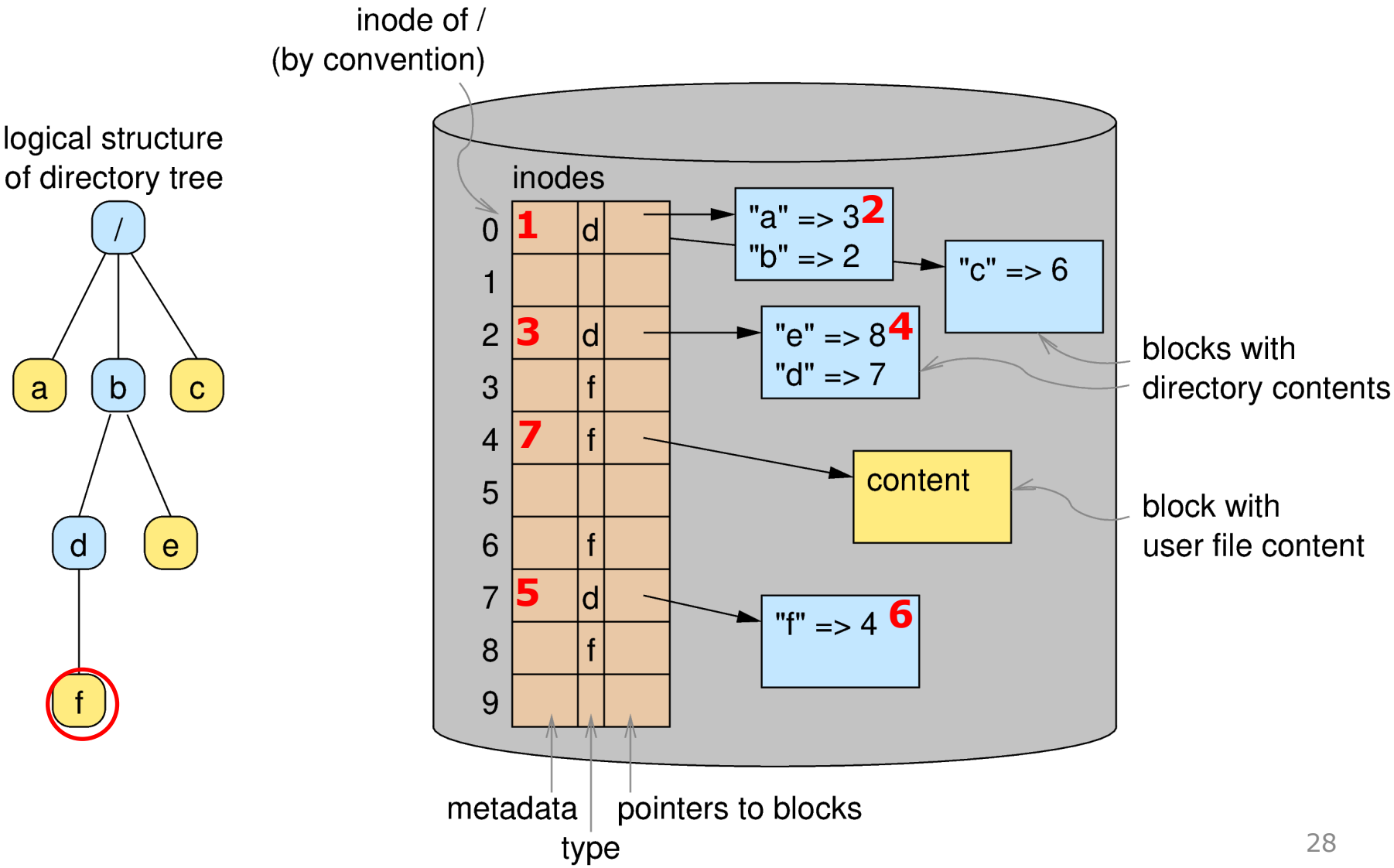
Gaining Access to a File



Gaining Access to a File

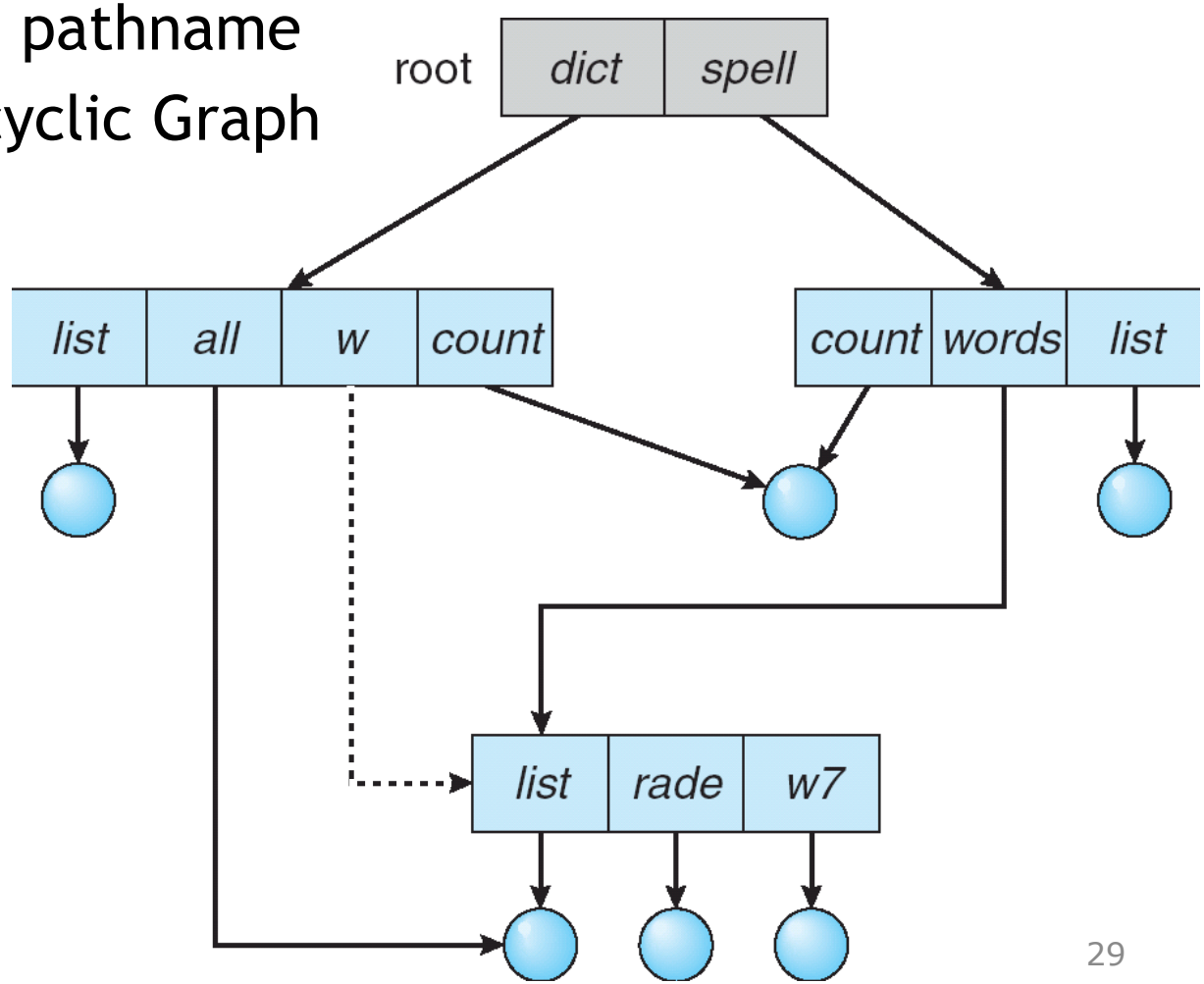


Gaining Access to a File



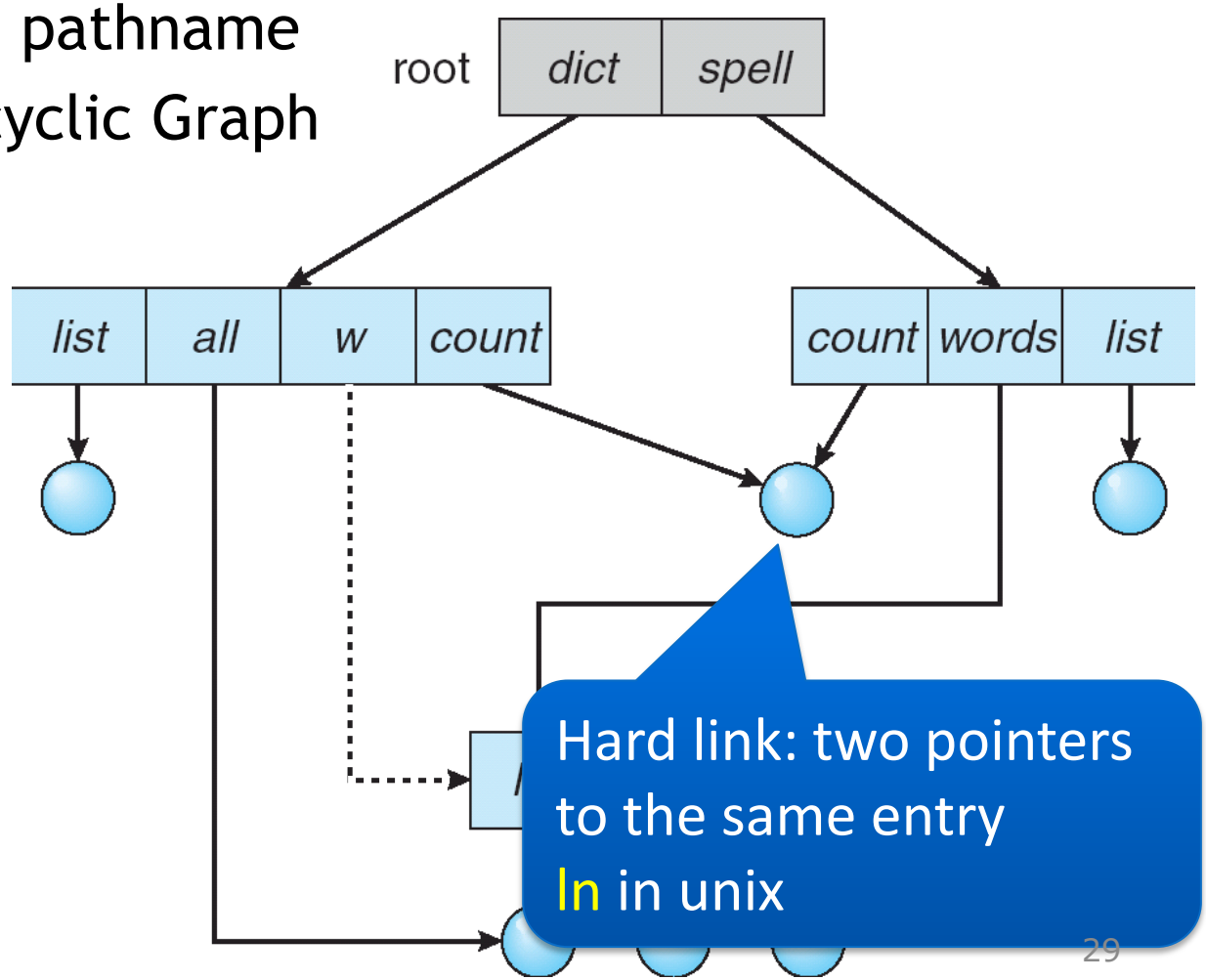
Files Can Have Multiple Names

- Same object linked from multiple directories
- Create an alias to a pathname
- Directories Form Acyclic Graph



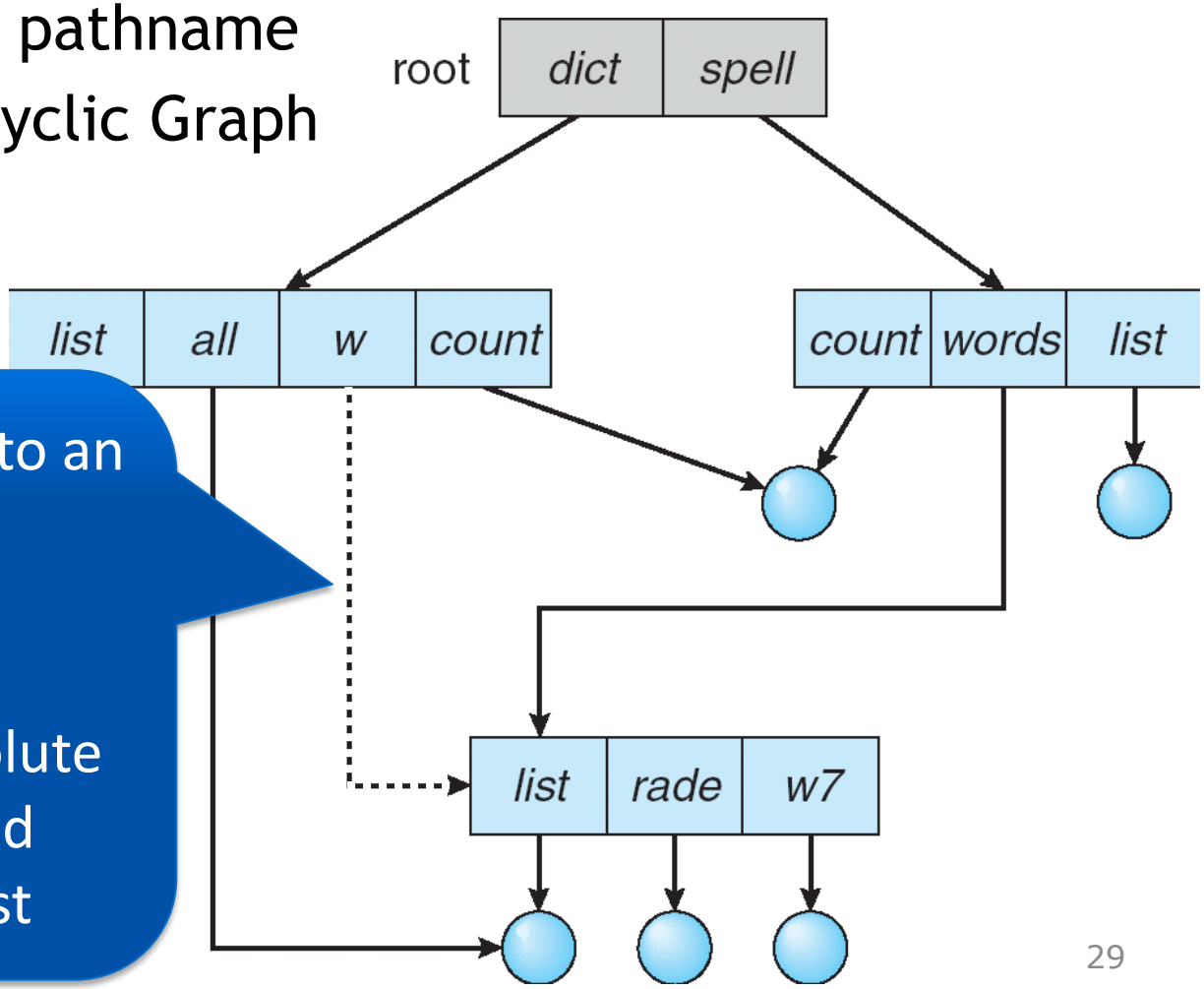
Files Can Have Multiple Names

- Same object linked from multiple directories
- Create an alias to a pathname
- Directories Form Acyclic Graph



Files Can Have Multiple Names

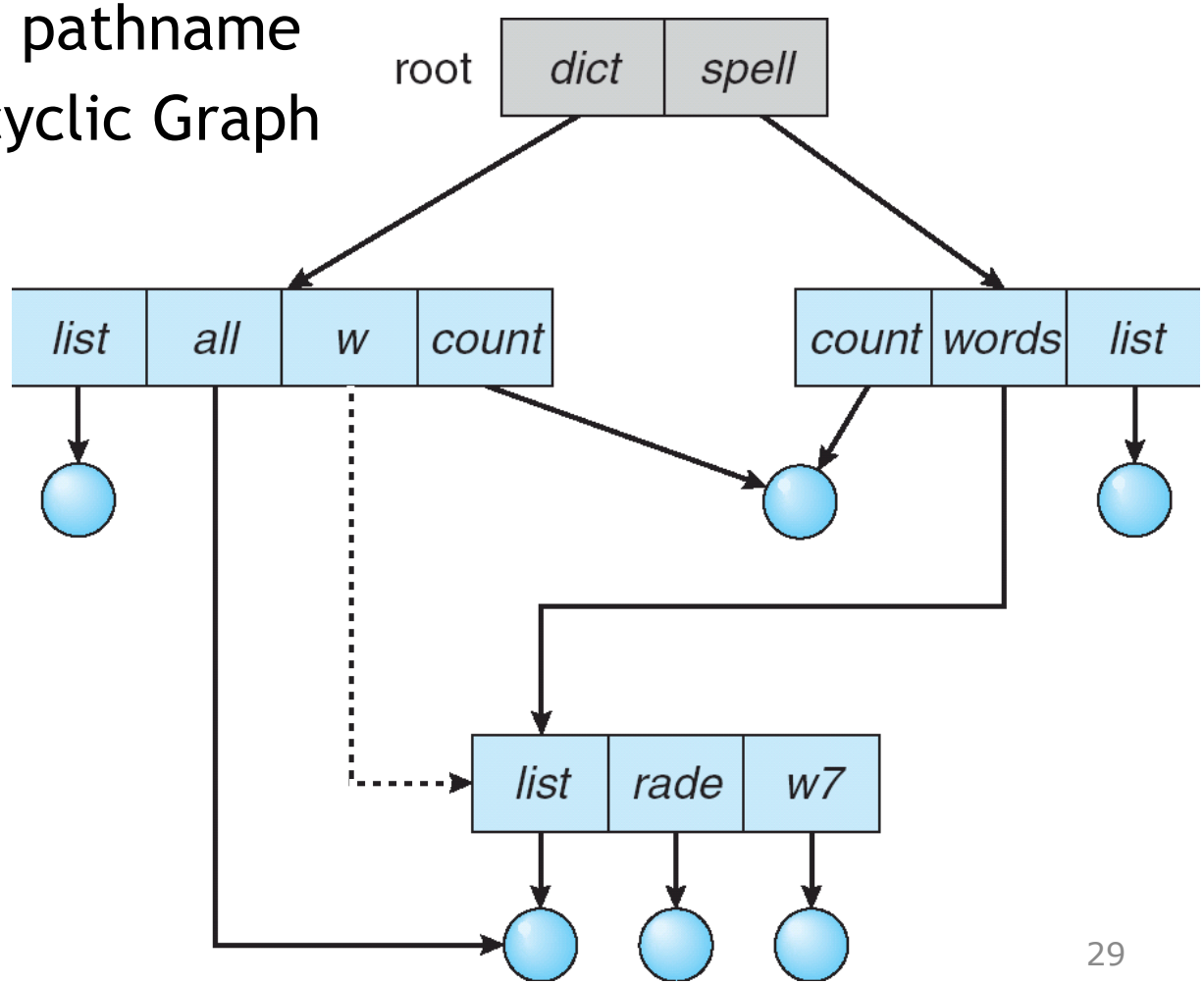
- Same object linked from multiple directories
- Create an alias to a pathname
- Directories Form Acyclic Graph



Symbolic link: alias to an existing name
ln –s in unix
/dict/w is a tiny file containing the absolute path of the indicated file: /spell/words/list

Files Can Have Multiple Names

- Same object linked from multiple directories
- Create an alias to a pathname
- Directories Form Acyclic Graph

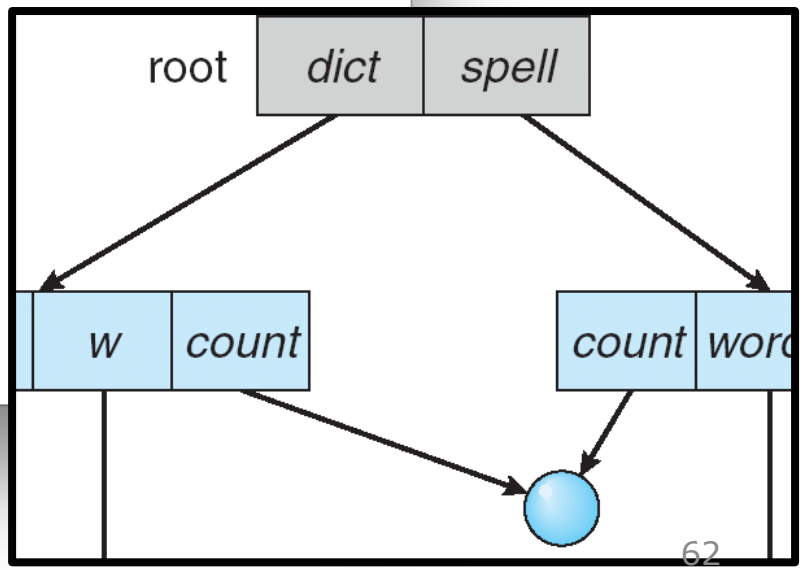


Symbolic vs. Hard Links Example

OS — -bash — 80×24

```
Last login: Wed May  3 13:18:40 on ttys001
Davids-MacBook-Pro:~ davidhay$ cd OS
Davids-MacBook-Pro:OS davidhay$ mkdir spell
Davids-MacBook-Pro:OS davidhay$ mkdir dict
Davids-MacBook-Pro:OS davidhay$ touch spell/count
Davids-MacBook-Pro:OS davidhay$ echo "aaa" > spell/count
Davids-MacBook-Pro:OS davidhay$ ln spell/count dict/count
Davids-MacBook-Pro:OS davidhay$ cat spell/count
aaa
Davids-MacBook-Pro:OS davidhay$ cat dict/count
aaa
Davids-MacBook-Pro:OS davidhay$
```

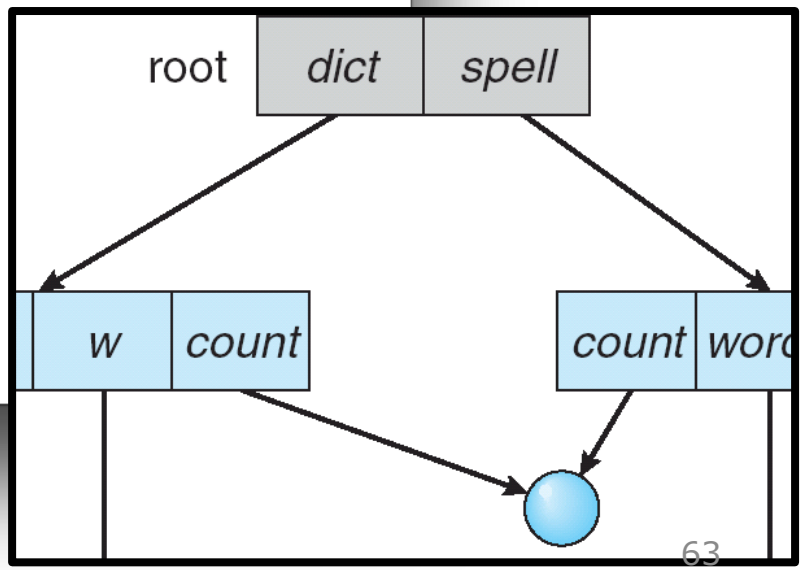
Hard link



Symbolic vs. Hard Links Example

OS — -bash — 80×24

```
Last login: Wed May  3 13:18:40 on ttys001
Davids-MacBook-Pro:~ davidhay$ cd OS
Davids-MacBook-Pro:OS davidhay$ mkdir spell
Davids-MacBook-Pro:OS davidhay$ mkdir dict
Davids-MacBook-Pro:OS davidhay$ touch spell/count
Davids-MacBook-Pro:OS davidhay$ echo "aaa" > spell/count
Davids-MacBook-Pro:OS davidhay$ ln spell/count dict/count
Davids-MacBook-Pro:OS davidhay$ cat spell/count
aaa
Davids-MacBook-Pro:OS davidhay$ cat dict/count
aaa
Davids-MacBook-Pro:OS davidhay$ mv spell/count spell/count1
Davids-MacBook-Pro:OS davidhay$ cat spell/count
cat: spell/count: No such file or directory
Davids-MacBook-Pro:OS davidhay$ cat spell/count1
aaa
Davids-MacBook-Pro:OS davidhay$ cat dict/count
aaa
Davids-MacBook-Pro:OS davidhay$
```

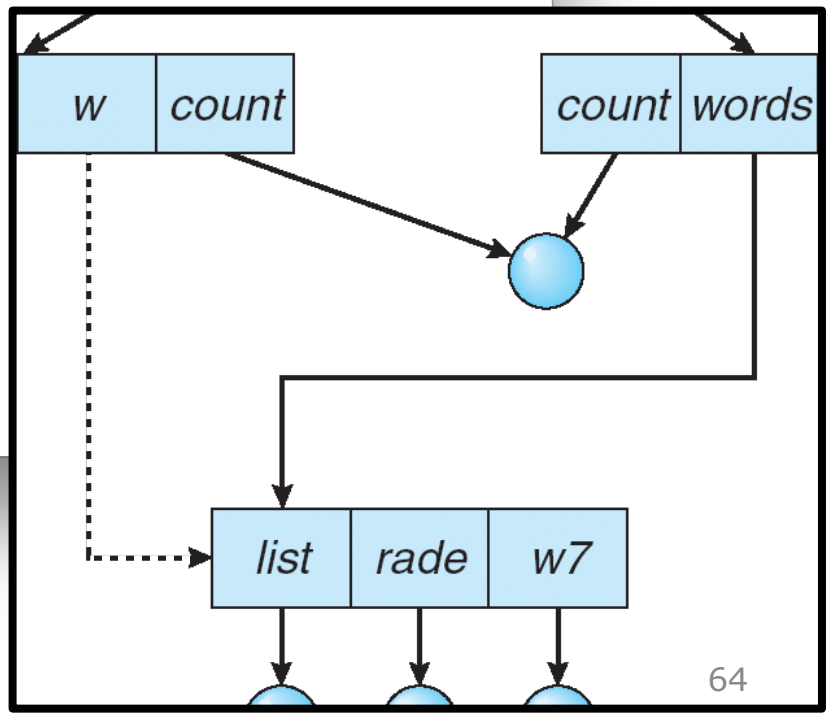


Symbolic vs. Hard Links Example

OS — -bash — 93×24

```
Davids-MacBook-Pro:spell davidhay$ rmdir words
Davids-MacBook-Pro:spell davidhay$ cd ..
Davids-MacBook-Pro:OS davidhay$ mkdir spell/words
Davids-MacBook-Pro:OS davidhay$ touch spell/words/list
Davids-MacBook-Pro:OS davidhay$ echo "bbb" > spell/words/list
Davids-MacBook-Pro:OS davidhay$ mkdir dict/w
Davids-MacBook-Pro:OS davidhay$ ln -s /Users/davidhay/OS/spell/words/list dict/w/list
Davids-MacBook-Pro:OS davidhay$ ls -l dict/w
total 8
lrwxr-xr-x  1 davidhay  staff  35 May  6 18:37 list -> /Users/davidhay/OS/spell/words/list
Davids-MacBook-Pro:OS davidhay$ cat spell/words/list
bbb
Davids-MacBook-Pro:OS davidhay$ cat dict/w/list
bbb
Davids-MacBook-Pro:OS davidhay$
```

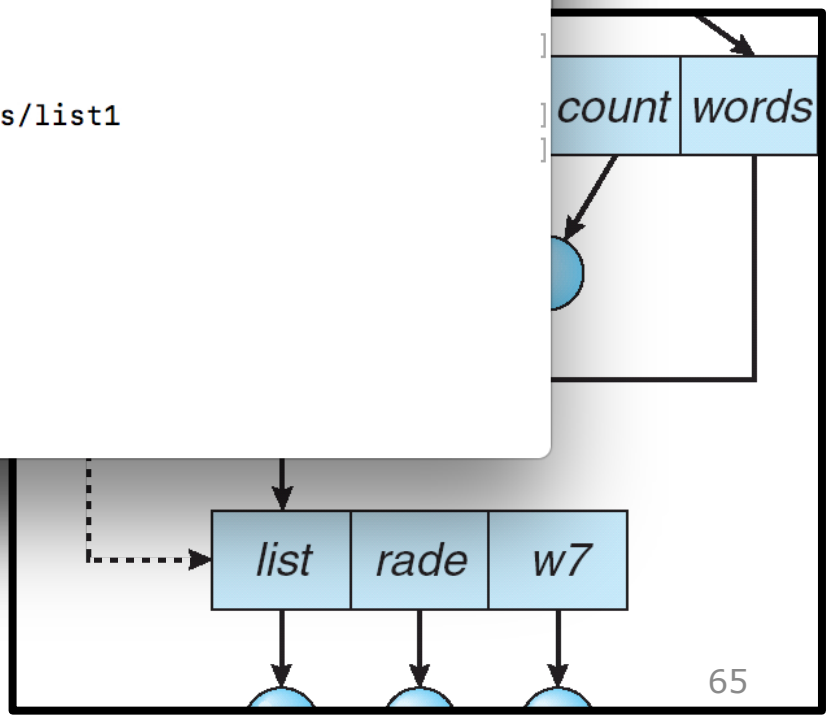
Symbolic link



Symbolic vs. Hard Links Example

OS — -bash — 93×24

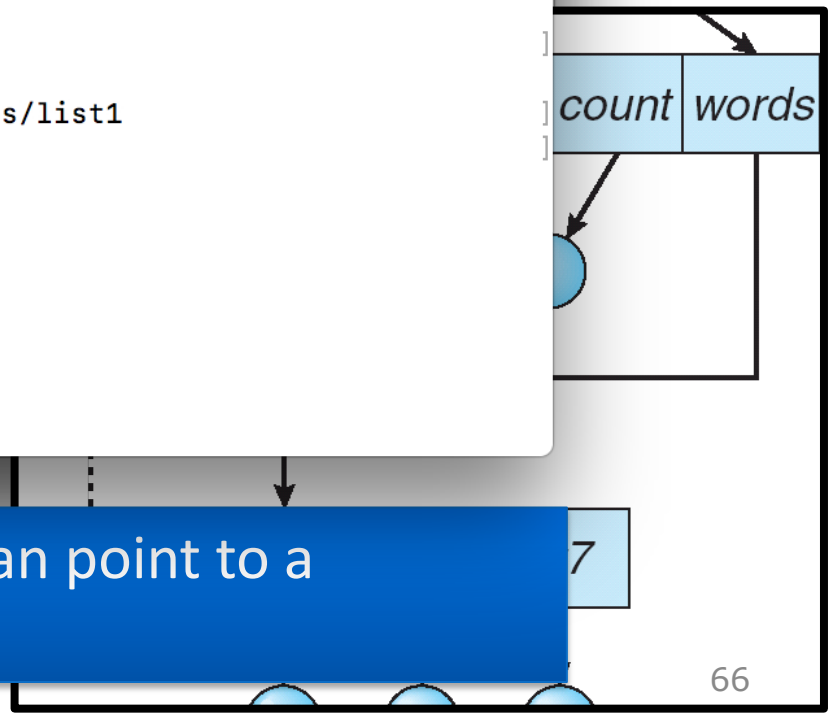
```
Davids-MacBook-Pro:spell davidhay$ rmdir words
Davids-MacBook-Pro:spell davidhay$ cd ..
Davids-MacBook-Pro:OS davidhay$ mkdir spell/words
Davids-MacBook-Pro:OS davidhay$ touch spell/words/list
Davids-MacBook-Pro:OS davidhay$ echo "bbb" > spell/words/list
Davids-MacBook-Pro:OS davidhay$ mkdir dict/w
Davids-MacBook-Pro:OS davidhay$ ln -s /Users/davidhay/OS/spell/words/list dict/w/list
Davids-MacBook-Pro:OS davidhay$ ls -l dict/w
total 8
lrwxr-xr-x  1 davidhay  staff  35 May  6 18:37 list -> /Users/davidhay/OS/spell/words/list
Davids-MacBook-Pro:OS davidhay$ cat spell/words/list
bbb
Davids-MacBook-Pro:OS davidhay$ cat dict/w/list
bbb
Davids-MacBook-Pro:OS davidhay$ mv spell/words/list spell/words/list1
Davids-MacBook-Pro:OS davidhay$ cat dict/w/list
cat: dict/w/list: No such file or directory
Davids-MacBook-Pro:OS davidhay$
```



Symbolic vs. Hard Links Example

OS — -bash — 93×24

```
Davids-MacBook-Pro:spell davidhay$ rmdir words
Davids-MacBook-Pro:spell davidhay$ cd ..
Davids-MacBook-Pro:OS davidhay$ mkdir spell/words
Davids-MacBook-Pro:OS davidhay$ touch spell/words/list
Davids-MacBook-Pro:OS davidhay$ echo "bbb" > spell/words/list
Davids-MacBook-Pro:OS davidhay$ mkdir dict/w
Davids-MacBook-Pro:OS davidhay$ ln -s /Users/davidhay/OS/spell/words/list dict/w/list
Davids-MacBook-Pro:OS davidhay$ ls -l dict/w
total 8
lrwxr-xr-x  1 davidhay  staff  35 May  6 18:37 list -> /Users/davidhay/OS/spell/words/list
Davids-MacBook-Pro:OS davidhay$ cat spell/words/list
bbb
Davids-MacBook-Pro:OS davidhay$ cat dict/w/list
bbb
Davids-MacBook-Pro:OS davidhay$ mv spell/words/list spell/words/list1
Davids-MacBook-Pro:OS davidhay$ cat dict/w/list
cat: dict/w/list: No such file or directory
Davids-MacBook-Pro:OS davidhay$
```



… but unlike hard links, symbolic links can point to a directory!

Link Semantics

- Support shared files and subdirectories
  - Why not copy the file?

Link Semantics

- Support shared files and subdirectories
  - Why not copy the file?
  - Linking creates a new directory entry
    - Link is a pointer to another file or subdirectory
    - Links are ignored when traversing FS
    - *ln* in UNIX, *fsutil* in Windows for hard links
    - *ln -s* in UNIX, shortcuts in Windows for soft links

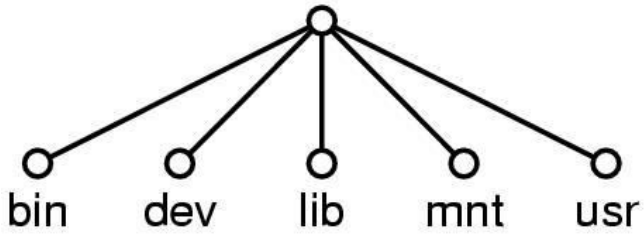
Link Semantics

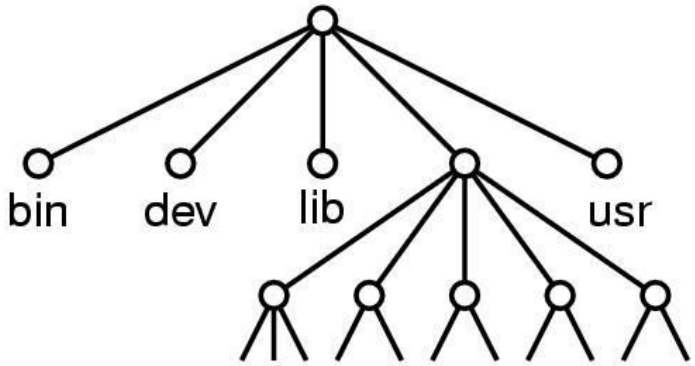
- Support shared files and subdirectories
  - Why not copy the file?
  - Linking creates a new directory entry
    - Link is a pointer to another file or subdirectory
    - Links are ignored when traversing FS
    - *ln* in UNIX, *fsutil* in Windows for hard links
    - *ln -s* in UNIX, shortcuts in Windows for soft links
- Issues?
  - Two different names (aliasing)
  - If *dict* deletes *count* $\Rightarrow$ dangling pointer
    - Keep backpointers of links for each file
    - Leave the link, and delete only when accessed later
    - Keep reference count of each file

File System Mounting

- Making a file system ready for use by the OS
- Mount allows two FSes to be merged into one
  - For example you insert your USB into the root FS

```
mount("/dev/usb0", "/mnt", 0)
```





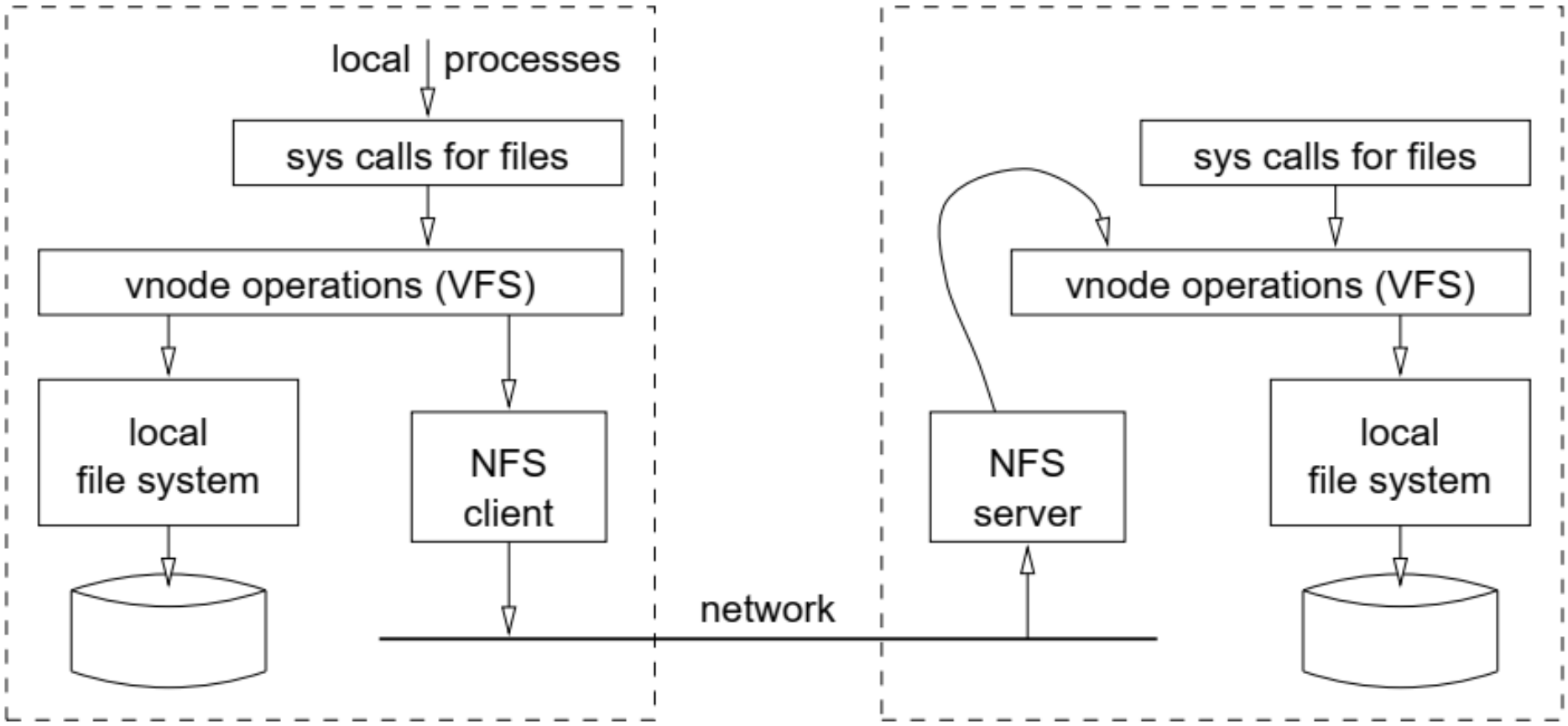
- When a mount-point is reached in parsing a path: start over from the root of the mounted file system

  (similar to symbolic link)

Remote file system mounting

- Same idea, but file system is actually on some other machine
- Implementation uses remote procedure call
  - Package up the user's file system operation
  - Send it to the remote machine where it gets executed like a local request
  - Send back the answer
- Very common in modern systems

Network File System (NFS)



File Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Create
  - Delete
  - List

File Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Create
  - Delete
  - List

– NOTE –
Creating, deleting, and listing files are operations on the directory

Categories of Users

- Individual user
  - Log-in establishes a user ID
  - Might be just local on the computer or could be through interaction with a network service
- Groups to which the user belongs
  - For example, “dhay” is in “faculty”

Expressing Access Rights

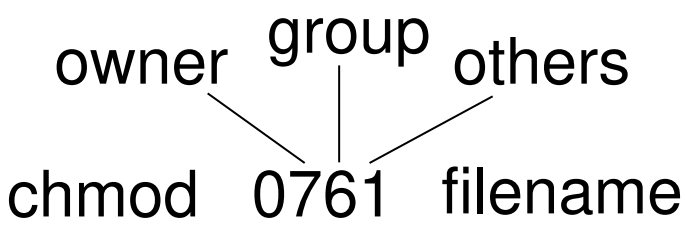
- Minimal:
  - Just a single owner, a single group, and the rest
  - Pro: Compact enough to fit in just a few bytes
  - Con: Not very expressive
  - Used in Unix / Linux
- Detailed:
  - A per-file list that tells exactly who can do what to that file
  - Pro: Highly expressive
  - Con: Harder to represent in a compact way
  - Used in Windows

Linux Access Rights

- Mode of access: read, write, execute
- Three classes of users

	R W X		
a) **owner access**	1 1 1	⇒	7
b) **group access**	1 1 0	⇒	6
c) **others access**	0 0 1	⇒	1
			octal

- For each file or directory define the desired access rights as 9 bits

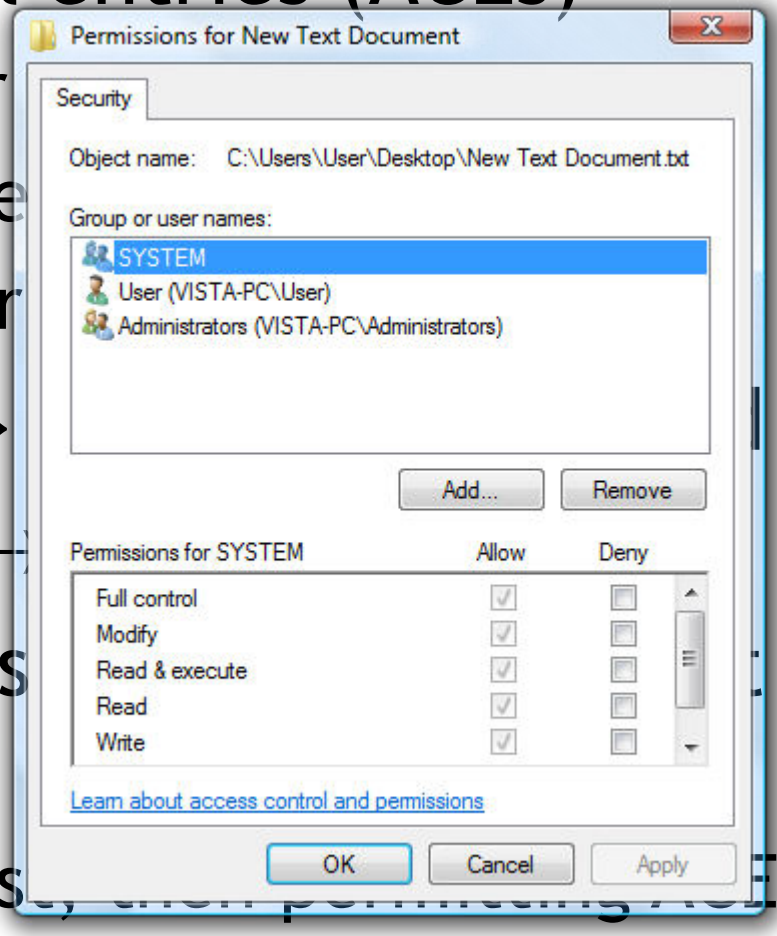


Access Control List

- A list of access control entries (ACEs)
  - Identification of a user or group
  - Identification of an operation on the object
  - Indication is allowed or denied
- Object without ACL → everything allowed
- Object with null ACL → nothing allowed
- Object with ACEs → use the first one that applies to the user
  - Put forbidding ACEs first, then permitting ACEs
  - If nothing applies → deny access

Access Control List

- A list of access control entries (ACEs)
  - Identification of a user
  - Identification of an ope
  - Indication is allowed or
- Object without ACL →
- Object with null ACL →
- Object with ACEs → us
  applies to the user
  - Put forbidding ACEs firs
  - If nothing applies → deny access



LAYOUT and ACCESS

Open & Close

- Why do we need to open files?

Open & Close

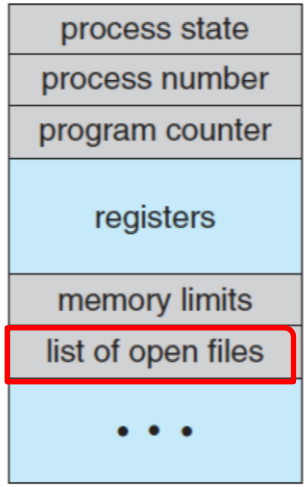
- Why do we need to open files?
- Resolving a name is a costly process
  - May need to traverse a long path
  - 2 disk accesses per path element (Unix)
- Open caches the file data in the open files table
  - Data access uses this data and does not need to parse the whole path again

Open Files of a Process

Open Files of a Process

Process Control Block (PCB)

How the process is represented in the OS



- Context
- Additional OS information needed:
  - Memory management
  - Accounting information
  - I/O status information

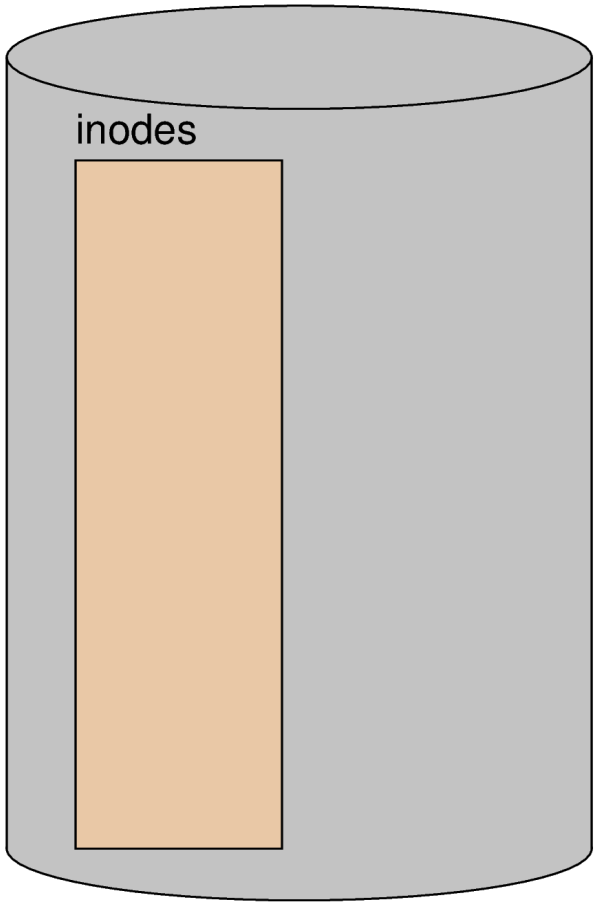
23

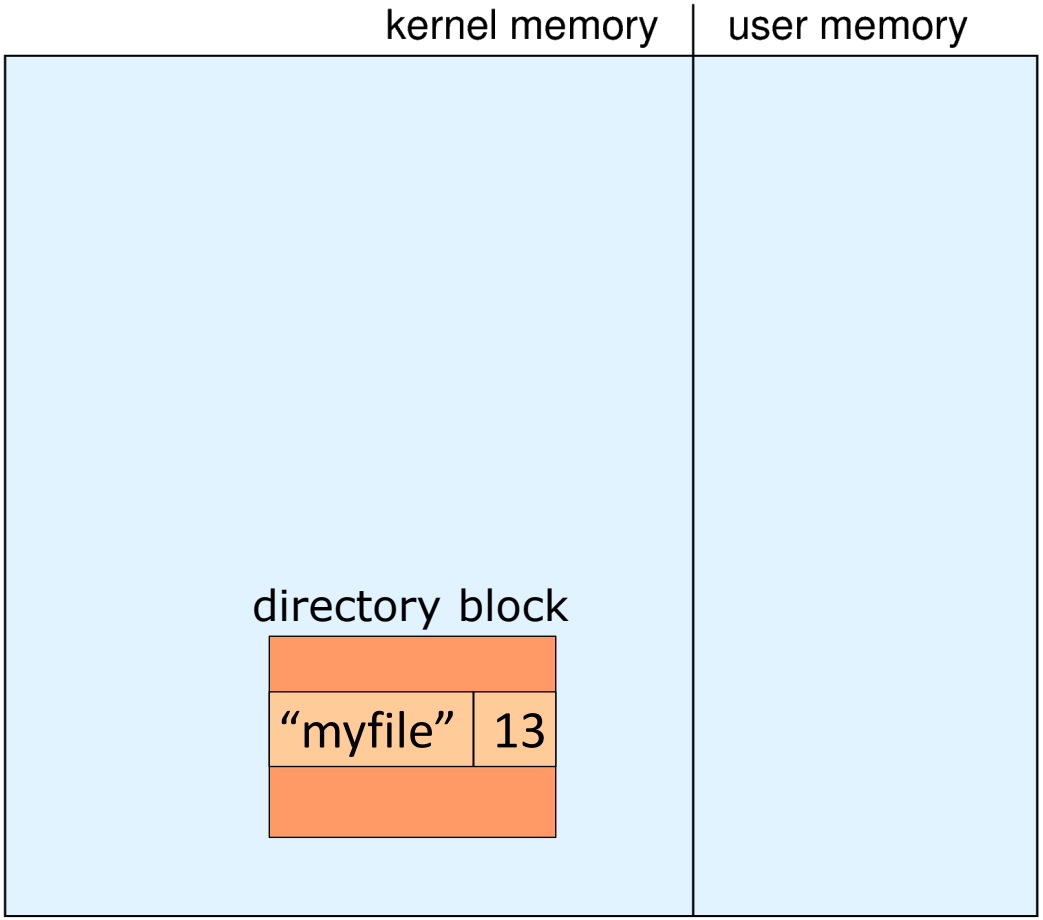
Lecture 3

Opening a File

fd = open("myfile");

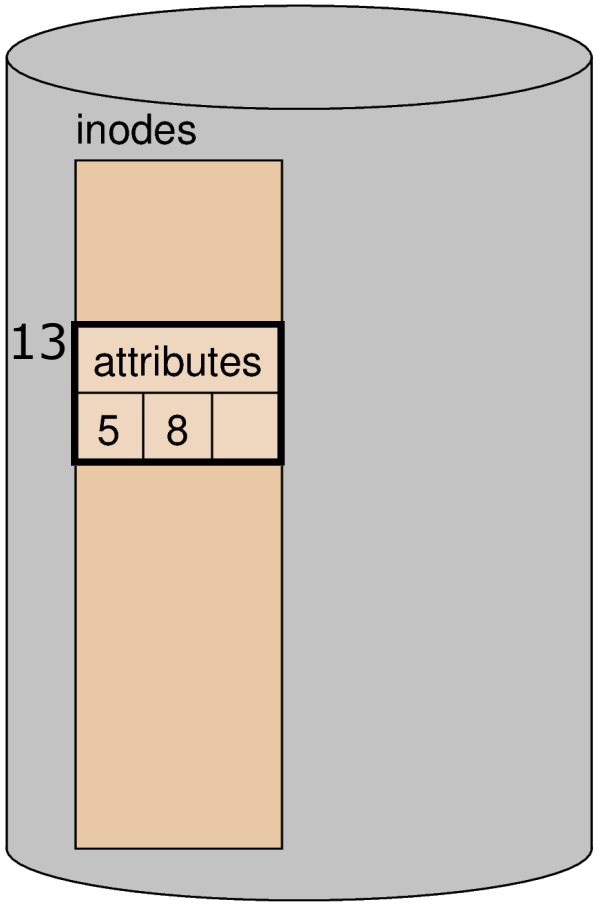
// "myfile" is inode 13

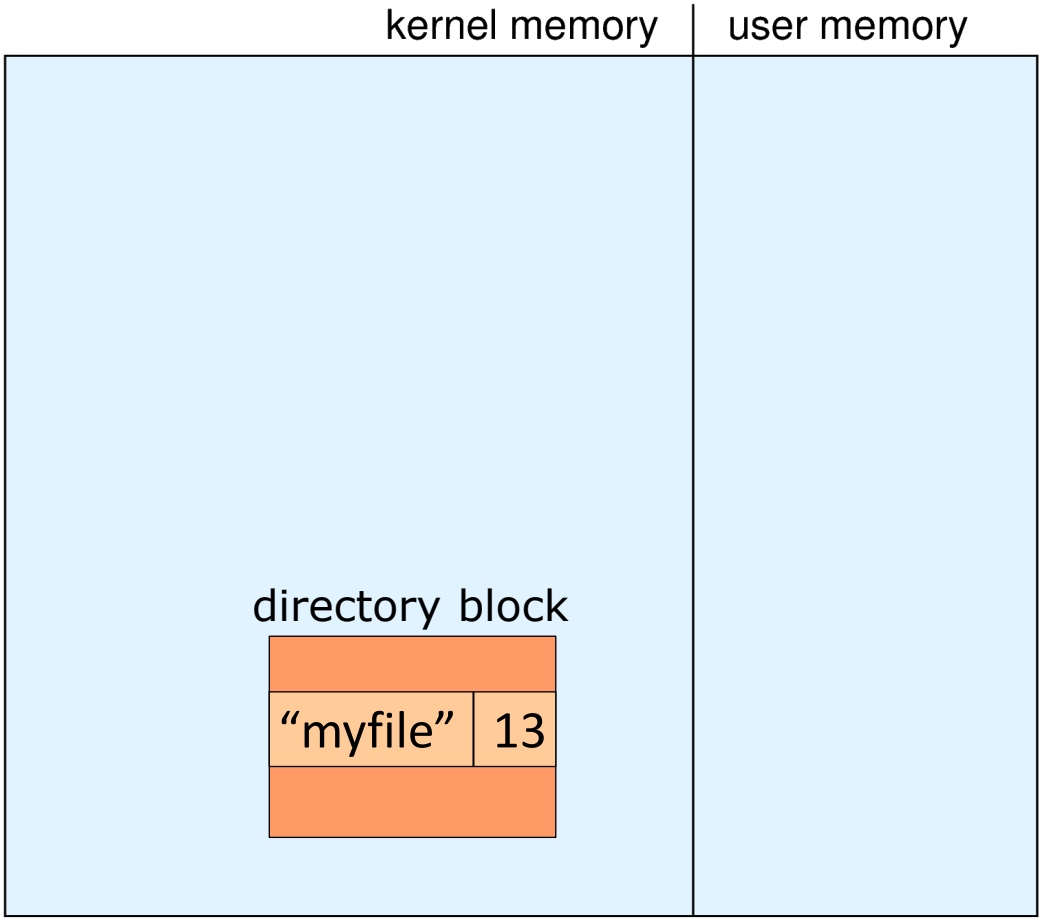




Opening a File

fd = open("myfile"); // "myfile" is inode 13

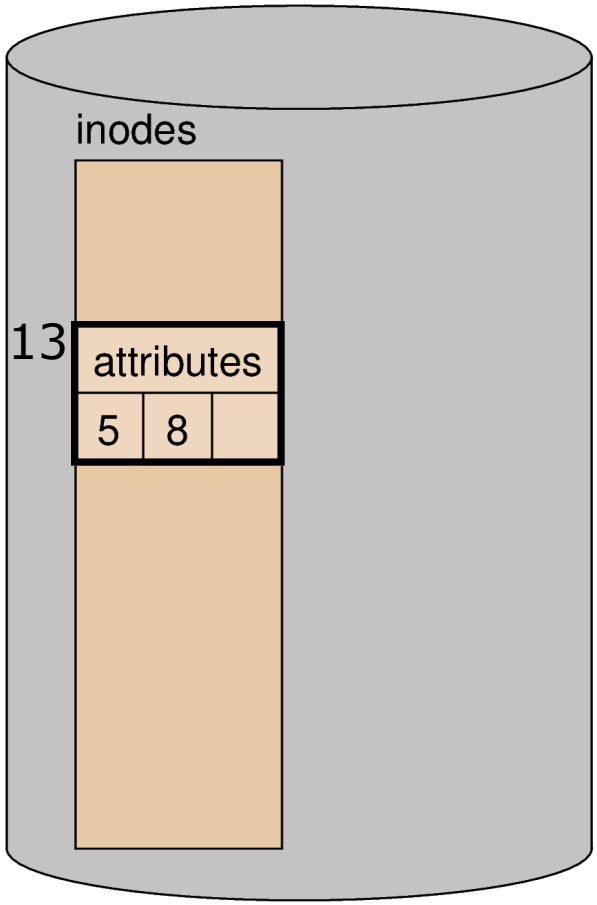


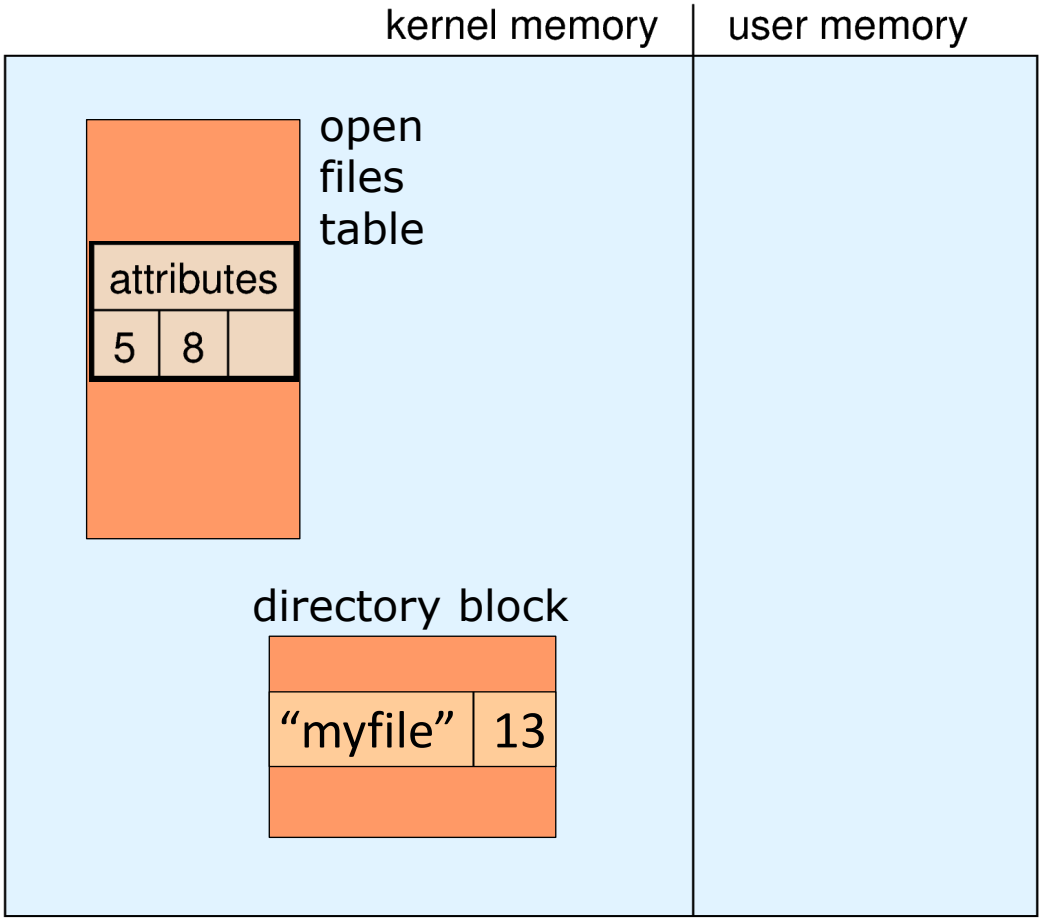


Opening a File

fd = open("myfile");

// "myfile" is inode 13

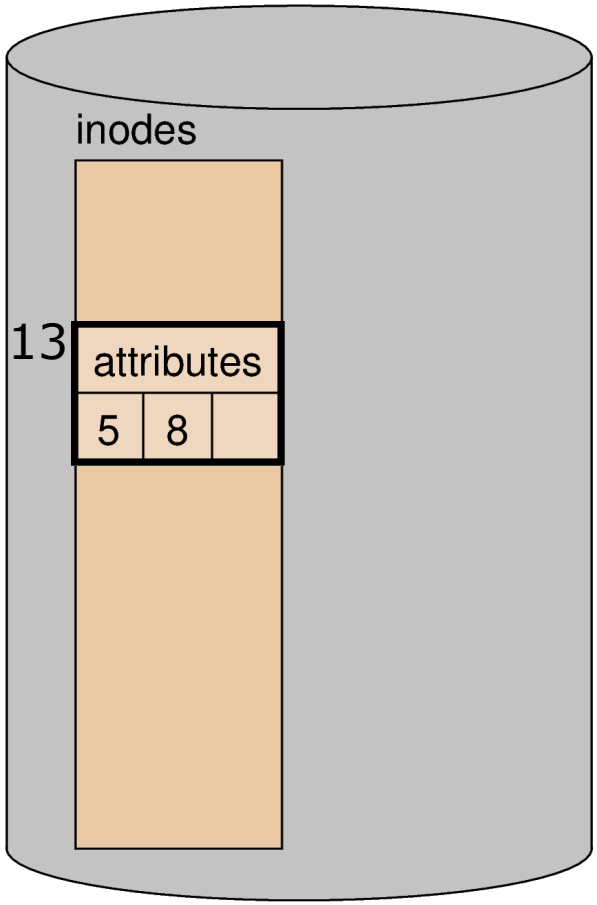


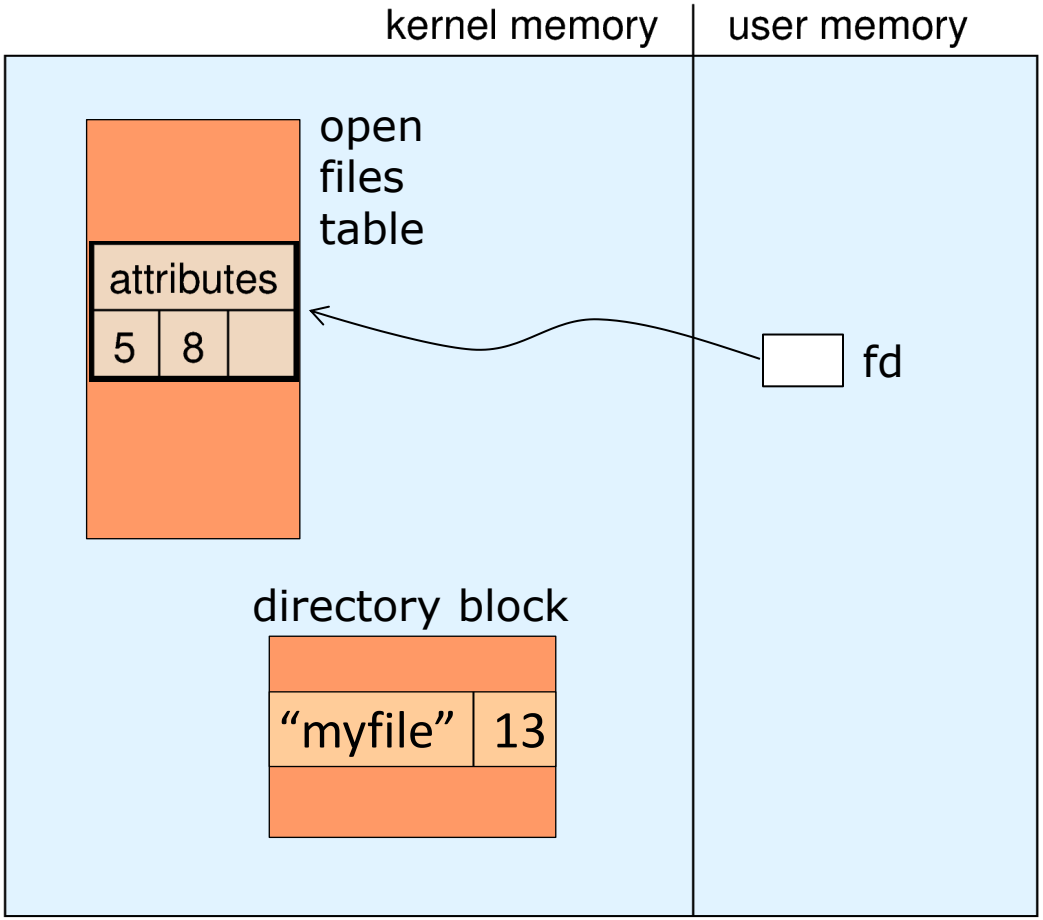


Opening a File

fd = open("myfile");

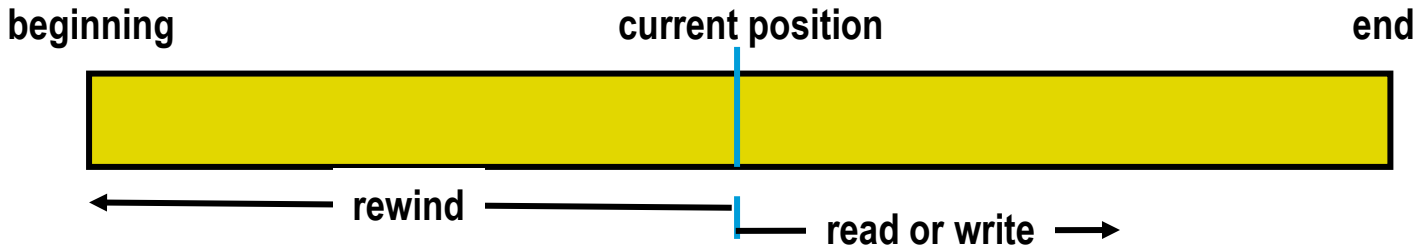
// "myfile" is inode 13





Access Semantics

- Maintain current location in open file
- Access is relative to this location
- Can change location using rewind or seek



File System Implementation

- What is the right structure in which to maintain data location information?
- How do we lay out the files on the physical disk?
- We need to support sequential and random access

File System Implementation

- What is the right structure in which to maintain data location information?
- How do we lay out the files on the physical disk?
- We need to support sequential and random access

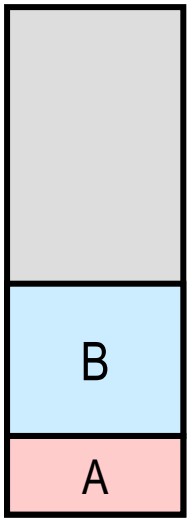
Many of the concerns in the implementation of a file system are similar to those of memory management + the way disks are organized

Contiguous Allocation

- May lead to FRAGMENTATION (as in memory)
  - Situation in which we have enough storage to allocate to a file, but not contiguously, as the holes are scattered all over storage space.

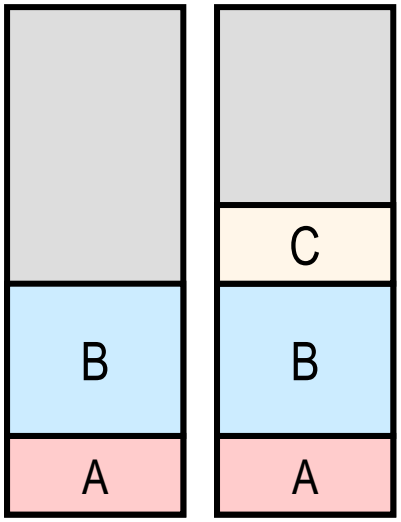
Contiguous Allocation

- May lead to FRAGMENTATION (as in memory)
  - Situation in which we have enough storage to allocate to a file, but not contiguously, as the holes are scattered all over storage space.



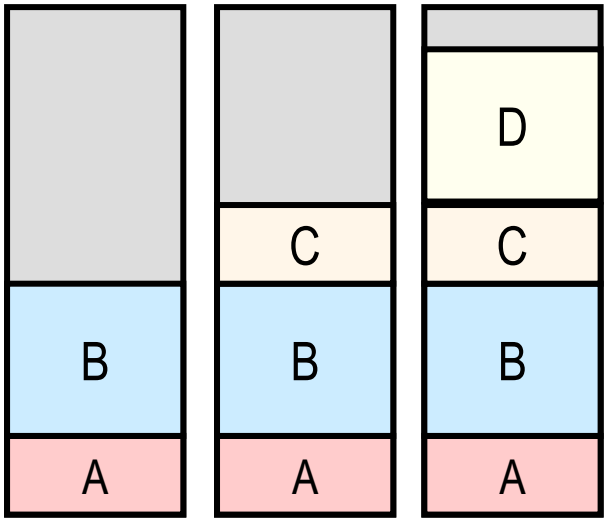
Contiguous Allocation

- May lead to FRAGMENTATION (as in memory)
  - Situation in which we have enough storage to allocate to a file, but not contiguously, as the holes are scattered all over storage space.



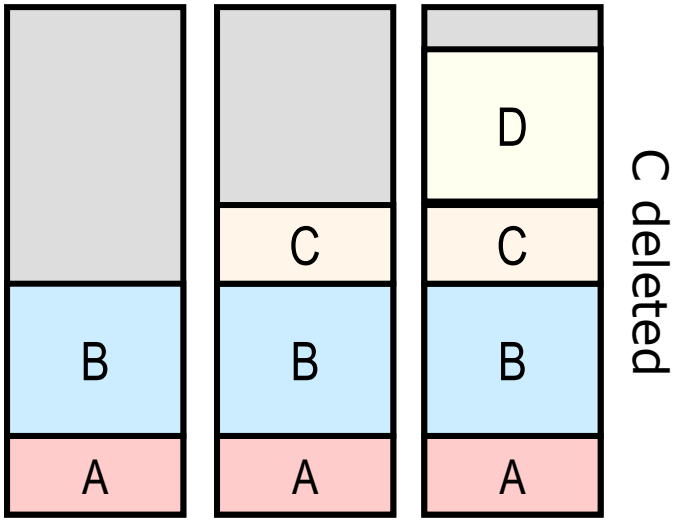
Contiguous Allocation

- May lead to FRAGMENTATION (as in memory)
  - Situation in which we have enough storage to allocate to a file, but not contiguously, as the holes are scattered all over storage space.



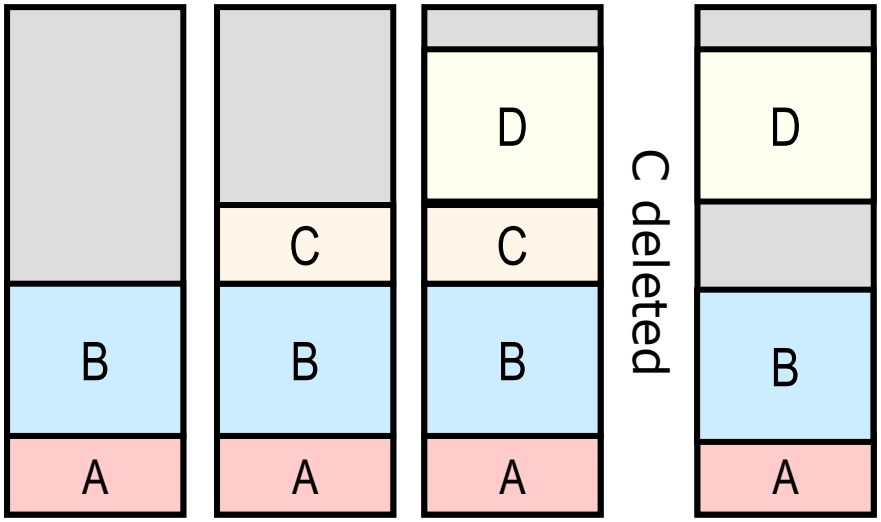
Contiguous Allocation

- May lead to FRAGMENTATION (as in memory)
  - Situation in which we have enough storage to allocate to a file, but not contiguously, as the holes are scattered all over storage space.



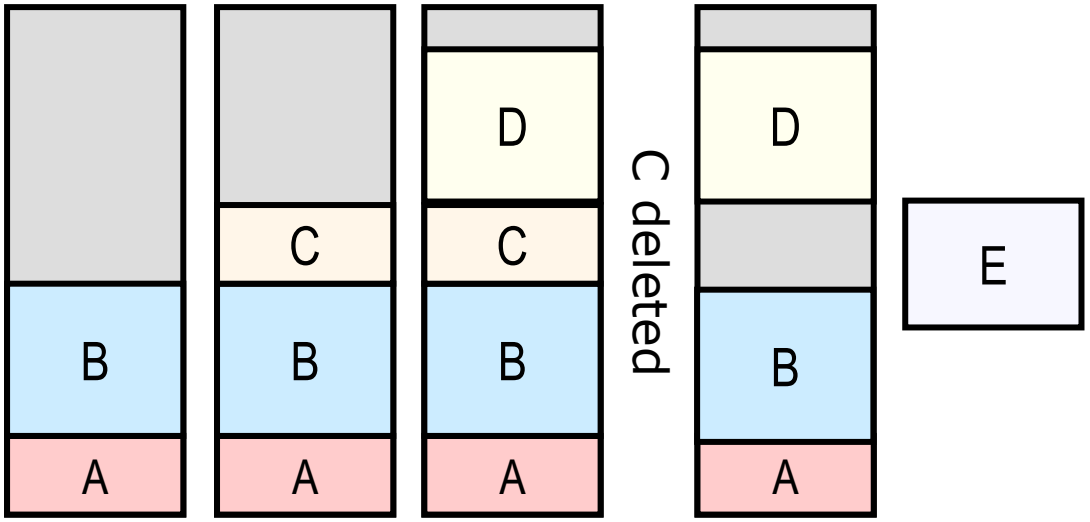
Contiguous Allocation

- May lead to FRAGMENTATION (as in memory)
  - Situation in which we have enough storage to allocate to a file, but not contiguously, as the holes are scattered all over storage space.



Contiguous Allocation

- May lead to FRAGMENTATION (as in memory)
  - Situation in which we have enough storage to allocate to a file, but not contiguously, as the holes are scattered all over storage space.



Contiguous Allocation

- May lead to FRAGMENTATION (as in memory)
  - Situation in which we have enough storage to allocate to a file, but not contiguously, as the holes are scattered all over storage space.
- Fragmentation Sometimes doesn't exist (CD, DVD media where files are not deleted)

Possible Solutions

Possible Solutions

- **Compaction:** Move around data in disk, to unify holes and create large enough holes to accommodate future files
  - May also improve performance (serial access)
  - **Problem:** Very costly operation. Sometime not enough space to move data

Possible Solutions

- **Compaction:** Move around data in disk, to unify holes and create large enough holes to accommodate future files
  - May also improve performance (serial access)
  - **Problem:** Very costly operation. Sometime not enough space to move data
- Divide files into fixed-size **blocks** (usually 4KB), all operations are made on blocks
  - Similar to pages in memory management

Semantic Gaps with Blocks

All operations are made on blocks

- User view: contiguous data records
- API view: sequence of bytes (arbitrary size)
- File system view: collection of blocks (4KB)
- Disk view: individual sectors (512 bytes)

Semantic Gaps with Blocks

All operations are ma locks

- User view: contiguo s
- API view: sequence of bytes (bitrary size)
- File system view: collection of blocks (4KB)
- Disk view: individual sectors (512 bytes)

Logical transfer unit

Physical transfer unit

Semantic Gaps with Blocks

All operations are made on blocks

- User view: contiguous data records
- API view: sequence of bytes (arbitrary size)
- File system view: collection of blocks (4KB)
- Disk view: individual sectors (512 bytes)

Accesses are not necessarily aligned with blocks; blocks may span multiple sectors

- getc(), putc() buffer 4 KB even if interface is one byte at a time

How are Blocks/Files Organized?

Need to decide:

How are Blocks/Files Organized?

Need to decide:

- Blocks layout on the storage device

How are Blocks/Files Organized?

Need to decide:

- Blocks layout on the storage device
- Data structure to support it

How are Blocks/Files Organized?

Need to decide:

- Blocks layout on the storage device
- Data structure to support it
- Free space management

How are Blocks/Files Organized?

Need to decide:

- Blocks layout on the storage device
- Data structure to support it
- Free space management
- Where meta-data is stored and how

How are Blocks/Files Organized?

Need to decide:

- Blocks layout on the storage device
- Data structure to support it
- Free space management
- Where meta-data is stored and how

Depends on many factors, including:

- Distribution of file sizes
- How file are accessed (serial vs. random?)
- Where files are stored (CD, disk, SSD…?)

How are Blocks/Files Organized?

Need to decide:

- Blocks layout on the storage device
- Data structure to support it
- Free space management
- Where meta-data is stored and how

Depends on many factors, including:

- Distribution of file sizes
- How file are accessed (serial vs. random?)
- Where files are stored (CD, disk, SSD…?)
- Implementation issues (access to legacy systems)

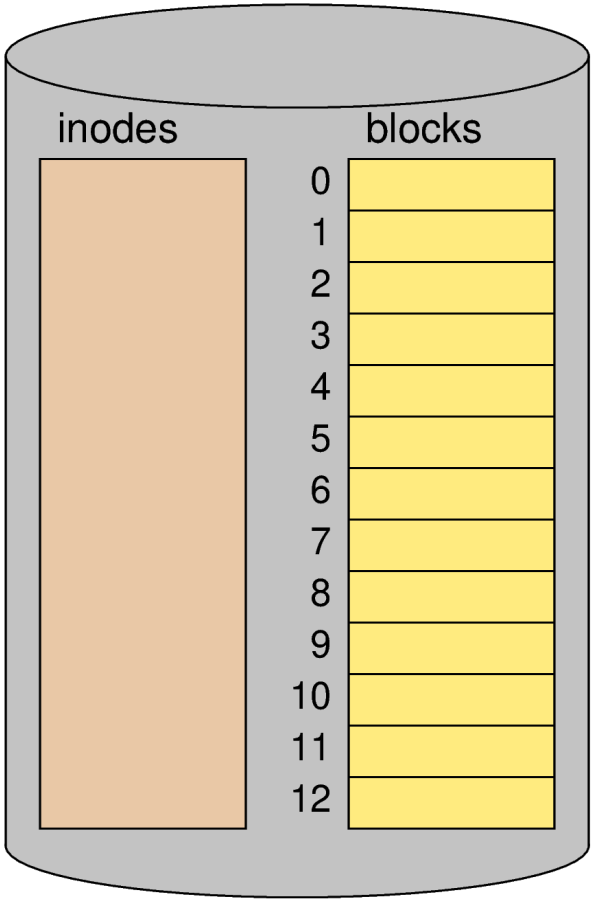
Reading Data

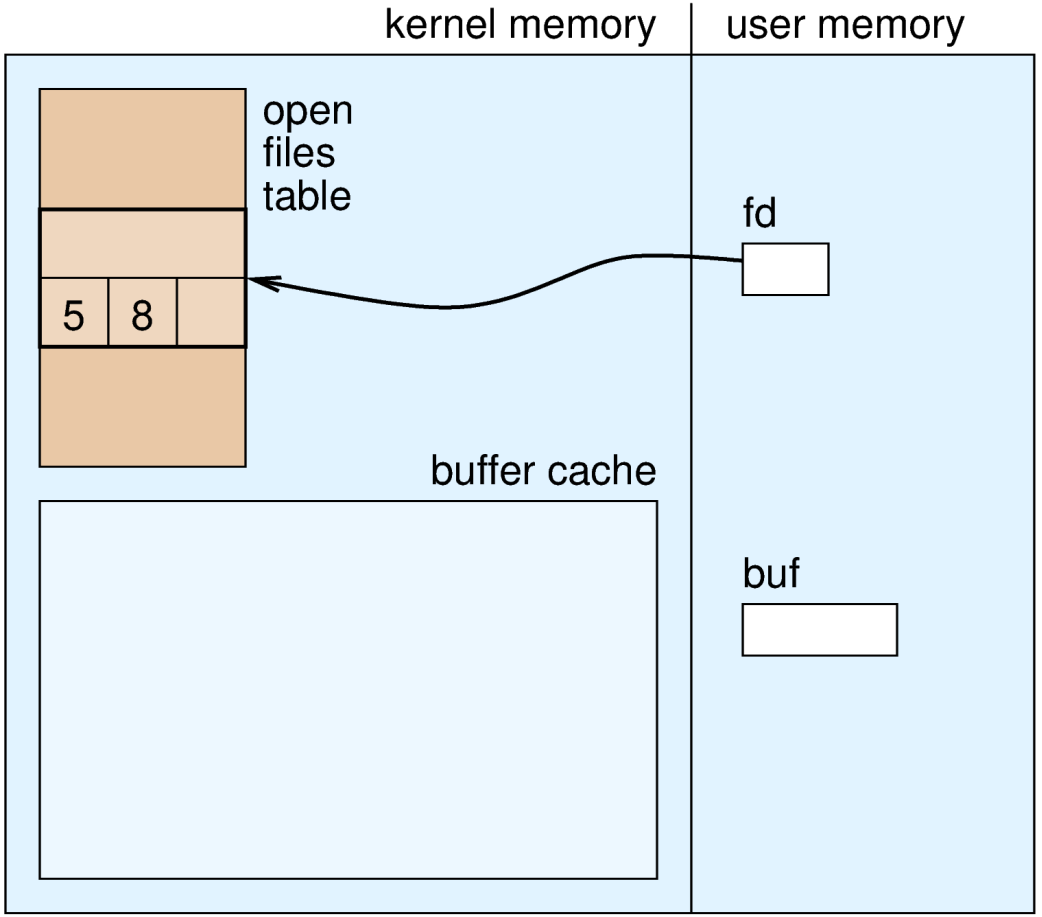
- API: read *N* bytes
  - Starting from current location
- Implementation:
  1. Identify blocks containing the data
  2. Read the blocks
  3. Copy relevant data to user buffer
     - May be partial blocks

Reading Data

read(fd, buf, 3000);

// read 3000 bytes from file to buf

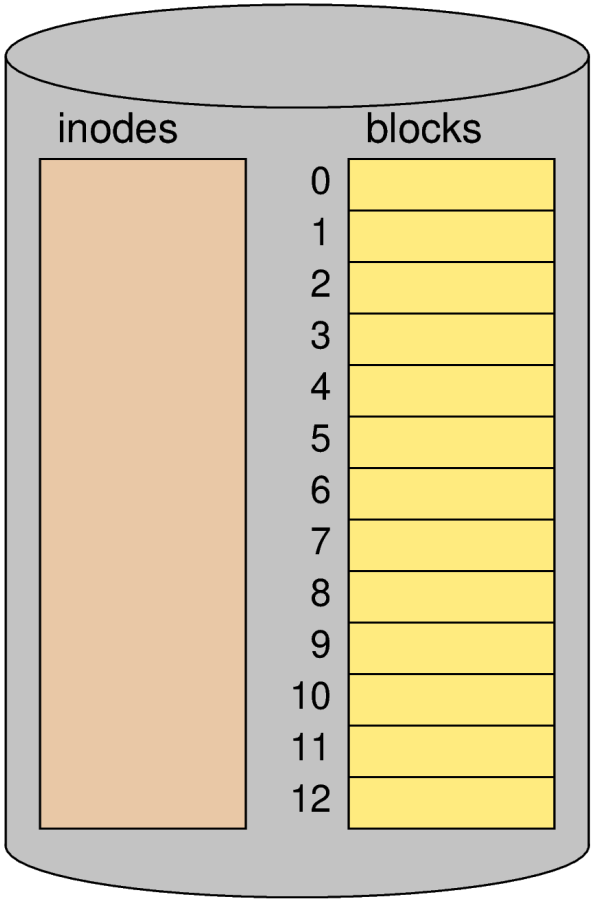


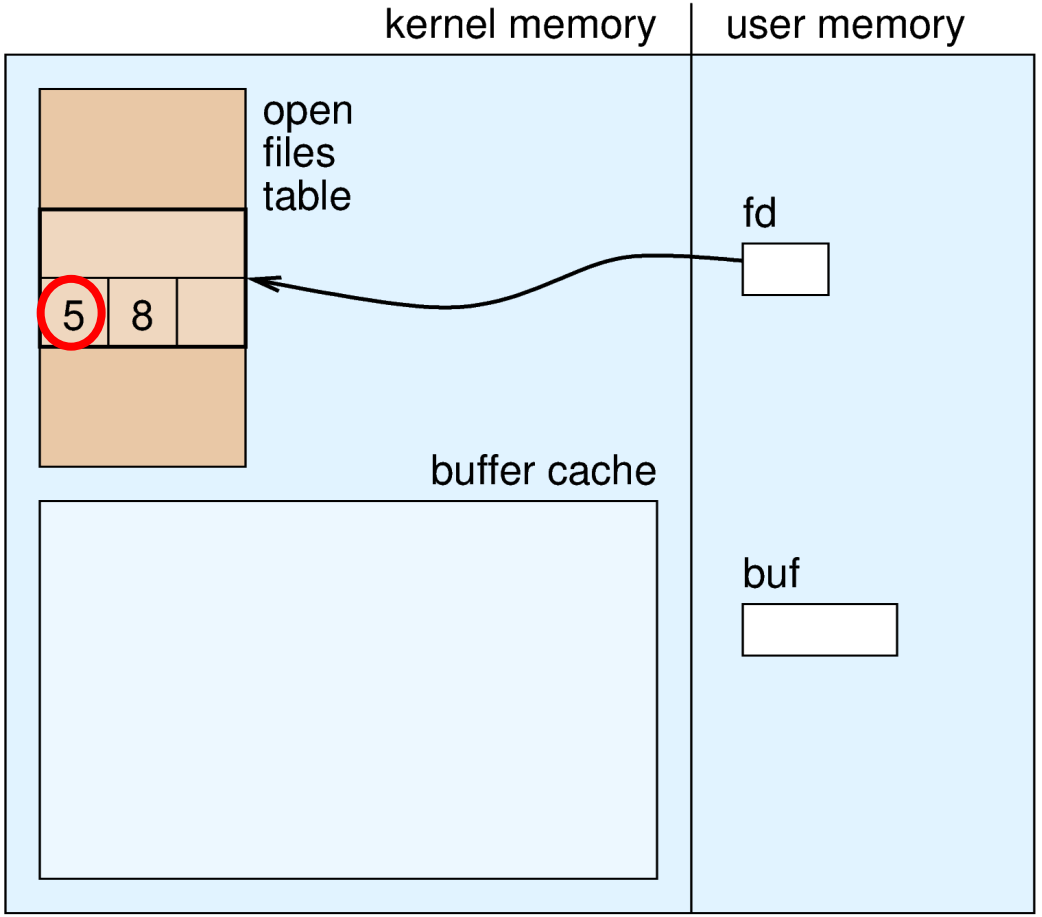


Reading Data

read(fd, buf, 3000);

// read 3000 bytes from file to buf

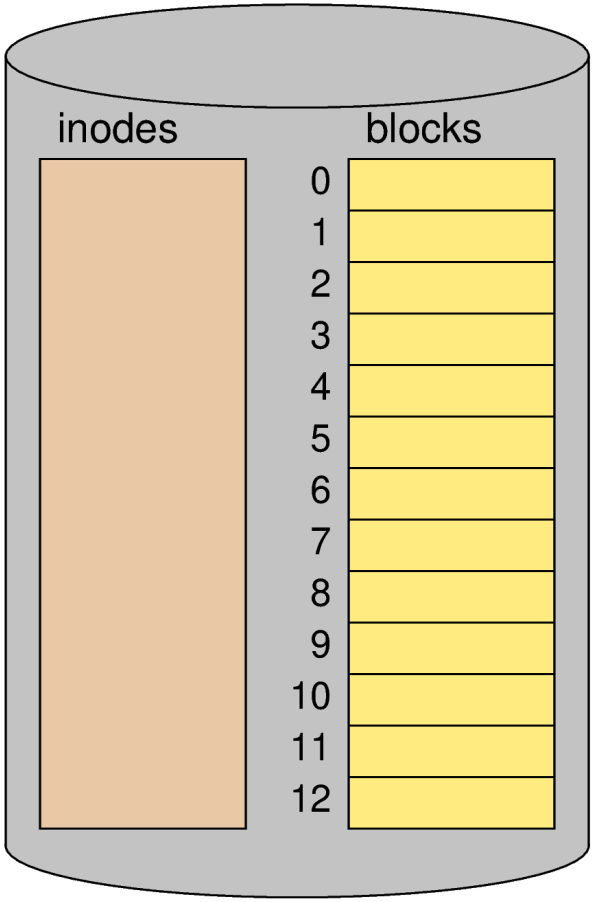


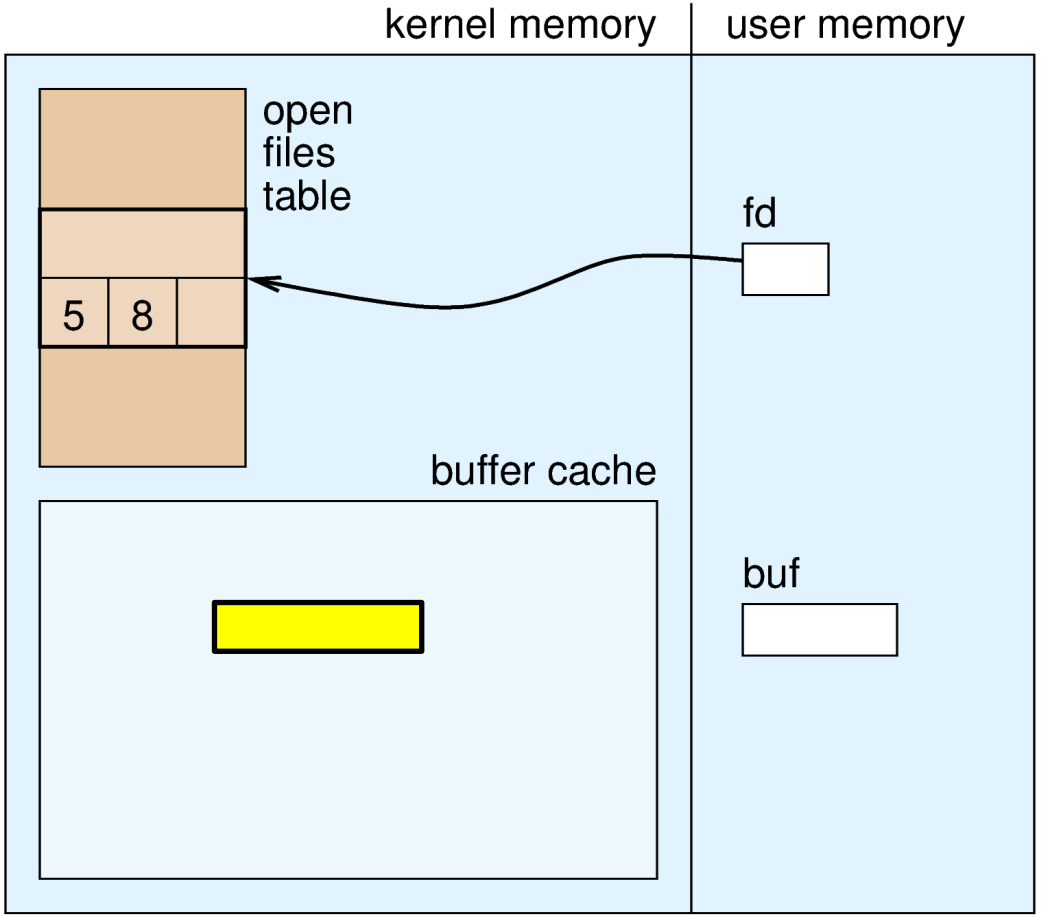


Reading Data

read(fd, buf, 3000);

// read 3000 bytes from file to buf

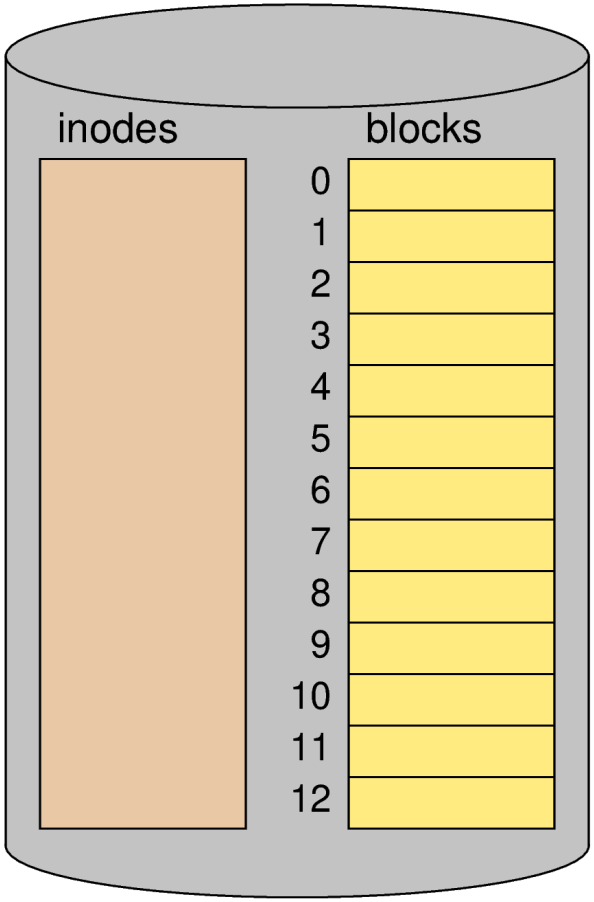


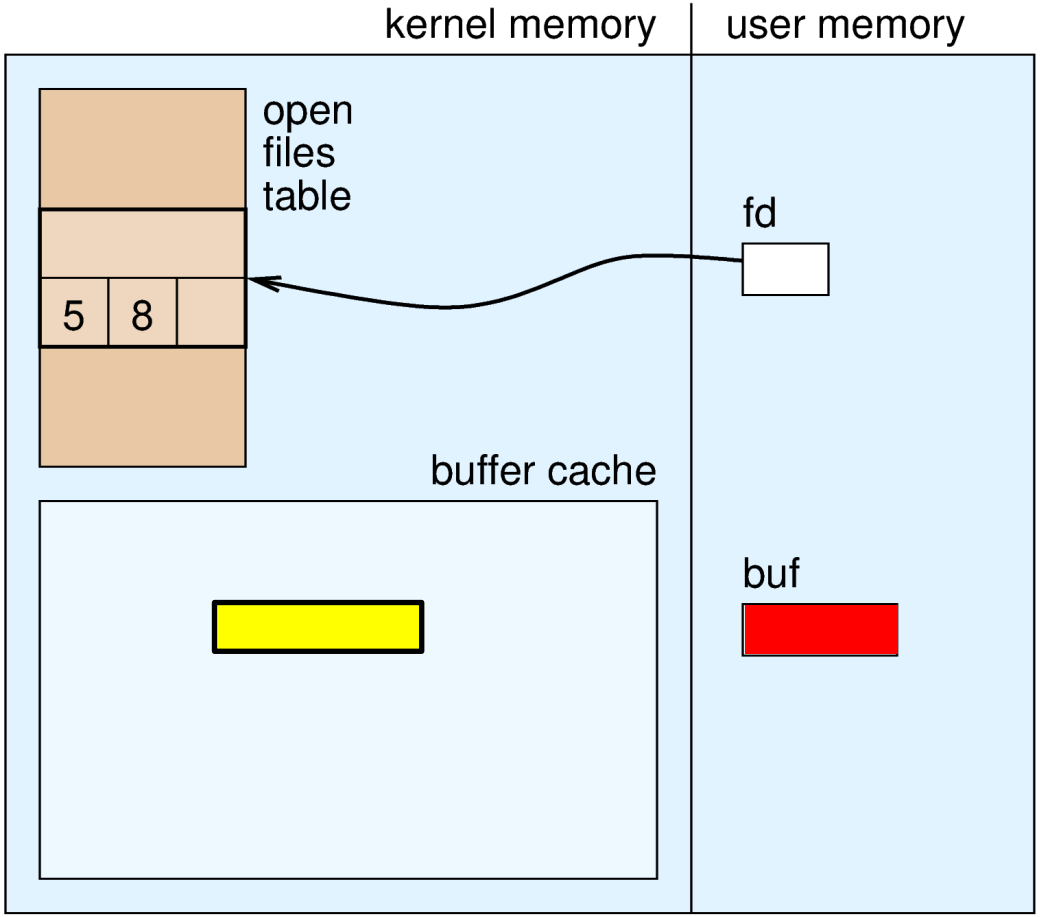


Reading Data

read(fd, buf, 3000);

// read 3000 bytes from file to buf



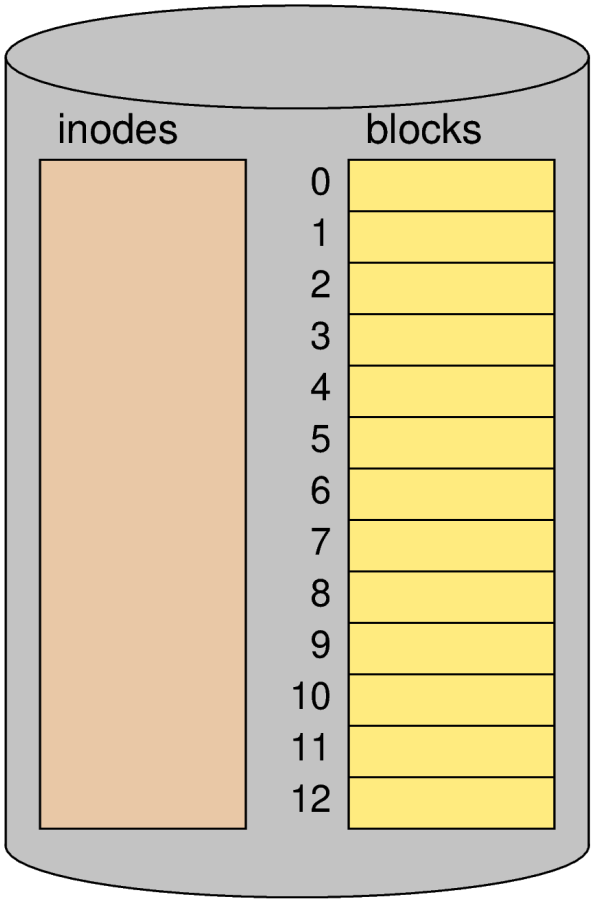


Writing Data

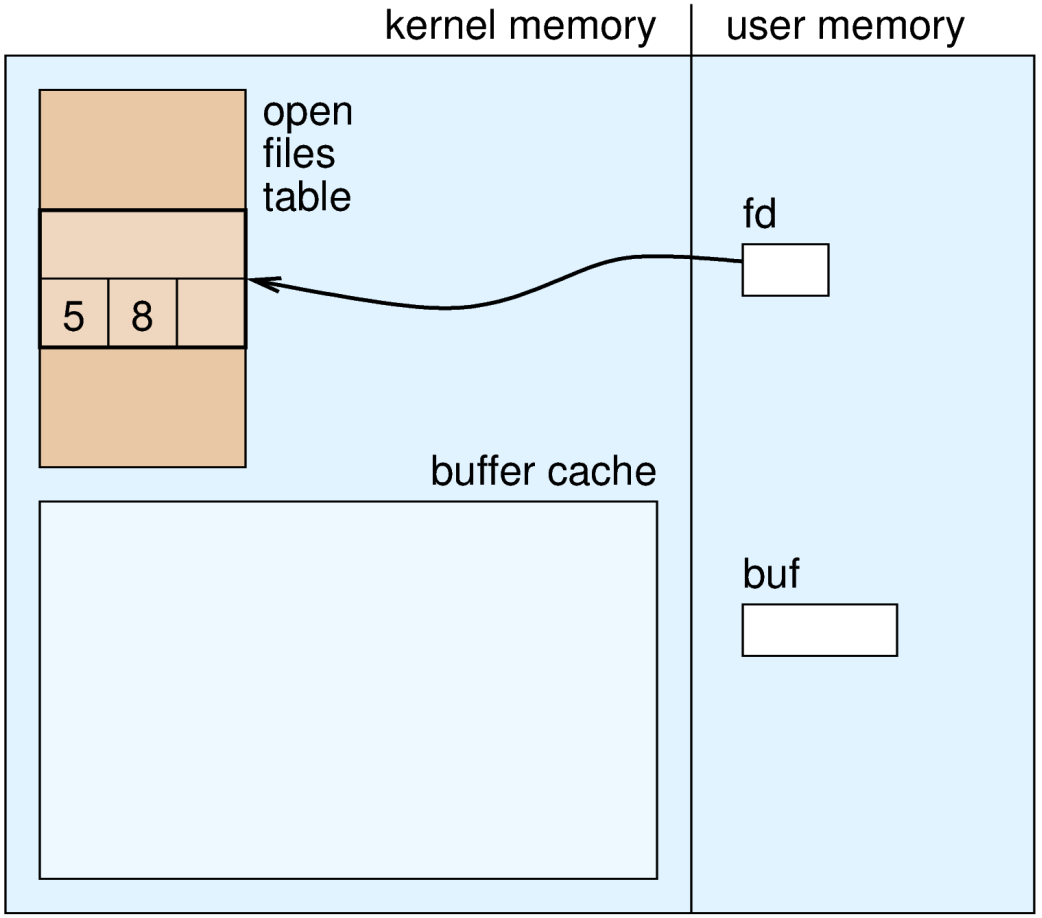
- API: write *N* bytes
  - Starting from current location
- Implementation:
  1. Identify blocks containing the data
  2. If not full blocks, read the blocks
  3. Copy data from user buffer
  4. Write blocks to disk
  5. If new blocks were added, update inode

Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```

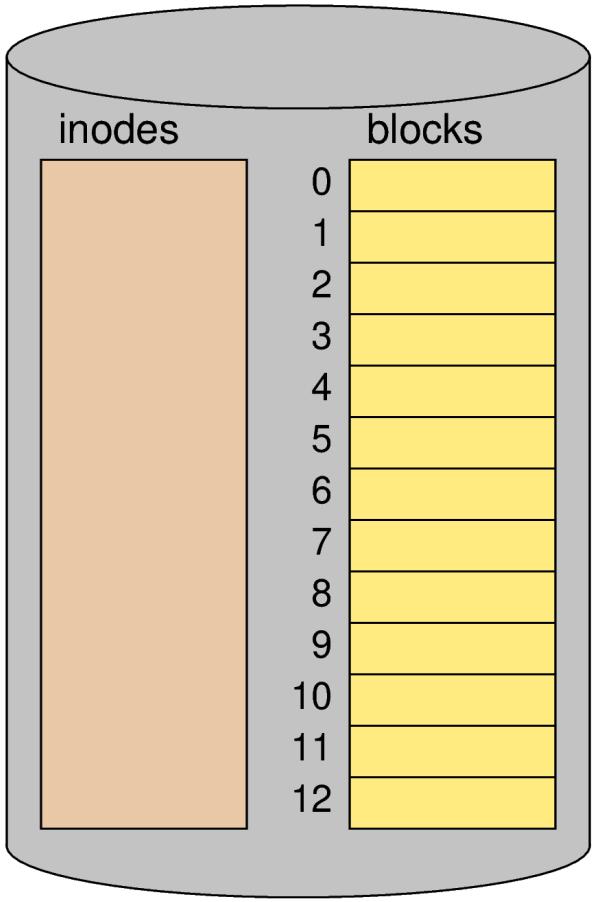



```
// write 3000 bytes at offset 7000
```

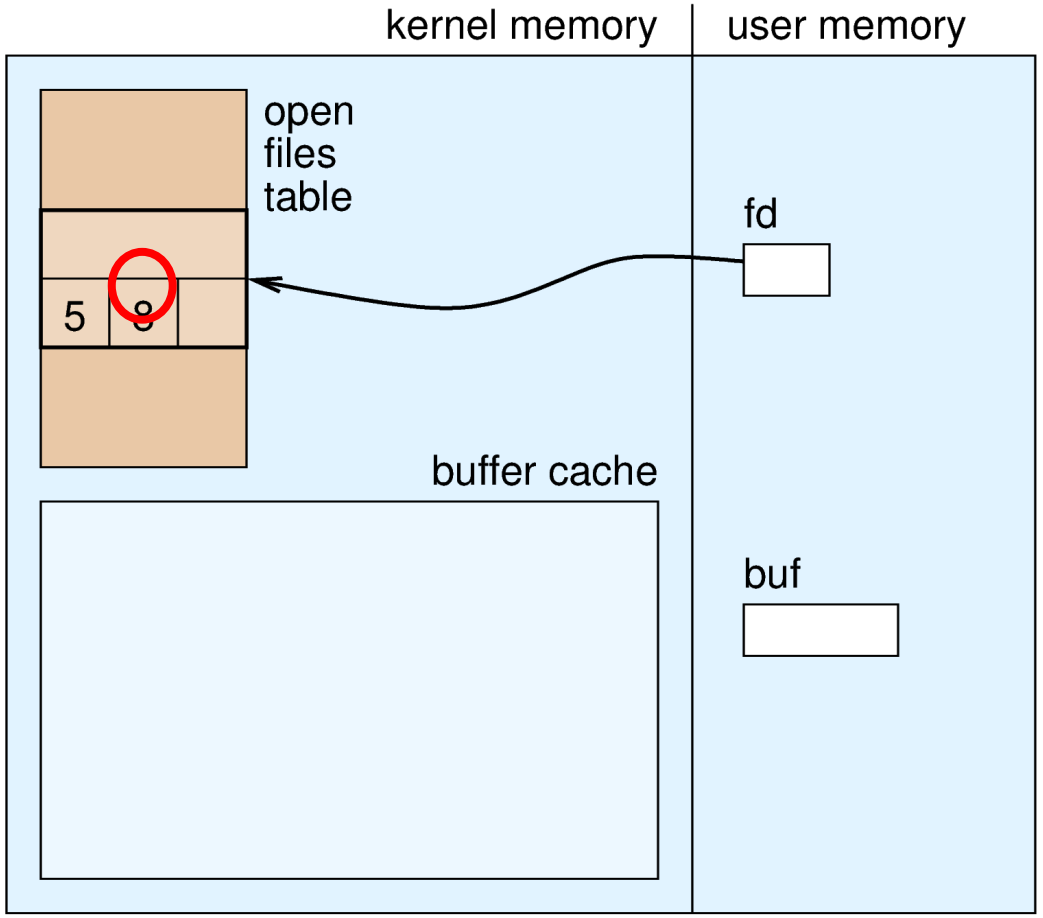


Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```

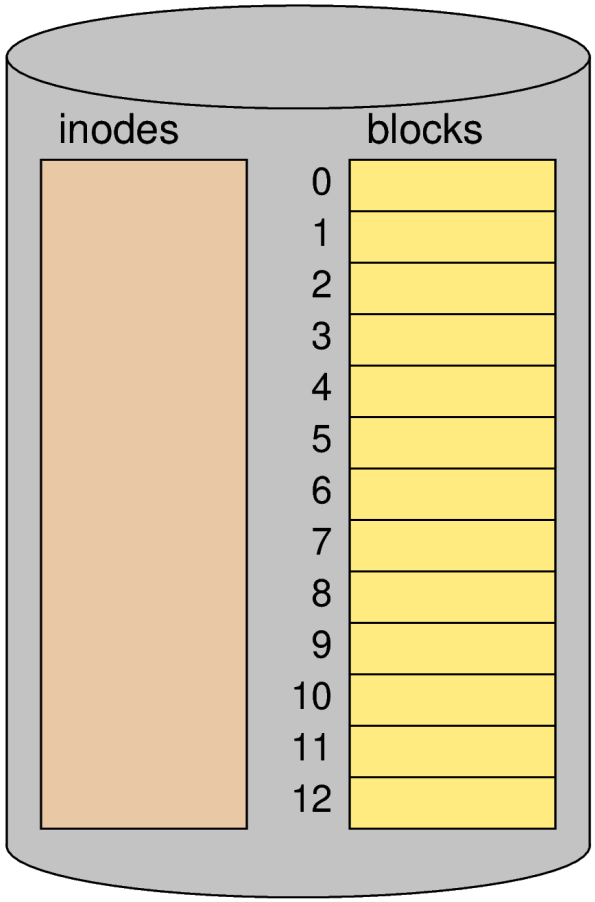



```
// write 3000 bytes at offset 7000
```

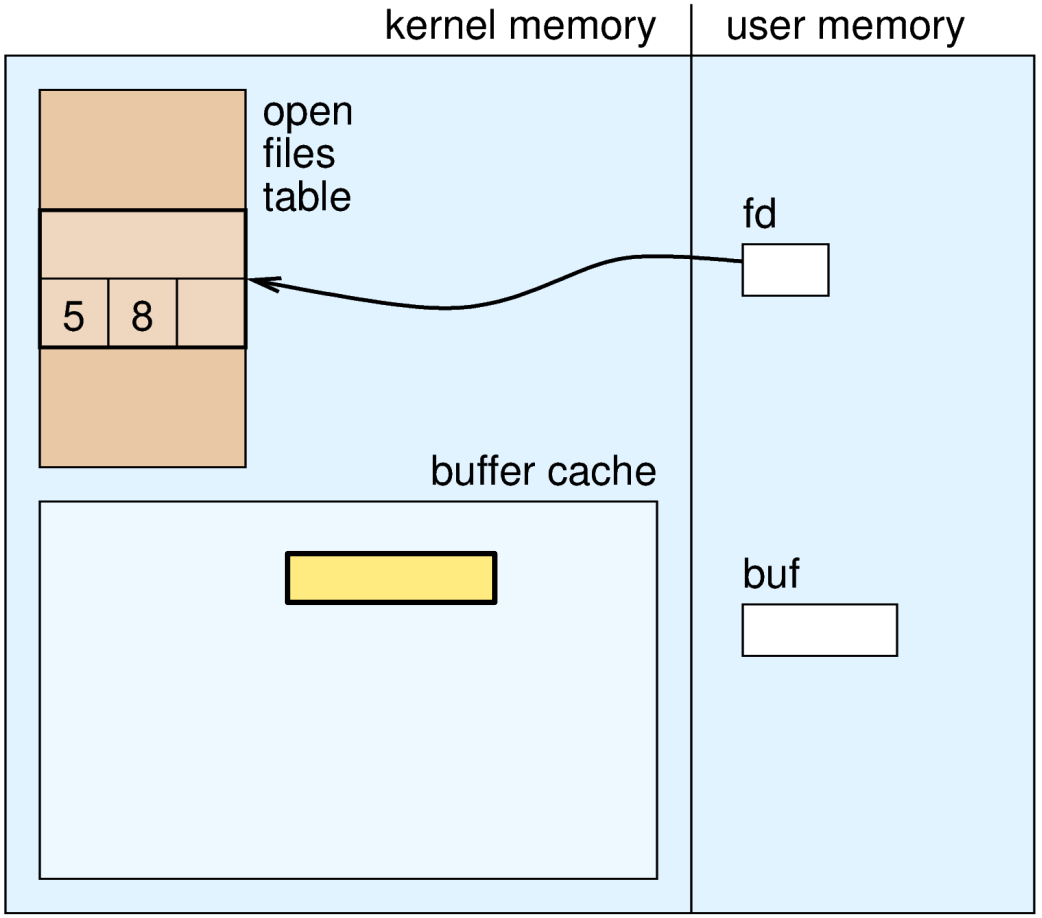


Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```

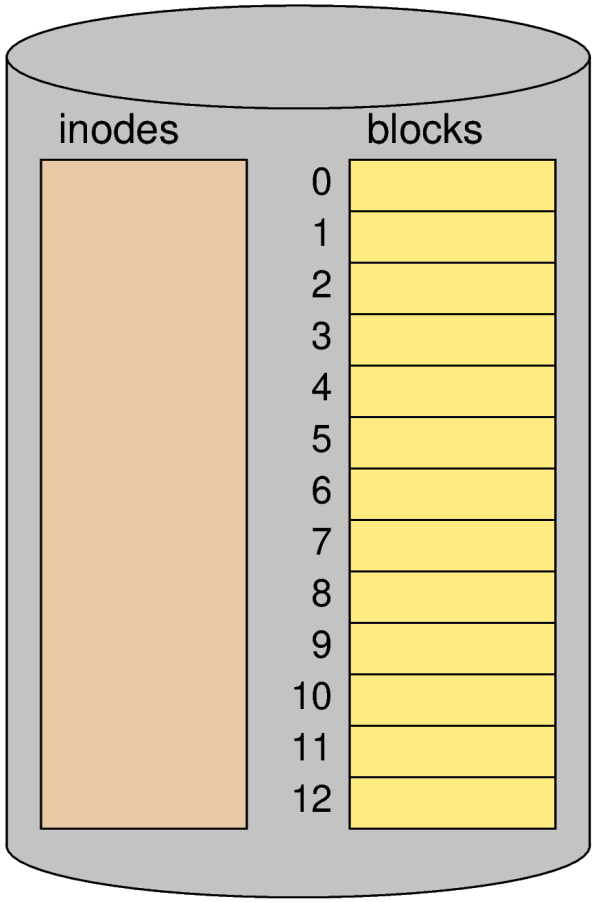



```
// write 3000 bytes at offset 7000
```

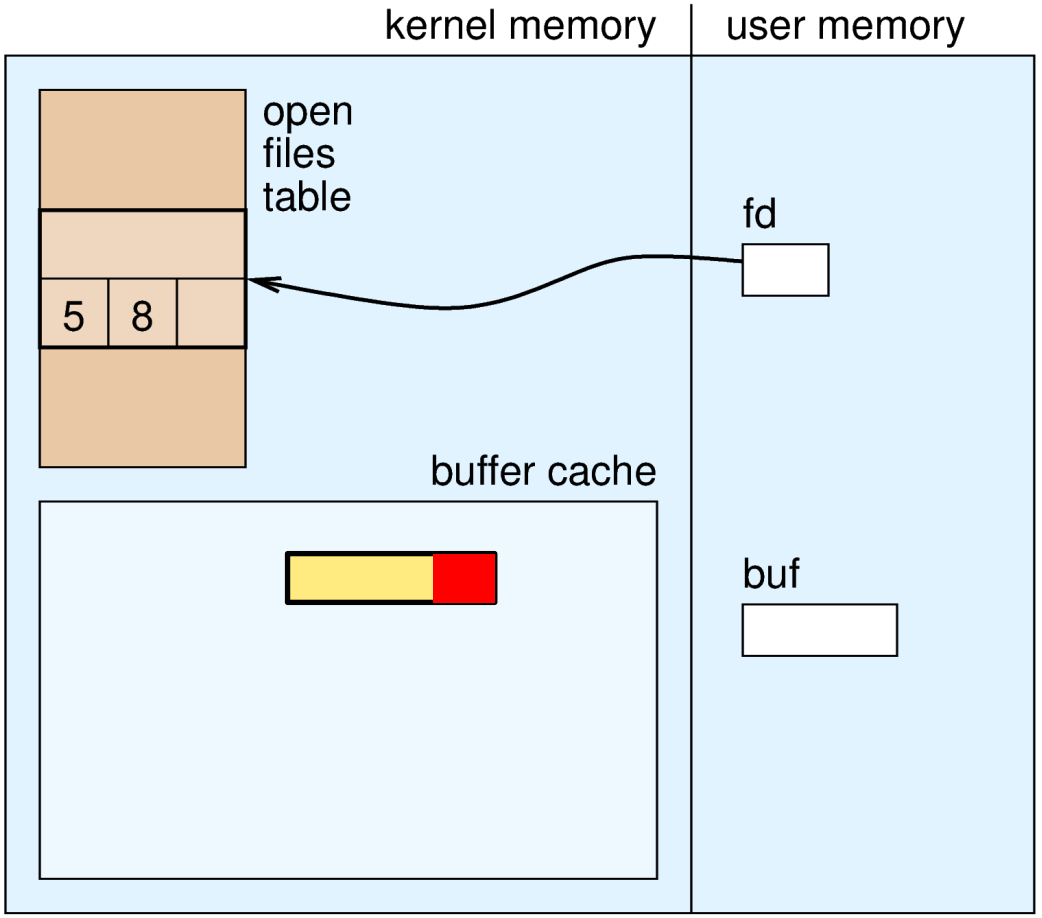


Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```



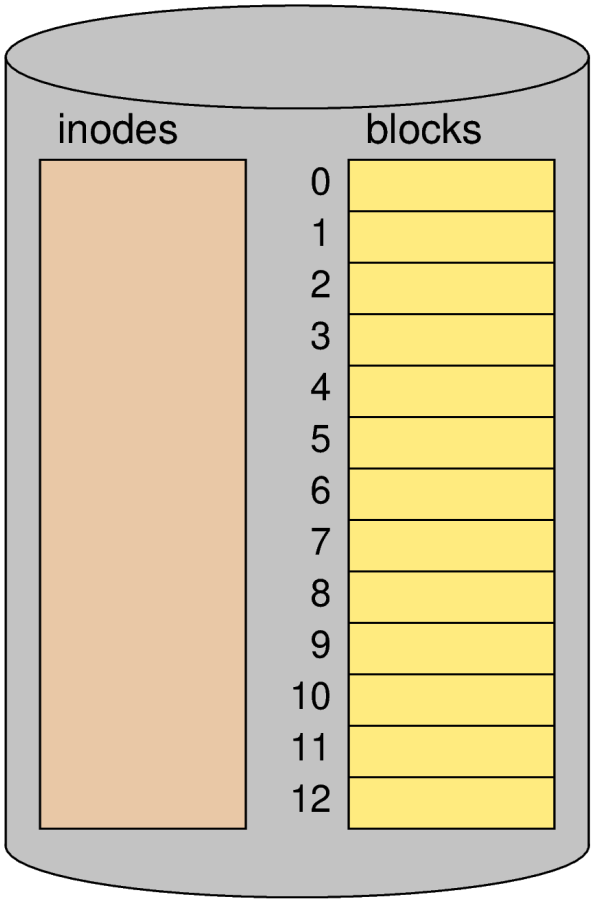

```
// write 3000 bytes at offset 7000
```

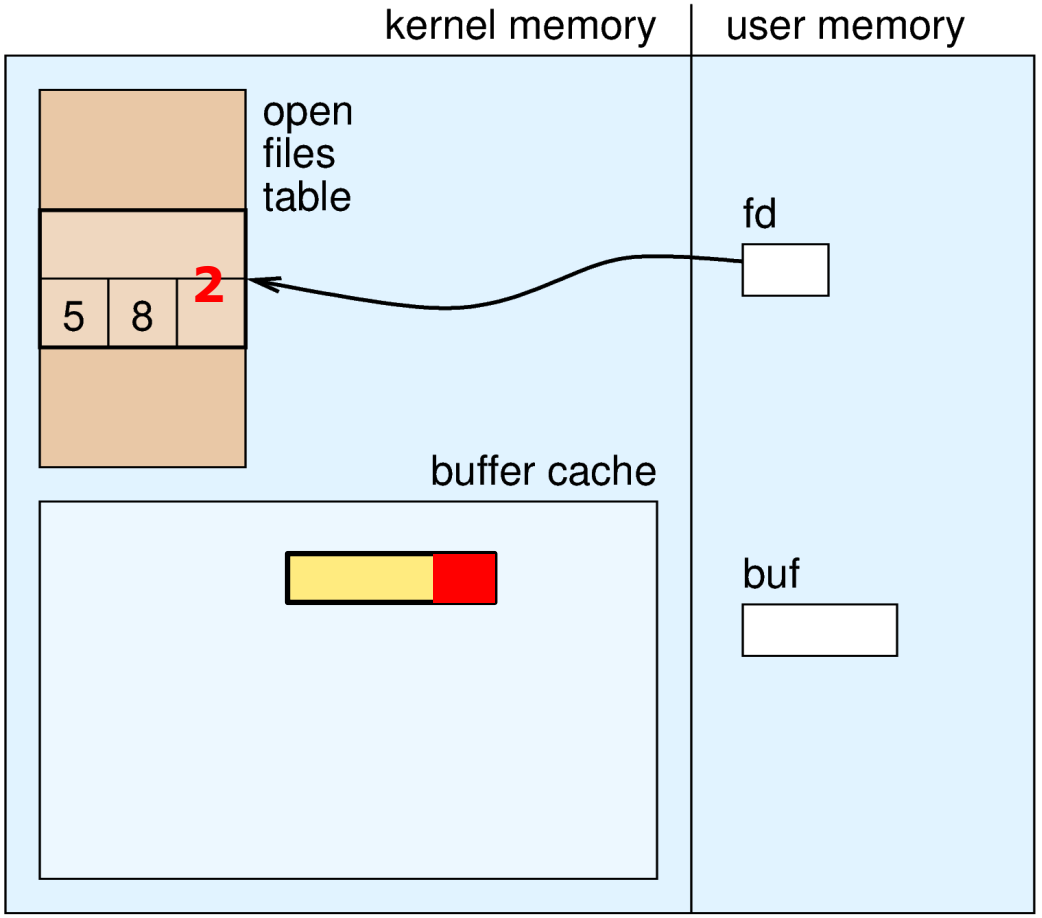


Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```

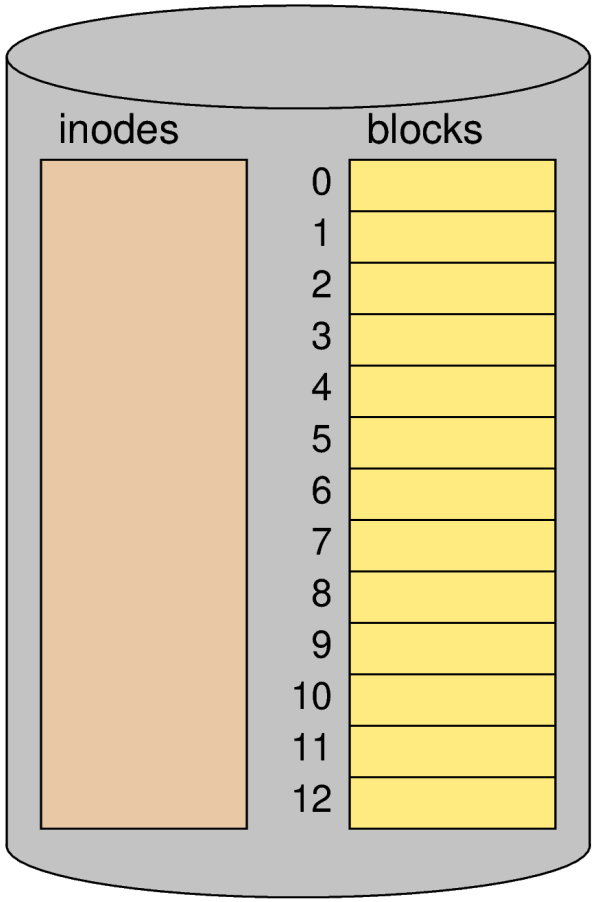
```
// write 3000 bytes at offset 7000
```



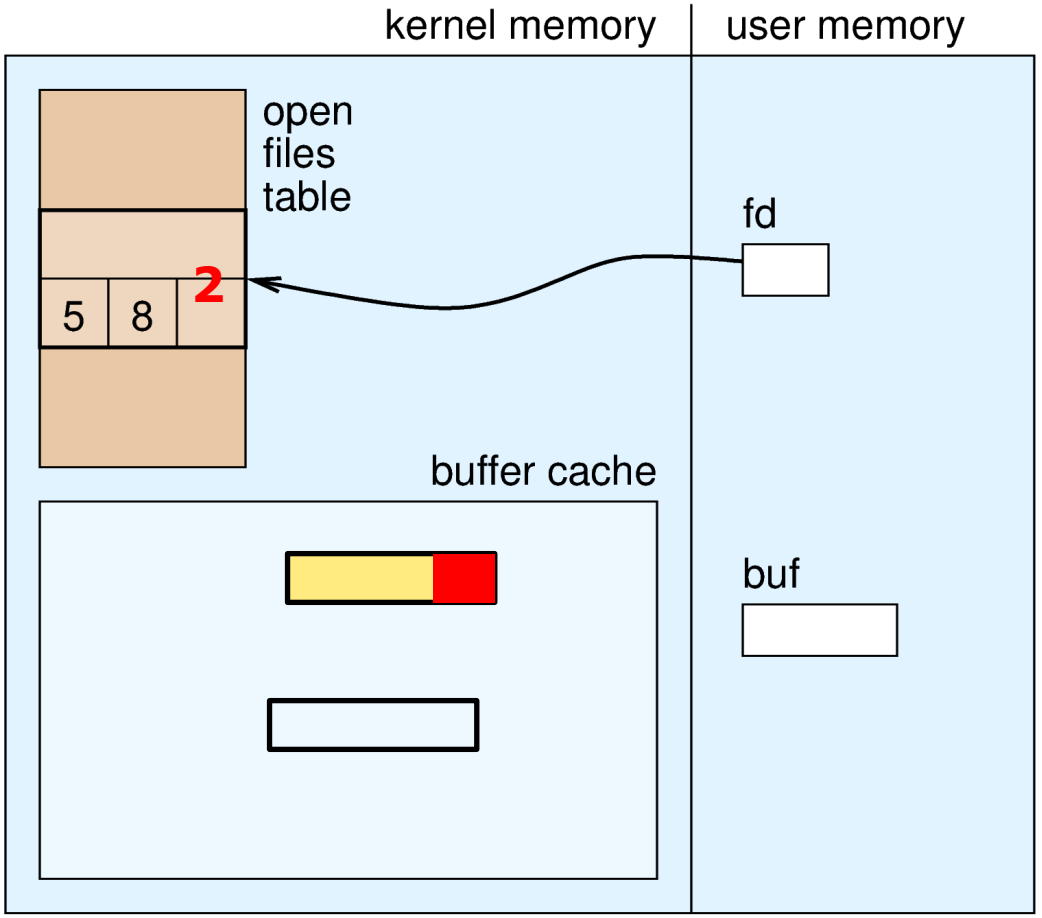


Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```

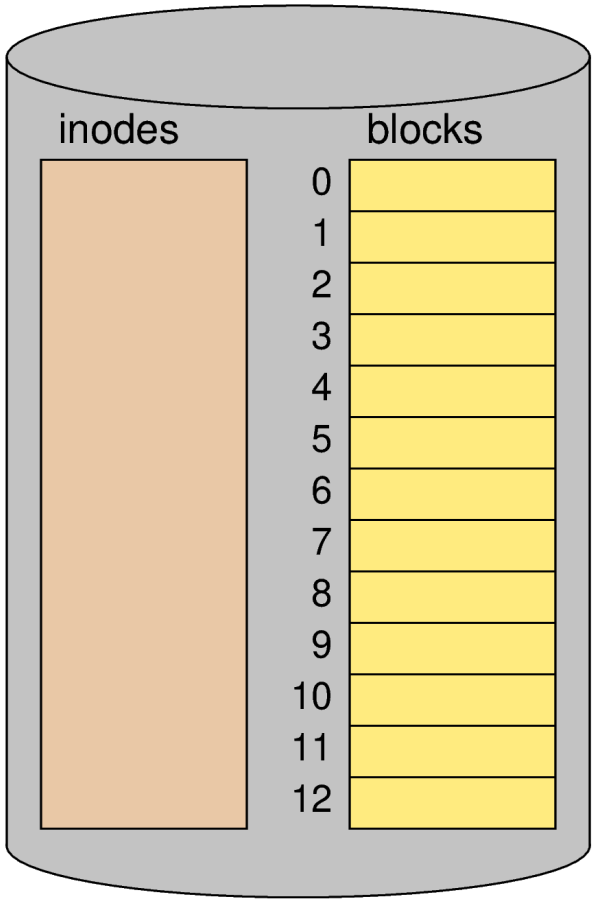



```
// write 3000 bytes at offset 7000
```

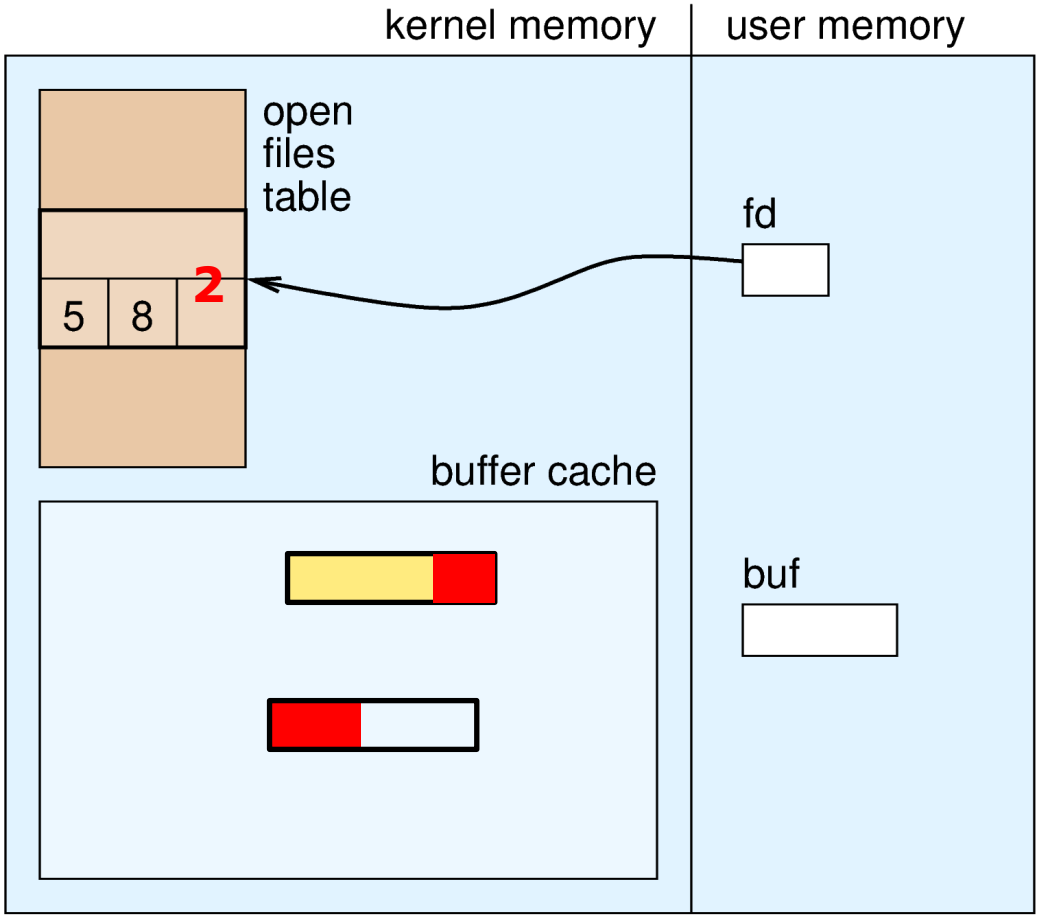


Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```

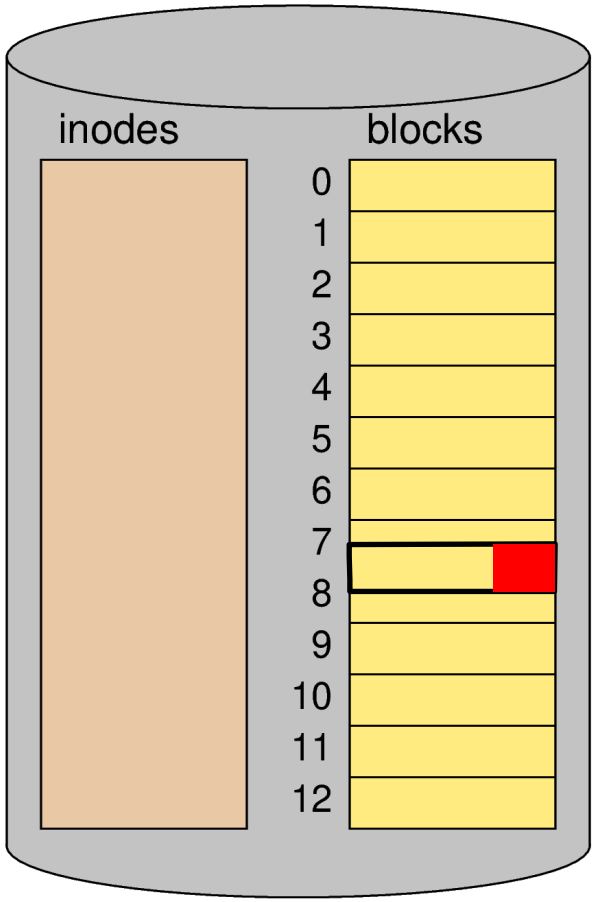



```
// write 3000 bytes at offset 7000
```

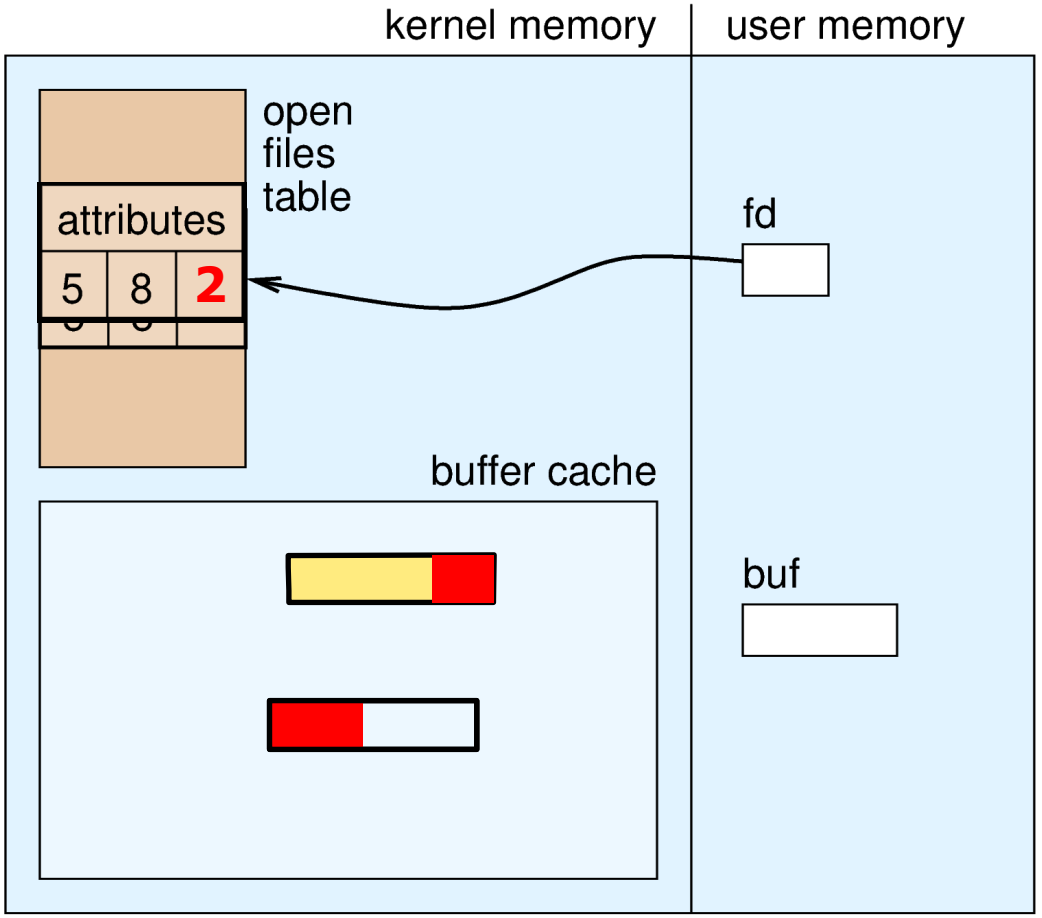


Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```

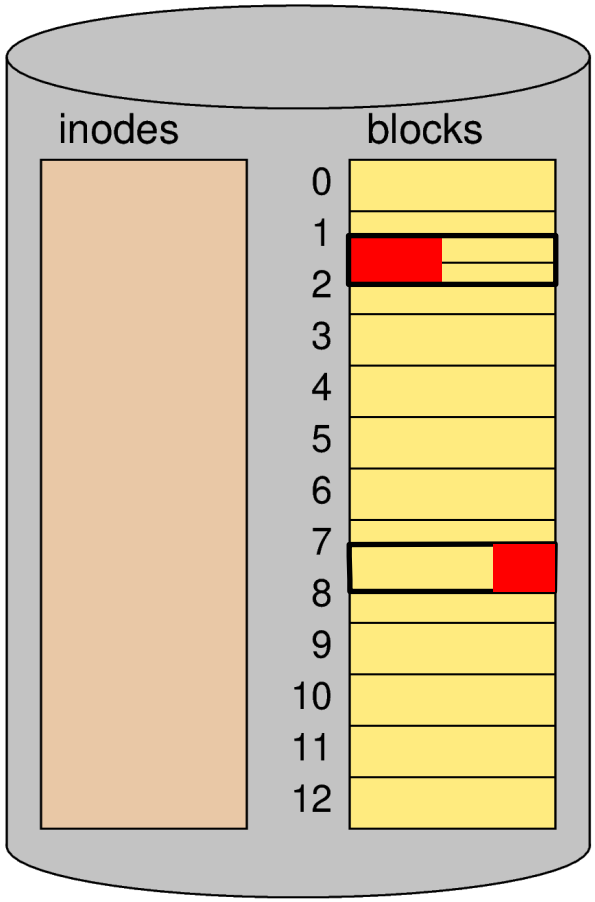



```
// write 3000 bytes at offset 7000
```

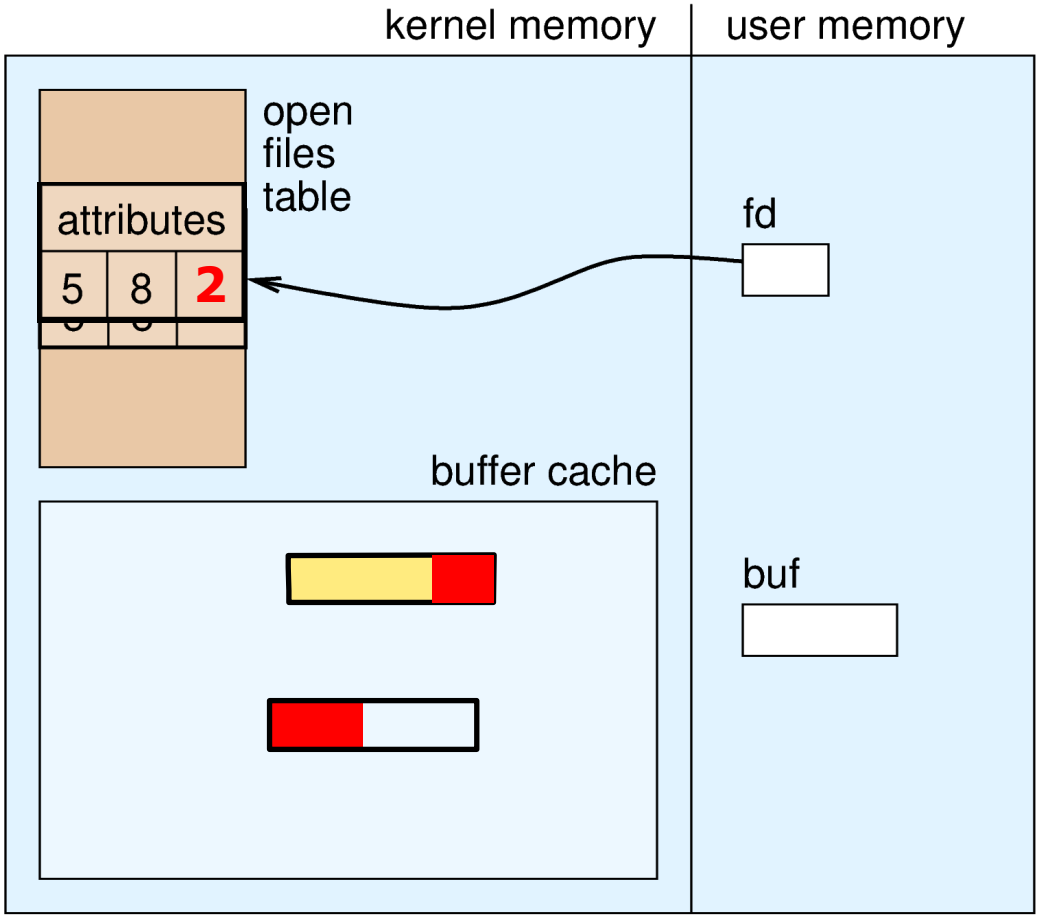


Writing Data

```
seek(fd, 7000);
write(fd, buf, 3000);
```

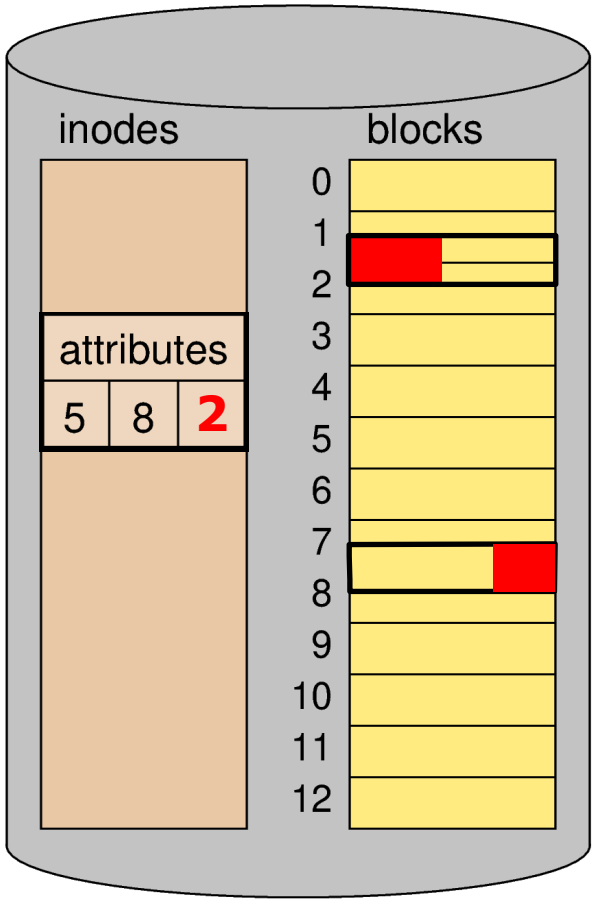



```
// write 3000 bytes at offset 7000
```

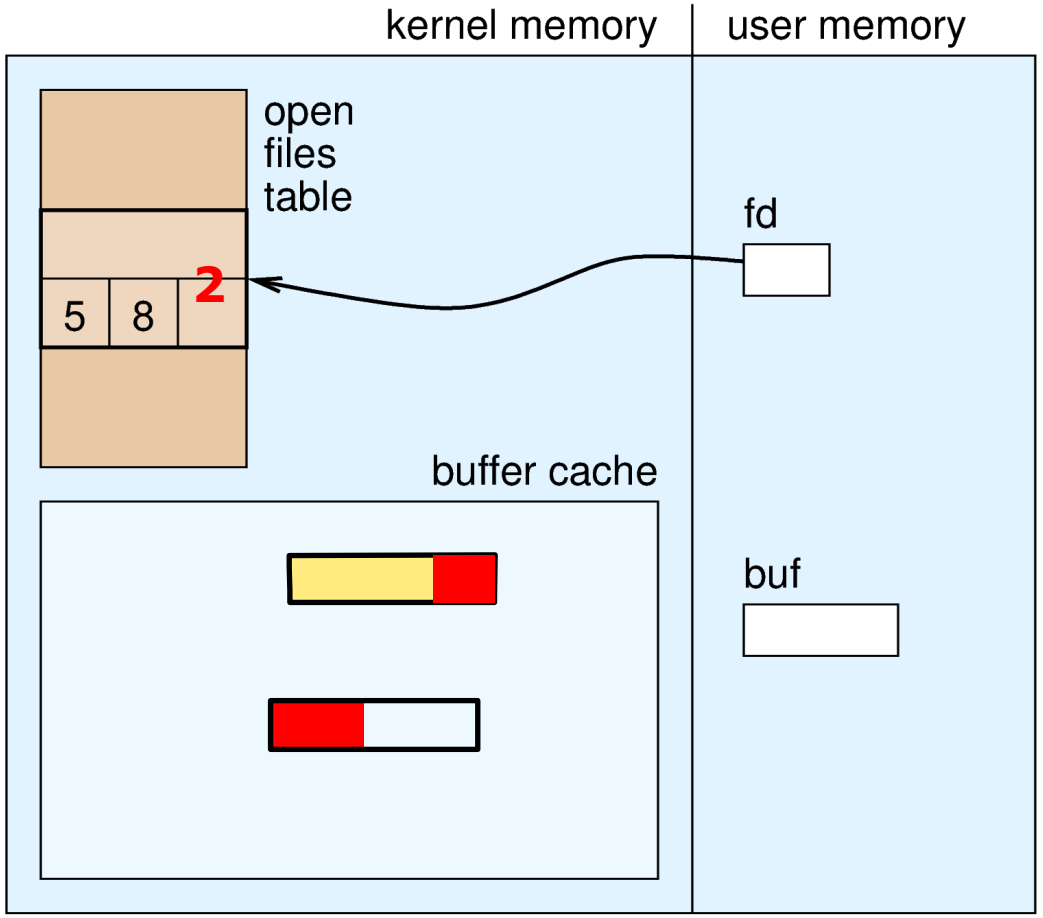


Writing Data

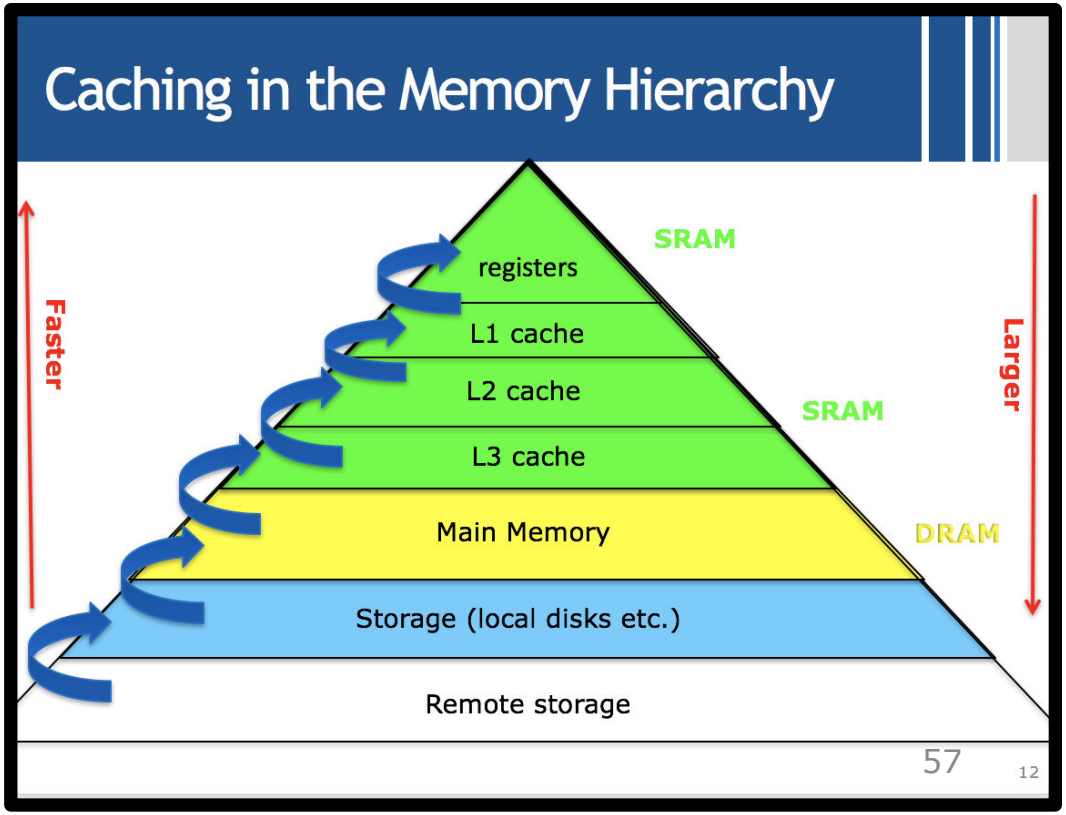
```
seek(fd, 7000);
write(fd, buf, 3000);
```




```
// write 3000 bytes at offset 7000
```

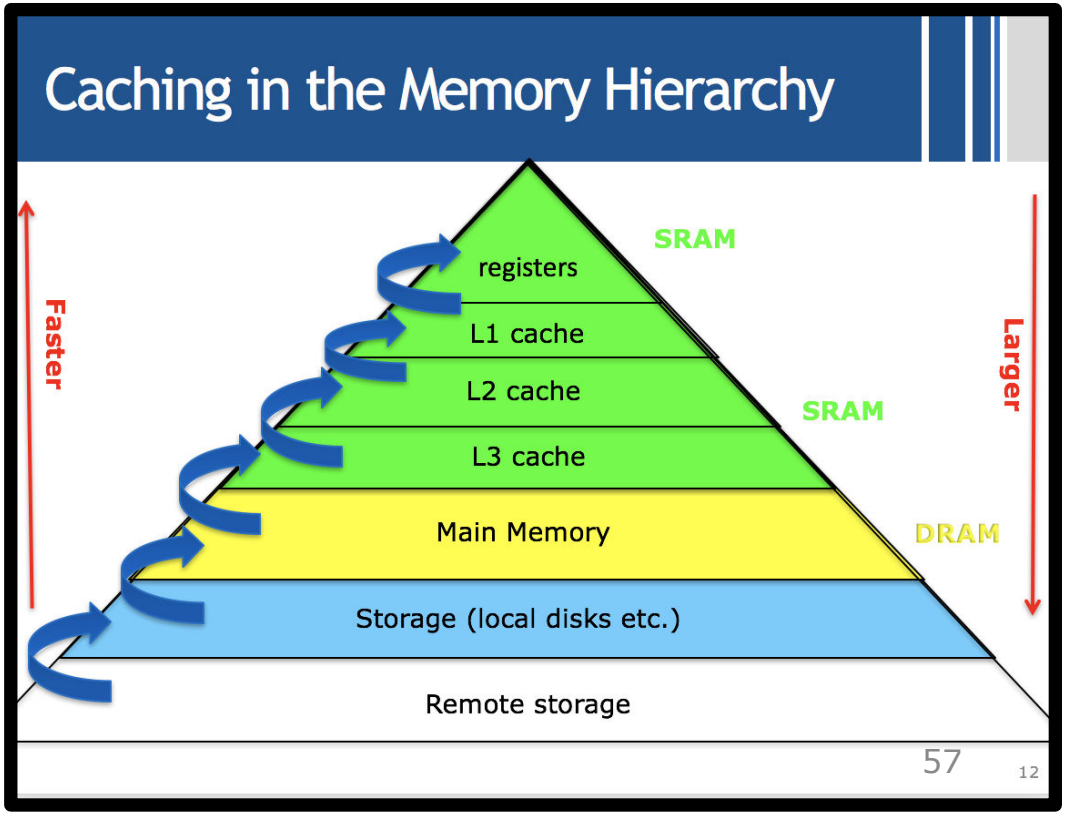


The Buffer Cache



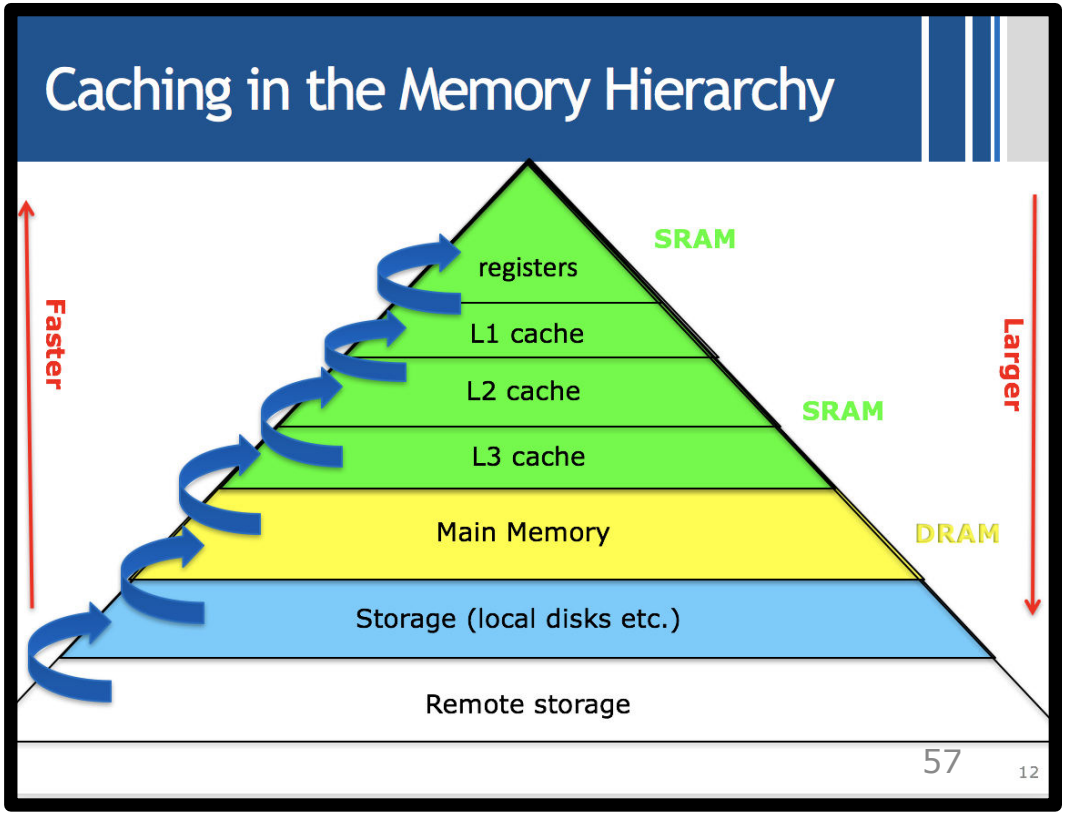
The Buffer Cache

- Separate section of main memory for file blocks



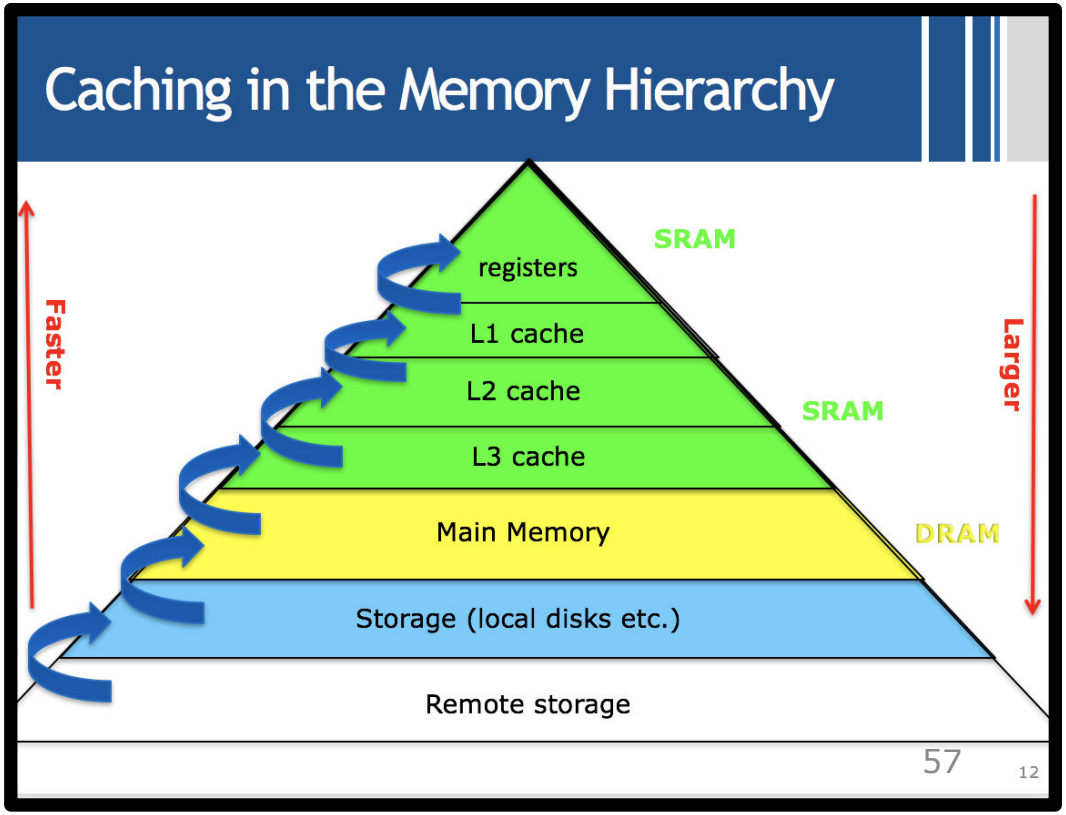
The Buffer Cache

- Separate section of main memory for file blocks
- Needed for partial block accesses



The Buffer Cache

- Separate section of main memory for file blocks
- Needed for partial block accesses
- Also improve performance by avoiding repeated accesses
  - Half of written data is overwritten within 5 minutes



Buffer Cache Management

- Done by OS at disk speeds, so not constrained like HW cache management
- Need to find arbitrary disk blocks
  - Use hash and link blocks by their hash key
- Need to decide what to evict to make space
  - Put all blocks on a big LRU linked list

Memory-Mapped Files

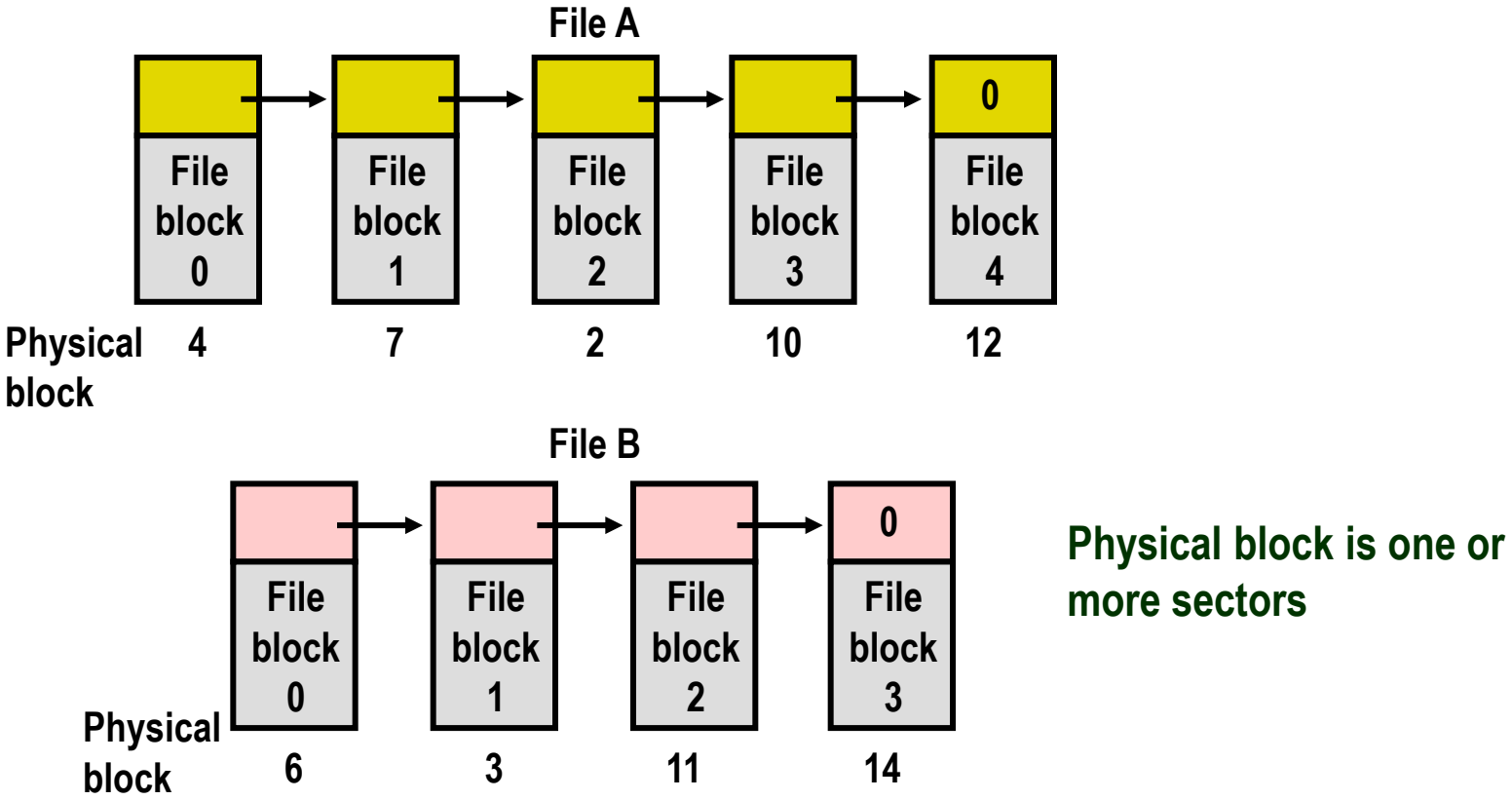
Combine files with memory management!

- File mapping:
  - Define a new segment of memory
  - Tell the OS that it is swapped out to a file
- Mapped file usage:
  - Read: just access data at a certain mapped addresses – will cause page faults if not there
  - Write: write data to a mapped address
- Advantages:
  - Save memory space for buffer cache
  - Save kernel-to-user copy by using direct access

Maintaining File Data Location

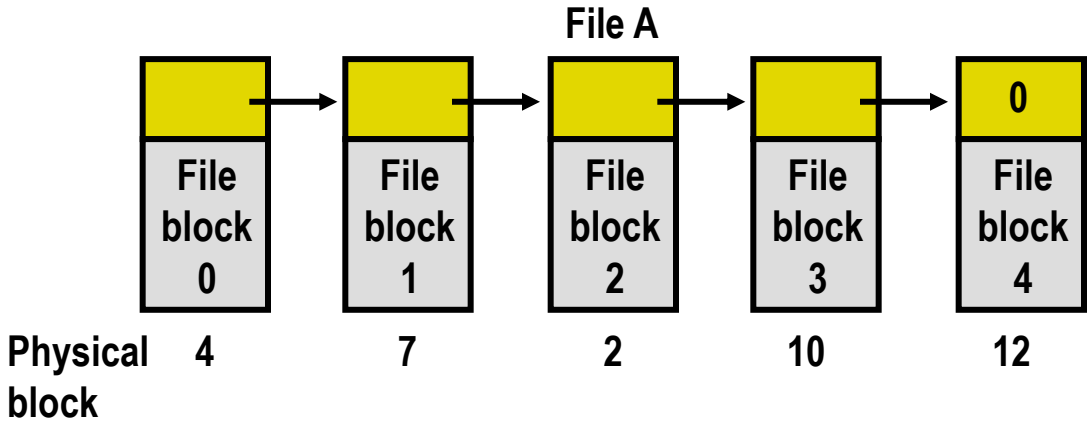
- File data is stored in a sequence of blocks
- Need to map these blocks
  - And their order!
- Option 1: linked list (FAT in MS-DOS)
- Option 2: index (inodes in Unix/Linux)

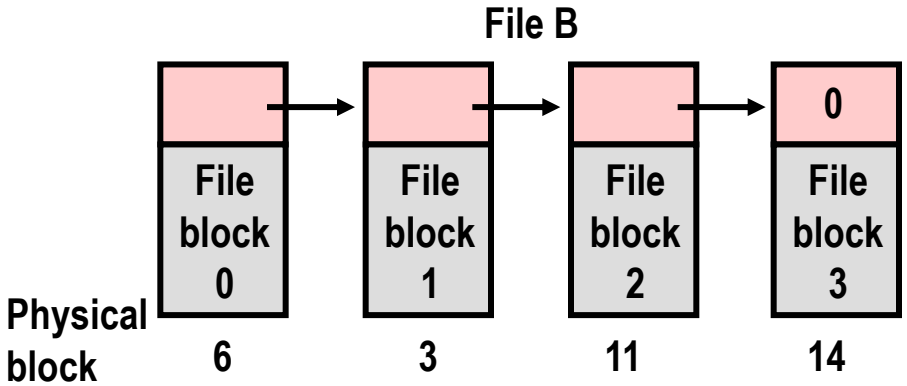
Naïve Linked List of File Blocks



Physical block is one or more sectors

Naïve Linked List of File Blocks



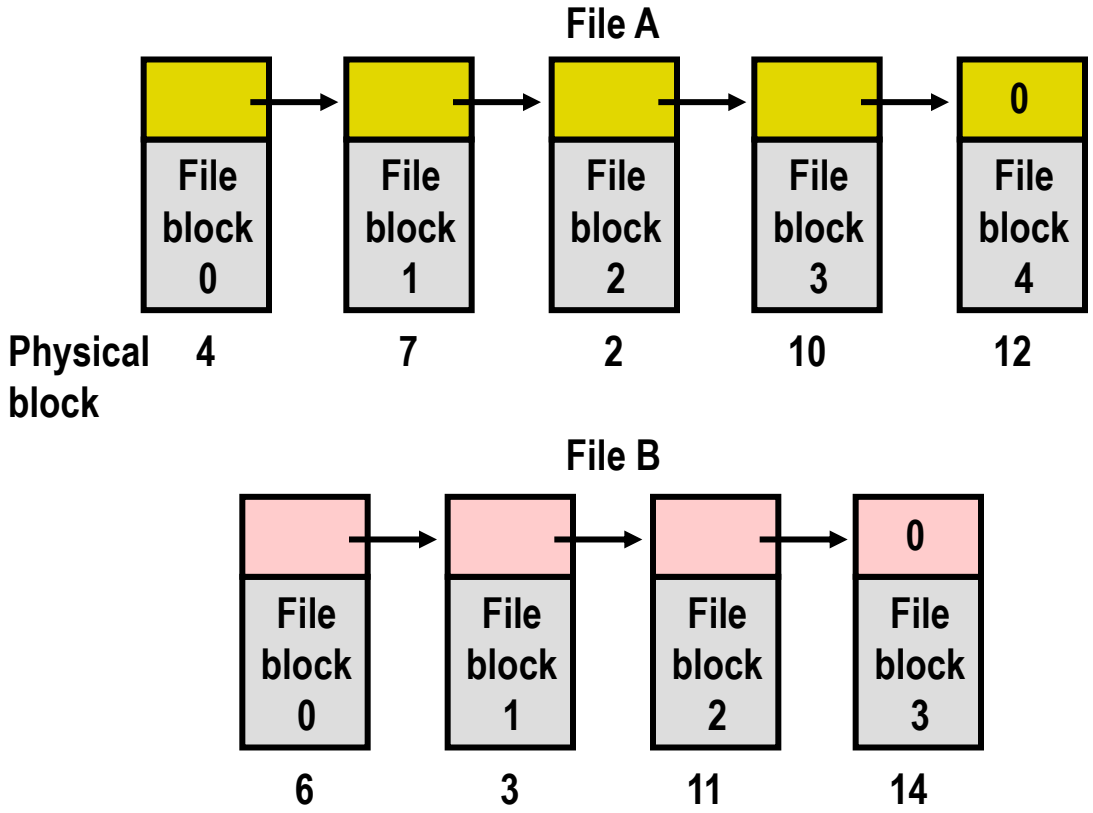


Physical block is one or more sectors

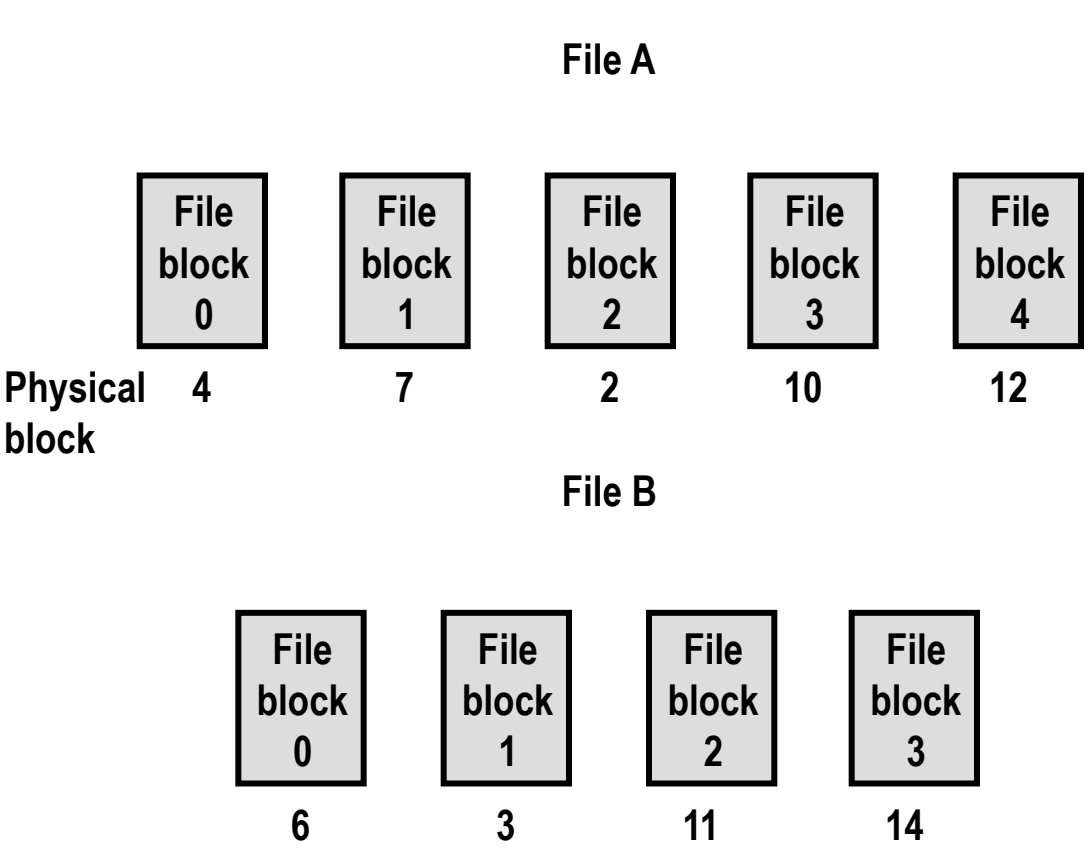
Problems:

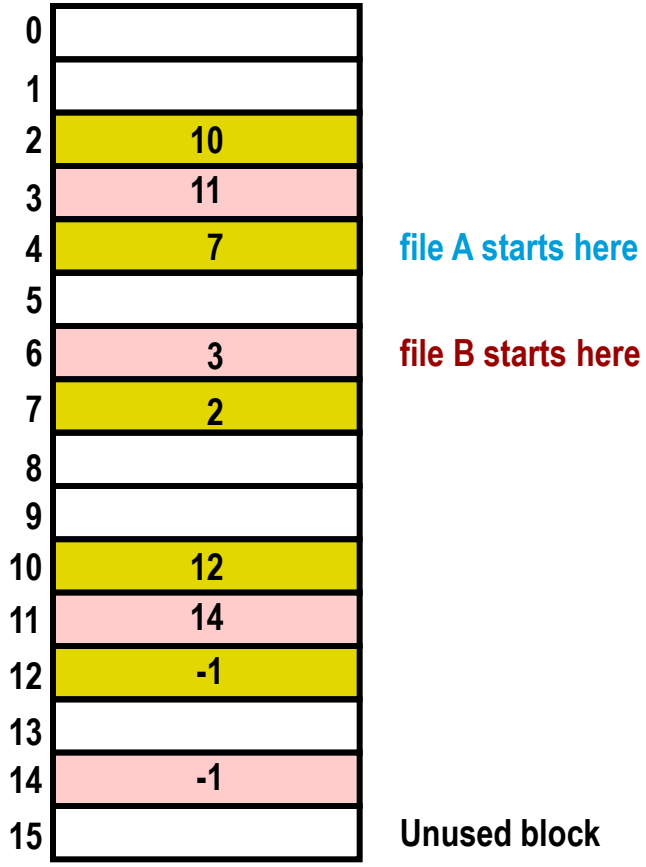
1. Random access is extremely slow
2. Block is no longer a power of 2 because the pointer takes up a few bytes

Linked List using FAT



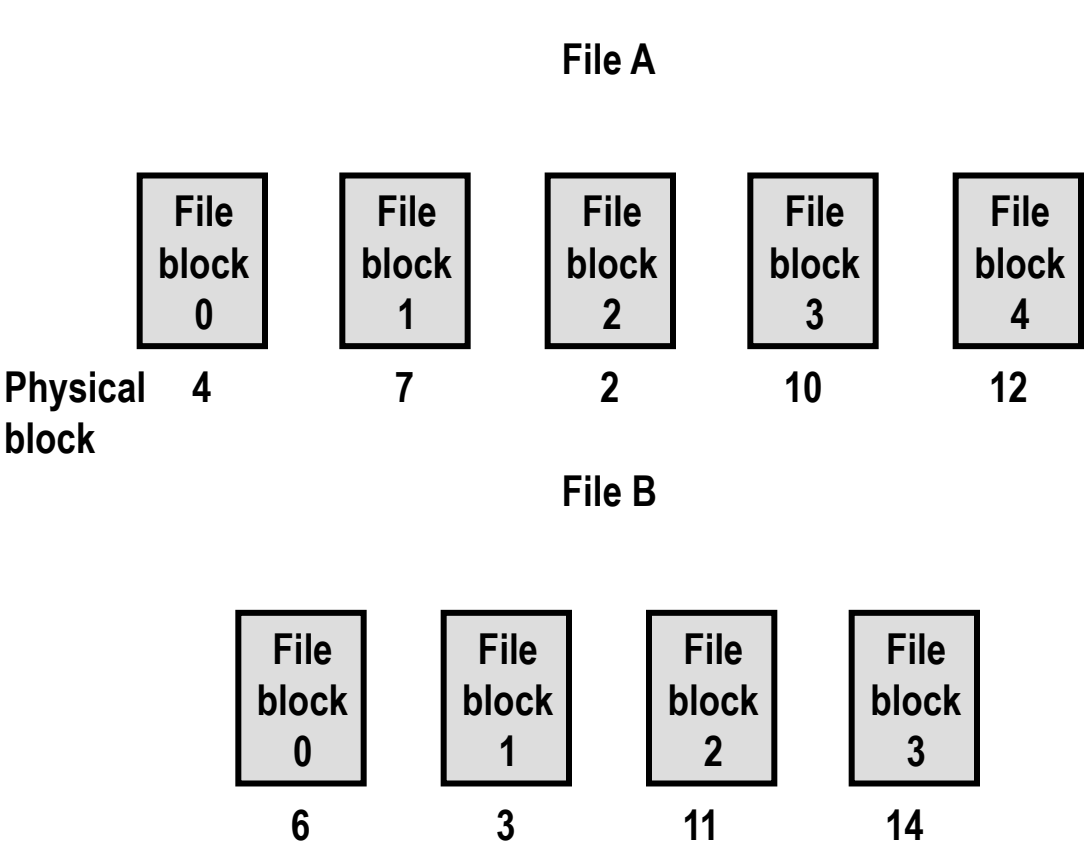
Linked List using FAT



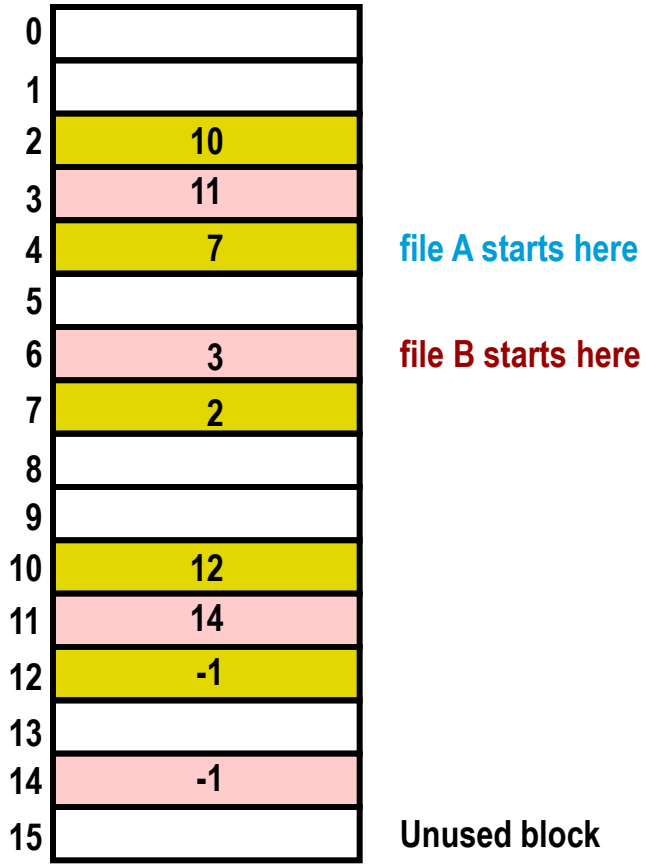


File Allocation Table (Stored in Memory)

Linked List using FAT

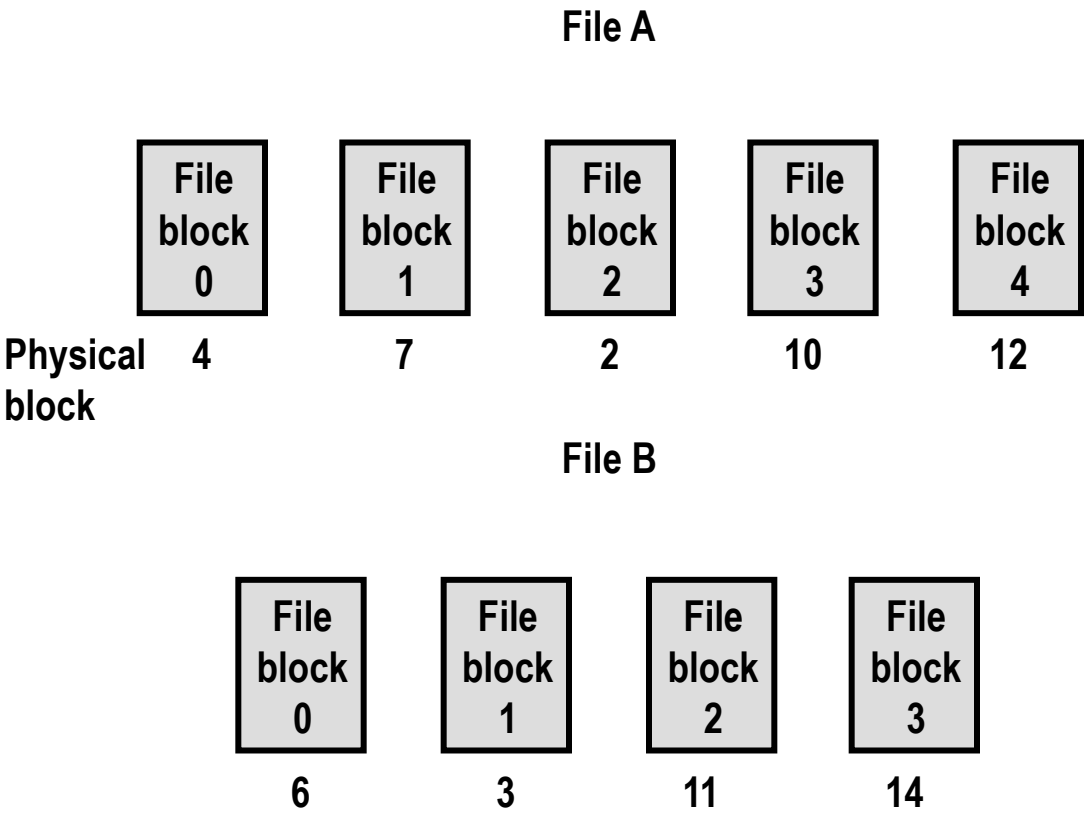


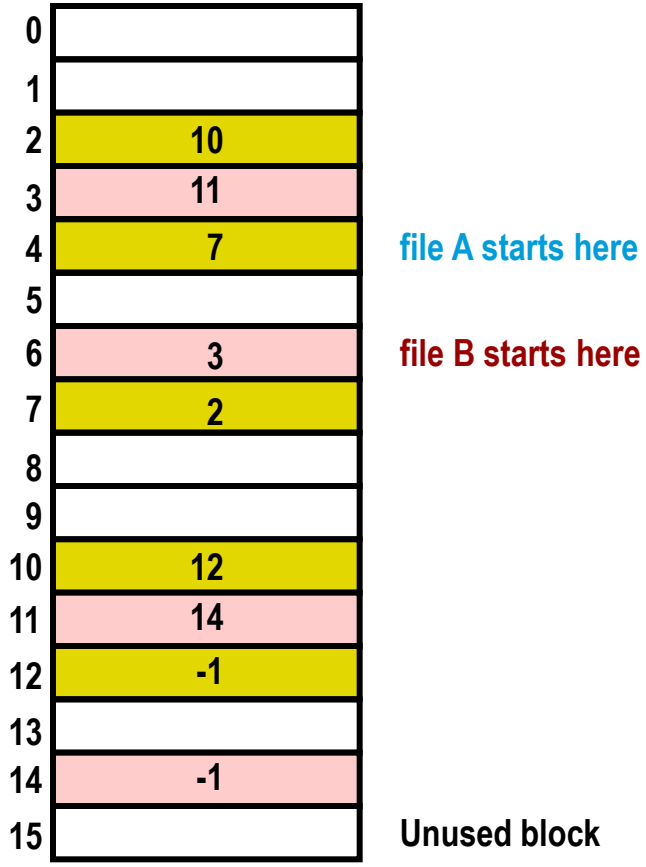
Block for attributes (File Control Block)



File Allocation Table (Stored in Memory)

Linked List using FAT





File Allocation Table (Stored in Memory)

Block for attributes (File Control Block)

Problems:

1. FAT can be very large
2. Reading a file can cause many seeks

FAT Discussion

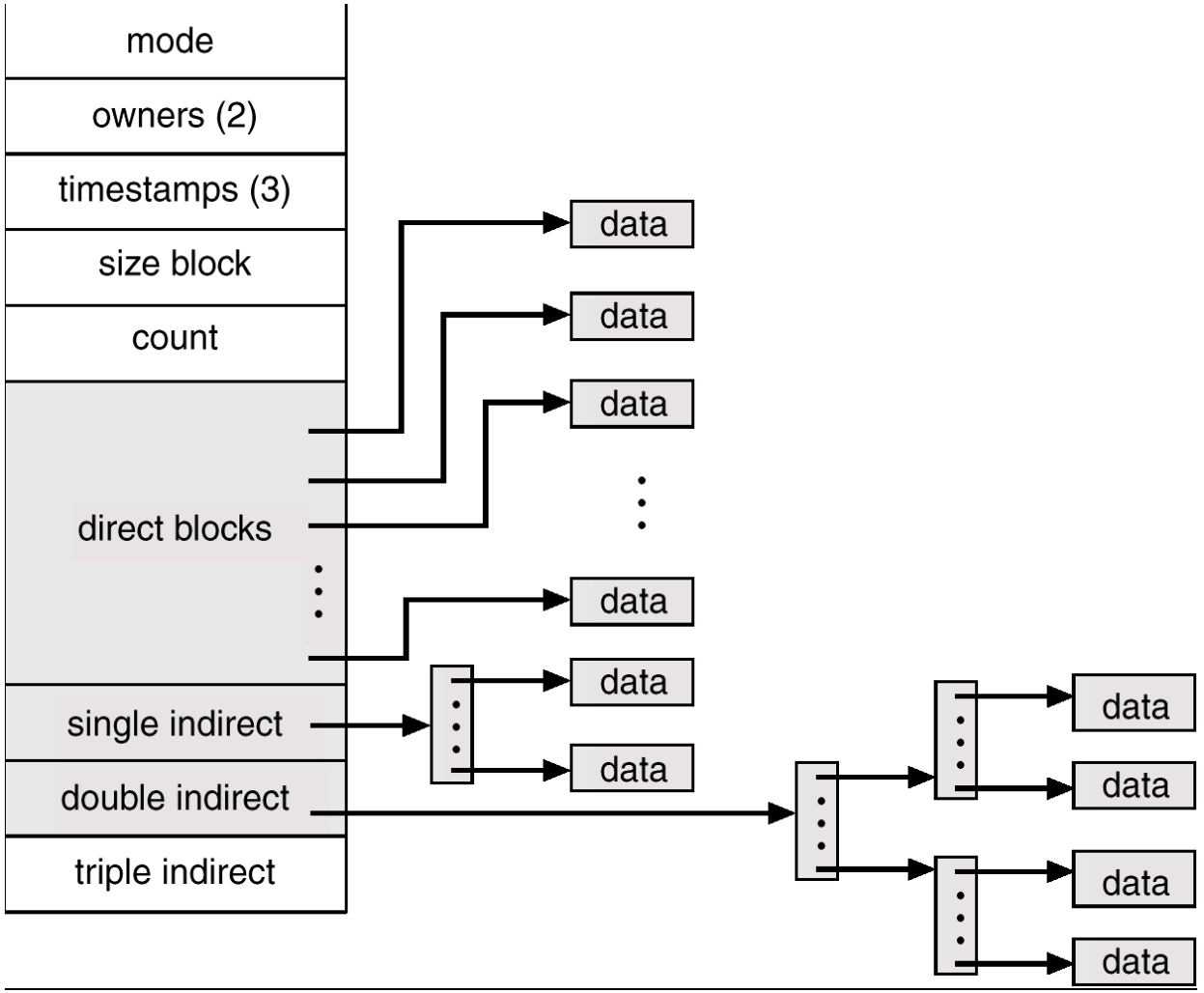
- Pros:
  - Entire block is available for data
  - Random access is much faster than linked list (no disk accesses)

FAT Discussion

- Pros:
  - Entire block is available for data
  - Random access is much faster than linked list (no disk accesses)

- Cons:
  - Many file seeks unless entire FAT is in memory
    - For 20 GB disk, 4KB block size, FAT has 5 million entries
    - If 4 bytes used per entry $\Rightarrow$ 20 MB of main memory required for the FAT

Indexed blocks: Unix inodes



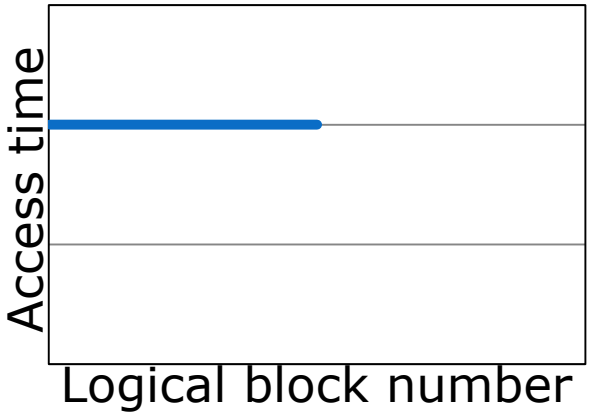
Unix inodes

- If data blocks are 4K, block numbers are 4B, and there are 12 direct pointers,
  - First 48K reachable from the inode
  - Next 4MB available from single-indirect
  - Next 4GB available from double-indirect
  - Next 4TB available through the triple-indirect block
- Any block can be found with at most 3 disk accesses

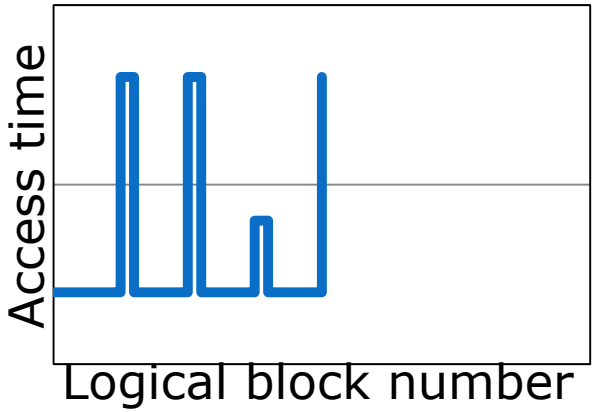
Suppose we have a file system that uses i-nodes, **with no buffer cache**. Assume we have a file with many (logical) blocks, numbered 0,1,2,…

Which graph describes better the **first access time to a block** as a function of the (logical) number of the block?

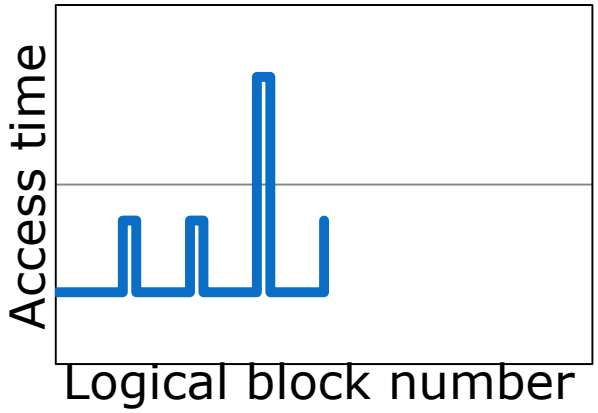
a.



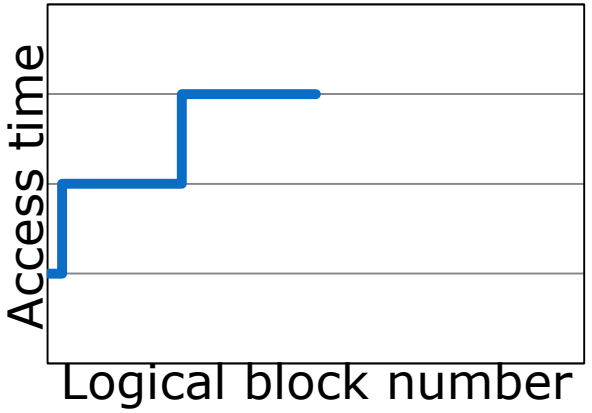
c.



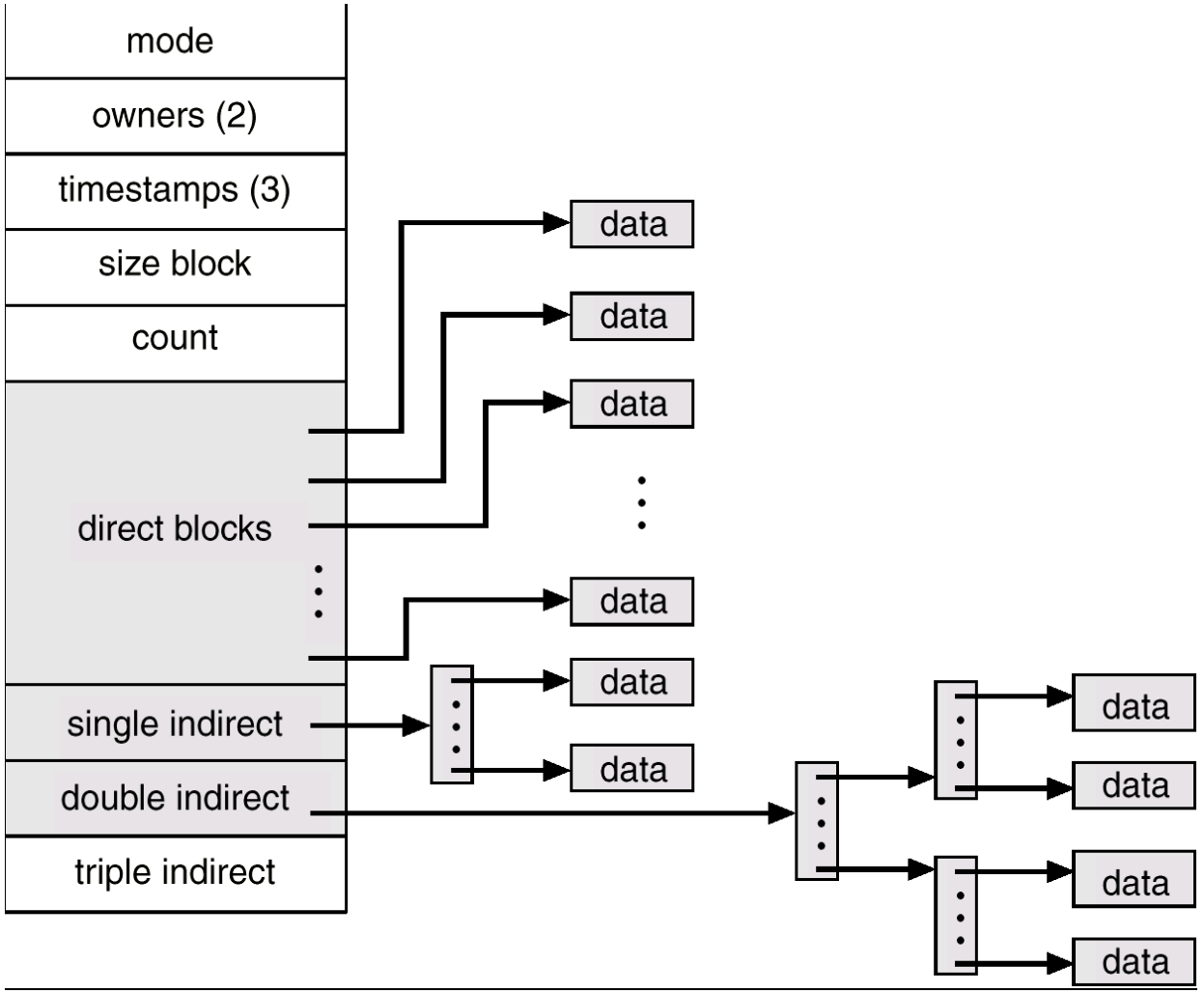
b.



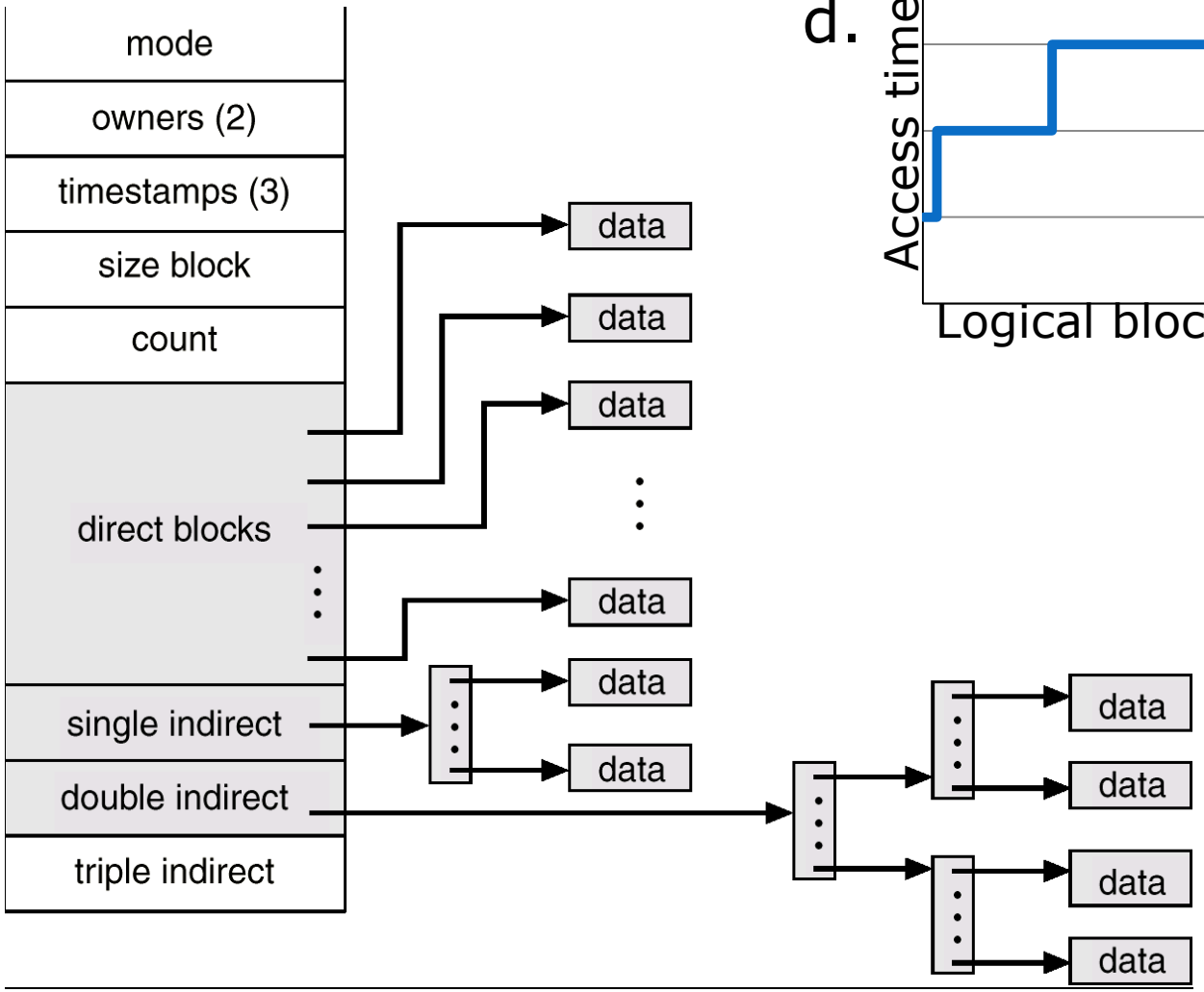
d.



inodes (Unix File System)



inodes (Unix File System)

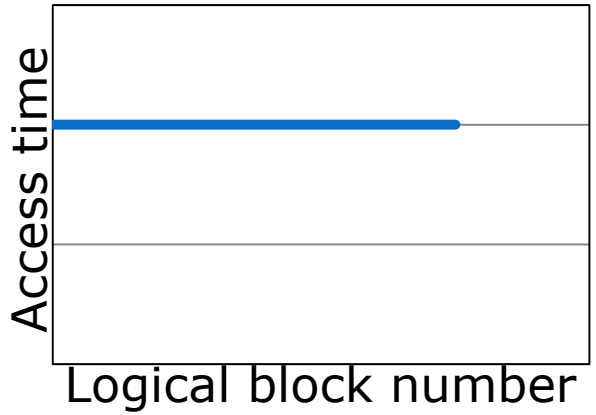


Suppose we have a file system that uses i-nodes, **with (infinite) buffer cache**.

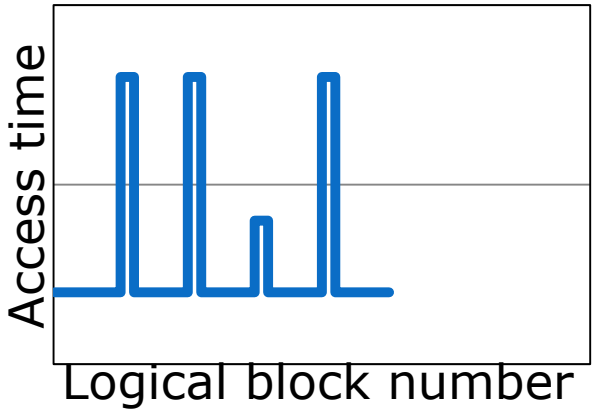
Assume we have a file with many (logical) blocks, numbered 0,1,2,…

Which graph describes better the first access time to a block as a function of the (logical) number of the block?

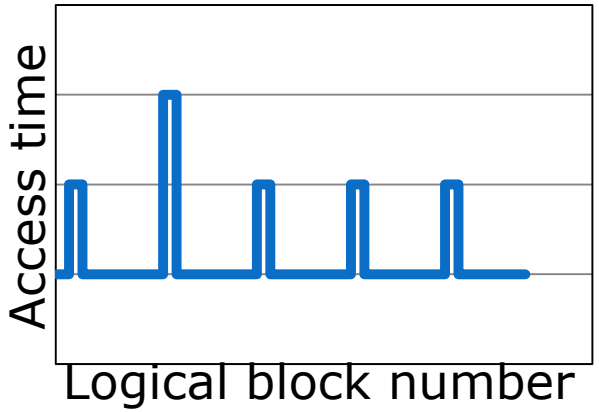
a.



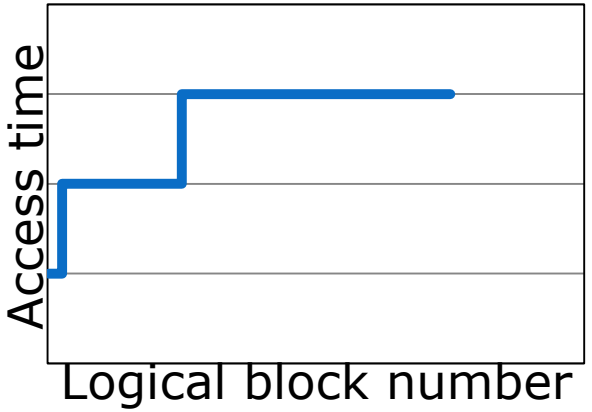
c.



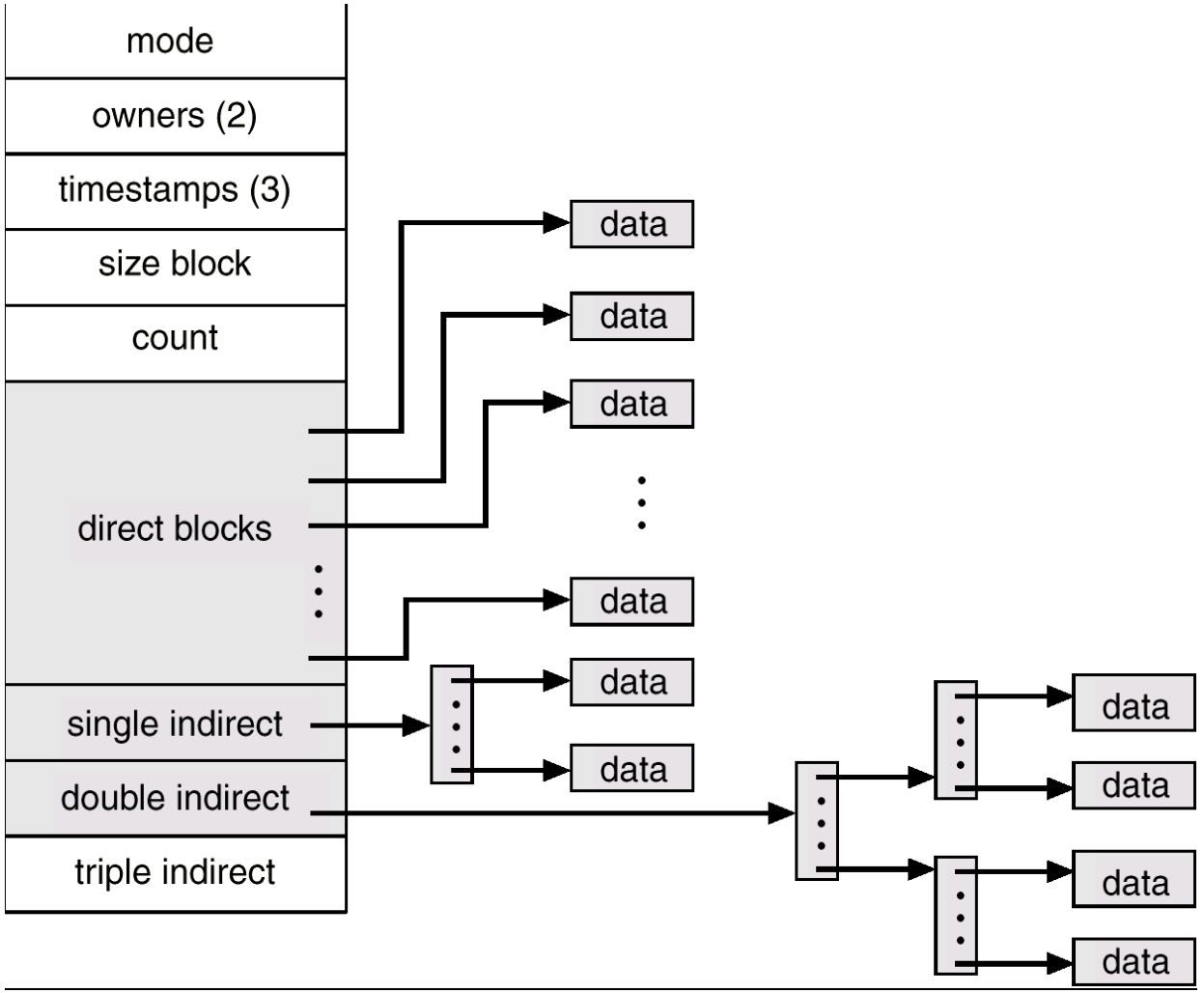
b.



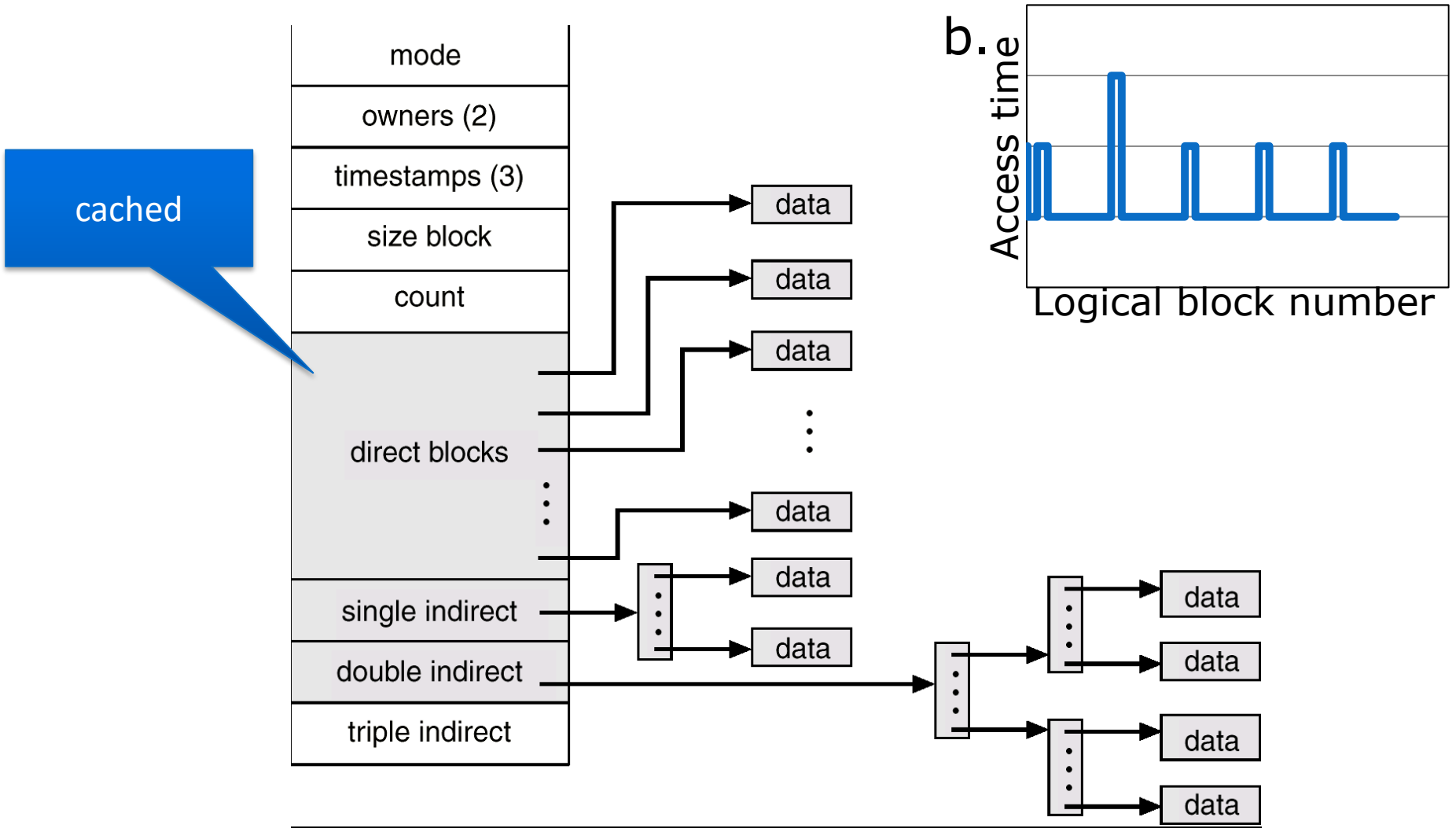
d.



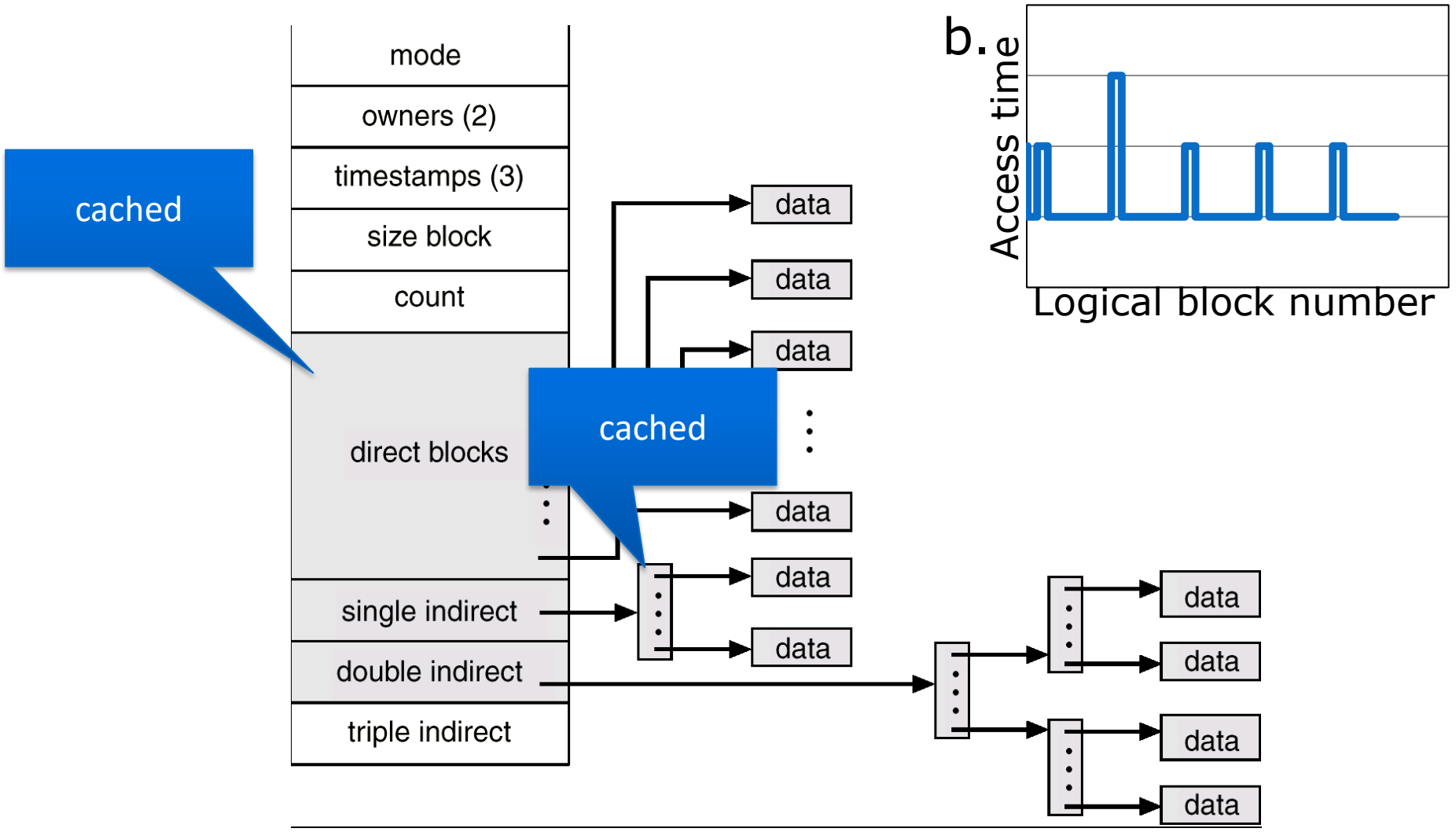
inodes (Unix File System)



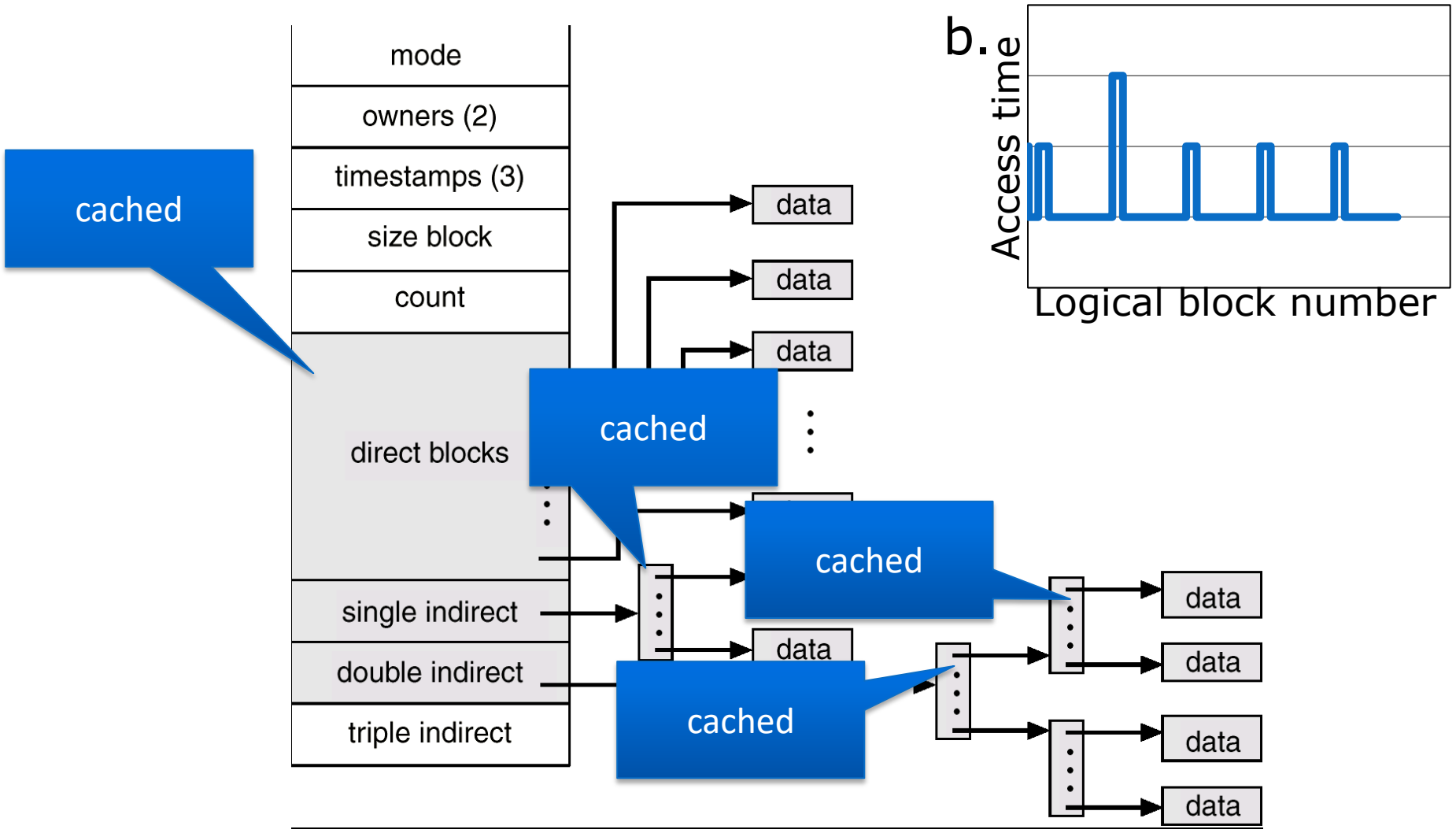
inodes (Unix File System)



inodes (Unix File System)

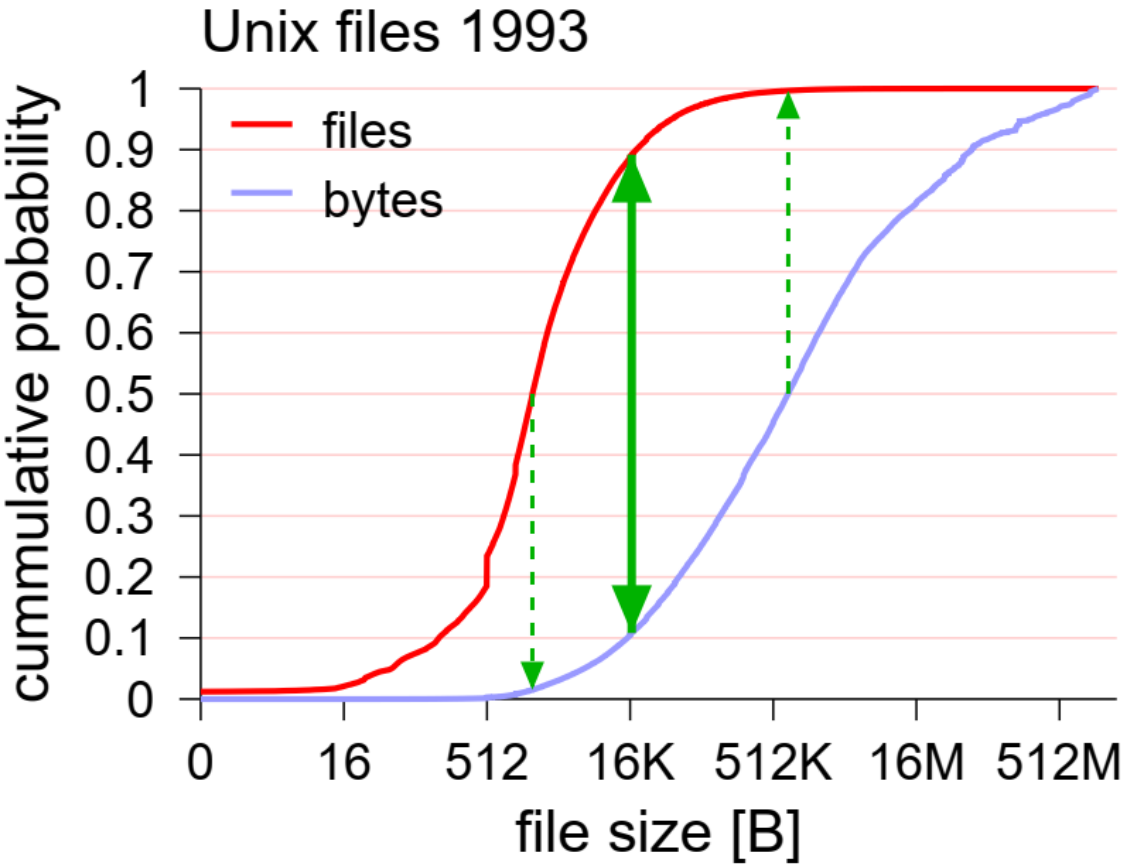


inodes (Unix File System)



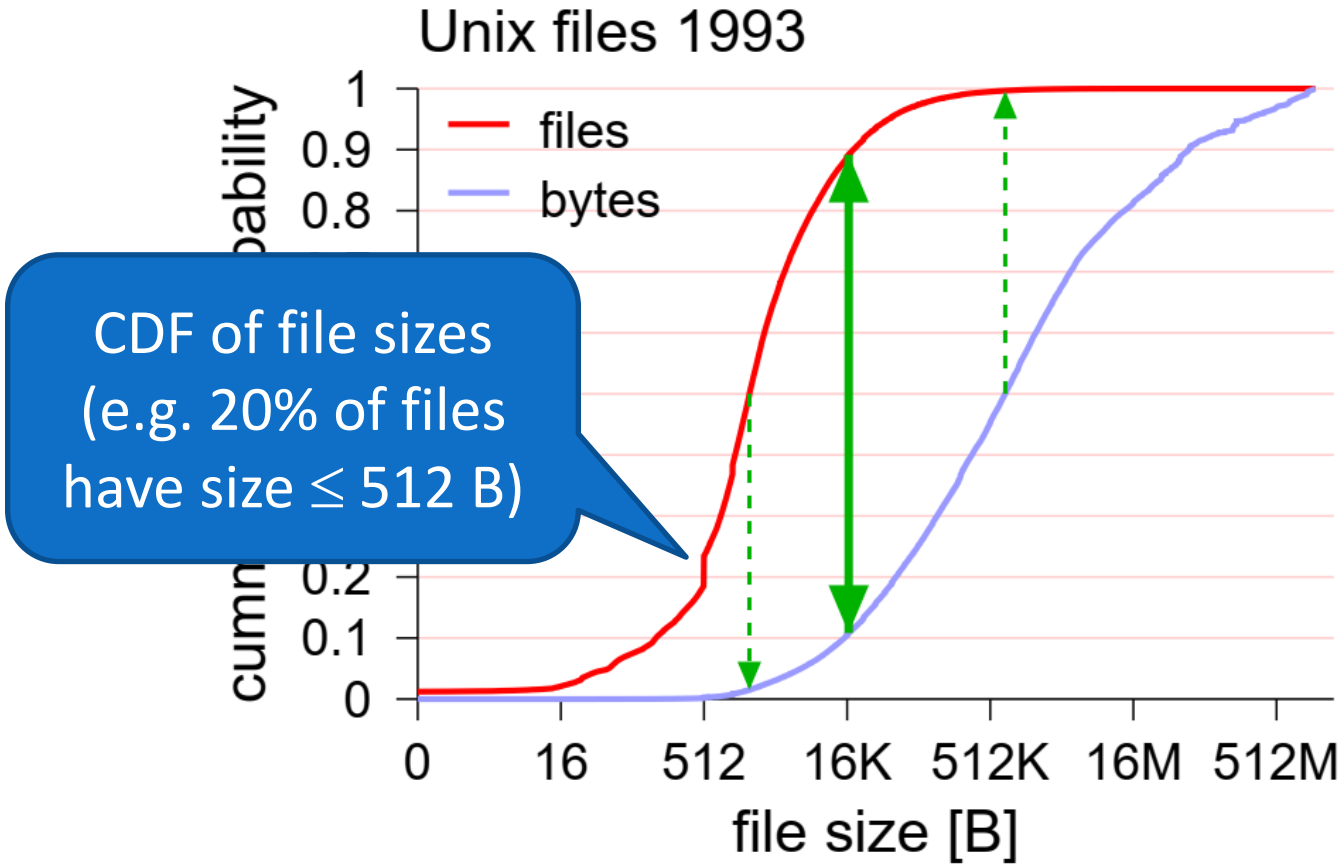
Support for Small Files

- Unix inode optimized for small files
  - While able to support (very) large files



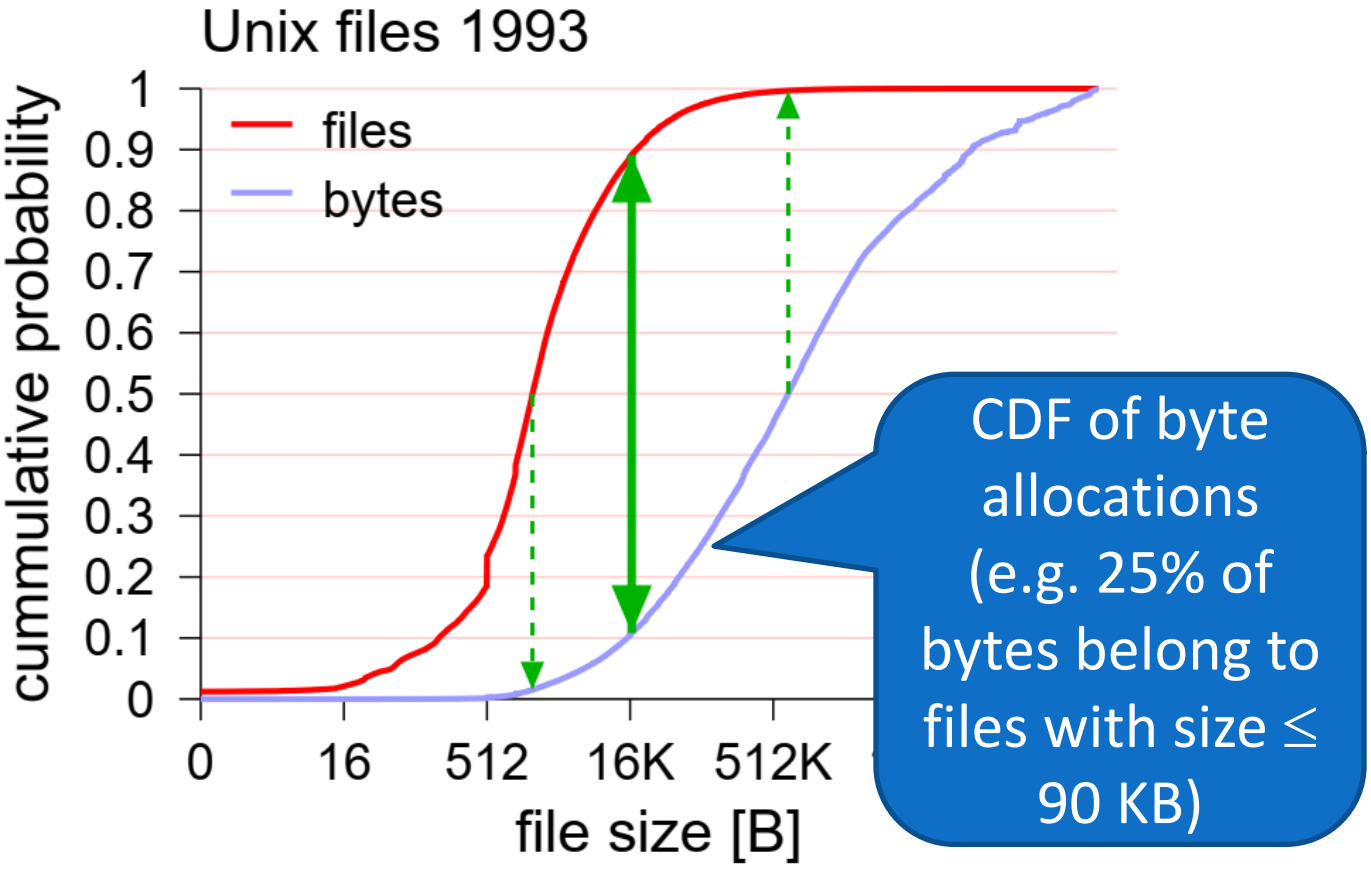
Support for Small Files

- Unix inode optimized for small files
  - While able to support (very) large files



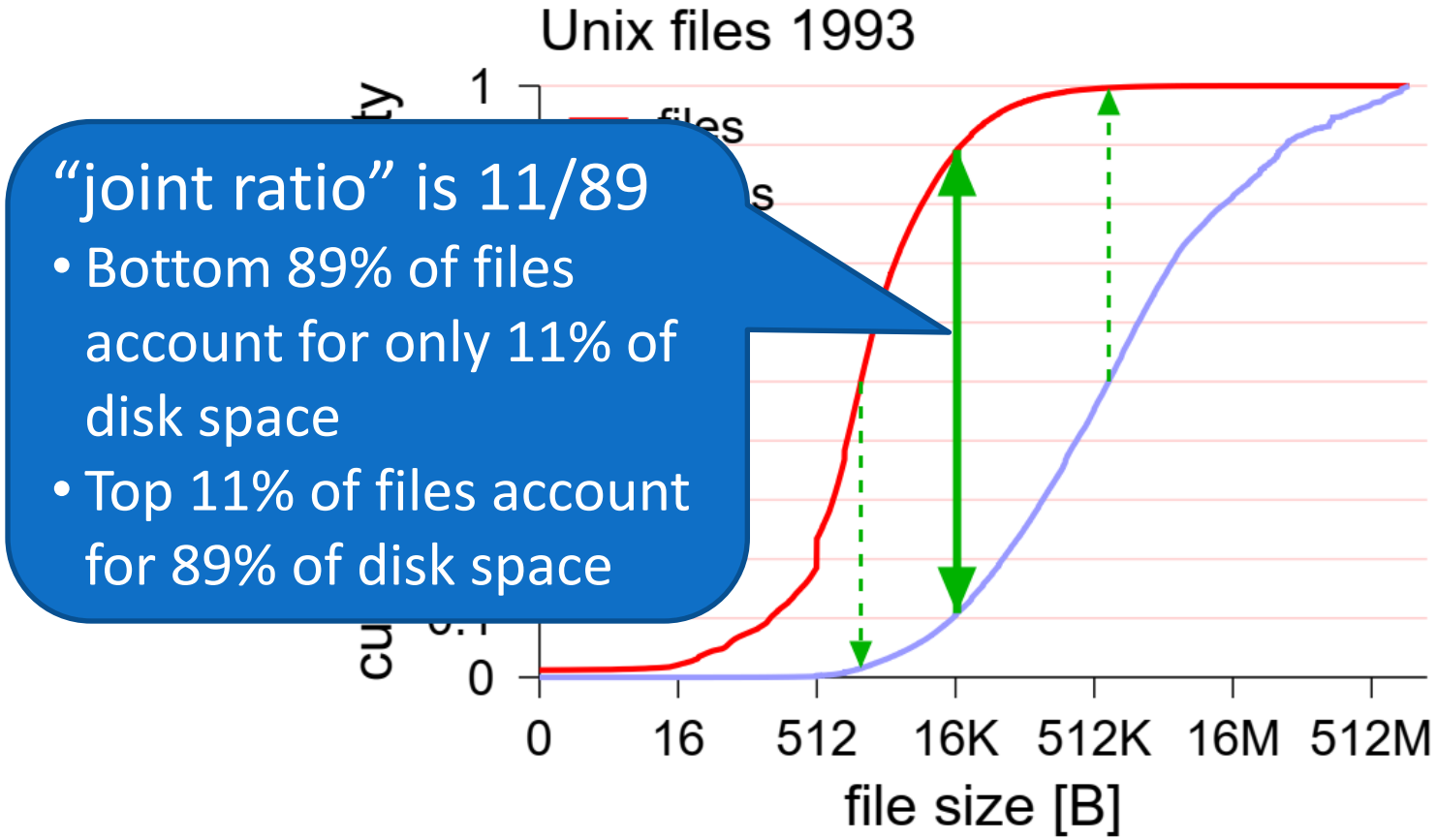
Support for Small Files

- Unix inode optimized for small files
  - While able to support (very) large files



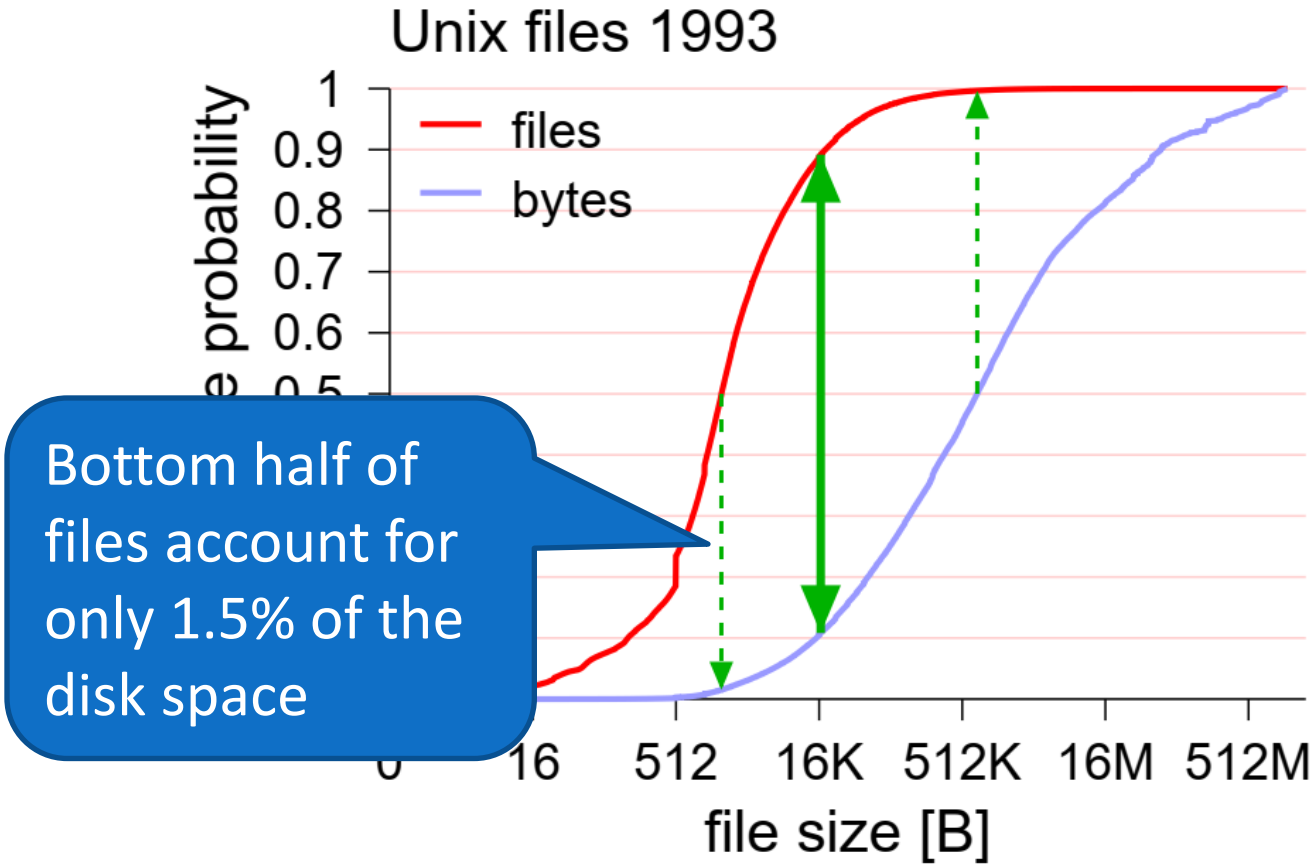
Support for Small Files

- Unix inode optimized for small files
  - While able to support (very) large files



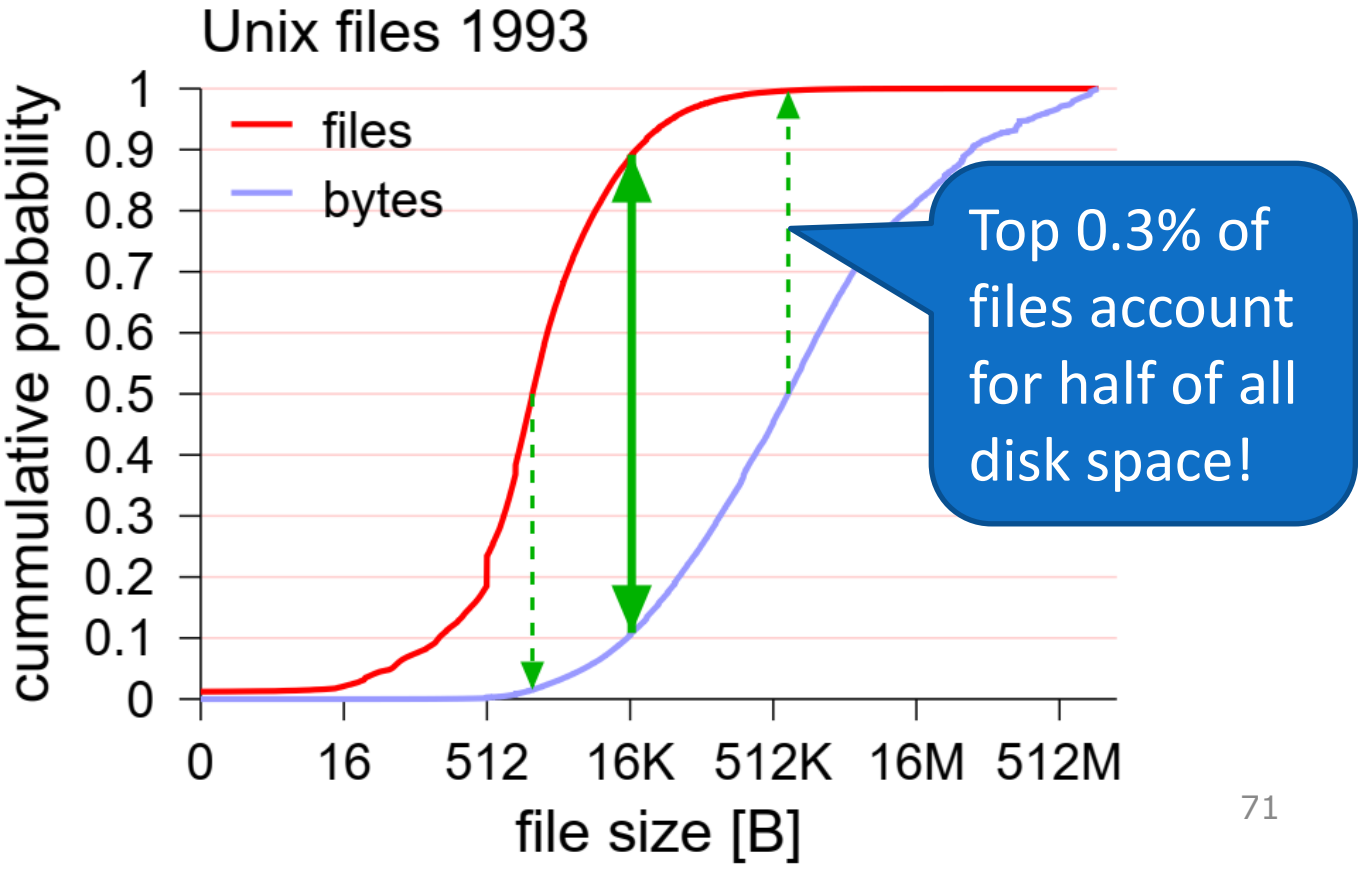
Support for Small Files

- Unix inode optimized for small files
  - While able to support (very) large files



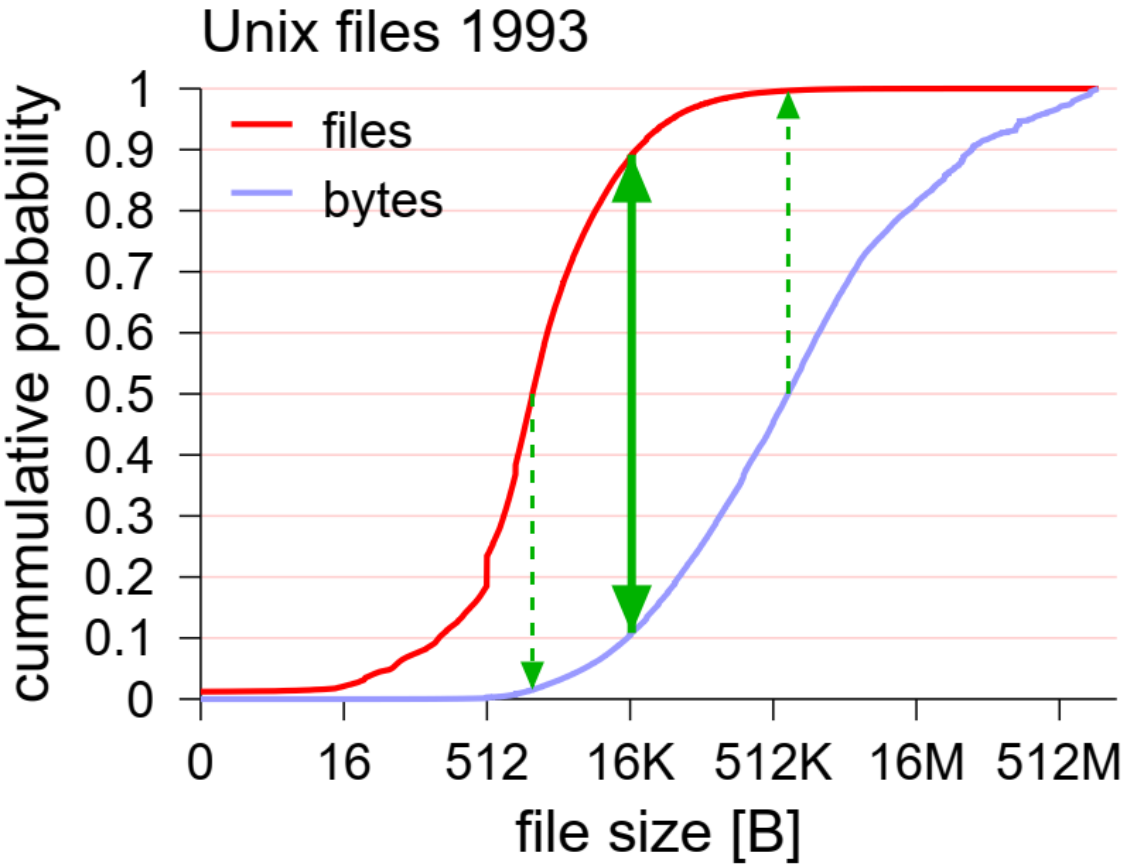
Support for Small Files

- Unix inode optimized for small files
  - While able to support (very) large files



Support for Small Files

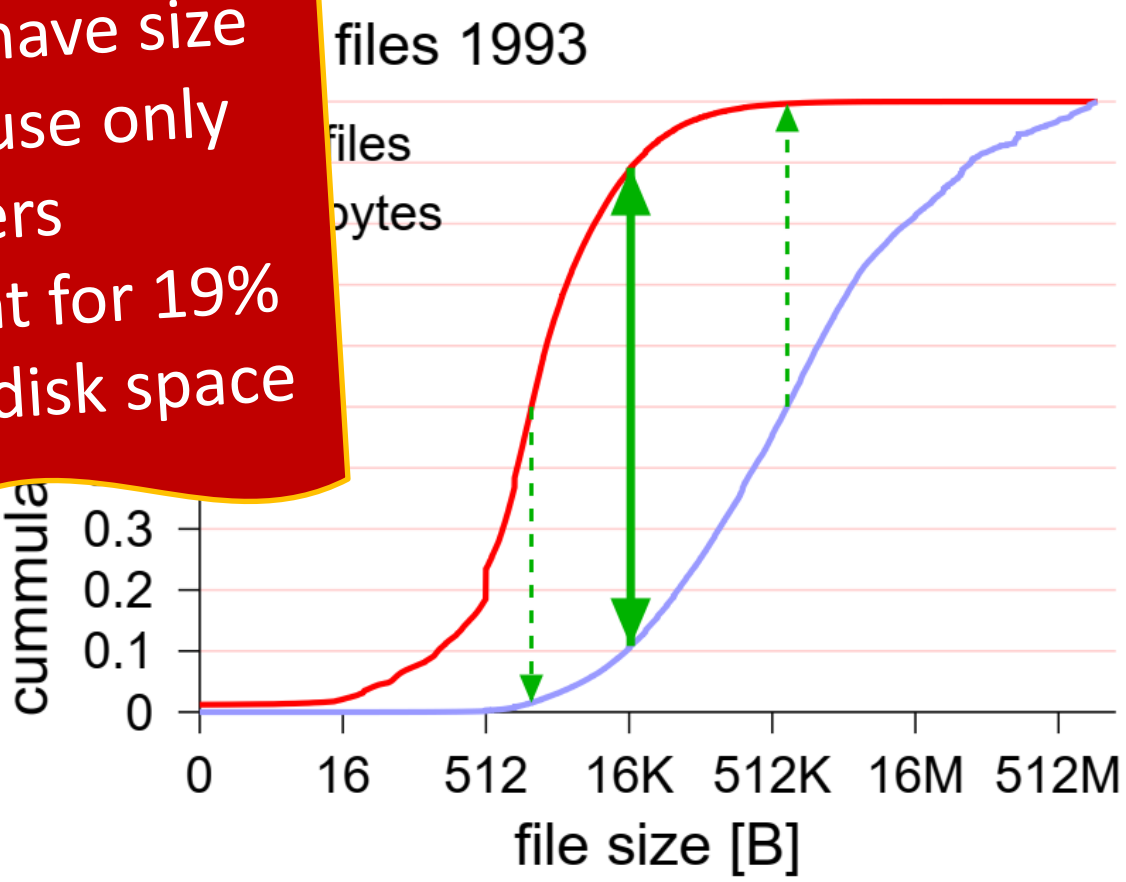
- Unix inode optimized for small files
  - While able to support (very) large files



Support for Small Files

- Unix inode optimized for small files
  - While able to support (very) large files

- 95% of files have size ≤ 48KB and use only direct pointers
- They account for 19% of the total disk space



Inlining Optimizations

- Inode includes 60 bytes (15 pointers of 4 bytes) for data access
- In special cases we can use them directly for the data itself!

Inlining Optimizations

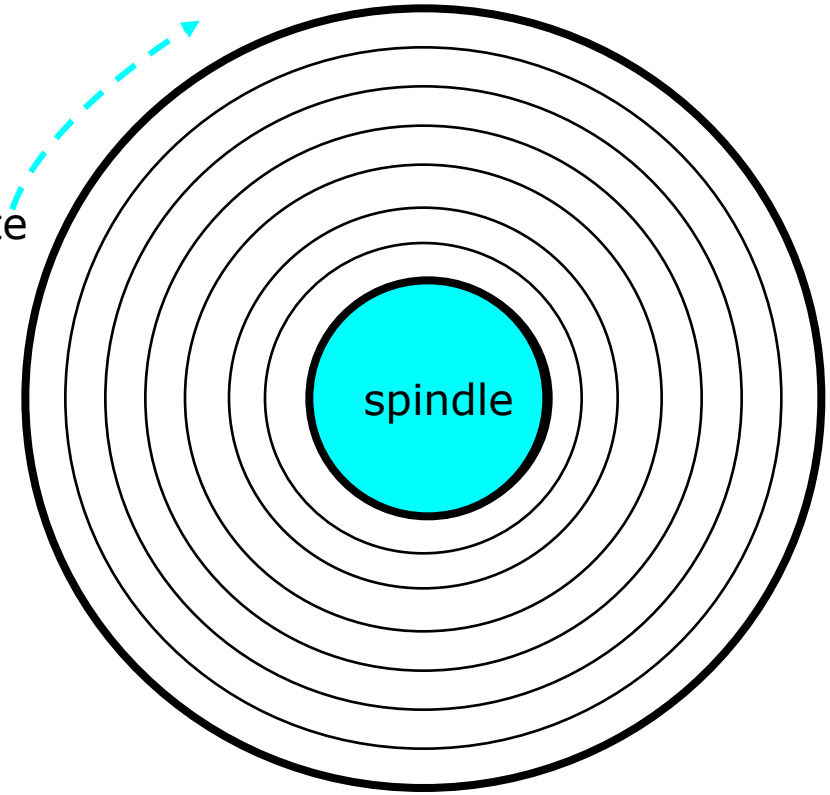
- Inode includes 60 bytes (15 pointers of 4 bytes) for data access
- In special cases we can use them directly for the data itself!
- Symbolic link to file with path of up to 60 characters

Inlining Optimizations

- Inode includes 60 bytes (15 pointers of 4 bytes) for data access
- In special cases we can use them directly for the data itself!
- Symbolic link to file with path of up to 60 characters
- Actual data of file that is up to 60 bytes long
  - Indicated by setting the “inlined” flag
  - Useful for 6% of the files

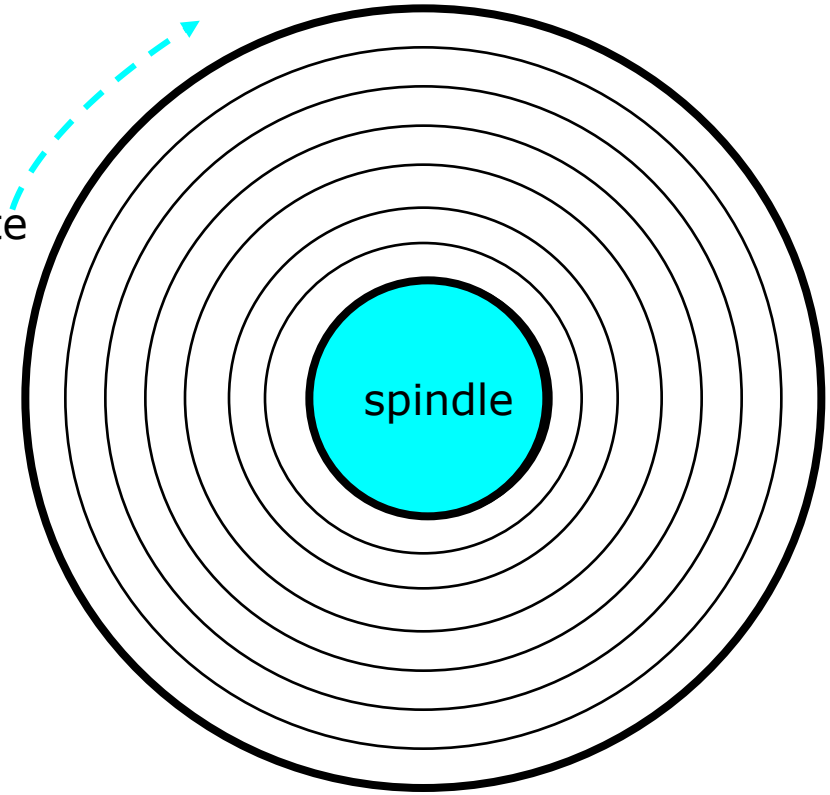
Disk Operation and Layout Optimization

The disk surface spins at a fixed rotational rate



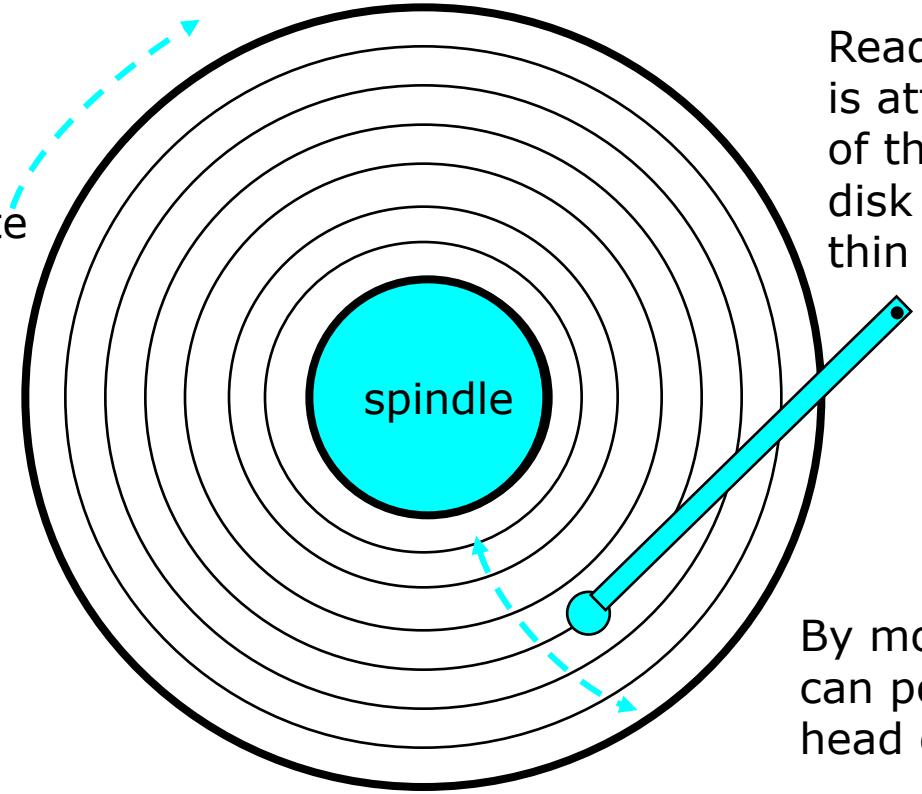
Disk Operation and Layout Optimization

The disk surface spins at a fixed rotational rate



Disk Operation and Layout Optimization

The disk surface spins at a fixed rotational rate

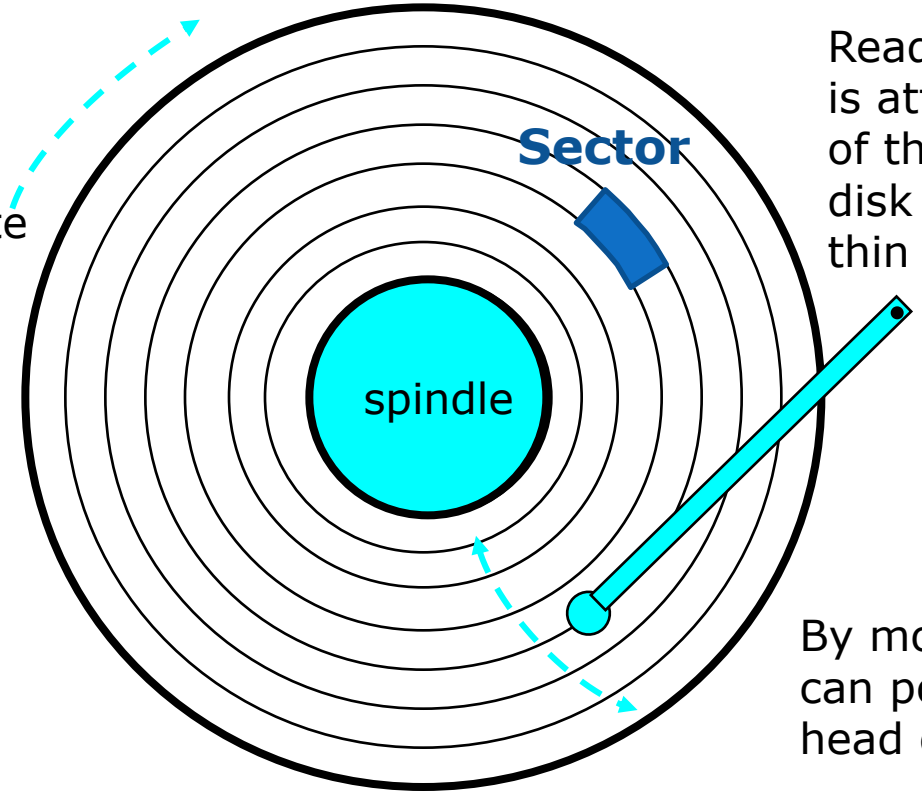


Read/write *head* is attached to end of the *arm* and flies over disk surface on thin cushion of air

By moving radially, arm can position read/write head over any track

Disk Operation and Layout Optimization

The disk surface spins at a fixed rotational rate



Read/write *head* is attached to end of the *arm* and flies over disk surface on thin cushion of air

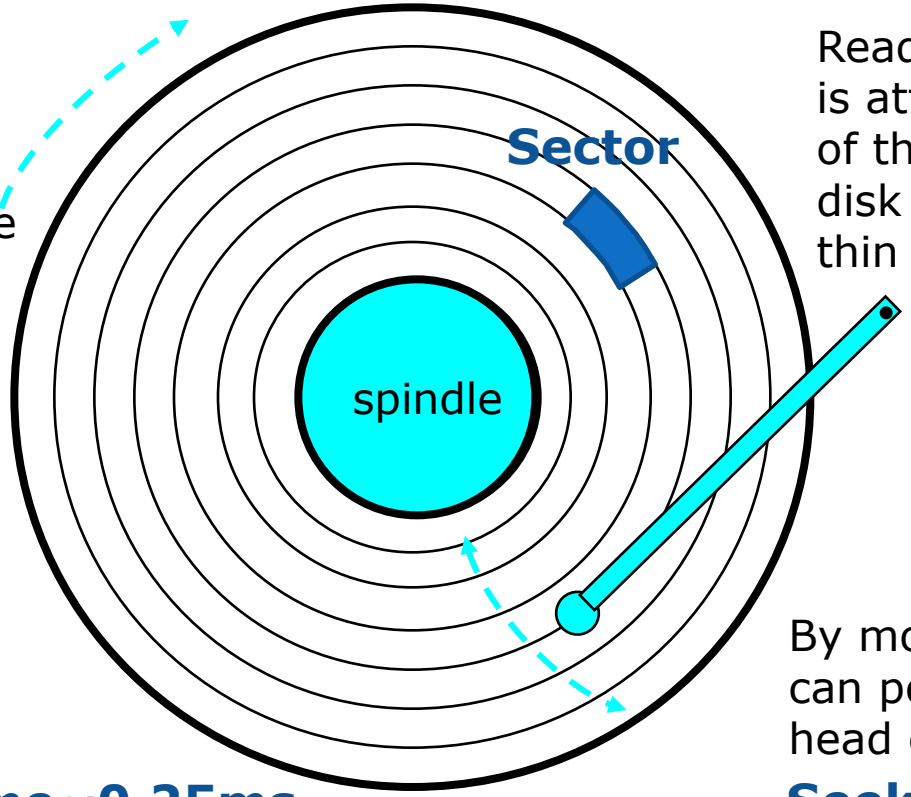
By moving radially, arm can position read/write head over any track

Disk Operation and Layout Optimization

The disk surface spins at a fixed rotational rate

7200 RPM

Rotational time~4ms



Read/write *head* is attached to end of the *arm* and flies over disk surface on thin cushion of air

By moving radially, arm can position read/write head over any track

Seek time~8-12ms

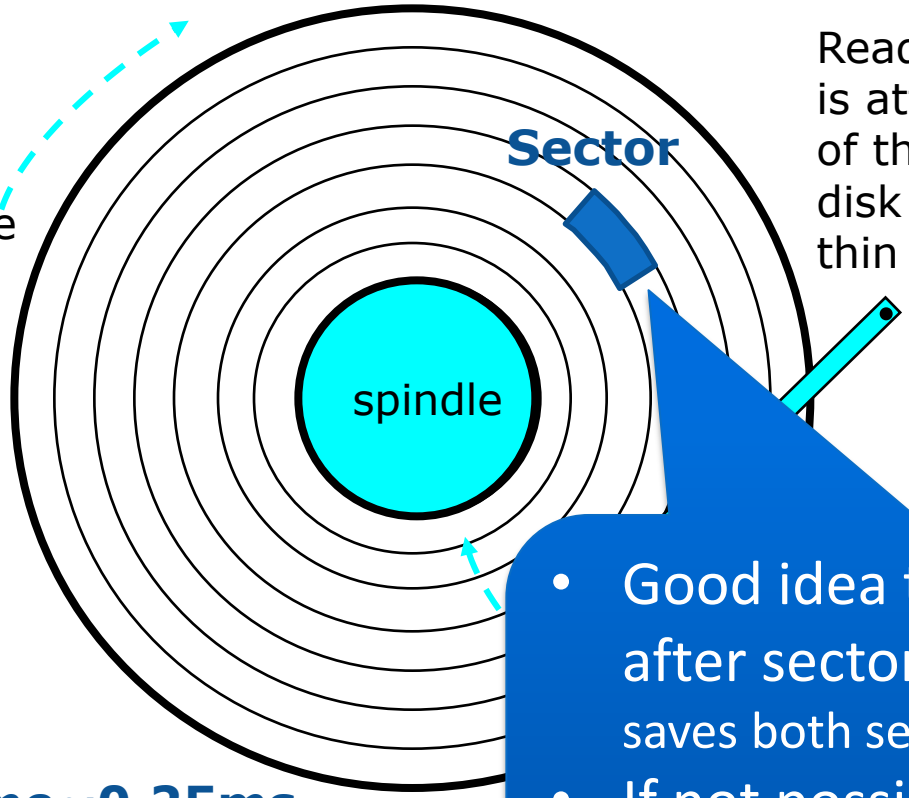
Transfer Time~0.25ms

Disk Operation and Layout Optimization

The disk surface spins at a fixed rotational rate

7200 RPM

Rotational time~4ms

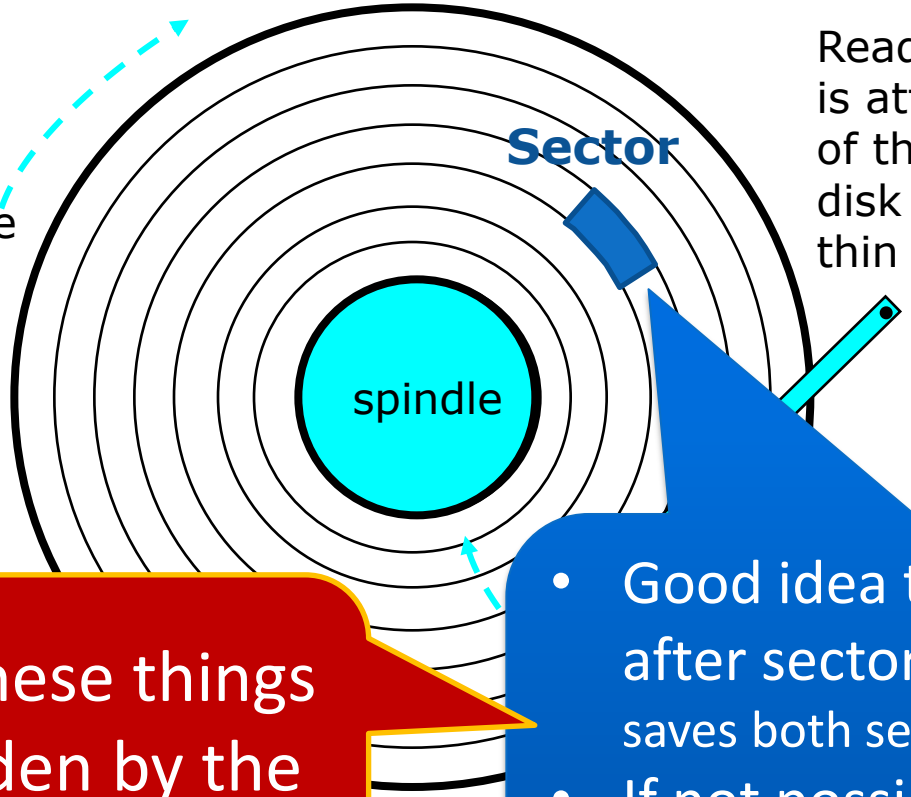


Read/write *head* is attached to end of the *arm* and flies over disk surface on thin cushion of air

Transfer Time~0.25ms

- Good idea to read/write sector after sector!
  - saves both seek and rotational time
- If not possible, stay on the same track
  - saves seek time only

Disk Operation and Layout Optimization



Today these things are hidden by the disk controller…

- Good idea to read/write sector after sector!
  saves both seek and rotational time
- If not possible, stay on the same track
  saves seek time only

Disk Scheduling

- If there are several disk I/O's waiting to be executed, what is the best way to execute them?

Disk Scheduling

- If there are several disk I/O's waiting to be executed, what is the best way to execute them?
  - **Goal: to minimize seek time!**

Disk Scheduling

- If there are several disk I/O's waiting to be executed, what is the best way to execute them?
  - **Goal: to minimize seek time!**
- FIFO: simple, but doesn't optimize seek time

Disk Scheduling

- If there are several disk I/O's waiting to be executed, what is the best way to execute them?
  - **Goal: to minimize seek time!**
- FIFO: simple, but doesn't optimize seek time
- Shortest seek time first (SSTF)
  - Choose the next request that is close as possible to the previous one.
  - Good in minimizing seeks, but can cause starvation for some requests.

Disk Scheduling

- If there are several disk I/O's waiting to be executed, what is the best way to execute them?
  - **Goal: to minimize seek time!**
- FIFO: simple, but doesn't optimize seek time
- Shortest seek time first (SSTF)
  - Choose the next request that is close as possible to the previous one.
  - Good in minimizing seeks, but can cause starvation for some requests.
- Scan ("elevator algorithm")
  - Same as SSTF except head is moving only in one direction.
  - Two flavors:
    1. In the disk edge, switch direction and start again.
    2. In the disk edge, choose the farthest request, and start again.

Disk Scheduling

- If there are several disk I/O's waiting to be executed, what is the best way to execute them?
  - **Goal: to minimize seek time!**
- FIFO: simple, but doesn't optimize seek time
- Shortest seek time first (SST
  - Choose the next request that i
    one.
  - Good in minimizing seeks,
    requests.
- Scan ("elevator algorithm
  - Same as SSTF except head is movi tion.
  - Two flavors:
    1. In the disk edge, switch direction and start again.
    2. In the disk edge, choose the farthest request, and start again.

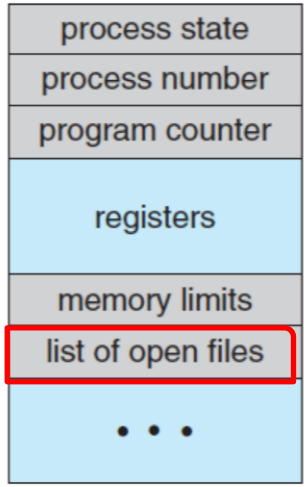
Again, in modern disks this is hidden by the disk controller

Open Files of a Process

Open Files of a Process

Process Control Block (PCB)

How the process is represented in the OS



- Context
- Additional OS information needed:
  - Memory management
  - Accounting information
  - I/O status information

23

Lecture 3

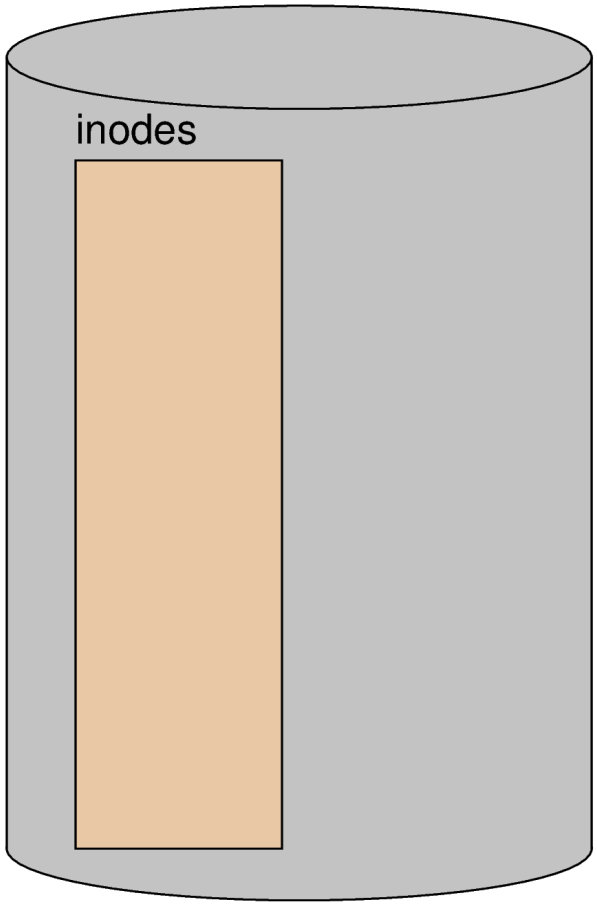
Open Files

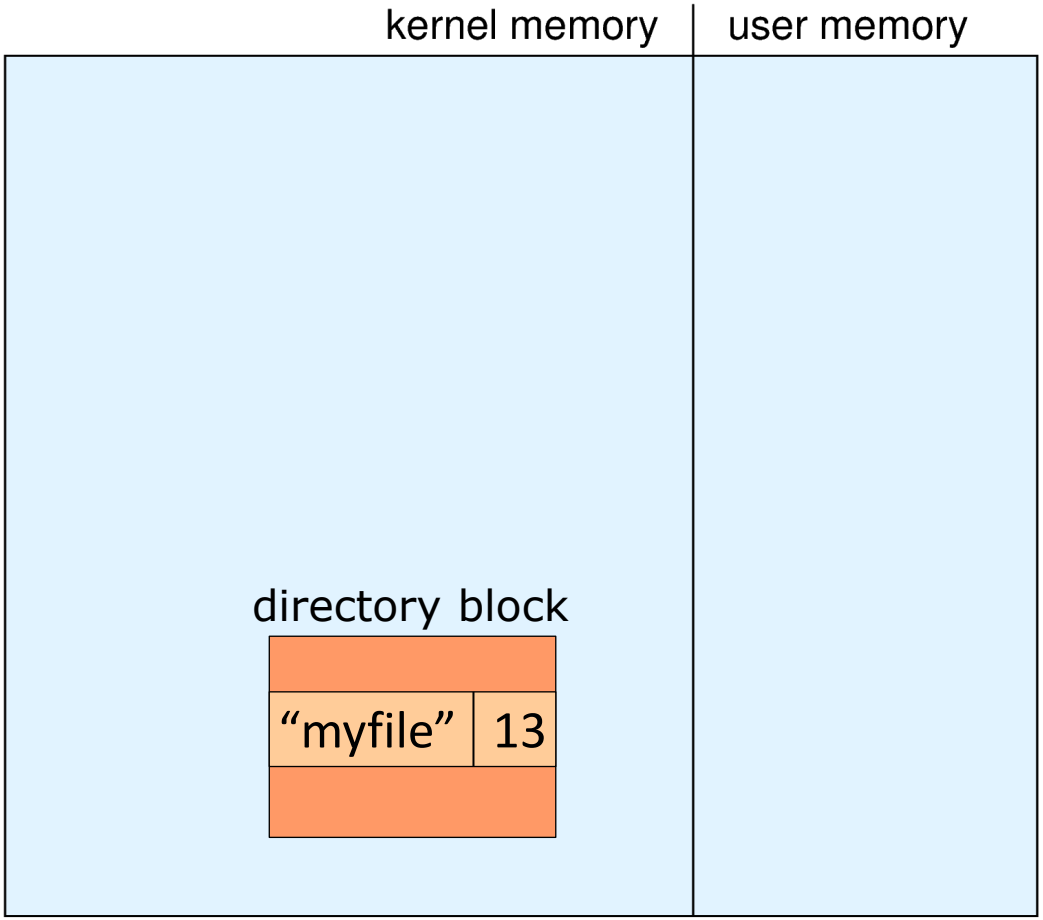
- Most FS require an open() system call before using a file
- OS keeps an in-memory table of open files, so when reading or writing is requested, they refer to entries in this table
- On finished with a file, a close() system call is necessary (creating & deleting files typically works on closed files)
- What data is kept in the open files table?
- What happens when multiple users (processes) open the same file at the same time?

Remember Opening a File

fd = open("myfile");

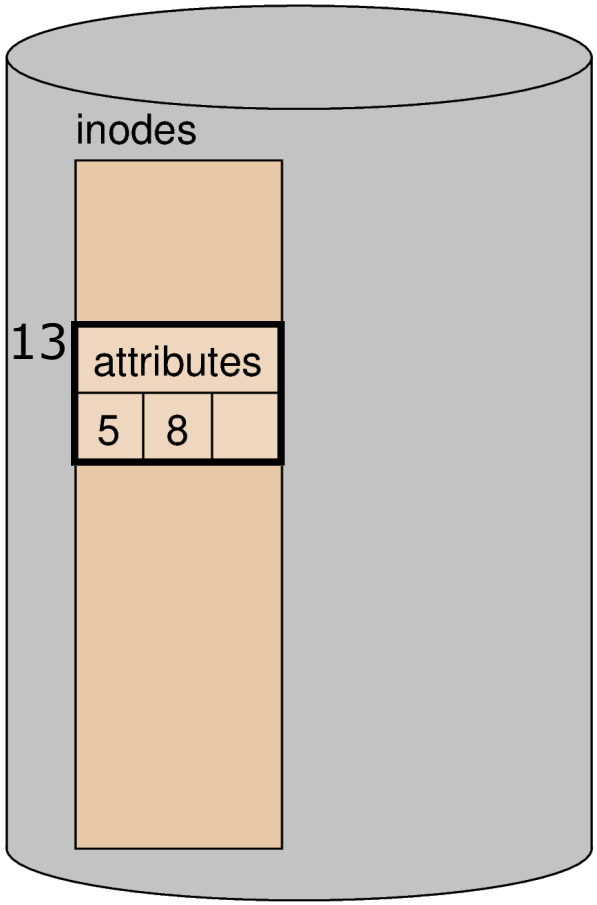
// "myfile" is inode 13

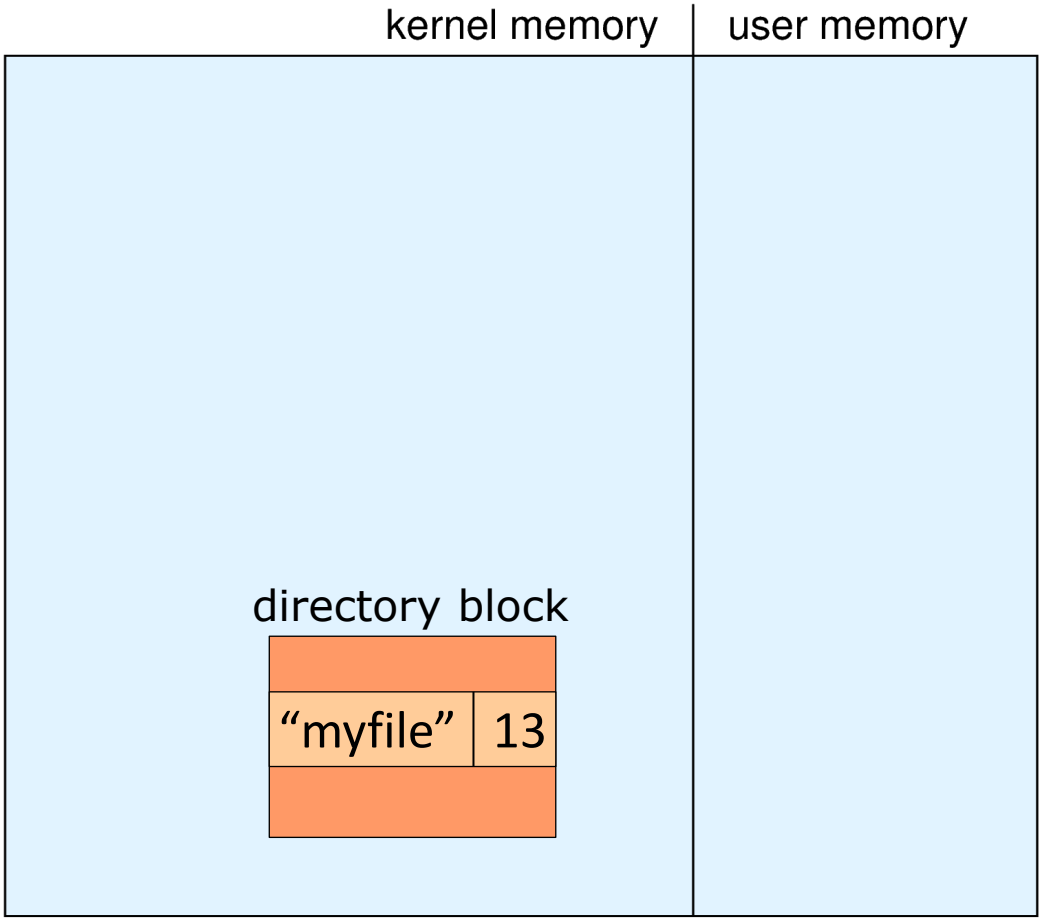




Remember Opening a File

fd = open("myfile"); // "myfile" is inode 13

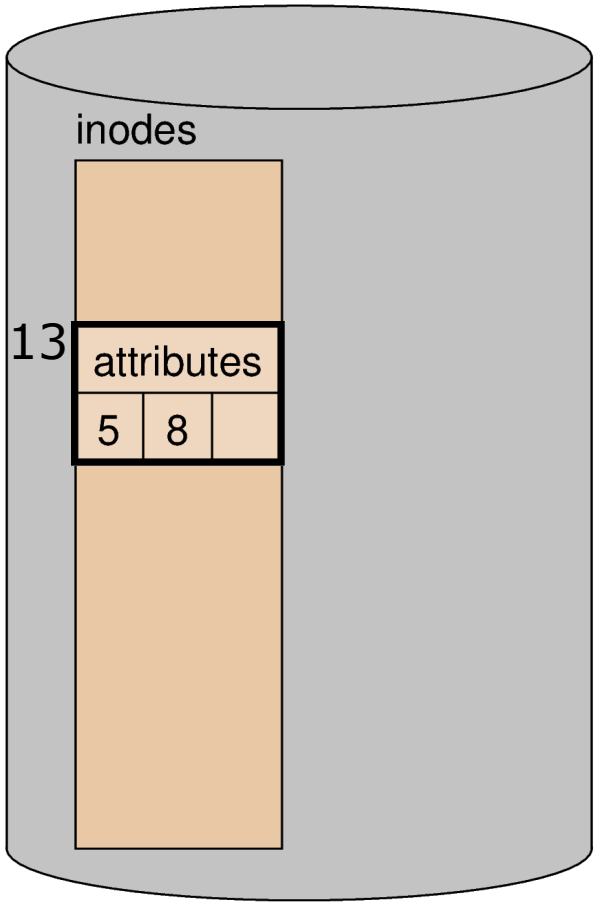


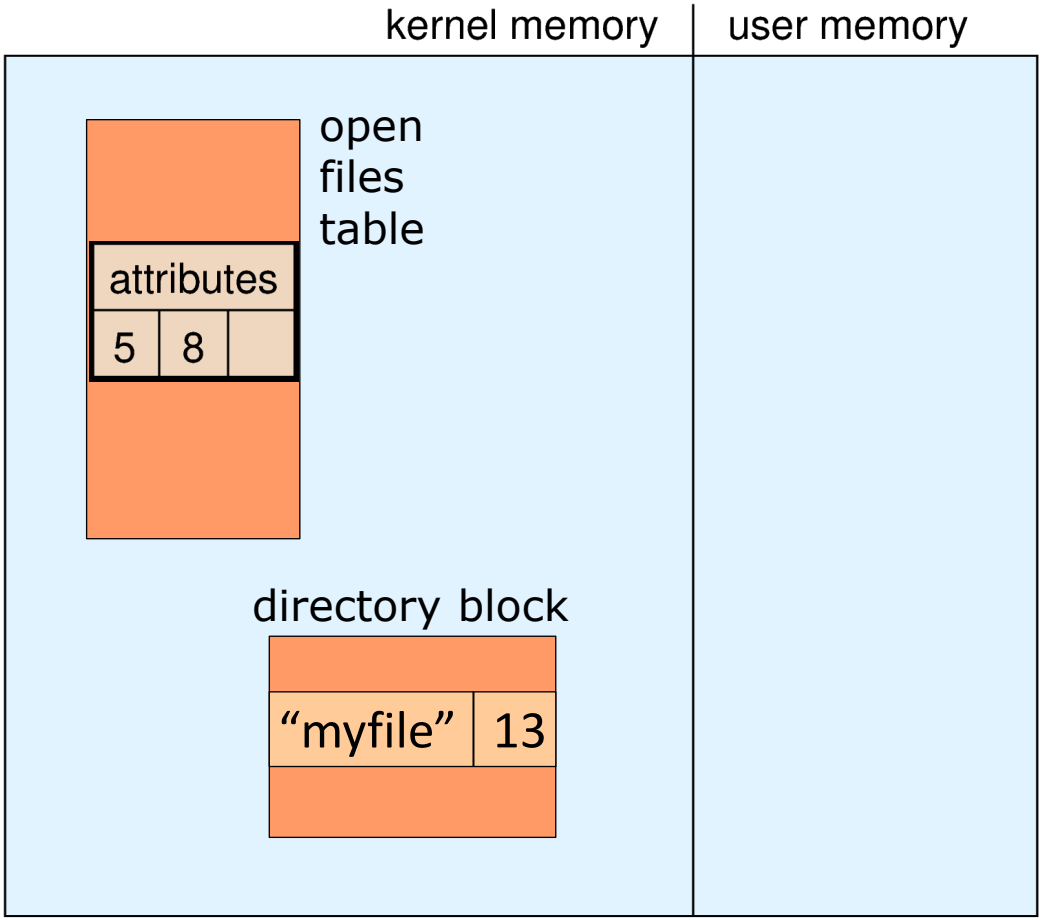


Remember Opening a File

fd = open("myfile");

// "myfile" is inode 13

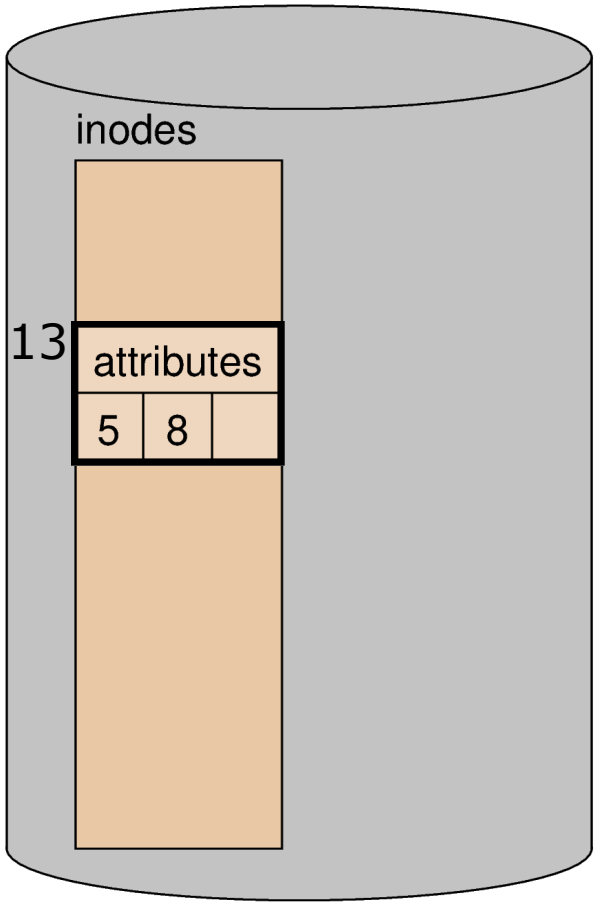


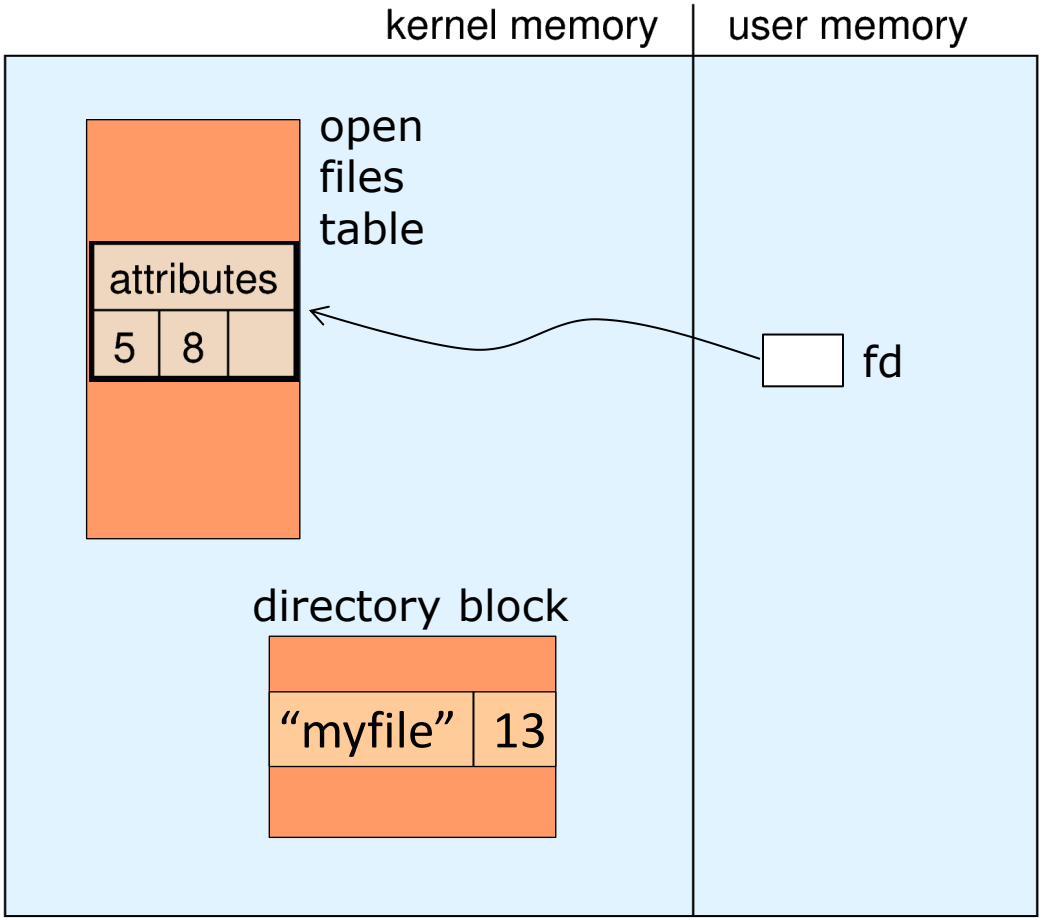


Remember Opening a File

fd = open("myfile");

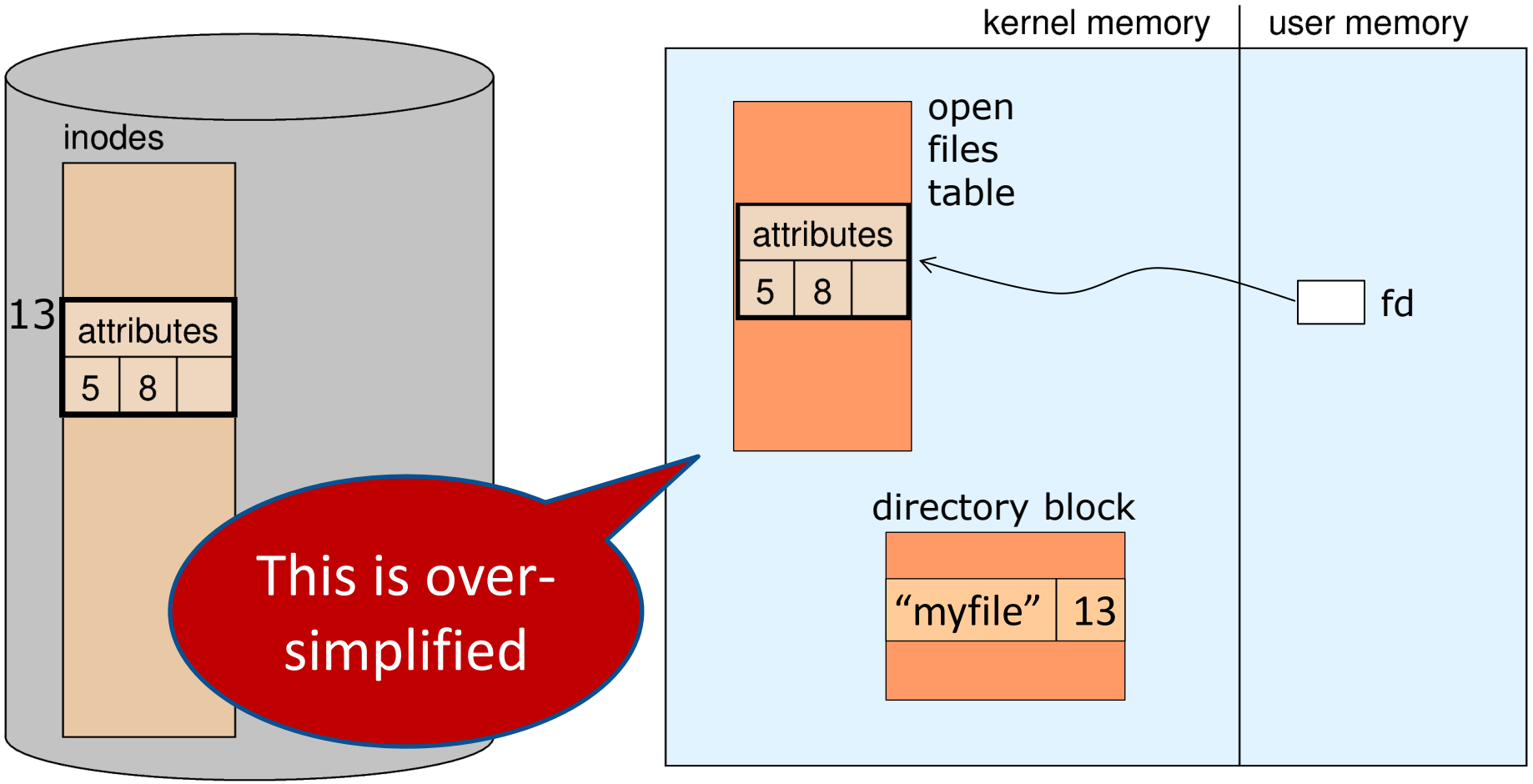
// "myfile" is inode 13





Remember Opening a File

fd = open("myfile"); // "myfile" is inode 13



Multiple Users of a File

OS typically keeps three levels of internal tables:

- i-node table (of open files) – only one per file
  - Location of file on disk
  - Access dates
  - File size
  - Count of how many processes have the file open (used for deletion)
- System wide table
  - Entry for each open of the file
  - Contains the offset of the process within the file
- Per-process table (within the PCB)
  - Lists the open files of a specific user (maps file descriptors to files)
- Some OSes store the per-process and system-wide tables together

Multiple Users of a File

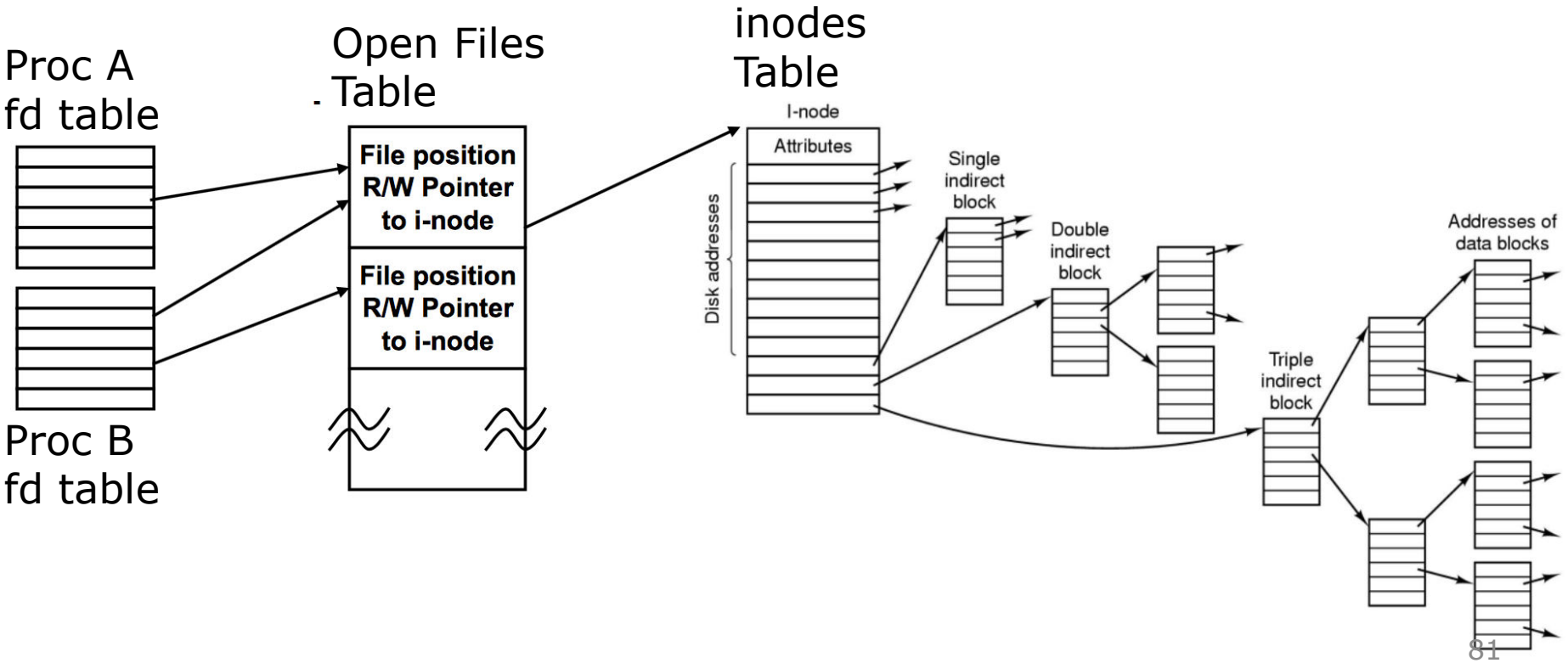
OS typically keeps three levels of internal tables:

- i-node table (of open files) – only one per file
  - Location of file on disk
  - Access dates
  - File size
  - Count of how many processe
    deletion)
- System wide table
  - Entry for each open of th
  - Contains the offset of the pr
- Per-process table (within the PCB)
  - Lists the open files of a specific user (maps file descriptors to files)
- Some OSes store the per-process and system-wide tables together

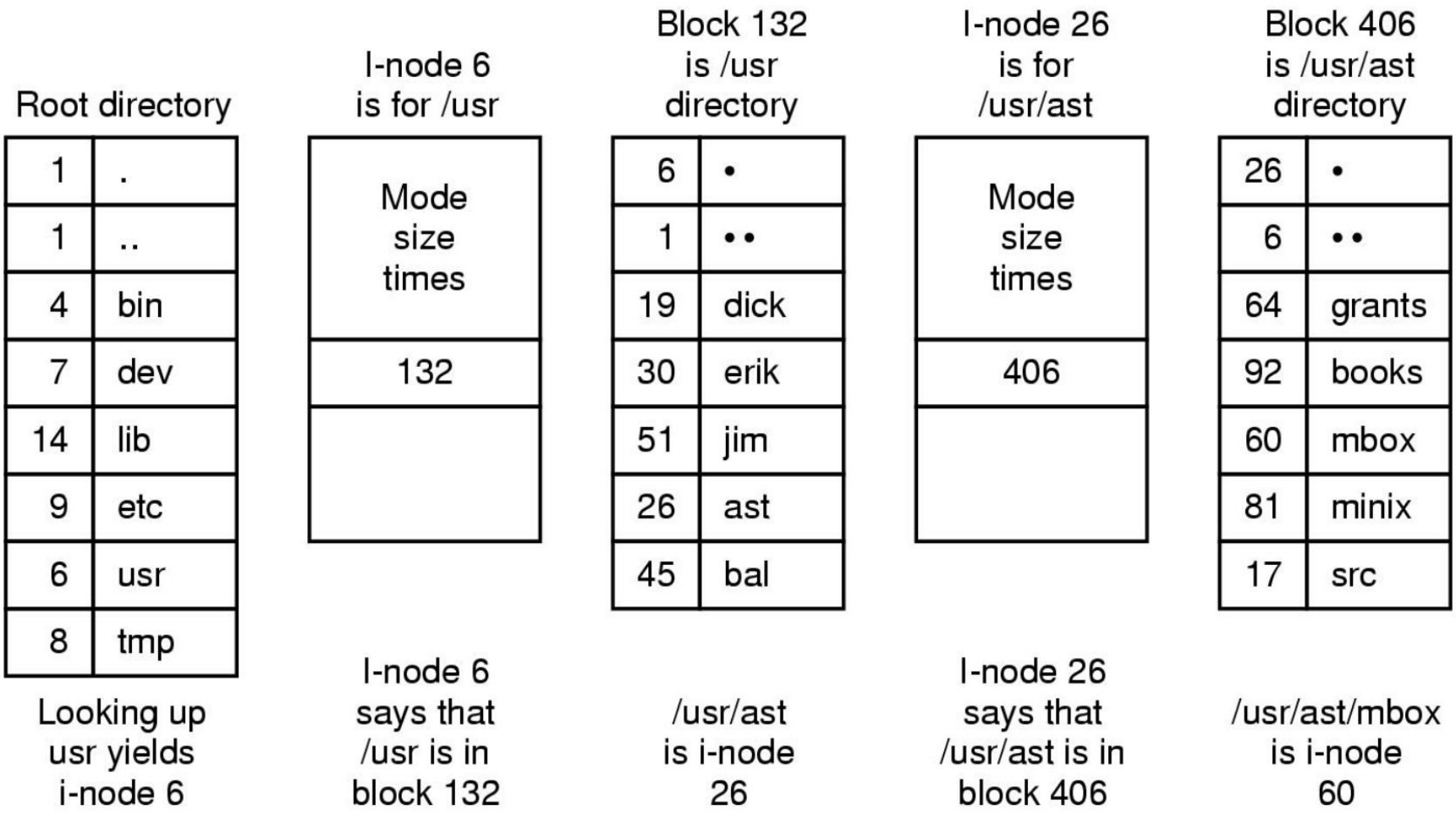
- The indices of this table are called file descriptors (UNIX) or file handles
- Some are reserved: STDIN, STDOUT, STDERR

More detailed View (Unix/Linux)

Having 3 tables supports various sharing patterns of file access

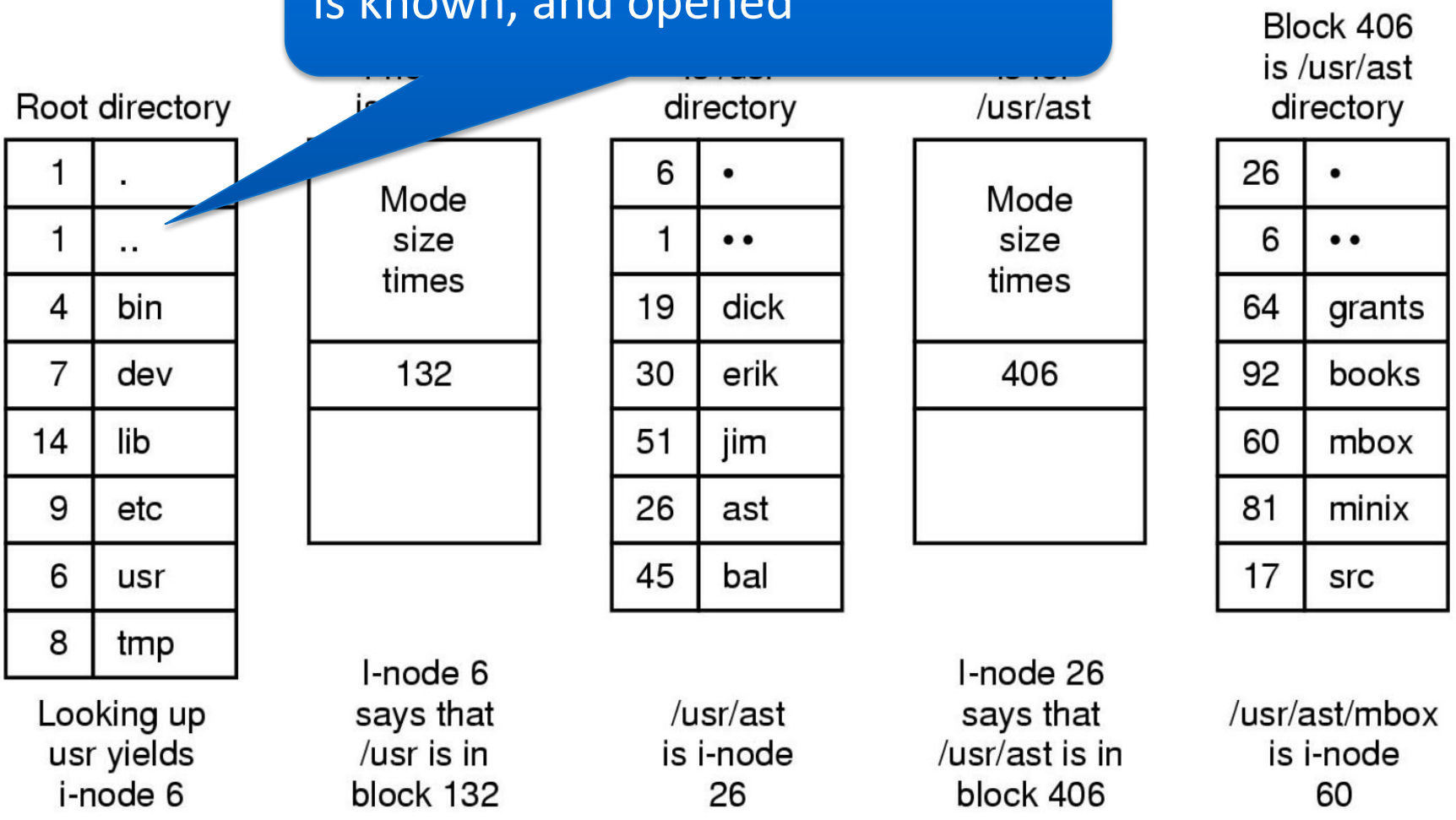


Example: Accessing byte number 1M in **/usr/ast/mbox** (Unix V7):

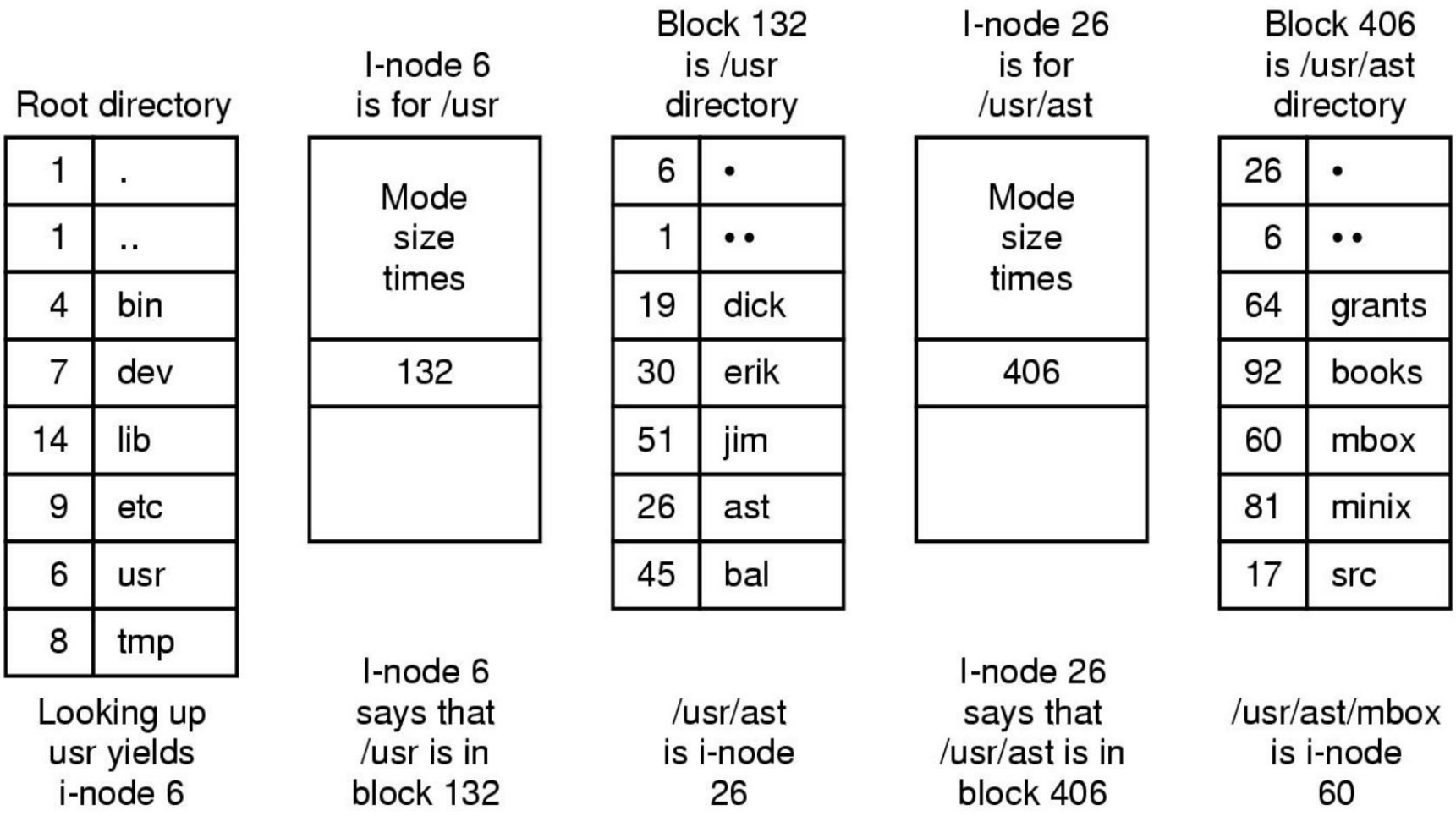


Example: Accessing byte number 1M in /usr/ast/mbox (Unix V7):

Root directory position in the disk is known, and opened

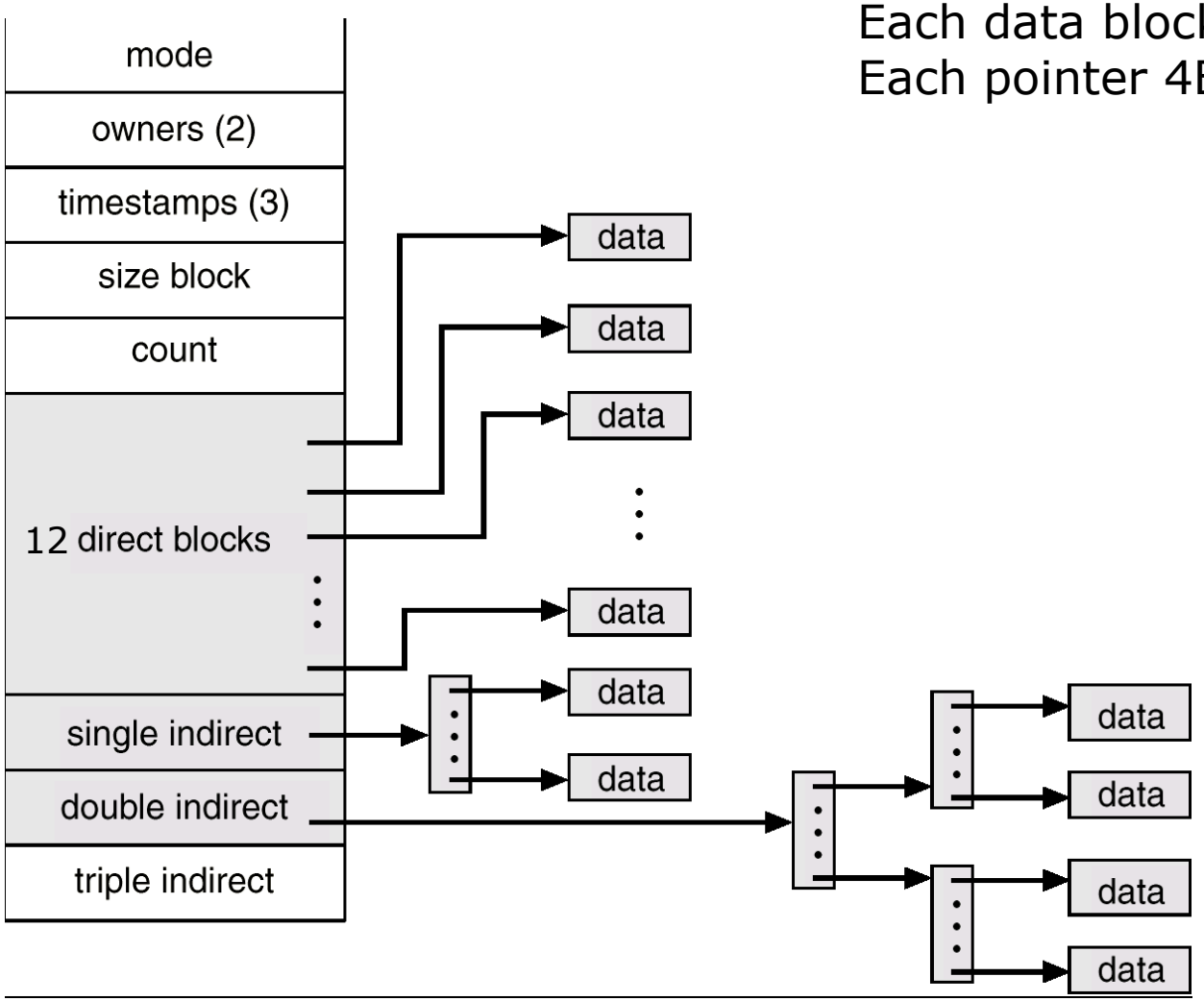


Example: Accessing byte number 1M in **/usr/ast/mbox** (Unix V7):



inodes (Unix File System)

i-node: 128B

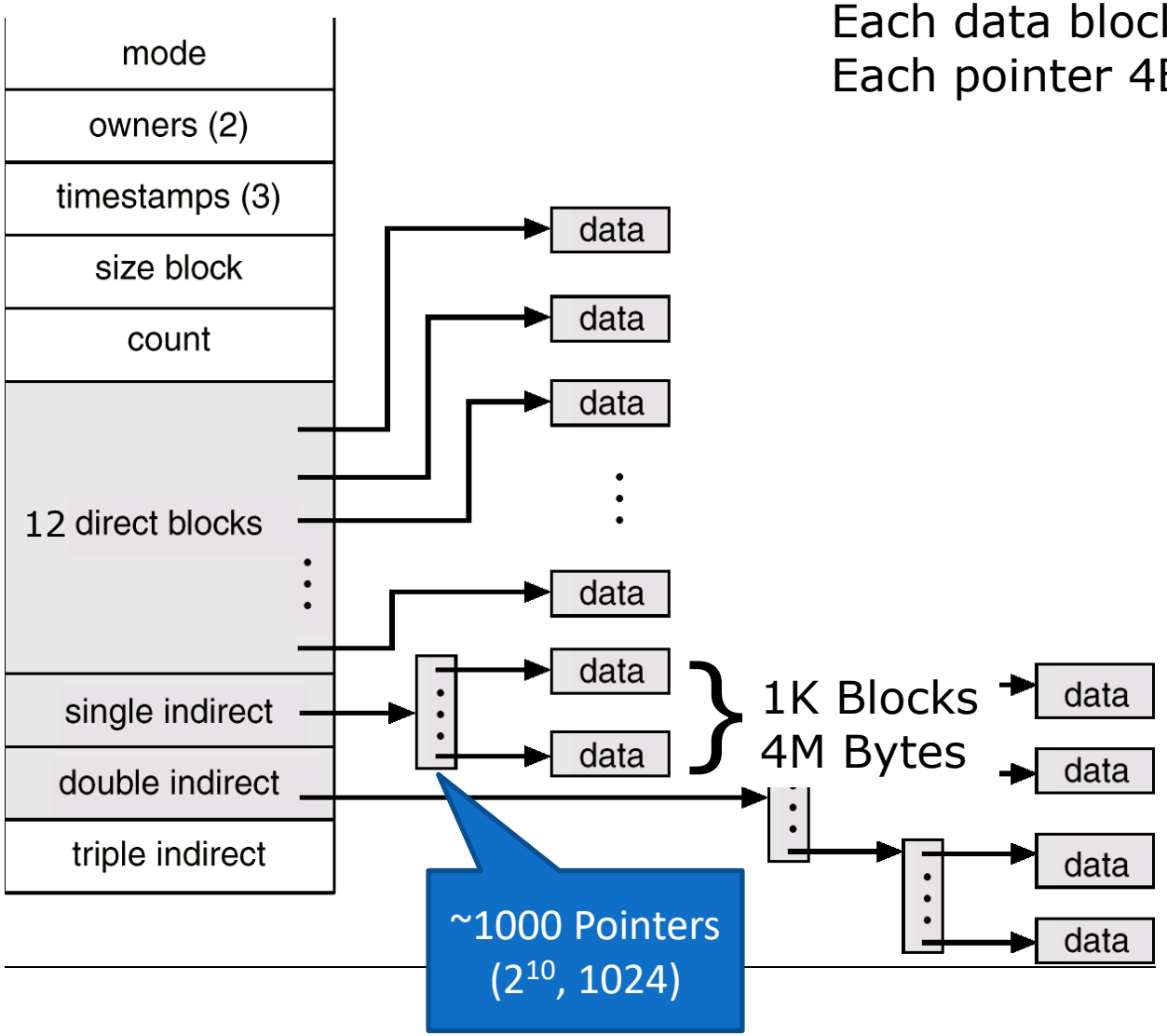


Each data block is 4KB
Each pointer 4B

inodes (Unix File System)

i-node: 128B

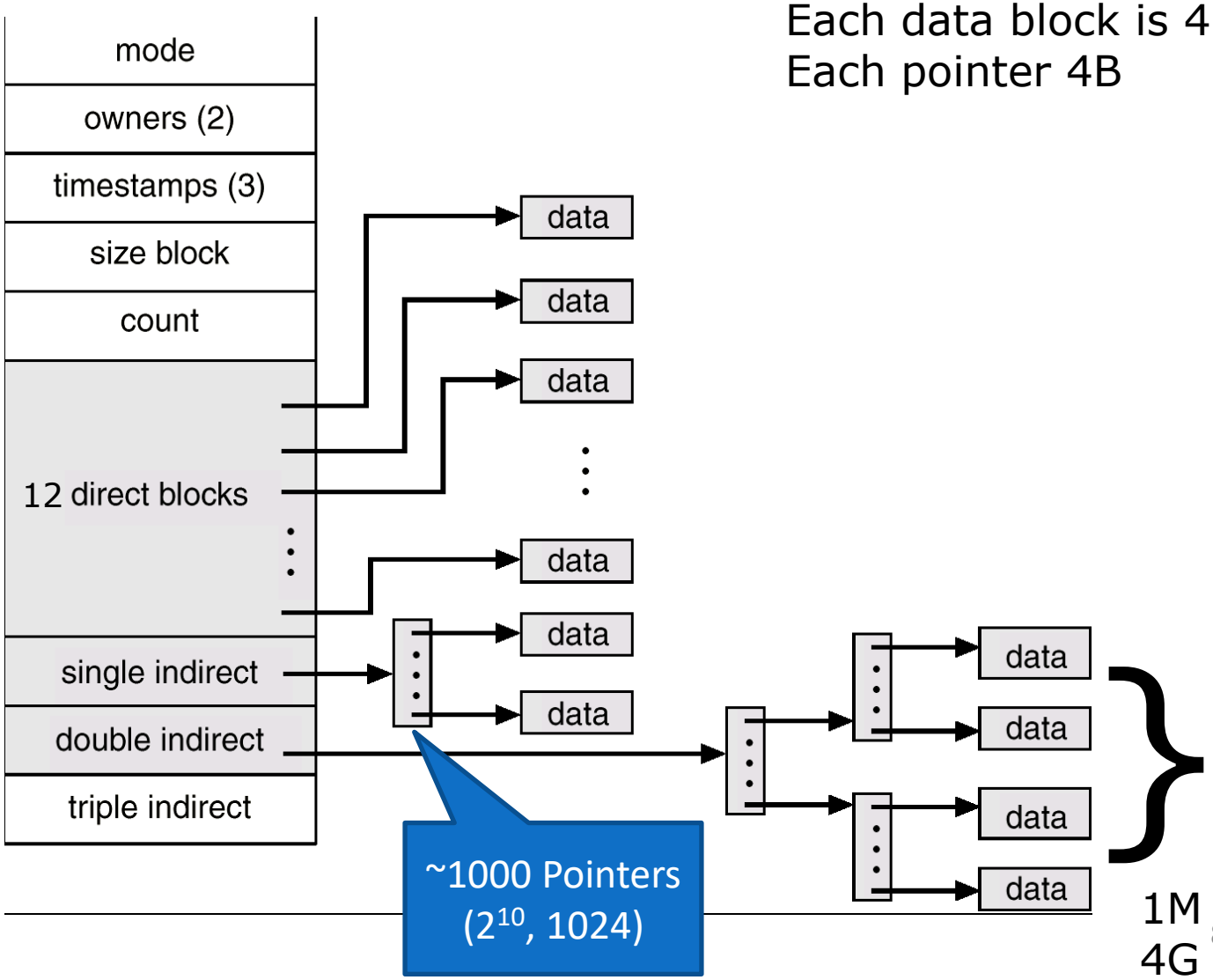
Each data block is 4KB
Each pointer 4B



inodes (Unix File System)

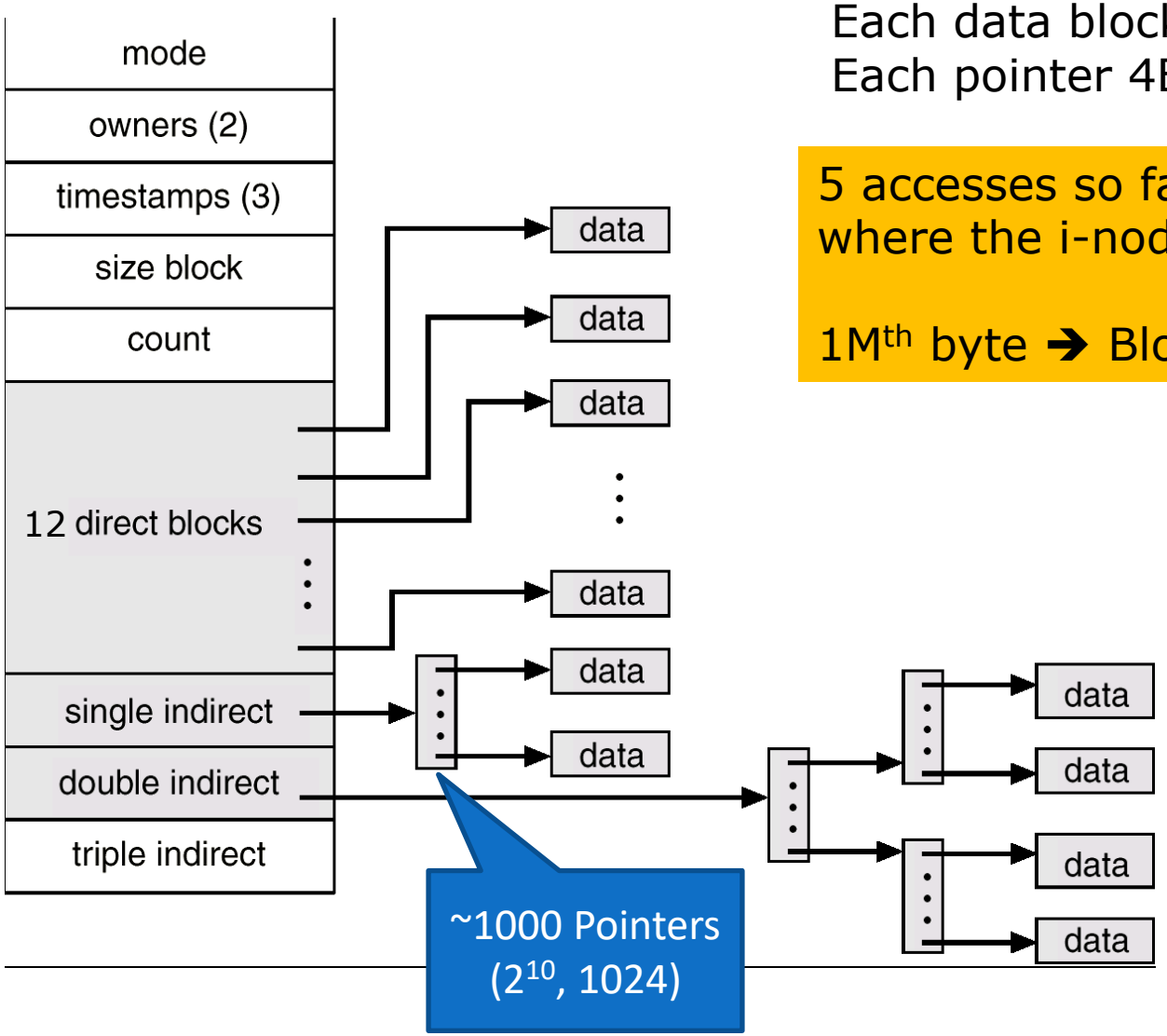
i-node: 128B

Each data block is 4KB
Each pointer 4B



inodes (Unix File System)

i-node: 128B



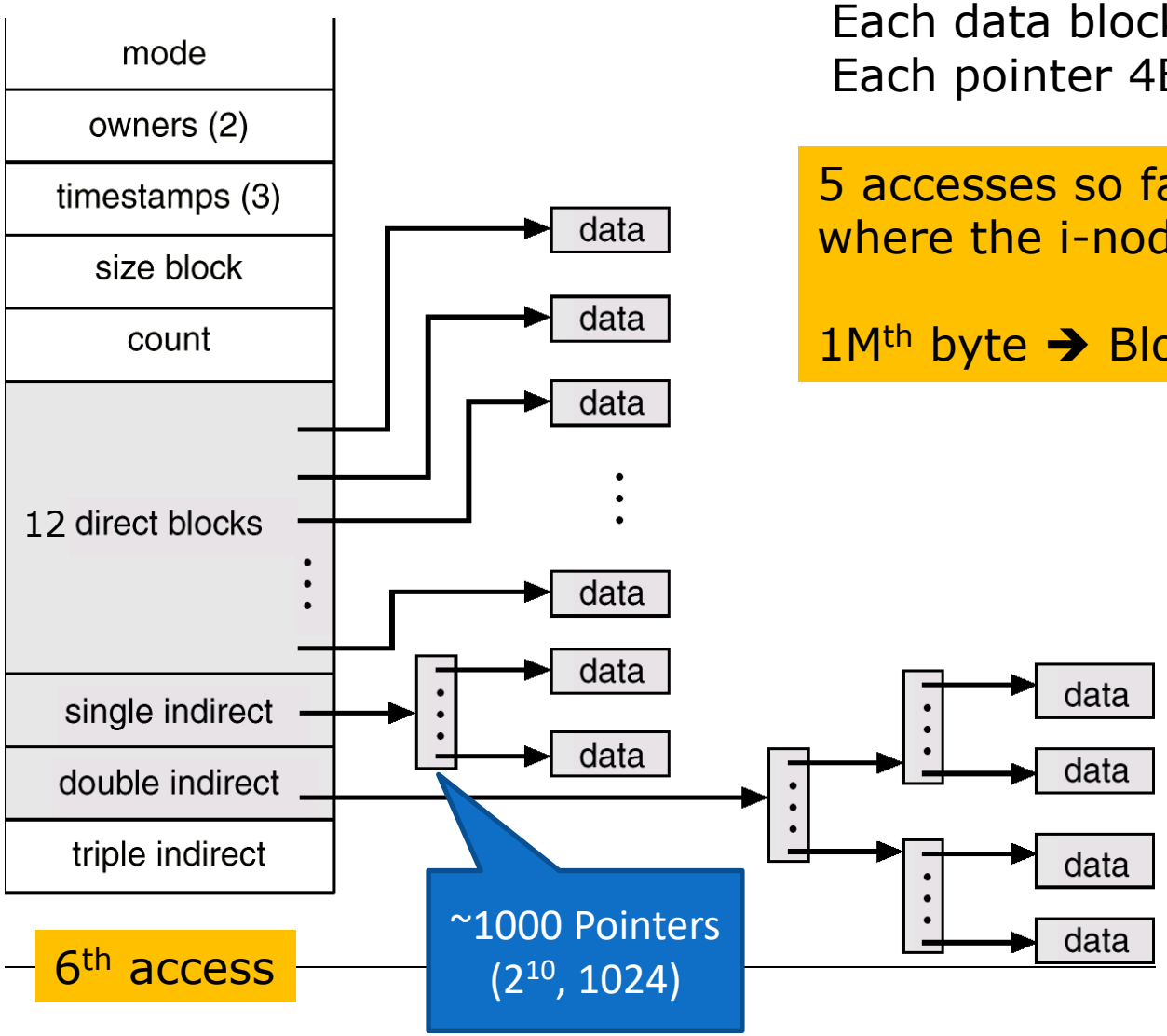
Each data block is 4KB
Each pointer 4B

5 accesses so far, to know where the i-node is.

1Mth byte ➔ Block 250

inodes (Unix File System)

i-node: 128B



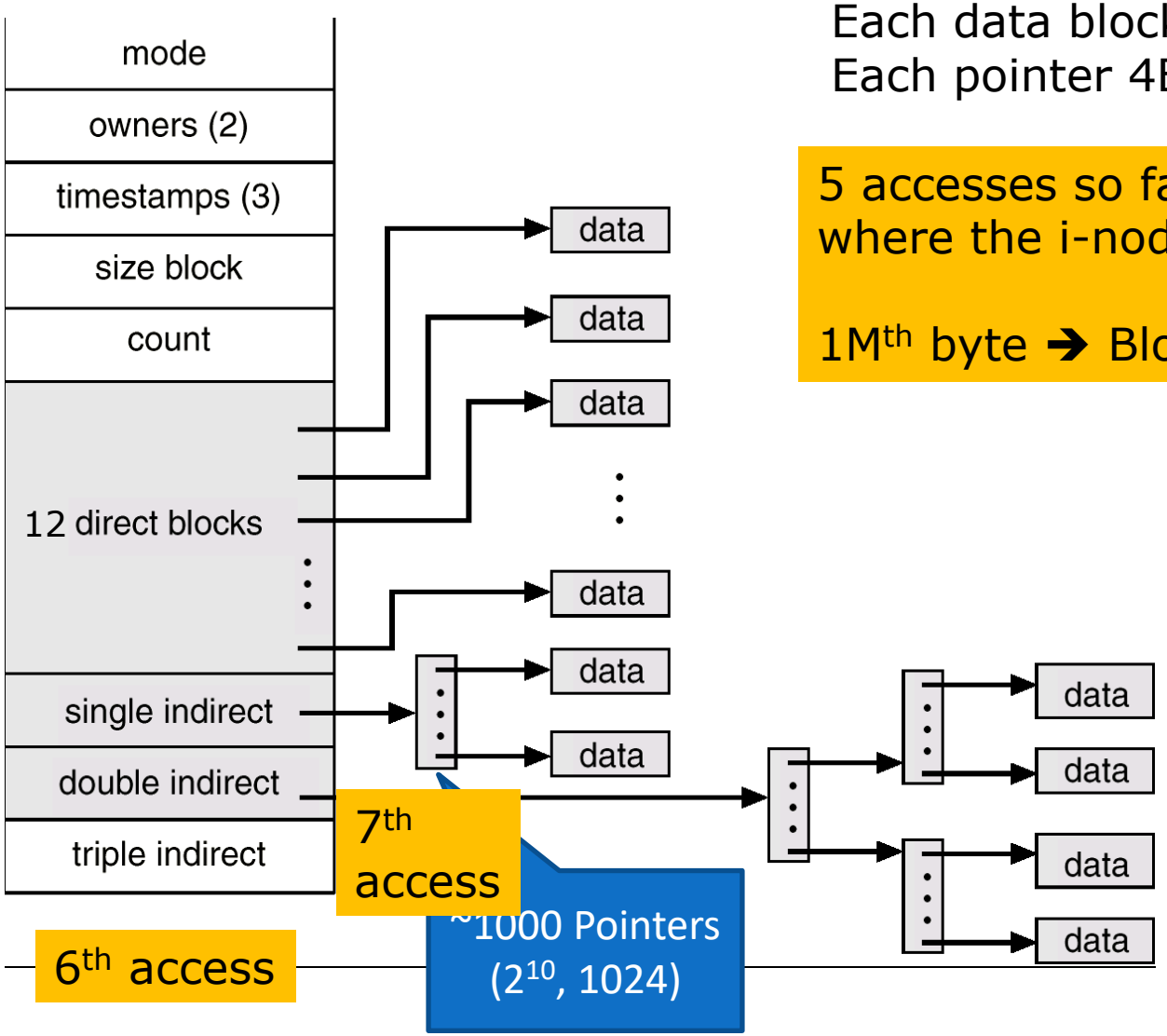
Each data block is 4KB
Each pointer 4B

5 accesses so far, to know where the i-node is.

1Mth byte ➔ Block 250

inodes (Unix File System)

i-node: 128B



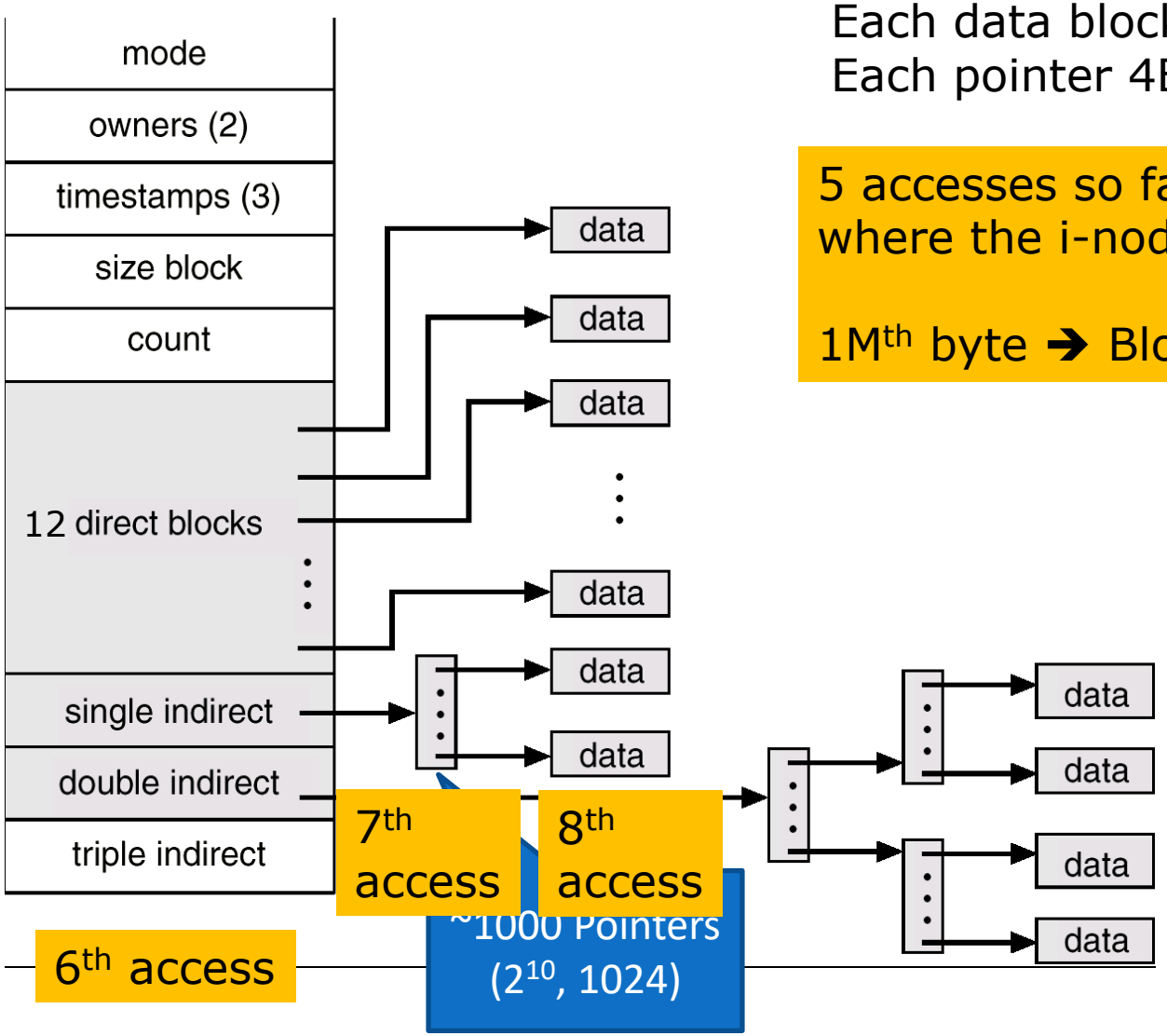
Each data block is 4KB
Each pointer 4B

5 accesses so far, to know where the i-node is.

1Mth byte ➔ Block 250

inodes (Unix File System)

i-node: 128B



Each data block is 4KB
Each pointer 4B

5 accesses so far, to know where the i-node is.

1Mth byte ➔ Block 250