Operating System Concepts

Lecture 29: File System Abstraction

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- What is a file system?
  - file abstraction
- File attributes and operations
- Access methods

What's a file system?

- file system is comprised of a collection of files, a directory structure, an index structure, and a free space list
  - directory structure maps the file name to the low-level file number
  - index structure maps the low-level file number to disk blocks

What’s a file system?

- file system is comprised of a collection of files, a directory structure, an index structure, and a free space list
  - directory structure maps the file name to the low-level file number
  - index structure maps the low-level file number to disk blocks
- file system is responsible for
  - managing and organizing files with directories
    - reading the file’s entry from directory

What's a file system?

- file system is comprised of a collection of files, a directory structure, an index structure, and a free space list
  - directory structure maps the file name to the low-level file number
  - index structure maps the low-level file number to disk blocks
- file system is responsible for
  - managing and organizing files with directories
    - reading the file's entry from directory
  - interacting with the lower-level I/O subsystem used to access the file data
    - disk I/O is done in blocks; reading or writing less than the block size needs translation and buffering

What’s a file system?

- file system is comprised of a collection of files, a directory structure, an index structure, and a free space list
  - directory structure maps the file name to the low-level file number
  - index structure maps the low-level file number to disk blocks
- file system is responsible for
  - managing and organizing files with directories
    - reading the file’s entry from directory
  - interacting with the lower-level I/O subsystem used to access the file data
    - disk I/O is done in blocks; reading or writing less than the block size needs translation and buffering
  - providing a more convenient API (compared to hardware interface) with
    - naming: finding file by name rather than block number
    - sharing: enabling shared access to files
    - protection: enforcing access restriction
    - reliability: keeping file intact despite power cycles, crashes, hardware failures, etc.
    - efficient access: maximizing sequential access and allowing efficient random access

File systems and disks

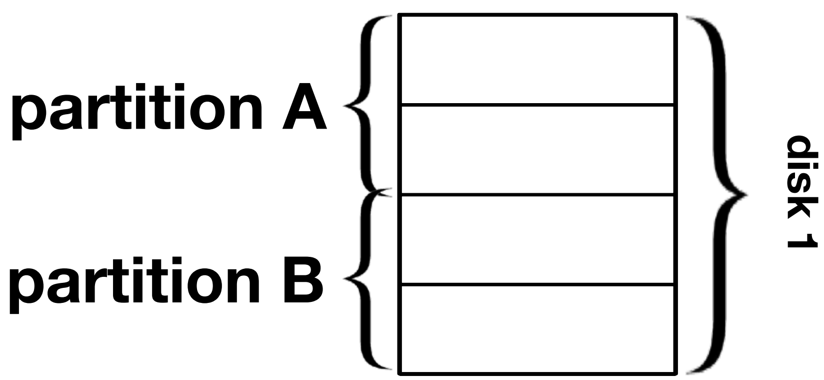
- disk is an array of blocks, where each block is a fixed-size data array
  - a disk that contains a file system is known as a **volume**
  - in Windows, a volume is specified by a letter followed by a colon (C: or D:)

File systems and disks

- disk is an array of blocks, where each block is a fixed-size data array
  - a disk that contains a file system is known as a **volume**
  - in Windows, a volume is specified by a letter followed by a colon (C: or D:)
- disk can be used in its entirety for a file system
  - the volume is the whole device

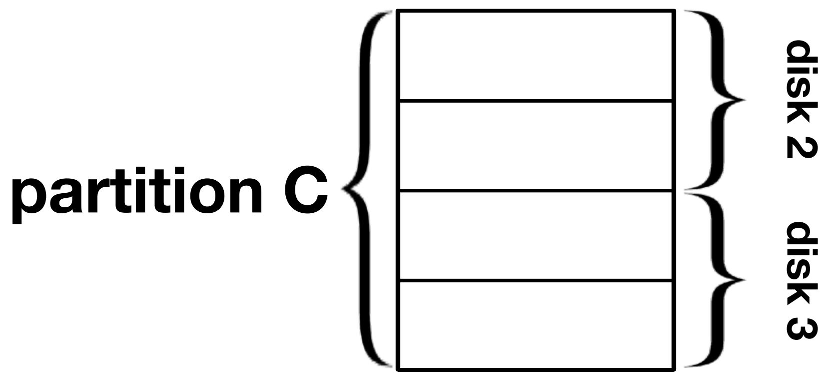
File systems and disks

- disk is an array of blocks, where each block is a fixed-size data array
  - a disk that contains a file system is known as a **volume**
  - in Windows, a volume is specified by a letter followed by a colon (C: or D:)
- disk can be used in its entirety for a file system
  - the volume is the whole device
- disk can be partitioned into quarters, each holding a separate file system
  - the volume is a subset of a device or the device is sliced up into several volumes
  - reasons to partition a disk
    - to limit the size of each file system
    - to have multiple file system types on a single disk
    - to leave an unformatted disk space for raw read/write operations



File systems and disks

- disk is an array of blocks, where each block is a fixed-size data array
  - a disk that contains a file system is known as a **volume**
  - in Windows, a volume is specified by a letter followed by a colon (C: or D:)
- disk can be used in its entirety for a file system
  - the volume is the whole device
- disk can be partitioned into quarters, each holding a separate file system
  - the volume is a subset of a device or the device is sliced up into several volumes
  - reasons to partition a disk
    - to limit the size of each file system
    - to have multiple file system types on a single disk
    - to leave an unformatted disk space for raw read/write operations
- multiple disks can be linked together to form a RAID set
  - the volume comprises multiple devices



What do hardware and OS provide?

- Hardware
  - persistence: disks provide **nonvolatile** memory
  - fast/efficient access: through random access

What do hardware and OS provide?

- Hardware
  - persistence: disks provide **nonvolatile** memory
  - fast/efficient access: through random access
- Operating System
  - reliability: redundancy allows recovery from some additional failures
  - sharing/protection: read, write, and execute privileges for different users and groups
  - ease of use:
    - associating names with chunks of data (i.e., files)
    - organizing large collections of files into directories
    - transparent mapping of the user's concept of files and directories onto locations on disks
    - search facility in file systems (e.g., SpotLight in Mac OS X)

File system design problems

- how to translate from user-specified names to unique file numbers?
- how to lay out the files on disk?
- how to speed up file operations for small files?
  - since most files are small
- how to support both sequential and random access?
- how to identify data blocks that belong to a file?
  - what is the right data structure to maintain this information?
- how to allocate new blocks to a file?
  - must balance locality with expandability

File

- abstraction for information stored on various storage media

File

- abstraction for information stored on various storage media
- formal definition:
  - system's view: a named collection of bytes/blocks recorded on secondary storage media; a logical unit of storage
  - user's view: a durable data structure

File

- abstraction for information stored on various storage media
- formal definition:
  - system's view: a named collection of bytes/blocks recorded on secondary storage media; a logical unit of storage
  - user's view: a durable data structure
- different types of information can be stored in a file
  - source code, executable program, numeric data, text data (ASCII characters), photo, music, video, etc.

File

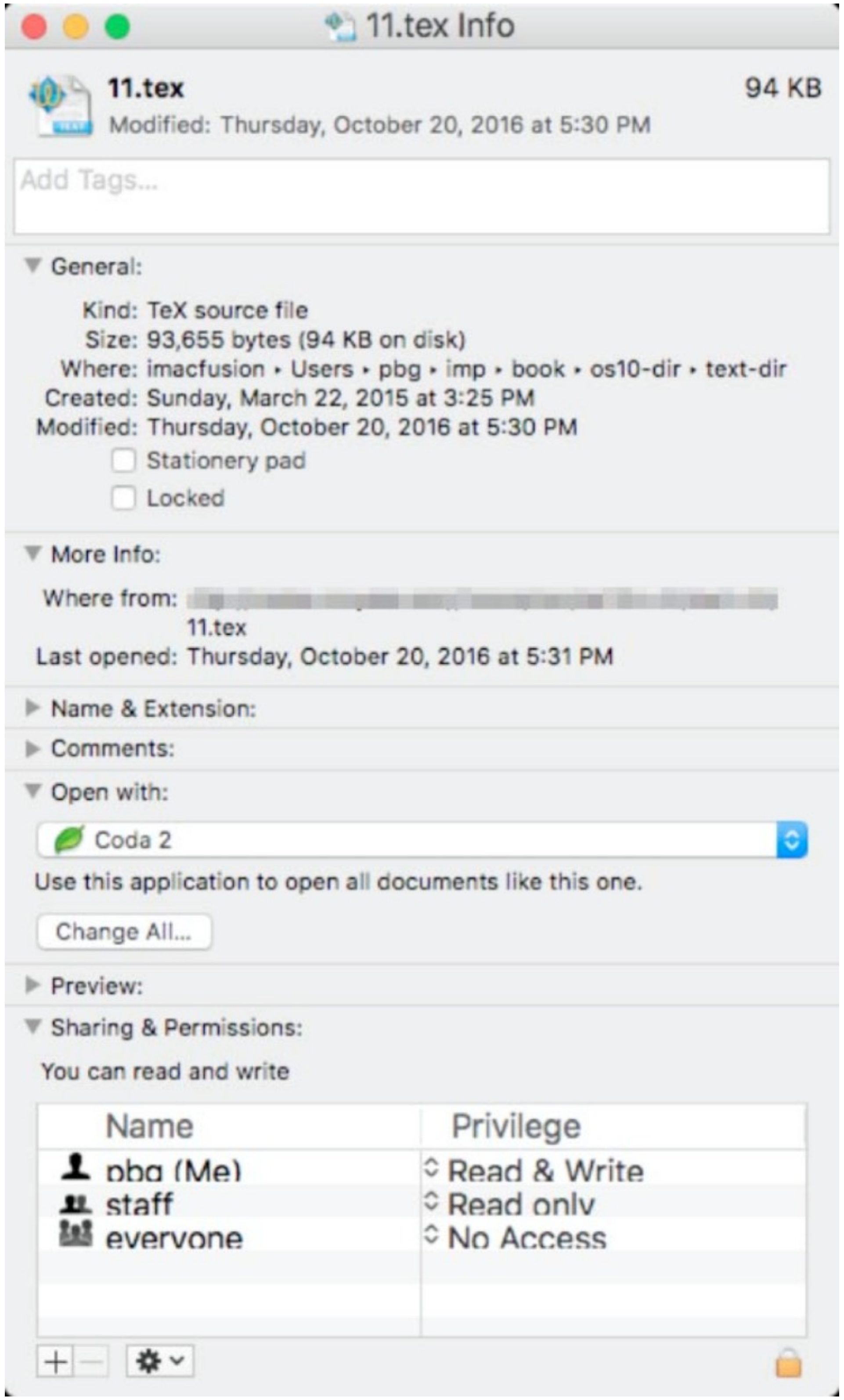
- abstraction for information stored on various storage media
- formal definition:
  - system's view: a named collection of bytes/blocks recorded on secondary storage media; a logical unit of storage
  - user's view: a durable data structure
- different types of information can be stored in a file
  - source code, executable program, numeric data, text data (ASCII characters), photo, music, video, etc.
- files can be structured or unstructured
  - UNIX defines files as streams of bytes (unstructured)
  - files can also be defined as a series of records or objects (structured)

File

- abstraction for information stored on various storage media
- formal definition:
  - system's view: a named collection of bytes/blocks recorded on secondary storage media; a logical unit of storage
  - user's view: a durable data structure
- different types of information can be stored in a file
  - source code, executable program, numeric data, text data (ASCII characters), photo, music, video, etc.
- files can be structured or unstructured
  - UNIX defines files as streams of bytes (unstructured)
  - files can also be defined as a series of records or objects (structured)
- most Operating Systems impose a minimal number of file structures
  - for example, executable files have a certain structure in UNIX and in Windows

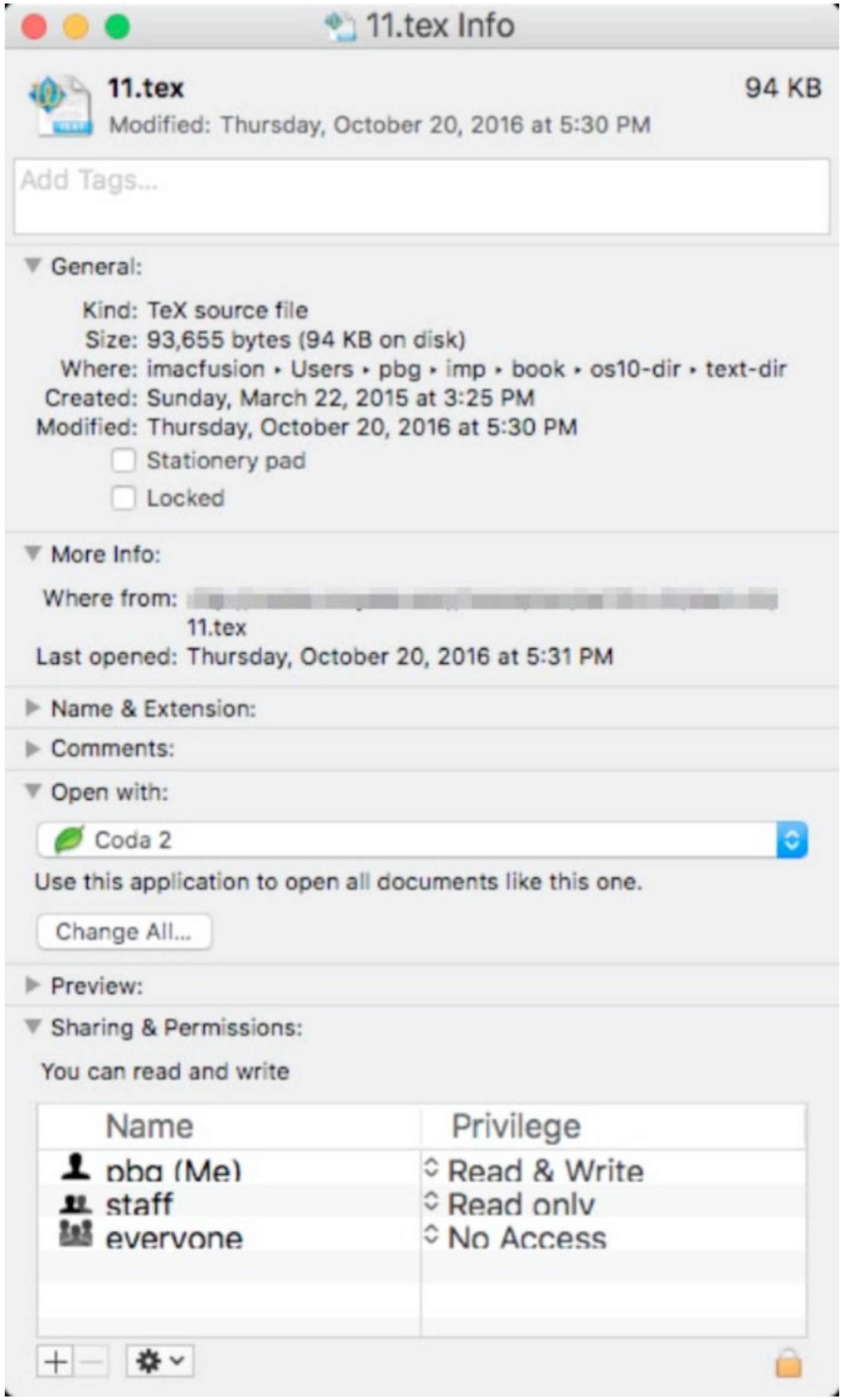
File attributes

- human-readable name
  - constraints: length, legal characters, case sensitiveness, etc.
  - must be unique in a name space



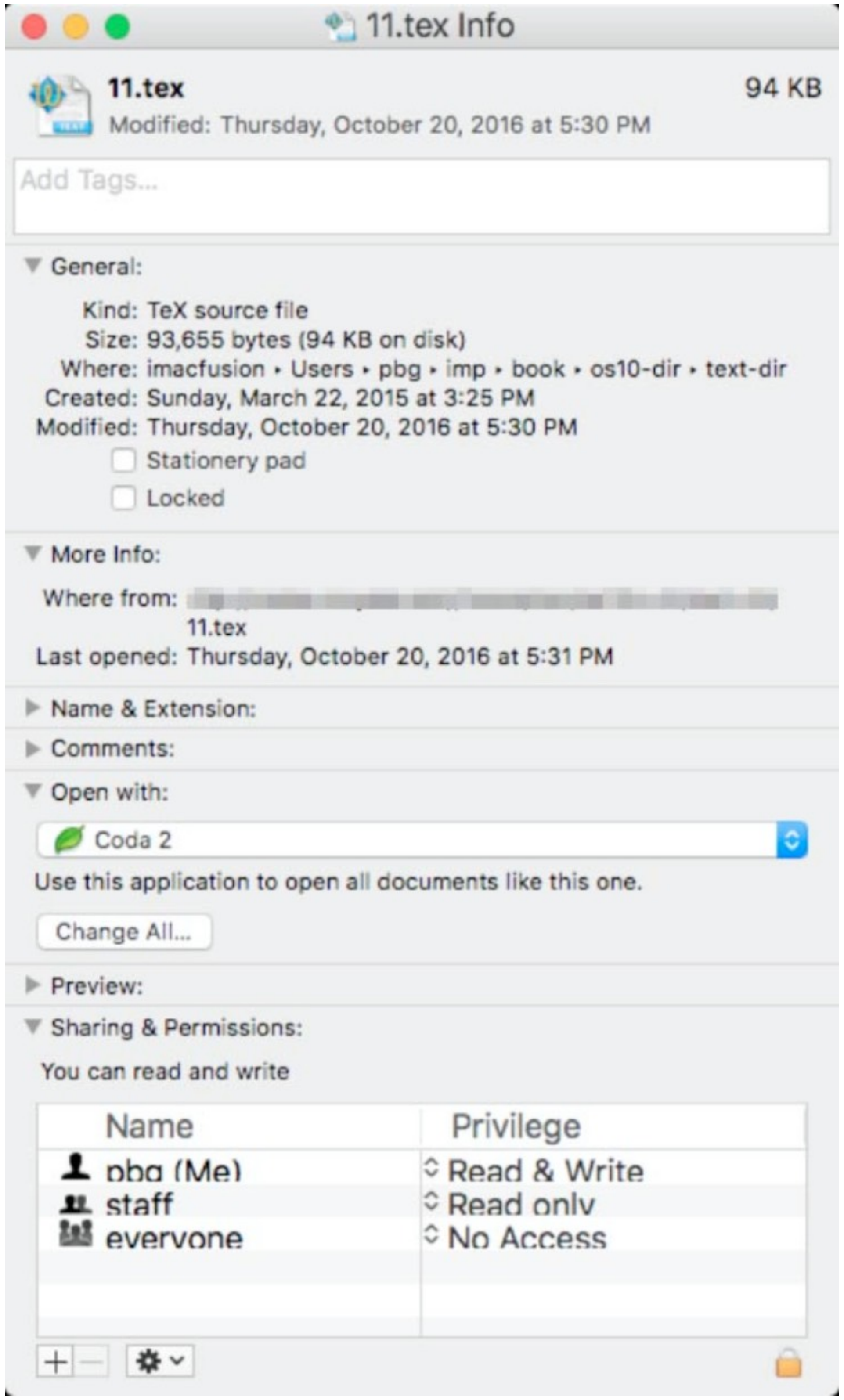
File attributes

- human-readable name
  - constraints: length, legal characters, case sensitiveness, etc.
  - must be unique in a name space
- unique identifier: a low-level name
  - also known as inode number



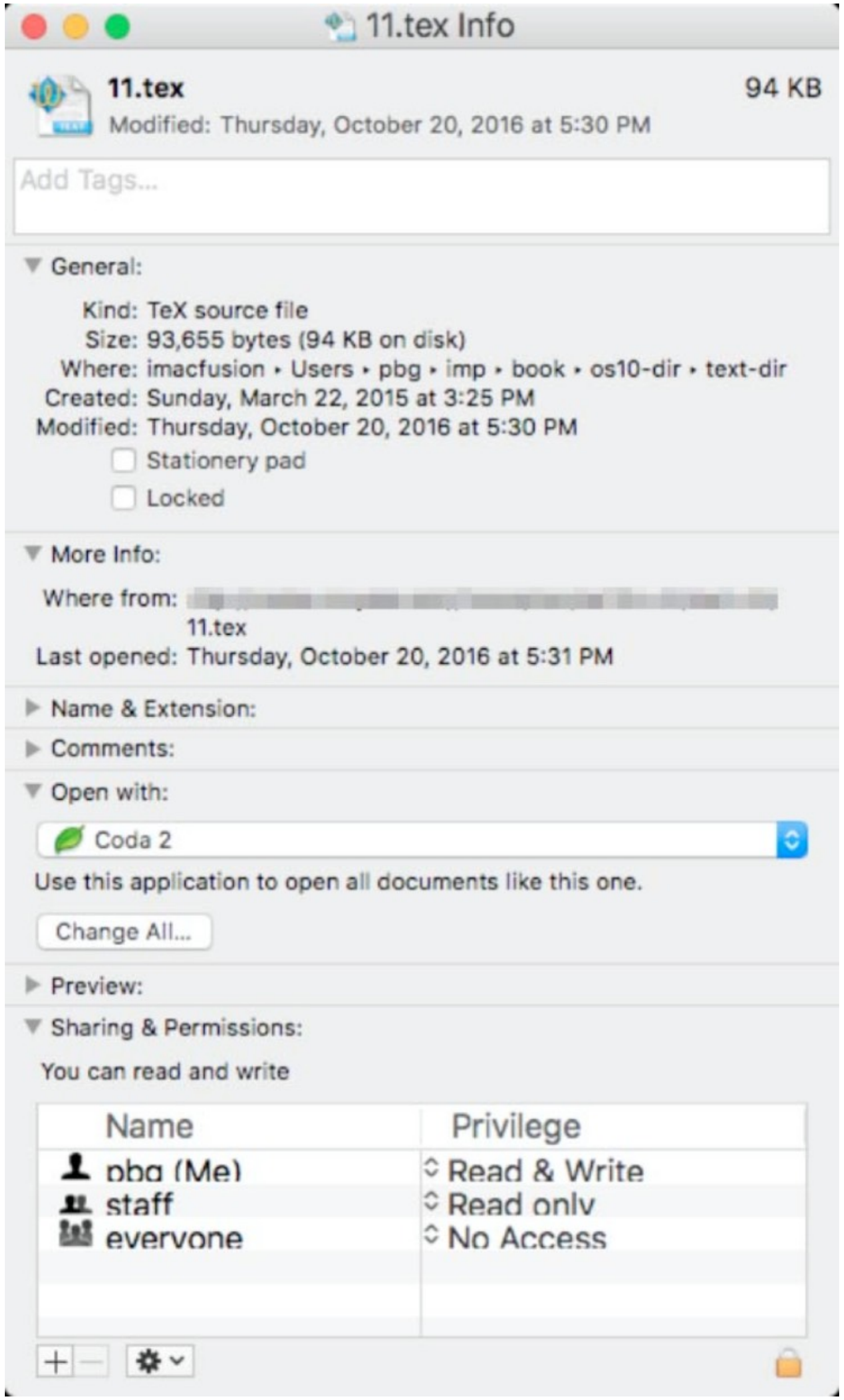
File attributes

- human-readable name
  - constraints: length, legal characters, case sensitiveness, etc.
  - must be unique in a name space
- unique identifier: a low-level name
  - also known as inode number
- type (recorded in Mac OS X/Windows)
  - advantage: error detection, default actions, user-friendliness
  - disadvantage: a more complex file system
  - file type can be part of the file name (i.e., the extension)
  - in UNIX, file extension is just a hint (not enforced)



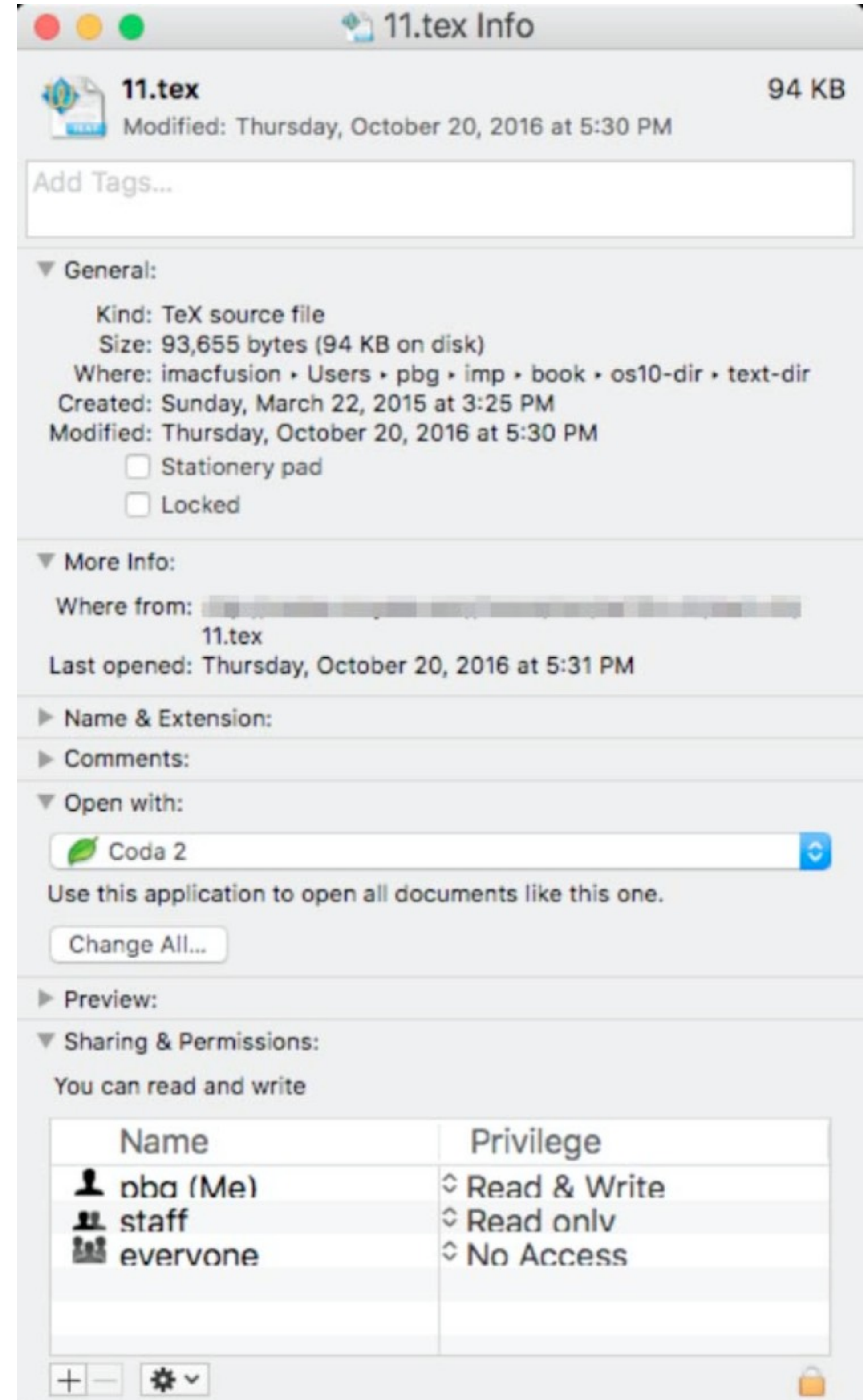
File attributes

- human-readable name
  - constraints: length, legal characters, case sensitiveness, etc.
  - must be unique in a name space
- unique identifier: a low-level name
  - also known as inode number
- type (recorded in Mac OS X/Windows)
  - advantage: error detection, default actions, user-friendliness
  - disadvantage: a more complex file system
  - file type can be part of the file name (i.e., the extension)
  - in UNIX, file extension is just a hint (not enforced)
- location: a pointer to the storage device and the file location on that device



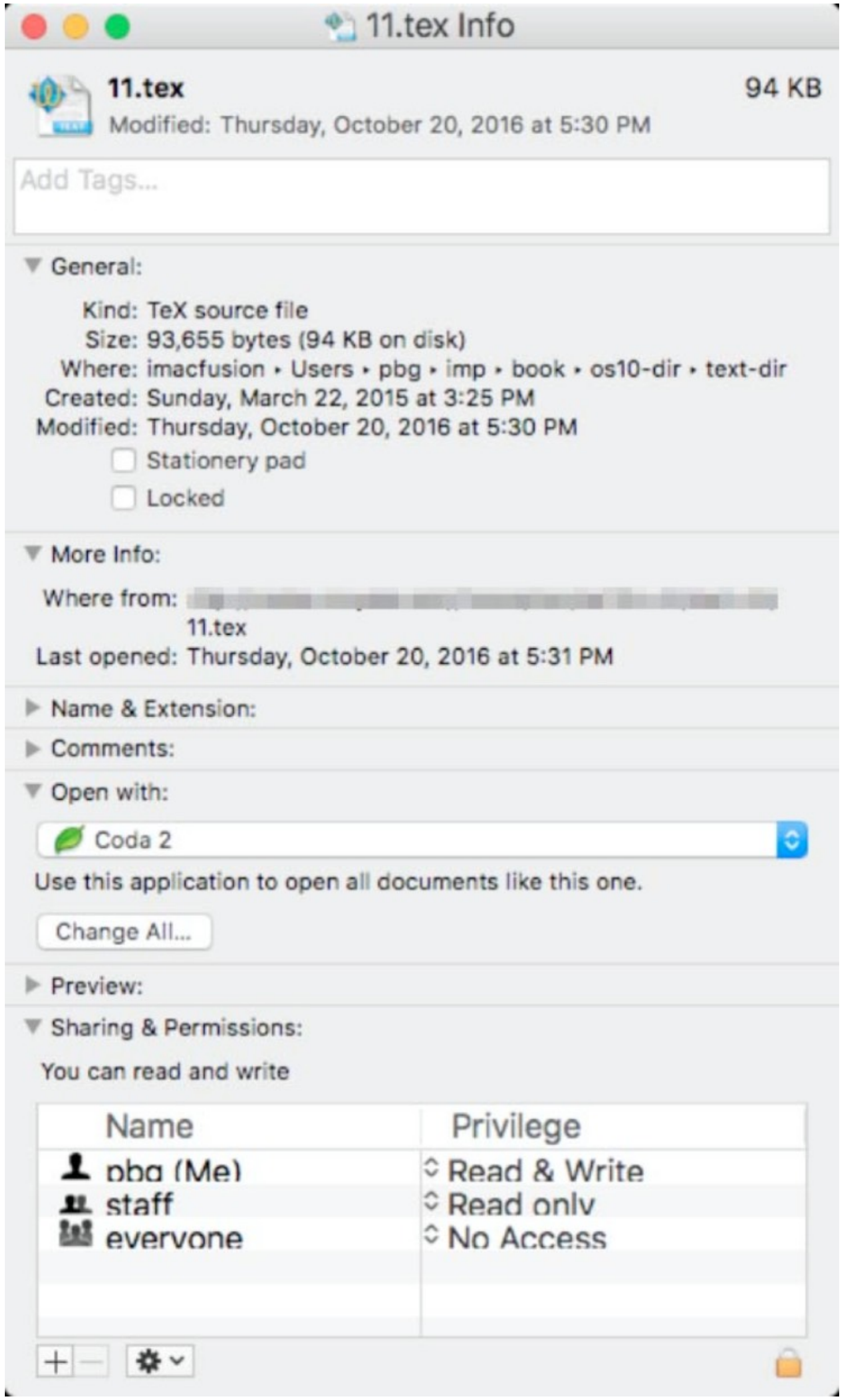
File attributes

- size: the current size (in bytes or blocks) and possibly the maximum supported size
  - file size is typically a multiple of block size because disk space is allocated in blocks
  - so some portion of the last block is generally wasted (**internal fragmentation**)
    - all file systems suffer from internal fragmentation



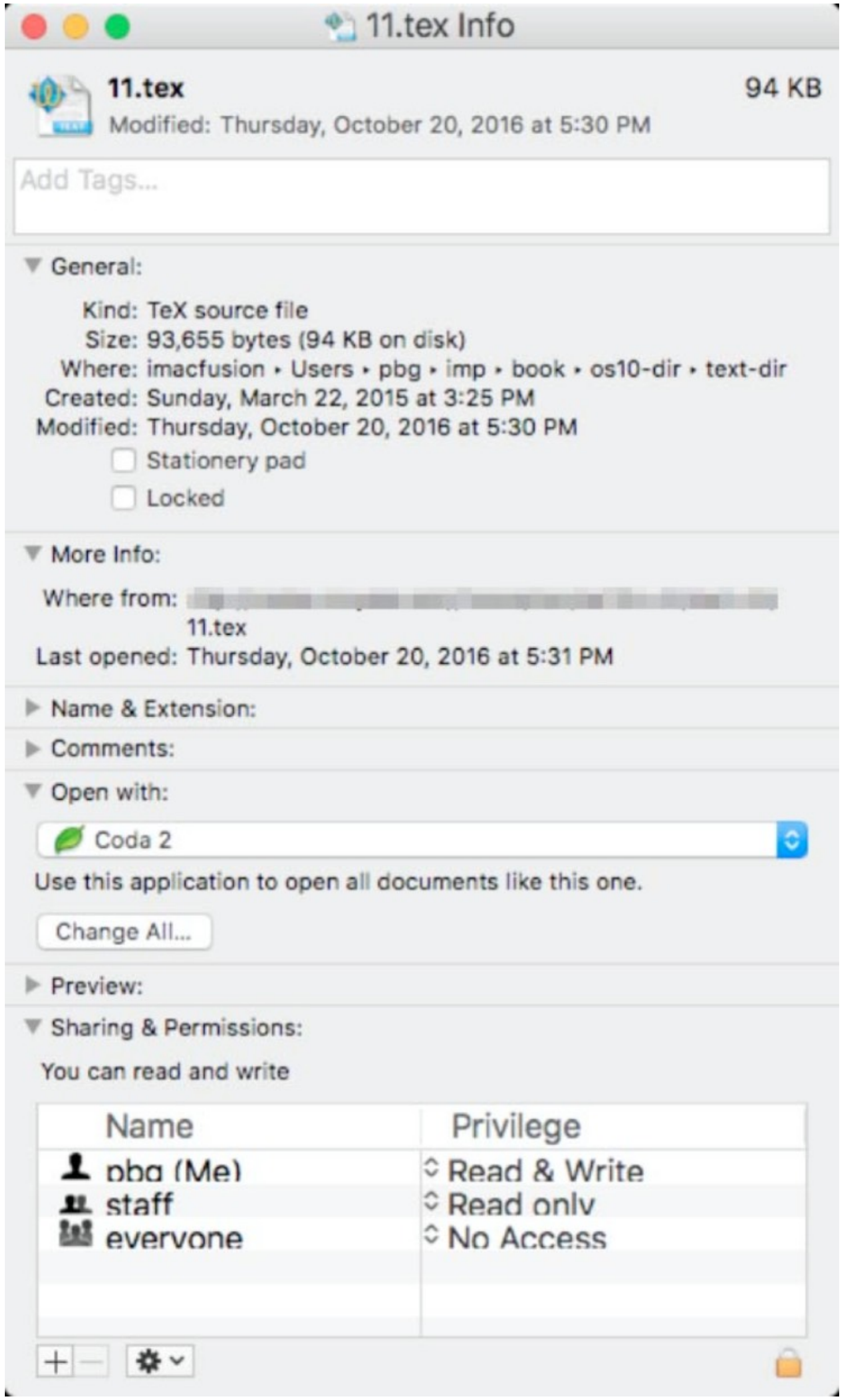
File attributes

- size: the current size (in bytes or blocks) and possibly the maximum supported size
  - file size is typically a multiple of block size because disk space is allocated in blocks
  - so some portion of the last block is generally wasted (**internal fragmentation**)
    - all file systems suffer from internal fragmentation
- protection: access control information



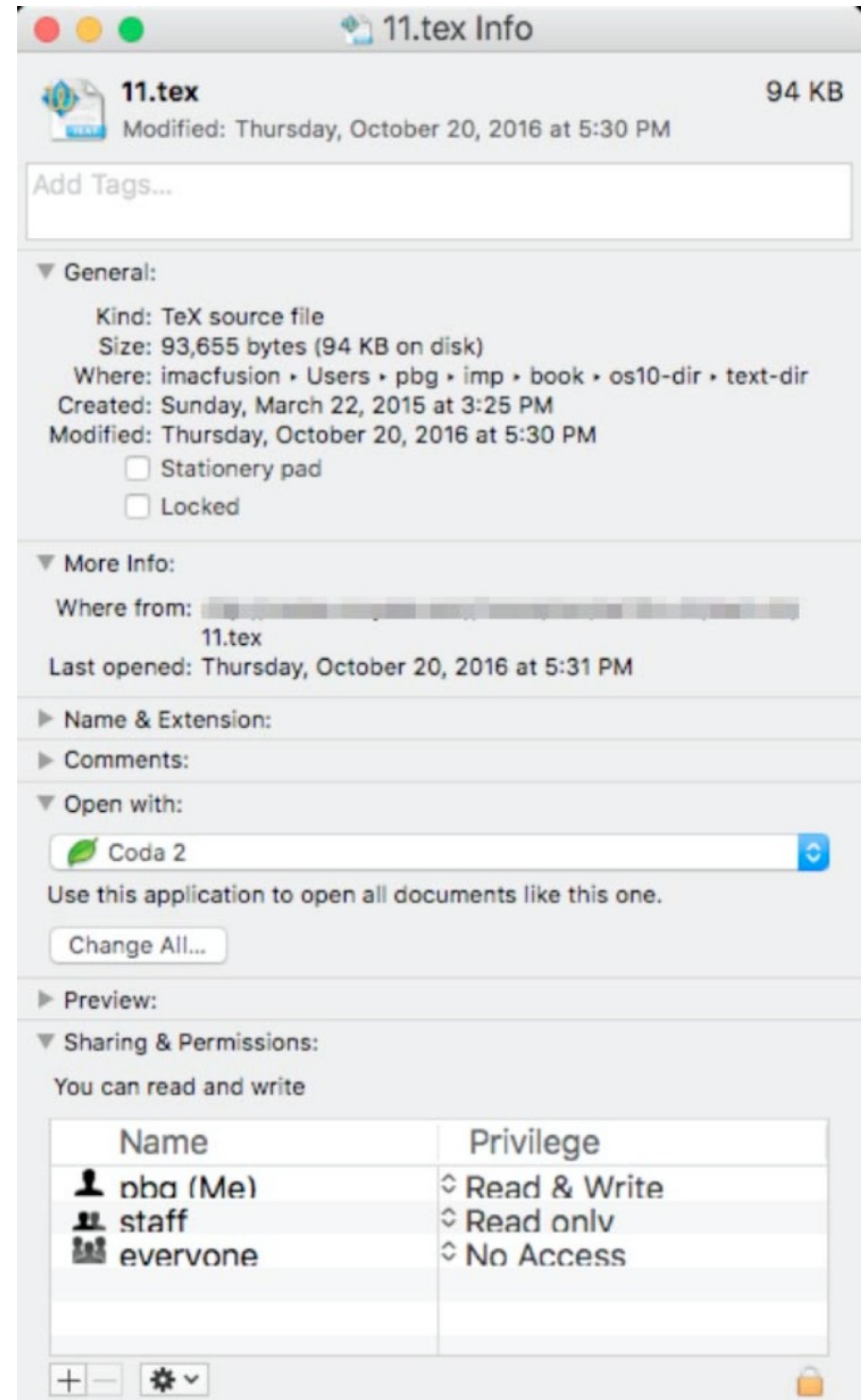
File attributes

- size: the current size (in bytes or blocks) and possibly the maximum supported size
  - file size is typically a multiple of block size because disk space is allocated in blocks
  - so some portion of the last block is generally wasted (**internal fragmentation**)
    - all file systems suffer from internal fragmentation
- protection: access control information
- user identification



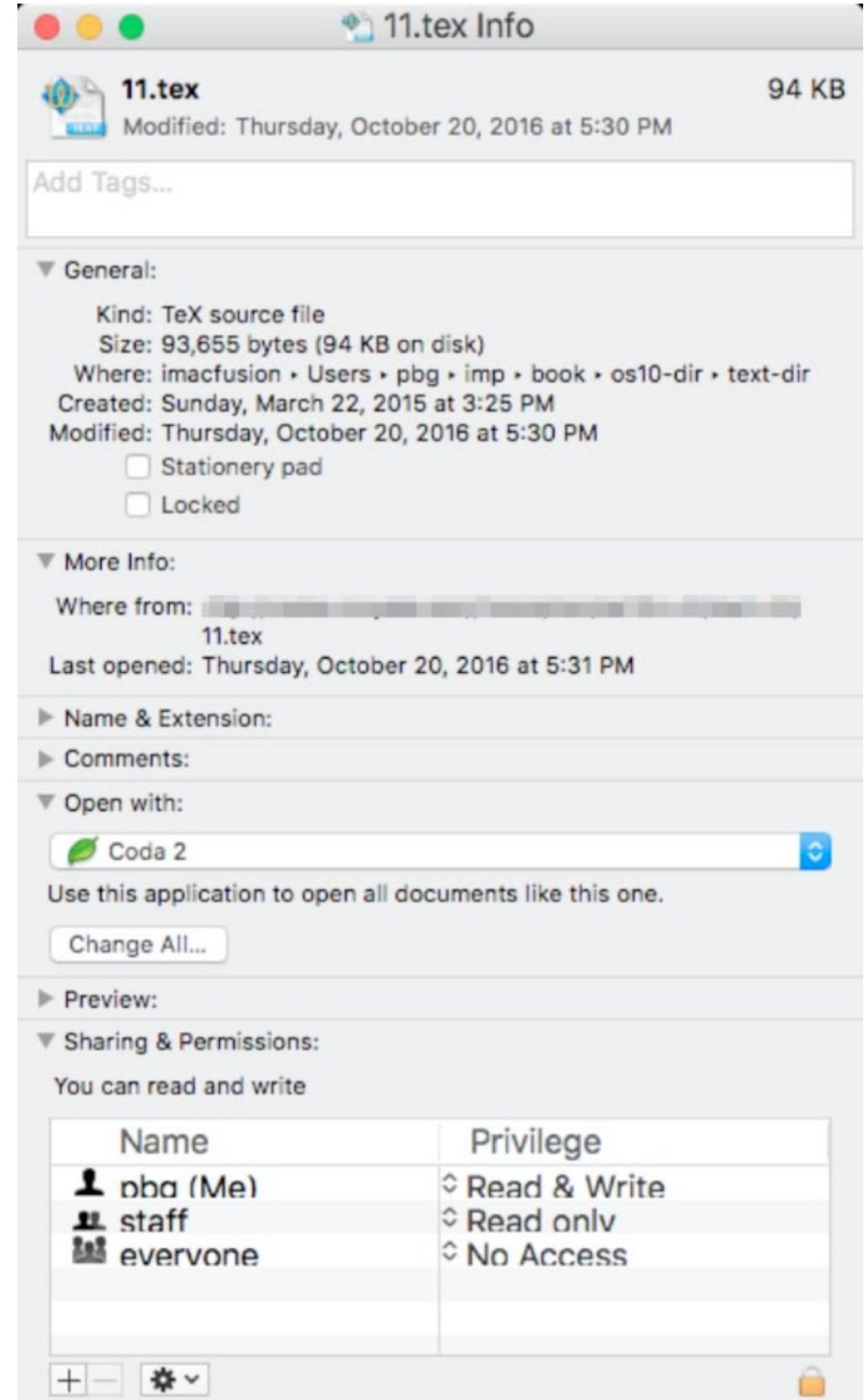
File attributes

- size: the current size (in bytes or blocks) and possibly the maximum supported size
  - file size is typically a multiple of block size because disk space is allocated in blocks
  - so some portion of the last block is generally wasted (**internal fragmentation**)
    - all file systems suffer from internal fragmentation
- protection: access control information
- user identification
- time and date: creation, last modified, and last used



File attributes

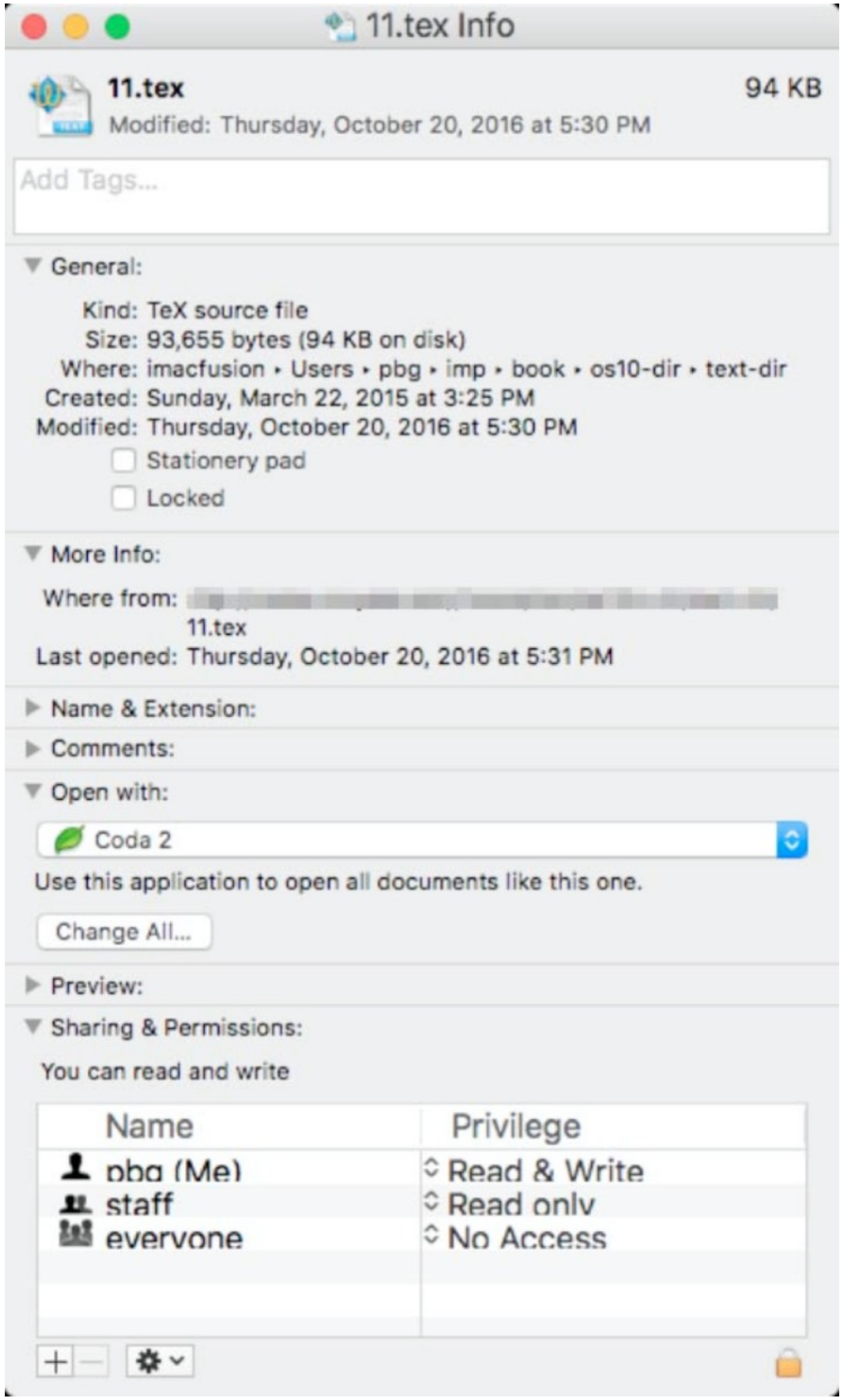
- size: the current size (in bytes or blocks) and possibly the maximum supported size
  - file size is typically a multiple of block size because disk space is allocated in blocks
  - so some portion of the last block is generally wasted (**internal fragmentation**)
    - all file systems suffer from internal fragmentation
- protection: access control information
- user identification
- time and date: creation, last modified, and last used
- and extended attributes
  - such as character encoding, checksum, etc.



File attributes

- size: the current size (in bytes or blocks) and possibly the maximum supported size
  - file size is typically a multiple of block size because disk space is allocated in blocks
  - so some portion of the last block is generally wasted (**internal fragmentation**)
    - all file systems suffer from internal fragmentation
- protection: access control information
- user identification
- time and date: creation, last modified, and last used
- and extended attributes
  - such as character encoding, checksum, etc.

see `struct file` in Linux



How many files are in a typical file system?

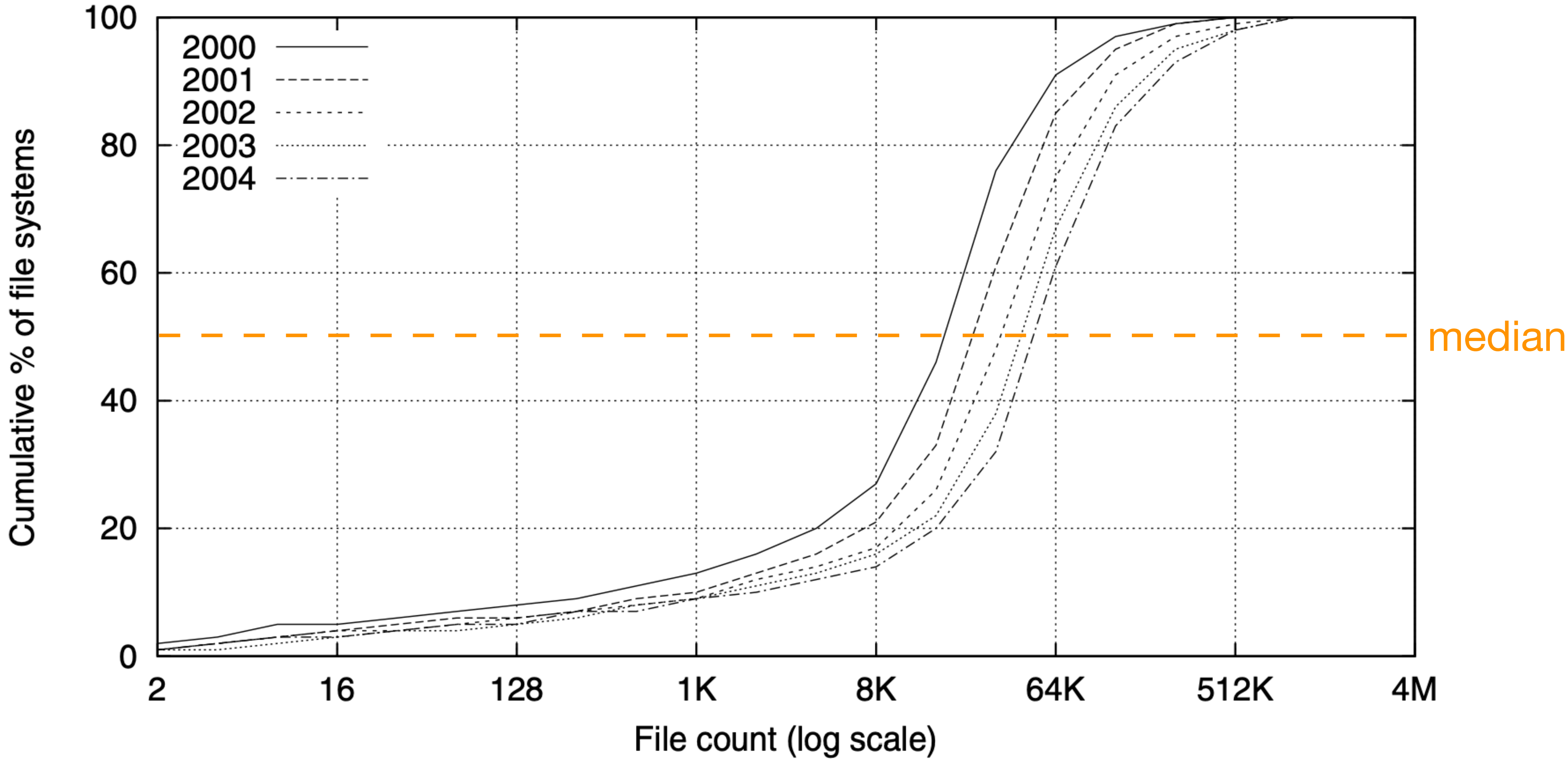


Fig. 1. CDFs of file systems by file count.

over the five-year period, the mean has grown from 30K to 90K files and the median has grown from 18K to 52K files (samples obtained from Microsoft desktops running Windows)

Reference: Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. 2007. A five-year study of file-system metadata. ACM Trans. Storage 3, 3, Article 9 (October 2007)

Most files are small but most disk space is taken up by large files

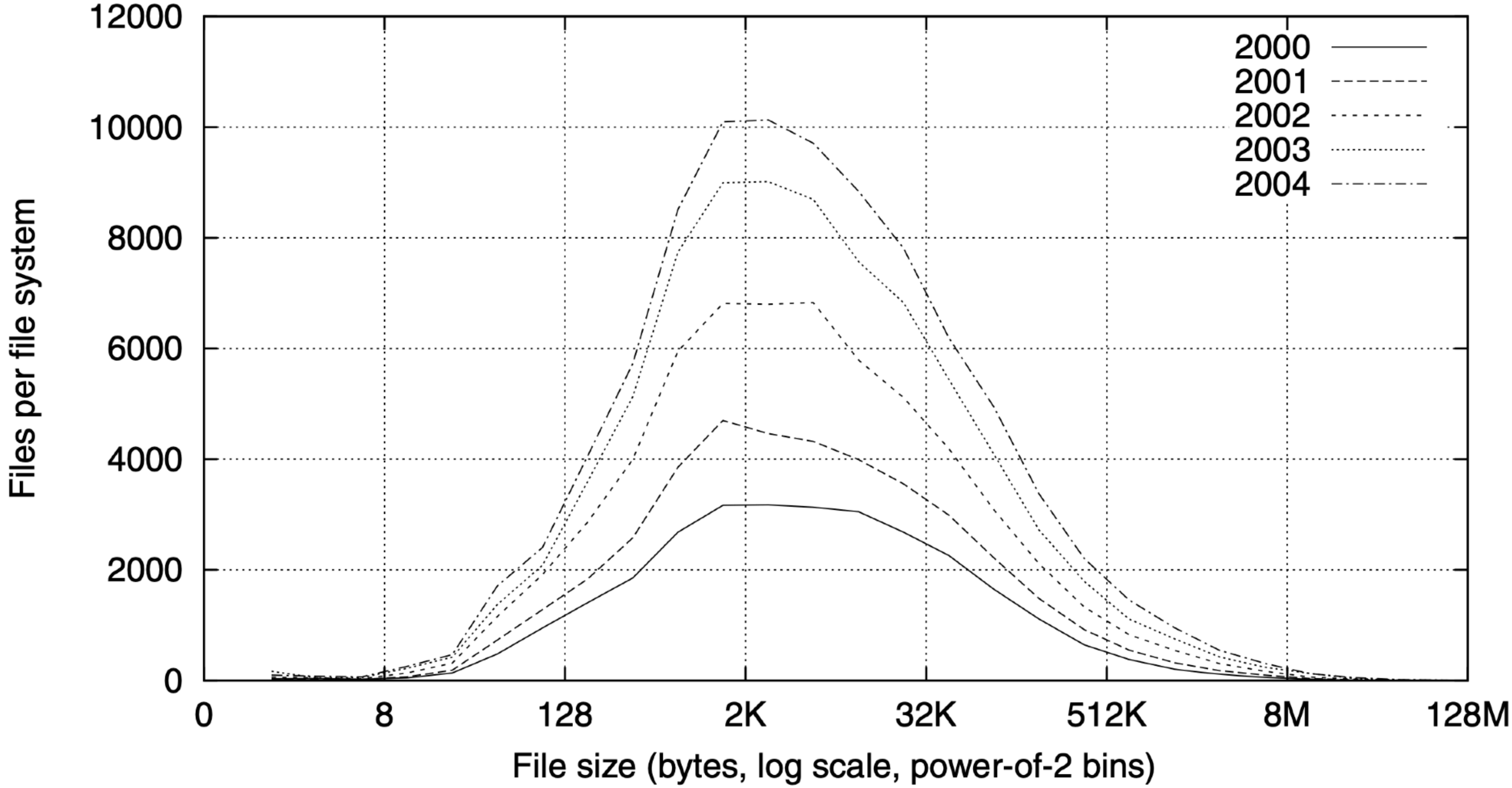


Fig. 2. Histograms of files by size.

thus small file access must be really fast, while large file access should be reasonably efficient

Reference: Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. 2007. A five-year study of file-system metadata. ACM Trans. Storage 3, 3, Article 9 (October 2007)

File system operations

- two categories of file operations
  - basic file system operations (mostly I/O related)
    - `Read(), Write(), Create(), Delete(), Truncate()`
    - `Seek(), Open(), Close()`
  - operations modifying the file attributes
    - `HardLink(), SoftLink(), SetAttribute(), GetAttribute(), Rename(), ChangeMode(), ChangeOwner()`

File operation: create

- take the file name as input
- find and allocate disk space
  - check disk quotas, permissions, etc.
- create a file descriptor (i.e., a handle) for the file
  - include name, location on disk, and all file attributes
- add the file descriptor to the directory that contains the file

File operation: write

- take the file name and data that must be written to the file
- search the directory for the entry associated with the named file
  - then find the file's location on disk
- write data to the file position specified by the **write pointer**
- update the write pointer

File operation: read

- take the file name and a buffer with specific size to read data into it
- search the directory for the named file
  - find the file's location on disk
- read data from the file position specified by the **read pointer** into the buffer
- update the read pointer

File operation: read

- take the file name and a buffer with specific size to read data into it
- search the directory for the named file
  - find the file's location on disk
- read data from the file position specified by the **read pointer** into the buffer
- update the read pointer
- **note:** read and write operations can use the same pointer since a process either reads from or writes to a file at once
  - this pointer is kept as a per-process **current-file-position pointer**

File operation: delete

- take the file name as input
- search the directory for the named file
- free all disk blocks allocated to the file
- remove the file descriptor from the directory
- other operations?
  - reference counting, hard links, etc.

File operation: seek

- take the file name and file offset as input
- search the directory for the named file
- reposition the current-file-position pointer to the given value
    - it does not require any disk I/O, just updates the pointer

File operation: seek

- take the file name and file offset as input
- search the directory for the named file
- reposition the current-file-position pointer to the given value
  - it does not require any disk I/O, just updates the pointer
- seek is typically used for direct/random access

File operation: truncate

- take the file name as input
- search the directory for the named file
- release all disk blocks allocated to the file
- keep file attributes unchanged, except for its size which is reset to zero
    - truncating is faster than deleting the file and recreating it

Two kernel data structures

- system-wide open-file table: a (doubly linked) list of files in use; shared by all processes with an open file
  - open count (how many processes have the file open)
  - file attributes, including ownership, protection information, access times, file sizes
  - location of file on disk
  - pointers to location of file in memory

Two kernel data structures

- system-wide open-file table: a (doubly linked) list of files in use; shared by all processes with an open file
  - open count (how many processes have the file open)
  - file attributes, including ownership, protection information, access times, file sizes
  - location of file on disk
  - pointers to location of file in memory
- per-process file descriptor table: a (doubly linked) list of pointers into the open-file table for files opened by the process; it is stored in the process's PCB
  - so for each file opened by the process, there is an entry in this list that contains
    - a pointer to its entry in the system-wide open-file table
    - current-file-position pointer (offset into the file)
    - mode in which the process will access the file (r, w, rw), and file locks (shared & exclusive)
    - accounting information

File operation: open

- take the file name and access mode as input
- search the directory for the named file

File operation: open

- take the file name and access mode as input
- search the directory for the named file
- check access mode (create, read-only, read-write, append-only, etc.) against file's permissions

File operation: open

- take the file name and access mode as input
- search the directory for the named file
- check access mode (create, read-only, read-write, append-only, etc.) against file’s permissions
- if the requested access is permitted, copy the entry associated with the file to the system-wide open-file table (if it isn’t opened already by another process) and increment the open count

File operation: open

- take the file name and access mode as input
- search the directory for the named file
- check access mode (create, read-only, read-write, append-only, etc.) against file's permissions
- if the requested access is permitted, copy the entry associated with the file to the system-wide open-file table (if it isn't opened already by another process) and increment the open count
- add a pointer to the open-file table entry to the process file descriptor table

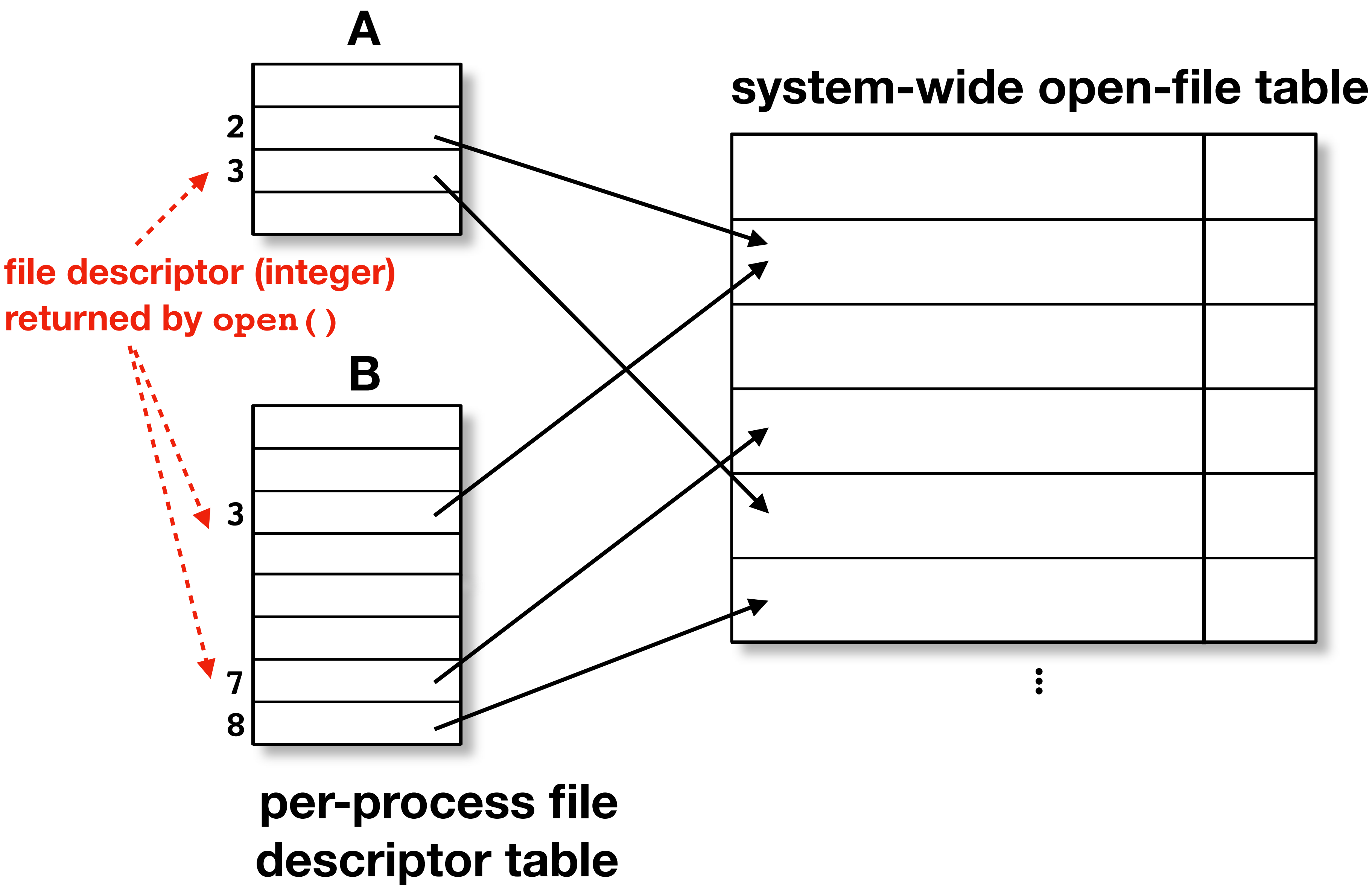
File operation: open

- take the file name and access mode as input
- search the directory for the named file
- check access mode (create, read-only, read-write, append-only, etc.) against file's permissions
- if the requested access is permitted, copy the entry associated with the file to the system-wide open-file table (if it isn't opened already by another process) and increment the open count
- add a pointer to the open-file table entry to the process file descriptor table
- initialize the current-file-position pointer to the start of the file

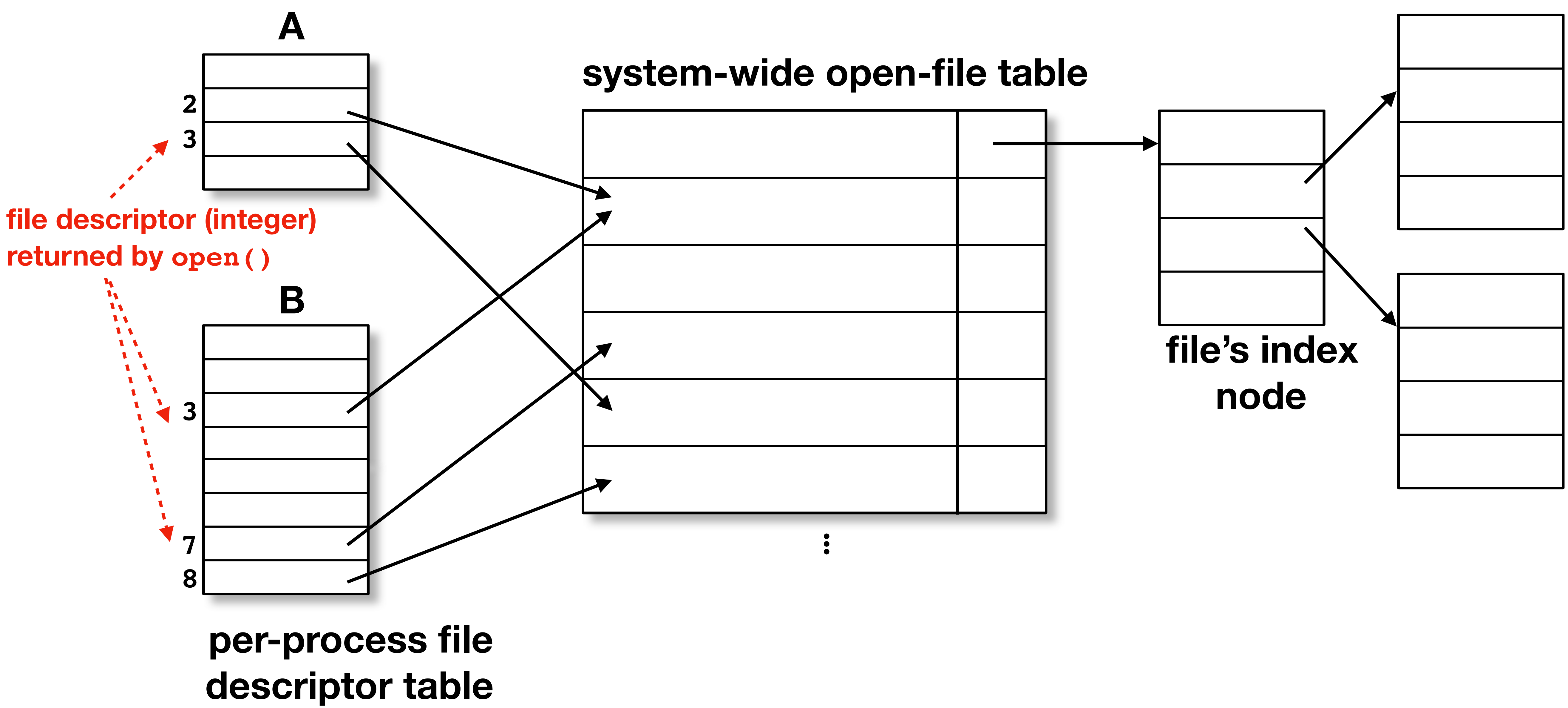
File operation: open

- take the file name and access mode as input
- search the directory for the named file
- check access mode (create, read-only, read-write, append-only, etc.) against file's permissions
- if the requested access is permitted, copy the entry associated with the file to the system-wide open-file table (if it isn't opened already by another process) and increment the open count
- add a pointer to the open-file table entry to the process file descriptor table
- initialize the current-file-position pointer to the start of the file
- return a pointer to the file entry in the process file descriptor table (referred to as **file descriptor**)
  - so when a file operation is requested, the file is specified via an index into the file descriptor table and locate the corresponding entry in the system-wide open-file table
  - eliminate the need to search the directory for the file entry each time (a one-time cost amortized over many operations)

Kernel's file-related data structures



Kernel's file-related data structures



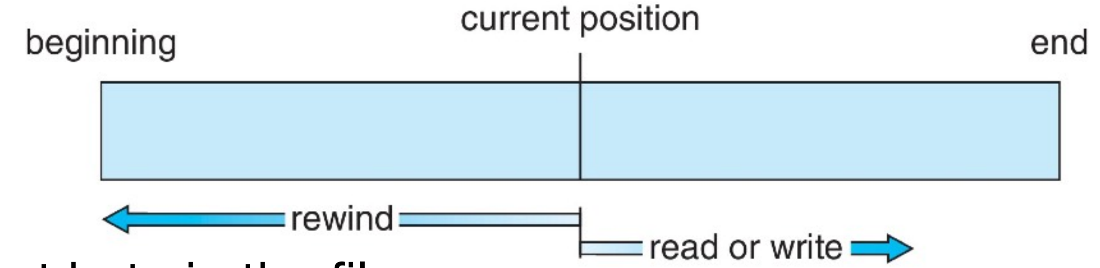
File operation: close

- locate and remove the file's entry from the process file table
- decrement the open count in the system-wide open-file table
- if the open count reaches zero, remove the entry from the system-wide open-file table
  - otherwise the kernel may run out space

Modified operations: using file descriptor

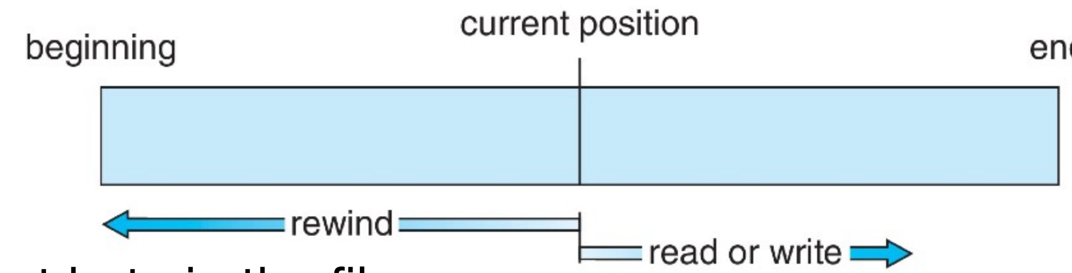
- fd = Open("filename")
- Close(fd)
- Truncate(fd)
- Seek(fd, offset)
- Read(fd, buffer, length)
- Write(fd, buffer, length)

File access methods



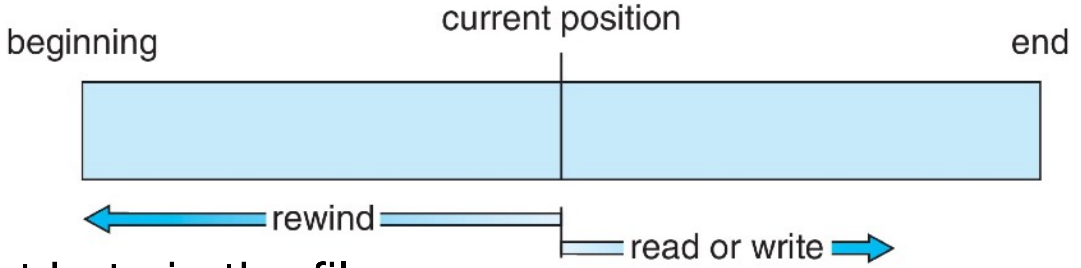
- sequential: information in the file is processed/accessed in order
  - is based on the tape model: keep a pointer to the next byte in the file
  - update the pointer on each read/write/seek operation
  - commonly used in text editors, compilers

File access methods



- sequential: information in the file is processed/accessed in order
  - is based on the tape model: keep a pointer to the next byte in the file
  - update the pointer on each read/write/seek operation
  - commonly used in text editors, compilers
- direct access: address any block in the file rapidly given its offset within the file
  - is based on the disk model: disks allow random access to any block
  - the block number to access is relative to the beginning of the file
    - so the programmer doesn't need to know where the file is stored on disk
  - it is possible to read block 8 then 12 and then 5
  - databases use this method of access; they compute which block contains the answer (e.g., by hashing the key provided in the query) and read that block directly

File access methods



- sequential: information in the file is processed/accessed in order
  - is based on the tape model: keep a pointer to the next byte in the file
  - update the pointer on each read/write/seek operation
  - commonly used in text editors, compilers
- direct access: address any block in the file rapidly given its offset within the file
  - is based on the disk model: disks allow random access to any block
  - the block number to access is relative to the beginning of the file
    - so the programmer doesn't need to know where the file is stored on disk
  - it is possible to read block 8 then 12 and then 5
  - databases use this method of access; they compute which block contains the answer (e.g., by hashing the key provided in the query) and read that block directly
- some systems only support one access method, some support both methods, and some support only the method declared when the file was created

Simulating sequential access on a direct-access 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 on a direct-access 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 direct access on a sequential-access file is extremely inefficient (**why?**)