# File Systems

**key concepts**

file, directory, link, open/close, descriptor, read, write, seek, file naming, block, i-node, crash consistency, journaling

**reading**

Three Easy Pieces: Chapters 39-40,42





## Files and File Systems

- files: persistent, named data objects
  - data consists of a sequence of numbered bytes
  - file may change size over time
  - file has associated meta-data
    - examples: owner, access controls, file type, creation and access timestamps

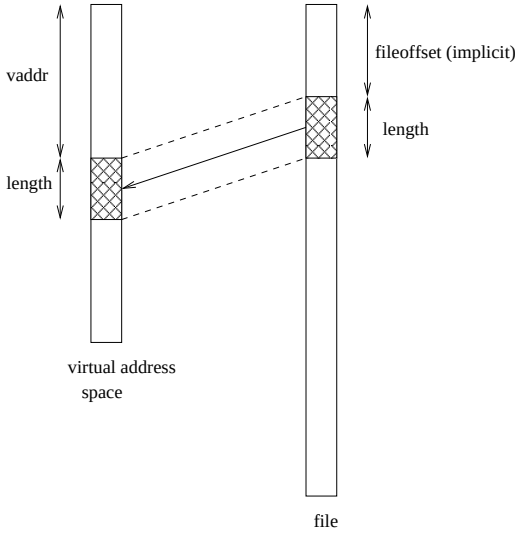
## File Interface: Basics

- open
  - open returns a file identifier (or handle or descriptor), which is used in subsequent operations to identify the file.
  - other operations (e.g., read, write) require file descriptor as a parameter
- close
  - kernel tracks while file descriptors are currently valid for each process
  - close invalidates a valid file descriptor
- read, write, seek
  - read copies data from a file into a virtual address space
  - write copies data from a virtual address space into a file
  - seek enables non-sequential reading/writing
- get/set file meta-data, e.g., Unix `fstat, chmod`





## File Read




```
read(fileID, vaddr, length)
```

## File Position and Seeks

- each file descriptor (open file) has an associated file position
- read and write operations
  - start from the current file position
  - update the current file position
- this makes sequential file I/O easy for an application to request
- seeks (`lseek`) are used for achieve non-sequential file I/O
  - `lseek` changes the file position associated with a descriptor
  - next read or write from that descriptor will use the new position





## Sequential File Reading Example

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i++) {
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, $512$ bytes at a time.

## File Reading Example Using Seek

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=1; i<=100; i++) {
  lseek(f,(100-i)*512,SEEK_SET);
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, $512$ bytes at a time, in reverse order.





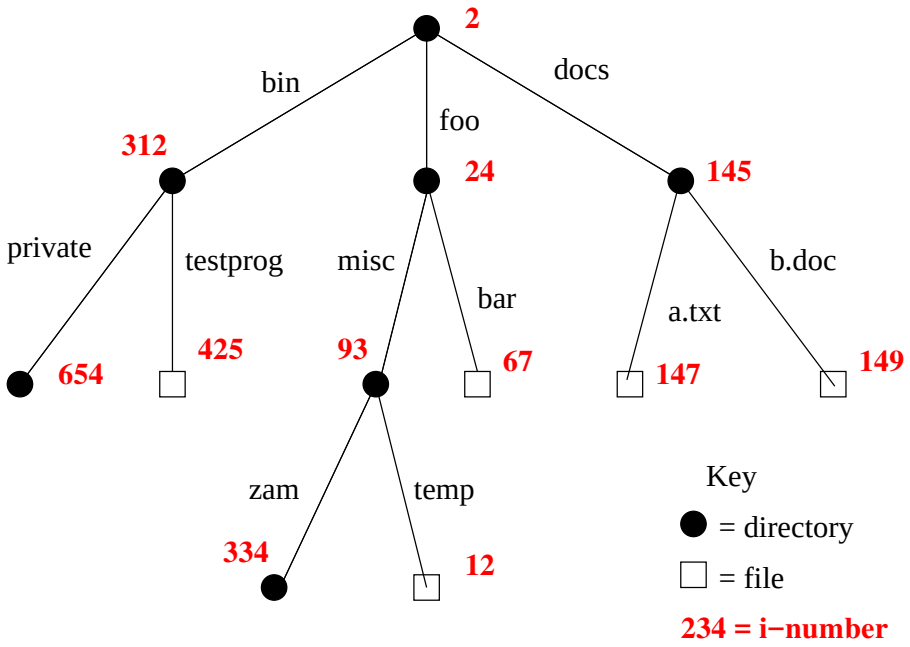
## Directories and File Names

- A directory maps *file names* (strings) to *i-numbers*
  - an i-number is a unique (within a file system) identifier for a file or directory
  - given an i-number, the file system can find the data and meta-data for the file
- Directories provide a way for applications to group related files
- Since directories can be nested, a filesystem's directories can be viewed as a tree, with a single *root* directory.
- In a directory tree, files are leaves
- Files may be identified by *pathnames*, which describe a path through the directory tree from the root directory to the file, e.g.:

  `/home/user/courses/cs350/notes/filesys.pdf`

- Directories also have pathnames
- Applications refer to files using pathnames, not i-numbers

## Hierarchical Namespace Example



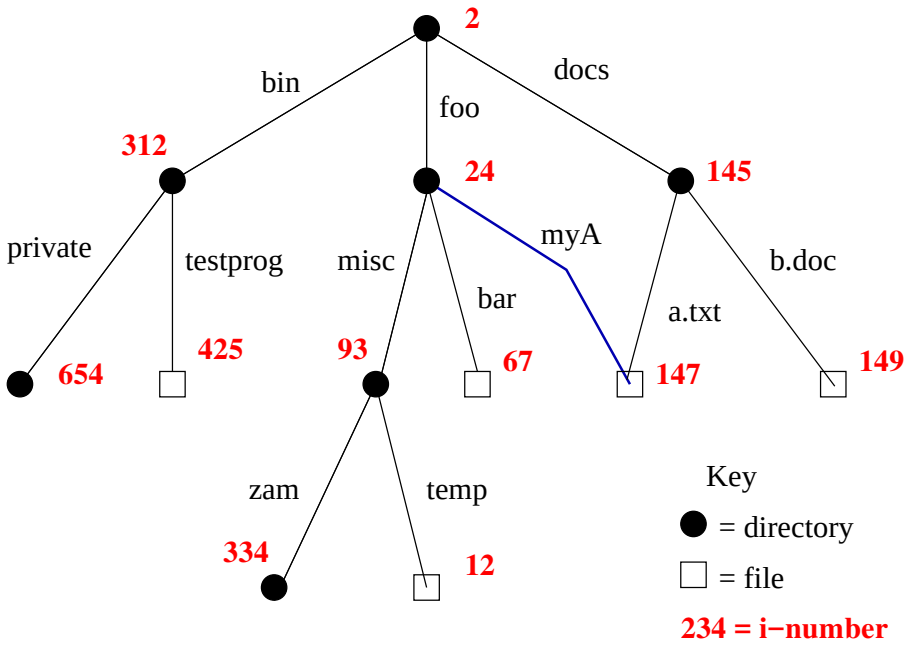





## Links

- a *hard link* is an association between a name (string) and an i-number
  - each entry in a directory is a hard link
- when a file is created, so is a hard link to that file
  - `open(/foo/misc/biz,O_CREAT|O_TRUNC)`
  - this creates a new file if a file called `/foo/misc/biz` does not already exist
  - it also creates a hard link to the file in the directory `/foo/misc`
- Once a file is created, *additional* hard links can be made to it.
  - example: `link(/docs/a.txt,/foo/myA)` creates a new hard link `myA` in directory `/foo`. The link refers to the i-number of file `/docs/a.txt`, which must exist.
- linking to an existing file creates a new pathname for that file
  - each file has a unique i-number, but may have multiple pathnames
- Not possible to `link` to a directory (to avoid cycles)

## Hierarchical Namespace Example








## Unlinking

- hard links can be removed:
  - `unlink(/docs/b.doc)`
  - this removes the link `b.doc` from the directory `/docs`
- when the last hard link to a file is removed, the file is also removed
  - since there are no links to the file, it has no pathname, and can no longer be opened

## Multiple File Systems

- it is not uncommon for a system to have multiple file systems
- some kind of global file namespace is required
- two examples:

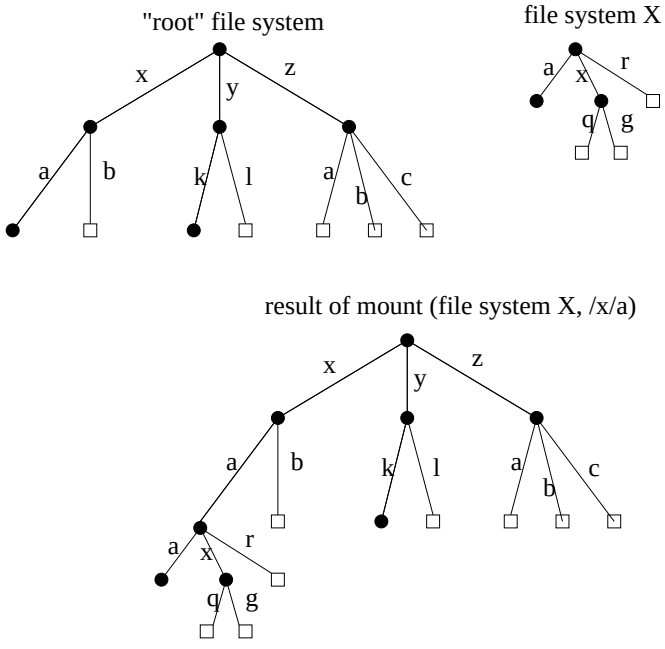
  **DOS/Windows:** use two-part file names: file system name, pathname within file system
  - example: `C:\user\cs350\schedule.txt`

  **Unix:** create single hierarchical namespace that combines the namespaces of two file systems
  - Unix `mount` system call does this
- mounting does *not* make two file systems into one file system
  - it merely creates a single, hierarchical namespace that combines the namespaces of two file systems
  - the new namespace is temporary - it exists only until the file system is unmounted





## Unix `mount` Example



## File System Implementation

- what needs to be stored persistently?
  - file data
  - file meta-data
  - directories and links
  - file system meta-data
- non-persistent information
  - open files per process
  - file position for each open file
  - *cached* copies of persistent data



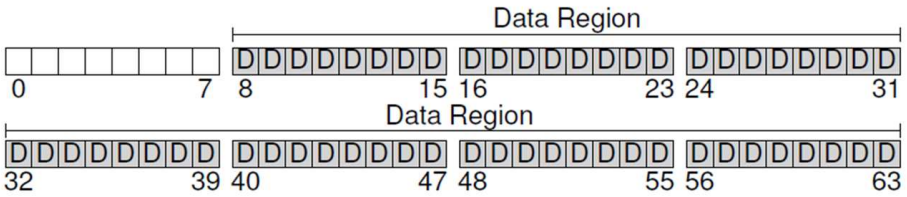


## File System Example

- Use an extremely small disk as an example:
  - 256 KB disk!
  - Most disks have a sector size of 512 bytes
    - Memory is usually *byte addressable*
    - Disk is usually "sector addressable"
  - 512 total sectors on this disk
- Group every 8 consecutive sectors into a block
  - Better spatial locality (fewer seeks)
  - Reduces the number of block pointers (we'll see what this means soon)
  - 4 KB block is a convenient size for demand paging
  - 64 total blocks on this disk

## VSFS: Very Simple File System (1 of 5)

- Most of the blocks should be for storing user data (last 56 blocks)

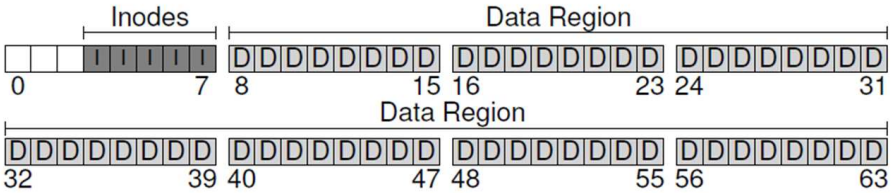






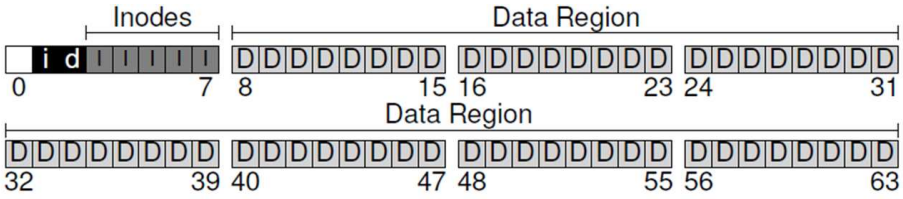

## VSFS: Very Simple File System (2 of 5)

- Need some way to map files to data blocks
- Create an array of i-nodes, where each i-node contains the meta-data for a file
  - The index into the array is the file's index number (i-number)
- Assume each i-node is 256 bytes, and we dedicate 5 blocks for i-nodes
  - This allows for 80 total i-nodes/files



## VSFS: Very Simple File System (3 of 5)

- We also need to know which i-nodes and blocks are unused
- Many ways of doing this:
  - In VSFS, we use a bitmap for each
  - Can also use a free list instead of a bitmap
- A block size of 4 KB means we can track 32K i-nodes and 32K blocks
  - This is far more than we actually need

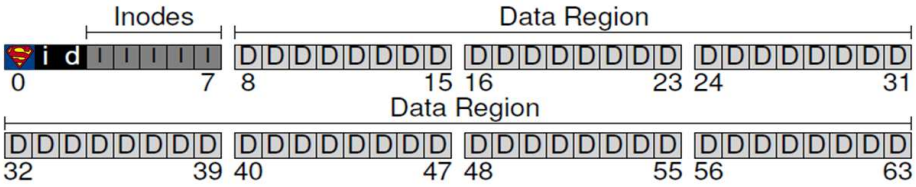




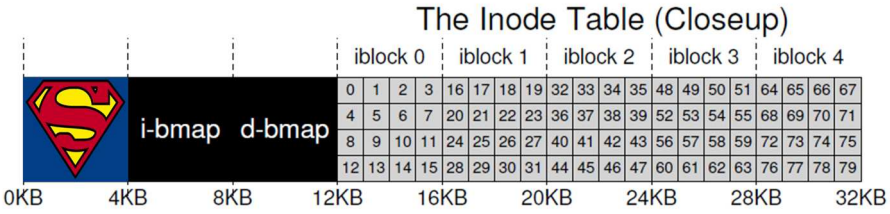



## VSFS: Very Simple File System (4 of 5)

- Reserve the first block as the **superblock**
- A superblock contains meta-information about the entire file system
  - e.g., how many i-nodes and blocks are in the system, where the i-node table begins, etc.



## VSFS: Very Simple File System (5 of 5)








## i-nodes

- An i-node is a *fixed size* index structure that holds both file meta-data and a small number of pointers to data blocks
- i-node fields may include:
  - file type
  - file permissions
  - file length
  - number of file blocks
  - time of last file access
  - time of last i-node update, last file update
  - number of hard links to this file
  - direct data block pointers
  - single, double, and triple indirect data block pointers

## VSFS: i-node

- Assume disk blocks can be referenced based on a 4 byte address
  - $2^{32}$ blocks, 4 KB blocks
  - Maximum disk size is 16 TB
- In VSFS, an i-node is 256 bytes
  - Assume there is enough room for 12 direct pointers to blocks
  - Each pointer points to a different block for storing user data
  - Pointers are ordered: first pointer points to the first block in the file, etc.
- What is the maximum file size if we only have direct pointers?
  - 12 * 4 KB = 48 KB
- Great for small files (which are common)
- Not so great if you want to store big files

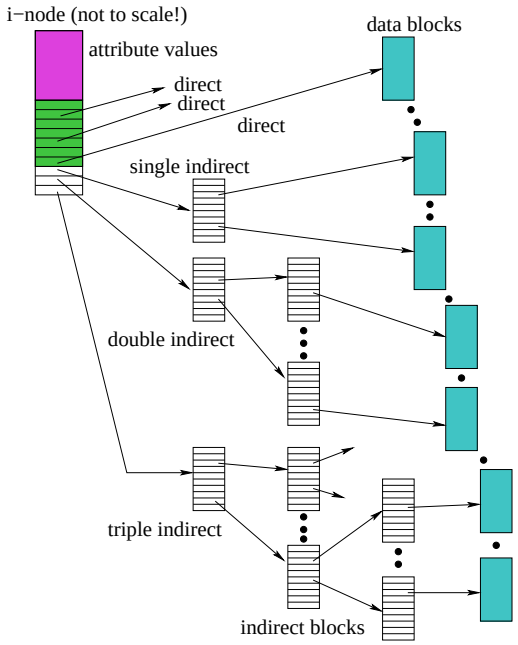




## VSFS: Indirect Blocks

- In addition to 12 direct pointers, we can also introduce an **indirect pointer**
  - An indirect pointer points to a block full of direct pointers
- 4 KB block of direct pointers = 1024 pointers
  - Maximum file size is: (12 + 1024) * 4 KB = 4144 KB
- This is more than enough for any file that can fit on our tiny 256KB disk, but what if the disk were larger?
- Add a **double indirect pointer**
  - Points to a 4 KB block of indirect pointers
  - (12 + 1024 + 1024 * 1024) * 4 KB
  - Just over 4 GB in size (is this enough?)

## i-node Diagram








## File System Design

- File system parameters:
  - *How many i-nodes should a file system have?*
  - *How many direct and indirect blocks should an i-node have?*
  - *What is the "right" block size?*

- For a general purpose file system, design it to be efficient for the common case

| | |
|---|---|
| **Most files are small** | Roughly 2K is the most common size |
| **Average file size is growing** | Almost 200K is the average |
| **Most bytes are stored in large files** | A few big files use most of the space |
| **File systems contains lots of files** | Almost 100K on average |
| **File systems are roughly half full** | Even as disks grow, file systems remain ~50% full |
| **Directories are typically small** | Many have few entries; most have 20 or fewer |

## Directories

- Implemented as a special type of file.
- Directory file contains directory entries, each consisting of
  - a file name (component of a path name) and the corresponding i-number

| name | i-number |
|---|---|
| . | 5 |
| .. | 2 |
| foo | 12 |
| bar | 13 |
| foobar | 24 |

- Directory files can be read by application programs (e.g., `ls`)
- Directory files are only updated by the kernel, in response to file system operations, e.g, create file, create link
- Application programs cannot write directly to directory files. (Why not?)





## In-Memory (Non-Persistent) Structures

- per process
  - descriptor table
    - which file descriptors does this process have open?
    - to which file does each open descriptor refer?
    - what is the current file position for each descriptor?
- system wide
  - open file table
    - which files are currently open (by any process)?
  - i-node cache
    - in-memory copies of recently-used i-nodes
  - block cache
    - in-memory copies of data blocks and indirect blocks

## Reading From a File (/foo/bar)

- First read the root i-node
  - At "well known" position (i-node 2)
  - i-node 1 is usually for tracking bad blocks

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open(bar) | | | read | | | read | | | | |
| | | | | read | | | | | | |
| | | | | | read | | read | | | |
| read() | | | | | read | | | | | |
| | | | | | write | | | read | | |
| read() | | | | | read | | | | | |
| | | | | | write | | | | read | |
| read() | | | | | read | | | | | |
| | | | | | write | | | | | read |





## Reading From a File (/foo/bar)

- Read the directory information from root
  - Find the i-number for foo
  - Read the foo i-node

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open(bar) | | | read | | | read | | | | |
| | | | | read | | | | | | |
| | | | | | read | | read | | | |
| read() | | | | | read | | | | | |
| | | | | | write | | | read | | |
| read() | | | | | read | | | | | |
| | | | | | write | | | | read | |
| read() | | | | | read | | | | | |
| | | | | | write | | | | | read |

## Reading From a File (/foo/bar)

- Read the directory information from foo
  - Find the i-number for bar
  - Read the bar i-node

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open(bar) | | | read | | | read | | | | |
| | | | | read | | | read | | | |
| | | | | | read | | | | | |
| read() | | | | | read | | | | | |
| | | | | | write | | | read | | |
| read() | | | | | read | | | | | |
| | | | | | write | | | | read | |
| read() | | | | | read | | | | | |
| | | | | | write | | | | | read |





## Reading From a File (/foo/bar)

- Permission check (is the user allowed to read this file?)
- Allocate a file descriptor in the per-process descriptor table
- Increment the counter for this i-number in the global open file table

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open(bar) | | | read | | | read | | | | |
| | | | | read | | | read | | | |
| | | | | | read | | | | | |
| read() | | | | | read | | | | | |
| | | | | | write | | | read | | |
| read() | | | | | read | | | | | |
| | | | | | write | | | | read | |
| read() | | | | | read | | | | | |
| | | | | | write | | | | | read |

## Reading From a File (/foo/bar)

- Find the block using a direct/indirect pointer and read the data
- Update the i-node with a new access time
- Update the file position in the per-process descriptor table
- Closing a file deallocates the file descriptor and decrements the counter for this i-number in the global open file table

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open(bar) | | | read | read | read | read | read | | | |
| read() | | | | | read<br>write | | | read | | |
| read() | | | | | read<br>write | | | | read | |
| read() | | | | | read<br>write | | | | | read |





## Creating a File (/foo/bar)

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| create (/foo/bar) | | read<br>write | read | read | read<br>write<br>write | read | read | write | | |
| write() | read<br>write | | | | read<br>write | | | write | | |
| write() | read<br>write | | | | read<br>write | | | | write | |
| write() | read<br>write | | | | read<br>write | | | | | write |

## Problems Caused by Failures

- a single logical file system operation may require several disk I/O operations
- example: deleting a file
  - remove entry from directory
  - remove file index (i-node) from i-node table
  - mark file's data blocks free in free space index
- what if, because of a failure, some but not all of these changes are reflected on the disk?

---

- system failure will destroy in-memory file system structures
- persistent structures should be *crash consistent*, i.e., should be consistent when system restarts after a failure

---





## Fault Tolerance

- special-purpose consistency checkers (e.g., Unix `fsck` in Berkeley FFS, Linux ext2)
  - runs after a crash, before normal operations resume
  - find and attempt to repair inconsistent file system data structures, e.g.:
    - file with no directory entry
    - free space that is not marked as free
- journaling (e.g., Veritas, NTFS, Linux ext3)
  - record file system meta-data changes in a journal (log), so that sequences of changes can be written to disk in a single operation
  - *after* changes have been journaled, update the disk data structures (*write-ahead logging*)
  - after a failure, redo journaled updates in case they were not done before the failure