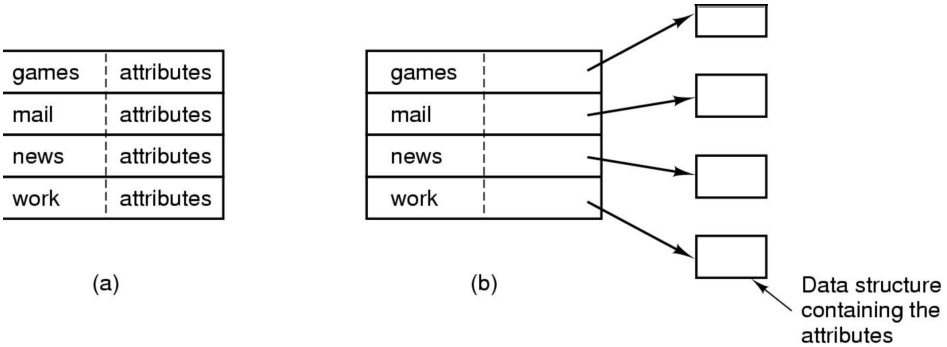
Karlsruhe Institute of Technology

Operating Systems

19. Directories & File System Examples

Prof. Dr. Frank Bellosa | WT 2021/2022

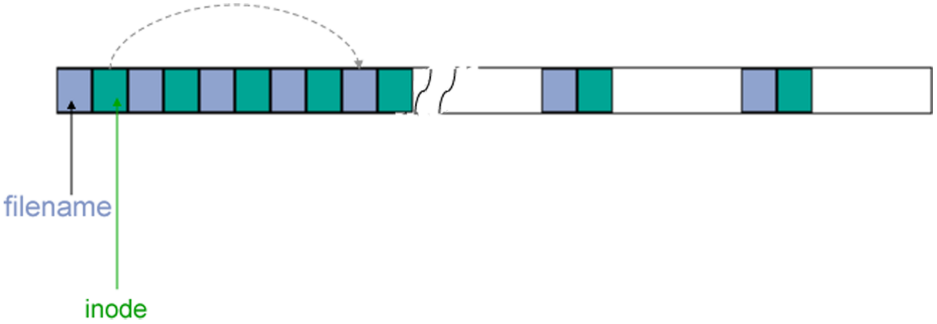
Implementing Directories (1)



(a) A simple directory (MS-DOS)
- fixed size entries
- disk addresses and attributes in directory entry

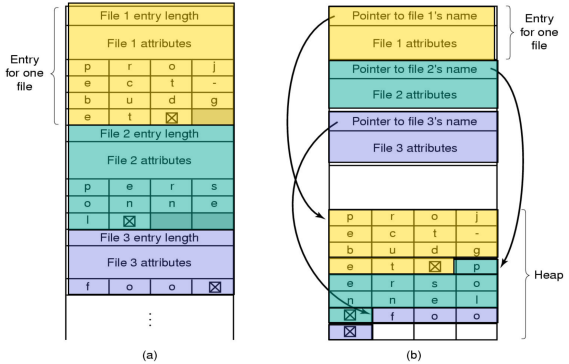
(b) Directory in which each entry just refers to an i-node (UNIX)

Implementing Directories (2)



- What to do when some entries are deleted?
  - Never reuse
    - Bridge over the directory holes
  - Compaction, but when?
    - eager or
    - lazy

Directory Entries & Long Filenames



- Two ways of handling long file names in directories
  - (a) In-line
  - (b) In a heap

Analysis: Linear Directory Lookup

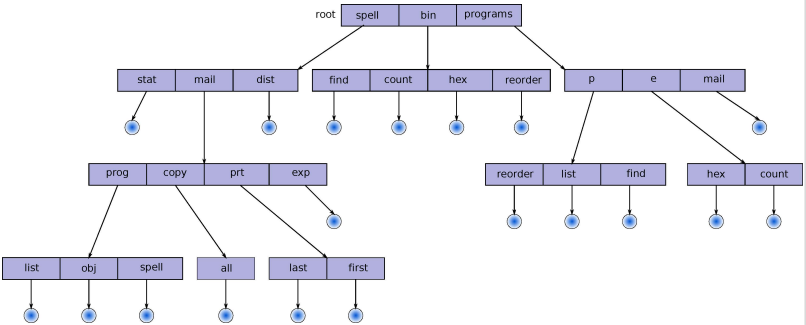
- Linear search $\Rightarrow$ for big directories not efficient
- Space efficient as long as we do compaction
  - Either eagerly after entry deletion or
  - Lazily (but when?)
- With variable file names $\Rightarrow$ deal with fragmentation
- Alternatives
  - (e.g. extensible) hashing
  - (e.g. B-)tree structures

Hashing a Directory Lookup

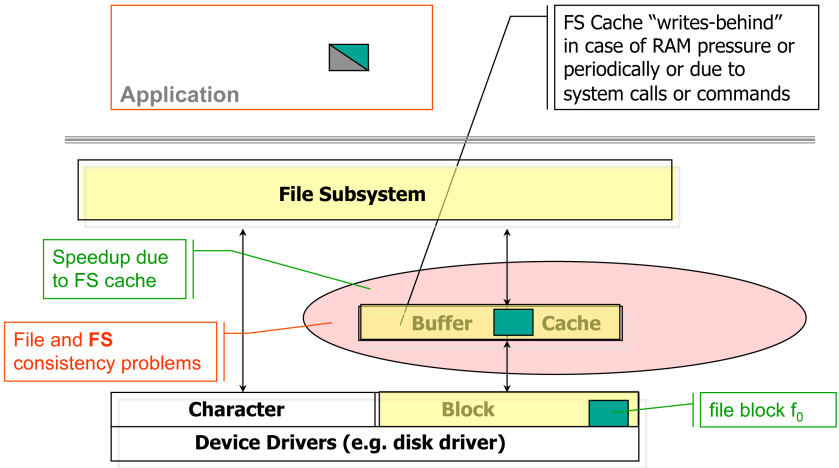
- Method:
  - Hashing a file name to an inode
  - Space for filename and meta data is variable sized
  - Create/delete will trigger space allocation and clearing
- Advantages:
  - Fast lookup and relatively simple
- Disadvantages:
  - Not as efficient as trees for very large directories

Tree Structure for a Directory

- Method:
  - Sort files by name
  - Store directory entries in a B-tree like structure
  - Create/delete/search in that B-tree
- Advantages:
  - Efficient for a large number
- Disadvantages:
  - Complex
  - Not that efficient for a small



UNIX File System Structure



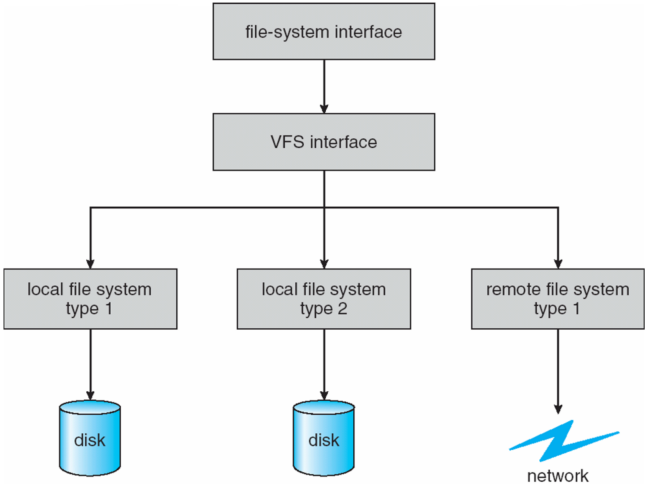
Buffer Cache

- Write-behind policy might lead to
  - data losses in case of system crash or power loss
    - forcing all dirty buffers to disk with `fsync()`
    - `rename()` is implemented atomic
  - inconsistent state of the FS (see [ADAD18] chapter 42)
    - requires file system check `fsck`
    - journaling (write-ahead logging)
- Always two copies involved
  - from disk to buffer cache (in kernel space)
  - from buffer to user address space
- FS Cache *wiping* if sequentially reading a very large file from end to end and not accessing it again

Virtual File Systems

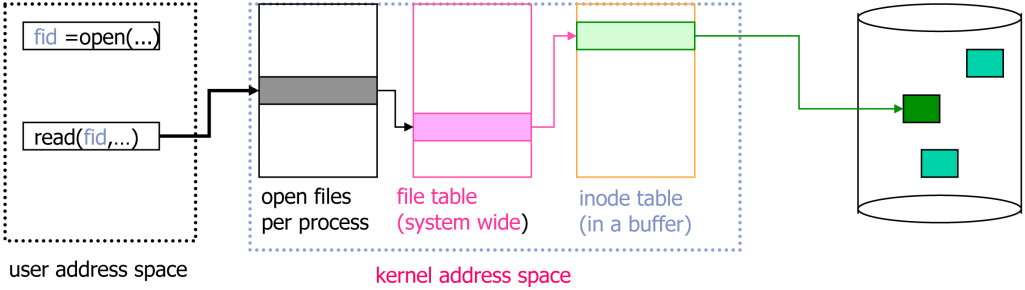
- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is for the VFS interface, rather than any specific type of file system.

Schematic View of Virtual File System [SGG12]

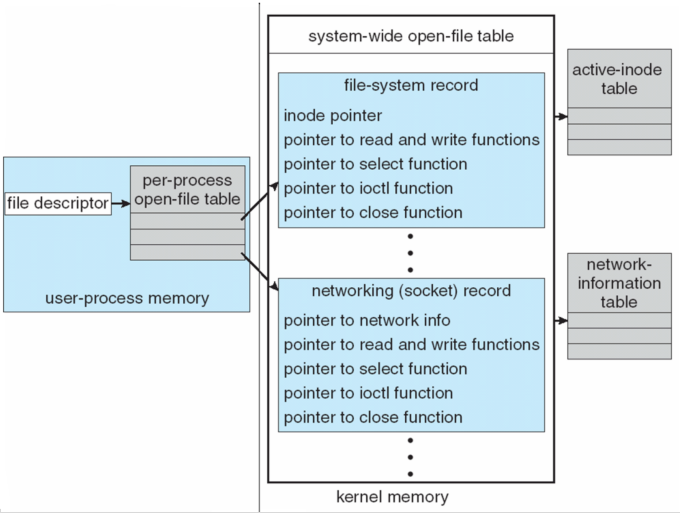


Using a UNIX File

- Opening a file creates a file descriptor fid
- Used as an index into a process-specific table of open files
- The corresponding table entry points to a system-wide file table
- Via buffered inode table, you finally get the data blocks

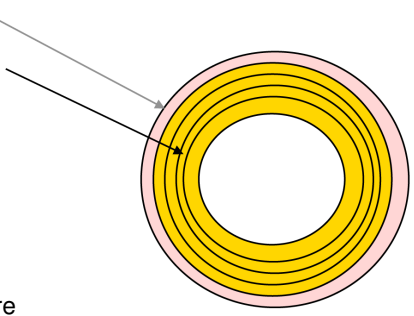


UNIX I/O Kernel Structure [SGG12]



Original UNIX File System

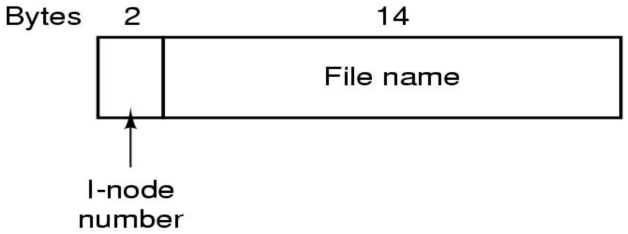
- Simple disk layout
  - Block size = sector size (512 bytes)
  - Inodes on outermost cylinders [1]
  - Data blocks on the inner cylinders
  - Freelist as a linked list
- Issues
  - Index is large
  - Fixed number of files
  - Inodes far away from data blocks
  - Inodes for directory not close together
  - Consecutive file blocks can be anywhere
  - Poor bandwidth for sequential access



[1] in very early UNIX FSs inode table in the midst of the cylinders

UNIX File Names

- Historically (Version 7) only 14 characters [TB15]



- System V up to 255 ASCII characters
  `<filename>.<extension>`

UNIX Inode

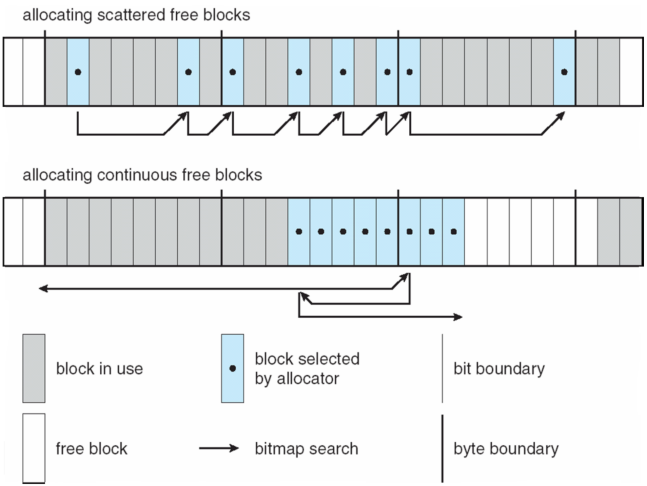
Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
UID	2	UID of the file owner
GID	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

BSD FFS (Fast File System)

- Use a larger block size: 4 KB
  - Allow large blocks to be chopped into 2,4 or 8 sub-blocks
  - Used for little files and pieces at the end of files
  - As the file grows, continue to allocate sub-blocks.
    If 4KB reached, copy sub-blocks to a full 4 KB block.
- Use bitmap instead of a free list
  - Try to allocate more contiguously
  - 10% free space reserve for system administrator
- Optimizations to better match disk characteristics
  - Cylinder groups for exploiting locality
  - Data and meta data in same cylinder group
  - Items of one directory in same or nearby cylinder groups

[ADAD18] chapter 41

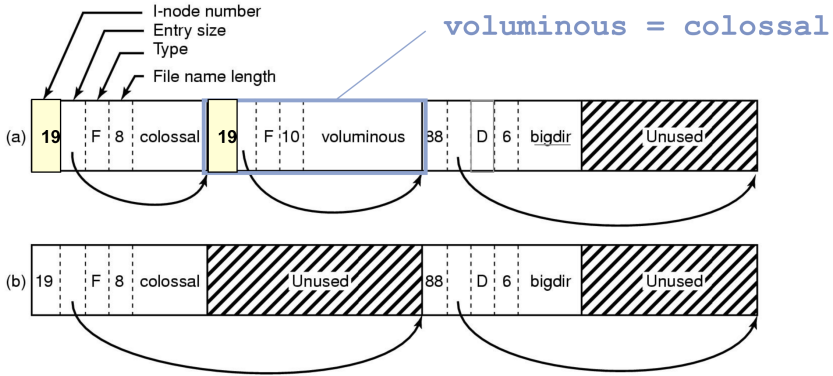
Ext2fs Block-Allocation Policies [SGG12]



FFS/Ext2 Directory (1)

- Directory entry needs three elements:
  - inode number (index to a table of inodes)
  - length of dir-entry (variable length of file names)
  - entry type
  - file name length
  - file name (up to 255 characters)
- Each directory contains at least two entries:
  - `..` = link to the parent directory (forming the directory tree)
  - `.` = link to itself
- FFS offers a “tree-like structure” (like Multics), supporting human preference, ordering hierarchically

Ext2 Directory Example [TB15]



- a) BSD directory tree entries (voluminous = hardlink to same file as colossal)
- b) Same directory after file voluminous has been removed

UNIX Directories

- Multiple directory entries may point to same inode (hard link) within the same file system
- Pathnames are used to identify files

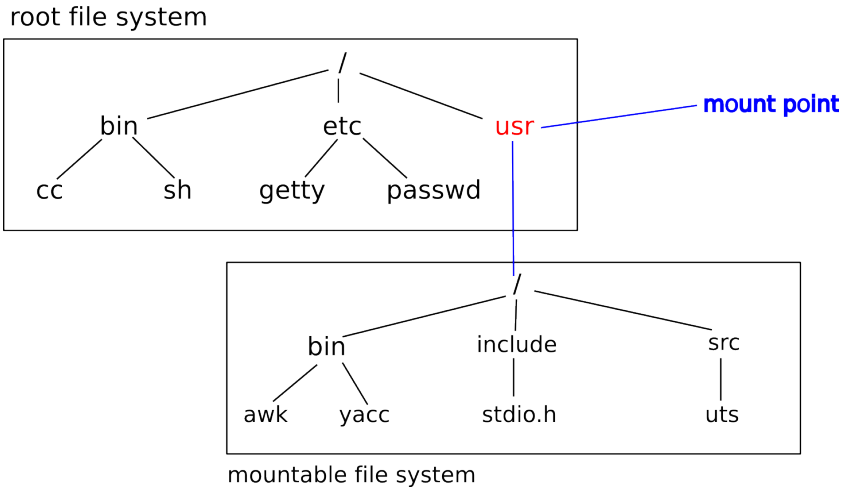
`/etc/passwd`	an absolute pathname
`../home/lief/examination`	a relative pathname

- Pathnames are resolved from left to right
- As long as it's not the last component of the pathname, the component name must be a directory
- With symbolic links you can address files and directories with different names. You can even define a symbolic link to a file currently not mounted (or even that never existed); i.e. a symbolic link is a file containing a pathname

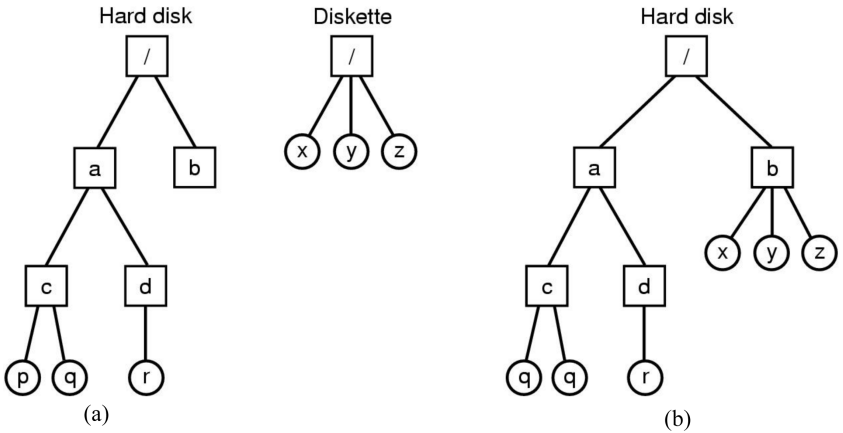
Hard Links $\leftrightarrow$ Symbolic Links

Hard link is another file name, i.e. $\exists$ another directory entry pointing to a specific file; its inode-field is the same in all hard links. Hard links are bound to the logical device (partition).
Each new hard link increases the link counter in file's i-node. As long as link counter $\neq$ 0, file remains existing after a `rm`. In all cases, a remove decreases link counter.
Symbolic link is a new file containing a pathname pointing to a file or to a directory. Symbolic links evaluated per access. If file or directory is removed the symbolic link points to nirvana.
You may even specify a symbolic link to a file or to a directory currently not present or even currently not existent.

Logical and Physical File System (1)



Mounting a File System [TB15]



(a) Before mounting

(b) After mounting

Logical and Physical File System (2)

- A logical file system can consist of different physical file systems
- A file system can be mounted at any place within another file system
- When accessing the “local root” of a mounted file system, a bit in its inode identifies this directory as a so-called mount point
- Using `mount` respectively `umount` the OS manages a so called mount table supporting the resolution of path name crossing file systems
- The only file system that has to be resident is the root file system (in general on a partition of a hard disk)

Journaling File Systems

- Journaling file systems record each update to the file system as a **transaction**
- All transactions are written to a **log**
  - A transaction is considered **committed** once it is written to the log
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed

Log-Structured File Systems

- Log-structured FS: use disk as a circular buffer
- Write all updates, including inodes, meta data and data to end of log
  - have all writes initially buffered in memory
  - periodically write these within 1 segment (1 MB)
  - when file opened, locate i-node, then find blocks
- From the other end, clear all data , no longer used

References

[ADAD18] R.H. Arpaci-Dusseau and A.C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2018.

retrieved from `http://pages.cs.wisc.edu/~remzi/OSTEP/`.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 4th edition, 2015.