File Systems

Željko Vrba
University of Oslo

(includes content by: Carsten Griwodz, Pål Halvorsen, Kai Li, Matija Pužar and Andrew Tannenbaum)

Files vs. disks

- Disks:
  - block oriented
  - physical addressing (legacy - CHS or sector numbers - LBA)
  - no protection among applications
  - crash recovery
- File system **abstraction**:
  - (most often) byte oriented files
  - provides logical naming
  - ensures proper sharing and concurrent access
  - more robust to crashes

File structures and types

- Structure: provided by the OS
- Unstructured files
  - sequence of bytes, uniterpreted by the OS
  - UNIX, Windows
- Structured:
  - record sequence, B-tree (key->value mapping)
  - MacOS (to some extent), MVS

- Type: interpretation of the content
- Multiple applications for single data type (e.g. audio, images, HTML, text, etc.)
- Can be interpreted by the OS (e.g. UNIX – directories, devices, etc.)

Unstructured files

- Most common today
  - Windows: text/binary distinction
- Other structures: application level
- Access type:
  - sequential read/write (mostly used with magnetic tapes)
  - Random access: seek to any position within a file
- Sparse files

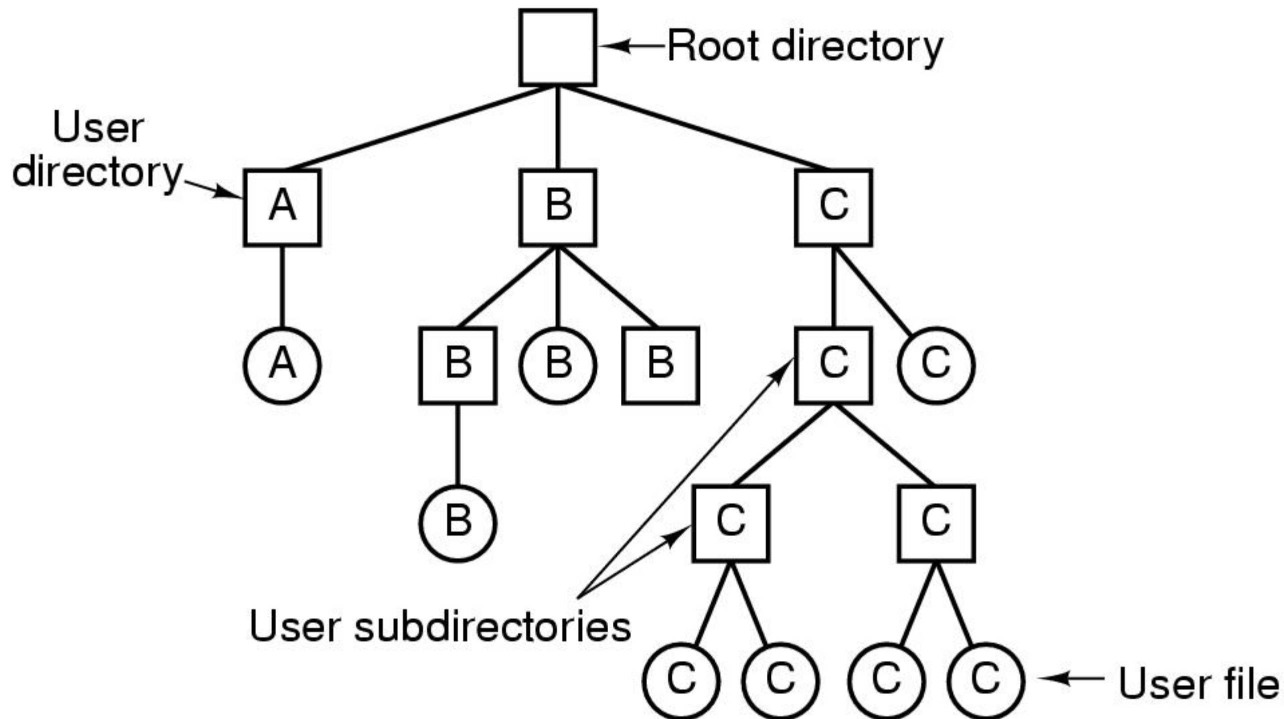
File metadata (attributes)

- standard: predefined by the OS, common for all files
- crucial for normal OS, file system and application operation
- owner, permissions, access times, size, ACL, compression, encryption, type (regular...), device numbers, etc...
- extended: arbitrary key -> value mapping attached to the file
- can be used e.g. for data indexing or advanced security schemes
- often specific for a single application
- namespace problem (unrelated app. using same keys)

Naming

- Each file has a **name** by which it is accessed
- Early FS: flat or single-level
- Modern FS:
  - files and directories
  - multiple-level tree hierarchy
  - links: single file with multiple names
  - files named by paths
  - the concept of current working directory
  - relative paths

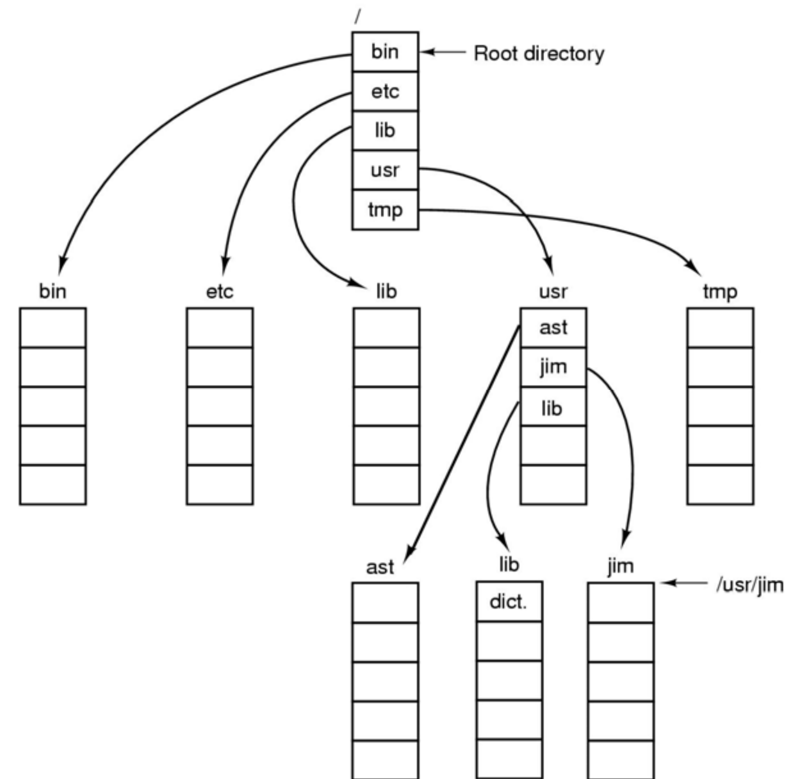
A hierarchical directory system



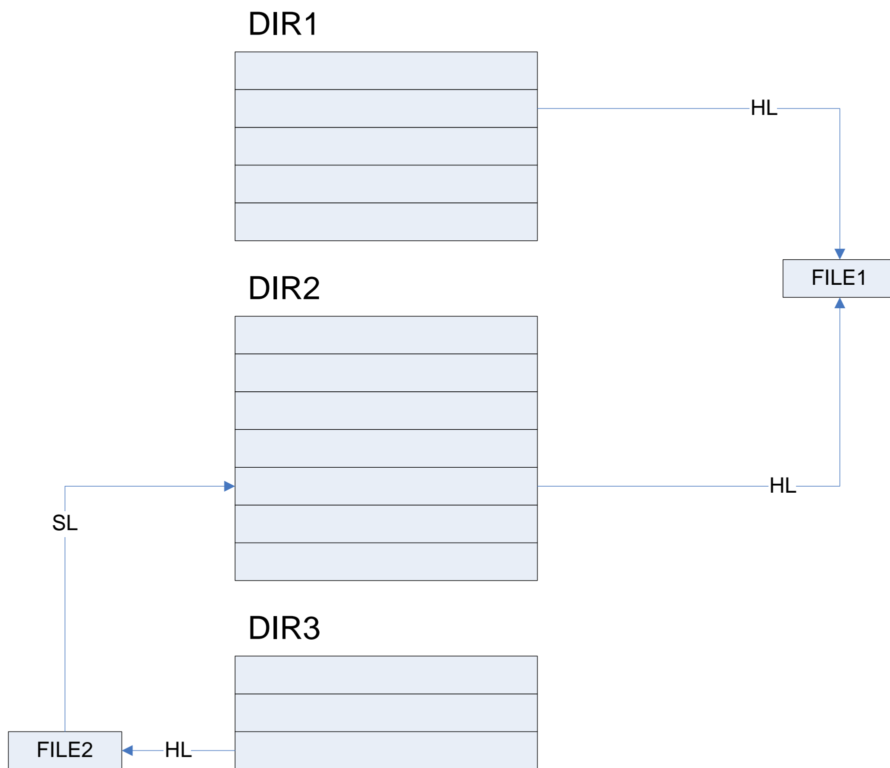
parent-child relationship of directories (. and .. entries)

Path names

- path is composed of files and directories
- file can be only the last component of a path
- multiple roots (e.g. Windows and drive letters; mounts provided too)
- actual syntax depends on the OS



Links



- Hard links
  - file and its “original” are indistinguishable
  - reference counting
- Symbolic links
  - “pointer” to another file
  - special type of file

File operations

- create, delete (unlink), rename, get/set attributes, create link
  - often do not require that a file is open (sometimes even impossible on open files)
- open
  - name+mode; returns a **file descriptor**
- close, read, write, append, seek, memory mapping
  - require open files
  - close mandatory
  - current file position

User-level API

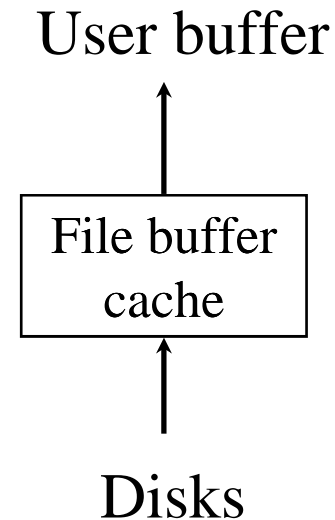
- Portable C <stdio.h>
  - fopen, fclose, fread, fwrite, fprintf, fscanf, fseek, ..
  - OS-independent, but limited in functionality
  - layer over the..
- ..OS native API
  - specific to each OS
  - POSIX and Win32 most widely used
  - POSIX: open, close, read, write, fcntl, lseek, ..
  - non-portable, but more features (sometimes essential, e.g. network communication)

Directory operations

- create, delete, rename
  - what if the directory is not empty?
- opendir, closedir, readdir
- link?

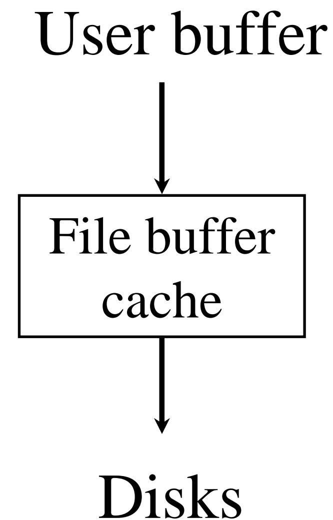
The buffer cache (1)

- read(fd, buf, n)
- find the disk block corresponding to current file position
- On a hit
  - copy from the buffer cache to a user buffer
- On a miss
  - replacement if necessary
  - read a file into the buffer cache



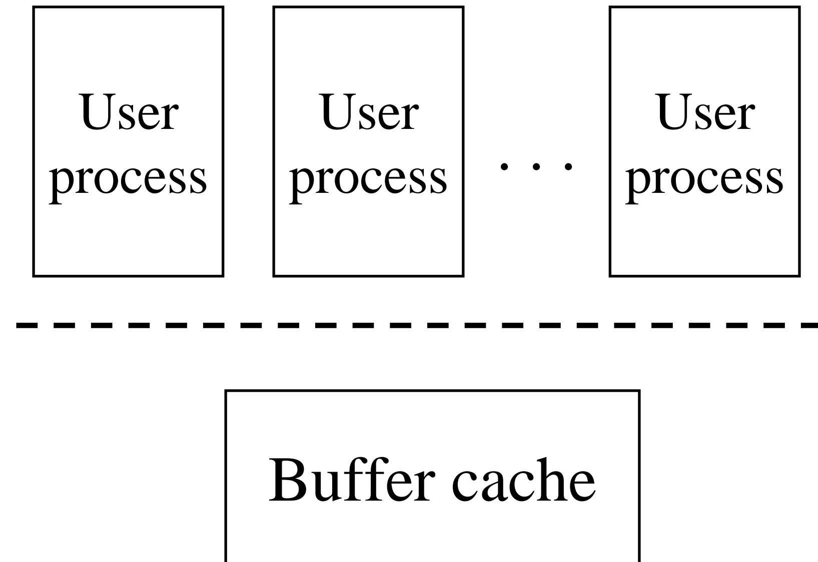
The buffer cache (2)

- write(fd, buffer, n)
- find the disk block corresponding to current file position
- On a hit
  - write to buffer cache
- On a miss
  - read the file to buffer cache if the file exists (possible replacement)
  - write to buffer cache
- When do you write the buffer cache to disk?
- In what order?



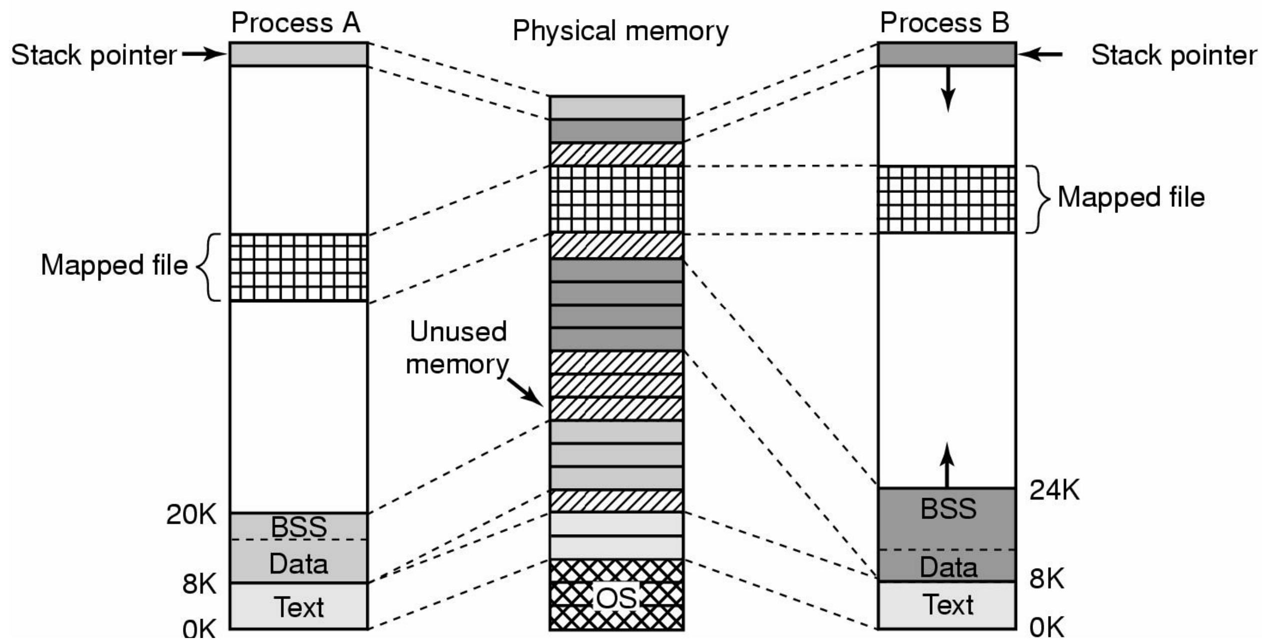
The buffer cache (3)

- in kernel
  - All processes share the same buffer cache
  - Global LRU
- if moved to user buffer:
  - duplications
  - pinning



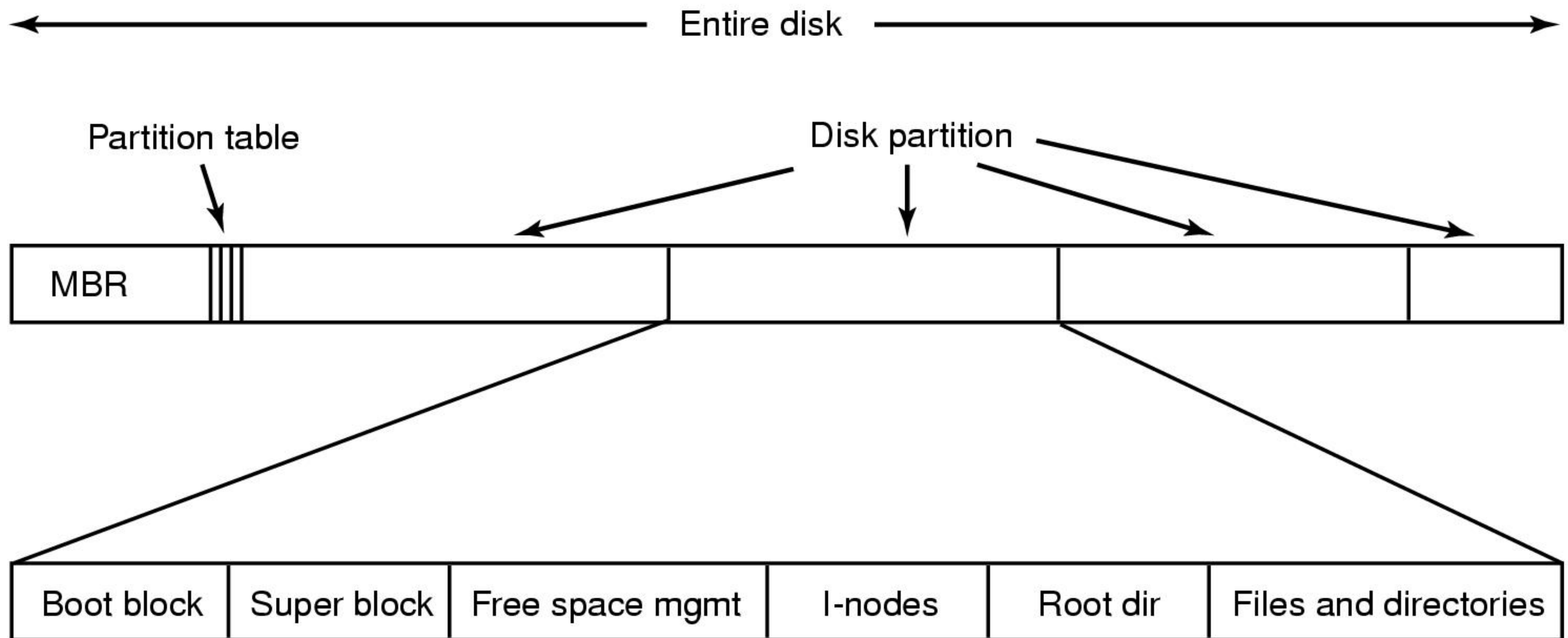
Memory mapping

file mapped into two different processes

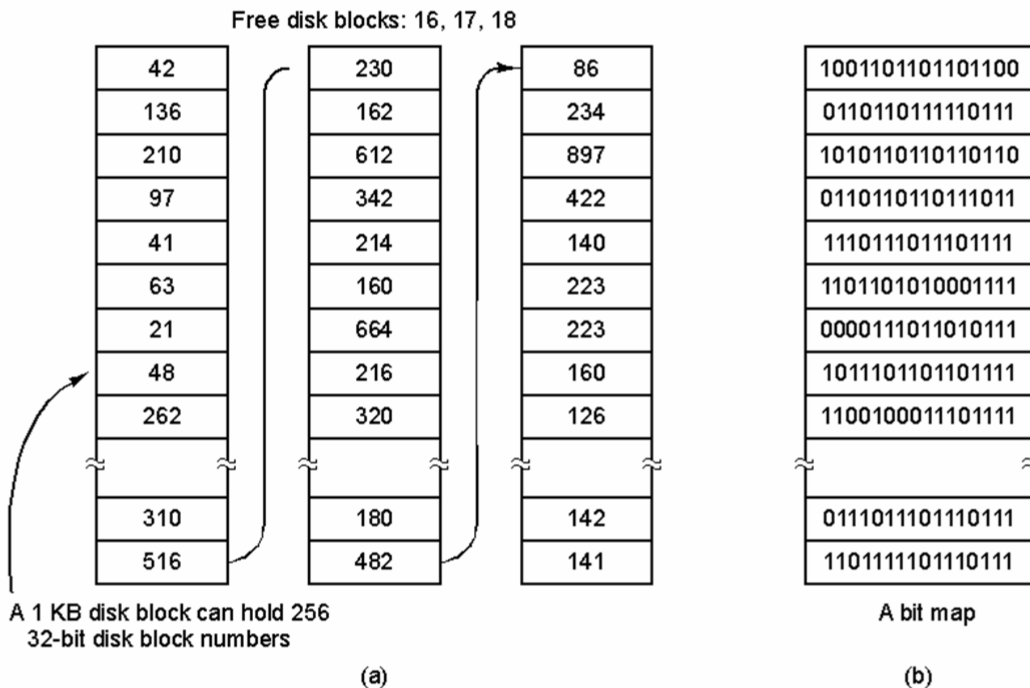


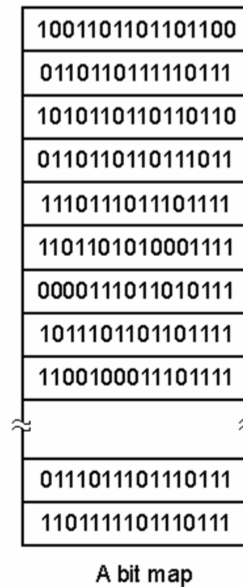
must maintain consistency with the buffer cache

On-disk layout



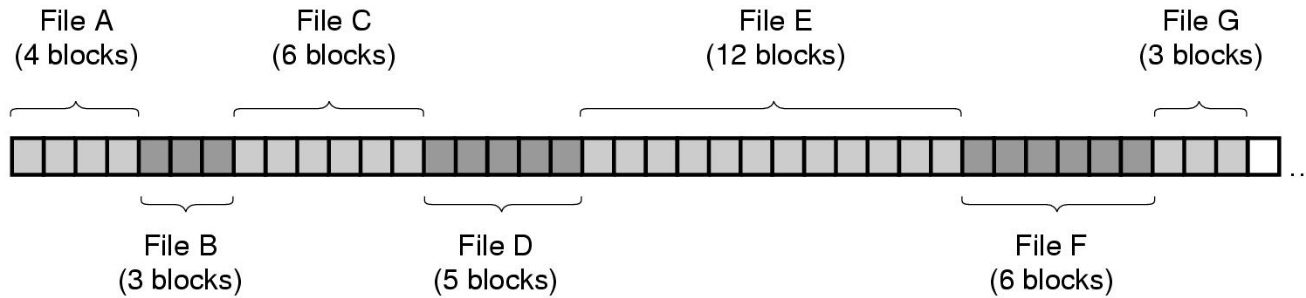
Free space management

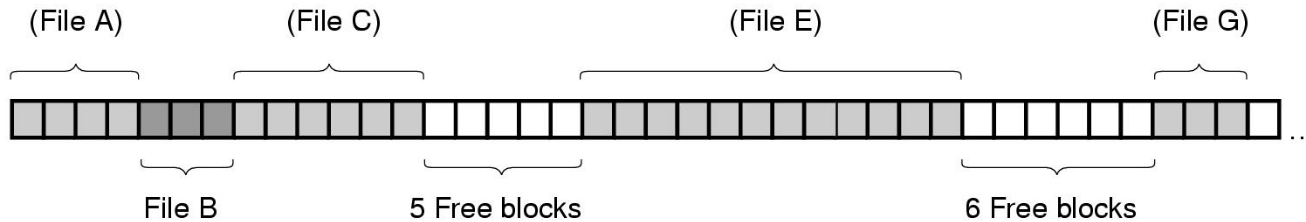




- units: file system blocks (different from disk sectors)
- free list
- bitmap: how large it needs to be for a whole disk, depending on the block size?

Implementing files: contiguous





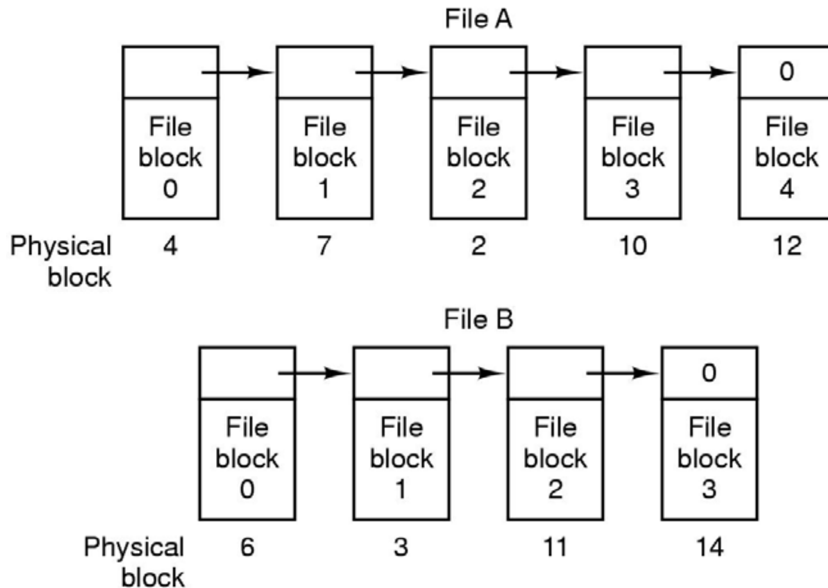
file header: start sector and # of sectors

(b): after removing D and E

pros: fast sequential and easy random access

cons: external fragmentation and hard to grow files

Implementing files: linked lists

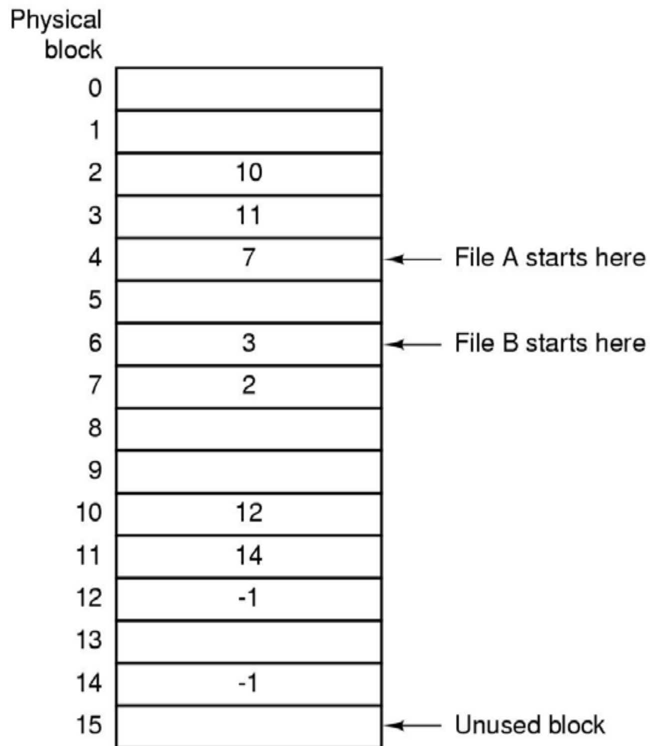


file header points to the first block of file; each block points to the next

pros: easy growing; free list similar to regular file

cons: horrible random access, unreliable (what happens if a block is lost?)

Implementing files: FAT



A: 4, 7, 2, 10, 12; B: 6, 11, 14

allocation for a single-block file?

A section of disk for each partition is reserved

One entry for each block

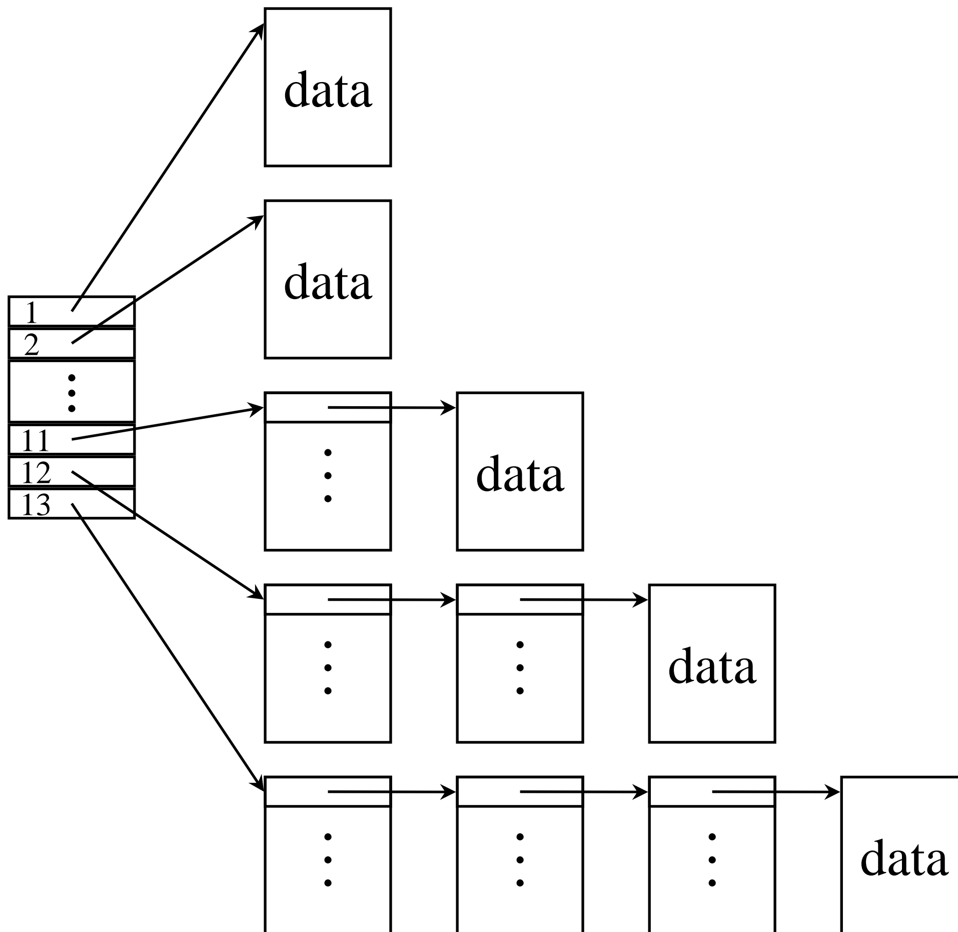
A file is a linked list of blocks

A directory entry points to the 1st block of the file

pros: simple

cons: always go to FAT; wasting space

Implementing files: inode



inode contains both the attributes and file block addresses

direct, 1-, 2-, and 3-level indirect blocks

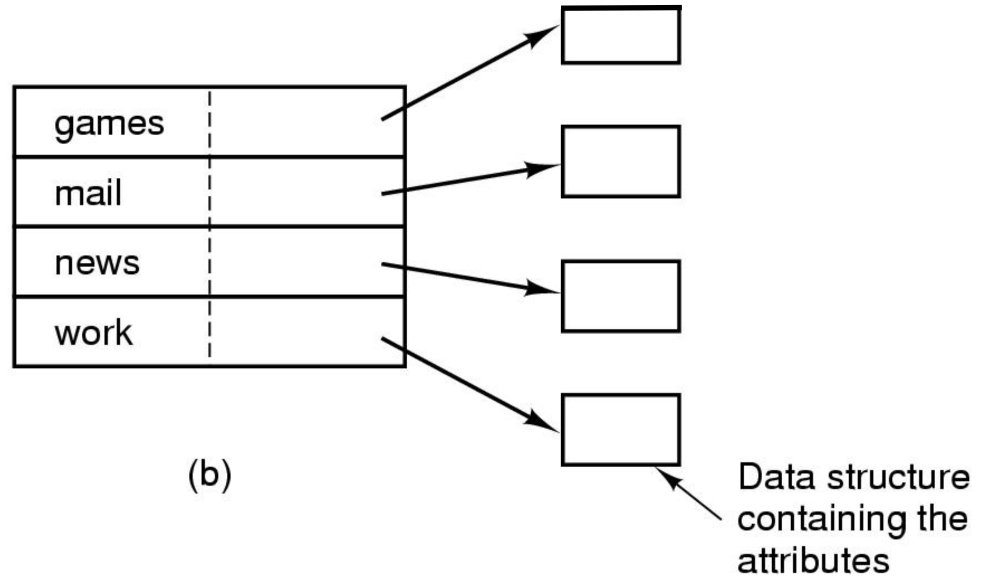
in favor of small files, can grow, lots of seeks for large files

maximum file size?

Implementing directories (1)

games	attributes
mail	attributes
news	attributes
work	attributes

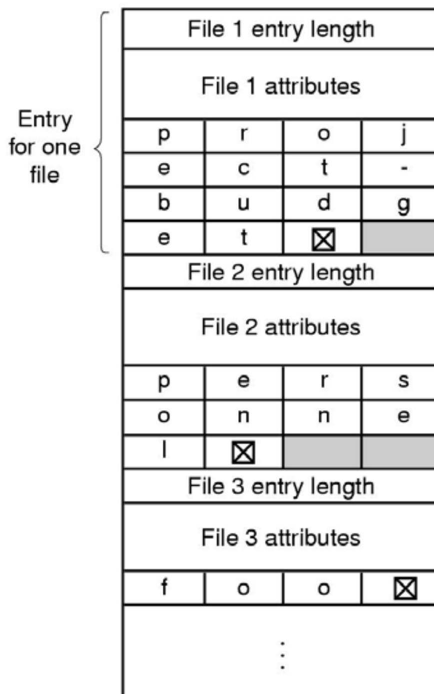
(a)

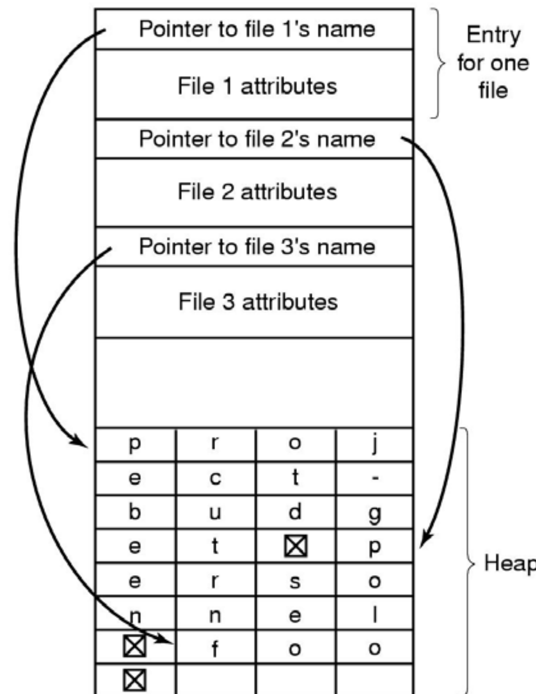


(a) fixed size-entries, inline attributes

(b) each entry refers to i-node which holds all attributes (incl. reference count)

Implementing directories (2)





Handling names:

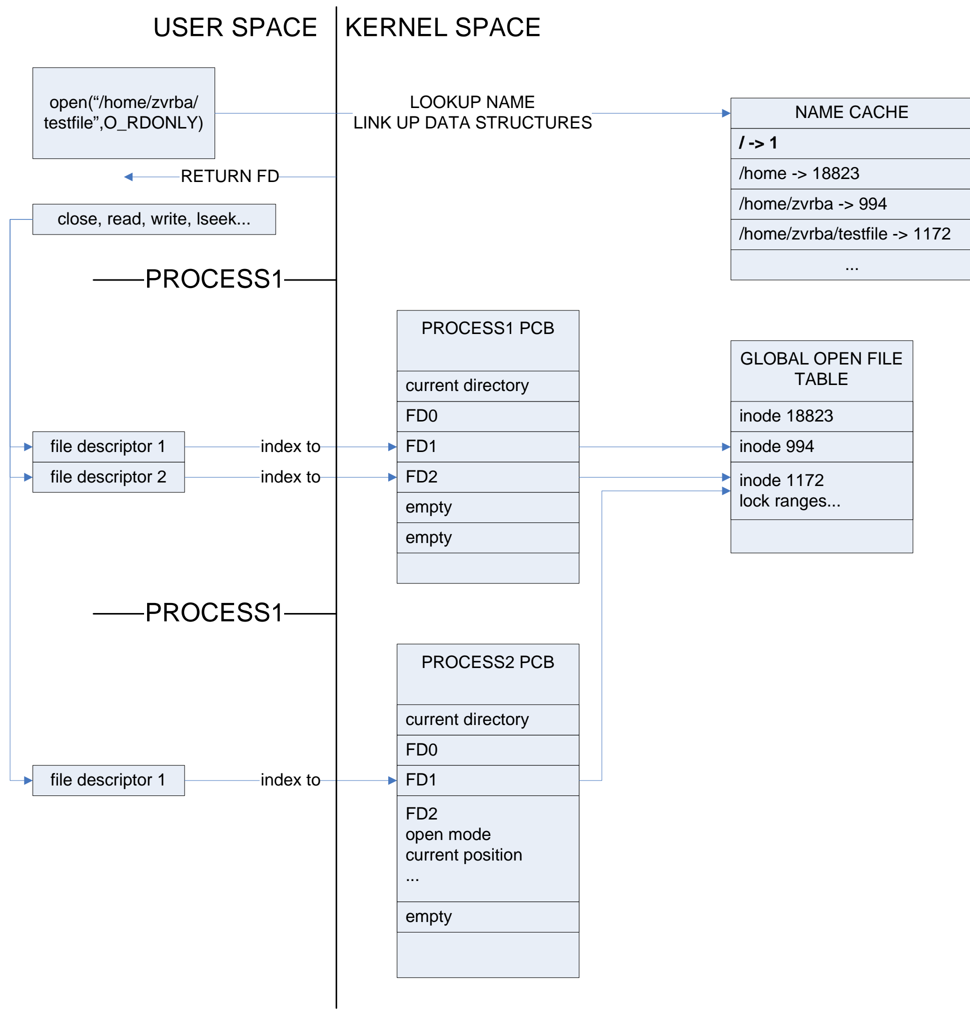
(a) in-line

(b) in a heap

Implementing directories (3)

- Association <name, inode>
- Linked lists: stored linearly in a special type of file
  - simple but slow for large directories (manual hashing schemes!)
- Trees (balanced); sorted by some criterion
  - efficient for large directories; complex; more space and may be slower for small # of files
- Hashing

Implementation: structures



- file descriptors are valid only within a single process
- name cache speeds up name lookup
- no duplicate inodes

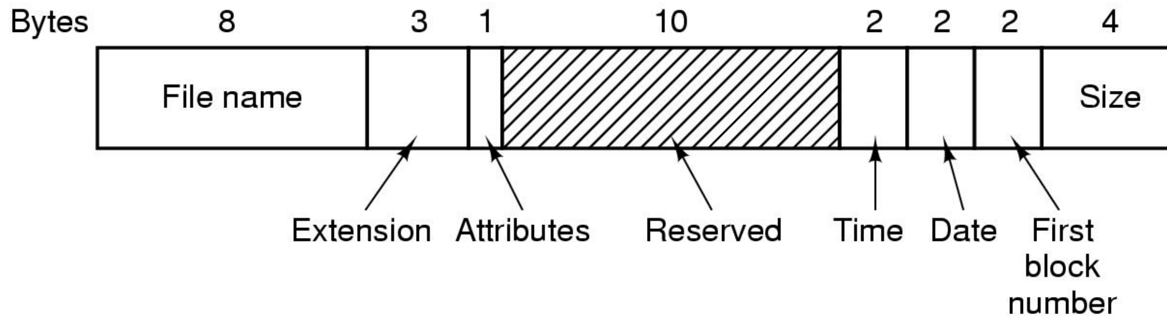
Implementation: VFS

- UNIX: the VFS layer
- object-oriented
- uniform support for multiple file systems

Example operations

- Create /home/zvrba/testfile
  - create directory entry and inode; ctime
- Write some bytes
  - find inode, write bytes; mtime
- Read some bytes
  - find inode, read bytes; atime
- What about just ./testfile in CWD?

The MS-DOS file system (FAT)



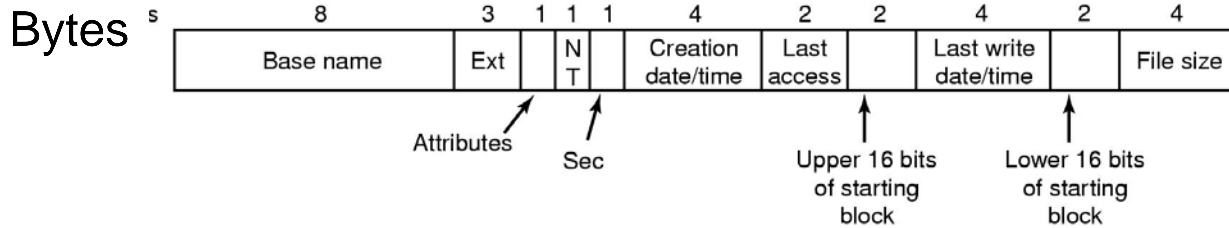
directory entry

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

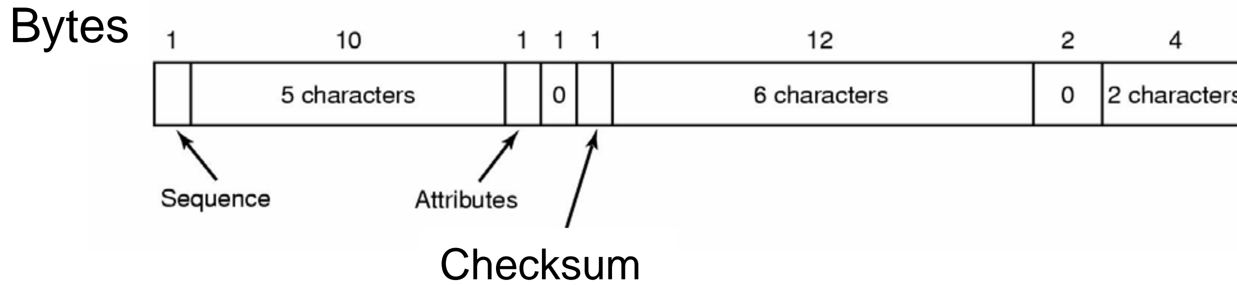
maximum partition for different block sizes (cluster)

empty: forbidden combination

Win98 file system (VFAT)



extended directory entry



entry for (a part) of long file name

68	d o g	A	0	CK		0	
3	o v e	A	0	CK	t h e l a	0	z y
2	w n f o	A	0	CK	x j u m p	0	s
1	T h e q	A	0	CK	u i c k b	0	r o
T	H E Q U I ~ 1	A	NT	S	Creation time / Last acc / Upp / Last write	Low	Size

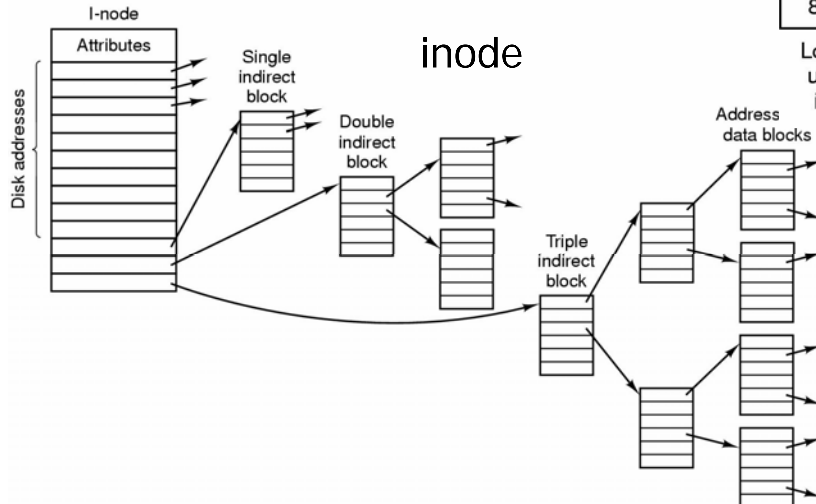
Bytes

how long file name is stored in a backward-compatible way

UNIX V7 file system

directory entry (max. 14 chars)

I-node number



inode

Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up usr yields i-node 6

I-node 6 is for /usr

Mode size times
132

I-node 6 says that /usr is in block 132

Block 132 is /usr directory

6	•
1	••
19	dick
30	erik
51	jim
26	ast
45	bal

/usr/ast is i-node 26

I-node 26 is for /usr/ast

Mode size times
406

I-node 26 says that /usr/ast is in block 406

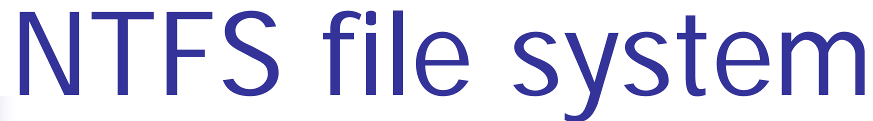
Block 406 is /usr/ast directory

26	•
6	••
64	grants
92	books
60	mbox
81	minix
17	src

/usr/ast/mbox is i-node 60

looking up /usr/ast/mbox

in UNIX, (almost) everything is a file!



NTFS file system

- uses 64-bit disk addressing; largest file 2^64 bytes
- unicode file names
- case-sensitive, but not fully supported by Win32 API
- file streams
- on-the-fly compression and encryption

- Motivation: slow system startup times after crash on large file systems
- Transactions and guaranteed consistency of data and/or metadata
- ReiserFS, XFS, JFS, NTFS
  - internally use balanced trees for directories