Operations:

- creating a file
  - free space management
- R/W file
  - file location (initially via file name)
  - current file pointer
- file search
  - customize current file pointer
- delete file
  - file location, free space management

Open file table

avoid multiple searches of file directors structure

contains only entries of opened files

index is returned as a pointer in open operation

handle pointer for all future I/O op. on the file.

multiprogramming environment

- per-process open file table
  - entry per opened file
  - every entry has a current-file pointer, and pointer to system wide table
- system wide table
  - contains info identical for all processes (location, size, data, protection info)
  - open file counter, number of refs to an entry (open++, close --, ==0 delete)

Memory mapped files

locking part of file in V-mem

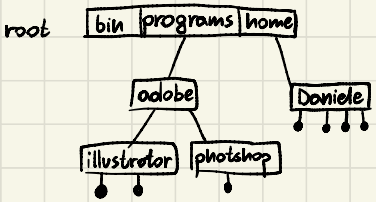
fast access, easy sharing via virtual page tables

File types -> most OS support

very small collection, recognisable through extension

Directors structure per partition

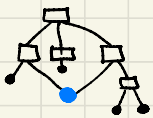
- Directors file per node
- entrs per sub directors and per file
- path = unique file name
- browse tree when opening files



MS-DOS -> first search in current directors, then search path

UNIX -> searches only path

Directors structure



fast access to files elseuhere unithout using a search path

symbolic links (the directors entry contains the actual path name)

no unique path nomes

watch out for closed paths

we hove to make some choices regarding the graph structure

- if we follow symbolic link and go back to parent directors where do we go? (shell dependent)
- can we delete file if there are references to it
  - can keep counter and only delete whe zero
  - hard links in UNIX keep counter, soft links do not guarantee existence of the file.

Implementation

- linear list:
  - always search whole list (even when creating for uniqueness)
  - delete -> mark entry as open or put the last in its place
- Binary tree (ordered list)
  - logarithmic search time
  - more work upon create/delete
- Hash function

  result points to linear list (many equal results)

Access and allocation methods

Access methods:

- sequential: read records from first to last, add info at the end
- direct or relative access: give number of logical record as a parameter

Contigous allocation

each file a number of consecutive blocks

fast sequential and direct access (low seek times)

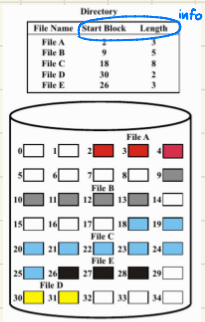
placement: first-fit, best-fit, next-fit, worst fit

analogous to dynamic partitioning

problem: Dinamically increase file size

internal fragmentation (often defragmentation routine available)

move data frequently



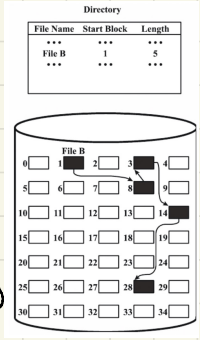
linked (chain) allocation

- each file forms linked list of random blocks
- fairly fast sequential access
- slow direct access
- positioning -> for each block we call free space algorithm

disadvantages:

- possibly slow access
- loose space for storing pointer

can be partially solved forming clusters (fixed num. of blocks)

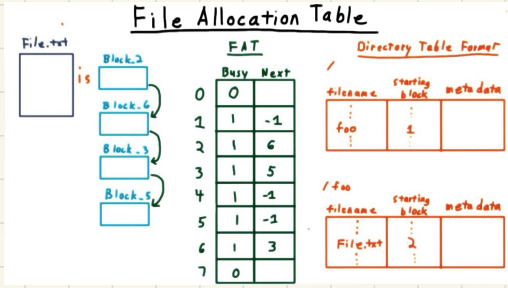


File allocation table (FAT)

- variant of linked allocation
- entry per disk block
  an entry contains a number of next block (same file)
  0 = block unused
- starting block number is indexed to FAT
- free space algo. selects first block with value 0

advantages:

- FAT is on start of disk, direct access really fast
- often part of FAT is in cache, so don't need to go back to it every time -> fast sequential access



Indexed Allocation

each file has number of random blocks + index block

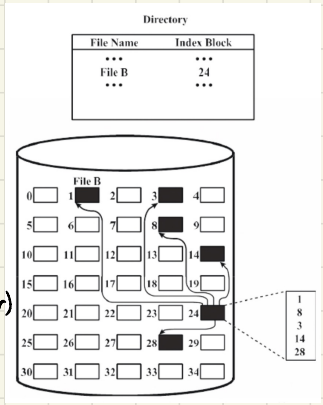
less fast sequential access

very fast direct access (2 pointers)

analogous to paging

the index should be at least one block, to support large files multiple blocks required (so that lists are shorter)

can do multilevel indexing



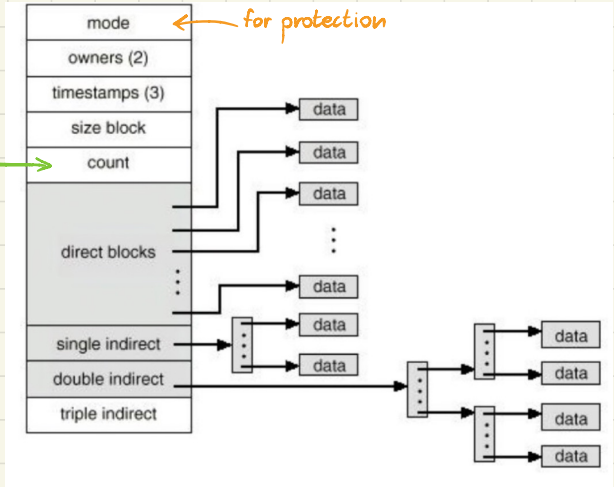
Unix inodes (index nodes)

inode per file/directory

directory only translates file name into inode number

placement of inodes is important

as close to data as possible



Free space management

- **Bit vector approach**
  - bit per block -> 1 is free
  - vector read word by word, if word≠0, the position of first 1 is determined

  becomes too long in modern systems
- **linked list approach**
  - very fast allocation
  - in file deletion -> add to back of the list
  - difficult to support continous allocation
  - in FAT it's integrated in the table
- **indexing approach**
  - index to free blocks
  - can do multilevel

Journaling

want to offer quick recovery of file system after a crash.

Used in Extz from Linux.

disk blocks divided into groups (inodes+data blocks)

each group contains two bitmap indicating usage of inodes and datablocks

consider a 1 data block file (1st pointer in inode points to data block)

adding a datablock takes 3 changes

1) Data written to data block
2) direct pointer from inode should point to the block
3) update data block usage bitmap

What if something goes wrong?

Data (1)	inode (2)	bitmap (3)	*Effect*
no	no	yes	Inconsistent
no	yes	no	Data loss
yes	no	no	Garbage nonsensical data
no	yes	yes	
yes	yes	no	Inconsistent
yes	no	yes	Inconsistent & Data loss

how can we prevent the errors?

specify on disk what changes we want to make before making them

do that in log or journal -> write ahead logging

1) write the 3 changes as a single transaction
2) once added to log, execute transaction
3) remove transaction from the log
4) if crash -> execute what's in log