National University of Singapore

CS2106 Operating System

Second Half Summary Notes

Dong Shaocong A0148008J

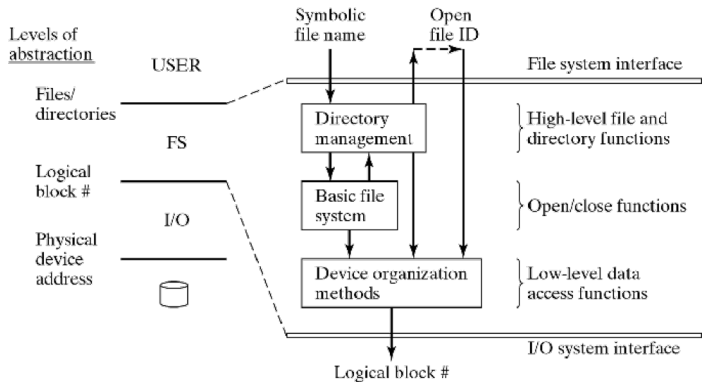
April 27, 2018

1 File System

Definition 1.1. File system

- Present logical (abstract) view of files and directories
  - Accessing a disk is very complicated: (2D or 3D structure, track/surface/sector, seek, rotation, ...)
  - Hide complexity of hardware devices
- Facilitate efficient use of storage devices: Optimise access e.g. to disk.
- Support sharing
  - Files persist even when owner/creator is not currently active (unlike main memory)
  - Key issue: Provide protection (control access)

Definition 1.2. Hierarchical View of File system

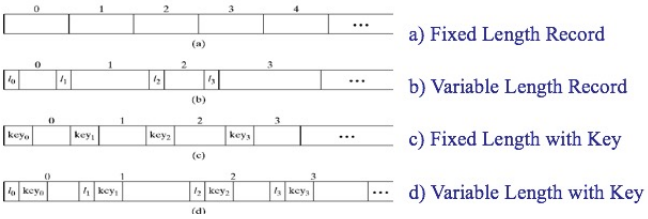


- **Directory management**: map logical name tounique Id, file descriptor
- **Basic file system**: open/close files
- **Physical device organization**: map file data to disk blocks

Definition 1.3. User-end view of File

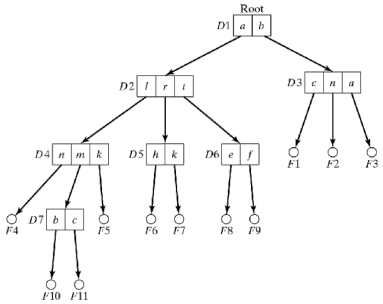
- **File name and type**
  - Valid name: number or characters, lower or upper cases, illegal characters ...
  - Extension: tied to type of file, used by applications
  - File type is recorded in header
    - Cannot be changed (even when extension changes)
    - Basic types: text, object, load file, directory

    - Application-specific types, e.g., .doc, .ps, .html
- **Logical file organization**
  - Most common: byte stream
  - Fixed-size or variable-size records
  - Addressed: **Implicitly** (sequential access to next record), or **Explicitly** by position (record#) or key

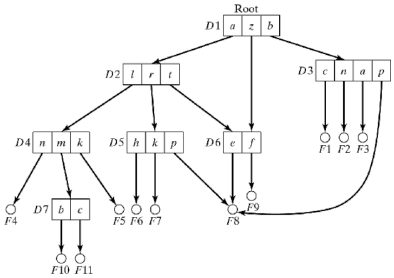


Definition 1.4. Directory Management

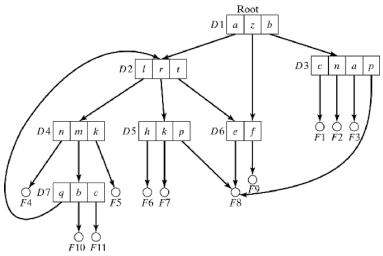
- **Main issues**:
  - **Shape** of the data structure
  - **What** info to keep about files
  - **Where** to keep the files in directory
  - **How** to organise entries for efficiency?
- **File directory data structure**:
  - **Tree-structured**
    - **Simple** search, insert, delete operations
    - Sharing is **asymmetric**



  - **DAG-structured**
    - **Symmetric** sharing
    - **Delete**: only last parent should remove files, so need **Reference count**

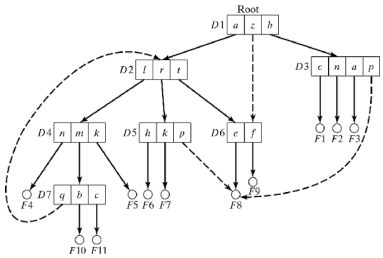


  - **DAG-structured with cycles**
    - Search is difficult with infinite loops
    - Deletion needs **garbage collection** (reference count not enough)



– **Symbolic links (shortcuts)**

  * **Compromise** to allow sharing but avoid cycles.
  * For **read/write**: symbolic link is the same as actual link.
  * For **deletion**: only symbolic link is deleted.



Definition 1.5. UNIX Hard Links

- Unix allows the same underlying file to have more than 1 filename, i.e. can create multiple references

```
% echo hello >test1
% ln test1 test2
% ls -li test[12]

5000 -rw-r--r-- 2 joe users 6 Apr 1 00:00 test1
5000 -rw-r--r-- 2 joe users 6 Apr 1 00:00 test2

% cat test2
hello
```

- Inode #5000 indicates that test1 & test2 refer to same file object

- newly created regular file only has one reference, e.g. 1 link
- any file object can have multiple filenames (pathnames) which are aliases created with a hard link (`link()` system call)

  Eg: any change to test1 is reflected in test2 since its the same underlying file object, if either test1/test2 is deleted the other file test2/test1 is not affected
- no file removal: reference removal with `unlink()` system call, delete in Unix is just `unlink`

  file object can only be freed if it has no more links
- hard links restricted to within the same filesystem on same device
- directory entry `..` is a hard link (to parent)
- can only link non-directories with `link()` (except for superuser)

Definition 1.6. Symbolic Links

- Looks like hard link but NOT the same!
- Creating symbolic link

```
% ln -s newpath file
% ls -l file
```

file -> newpath

  is an indirection giving a new pathname which can be resolved (again if another symbolic link)

```
% ls -l /etc/X11
  lrwxrwxrwx 1 root root 14 Jan 1 04 X11 ->
  /var/X11R6/lib
```

  indirection continues if another symbolic link, repeats until final pathname is not a symbolic link (kernel has limit on # indirections, ELOOP open error)
- Symbolic link: special file where data is another pathname (absolute or relative), new pathname may itself be another symbolic link!

- provides a reference to pathname (hard link provides reference to underlying file object)
- does not correspond to making a new edge in graph like a hard link – rather another pathname (which may exist or not!)
  - Can link to non-existent file

```
% ln -s nosuchfile testfile
% ls
testfile
% cat testfile
cat: cannot open testfile
```

- can link directories (user created hard links restricted to regular files) – so can create general graphs + loops!

Remark. Windows Shortcuts

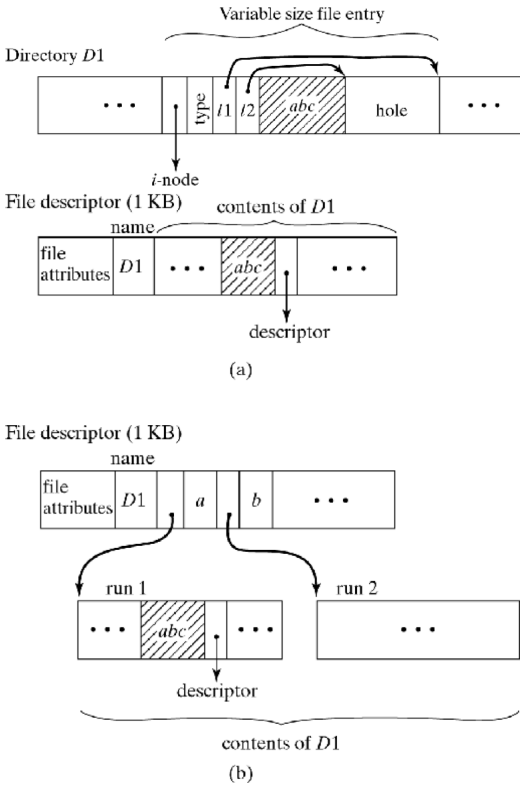
- Similar to symbolic links but does not re-expand further
- Only understood by GUI shell! Created also from GUI shell
- Win NTFS has hard links, symbolic links using junction mechanism (not normally used)

Remark. File Directories - Path name

- Concatenated local names with delimiter (. or / or \ )
- **Absolute** path name: start with root (/)
- **Relative** path name: start with current directory (.)
- Notation to move upward in hierarchy (..)

Method 1.1. Implementation of Directories

- **What information to keep in each entry**
  - **All** descriptive information, directory can become very large, searches are difficult / slow.
  - Only symbolic **name and pointer** to descriptor
    - Needs an extra disk access to descriptor
    - Variable name length
- **How to organise entries within directory**
  - **Fixed-size** entries: use array of slots
  - **Variable-size** entries: use linked list
  - **Size of directory**: fixed or expanding
- **Example**: Windows 2000: When # of entries exceeds directory size, expand using $B^+ - tree$.



Definition 1.7. File Descriptor Owner id; File type (whether it's a folder, a file, a symbolic link etc.); Protection information; Mapping to physical disk blocks (**TOC - table of contents**); Time of creation, last use, last modification; Reference counter.

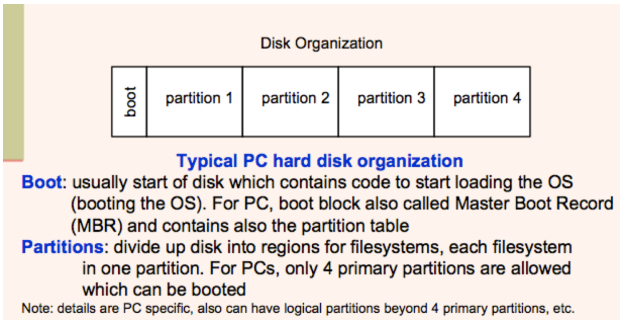
Definition 1.8. OFT - Open file table keeps track of currently open files. **OFT entries**: current position (read to which part); information from descriptor (file length, disk location); pointers to allocated buffers. **Two types**: system OFT (system-wide), process OFT (open files by a process)

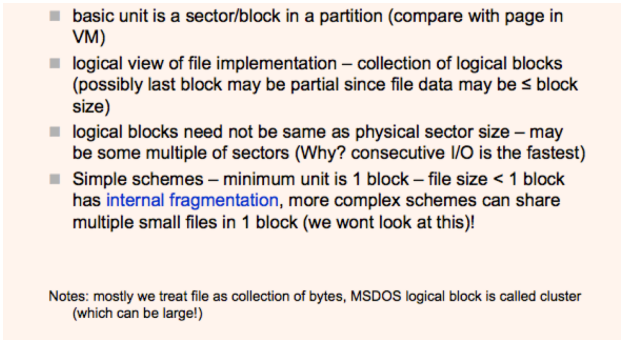
Definition 1.9. Basic File System Operations

- **open file** FILE *fopen(const char *filename, const char *mode)
  1. Search directory for given file and verify access rights
  2. Allocate and fill in OFT entries and read/write buffers
  3. Return OFT index
- **close command** int fclose(FILE *stream)
  1. Flush modified buffers to disk and release buffers
  2. Update file descriptor (file length, disk location, usage information)
  3. Free OFT entry

- **read command** size_t fread(void *ptr, size_t size (of element), size_t nmemb (number of elements), FILE *stream)
  - assume file is open for **sequential** access
  - **Buffered read**: current block kept in r/w buffer
    1. copy from buffer to memory until desired count or end of file is reached: - update current position, return status
    2. copy from buffer to memory until end of buffer is reached: write the buffer to disk if modified; read the next block; continue copying
  - **Unbuffered read**: read the entire block containing the needed data from disk
- **write command (buffered)** size_t fwrite(const void *ptr, size_t size, size_t number, FILE *stream)
  1. Write into buffer,
  2. When full, write buffer to disk. If next block does not exist (file is expanding): - allocate new block -update file descriptor - update bit map (free space on disk).
  3. Update file length in descriptor
- **seek command** int fseek(FILE *stream, long int offset, int whence)
  1. Set current position as specified by parameter
  2. Read block containing current position into buffer
- **rewind command**: analogous to seek but position is zero

Definition 1.10. Organising data on a disk



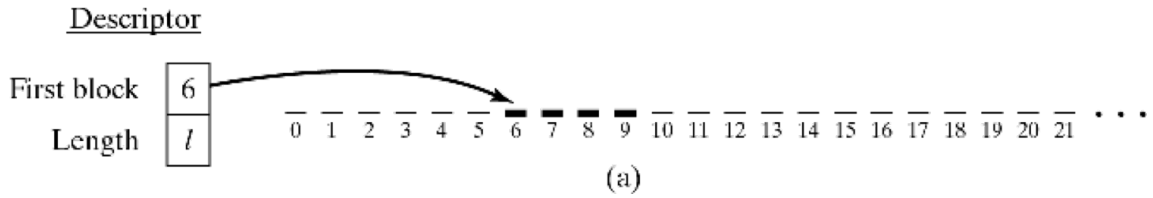


Definition 1.11. Fragmentation

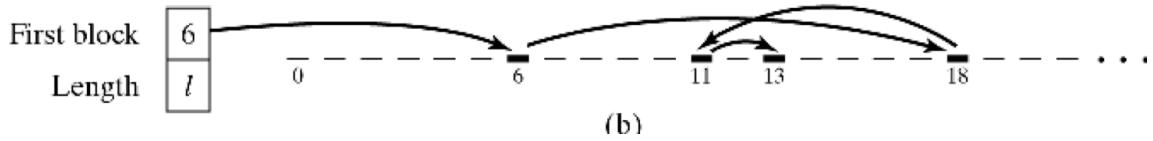
- **Internal fragmentation**: Due to the rules governing memory allocation, more computer memory is sometimes allocated than is needed. Any process, no matter how small, occupies an entire partition. This waste is called internal fragmentation. (unused memory within the allocated region)
- **External fragmentation**: External fragmentation arises when free memory is separated into small blocks and is interspersed by allocated memory. (unusable storage is outside the allocated regions.)

Definition 1.12. Data structures on Disk Note: need to record which block belongs to which part of files. Data structure must also be stored on disk.

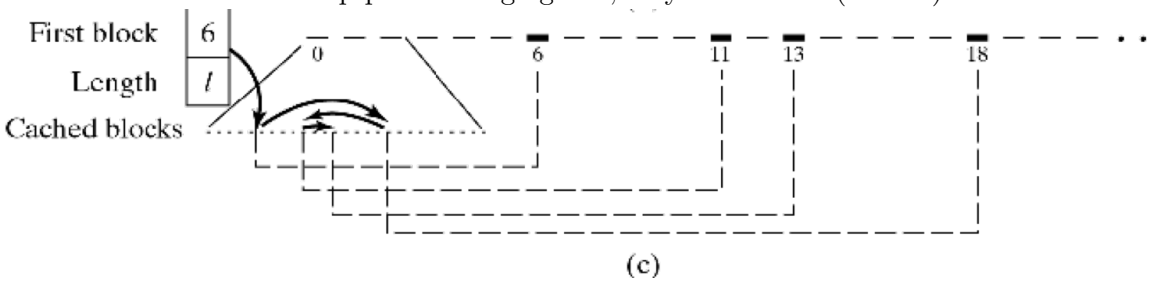
- **Contiguous organization**
  1. **Advantages**: Simple implementation; Fast sequential access (minimal arm movement)
  2. **Disadvantages**: Insert/delete is difficult; How much space to allocate initially? External fragmentation



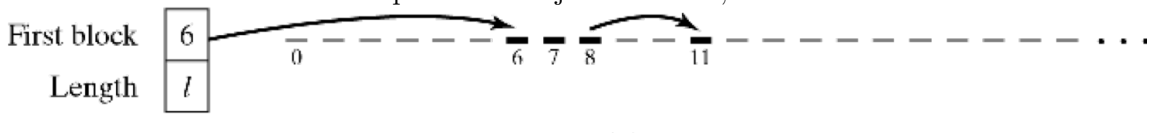
- **Linked organization**
  1. **Advantages**: Simple insert/delete, no external fragmentation
  2. **Disadvantages**: Sequential access less efficient (seek latency); Direct access not possible; Poor reliability (when chain breaks)



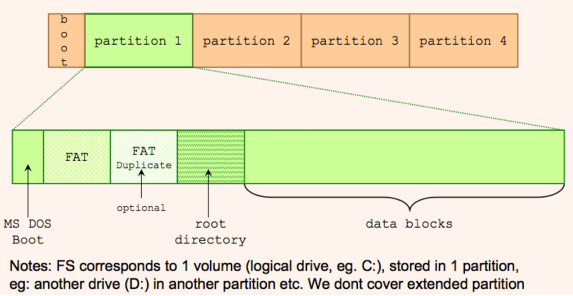
  3. **Linked Variation 1**: keep pointers segregated, may be cached. (FAT32)



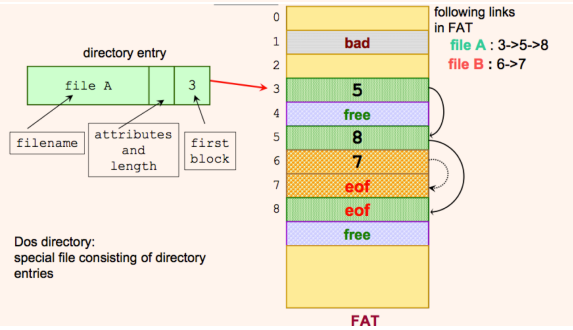
  4. **Linked Variation 2**: Link sequences of adjacent blocks, rather than individual blocks



- MSDOS uses File Allocation Table (FAT)
- Linked allocation but stored completely in FAT (after reading from disk)
- FAT kept in **RAM** (stored in disk but duplicated in RAM) – gives fast access to the pointers
- FAT table contains either: **block number** of next block, **EOF** code (corresponds to NULL pointer), **FREE** code (block is unused), **BAD** block (block is unusable, i.e. disk error) – combines bitmap for free blocks with linked allocation for list of blocks
- FAT table is 1 entry for every block. Space management becomes an array method (but bigger than bitmap)



- special file (type directory) containing directory entries (32 byte structures, little endian)
- **fat directory entry**: **filename + extension** (8+3 bytes), file **attributes** [1 byte: readonly, hidden, system, **directory flag** (distinguish directories from normal files), archive, volume label (disk volume name is dir entry)], time+date created, last access (read/write) date, time+date last write (creation is write), **first block** (**cluster**) **number**
- root directory is special (already known, FAT16 has limited root dir size, 512 entries), other directories distinguished by type



- Linked list implemented as array of FAT entries:
  4 types: block number (part of a linked list), EOF type (end of linked list), special FREE type, special BAD type – some numbers are block numbers, some are reserved block numbers for EOF,FREE,BAD
- Blocks for file linked together in FAT entries, eg. file A: 3 -> 5 -> 8
- Free blocks indicated by FREE entry
- Bad blocks (unusable blocks due to disk error) marked in FAT entry: BAD cluster value

How MSDOS deletes file/directory:

- Set first letter in filename to 0xE5 (destroys first byte of original filename)
- Free data blocks: set FAT entries in link list to FREE (tags all the data blocks in file free)

Can attempt to **undelete** – some information lost but can guess!

- FAT16: fat entry block # is **16 bits** (16 bit numbers in FAT entries), sector size=512, how to deal with disks bigger than 64K*512 (32M)
- Logical block size = multiple of sectors. MSDOS calls this the **cluster size**
- Maximum cluster size = 32K (normally)
- Maximum file system size for FAT16 = 64K*32K = 2G
- Maximum file size: slightly less than 2G
- large cluster size means large internal fragmentation!

Note: can use 64K cluster size, so limits become 4G

- VFAT: adds long filenames (255 chars, introduced in Win95)
  - compatible with FAT16 by tricking it!
  - short FAT16 filename for FAT16 and long version, short filename created from long one (e.g. `longna~1.doc`)
  - for compatability: long name stored as multiple directory entries using illegal attributes (not used by FAT16)
  - trick: have to manage 2 kinds of names, old SW uses short names, aliasing issue due to 2 names

- Increase FAT size to 28 bits, cluster numbers 28-bits
- max filesystem size increased: Win98 ~127G limit, Win2K can only format up to 32G limit
- decreases internal fragmentation (cluster size can decrease)
- FAT table size increases
- FAT16 root directory size limit removed – normal directory
- Maximum file size 2^32 - 1

Notes: compare with direct paging, changing size of page and effect on page table given different sized virtual addresses

Fast disk access:

- contiguous blocks (from geometry/processing viewpoint)
- blocks in same cylinder

Suppose currently FS is optimally allocated (fresh install). Is this sustainable?

1. Delete files/blocks
2. Insert new files/blocks

After some operations – block ordering becomes more **random**!

Disk Fragmentation: logical contiguous blocks are **"far apart"** on disk (this is different from memory fragmentation)

Notes: internal fragmentation still exists from block size

FAT:

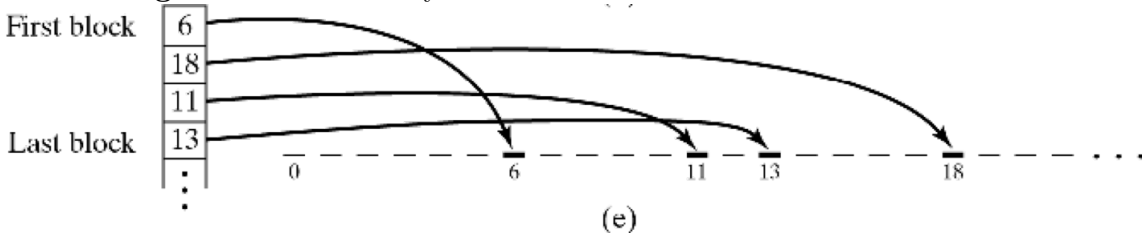
- affects FAT FS
- fragmentation effect less with large cluster size (but **large** internal fragmentation!)
- MSDOS solution: run defragmentation (like compaction) on entire FS – move all used blocks to be contiguous .. one big free space chunk after defragmentation. May take a long time to defrag! (Windows: Disk Defragmenter)

Unix S5FS:

- also has disk fragmentation
- may be worse than DOS since smaller logical block size

Alternative (not covered): FS with **fragmentation resistance** (not necessarily optimal but dont need to defragment).

- **Indexed organization**
  1. **Index table**: sequential list of records; **implementation**: keep index list in descriptor
  2. **Advantage**: Insert/delete is easy; Sequential and direct access is efficient
  3. **Disadvantage**: file size limited by number of index entries



(e)

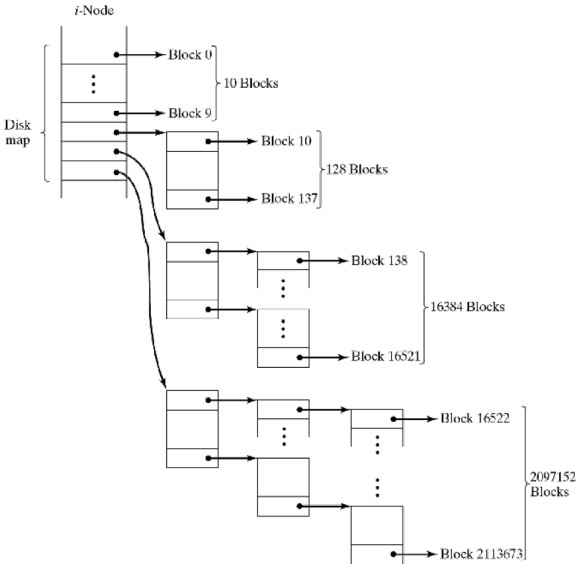
Block 6 contains bytes 0-1023, block 18 contains bytes 1024-2047, etc (1024 bytes block).

  4. **Multi-level index hierarchy**: Primary index points to secondary indices **Problem**: number of disk accesses increases with depth of hierarchy.
  5. **Incremental indexing**: number of disk accesses increases with depth of hierarchy; When insufficient, allocate additional index levels; **Example**: Unix, 3-level expansion.

- Look at System V File System (s5fs) – simpler than modern FS implementations (rather old but some of the design remains in current implementations)
- inodes: represents every file
- directories: contain names of files (recall maps filename to eventual file (hard link) or pathname (symbolic link))
- file allocation using a multi-level tree index

Note: s5fs is the original Unix v7 FS. It's much simpler and less sophisticated than more modern Unix FS.

- actual file object
- every file has one inode (many to one mapping because of hard links)
- contains all meta-data about file **except filename**
- contains reference count i.e. # hard links, reference count = 0 means file can be deleted [subjected to no open files condition - all file descriptors to file object are closed]
- meta-data in inode includes Table of Contents (TOC) which gives mapping of file data to disk blocks (TOC is per file - contrast with MSDOS which has only **global** TOC (FAT))



Incremental indexing in Unix (file size vs. speed of access)

- blocks 0-9: 1 access (direct).
- blocks 10-137: 2 access.
- blocks 138-16521: 3 access. etc.

- Direct block pointers:
  used for small files, no extra disk overhead, efficient direct access. VM analogy: TLB (however not a cache)
- Single indirect block:
  files bigger than direct blocks & smaller than double indirect. Disk overhead is 1 block. File access slightly slower than direct. VM analogy: direct mapped page table
- double + triple indirect blocks:
  files bigger than single indirect block (for sufficiently big block size, triple indirect usually not needed) . More disk overhead but is small fraction of file size. Random file access requires looking up the indirection blocks – slower than indirect. VM analog: 2-3 level page tables

File sizes: direct << single indirect << double indirect << triple indirect

- # of direct blocks (eg. s5fs: 10, ext2: 12)
- # indirection levels (eg. max is 2 or 3)
- logical block size - determines efficiency of disk I/O, can be composed of several contiguous physical blocks, space wastage from internal fragmentation, determines # block pointers in index blocks and max indirection levels (eg. ext2 can select 1,2,4K when creating filesystem)
- block pointer size - affects indexing, determines max addressable disk block)

- small files only use the direct blocks in TOC, number of direct blocks can vary
- larger file requires first indirect block (this is like 1-level page table)
- even larger file uses double indirect block which points to indirect blocks (similar to 2-level page table). Even larger may have triple indirect (but the number of indirection levels may vary)
- like page tables, can allow blocks which do not exist (**logical zero filled holes** in the file) – set the block pointer to NULL in the TOC, return zeroes for the logical block when read

- a file of type directory – accessing directory is like any other file
- only special directory operations allowed (read directory, link + unlink filenames)
- s5fs directory is array of 16 byte entries (inode: 16 bits, filename: 14 bytes), simplifies directory management (note dont need null pointer if filename length = 14)
- deleted file has inode 0
- note: most modern implementations allow long file names (ext2: 255-byte filenames), so space issues like in memory allocation

Inode Number	Filename
25	.
71	..
100	file1
200	file2

- Create new file: new directory entry with new inode
- Hard link: new directory entry (in appropriate dir) with inode of the linked file: eg. `link(path1, path2)`
  i1 = inode for file with path1;
  let path2 = dir2/file2; // dont allow linking directories
  add new directory entry to dir2: [i1, file2]; // assume file2 // not there
- Deleting (deleting is unlink since graph is DAG): remove directory entry, decrement inode link count, free file object when link count = 0 (plus open fd's condition)

Definition 1.13. Free storage space management

- **Linked list organization**
  1. Linking **individual** blocks: inefficient; no blocks clustering to minimise seek operations; groups of blocks are allocated/released one at a time.
  2. Better: Link **groups** of consecutive blocks
- **Bit map organization** (residing in super block)
  1. Analogous to main memory
  2. A single bit per block indicating if free or occupied