Operating Systems — File System

INF107

Stefano Zacchiroli

2023

File-System Interface

File

- Computers have varied forms of persistent, secondary storage: NVM, HDD, tapes, etc.
- OSes provide a **unified view** over secondary storage, generally based on the notion of *file*

File

- A file is a **named collection of related information** that is recorded on secondary storage
- From a user's perspective, a file is the *smallest allotment of logical secondary storage*
  - That is, data cannot be written to secondary storage unless they are within a file

- Files can contain different kinds of information
  - OS point of view, at least two types: data, executable programs
  - User point of view: text, numeric, multimedia, etc.
- A **filesystem** (or *file system*, or *FS*) is the part of an OS responsible for:
  - Providing an abstract view of files to users and programs
  - Translate file operations to I/O operations on mass storage devices

File Attributes

- In addition to their *content*, files are associated to several **files attributes**, such as:
  - **Name** – only information kept in human-readable form
  - **Identifier** – unique tag (number) identifies file within file system
  - **Type** – needed for systems that support different file types
  - **Location (physical)** – pointer to file location on device
  - **Size** – current file size
  - **Protection** – controls who can do reading, writing, executing
  - **Time, date, and user identification** – data for protection, security, and usage monitoring
- Stored in the **File Control Block (FCB)** of each file
- Many variations across OSes, including extended file attributes such as file checksum

File Attributes

- In addition to their *content*, files are associated to several **files attributes**, such as:
  - **Name** – only information kept in human-readable form
  - **Identifier** – unique tag (number) identifies file within file system
  - **Type** – needed for systems that support different file types
  - **Location (physical)** – pointer to file location on device
  - **Size** – current file size
  - **Protection** – controls who can do reading, writing, executing
  - **Time, date, and user identification** – data for protection, security, and usage monitoring
- Stored in the **File Control Block (FCB)** of each file
- Many variations across OSes, including extended file attributes such as file checksum

```
$ stat Makefile    # Example using the UNIX stat command on the Makefile used to build these slides
  File: Makefile
  Size: 758         Blocks: 8          IO Block: 4096   regular file
Device: 254,1   Inode: 1582010     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/    zack)   Gid: ( 1000/    zack)
Access: 2023-09-10 16:25:33.265416874 +0200
Modify: 2023-09-07 09:48:55.986798677 +0200
Change: 2023-09-07 09:48:55.986798677 +0200
 Birth: 2023-09-07 09:48:55.986798677 +0200
```

File Operation

- We can see a file as an *abstract data type* (ADT), fully determined by the operations it supports
- Common **file operations** that OS provides—usually as a set of corresponding *system calls*—are:
  1. *Create*
  2. *Write* — at **write pointer location**
  3. *Read* — at **read pointer location** (can be the same as write pointer location)
  4. *Reposition* within file — or **seek**
  5. *Delete*
  6. *Truncate*
- Depending on the OS, higher-level operations can also be provided, e.g.:
  - *Lock/unlock* file — concurrency control among unrelated processes
  - *Memory map* — manipulate a file as an in-memory buffer
  - …

Open Files

- The OS keeps track of **open files** in the system using several pieces of data
- Two levels of **open file tables**
  - **Per-process** open file table: one entry for each file opened by a process
    - Contain process-specific file information, e.g., current file pointer for read/write operations
    - Each entry also *points* to the relevant entry in the *system-wide table* of open files
  - **System-wide** open file table: one entry for each open file in the entire system
    - Contain process-independent file information, e.g., file physical location on mass storage, access rights
    - Keeps a counter (**open count**) of processes having opened a file, for *garbage collection* of the entry when it reaches 0

File Locking

- High-level operations provided by some OS and FS
- Enables concurrency control among unrelated processes via the FS as a shared resource
- Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire it concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- File-locking mechanisms:
  - **Mandatory** – access is denied depending on locks held and requested (common on Windows)
  - **Advisory** – processes can find status of locks and decide what to do (common on UNIX)

File Locking on UNIX — Example

- Many different APIs exist for file locking on UNIX
  - **Flock** is a popular (but non-standardized) one that support **entire-file advisory locking**
  - **fcntl** is a POSIX alternative, supporting **advisory record locking** (byte range granularity)
- Flock can be used via the `flock()` syscall or the homonymous CLI command:

```
flock [options] file|dir -c command [arguments]
```

  - `command` is executed wrapping it into a lock acquisition on `file` or `dir`
  - By default (or with `-x`) acquires an **exclusive lock**
  - `-s` requests a **shared lock** instead
  - `-w sec` fail if lock cannot be acquired within `sec` seconds
- Examples (from `flock(1)`):

```
shell1> flock /tmp -c cat
shell2> flock -w .007 /tmp -c echo; /bin/echo $?
#   Set exclusive lock to directory /tmp and the second command will fail.
```

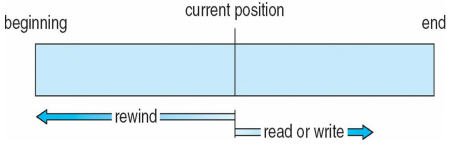
```
shell> flock -x local-lock-file echo 'a b c'
#   Grab the exclusive lock "local-lock-file" before running echo with 'a b c'.
```

File Types

- The content of files can be expected/required to be in a given **file format**
  - See: https://en.wikipedia.org/wiki/List_of_file_formats
- The OS can be either *agnostic* (does not care) or *opinionated* about file formats (supporting them explicitly in system calls)
  - At the very minimum the OS should support an **executable file format** for program execution
  - Other than that modern OSes tend to be file-format agnostic
    - Windows support: **textual vs binary** files
    - UNIX does not care: all files are just **byte sequences**
- **Filename extensions**, where used, are just *hints* to programs about what a file *might* contain
  - See: https://en.wikipedia.org/wiki/List_of_filename_extensions

Access Methods

- The OS can provide file access in various styles
- **Sequential access**: a file is just a **sequence of logical records**, which are read and/or written one after another, moving the read/write pointer as we go
  - Degenerate (but very common!) case: a logical record is a single byte → file = sequence of bytes



- **Direct access**: a file is an array of records; each record is identified by an integer n
  - Syscalls for read/write systematically take n as parameter: `read n`, `write n`, etc.
- **Index access**: generalization of direct access, with a reach index (e.g., a string key)
  - Requires an index present somewhere, e.g., in each file, cached into memory upon opening
- Sequential access is the simplest and most common file access method. It is the main one used on UNIX.

Sequential Access on UNIX

- Based on the sequential access method, with one syscall per operation
- **Opening** a file and obtaining an integer **file descriptor** as handle for future operations:

```
int open(const char *pathname, int flags, ... /* mode_t mode */);
    // Returns file descriptor on success, or -1 on error
```

- **Reading** content from file to process memory, chunk-by-chunk

```
ssize_t read(int fd, void *buffer, size_t count);
    // Returns number of bytes read, 0 on EOF, or -1 on error
```

  - `read` is very different from `fread`. `read` is a *system call*, which invokes a OS service; `fread` is a *library function* (from the C standard library) which executes user code and eventually calls `read` itself.
  - `fread` entails a **double copy**, i.e., memory is copied *twice*: (1) from kernel buffer to stdlib buffer, (2) from stdlib buffer to user process buffer.

Sequential Access on UNIX (cont.)

- **Writing** content from process memory to file, chunk-by-chunk

```
ssize_t write(int fd, void *buffer, size_t count);
    // Returns number of bytes written, or -1 on error
```

- `read` and `write` read/write starting from the position of a **shared file pointer**.[1] Then, they advance the pointer by the number of bytes read/written.
- It is also possible to explicitly **move the file pointer**:

```
off_t lseek(int fd, off_t offset, int whence);
    // Returns new file offset if successful, or -1 on error
    //
    // whence is one of:
    // - SEEK_SET offset is set offset bytes from the beginning of the file
    // - SEEK_CUR offset adjusted by bytes relative to the current file offset
    // - SEEK_END offset set to the size of the file plus offset
```

[1] not to be confused with a C language pointer; a file pointer is just the current position in an open file

Sequential Access on UNIX — Example

```c
/* Based on copy.c from "The Linux Programming Interface" book, adapted for teaching purposes.
   Copyright (C) Michael Kerrisk, 2010. License: GNU AGPL, version 3 or above.

   Copy the file named argv[1] to a new file named in argv[2].
*/

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#define BUF_SIZE 1024
#define errExit(msg) { fprintf(stderr, "%s", (msg)); exit(EXIT_FAILURE); }

int main(int argc, char *argv[]) {
    int inputFd, outputFd;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3)  errExit("Usage: copy OLD_FILE NEW_FILE");
```

Sequential Access on UNIX — Example (cont.)

```
    inputFd = open(argv[1], O_RDONLY);  /* open source file */
    if (inputFd == -1)  errExit("error opening source file");

    outputFd = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, 0666);  /* open destination file */
    if (outputFd == -1)  errExit("error opening destination file");

    /* Transfer data until we encounter end of input or an error */
    while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
        if (write(outputFd, buf, numRead) != numRead)
            errExit("error: cannot write whole buffer");
    if (numRead == -1)  errExit("read error");

    if (close(inputFd) == -1)  errExit("error while closing source file");
    if (close(outputFd) == -1)  errExit("error while closing output file");
    exit(EXIT_SUCCESS);
}
```

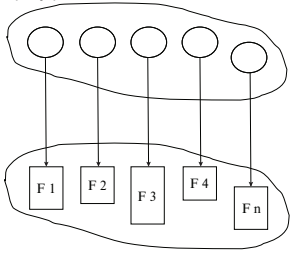
```
$ gcc -Wall -g -o copy copy.c
$ echo foo > foo.txt
$ ./copy foo.txt bar.txt       # try also "strace ./copy foo.txt bar.txt" to see syscall trace
$ cat bar.txt
foo
```

Directory Structure

- Files are *organized in directories*
- A directory can be viewed as a **symbol table** translating file names to FCBs
- As an abstract data type, directories support the following operations:[a]
  1. **Search** for a file
  2. **Create** a file
  3. **Delete** a file
  4. **List** directory content
  5. **Rename** a file
  6. **Traverse** the FS (entire or parts of)

[a] Note how (2), (3), and (5) are operations *on directories*, rather than files.

- Information about *all* files (bottom of the figure) in the FS are kept in a global **directory structure** (top)

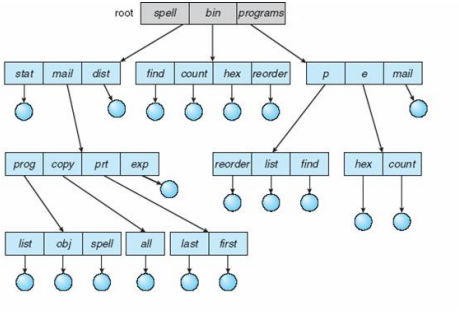


- Both the directory structure and the files reside on disk
  - *How* the directory structure and files are represented in mass storage (= which sequence of bytes) depends on the FS implementation

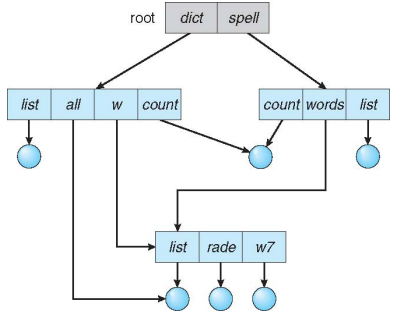
Directory Organization

- Several different *logical organizations* for the directory structure are possible
  - **Single-level directory:** all FS files in a single directory(!) — simple, but too prone to filename clashes
  - **Two-level directories** — e.g., one home directory per user, no further nesting
- Most used directory organizations are hierarchical, either trees or DAGs
  - **Paths** are hierarchical file names used to navigate the hierarchical structure

Tree-structured directories



Direct Acyclic Graph (DAG) directories



- **Full graph** directories are possible, but avoided due to garbage collection complexity

DAG Organization — UNIX example

- The UNIX FS model is a DAG, implemented using different **file types**
  - Regular: “normal” files containing data or programs
  - Directory: symbol tables, associating to each filename another FS entity (e.g., file or directory)
- **Symbolic link** (symlinks): files containing paths, followed transparently and by default by the FS

```
$ echo foo > foo.txt

$ ln -s foo.txt bar.txt    # create a symbolic link bar.txt pointing to foo.txt
$ ls -l bar.txt
lrwxrwxrwx 1 zack zack 7 Sep 12 11:52 bar.txt -> foo.txt

$ cat bar.txt    # the OS follows the symbolic link automatically
foo
```

  - Symbolic links allow to *create cycles* (e.g., `/home/zack/my_dir -> /`), but the fact they have a dedicated file type allows to skip them when performing FS travels or removing files

DAG Organization — UNIX example (cont.)

- **Hard links** (not a file type!) allow two separate directory entries to point to the same file (they are forbidden for directories)

```
$ echo foo > foo.txt
$ ln foo.txt bar.txt    # add bar.txt as a new name for foo.txt in current dir
$ ln foo.txt baz.txt    # and another one! (baz.txt)
$ ls -l
-rw-r--r-- 3 zack zack 4 set 12 11:52 bar.txt
-rw-r--r-- 3 zack zack 4 set 12 11:52 baz.txt
-rw-r--r-- 3 zack zack 4 set 12 11:52 foo.txt
```

  - All three directory entries point to the same FCB on the FS; none of them is the “original” file vs links to it, as it happens with symlinks

DAG Organization — UNIX example (cont.)

- **Hard links** (not a file type!) allow two separate directory entries to point to the same file (they are forbidden for directories)

```
$ echo foo > foo.txt
$ ln foo.txt bar.txt    # add bar.txt as a new name for foo.txt in current dir
$ ln foo.txt baz.txt    # and another one! (baz.txt)
$ ls -l
-rw-r--r-- 3 zack zack 4 set 12 11:52 bar.txt
-rw-r--r-- 3 zack zack 4 set 12 11:52 baz.txt
-rw-r--r-- 3 zack zack 4 set 12 11:52 foo.txt
```

  - All three directory entries point to the same FCB on the FS; none of them is the “original” file vs links to it, as it happens with symlinks

- Q: when can we free the disk space used by a file in a DAG FS?
  A: when the number of (hard) links to it reaches 0

  - In the example above, `3` denotes the number of inbound hard links to the file
  - This is why the UNIX syscall to “remove” files is called `unlink()`
  - (It is also why garbage collection is complicated in general graph FS.)

File Protection

- In a multi-user system, the OS needs to ensure that *only valid accesses to files* are permitted
  - Users should be able to decide if/how/which other users can access their files
- For each different type of file and directory access—read, write, execute, create, delete, etc.—the OS will verify if it is permitted and terminate the syscall with an error (e.g., `EPERM`) if not
- Most general mechanism: **Access Control List (ACL)**
  - Each action is performed by a user via a syscall invocation
  - Each FS object associated to owner user + list of ⟨user, action⟩ pairs of permitted actions & to whom
  - Each action verified against the ACL; fail if there is no match
- Problem: **ACL length** → FCB can no longer be fixed-size, making FS implementation complicated
  - Common approach: *group* together actions and/or users in the ACL
  - E.g., group together all actions performed by:
    1. file owner,
    2. users member of the same **user group**,
    3. anyone else (“others”)

File Protection — UNIX Example

- 3 permission sets: file owner (“**user**”), members of the same **group**, anyone else (“**others**”)
- 3 permission bits for each set
  - **Read** — ability to read the content of the object (data for files, list content for directories)
  - **Write** — ability to change the content (note: directories need this to create/remove files)
  - **Execute**
    - For files: ability to execute the file as a program
    - For directories: ability to resolve paths that contain the directory anywhere
- Syscalls and commands for UNIX permission manipulation: `chmod`, `chown`, `chgrp`
  - UNIX also support optional, variable-size **extended ACLs**; manipulated with: `getfacl`, `setfacl`

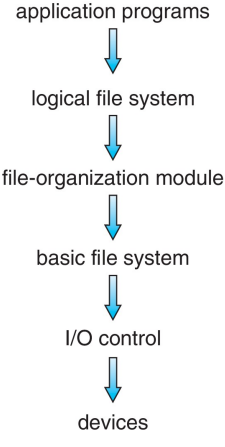
```
$ touch backup_password.txt              # create empty file (set owner user implicity)
$ chgrp adm backup_password.txt          # set its group to "adm"
$ chmod u=rw,g=r,o= backup_password.txt  # r/w for owner, read-only for admins, nothing for others
$  echo s3cr3t >> backup_password.txt    # write password to file (quiz: why not before?)
$ ls -l backup_password.txt
-rw-r----- 1 zack adm 7 set 12 13:45 backup_password.txt
```

```
$ ls -l /home/ | grep zack
drwx--x--x 61 zack zack 12288 set 12 12:50 zack
# owner can manipulate dir freely; others can just traverse paths through it (but not ls!)
```

File-System Implementation

File-System Structure

- Concerns in implementing a filesystem:
  1. defining how the FS should look to the user
  2. map the logical FS on the physical secondary storage
- Common software architecture for FS: **layered file system**
  - **I/O control:** device drivers and interrupt handlers for data transfer to/from storage devices
    - E.g., `retrieve disk block 123`
  - **Basic file system:** issue commands to the appropriate device driver to read/write logical blocks
    - Handle *scheduling* of disk operations, *caching* and I/O buffers
    - Linux example: the block I/O subsystem
  - **File-organization module:** knows about files, block logical and physical addresses
    - Translates logical block n. $\leftrightarrow$ physical block n.
    - Handle free space and disk allocation
  - **Logical file system:** knows about file *metadata* (= FCB information)
    - Translate file names/paths into file numbers
    - Expose FS operations to programs via syscalls and enforce ACLs



Filesystems

- May different filesystems exist and can be supported by the same OS
- Examples:
  - CD-ROM: ISO 9660
  - UNIX: UFS, FFS, …
  - Windows: FAT, FAT32, NTFS, …
  - Linux supports 130+, with extended file system ext3 and ext4 as current defaults
- Distributed file systems
- FUSE: general mechanism on Linux to implement filesystems in user space
- Active ares of R&D with new FS still arriving and gaining popularity
  - E.g., XFS, ZFS, btrfs

Filesystem Structures

On disk

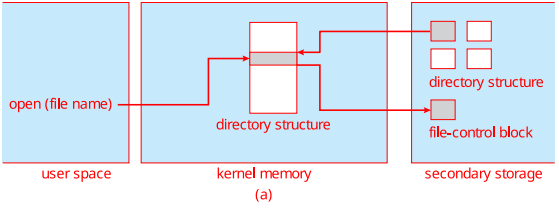
- Physical disks organized in *partitions*, containing logical **volumes**, where filesystems reside
- Each volume contains
  - (Optional) **Boot control block** with info needed to boot OS
  - **Volume control block** (called *superblock* on UNIX): total n. of blocks, n. of free blocks, pointers to free blocks, block size
  - **Directory structure**
- File Control Block (already discussed) for each file

In memory

- **Mount table** listing all FS currently active ("mounted") in the system
- *System-wide open-file table*
- *Per-process open-file table*
- Lots of caches!
  - E.g., copy of the FCB of each open file, copy of the directory content of recently used directories, I/O buffers, etc.

Filesystem Structures in Action — Example

- File open (a)
  1. pass file name to logical file system
  2. use directory structure (possibly cached) to obtain file id
  3. retrieve FCB from disk and cache it
  4. update both open-file tables as needed
  5. return file handle as syscall return value
- File read (b)
  1. retrieve FCB from file handle via open file table(s)
  2. based on current file read offset, locate relevant data block
  3. schedule disk read
  4. update memory, return from syscall



(Simplified for brevity.)

Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash table** – linear list with hash data structure pointing into it
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Allocation Method

Filesystem Allocation Problem

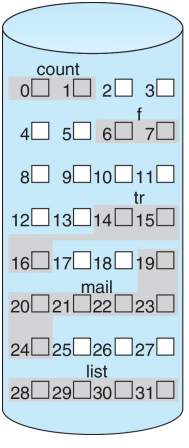
How to allocate storage space to files so that:

1. Storage space is utilized effectively, and
2. Files can be accessed quickly.

- Context: secondary storage disks are accessible (for read/write) at the granularity of **fixed-size blocks** (e.g., 512 bytes to 10 KiB), not individual bytes
- An **allocation method** refers to how disk blocks are allocated to files
- Three major allocation methods:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

Contiguous Allocation

- Each file occupies a **set of contiguous blocks**
- Simple – only starting location (block n.) and length (number of blocks) are required to identify a file on disk
- Support both sequential and direct access
- Best performances in most cases
- Problems
  - Finding space on the disk for a file
    - There is enough *total* space in the disk to store a file, but scattered all around, so we cannot use it
    - Known as **external fragmentation** problem
  - Knowing file size *in advance*, at creation time
  - Need for **defragmentation** either off-line (downtime) or on-line (performances impacted for the process duration)

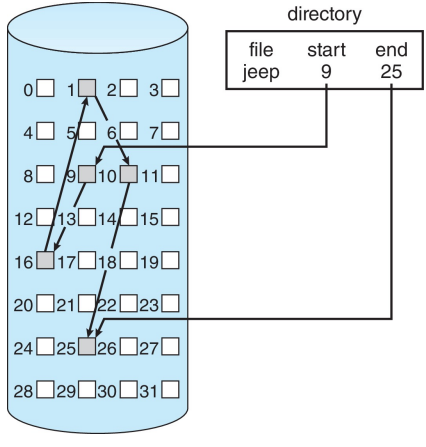


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

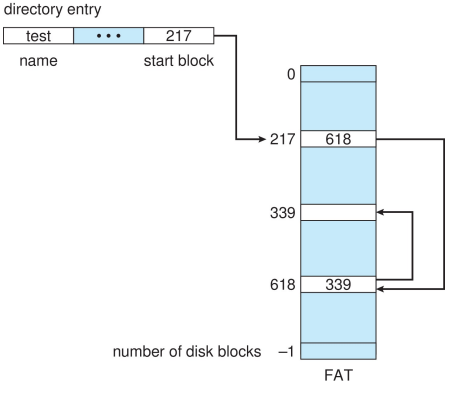
Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
  - FCB points to first (and possibly last) block
  - Each block points to the next
- Solves completely external fragmentation problem
  - Blocks can be scattered anywhere on disk
- Problems
  - Does not support direct access
  - Pointers take some space
  - **Reliability:** what if you lose a block in the middle of the chain?



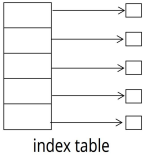
File-Allocation Table (FAT)

- The **File-Allocation Table (FAT)** was a very popular filesystem for MS-DOS and early Windows releases
  - Still used on simple devices like USB sticks
- Linked allocation method that separates the block list from data blocks
- Beginning of the volume contain a **linked list of block IDs** (but not the data!)
- To read a file:
  1. Read FAT; find desired file entry
  2. Follow list to desired block; obtain block ID
  3. Read block from disk
- Pro: better direct access: “only” FAT needs to be read
- Pro: can keep multiple FAT copies to improve reliability
- Con: if FAT not cached, many seeks back/forth with disk head

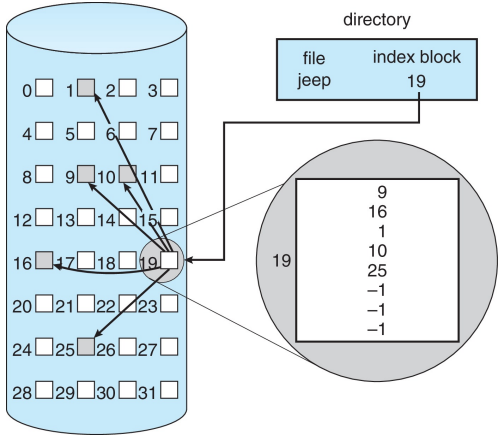


Indexed Allocation

- **Indexed allocation** brings *all data pointers for a file into one location*
  - Each file has its own **index block**(s) of pointers to its data blocks

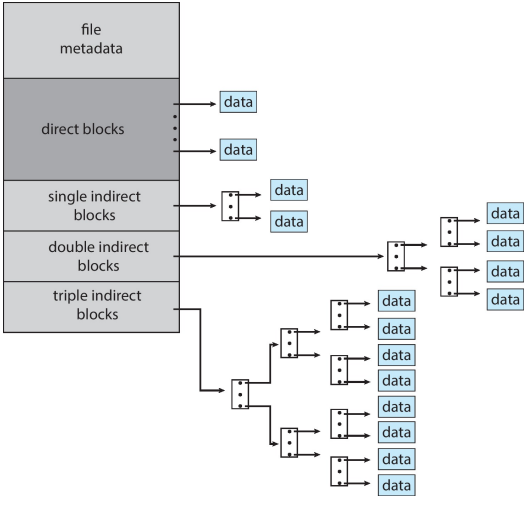


- Supports direct access: (1) locate the desired pointer in the index block, (2) follow it
- No external fragmentation
- Problem: maximum file size limited by the size of index block (how many pointers can it fit?)
  - Trade-off on index block size:
    - large → more space used by pointers
    - small → limited maximum file size



Combined Indexed Allocation — UNIX Example

- FCB on UNIX FS called **i-node** (or *inode*, for "index node")
  - Stores file metadata + data block pointers
- Data pointers separated in tiers
  - First tier (e.g., 15) point to **direct blocks** containing file data (ptrs → data)
  - Second tier point to **single indirect blocks**: disk blocks containing *themselves* pointers to blocks with file data (ptrs → ptrs → data)
  - Third tier: **double indirect blocks** (ptrs → ptrs → ptrs → data)
  - And so on up to **triple indirect blocks**
- Analysis
  - Keeps inode size small (e.g., 2 KiB)
  - Fast direct access to initial part of a file
  - Good direct access (via few indirections) to entire file



A Closer Look to I-nodes

On UNIX, user programs can retrieve the information contained in the i-node of a file using the `stat` family of syscalls (the already shown `stat` CLI command provides the same functionality):

```
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);  // same but doesn't follow pathname if symlink
int fstat(int fd, struct stat *statbuf);                // same for an already open file
    // All return 0 on success, or -1 on error
```

```
struct stat {
    dev_t      st_dev;      /* IDs of device on which file resides */
    ino_t      st_ino;      /* I-node number of file */
    mode_t     st_mode;     /* File type and permissions */
    nlink_t    st_nlink;    /* Number of (hard) links to file */
    uid_t      st_uid;      /* User ID of file owner */
    gid_t      st_gid;      /* Group ID of file owner */
    dev_t      st_rdev;     /* IDs for device special files */
    off_t      st_size;     /* Total file size (bytes) */
    blksize_t  st_blksize;  /* Optimal block size for I/O (bytes) */
    blkcnt_t   st_blocks;   /* Number of (512B) blocks allocated */
    time_t     st_atime;    /* Time of last file access */
    time_t     st_mtime;    /* Time of last file modification */
    time_t     st_ctime;    /* Time of last status change */
};
```

A Closer Look to I-nodes (cont.)

- UNIX mantra: **“everything is a file”**
- In particular, FS objects used to create the **filesystem DAG structure** are files themselves
  - Directory
    - FS object of type directory (`S_ISDIR(stat.mode_t)` is true), containing a list of associations ⟨filename, i-node⟩
    - The list of pairs is stored in the data blocks of the FS object; as a list or hash table, depending on the FS implementation
  - Symbolic link
    - FS object of type symlink (`S_ISLNK(stat.mode_t)` is true), containing a string that is interpreted as a path and followed transparently upon file access
    - The string (= link destination) is stored in the data blocks of the FS object

Free-Space Management

How do you find a free block to allocate to a given file when needed?

- FS maintains a **free-space list** to track available blocks
- Common implementation: a **bit vector** (or *bitmap*) with 1 bit per logical block
  - bit = 1 if a block is free; 0 otherwise
  - Example: consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be:
    `00111100111111000110000011100000…`
- Very efficient: find n contiguous free blocks → find a string of n bits = 1
  - Modern CPUs have dedicated instructions for complex bit operations
- Bitmap requires extra space, ideally in memory for caching. Example:
  - block size = 4 KiB = 2^{12} bytes
    disk size = 2^{40} bytes (1 TiB)
    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32 MiB)
  - Growing more and more as disk sizes increase…

Efficiency and Performance

- FS efficiency is critical for system performances
  - Rationale: files are the main tool for data storage + disks are the slowest storage
- FS efficiency depends on:
  - Disk allocation method and directory algorithms
  - Types of data kept in file's directory entry
    - E.g., to list the size of all directory entries do I need a separate `stat()` call per entry?
  - Pre-allocation or as-needed allocation of metadata structures
    - E.g., UNIX i-nodes are pre-allocated on disk so that they cost nothing to create
  - Fixed-size or varying-size data structures

Efficiency and Performance (cont.)

- Modern FS use many different techniques to improve performances
- Keeping data and metadata close together
  - E.g., UNIX i-nodes are spread around the disk so that they can be near to their data blocks
- **Buffer cache** – OS keeps a (possibly very large!) part of main memory as cache for frequently used data blocks and i-nodes
  - E.g., see the `buffer/cache` column in the output of the `free` command (on Linux)
- **Synchronous writes** sometimes requested by apps or needed by OS
  - Bypass buffering/caching – writes must hit disk before `write()` syscall return
- **Asynchronous writes** are more common, buffer-able, faster
  - `write()` returns as soon as data is written *to buffer cache*; the OS will flush it to disk later, when convenient/efficient
- Optimizations for *sequential access:*
  - **Free-behind:** remove an entry from the buffer cache as soon as next one is requested
    - Rationale: it will not be requested soon anyway
  - **Read-ahead:** read at once from disk several blocks *after* the requested one
    - Rationale: they will be requested soon + larger disk reads can be more efficient and benefit from DMA

Recovery

- FS implementation often keeps the *same information in various places*
  - E.g., on-disk, in-memory, in caches
- FS operations often need to update *different structures* consistently
  - E.g., allocate an i-node, fill corresponding data blocks with pointers + data
- **What if a crash or data corruption breaks FS data consistency?**
- Standard approach to the problem: (1) detect, (2) correct (when possible)
- **Consistency checking** – compares data in directory structure with data blocks on disk, tries to fix
  - E.g., `fsck` on UNIX
  - Can be slow and is not foolproof
  - Often run at system boot: at given intervals and/or if FS was not unmounted cleanly

Recovery

- FS implementation often keeps the *same information in various places*
  - E.g., on-disk, in-memory, in caches
- FS operations often need to update *different structures* consistently
  - E.g., allocate an i-node, fill corresponding data blocks with pointers + data
- **What if a crash or data corruption breaks FS data consistency?**
- Standard approach to the problem: (1) detect, (2) correct (when possible)
- **Consistency checking** – compares data in directory structure with data blocks on disk, tries to fix
  - E.g., `fsck` on UNIX
  - Can be slow and is not foolproof
  - Often run at system boot: at given intervals and/or if FS was not unmounted cleanly

- Periodically use system programs to **backup** FS data to another storage device (magnetic tape, other magnetic disk, optical)
  - Recover lost file or disk by **restoring** data from backup

Journaling

- **Journaling** is a feature of modern FS, inspired by *database log-based recovery algorithms*
  - Observation: with consistency-checking we *allow* to break and detect+repair it later
  - Journaling goal: prevent failures to become FS inconsistencies; FS always consistent by design
- Record each metadata update to the file system as a **transaction**
- All transactions are **written to a log** (!= to FS structures on disk)
  - A transaction is considered committed once written to the log
  - Log could be on a separate device or section of disk
  - However, the file system may not yet be updated
- Transactions are **asynchronously written from the log to the FS structures** (flush)
  - After FS structures updated, transaction is removed from the log
- In case of crash, all remaining transactions in the log must be **replayed** at mount
- Faster recovery from crash, removes chance of inconsistency of metadata
  - Bonus point: transactions can be batched together for flush, improving write performances

Reading List

You should study on books, not slides! Reading material for this lecture is:

- Silberschatz, Galvin, Gagne. Operating System Concepts, Tenth Edition:
  - Chapter 13: File-System Interface
  - Chapter 14: File-System Implementation

Credits:

- Some of the material in these slides is reused (with modifications) from the official slides of the book Operating System Concepts, Tenth Edition, as permitted by their copyright note.