Where are new files allocated?`

Introduction

The file system provides mechanisms to save data permanently. It icnludes management of Files, Directories, and Disk & Disk partitions

Files

A set of correlated information. All information (i.e., numbers, characters, images, etc.) are stored in a (electronic) device using a coding system.
Contiguous address space

ASCII encoding

It's the standard. American Standard Code for Information Interchange.
Originnaly based on the English alphabet, 128 characters are coded in 7-bit (binary numbers)
Extended ASCII:
Extension of ASCII to 8-bit and 255 characters. Several versions exist: ISO 8859-1 (ISO Latin-1), ISO 8859-2 (Eastern European languages), ISO 8859-5 for Cyrillic languages, etc.

Unicode encoding

Industrial standard that includes the alphabets for any existing writing system. It contains more 110,000 characters and it includes more than 100 sets of symbols.
Several implementations exist:

- UCS (Universal Character Set)
- UTF (Unicode Tranformation Format)
  - UTF-8, groups of 8 bits size (1, 2, 3 or 4 groups)
    - ASCII coded in the first 8 bits
  - UTF-16, groups of 16 bits size (1 or 2 groups)
  - UTF-32, groups of 32 bits size (fixed length)

Textual and binary files

A file is a sequence of bytes. They are all binary. We usally distinguish betwen:

- Textual files (or ASCII)
  - C sources, C++, Java, etc.
- Binary files
  - Executables, Word, Excel etc.

Textual files

Files consisting of data encoded in ASCII, texutal files are usally line-oriented:
Newline: go to the next line
In windows its `Line Feed` + `Carriage Return`, in unix only `Line Feed

Binary files

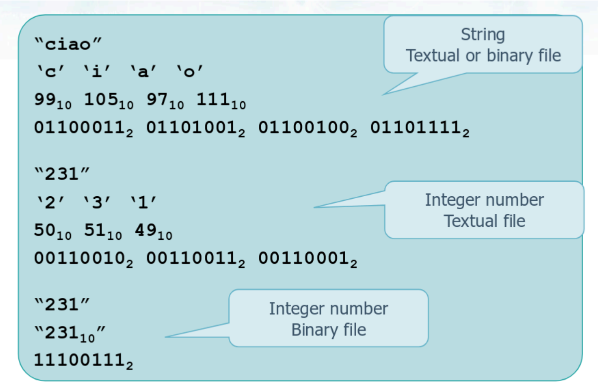
A sequence of 0 and 1, not “byte-oriented".
The smallest unit that can be read/written is a bit.

The adveantages are:

- Compactness
- Ease of editing the file
- ease of positioning on the file

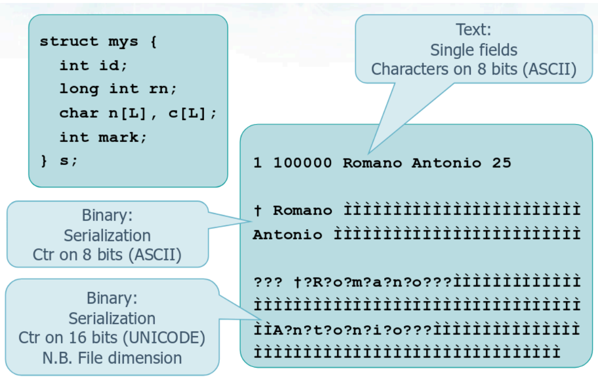
Drawbacks:

- Limited portability
- Impossibility to se a standard editor



Serialization

Process of translating a structure (e.g., C struct) into a storable format. Using serialization, a struct can be stored or transmitted (on the network) as a single entity. When the sequence of bits is read, it is done in accordance with the serialization process, and the struct is reconstructed in an identical manner.
Many languages support serialization using R/W operations on a file: Java, Python, Objective-C, Ruby, etc.



ISO C Standard library

I/O operations with ANSI C can be performed through different categories of functions:

- Character by character
- Row by row
- Formatted I/O
- Binary I/O

[Read examples](#)
[Write examples](#)
[Binary I/O examples](#)

Standard I/O is fully buffered:

1. The I/O operation is performed only when the I/O buffer is full
2. The "flush" operation indicates the actual write of the buffer to the I/O

```
#include <stdio.h>
void setbuf (FILE *fp, char *buf);
int fflush (FILE *fp);
```

For concurrent processes, use:

```
setbuf (stdout, 0);
```

```
fflush (stdout);
```

Open and close a file

```
include <stdio.h>
FILE *fopen (char *path, char *type);
FILE *fclose (FILE *fp);
```

Access methods:
r, rb, w, wb, a, ab r+, r+b, etc.
read, write, append, `r+` is read and write
`b` is for binary, unix doesnt make any distinction on these, so in UNIX `r = rb`

Characters by character

```
include <stdio.h>
int getc (FILE *fp);
int fgetc (FILE *fp);
int putc (int c, FILE *fp);
int fputc (int c, FILE *fp);
```

Returned values are:

- A character on success
- EOF on error, or when the end of the file is reached

The function

- `getchar` is equivalent to `getc(stdin)
- `putchar` is equivalent to `putc(c, stdout)`

row by row

```
#include <stdio.h>
char *gets (char *buf);
char *fgets (char *buf, int n, FILE *fp);
int puts (char *buf);
int fputs (char *buf, FILE *fp);
```

Returned values are:

- buf (gets/fgets), or a non-negative value (puts/fputs) in the case of success
- NULL (gets/fgets), or EOF for errors or when the end of file is reached (puts/fputs)

Lines must be delimited by "new-line"

Formatted

```
#include <stdio.h>
int scanf (char format, …);
int fscanf (FILE *fp, char format, …);
int printf (char format, …);
int fprintf (FILE *fp, char format, …);
```

High flexibility in data manipulation

- Formats (characters, integers, reals, etc.)
- Conversions

Binary

```
#include <stdio.h>
size_t fread (void *ptr, size_t size, size_t nObj, FILE *fp);
size_t fwrite (void *ptr, size_t size, size_t nObj, FILE *fp);
```

Each I/O operation (single) operates on an aggregate object of specific size

- With `getc` / `putc` it would be necessary to iterate on all the fields of the struct
- With `gets` / `puts` it is not possible, because both would terminate on NULL bytes or new-lines

Returned values are:

- Number of objects written/read
- If the returned value does not correspond to the parameter nObj
  - An error has occurred
  - The end of file has been reached

They are often used to manage binary files

- serialized R/W (single operation for the whole struct)
- Potential problems in managing different architectures
  - Data format compatibility (e.g., integers, reals, etc.)
  - Different offsets for the fields of the struct

POSIX Standard Library

I/O in UNIX can be entirely performed with only 5 functions:

- open
- read
- write
- lseek
- close

This type of access js part of POSIX and of the Single UNIX Specification, but not of ISO C. It is normally defined with the term "unbuffered I/O", in the sense that each read or write operation corresponds to a system call.

System call open()

In the UNIX kernel a "file descriptor" is a non-negative integer
Conventionally (also for shells):

- Standard input
  - 0 = STDIN_FILENO
- Standard output
  - 1 = STDOUT_FILENO
- Standard error
  - 2 = STDERR_FILENO

```
include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *path, int flags);
int open (const char *path, int flags, mode_t mode);
```

It opens a file defining the permissions
Returned values are:

- The descriptor of the file on success
- -1 on error

The `mode` parameter is optional.
Path indicates the file to open.
Flags has multiple options:
Can be obtained with the OR bit-by-bit of constants defined in the header file `fcntl.h`
One of the following three constants is mandatory:

- O_RDONLY open for read-only access
- O_WRONLY open for write-only access
- O_RDWR open for read-write access

Optional constants:

- O_CREAT creates the files if not exist
- O_EXCL error if O_CREAT is set and the file exists
- O_TRUNC remove the content of the file
- O_APPEND append to the file
- O_SYNC each write waits that the physical write operation is finished before continuing

`mode` specifies access permissions:

- `S_I[RWX]USR` -> `rwx --- ---`
- `S_I[RWX]GRP` -> `--- rwx ---`
- `S_I[RWX]OTH` -> `--- --- rwx`

When a file is created, actual permissions are obtained from the umask of the user owner of the process.

System call read()

```
#include <unistd.h>
int read (int fd, void *buf, size_t nbytes);
```

Read from file fd a number of bytes equal to nbytes, storing them in buf
Returned values are:

- number of read bytes on success
- -1 on error
- 0 in the case of EOF

The returned value is lower that `nbytes`

- If the end of the file is reached before nbytes bytes have been read
- If the pipe you are reading from does not contain `nbytes` bytes

System call write()

```
#include <unistd.h>
int write (int fd, void *buf, size_t nbytes);
```

Write `nbytes` bytes from buf in the file identified by descriptor `fd`
Returned values are:

- The number of written bytes in the case of success, i.e., normally nbytes
- -1 on error

It's important to know that write writes on the system buffer, not on the disk `fd = open (file, O_WRONLY | O_SYNC);`.
`O_SYNC` forces the sync of the buffers, but only for `ext2` file systems.

System call lseek()

```c
#include <unistd.h>
off_t lseek (int fd, off_t offset, int whence);
```

The current position of the file offset is associated to each file

- The system call lseek assigns the value offset to the file offset
- The offset value is expressed in bytes

whence specifies the interpretation of offset

- If `whence==SEEK_SET`. The offset is evaluated from the beginning of the file
- If `whence==SEEK_CUR`. The offset is evaluated from the current position. Offset can be pos or neg
- If `whence==SEEK_END`. The offset is evaluated from the end of the file. Offset can be pos or neg. It is possible to leave "holes" in a file (filled with zeros).

Returned values are:

- new offset on success
- -1 on error

System call close()

```c
#include <unistd.h>
int close (int fd);
```

Returned values are:

- 0 on success
- -1 on error

All the open files are closed automatically when the process terminates

Example R/W

This program works indifferently on text and binary files

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFFSIZE 4096
int main(void) {
     int nR, nW, fdR, fdW;
     char buf[BUFFSIZE];
     fdR = open (argv[1], O_RDONLY);
     fdW = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC,
     S_IRUSR | S_IWUSR);

     if ( fdR==(-1) || fdW==(-1) ) {
          fprintf (stdout, “Error Opening a File.\n“);
          exit (1);
     }

     while ( (nR = read (fdR, buf, BUFFSIZE)) > 0 ) {
          nW = write (fdW, buf, nR);
          if ( nR!=nW ){
               fprintf (stderr, "Error: Read %d, Write %d).\n", nR, nW);
          }
     }

     if ( nR < 0 ){
          fprintf (stderr, "Write Error.\n"); //Error check on the last reading operation
     }
     close (fdR);
```

```
    close (fdW);
    exit(0);
    }
```

Directories

No storage system contains a single file. They are organized in directories. We can see them both as trees and graphs. Both files and directories are saved in mass memory.
Operations that can be performed on directories are similar to the ones applied to files. Creation, deletion, listing, rename, visit, search, etc.

Structure

Structuring a file systems by means of directories has several advantages:

- Efficiency: Speed in modifying the file system, e.g., searching a file
- Naming: Allow to assign the same name to different files
- Grouping (Organization)

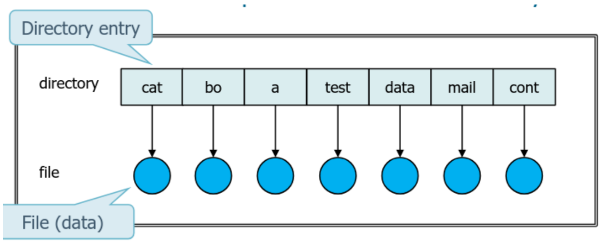
Directories with one level

The simplest structure has only one level. All the files of the file system are stored within
the same directory. The files are differentiated by their name only and each name is unique within the entire file system.
For each file, two structures are exploited:

- Directory entry: indicates and name of the file and possibly other information about the file
- Data: stored in a different location than the directory entry, they are referred from the directory entry with a pointer

Performance:

- Efficiency
  - Easily understandable and usable structure
  - Easy and efficient managing of the file system
- Naming
  - Files must have unique names
  - It has evident limitations as the number of stored files increases
- Grouping
  - Management of files of a single user is complex
  - Management of multiple users is practically impossible



Directories with two levels

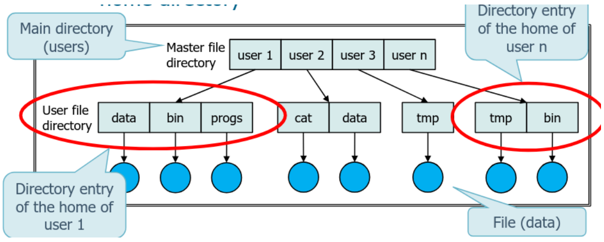
Files are contained in a two-level tree
Each user can have their own directory. This means that:

- Each user has its own directory
- All the operations are executed only in the correct
  home directory

Performance

- Efficiency
  - “user oriented” view of the file system
  - Simplified and efficient searches on a single user

- Naming
  - It is possible to have files with the same name if they belong to different users
  - A path name must be specified for each file
- Grouping
  - Simplified between different users
  - Complex for each individual user



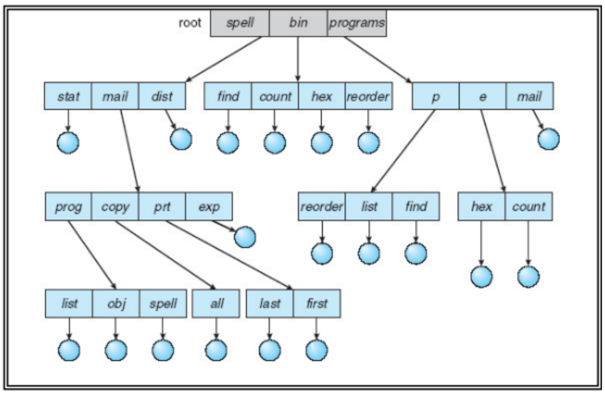
Tree directories

Generalize previous directories systems
Directories and files are organized as a tree

- Every node/vertex of the tree can include as entry other nodes/vertex of the tree

Every user can manage both files and directories (and subdirectories).
Performance:

- Efficiency
  - Efficient searches based on the tree structure and therefore to its depth and breadth
- Naming
  - With absolute path or relatice to the current working directory
- Grouping
  - Extended possibilities, flexible

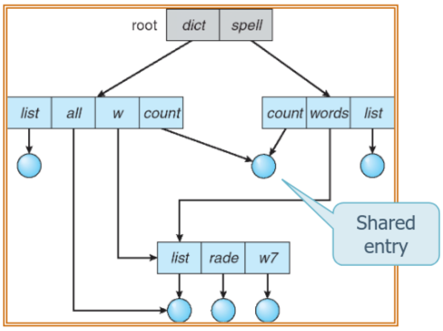


Acyclic graph directories

A tree file system does not allow sharing
It is often useful to refer to the same object in the file system with different filenames:

- Same user refers to an object with different pathnames
- Different users want to share objects
- It is worth noting that duplication of the object (i.e., the copy) is not a solution

Tree file systems can be generalized organizing them as acyclic graphs:



In UNIX-like systems, the standard stategy is the use of `links`. A link is a reference (pointer) to another (pre-existing) entry. But the presence of links increases difficulty in managing file systems. Necessary to distinguish between native entries and relative links, during creation, modification, and removal.

Durijg a visit or a search

- If the entry is a link, the operating system must use an indirect addressing, i.e., it has to "resolve" the link to access the original entry
- By means of links, each entry of the file system can be reached with different absolute pathnames (and with different names)
  - Analysis on the content of the file system (e.g., statistics on how many ".c" files are present) are much more complexxample during a visit or a search:

During the removal of an entry

- It is necessary to establish how to manage the link and the referred object
  - The removal of a link is usually performed immediately, and in general it does not affect original object
  - It is important to define how to delete the data
    - If you delete the object, what do you do with the links that point to the object?
    - When can the space reserved for the object be reused?

If you delete data immediately:

- It is possible to leave links pending (dangling)
- The OS is notified that the link does not point to an entry when it tries to use it

Delete data when the last link is deleted:

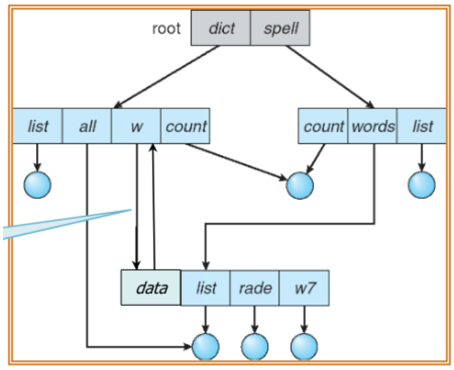
- To avoid pending links we can track them, we have to manage the presence of multiple links and objects
  - Maintaining the list of all the links is expensive (it is a list of variable length)
  - Delete all the links (i.e., the entries) when the object is deleted is expensive, because you need to search all the links
- It is convenient to store only a counter (number of links)
  - In UNIX systems this counter is stored in i-node
  - Increased and decreased appropriately

Creating a new link to a directory could cause the generation of a cycle in the file system:

- Managing a cyclic graph is more complex
  - Search and visit has to avoid infinite recursion
- The simplest strategy is to avoid the creation of a link pointing a directory

Cyclic graph directories

The alternative to acyclic graphs is cyclic graphs



Allocation

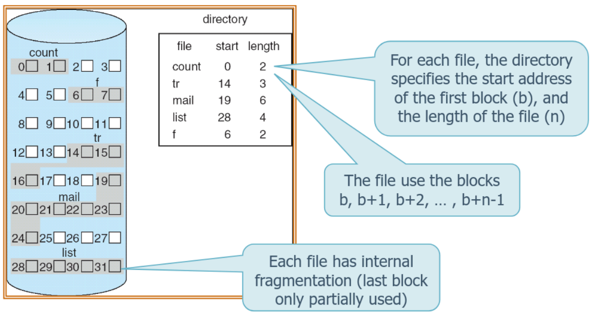
Techniques

For allocation we mean techniques for choosing the blocks of the disks to store files
We will not deal with the structure of the storage units.
Those unit can be modelled as a linear indexable set (a vector) of blocks.

Main allocation techniques:

Contiguous allocation

Each file is stored in a contiguous set of blocks

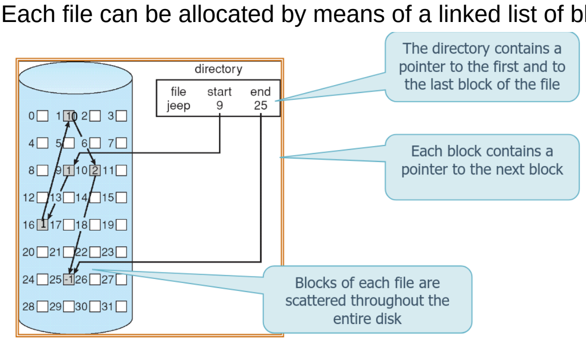


Pros:

- Really easy allocation strategy. Really little information is stored for each file
- It allows immediate and sequential accesses. They are sequential
- It allows simple and direct access. The block i starting from block b is at address $b+i-1$
Cons:
- An allocation policy is needed. Where are new files allocated? You need several algorithms (first-fit, best-fit, worst-fit etc)
- No allocation algorithm is free of defects consequently there is a waste of space
  - The waste is knows as external fragmentation
  - possible re-compaction (on-line and off-line)
- Dynamic allocation problems
  - Files cannot grow indefinitely, becasue the available space is limited by the next file

Linked location

Each file can be allocated by means of a linked list of blocks



Pros

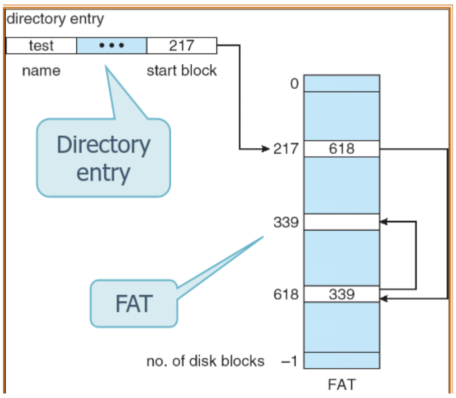
- Resolve problems of contiguous allocation
    - Allows dynamic allocation of file
    - Eliminate the external fragmentation
    - Avoid the use of complex allocation algorithms

Cons:

- Each read operation imply a sequential access to the blocks
- It is efficient only for sequential accesses
    - Direct access requires reading a chain of pointers until the desired address is reached
    - Each access to a pointer (or block) consists in a read operation
- To store pointers
    - Space is required
    - Pointers are critical from the viewpoint of reliability
    - Decrease the space usable to store data

File allocation table

FAT was develpoetd by IBM and later by microsoft. It was the primary file system for many Microsoft Windows vased OSs.
In it References are not stored inside the data blocks on the disk, but directly in a specific block containing the FAT.
Its a table with one element for each block on the disk. The sequence of blocks reffered to a file is identified starting from the directory using:
Starting block of the file in the FAT
Sequence of pointers available (directly) in the FAT (no longer in the blocks)



The reading of each block requires two disk accesses (one to the FAT and one to the block to read)

- First access on the FAT
- Second on the data block

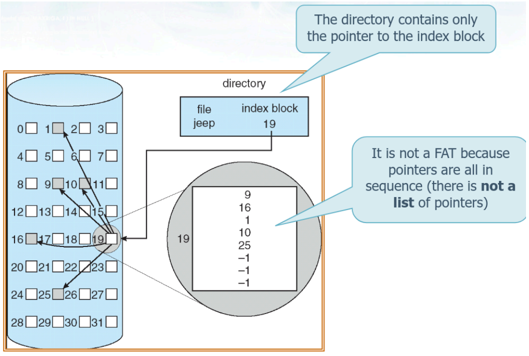
Limits:

- Slow access
- Criticism on reliability (if the FAT is lost, evrything is lost)
- The dimension of the FAT is a critical aspect

Indexed allocation

To allow an efficient and direct access it is possible to incorporate all the pointers into a table of pointers. This table of pointers is usually named `index block` or `i-node`
Each file has its own table, which is a vector of addresses of the blocks in which the file is contained

- The i-th element of the vector identifies the i-th block of the file

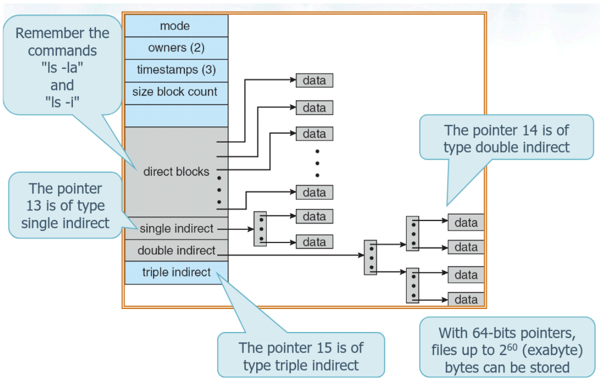


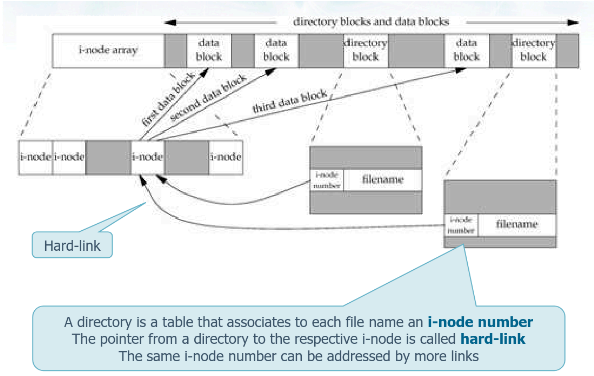
  Compared to the linked allocation, the allocation of an index block is always needed
  Index blocks of limited size allow to reduce the waste of space.
  Index blocks of extended size increase the number of references that can be inserted in the index block
  In any case, it is necessary to manage situations in which the index block is not sufficient to contain all pointers to the block of the file
  There are different schemes:
- With linked index blocks
- With multi-level index blocks
- Combined (used by UNIX/linux)

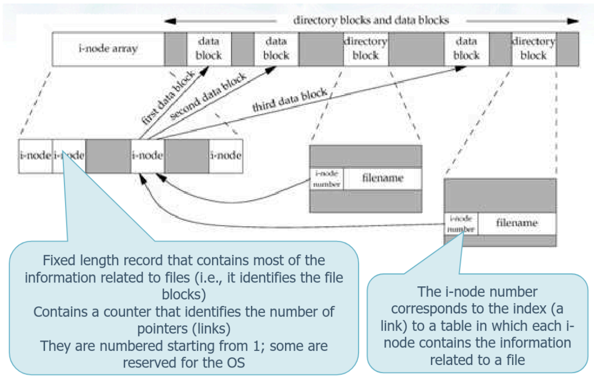
Combined scheme

To each file is associated a block named `i-node`
Each `i-node` contains different information including 15 pointers to the data blocks of the file.

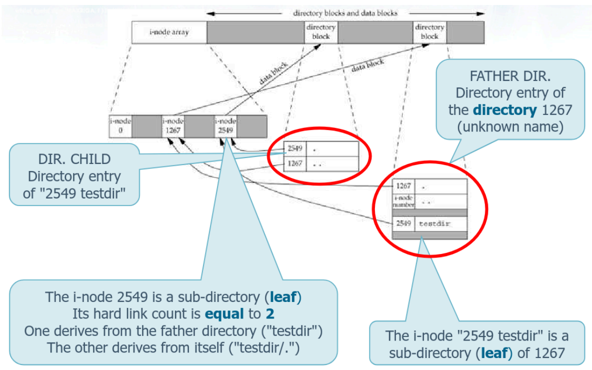
- The first 12 pointers are direct.
- The remaining are indirect pointers with increasing adressing level.
  - The block addressed by a pointer does not contain data, but pointers / pointers to pointers / pointers to pointers to pointers, to the data blocks of the file

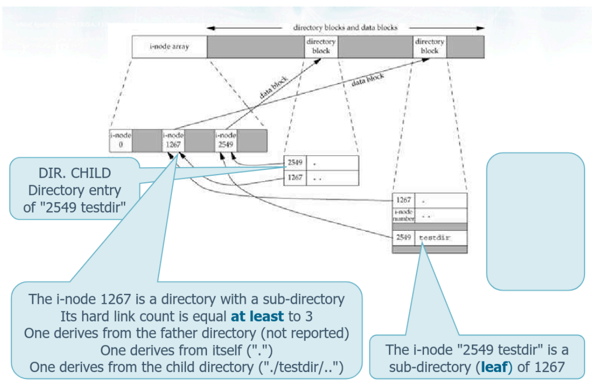






- Hard link (physical link)
  - Directory entry that points (links) an i-node
  - No hard link
    - To directory (to avoid file system with cyclic graph directories)
    - To file on other file systems
  - A file is physically removed only when all the hard links have been removed
- Soft link (Symbolic link)
  - The data block identified by the i-node points to a data block that contains the path name of the file
  - Basically, it is a file that in its only data block has the name of another file





Modern file systems

Attribute / File System	FAT32	exFAT	NTFS	Ext4
Maximum dimension of the disk	2 TB	64 ZB	2 TB (extensible to 26 TB)	1 EB

Attribute / File System	FAT32	exFAT	NTFS	Ext4
Maximum dimension of the file	4 GB	16 ZB	As much as the disk	16 TB
Main use	USB key	USB key	Internal disk of Windows	Internal disk of Linux and USB key

FAT

FAT16 (or simply FAT, 1987)

- First version, it does not support files larger than 2GByte, and a disk of maximum dimension of 32GBytes

FAT32

- Evolution of FAT16, with cluster of 32 bit, increases the support for larger files and disks

exFAT (extended FAT or FAT64, 2006)

- Increase support for larger files and disks again, designed to be light for flash drives / USB keys

NTFS

Compared to FAT, it increases the supported size. Like the latest Ext file systems, it supports
journaling and disk encryption.
It is not as dastr as FAT or Ext, but it is the stndard choice for Windows. MAC and Linux support NTFS with specific drivers

EXT

Ext (1992)

- The main lack of Ext was that it can manage a single timestamp per file, unlike the 3 timestamps we use today (creation, last modification, last access)

Ext2 (1993)

- Size extension
- It does not guarantee il journaling
  - If the computer was turned off during the writing phase, perhaps due to a power failure, the file system is corrupted, making it impossible to access the files on the disk.

Ext3 (2001)

- Fixes the problem of file system corruption
- In practice, when writing a file, it is first written to the disk, then, if the writing was successful, it is recorded on the file system
  - If the write process is interrupted without being completed, the file system remains unaffected, and the user does not notice anything

Ext4 (2006)

- It increases support for ever-increasing disk size and improves performance (i.e., increasing read and write performance in terms of speed)
- Retro-compatible with ext3

Management of the file system

The POSIX standard provides a set of functions to perform the manipulation of directories.

- The function stat
  - Allows to understand the type of "entry" (file, directory, link, etc.)
  - This operation is permitted using the C data structure returned by the function, i.e. struct stat
- Some other functions to manage the file system
  - getcwd, chdir (Positioning)
  - mkdir, rmdir (Creation/cancelation)

- opendir, readdir, closedir (Visit/inspection)

stat()

The function stat returns a reference to the structure sb (struct stat) for the file (or file descriptor) passed as a parameter.

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *path, struct stat *sb);
int lstat (const char *path, struct stat *sb);
int fstat (int fd, struct stat *sb);
```

`path` : path to return information about
`sb` : returned data structure
`returns` : 0 on success, -1 on error

`lstat` returns information about the symbolic link, not the file pointed by the link (when the path is referred to a link
`fstat` returns information about a file already opened (it receives the file descriptor instead of a path)

```
struct stat {
     mode_t st_mode; /* file type & mode */
     ino_t st_ino; /* i-node number */
     dev_t st_dev; /* device number */
     dev_t st_rdev; /* device number */
     ...
};
```

The second argument of stat is the pointer to the structure stat
The field `st_mode` encodes the file type
Some macros allow to understand the type of the file:

- S_ISREG regular file
- S_ISDIR directory,
- S_ISBLK block special file
- S_ISCHR character special file
- S_ISFIFO FIFO
- S_ISSOCK socket,
- S_ISLNK symbolic link

getcwd()

Get the path of the working directory.

```
#include <unistd.h>
char *getcwd (char *buf, int size);
```

`size` : dimension of buf
`return` : The buffer `buf` on success. `Null` on error

chdir()

Change the path of the working directory

```
#include <unistd.h>
int chdir (char *path);
```

`return` : 0 on success, -1 on error

mkdir ()

mkdir creates a new (empty) directory

```c
#include <unistd.h>
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode);
```

`return` : 0 on success, -1 on error

rmdir()

deletes a directory (if it is empty)

```c
#include <unistd.h>
#include <sys/stat.h>
int rmdir (const char *path);
```

`return` : 0 on success, -1 on error

Additional material (Not required at the exam)