File System

Directories in Linux

Stefano Quer, Pietro Laface, and Stefano Scanzio

Dipartimento di Automatica e Informatica

Politecnico di Torino

skenz.it/os stefano.scanzio@polito.it

Directories

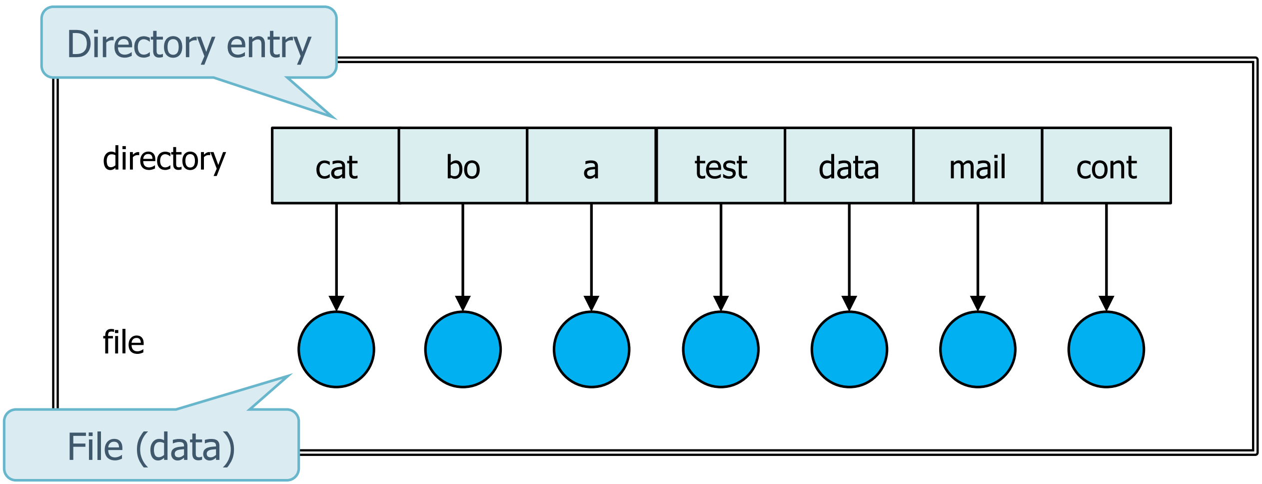
- No storage system contains a single file
- Files are organized in directories
  - A directory is a node (of a tree) or a vertex (of a graph) that stores information about the (regular) file that it contains
  - Both directories and files are saved in mass memory
- Operations that can be performed on 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
    - Simplicity for a user to identify his files
    - Allow to assign the same name to different files
  - Grouping (organization)
    - Grouping programs and data according to their characteristics
      - Editors, compilers, documents, etc.

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
  - Each name is unique within the entire file system

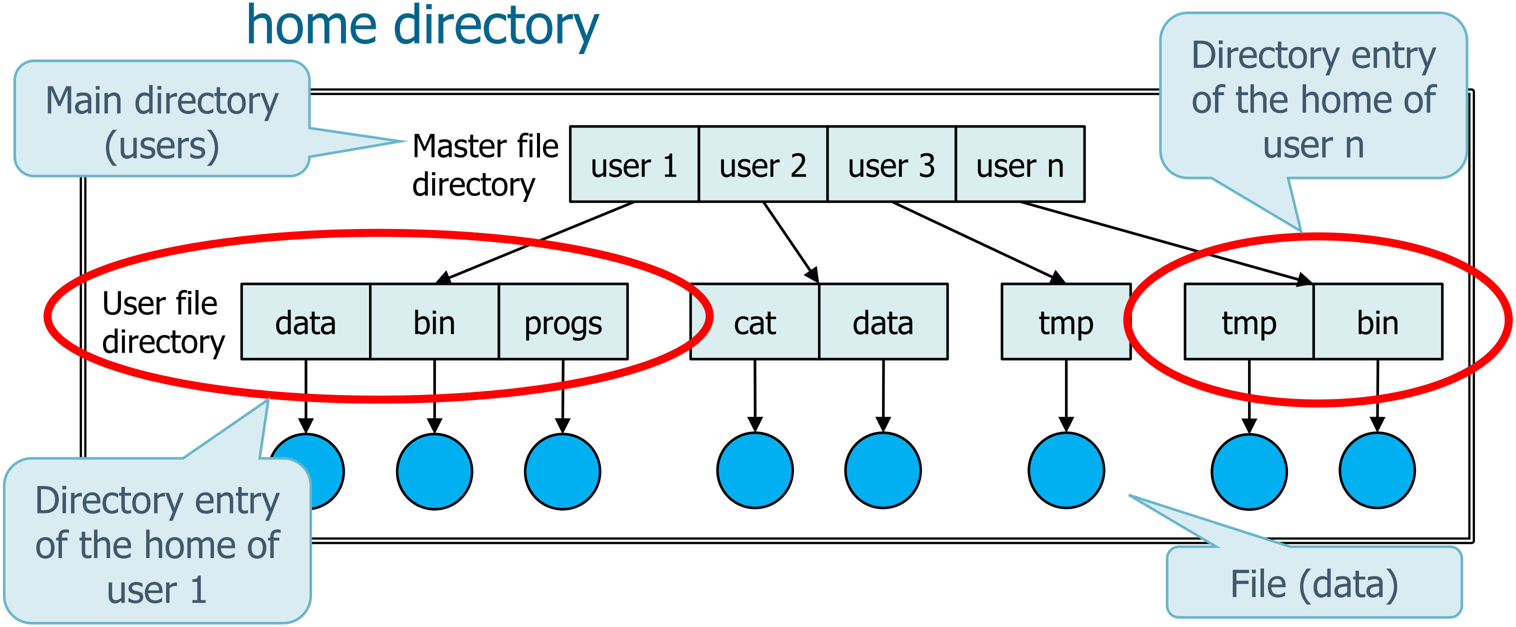


Directories with one level

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

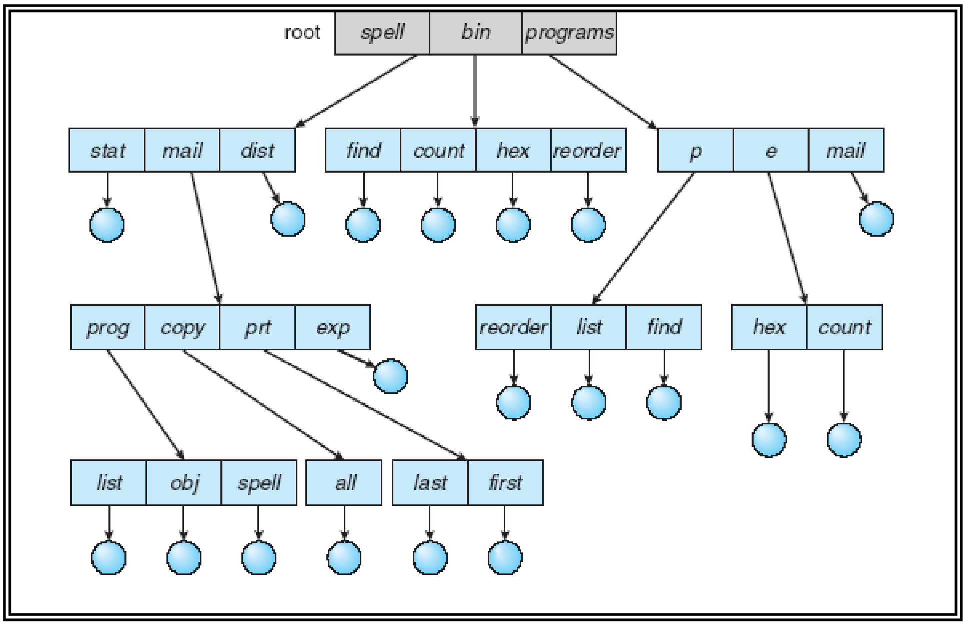


Directories with two levels

- 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



Tree directories

- Every user can manage both files and directories (and subdirectories)
  - Concept of: current work directory, change of directory, absolute and relative path name, etc.

Concepts analysed in the experimental part related to Linux

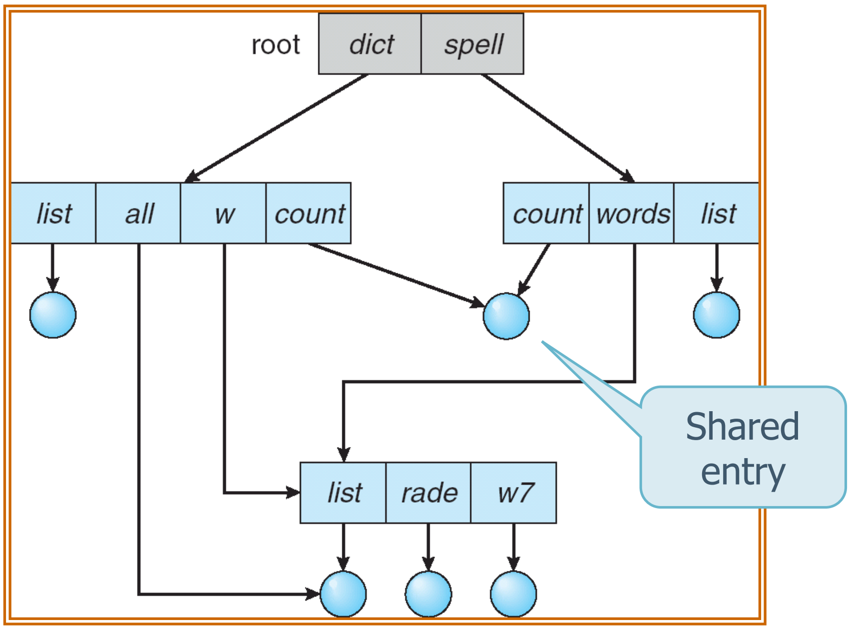
- Performance
  - Efficiency
    - Efficient searches based on the tree structure and therefore to its depth and breadth
  - Naming
    - With absolute path or relatice to the current working directory
  - Grouping
    - Extended possibilities, flexible

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 because of
    - Increase of file system occupation
    - Possible information incoherence in one or more copies

Acyclic graph directories

- Tree file systems can be generalized organizing them as acyclic graphs.
  - They allow to share information, making it visible with different paths



Acyclic graph directories

- Method
  - The sharing of an entry can be obtained in different ways
  - In UNIX-like systems, the standard strategy is the use of **links**
    - A link is a reference (pointer) to another (pre-existing) entry
  - The presence of links increases difficulty in managing file systems
    - Necessary to distinguish between native entries and relative links, during creation, modification, and removal

Acyclic graph directories

- During 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 complex

Acyclic graph directories

- 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?

Acyclic graph directories

- 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

Soft-link UNIX

Acyclic graph directories

- 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

Hard-link UNIX

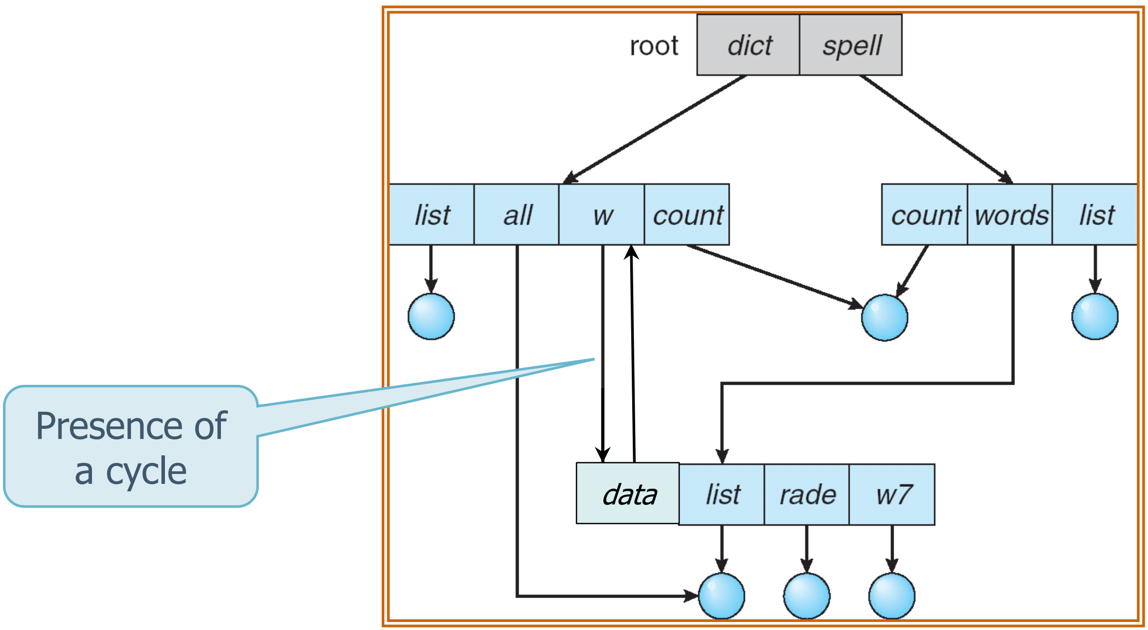
"ls -l" command

Acyclic graph directories

- 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
  - Allow the creation of cycles
  - Need to manage them appropriately in all phases



Cyclic graph directories

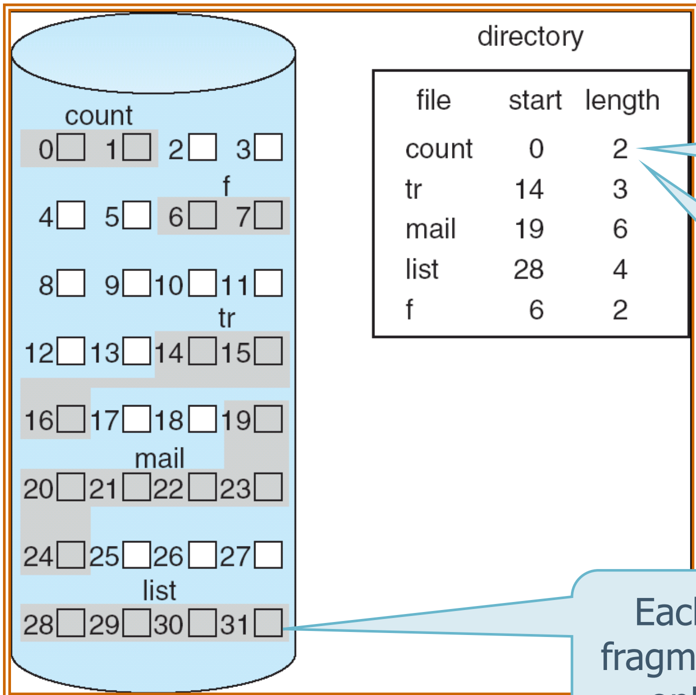
- Different approaches could be used to manage cyclic graphs
- These approaches should take into account different problematics
  - An element may self-reference itself, and never be deleted and/or detected
- The simplest method is not to visit links or sub-categories of the link

Allocation

- Allocation techniques
  - For **allocation** we mean techniques for choosing the blocks of the disks to store files
  - Observation
    - 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 thechnique
  - Contiguous
  - Linked
  - Indexed

Contiguous allocation

- Each file is stored in a contiguous set of blocks



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

For each file, the directory specifies the start address of the first block (b), and the length of the file (n)

The file use the blocks b, b+1, b+2, … , b+n-1

Each file has internal fragmentation (last block only partially used)

Contiguous allocation

- Advantages
  - Really easy allocation strategy
    - Few information is stored for each file
  - It allows immediate and sequential accesses
    - Each block is after the previous one and before the following one (i.e., blocks are consecutive)
  - It allows simple and direct accesses
    - The block i starting from block b is at address b + i-1

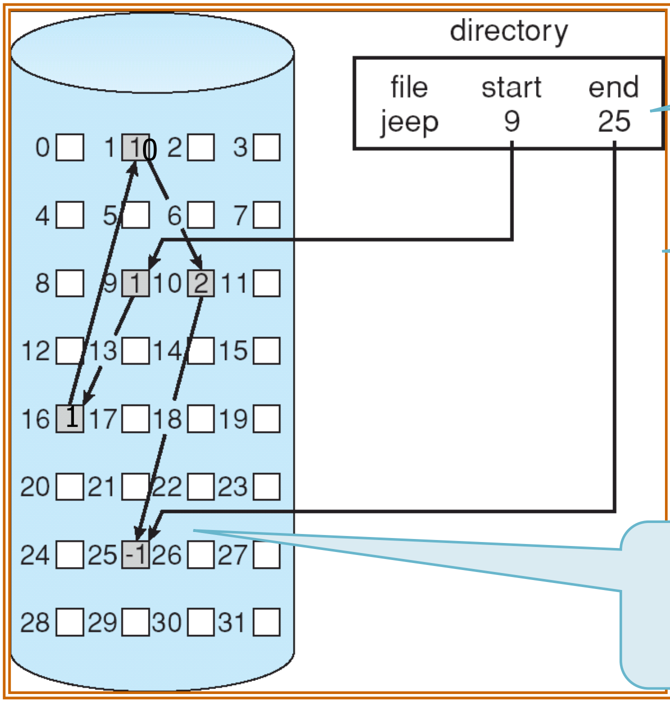
Contiguous allocation

- Drawbacks
  - An allocation policy is needed
    - Where are new files allocated?
      - Algorithms: first-fit, best-fit, worst-fit, etc.
      - How can the required space be determined?
  - No allocation algorithm is free of defects, consequently there is a waste of space
    - This waste is known as **external fragmentation**
    - Possible re-compaction (on-line and off-line)
  - Dynamic allocation problems
    - Files cannot grow indefinitely, because the available space is limited by the next file

It is necessary to find a contiguous free space of sufficient size

Linked allocation

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



The directory contains a pointer to the first and to the last block of the file

Each block contains a pointer to the next block

Blocks of each file are scattered throughout the entire disk

Linked allocation

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

Linked allocation

- Drawbacks
  - 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

Linked allocation: FAT

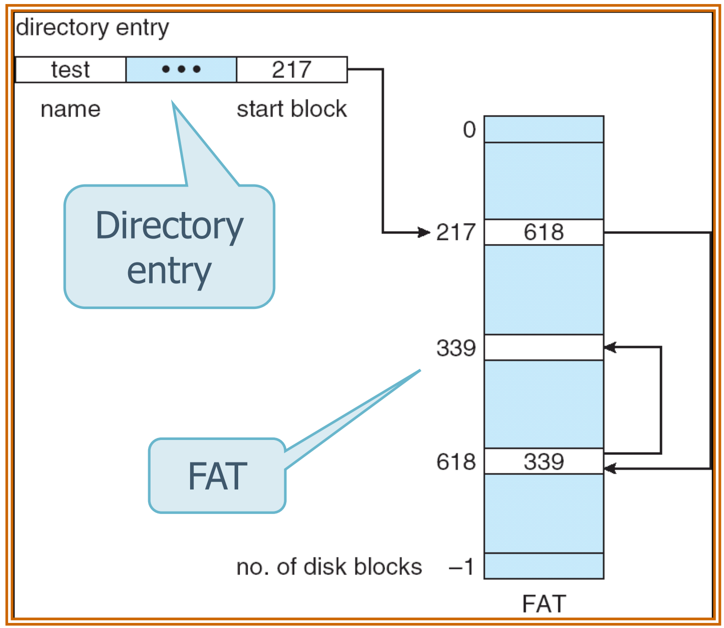
- It is the allocation used by da MS-DOS
  - Based on a FAT (File Allocation Table)
  - It is a variant of the typical linked allocation

Move pointers from the blocks to one specific block

- FAT
  - Is a table with an element for any block of the disk
  - The sequence of blocks related to a file are reported inside the directory through
    - The first element of the file in the FAT
    - Sequence of pointers located (directly) inside the FAT (instead of inside each block as in the linked allocation)

Linked allocation: FAT

- References are not stored inside the blocks on the disk but directly in the elements of the FAT
- The reading of each block requires two disk accesses (one to the FAT and one to the block to read)
  - Slow access
  - Criticism on reliability (if the FAT is lost, everything is lost)
  - What is the size of the FAT?

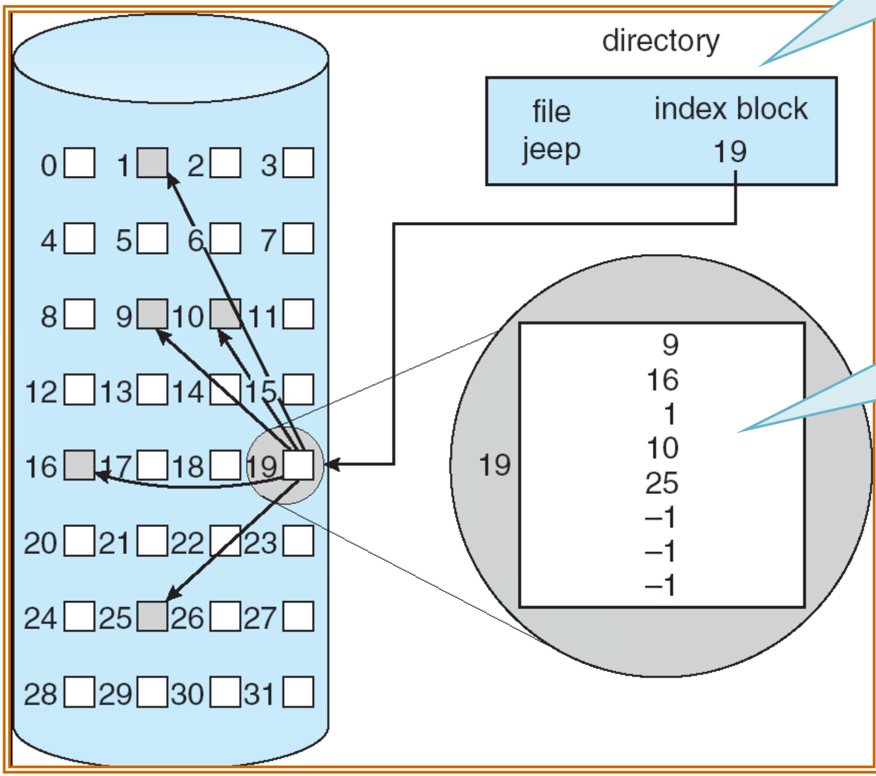


Indexed allocation

- 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

Indexed allocation

The directory contains only the pointer to the index block



It is not a FAT because pointers are all in sequence (there is **not a list** of pointers)

Indexed allocation

- 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 the pointers to the blocks of the file
    - There are different schemes
      - With linked index blocks
      - With multi-level index blocks
      - **Combined**

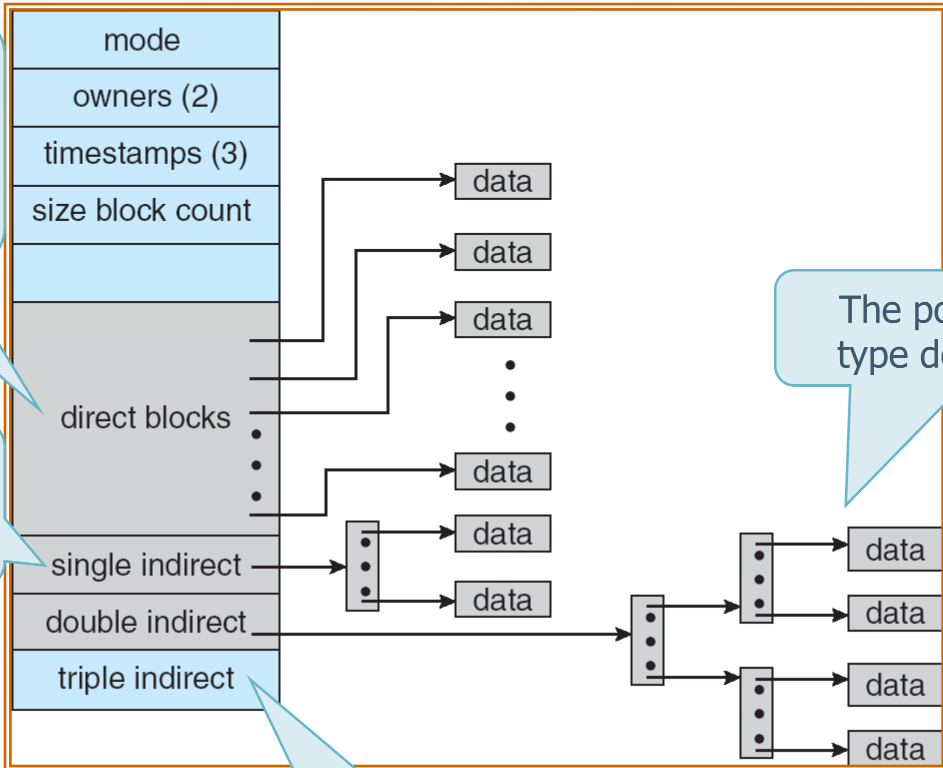
Schema UNIX/Linux

Indexed allocation: combined schema

- Combined schema is used in UNIX/Linux systems
- 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, i.e., they points to the blocks of the files
  - Pointers 13, 14 and 15 are indirect pointers, with increasing addressing 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

Indexed allocation: combined schema

Remember the commands "ls -la" and "ls -i"



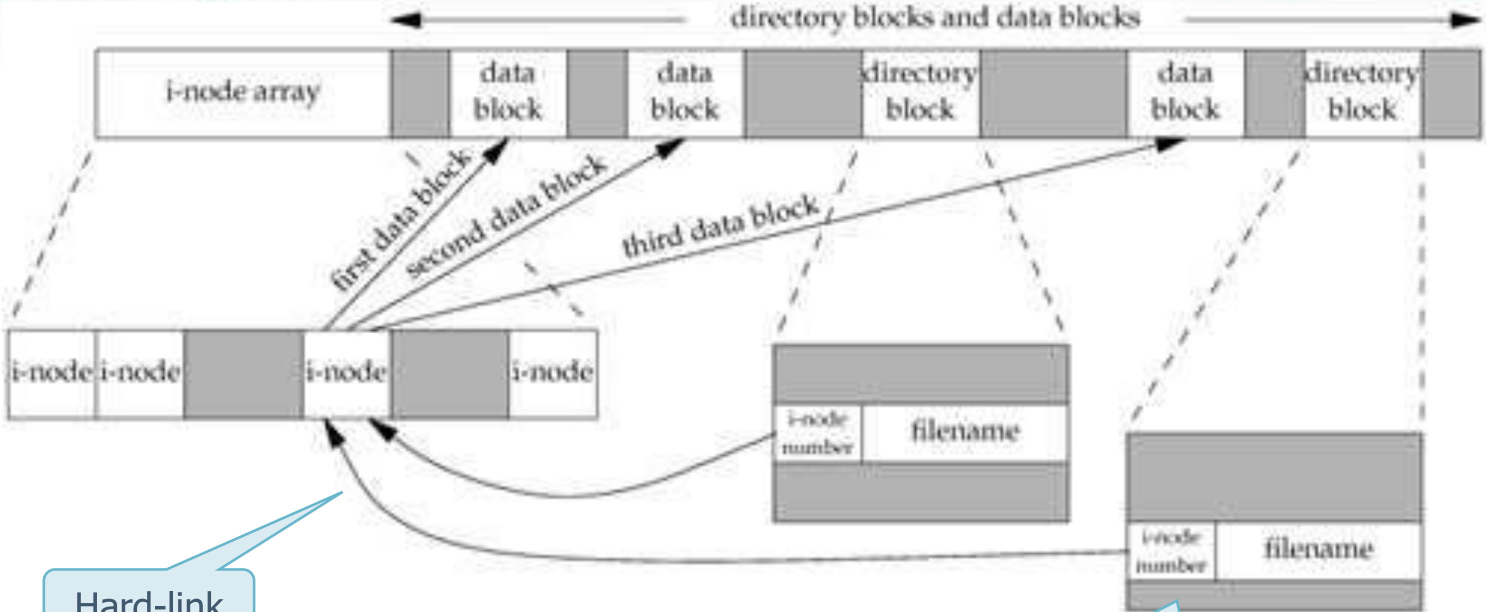
The pointer 13 is of type single indirect

The pointer 14 is of type double indirect

The pointer 15 is of type triple indirect

With 64-bits pointers, files up to 2^{60} (exabyte) bytes can be stored

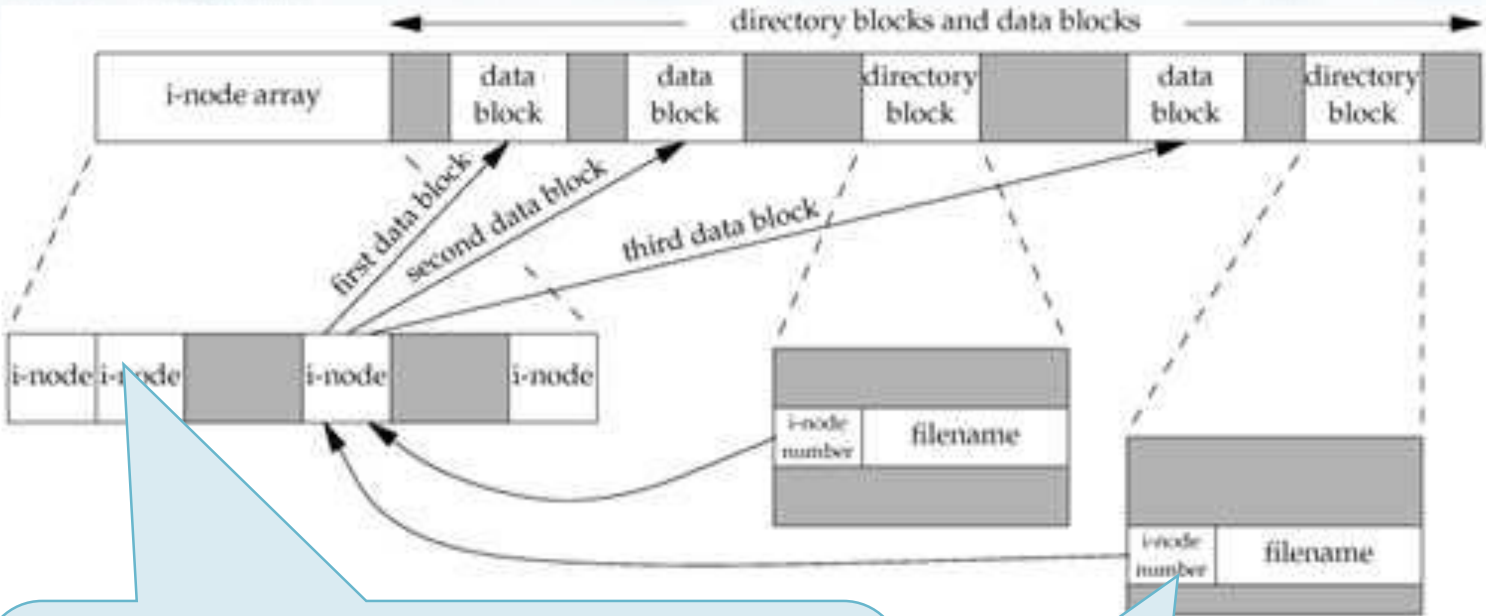
Indexed allocation: combined schema



Hard-link

A directory is a table that associates to each file name an **i-node number**
The pointer from a directory to the respective i-node is called **hard-link**
The same i-node number can be addressed by more links

Allocazione indicizzata: schema combinato



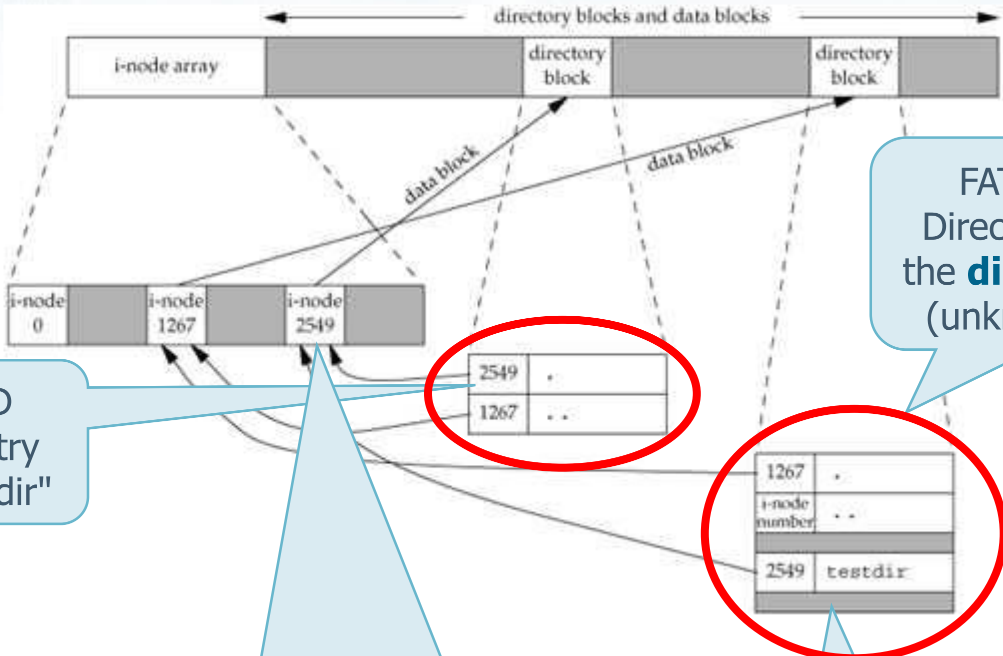
Fixed length record that contains most of the information related to files (i.e., it identifies the file blocks)
Contains a counter that identifies the number of pointers (links)
They are numbered starting from 1; some are reserved for the OS

The i-node number corresponds to the index (a link) to a table in which each i-node contains the information related to a file

Allocazione indicizzata: schema combinato

- Hard link (physical link)
  - Directory entry che 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

The UNIX file system: An example



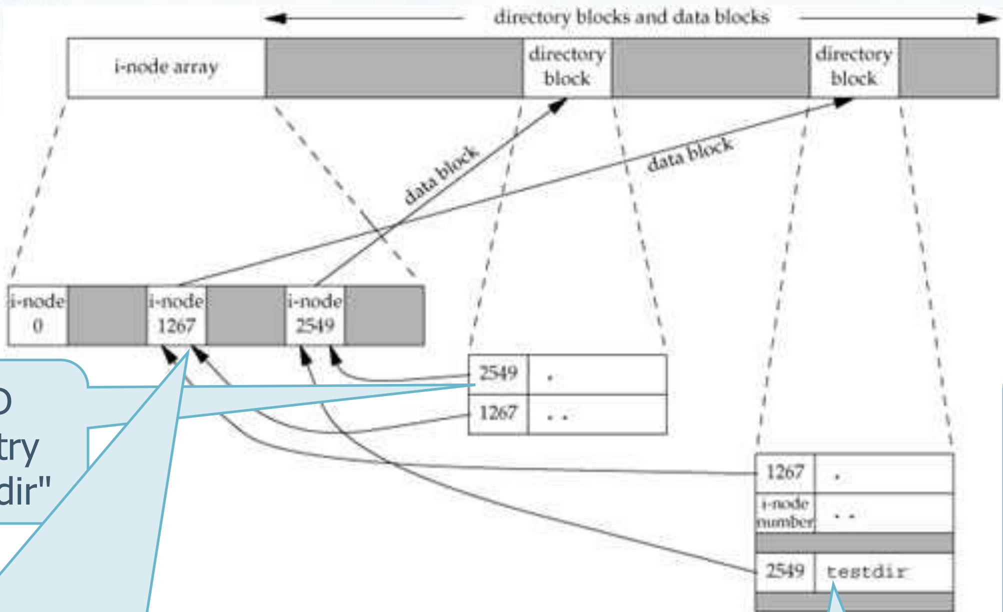
FATHER DIR.
Directory entry of the **directory** 1267 (unknown name)

DIR. CHILD
Directory entry of "2549 testdir"

The i-node 2549 is a sub-directory (**leaf**)
Its hard link count is **equal** to **2**
One derives from the father directory ("testdir")
The other derives from itself ("testdir/.")

The i-node "2549 testdir" is a sub-directory (**leaf**) of 1267

The UNIX file system: An example



DIR. CHILD
Directory entry
of "2549 testdir"

The i-node 1267 is a directory with a sub-directory
Its hard link count is equal **at least** to 3
One derives from the father directory (not reported)
One derives from itself (".")
One derives from the child directory ("./testdir/..")

The i-node "2549 testdir" is a
sub-directory (**leaf**) of 1267

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
    - mkdir, rmdir
    - opendir, readdir, closedir

Returned data structure

Positioning

Creation Cancellation

Visit / Inspection

stat ()

```
#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 to return information about

Returned data structure

- The function **stat** returns a reference to the structure **sb** (**struct stat**) for the file (or file descriptor) passed as a parameter
- Returned values
  - 0 on success
  - -1 on error

stat ()

```
#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);
```

- The function
  - **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)

stat ()

```
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

stat ()

```
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 */
  ...
};
```

- 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

Example

```
struct stat buf;
...
if (lstat(argv[i], &buf) < 0) {
  fprintf (stdout, "lstat error.\n");
  exit(1);
}
if      (S_ISREG(buf.st_mode))  ptr = "regular";
else if (S_ISDIR(buf.st_mode))  ptr = "directory";
else if (S_ISCHR(buf.st_mode))  ptr = "char special";
else if (S_ISBLK(buf.st_mode))  ptr = "block special";
else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
else if (S_ISLNK(buf.st_mode))  ptr = "symbolic link";
else if (S_ISSOCK(buf.st_mode)) ptr = "socket";
     printf("%s\n", ptr);
}
```

Allow to understand if it is a directory !

getcwd () and chdir ()

```
#include <unistd.h>

char *getcwd (char *buf, int size);

int chdir (char *path);
```


- Get (change) the path of the **working directory**
- Returned values
  - getcwd
    - The buffer buf on success; NULL on error
  - chdir
    - 0 on success; -1 on error

Example

```
#define N 100

char name[N];

if (getcwd (name, N) == NULL)
  fprintf (stderr, "getcwd failed.\n");
else
  fprintf (stdout, "dir %s\n", name);

if (chdir(argv[1]) < 0)
  fprintf (stderr, "chdir failed.\n");
else
  fprintf (stdout, "dir changed to %s\n", argv[1]);
```

mkdir () and rmdir ()

See system call open

```
#include <unistd.h>
#include <sys/stat.h>

int mkdir (const char *path, mode_t mode);

int rmdir (const char *path);
```

- **mkdir** creates a new (empty) directory, **rmdir** deletes a directory (if it is empty)
- Returned values
  - 0 on success
  - -1 on error

opendir (), dirent () and closedir ()

```
#include <dirent.h>

DIR *opendir (
  const char *filename
);

struct dirent *readdir (
  DIR *dp
);

int closedir (
  DIR *dp
);
```

Open a directory for reading
Returned values:
The pointer to the directory on success
The NULL pointer on error

Proceed with the reading of the directory
Returned values:
The pointer to the directory on success
The NULL pointer on error, or at the end of the reading operation

Terminate the reading
Returned values:
0 on success
-1 on error

dirent structure

```
struct dirent {
  inot_t d_no;
  char d_name[NAM_MAX+1];
  ...
}
```

- The structure **dirent** (**DIR ***) returned by **readdir**
  - Has a format that depends on the specific implementation
  - It contains at least the following fields
    - The i-node number
    - The file name (null-terminated)

Example

```
#define N 100
...
struct stat buf;
DIR *dp;
char fullName[N];
struct dirent *dirp;
int i;
...
if (lstat(argv[1], &buf) < 0 ) {
  fprintf (stderr, "Error.\n"); exit (1);
}
if (S_ISDIR(buf.st_mode) == 0) {
  fprintf (stderr, "Error.\n"); exit (1);
}
if ( (dp = opendir(argv[1])) == NULL) {
  fprintf (stderr, "Error.\n"); exit (1);
}
```

Structure for lstat

Directory "handle"

Structure for readdir

Ask information about the path in argv[1]

If it is not a directory, the program terminates

Otherwise, the directory is open

Example

```
i = 0;
while ( (dirp = readdir(dp)) != NULL) {
  sprintf (fullName, "%s/%s", argv[1], dirp->d_name);
  if (lstat(fullName, &buf) < 0 ) {
    fprintf (stderr, "Error.\n"); exit (1);
  }
  if (S_ISDIR(buf.st_mode) == 0) {
    fprintf (stdout, "File %d: %s\n", i, fullName);
  } else {
    fprintf (stdout, "Dir  %d: %s\n", i, fullName);
  }
  i++;
}
if (closedir(dp) < 0) {
  fprintf (stderr, "Error.\n"); exit (1);
}
```

Read the directory (iterating over all entries)

Request information about the entry fullName

Display data

Closure and termination