

# Sistemi Operativi 2

Marco Casu



# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Breve Panoramica su Unix . . . . .	3
1.2	La Shell . . . . .	3
<b>2</b>	<b>Il File System</b>	<b>5</b>
2.1	Operazioni sulle Directory . . . . .	5
2.1.1	Comandi di Base . . . . .	6
2.2	Struttura del File System . . . . .	7
2.2.1	Mounting e Partizioni . . . . .	7
2.2.2	Inode e Metadati . . . . .	8
2.3	Permessi di Accesso . . . . .	9

# 1 Introduzione

## 1.1 Breve Panoramica su Unix

Moltissimi sistemi operativi moderni, come *MacOs*, *Linux*, e molti altri, sono basati su *Unix*, un sistema operativo il cui sviluppo cominciò nel lontano 1965. Inizialmente implementato totalmente in assembly, e limitato esclusivamente ad un tipo di architettura, si decise di costruire dei linguaggi di programmazione di più alto livello per garantire la portabilità, le versioni di Unix nei linguaggi *B* e *C* permisero di portare Unix su diverse CPU.

Venne distribuito con codice sorgente a centri di ricerca ed università, si diffuse rapidamente e nacquero diverse versioni. Le caratteristiche principali di un OS basato su Unix sono le seguenti:

- Supporta più utenti e la multiprogrammazione
- Il file system ha un'organizzazione gerarchica
- Ha un kernel rappresentante il cuore del sistema
- Si interagisce col kernel tramite le chiamate di sistema
- Ha una *shell* (si vedrà in seguito)
- È modulare e fornisce ambienti di programmazione

È composto da una serie di programmi limitati che eseguono molto bene un compito specifico, presentano un output minimale, sono detti "silenziosi", e lavori più complessi possono essere svolti componendo ed articolando diversi programmi semplici. I programmi solitamente manipolano file di testo (interpretabili dall'uomo secondo la codifica ASCII) e non file binari. Qualsiasi risorsa può essere vista come file o come processo.

## 1.2 La Shell

La **Shell** non è altro che un programma che esegue dei comandi, che possono essere scritti sul terminale, esistono vari tipi di shell, come quella denominata *bash*. La *bash* scrive il cosiddetto *prompt*, ossia una dicitura che indica che il terminale è in attesa di ricevere comandi, molto spesso è costituito dal nome dell'utente, il nome del computer, ed il percorso della directory nella quale è aperto il terminale.

```
nomeutente@nomemacchina:~$
```

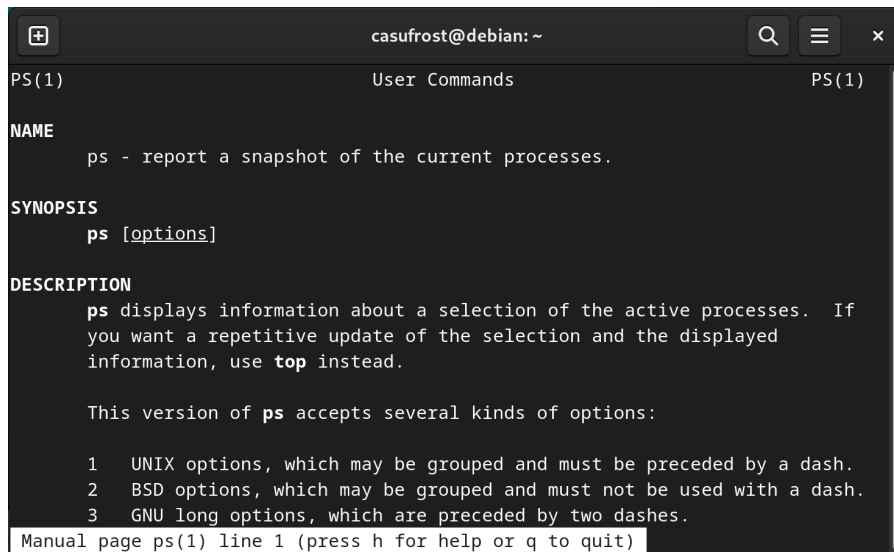
*in attesa di ricevere comandi*

Ogni **comando** seguirà il seguente template : Prima il nome del comando, poi le varie opzioni, e poi gli argomenti del comando, ci deve essere almeno un argomento obbligatorio, e zero o più argomenti opzionali, gli argomenti vanno separati da un carattere se indicato. *Esempio* :

```
casufrost@debian:~$ ps -p $$ -ocmd -h
```

Uno dei comandi fondamentali è **man**, e sta per *manuale*, è un comando fondamentale che fornisce le informazioni più autorevoli possibile riguardo un comando, basta chiamarlo, dando come argomento appunto, il nome di un comando, e fornirà una lista dettagliata di tutte le opzioni ad esso correlato.

casufrost@debian:~\$ man ps produce il seguente output :



```
casufrost@debian: ~
PS(1) User Commands PS(1)

NAME
    ps - report a snapshot of the current processes.

SYNOPSIS
    ps [options]

DESCRIPTION
    ps displays information about a selection of the active processes.  If
    you want a repetitive update of the selection and the displayed
    information, use top instead.

    This version of ps accepts several kinds of options:

    1  UNIX options, which may be grouped and must be preceded by a dash.
    2  BSD options, which may be grouped and must not be used with a dash.
    3  GNU long options, which are preceded by two dashes.

Manual page ps(1) line 1 (press h for help or q to quit)
```

Consultando il manuale è possibile capire come utilizzare i comandi fondamentali :

- **ps** - Mostra le informazioni dei processi attualmente in esecuzione.
- **ls** - Mostra una lista delle risorse contenute in una directory.
- **cp** - Copia file e directory.

Analizziamo il comando di esempio scritto in precedenza, ossia:

casufrost@debian:~\$ ps -p \$\$ -ocmd -h tale comando mostra i processi attualmente in esecuzione, l'opzione serve a selezionare il processo in base al suo PID, e prede come parametro \$\$, ossia il codice identificativo del processo bash.

Il comando mostra varie informazioni, come il PID del processo ed il nome, L'opzione **-o** serve a selezionare solo uno dei parametri del processo, in questo caso prende come argomento **cmd** e seleziona quindi solo quel campo, ossia il nome. Si noti come in questo caso non è necessario lasciare uno spazio fra l'opzione e l'argomento. L'ultima opzione, ossia **-h**, serve a rimuovere il nome dei campi visualizzati, ci si aspetta quindi che tale comando in output restituirà esclusivamente il nome del processo bash :

```
casufrost@debian:~$ ps -p $$ -ocmd -h
bash
```

Un altro esempio :

```
casufrost@debian:~$ ps -p $$
  PID  TTY      TIME  CMD
 3134 pts/0    00:00:00 bash
```

Durante la configurazione di un sistema Unix è necessario specificare almeno un utente. Differenti utenti hanno differenti privilegi, l'utente **root**, o **superutente**, è predefinito in ogni

sistema e possiede tutti i privilegi, è detto *amministratore di sistema*.

Tale utente però, non può effettivamente eseguire un login ed entrare nel sistema, se necessario eseguire un'operazione privilegiata, gli altri utenti possono acquisire temporaneamente i diritti di amministratore tramite il comando **sudo**, acronimo di *super user do*.

Gli utenti appartengono a dei *gruppi* che definiscono diversi privilegi, coloro che possono richiedere temporaneamente i diritti di amministratore appartengono al gruppo dei *sudoers*, è possibile mostrare a quali gruppi appartiene un utente tramite il comando **groups**.

**sudo** è un comando che prende come input un altro comando da eseguire in modalità root, esiste anche un altro comando chiamato **su**, e sta per *substitute user*, e serve per cambiare utente, e diventare possibilmente amministratore, risulta comunque meno sicuro del comando **sudo**, in quanto quest'ultimo permette solo di eseguire un'operazione in maniera privilegiata, senza rimanere nella condizione di root.

## 2 Il File System

La gestione dei file in un sistema Unix non è delegata al kernel, essi non risiedono infatti in quest'ultimo, e più file system possono coesistere nello stesso sistema : diversi dischi (o altre unità di archiviazione) possono gestire i file in maniera diversa. Tutti i file convivono sotto la stessa cartella radice, detta *root*, in maniera totalmente trasparente.

Il file system gestisce la memoria secondaria ed è organizzato in maniera **gerarchica**, vi è una *directory* (cartella) principale che si trova alla radice dell'albero, che si dirama in più directory fino ad arrivare alle foglie, ovvero i file (che non possono ospitare altri file), è una struttura ricorsiva. Ci sono due tipi di file in Unix :

- **file non regolari** - Quei file che danno un *accesso astratto* a periferiche e dispositivi, come già accennato, in Unix tutto può essere visto come un file o come un processo, de facto anche lo stesso mouse viene modellato come un file, e sarà appunto un file di questo tipo.
- **file regolari** - Tutti i restanti file ordinari, come un file di testo o l'eseguibile di un programma.

La directory root è indicata dal simbolo **/**, tutto è contenuto in tale cartella, anche una chiavetta USB che viene inserita nella macchina, i drive riceveranno una cartella sotto la root. Il file system impone alcune restrizioni sui nomi dei file :

- Non è possibile creare due file con lo stesso nome nella stessa directory.
- Non è possibile creare due directory con lo stesso nome nella stessa directory.
- Non è possibile creare una directory ed un file con lo stesso nome nella stessa directory.

### 2.1 Operazioni sulle Directory

Ogni singolo file del file system è raggiungibile mediante un cammino detto *path*, una sequenza di nomi di directory separate dal carattere **/** che indicano la locazione del file, tale path può

essere assoluto o relativo. In ogni sistema Unix, la directory *home* dell'utente può anche essere indicata con il carattere `~`, infatti, i seguenti due path sono equivalenti :

```
~/Immagini(faces)
home/Immagini(faces)
```

Nella shell, viene sempre visualizzata la directory corrente nella quale ci si trova, che di default è home appena si apre il terminale, cambiando la directory cambierà anche la dicitura che indica la current directory sul prompt.

### 2.1.1 Comandi di Base

Il comando `cd`, che sta per *change directory*, serve a cambiare la directory corrente, e richiede come argomento un path, che può essere assoluto o relativo, se relativo, si riferirà alle directory contenute nella current directory, altrimenti si può dare il cosiddetto path assoluto, specificando la posizione a partire dalla root.

```
casufrost@debian:~$ cd Immagini      per spostarsi nella cartella immagini
casufrost@debian:~/Immagini$
```

Se non si passa alcun argomento, il comando sposta l'utente nella home. È possibile riferirsi alla directory padre con `cd ..` o alla directory corrente con `cd .`.

È possibile creare nuove cartelle con il comando `mkdir`, che sta per *make directory*, prende come argomento il path di una cartella, se essa non esiste, verrà creata, è possibile utilizzare l'opzione `-p` per creare tutto un intero path specificato, creando l'intero cammino in un solo comando.

Un path assoluto è valido in qualunque caso, un path relativo è ovviamente dipendente dalla current directory, potrebbe quindi risultare non valido. Supponiamo che nel file system esista il seguente path : `home/animal/dog`, si avrà che :

```
casufrost@debian:~$ cd dog  Non è valido, funzionerebbe se fossimo in animal
casufrost@debian:~$ cd /home/casufrost/animal/dog  assoluto, quindi valido
```

Un altro comando estremamente utile è `ls`, che sta per *list segments*, prende come argomento una directory (se omesso considera quella corrente), e mostra tutti i contenuti all'interno di essa, l'opzione `-l` mostra ulteriori informazioni utili, come i permessi di lettura o scrittura (verrà approfondito in seguito).

In Unix i file che iniziano con il punto, ad esempio `.secret.txt` sono nascosti, è possibile visualizzarli con il comando `ls -a`.

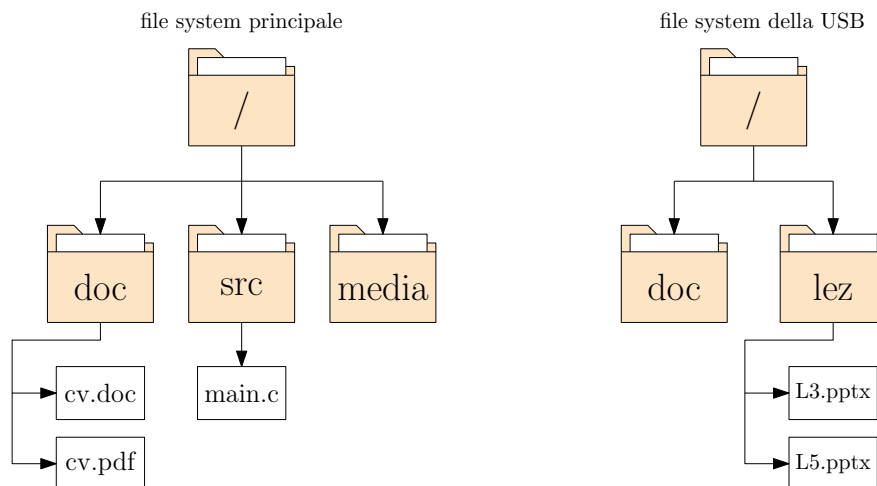
Esistono altri comandi secondari piuttosto utili : `tree` mostra l'alberazione della directory, `touch` crea un file vuoto, `pwd` stampa a schermo la directory corrente.

## 2.2 Struttura del File System

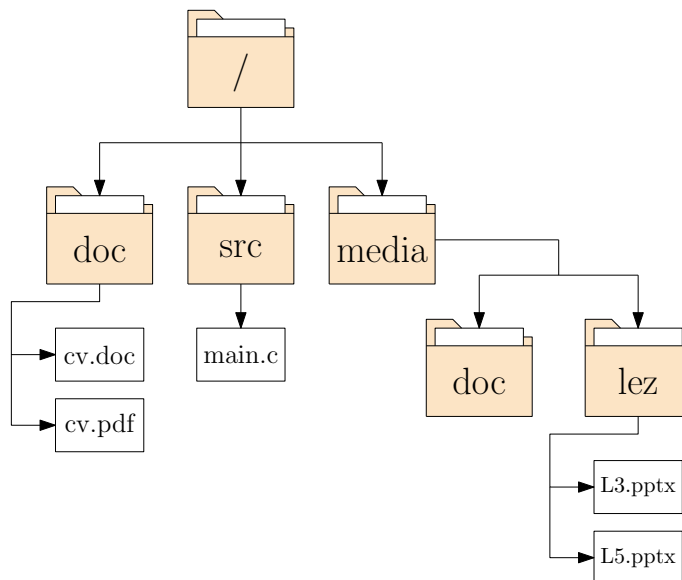
### 2.2.1 Mounting e Partizioni

Il file system può contenere dischi di memoria secondaria, anche dischi in rete, o addirittura una porzione della RAM, tutto grazie al meccanismo di **mounting**, ossia l'operazione di attaccare la radice del file system di un'unità secondaria, ad una cartella appartenente al file system principale, ossia quello della radice root.

Inserendo la USB nella macchina, ottengo i due seguenti file system



Eseguo il mounting della USB ed ottengo un unico file system :



Una qualsiasi directory dell'albero principale può diventare un *punto di mount* di una qualsiasi altro file system, l'unico fatto da considerare è il seguente : Se viene montato un disco su una directory contenete dei file, tali file non saranno accessibili finché il mounting verrà rimosso, è quindi opportuno montare i dischi su directory vuote.

Un disco, può essere anche **partizionato**, ossia diviso in unità logiche disgiunte che il sistema operativo vedrà come dischi differenti, è solito partizionare il disco in modo da avere un unità per il sistema operativo (che viene montata sulla root) ed un unità per i dati dell'utente (che viene

montata sulla home). Partizionare il disco risulta comodo ed aumenta il grado di sicurezza nell'organizzazione dei file, se dovesse accadere qualcosa all'unità dell'OS, i dati utente non rimarrebbero corrotti.

Linux funziona con diversi tipi di file system, un file system piuttosto utilizzato è quello di tipo *Journal*, esso tiene traccia di tutte le operazioni da eseguire sul disco, in caso di problemi ad esso, consultando il "diario", è possibile avere informazioni riguardo le operazioni eseguite, e le operazioni da eseguire ma non completate, riducendo il danno.

Diversi file system differiscono in caratteristiche quali :

- Dimensione massima delle unità di una partizione
- Dimensione massima di un file
- Lunghezza massima per il nome di file o directory

In unix esistono due importanti file che tengono traccia dei dischi montati, il file `/etc/mtab` contiene informazioni sui dischi montati all'avvio della macchina, `/etc/fstab` contiene informazioni sui dischi montati dinamicamente dall'utente, è possibile visualizzare il contenuto di questi file (come di un qualsiasi altro file) tramite il comando `cat`.

### 2.2.2 Inode e Metadati

Come vengono gestiti i file all'interno del file system, o meglio, quale struttura dati è impiegata? Esiste una struttura chiamata **inode**, contenente ogni singolo file, identificato da un codice univoco detto *inode number*, fisicamente quindi non vi è un'implementazione ad albero/gerarchica.

Quando un file viene eliminato, non viene realmente cancellato fisicamente, viene solamente etichettato come "libero" l'inode number ad esso associata, permettendo a nuovi eventuali file di venire associati a quel numero. Il numero totale di entrate nell'inode è elevatissimo ma limitato, esiste quindi un limite al numero di file che possono coesistere nel sistema.

Ogni file ha quindi un'entrata nell'inode, che ne specifica il codice, ed altri *metadati* utili riguardanti il file, essi sono :

- Type - il suo tipo, se regolare o non regolare
- User Id - l'Id dell'utente proprietario del file
- Group Id - l'Id del gruppo primario del proprietario
- Mode - permessi di lettura, scrittura ed accesso, riguardanti il proprietario, il gruppo del proprietario e tutti gli altri utenti
- Size - le dimensioni in byte del file
- Timestamps - 3 marcature temporali riguardanti l'istante di cambiamento di un metadato, modifica e lettura del file.
- Data pointers - puntatore alla lista di blocchi che compone il file.

Le directory contengono informazioni sugli inode number dei file che contengono, permettendo al sistema di seguire i percorsi specificati dall'utente. Tramite il comando `ls -li`, è possibile visualizzare l'inode number nel terminale.



## 2.3 Permessi di Accesso

Nei metadati di un file, sono contenute delle informazioni riguardanti l'accesso, appunto del file, da parte dei vari utenti che co-abitano l'ambiente di sistema.

Supponiamo che il sig. Rossi per non perdere traccia dei suoi accessi ai vari servizi online, si scriva tutte le sue password su un file di testo, che salva sul desktop. Il sig. Verdi, accede al medesimo sistema (ovviamente con un utente differente), esso può visualizzare le password del sig. Rossi?

Ogni file contiene dei bit che definiscono i permessi di accesso per i vari utenti, esistono 3 diversi tipi di accesso :

- **r** - *read*, accesso in lettura
- **w** - *write*, accesso in scrittura
- **x** - *execute*, accesso in esecuzione

Quest'ultimo è valido per le directory, una directory con l'accesso **x** può essere navigata. Per un file eseguibile non ha senso l'accesso **x** singolarmente, in quanto è necessario anche che il file venga letto. Quindi ogni file ha 3 bit, che ne identificano gli accessi, essendo 3 bit, possono anche essere visti come un numero intero da 0 a 7.

	4	2	1	permessi
0	-	-	-	nessun permesso
1	-	-	x	solo esecuzione
2	-	w	-	solo scrittura
3	-	w	x	scrittura ed esecuzione
4	r	-	-	solo lettura
5	r	-	x	lettura ed esecuzione
6	r	w	-	lettura e scrittura
7	r	w	x	tutti i permessi

Il fatto è che non esiste una sola tripla di bit, ma ne esistono ben 3, la prima identifica i permessi per l'utente proprietario del file in questione, la seconda identifica i permessi per gli utenti che appartengono al gruppo primario del proprietario del file, la terza identifica i permessi per tutti gli altri utenti. Ad esempio, se il file `frost.txt` ha i permessi `rwX,rwX,r--` o `774` può essere letto, scritto ed eseguito dal proprietario e dai membri del suo gruppo primario, mentre può essere esclusivamente letto da tutti gli altri utenti.

Esistono inoltre 3 ulteriori bit nei metadati di ogni file che consentono ulteriori permessi :

- **sticky bit** - Serve a proteggere le directory ed il loro contenuto dagli utenti non proprietari, un file in una directory può essere acceduto da chi ha i permessi della directory ma non del file, con lo sticky bit, sarà necessario essere proprietario del file in questione.
- **setuid bit** - Utilizzato esclusivamente per i file eseguibili, permette a coloro che eseguono il file in questione di ottenere momentaneamente gli stessi permessi del proprietario del file, ad esempio il comando `passwd` sfrutta tale bit, e permette ad un utente di modificare la propria password anche se il proprietario del file è l'utente root.
- **setgid bit** - Analogo al precedente, ma riguardante il gruppo.