Linux per High Performance Computing

Giacomo Trinca Cintioli

10 gennaio 2023

Indice

Parte I

Introduzione a Linux

Capitolo 1

Linux - Cenni Storici

1.1 Predecessori di Linux

1.1.1 1969 - Unix

Unix è stato un sistema operativo sviluppato nei laboratori Bell. Esso parte da un progetto preesistente, chiamato Multics[1]. Quest'ultimo fu abbandonato data la complessità e l'eccessiva difficoltà. La vera svolta fu data dall'invenzione del linguaggio C, ad opera di Thompson e Ritchie (1969-73). Questo ha permesso di portare il kernel su piattaforme diverse dal dispositivo orgiinario, costituendo di fatto il primo software della storia ad essere in grado di funzionare su più ambienti diversi.

1.1.2 1990 - POSIX Standard

POSIX (Portable Operating System Interface for UNIX) è una famiglia di standard definiti dall'IEEE[2] il cui compito è quello di definire alcuni concetti base che vanno seguiti durante la realizzazione del sistema operativo. Esso standardizza, ad esempio, l'interfaccia di UNIX. Molti dei comandi di questo corso, e molte delle cose che ancora utilizziamo per costruire e/o modificare i sistemi operativi sono delle direttive standardizzate POSIX.

1.2 GNU/LINUX

Quando parliamo di linux, generalmente, ci riferiamo al cosiddetto "Kernel Linux". Esso è la il cuore del sistema operativo, ovvero la sua parte essen-

[1] Curiosità: tale progetto nasce dalla necessità di connettere due stampanti diverse.
[2] Institute of Electrical and Electronics Engineers

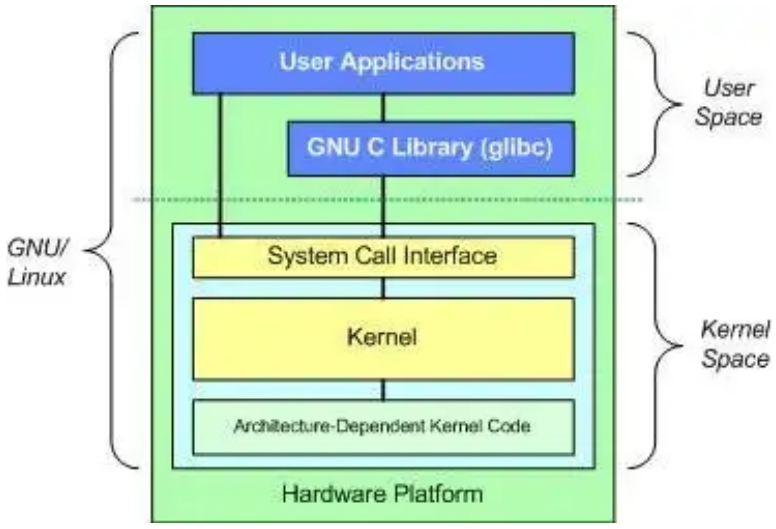

Figura 1.1: Schema semplificato dell'interfaccia tra kernel, la libreria GNU C e le applicazioni dell'utente in un sistema operativo linux.

ziale. Con GNU (GNU's Not UNIX) ci riferiamo invece ad un set di utilities, comandi e altre componenti costruite sopra al kernel, e rimangono sempre separato da esso. Questo ci permette ad esempio un update delle librerie di compilazione separato dall'update del kernel.

1.2.1 Distribuzioni Notevoli

- Red Hat Enterprise Linux (Server)
  Vi sono delle alternative Desktop, utilizzate spesso per testare nuove features e pacchetti prima di includerli nella versione Enterprise. Le maggiori in voga sono:
  - Fedora
  - CentOS
- Debian (Desktop)
  Debian è un sistema operativo completamente libero, ed usa solo software e driver open source. Vi sono delle alternative, ad esempio Ubuntu, incentrata sull'utilizzo per utenti non esperit della riga di comando. Da Ubuntu sono derivate altre distribuzioni, come Mint, la quale è incentrata sulla leggerezza dell'interfaccia, oppure Kali, utilizzata per hacking e cyber security.

  - Ubuntu
  - Kali
  - Linx Mint

- Suse (Dektop / Server)
  L'alternativa a Red Hat per il mondo server. Come quest'ultimo, anche Suse è un sistema operativo modulare. Suse ha anche una controparte open source

  - OpenSUSE

- Arch Linux (Desktop)
  La distribuzione linux per eccellenza. Incentrata sul testing di nuove features del kernel linux e dei nuovi pacchetti. Arch Linux è il sistema operativo più malleabile e personalizzabile. Il contro è la modalità di installazione, che avviene solo tramite linea di comando. Da questa, sono derivate diverse distribuzioni, volte a ridurre la difficoltà di installazione di mantenimento.

  - Manjaro
  - Garuda
  - Endevour OS

Capitolo 2

Riga di Comando

La riga dei comandi, o prompt dei comandi (CLI - Command Line Interface, Condole, Terminal, Shell) è la via più diretta per comunicare con il sistema operativo. I vantaggi sono:

- E' semplice, ed a meno che non si commettono errori, funziona sempre.
- E' facile da programmare (rispetto alle applicazioni con interfaccia grafica (GUI))
- E' veloce nell'esecuzione dei comandi impartiti, ed efficiente nel gestire le risorse necessarie per far girare il programma.

Un altro grande vantaggio è che se il sistema operativo in qualche modo si rompe, la riga di comando è sempre accessibile (basta digitare Ctrl+Alt+F2 all'avvio del sistema operativo) attraverso quella che si chiama shell di login tty. Uno degli "svantaggi" è che bisogna conoscere i comandi. Ad esempio, se si vuole copiare un file in una cartella, si deve conoscere il comando di copia

```
cp nome_file_vecchio ~/cartella/nome_file_nuovo
```

stessa cosa se si volesse creare una cartella

```
mkdir nome_cartella
```

La riga di comando può essere programmata attraverso degli script bash. E' importante sapere sempre dove ci trova nella shell, o meglio sapere qual è la nostra *working directory*. Dobbiamo anche sapere quali comandi verranno stoppati se chiudiamo la nostra shell.

2.0.1 Elementi della shell

Una volta aperta la shell `tty` , o un emulatore del terminale in linux, il prompt dei comandi si presenta nella forma:

```
[username@login_shell ~] $
```

dove lo user name è il nome dell'utente che ha effettuato il login sulla shell, e `login_shell` è il nome con il quale il personal computer viene identificato nella rete locale, anche detto host name. Il simbolo `~` identifica la `home directory`, che si trova in `/home/nome_utente/home/`. Per convenzione si utilizza il simbolo `$` per identificare un utente standard, mentre `#` per identificare un utente con privilegi di root. Un utente con privilegi di root può scrivere nelle cartelle diverse dalla home directory.
La struttura tipica di un comando `bash` è la seguente

```
[username@login_shell ~] $ hostname -f
```

per eseguire il comando basta premere `Enter` sulla tastiera. Tale comando stampa su schermo il nome del computer, l'output sarà dunque

```
login_shell
```

vediamo che accanto al comando `hostname` viene aggiunta un'opzione `-f`. I comandi hanno tutti differenti opzioni, le quali possono essere stampate con l'ausilio del comando `man`, oppure aggiungendo l'opzione `--help`.
Una volta che il comando è stato eseguito, sarà di nuovo visibile il prompt di `bash`, dove sarà possibile digitare un nuovo comando. Finché non è stato eseguito un comando, non sarà possibile eseguire nessun altro comando, a meno che non si apra una nuova shell.
Possiamo navigare nella storia dei comandi della nostra shell utilizzando le frecce della tastiera, senza dover riscrivere il comando da capo. Se dobbiamo quindi eseguire più volte lo stesso comando possiamo cercarlo nella nostra `bash_history`.
Il tasto `Tab` serve per l'autocompletamento (Tab completation) dei comandi. Se vogliamo che funzioni, dobbiamo scrivere abbastanza lettere per far si che il sistema operativo trovi il comando, che, come vedremo più in là sarà collegato ad un file binario all'interno di `/usr/bin` o `/usr/sbin`. Premendo una seconda volta `Tab` è possibile vedere una lista di comandi compatibili con le lettere già digitate nel prompt di `bash`.
L'ultima combinazione di tasti notevole è `Ctrl-C`, la quale ferma il comando che è attualmente in esecuzione. Questo fa si che il comando in esecuzione venga fermato con un messaggio di errore, spesso identificato come

`Keyboard Abort`.
La riga di comando è sempre *case sensitive*, ovvero fa differenza tra caratteri maiuscoli e minuscoli. Molto spesso le opzioni dei comandi possono essere scritte in più modi. Possiamo ad esempio scrivere

```
sbatch --time 0:30:00
```

oppure

```
sbatch -t 0:30:00
```

questi due comandi sono identici. Il primo viene detto "full name command", mentre il secondo viene chiamato "shorthand command".
Uno svantaggio per gli utenti poco esperti della riga di comando è l'assenza totale del tasto `Annulla`. Ovvero, una volta mandato in esecuzione un comando non si può tornare indietro. Conseguentemente non vi saranno finestre di dialogo che chiedono se si è sicuri di voler procedere. Ad esempio, se vogliamo eliminare un file utilizzando il comando `rm` la shell non ci chiederà se siamo sicuri di volerlo fare, e procederà con l'esecuzione del comando.
Questo potrebbe causare, nell'eventualità di utenti con permessi di root, gravi problemi se si eliminano o modificano files necessari al sistema operativo, ad esempio come[1]

```
sudo rm -rf /
```

Morale: mai mandare in esecuzione un comando se non si sa cosa esso faccia esattamente.

[1] Mai mandare in esecuzione questo comando!

Capitolo 3

Struttura delle Cartelle in Linux

3.1 Struttura ad albero - Filesystem Tree

Diversamente dai sistemi Windows, dove i volumi montati nel sistema (sia interni che esterni) sono etichettati da lettere (come ad esempio `C:/`), su Linux esiste una sola root directory `/`. Il livello più alto di questo albero di cartelle è uguale in tutti i sistemi linux, ovvero è standardizzato. I volumi esterni e interni vengono montati in delle cartelle (spesso `/mnt`), chiamati mounting points, ovvero punti di montaggio. Il percorso di un file può essere identificato in modo assoluto, ad esempio

```
/home/nome_utente/Documenti/documento.pdf
```

oppure in modo relativo sapendo quel è la working directory della nostra shell. Per sapere qual è la working directory si usa il comando `pwd`.

Come vediamo nella Figura 3.1 vi sono molte cartelle, che riassumiamo qui:

- `/`: è la cartella di livello più alto, la quale contiene tutte le altre cartelle.
- `bin/`: qui vi sono tutti i programmi (come le shells, ad esempio)
- `etc/`: qui vi sono tutti i files di configurazione
- `home/`: qui è dove l'utente spende la maggior parte del tempo. Vi sono tutti i files personali dell'utente, ed è l'unica cartella dove un utente non root può scrivere. Molte distribuzioni linux, tra le quali

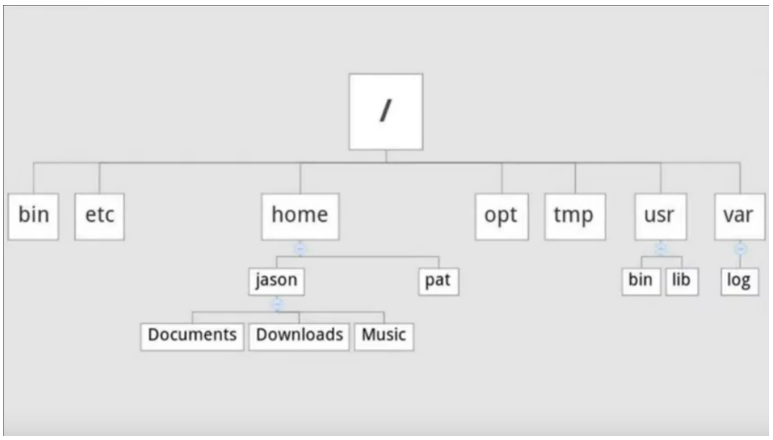

Figura 3.1: Tipica struttura ad albero di un sistema Linux.

OpenSUSE, crea una partizione dedicata per questa cartella per aumentare la stabilità e la sicurezza del sistema. In questa cartella vi è una sottocartella per ogni utente del sistema.

- `opt/`: in questa cartella ci sono i programmi opzionali. Come vedremo, ad esempio in questa cartella potranno essere caricati driver opzionali (ad esempio quelli della scheda grafica, bluetooth, etc.)

- `tmp/`: contiene i files temporanei utili ai programmi per poter girare. Ad esempio un estrattore di archivi zip può utilizzare questa cartella per contenere l'archivio da estrarre.

- `usr/`: acronimo di "unix system resources", dove vi sono ancora altri programmi e librerie.

- `var/`: contiene dei files che cambiano nel tempo (variable files) come ad esempio i file di log, che possono essere consultati in caso di errore di esecuzione di qualche comando o programma.

Ogni cosa in linux è rappresentato sottoforma di file, molto spesso sotto forma di file di testo. Ad esempio, la configurazione di una shell grafica (ad esempio i3, gnome, kde) può essere configurata a partire da un file di configurazione senza passare per l'applicazione grafica che permette di personalizzare le impostazioni grafiche. Nel filesystem tree vi sono anche altre cartelle, il quale nome può variare a seconda della distribuzione. Ad esempio nella cartella `/dev/` ci sono i files relativi ai dispositivi collegati

alla scheda madre (ad esempio hard disks, pendirves, lettori DVD) mentre nella cartella `/proc/` vi sono tutti i files relativi alle informazioni del sistema operativo.

Ogni comando che viene scritto nel prompt di bash è un programma o uno script, che giace in una cartella del filesystem. Per scoprire dove risiede tale programma o script basta scrivere

```
which <comando>
```

che darà come output

```
/cartella/.../<comando>
```

ad esempio se apriamo un shell bash e digitiamo il comando `firefox`, l'hostname command `firefox` punterà al programma Firefox che aprirà una finestra con il browser internet. Vediamo alcune abbreviazioni per le cartelle nella shell di bash:

- `.` indica la cartella corrente (working directory)
- `..` indica la cartella padre della cartella in cui siamo
- `~` indica la cartella `/home/nome_utente/`.

Per navigare nel filesystem si usa il comando `cd`, che sta per change directory, che è parte dello standard POSIX, ovvero è presente in tutti i sistemi linux / unix. Ad esempio con

```
cd ..
```

ci spostiamo nella cartella padre della nostra attuale working directory. Notiamo che dopo il comando `cd` vi è uno spazio, che delimita il comando dal suo argomento `..` ovvero la cartella di destinazione.
Un altro comando fondamentale è `ls`. Esso ci dice quali files giacciono in una specifica cartella. La sintassi è la seguente

```
ls [OPTIONS] /percorso_cartella
```

le opzioni di `ls` sono molteplici. Le più importanti sono

- `-l` la quale ci fornisce più informazioni per i files all'interno della cartella. E' così comune che ha un suo link, ovvero un comando dedicato `ll` = `ls -l`
- `-a` che ci fa vedere anche i files nascosti (i quali iniziano con un punto)
- `-t` che ordina i files per data di modifica.

3.1.1 Gestione dei Files

Come prima cosa diciamo che negli ambienti Linux l'estensione del file non importa. L'estensione viene utilizzata per facilitare la lettura da parte degli utenti. Qualche programma potrebbe cercare l'estensione dei file, ma sono casi rari. Ciò che importa al sistema è se un file sia di testo o meno. Esempi di file di testo sono

- File di configurazione
- Scripts e programmi (`.sh`, `.c`, `.cpp`, `.p`)
- File di informazione del sistema

L'altra grande categoria di files sono i files binari. Questi non possono essere cercati ne letti (ad esempio con il comando `cat` o `more`). Ci viene in aiuto il comando `file`, il quale ci indica il tipo di file.
Ad esempio, se abbiamo un file di testo che abbiamo denominato `test.jpeg` e digitiamo

```
file test.jpeg
```

l'output sarà

```
test.jpeg: ASCII text
```

ovvero il file `test.jpeg` è un file di testo codificato con codice `ASCII`. Per un'immagine vera, l'output sarà diverso, ad esempio:

```
JPEG image data, Exif standard: [TIFF image data, little-endian,

direntries=0], baseline, precision 8, 3840x2160, components 3
```

Molti dei comandi che manipolano i files, funzionano anche per le cartelle, delle volte basta aggiungere l'opzione `-r` che sta per `--recursive`. Nei casi più semplici, come ad esempio il comando `mv` che sposta/rinomina i files e le cartelle non è necessario aggiungere l'opzione `-r`.

```
mv <vecchio_nome> <nuovo_nome>
```

Un altro comando fondamentale è quello utilizzato per copiare i files

```
cp <vecchio_nome> <nuovo_nome>
```

Se vogliamo copiare un'intera cartella basta aggiungere l'opzione `-r`:

```
cp -r <vecchia_cartella> <nuova_cartella>
```

Per creare un file vuoto si utilizza il comando `touch`:

```
touch nome_file
```

questo viene utilizzato anche per accedere al file e modificare dunque la data di accesso ad un file.
D'altra parte esiste anche un comando utilizzato per rimuovere un file o una cartella.

```
rm <nome_file>
rm -r <nome_cartella>
```

E' possibile utilizzare anche l'opzione `-f` che sta per `--force`, che fa sì che la shell non ci chieda ogni volta se siamo sicuri di eliminare il suddetto file.
Tutti questi comandi funzionano con un solo file. Possiamo passare come argomenti più files, oppure

- Globbing `?` per il numero di caratteri che mancano. Ad esempio se abbiamo dei files nella nostra cartella

  ```
  file1.txt file2.txt file3.txt file4.txt file11.txt
  ```

  e digitiamo

  ```
  rm file?.txt
  ```

  esso rimuoverà tutti i files tranne `file11.txt`.

- Globbing `*`. Digitando invece

  ```
  rm file*.txt
  ```

  verranno rimossi tutti i files.

- Globbin `[]`. Digitando

  ```
  rm file[1-3].txt
  ```

  verranno rimossi tutti i files tranne `file4.txt` e `file11.txt`.

Se vogliamo cercare un file possiamo utilizzare il comando `find`. Ad esempio digitando

```
find . -name "file1.txt" -type f
```

cerchiamo un file nella cartella `.`, ovvero la nostra working directory denominato `file1.txt` di tipo `f` ovvero file. Se cerchiamo una directory l'opzione da aggiungere sarà `-type d`.
Possiamo scegliere tante opzione, le quali rendono questo comando molto potente. Possiamo ad esempio cercare dei file che sono stati modificato solo dopo un certo tempo, oppure eseguire un comando per ogni file che viene trovato (opzione `-exec`). Le wildcards, ovvero i caratteri `? * []` introducono dei problemi nella shell. Infatti, può capitare che passando un argomento come ad esempio `file*.txt` si raggiunga il limite massimo degli argomenti passabili al comando. Questo succede perché la shell prima crea una lista di argomenti con tutti i possibili nomi associati a `file*.txt` e poi li passa come argomento al comando. Per ovviare a questo problema, ad esempio nel comando `find` si può utilizzare l'identificatore di stringa (""):

```
find . -type f -name "file*.txt"
```

in questo modo non è la shell che crea la lista di argomenti, ma se ne occupa il comando `find`.

3.2 Canali I/O

La shell bash ha fondamentalmente tre modi per comunicare con l'utente:

- Standard input (`stdin`): cosa scriviamo nel prompt della shell
- Standard output (`stdout`): output normale della shell
- Standard error (`stderr`): output di errore della shell

E' molto comodo separare i canali di output. Un canale, o stream, è un deposito di memoria dal quale escono i dati di output (che andranno nei due canali di output `stdout` e `stderr`, e riceve dati dal canale di input `stdin`).

Lo standard input è un canale da cui giunge un flusso di dati (spesso testuali) in ingresso ad un programma. Il programma li trasferisce effettuando operazioni di lettura. Non tutti i programmi necessitano di dati in input: ad esempio i comandi ls o dir (che mostrano il contenuto delle directory) svolgono il loro compito senza bisogno di leggere dati in input. Il flusso

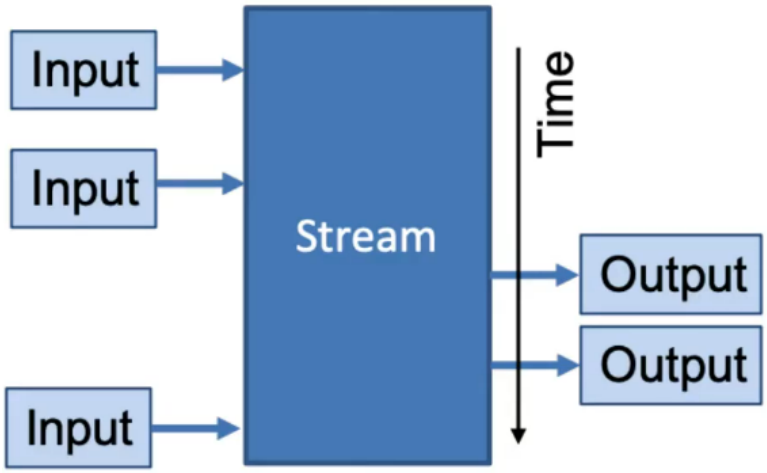

Figura 3.2: Schematizzazione semplificata di un canale.

di input, salvo casi di redirezione[1], proviene dal terminale (es. dall'utente tramite tastiera) da cui il programma è stato avviato.

Lo standard output è il canale su cui un programma scrive i suoi dati in output. Il programma trasferisce i dati effettuando operazioni di scrittura. Non tutti i programmi producono output: ad esempio il comando mv (che cambia il nome di un file o di una directory) normalmente non produce dati in output. Il flusso di output, salvo casi di redirezione, è diretto al terminale da cui il programma è stato avviato (es. a monitor o console a favore dell'utente).

Lo standard error è un altro canale di output, tipicamente usato dai programmi per i messaggi d'errore o di diagnostica. Esso è un canale indipendente dallo standard output, e se ne può effettuare la redirezione separatamente dagli altri. La sua destinazione è solitamente il terminale da cui il programma è stato avviato, in modo da rendere massime le possibilità di essere visto anche quando lo standard output è diretto altrove: ad esempio, nel caso di una pipeline software, l'output di un programma è fornito come input al programma successivo, ma i suoi messaggi d'errore sono ancora visualizzati sul terminale. È accettabile e normale che lo standard output e lo standard error abbiano la stessa destinazione, come nel caso del terminale testuale: i messaggi compaiono nello stesso ordine in cui il programma li

[1]Deviazione dei canali standard (standard input, standard output e standard error) di un dato comando verso destinazioni (o da sorgenti, nel caso dello standard input) che sono diverse da quelle predefinite.

scrive, salvo quando sono in uso dei buffer (ad esempio quando lo standard error non ha buffer e lo standard output ha un buffer di linea: in questo caso i dati scritti sullo standard error in un secondo momento possono apparire prima dei dati scritti su standard output in un primo momento, in quanto il buffer dello standard output potrebbe non essere ancora stato riempito).

Come vediamo in Figura 3.2 l'input e l'output possono talvolta sovrapporsi, e non ci sono regole che gestiscono l'ordine in cui avvengono l'input e l'output.

Possiamo inoltre redirezionare l'input e output ad altre sorgenti. Possiamo ad esempio redirezionare l'input di un comando e prenderlo dall'output di un altro comando, oppure possiamo redirezionare l'output in uno stream su file con la sintassi:

```
comando > file
```

Con il simbolo `>` possiamo infatti redirezionare lo `stdout` su un file denominato `file`. Per redirezionare lo `stdout` si usa la sintassi

```
comando 2> file
```

mentre se vogliamo dare in input ad un comando un file, e quindi redirezionare lo `stdin` si usa

```
comando < file
```

Possiamo poi, come già detto, utilizzare l'output di un comando (`comando2`) come input di un altro comando (`comando1`) utilizzando il simbolo pipe `|`:

```
comando1 | comando2
```

Potremmo aver bisogno di appendere output di più comandi sullo stesso file senza sovrascriverlo ogni volta. Per fare questo si utilizza il simbolo `>>`:

```
comando >> file
```

Nei sistemi Linux, i canali I/O standards sono numerati seguendo la convenzione del linguaggio C.

```
0: stdin
1: stdout
2: stderr
```

Potremmo voler redirezionare lo `stdout` di un programma in un file diverso dal canale `stderr`.

```
comando > output.log 2> error.log
```

Con questa sintassi si redireziona lo standard output nel file `output.log` mentre lo standard error in `error.log`. Possiamo inoltre redirezionare entrambi i canali sullo stesso file con la sintassi

```
comando 2>&1 > output_error.log
```

3.2.1 Visualizzazione e Ricerca

Vediamo ora alcuni comandi per la visualizzazione del contenuto di un file di testo sulla console bash. Per visualizzare il contenuto di un file esistono diversi modi. Il più semplice è l'utilizzo del comando `cat`. Con il comando `less` inoltre possiamo scorrere il testo (ad esempio il comando `man` usa questo comando per visualizzare il file di testo nel quale vengono spiegati i vari utilizzi dei programmi). Altri comandi, come ad esempio `head` o `tail` stampano in `stdout` le prime righe o le ultime di un file.
Il comando `grep` viene utilizzato per la ricerca di contenuti nei file di testo. La sintassi è

```
grep [OPZIONI] [stringa da cercare] [nome del file]
```

ad esempio

```
grep -i -r "test" example*.txt
```

le opzioni più comuni sono

```
-r: ricerca ricorsiva
-i: ignora maiuscole e minuscole
-I: ignora i file di tipo binario
```

Questo comando è molto utile quando ci ritroviamo con dei files di grandi dimensioni e vogliamo cercare una specifica stringa all'interno di esso. Come il comando `find`, a causa delle wildcards e la moltitudine di opzioni, è molto potente.
Possiamo utilizzare il comando `grep` combinandolo con il comando `ll` (ovvero `ls -l`) usufruendo del simbolo pipe `|` con questa sintassi:

```
ll | grep -i test
```

questo comando fa vedere tutti i files che hanno al loro interno una stringa `test`.

Capitolo 4

Utenti e Processi

Come ogni sistema operativo moderno, anche Linux è un sistema multiutente. Questo vuol dire che il login nel sistema operativo può essere effettuato da più utenti. Esistono fondamentalmente due tipologie di utente: standard o amministratore. Ogni utente, nei sistemi Linux possiede una cartella `home` dedicata.

4.1 Permessi

L'accesso ai files di sistema viene regolamentato attraverso dei `permessi` specifici per ogni utente. Solo l'utente `root` può fare qualsiasi cosa all'interno del sistema operativo. Ad esempio ad un utente standard potrebbe essere persino impedito di leggere in alcune cartelle. Questi permessi possono essere configurati solo dagli utenti amministratori.
Ogni file ed ogni cartella ha specifici permessi. Tali regole specificano cosa può fare o non fare con quel file o quella cartella[1].
Gli utenti standard, nel caso in cui conoscano la `password` dell'amministratore, possono ottenere i permessi di `root` e lanciare programmi o comandi con tali permessi digitando

```
sudo <comando>
```

Ogni utente appartiene a uno o più gruppi. Non appena viene creato, un utente appartiene ad un gruppo primario. Anche in base a quest'appartenenza si possono configurare i permessi per i files e cartelle. Esistono tre tipi di permessi:

[1] "Non puoi rompere ciò che non vedi".

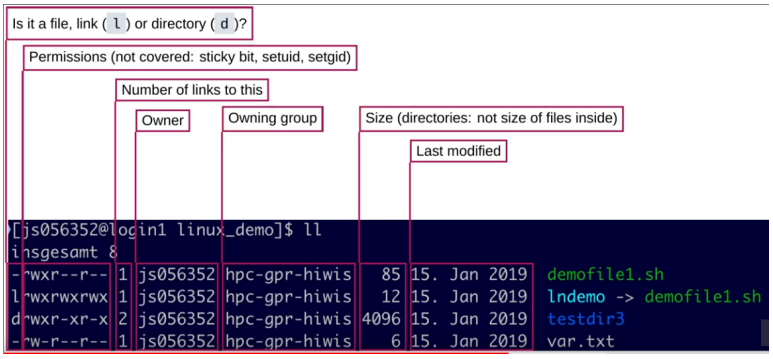

Figura 4.1: Output del comando ll.

- Lettura (`r`): chi può vedere il contenuto di files e cartelle
- Scrittura (`w`): chi può cambiare il contenuto di files e cartelle
- Esecuzione (`x`): chi può eseguire un file (ad esempio uno script bash `.sh`)

Questi tre tipi di permessi di files non sono indipendenti. Ovviamente per poter modificare un file deve essere necessario vederli. Quindi in molti casi, se non tutti, il permesso di scrittura include quello di lettura. E' possibile però poter eseguire uno script senza essere in grado di leggere il suo contenuto.
Per poter vedere chi ha il permesso su un file, chi lo ha creato, e quali permessi hanno i gruppi, etc. si utilizza il comando `ll`.

Vediamo più in dettaglio l'output introdotto in Figura4.1

```
drwxr-xr-x 2 user groups size data_ultima_modifica nome_file
-rw-r--r-- 1 user groups size data_ultima_modifica nome_file
```

la prima lettera (prima riga `d`, seconda riga `-`) indica il tipo di file.

```
d: directory
f: file
-: not set
l: link
```

poi abbiamo tre sequenze di `rwx`. La prima di queste ci indica i permessi che possiede l'utente `user` sul file o cartella. La seconda sequenza indica i permessi dei gruppi `groups` che possiedono il file. La terza sequenza indica

quali permessi degli utenti di altri gruppi che non possiedono il file. Nella seconda riga vediamo che alcuni permessi non ci sono. Quando un utente o un gruppo non ha un permesso, esso viene visualizzato con il simbolo `-`.

4.1.1 Cambiare i permessi

Esistono dei comandi volti a cambiare i permessi di files/cartelle, e la modifica del possessore di quest'ultimi. Con il comando `chown` si può cambiare il possessore di un file. Esso necessita di permetti di `root`:

```
sudo chown <nuovo_possessore> <nome_file>
sudo chown <nuovo_possessore>:<nuovo_gruppo> <nome_file>
```

Per cambiare i permessi di un file, possiamo utilizzare il comando `chmod` con la seguente sintassi di esempio:

```
chmod u+x <nome_file>
```

dobbiamo quindi specificare tre opzioni. La prima specifica per "chi" cambiare il permesso, ed abbiamo quattro possibilità

```
u: user
g: group
o: others
a: all
```

La seconda opzione specifica se aggiungere (`+`) o togliere (`-`) il permesso, mentre la terza deve specificare quale permesso `r`, `w`, oppure `x`.

4.2 Gestione dei Processi

In Linux, un file eseguibile memorizzato su disco costituisce un programma. Quando un programma viene lanciato e caricato in memoria viene chiamato processo. Con linguaggio tecnico possiamo definire il processo come l'istanza di un programma.
I processi hanno un ciclo vitale simile a quello degli organismi viventi: nasce (il programma viene lanciato), vive (il programma è in esecuzione) e muore (il programma viene terminato).
Esistono tre tipi di processi:

- Processi di sistema (`system`), anche detti demoni (`daemons`)
- Processi dell'utente (`user`)

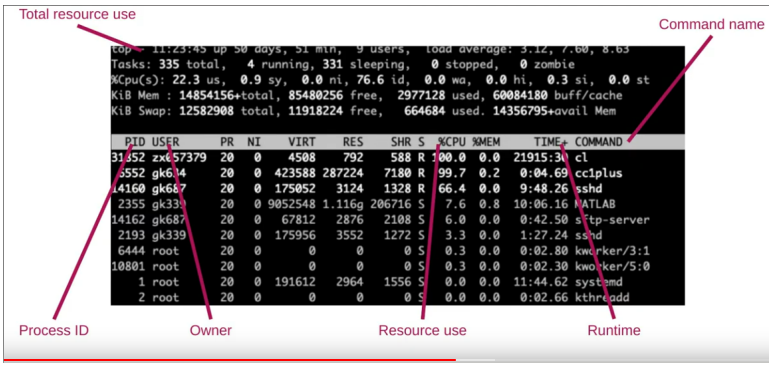

Figura 4.2: Tipico output del comando top.

- Processi dell'utente manuali (`user manually launched`)

I primi sono i processi necessari al sistema operativo per essere in esecuzione (ad esempio il processo che gestisce un driver). La seconda tipologia indica i processi dell'utente necessari per l'avvio di altri processi (ad esempio il gestore delle finestre, o window manager (`wm`)) Possiamo visualizzare sulla shell di bash i processi attualmente in esecuzione con il comando `top`. Altri comandi danno delle specifiche differenti sui processi, come ad esempio `pstree`, il quale ci indica l'albero delle dipendenze dei processi. Ad esempio, lanciando il browser web `firefox`, esso ha bisogno che sia attivo il processo che gestisce le finestre, in quanto `firefox` è un browser web grafico.

Come per i file e le cartelle, anche i processi hanno un possessore. Solo il possessore del processo può cambiarne lo stato. Ad esempio, sapendo che il gestore delle finestre è posseduto dall'utente `root`, solo chi conosce la password di `root` può lanciare o chiudere il gestore. In generale vale la regola che un processo può fare solo quello che il suo possessore può fare. Ad esempio un programma, che lanciato diventa un processo, non può scrivere in una cartella in cui il possessore del processo non ha il permesso di scrittura.
Quando viene avviato un processo gli viene assegnato un numero univoco chiamato ID processo (`PID`) che lo identifica in modo univoco nel sistema. Il `PID` non è modificabile e non varia per tutta la durata del processo. Solitamente il valore di `PID` viene assegnato in modo sequenziale: un nuovo processo, quindi, assumerà un valore di `PID` maggiore di uno rispetto all'ultimo processo creato.

Conoscendo il `PID` è possibile agire sul processo, ad esempio terminandolo. Il programma `top` ha diversi comandi

- `u`: filtra i processi in base all'utente che li possiede
- `k`: uccide uno specifico processo
- `h`: mostra i comandi (identico a `--help`)
- `f`: gestisce le colonne mostrate da `top`
- `x`: sottolinea e ordina secondo una colonna
- `<>`: sceglie la colonna secondo la quale ordinare i processi
- `R`: inverte l'ordine della colonna selezionata
- `q`: esce da `top`

Abbiamo già visto che un comando lanciato dalla shell diventa un processo. Lanciando solamente il programma, esso verrà eseguito nella shell, e terrà la shell occupata fino alla fine dell'esecuzione dello stesso. Digitando invece

```
<comando> &
```

esso verrà eseguito in background. Questa sintassi ci è utile se il programma viene eseguito su una finestra. In questo modo, infatti, la shell continuerà ad essere utilizzabile.
Possiamo sospendere un comando che è stato mandato nella shell utilizzando la combinazione da tastiera `Ctrl+z`. Un comando che è stato messo in pausa può essere mandato in background utilizzando il comando `bg`. Possiamo inoltre riportare nella shell un comando che è stato messo in background con il comando `fg`, con la sintassi

```
fg <job-ID>
```

dove il `job-ID` è diverso dal `PID`, e può essere visualizzato con il comando `jobs`.

Capitolo 5

Strumenti e Utilities

5.1 L'editor di testo VIM

L'editor di testo predefinito di Linux si chiama `vi`. Questo ha anche una versione migliorata: `vim` (`vi` improved). Esso è un editor da console, ovvero non viene lanciata una finestra grafica, ma il file viene editato direttamente all'interno della shell bash.
Tra i vantaggi troviamo

- Sempre disponibile, in tutti i sistemi Linux
- A patto di conoscerne i comandi, è molto efficiente e veloce.

Mentre abbiamo lo svantaggio di avere un interfaccia utente molto poco intuitiva, e la mole di comandi disponibili è eccessiva.

Da `vi` mantiene la caratteristica di essere modale, ovvero di avere modalità diverse nelle quali i normali caratteri della tastiera hanno significato di inserimento testo o di comandi. In questo modo, è possibile usarlo senza far uso del mouse, né dei tasti meta, permettendo una velocità maggiore di scrittura, a prezzo di maggiore difficoltà di utilizzo da parte di nuovi utenti. Vediamo qui di seguito alcuni dei comandi più comuni. Per aprire un file vuoto si utilizza il comando `vim`, mentre per aprire un file con uno specifico nome

```
vim <nome_file>
```

I comandi più comuni sono:

- Gestione file:

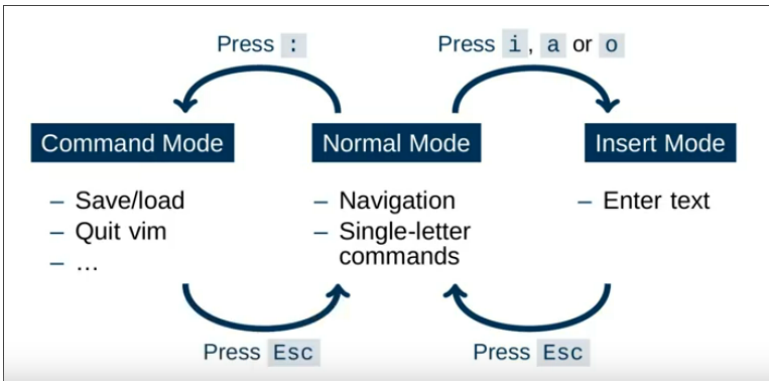

Figura 5.1: Schema di comandi per cambiare la modalità di vim.

  - `:w` per salvare il file (`write`)
  - `:w␣<nome_file>` per salvare il file con nome `<nome_file>`
  - `:wq`, `:x`, `ZZ` per salvare e chiudere
  - `:q!` per chiudere il file senza salvare.

- Muoversi all'interno del file:
  - `frecce` per muovere il cursore nella direzione della freccia
  - `h,j,k,l` per muovere il cursore a sinistra, giù, su, destra
  - `$` per muovere il cursore alla fine del file
  - `gg` per muovere il cursore alla prima riga
  - `G` per muovere il cursore all'ultima riga
  - `w` per saltare alla parola successiva
  - `b` per saltare alla parola precedente
  - `%` per saltare al carattere corrispondente, ad esempio `(...),[...],{...}`

- Modifica del testo:
  - `u`: annulla l'ultima modifica
  - `Ctrl␣+␣r`: rifai l'ultima modifica
  - `.`: ripeti l'ultimo comando
  - `x`: cancella carattere
  - `dd`: elimina l'intera riga

  - `yy` o `Y`: copia l'intera riga
  - `p`: incolla contenuto dopo il cursore

- Ricerca:
  - `/pattern`: vai avanti nella ricerca per espressioni regolari
  - `?pattern`: vai indietro nella ricerca espressioni regolari
  - `n`: ripeti l'ultima ricerca
  - `N`: ripeti l'ultima ricerca nella direzione opposta
  - `%s/old/new/`: sostituisce `old` con `new` nella riga corrente
  - `%s/old/new/g`: sostituisce `old` con `new` nell'intero file

5.2 Shell Scripts

Una volta che sappiamo come interagire con la la shell di Linux, sappiamo anche dare una serie di istruzioni, e salvare queste in un file eseguibile. Questi sono detti *scripts*. Questo file di testo viene dato in input nel terminale, ed esso eseguirà tutte le istruzioni contenute al suo interno. Il vantaggio di questo metodo di scripting è ovviamente l'automazione di alcune operazioni che viene fatta molto semplicemente scrivendo le istruzioni. Lo svantaggio, se così si può dire, è che non sappiamo della presenza di errori fino al momento in cui lo script non viene lanciato nel terminale.
Per eseguire uno script `.sh` basta scrivere nel terminale:

```
./script.sh
```

avendo cura che l'utente abbia il permesso di eseguire il file nella shell.
Come vediamo dalla sintassi, dobbiamo specificare in che cartella si trova il nostro script, infatti nel nostro caso abbiamo identificato la working directory con `./`. Questo perché Linux cerca i comandi nella cartella in cui siamo, al massimo nelle directories incluse nella variabile di ambiente della `$PATH`[1] per questioni di sicurezza. Vediamo un primo esempio di script bash.

Esempio 5.3. \\

```
1 #!/bin/bash
2
3 #Questo è un commento
4 echo "Hello World"
```

[1] Approfondiremo il concetto e gli utilizzi delle variabili d'ambiente successivamente.

```
5
6 ls -l
7 sleep 3s
8 ls \
9  -l
```

Nella riga `1` viene specificato l'interprete di `bash`. Questa direttiva deve trovarsi sempre nella prima riga di ogni script. Esso si specifica con il carattere utilizzato per i commenti, `#` seguito dal punto esclamativo `!`. Vediamo che alla riga `8` c'è il simbolo `\`. Esso specifica che il comando continua a capo.
Durante l'esecuzione degli script verrà stampato sulla console l'output di tutti i comandi. Utilizzando le redirezioni possiamo, ovviamente, stampare su file tali output.
Possiamo assegnare un valore ad una variabile tramite l'operazione binaria `=`.

```
var="valore"
```

dove notiamo che ai lati del simbolo `=` non vi sono spazi. I simboli `"..."` servono quando nel valore vi sono degli spazi o dei caratteri speciali.
Possiamo stampare il valore di `var` con

```
echo $var
```

Un errore comune che viene commesso durante l'assegnazione delle variabili è confondere il ruolo delle doppie virgolette e quello delle singole virgolette. Utilizzando le singole virgolette verrà assegnata alla variabile il contenuto esatto. Ad esempio scrivendo:

```
var='$ciao'
```

nella variabile `var` ci sarà la stringa `$ciao`, mentre usando

```
var1='ciao'
var2="$var1"
```

nella variabile `var2` ci sarà il valore effettivo di `var1`, che è in questo caso la stringa `ciao`. Se racchiudiamo tra parentesi un comando, esso verrà primo eseguito. Ad esempio digitando

```
var=$(bla)
```

verrà prima eseguito il comando `bla` e il suo output verrà memorizzato nella variabile `var`.
Possiamo passare degli argomenti quando eseguiamo lo script dal terminale. Ad esempio digitando:

```
./script.sh -f 5.0
```

nello script avremo le seguenti variabili assegnate

```
$0=script.sh
$1=-f
$2=5.0
```

Possiamo inoltre usufruire dei ciclo e degli statements `if`

Esempio 5.4.
```
#!/bin/bash
for file in $( ls ); do
echo item: $file
done

if[ -e $filename ]; then
echo "$filename exists."
fi
```

Capitolo 6

Variabili di Ambiente

Le variabili d'ambiente, nei sistemi Unix-like, e più in generale negli ambienti `POSIX` sono delle variabili specifiche per ogni processo. Ogni processo possiede il proprio insieme di variabili d'ambiente distinto e separato da quello degli altri processi, ed in nessun caso un processo può modificare le variabili d'ambiente di un altro processo; esse tuttavia si propagano, duplicandosi, di processo padre in processo figlio.
Le variabili d'ambiente hanno un ruolo tradizionalmente significativo nella configurazione dei programmi: esse sono usate ad esempio per specificare le impostazioni di localizzazione, o per indicare ad un programma le directory ove reperire le proprie risorse, o ancora per modificarne il comportamento predefinito.

6.1 Caratteristiche

Possiamo assegnare una variabile di ambiente ad un processo[1] o direttamente alla shell. Ad esempio, per la console del terminale la variabile di ambiente `HOME` si può stampare su schermo con il comando:

```
echo $HOME
```

che darà come risultato

```
/home/nome_utente
```

[1] Ogni processo ha la sua lista di variabili che possiamo assegnare.

Le variabili di ambiente sono molto utili per la creazione di script, in quanto possiamo modificare le cose localmente. Ad esempio è possibile impostare percorsi differenti per librerie diverse. Supponiamo di voler lanciare un codice con alcune librerie su due diverse architetture di processori. Grazie alle variabili di ambiente possiamo configurare le librerie che il nostro codice utilizza prima della compilazione, in questo modo se andiamo a compilare il nostro codice su un'architettura `arm64` o `amd64`, non dobbiamo cambiare il codice, ma solamente il percorso delle librerie che saranno differenti per ogni architettura.

Molte variabili di ambiente vengono inizializzate quando il sistema operativo viene installato, ad esempio la variabile `HOME` che abbiamo visto prima, oppure la variabile `USER`, mentre alcune variabili di ambiente vengono assegnate quando installiamo del software, ad esempio delle librerie grafiche, o il compilatore `gcc/g++`.
Per visualizzare tutte le variabili di ambiente impostate basta utilizzare il comando `env`. Per crearne una nuova, si utilizza la convenzione di definirle in maiuscolo, ad esempio:

```
export MY_VAR="value"
```

D'ora in poi la variabile `MY_VAR` sarà disponibile nei processi figlio.

6.1.1 Vincoli

Le variabili d'ambiente sono implementate come un array di stringhe[2] secondo le convenzioni del linguaggio C[3], nella forma `"nome=valore"`: questo implica sia che i nomi non possano contenere il carattere `"="`, sia che né il nome né il valore possano contenere il carattere `ASCII␣0x00,␣NUL`. Due nomi sono considerati uguali solo se contengono esattamente gli stessi caratteri, per cui viene ad esempio fatta distinzione tra lettere maiuscole e minuscole.

A parte questi vincoli, la scelta di nomi per variabili d'ambiente è anche fortemente influenzata dai vincoli di sintassi delle shell testuali, che di fatto restringono la scelta a nomi composti da una lettera o un trattino basso (`"_"`) seguiti da zero o più lettere, cifre e trattini bassi, ad esempio "A", `"_Alice_12",␣"file_temporaneo"`. Possono esistere variabili che non seguono queste convenzioni nei nomi, tuttavia ciò ne rende alquanto difficile l'utilizzo pratico.

[2] Ovvero come delle frasi.
[3] Documentazione linguaggio C.

Non vi sono limiti sulle dimensioni massime ed il numero delle variabili d'ambiente, se non quelli dettati dalla disponibilità di memoria; tuttavia le chiamate di sistema della famiglia `exec`, che sostituiscono il processo invocante con un altro programma, possono fallire se lo spazio totale occupato delle variabili d'ambiente di un processo, unitamente allo spazio occupato dei parametri per il nuovo programma, supera la dimensione in byte definita della costante `ARG_MAX` definita nello header file `limits.h`. Il valore minimo di tale costante per lo standard `POSIX` è di `4096␣byte`, definito dalla costante `_POSIX_ARG_MAX`.

6.1.2 Moduli di Ambiente

In un contesto cluster, è molto probabile, data la mole di utenti con differenti esigenze, che siano necessarie diverse versioni dello stesso software. Un esempio è il modulo `OpenMPI`. Per adattarsi alle differenti esigenze, potrebbe essere necessario utilizzare una o l'altra versione dello stesso compilatore. Per risolvere il problema, si possono costruire più ambienti, dove le variabili assumono valori diversi. Questo viene fatto perché in linea di principio un amministratore del sistema non sa cosa serve all'utente, e quindi è meglio dare la possibilità di caricare diverse versioni dello stesso modulo. Ad esempio:

```
module load openmpi/gcc/64/1.10.3
```

carica il modulo `OpenMPI` compilato con `gcc` per sistemi `x64` nella versione `1.10.3`.
Ad ogni modulo viene generalmente associato un file di definizione, che può essere uno script `LUA`[4] o `Tcl`[5].

6.2 Esempi di utilizzo nella shell testuale.

Nelle shell testuali è possibile usare il comando env sia per visualizzare le impostazioni correnti, sia per avviare nuovi programmi con valori particolari delle variabili d'ambiente.

In aggiunta a questo, per le shell derivate dalla Bourne shell come Bash o la Korn shell, il meccanismo delle variabili della shell si sovrappone in parte a quello delle variabili d'ambiente, per cui le variabili d'ambiente sono automaticamente anche variabili della shell, e le variabili della shell possono

[4] Documentazione LUA
[5] Documentazione Tcl

a loro volta divenire variabili d'ambiente (ad esempio tramite il comando "`export␣nome_variabile`"). Ad esempio:

Esempio 6.3. *Script* `bash` *di esempio per manipolare le variabili di ambiente.*

```
#!/bin/bash

# PATH è una variabile di ambiente già avvalorata
# e la si può usare come qualsiasi altra variabile
# di shell
echo "$PATH"

# MY_VAR è una variabile di shell che non esisteva in precedenza.
# I comandi avviati non ne hanno visibilità, se non passandola
# esplicitamente come parametro.
MY_VAR=123
echo "$MY_VAR"

# Cerchiamo MY_VAR tra le variabili di
# ambiente (non la troveremo).
env | grep "^MY_VAR="

# Con il comando export è possibile rendere MY_VAR anche una variabile di
# ambiente, automaticamente visibile ai comandi avviati in seguito.
export MY_VAR

# Cerchiamo nuovamente MY_VAR tra le variabili di
# ambiente (ora invece la troveremo).
env | grep "^MY_VAR="

# Le shell più comuni (Bash e Korn shell) permettono
# di combinare il comando export con una assegnazione,
# ad esempio con
export MY_VAR="nuovo valore"

# Attenzione: a differenza di MS-DOS, impostare una variabile di
# ambiente ad un valore vuoto non la cancella. Essa continua
# ad esistere, solo che ha un valore vuoto.
export MY_VAR=
env | grep "^MY_VAR="
```

```
# Per eliminare del tutto una variabile di
# ambiente occorre usare il comando "unset"
unset MY_VAR
```

6.4 Variabili di ambiente notevoli

6.4.1 EDITOR e VISUAL

Queste variabili suggeriscono il comando da usare per avviare un editor di testo per modificare un file di testo.
Valori tipici possono essere "vi", "emacsclient" o anche "gedit"

6.4.2 ENV

Questa variabile indica il nome di uno script di shell da eseguire ad ogni avvio in modalità interattiva della shell, in aggiunta ad altri script eseguiti automaticamente all'avvio in seguito alla procedura di login.
Valori tipici possono essere ad esempio `"~/.bashrc"` o `"~/.kshrc"`.

6.4.3 HOME

Normalmente viene avvalorata automaticamente dal sistema con il pathname della home directory dell'utente corrente.

6.4.4 http proxy e ftp proxy

Queste variabili indicano quale proxy usare per connessioni HTTP e FTP. Ad esempio, in uno script di shell:

```
# Proxy HTTP che richiede autenticazione
http_proxy="http://utente:password@proxy:8080"
export http_proxy

# Proxy HTTP che non richiede autenticazione
http_proxy="http://proxy:8080"
export http_proxy
```

6.4.5 LANG

Questa variabile specifica le impostazioni locali, come lingua da usare per l'interfaccia utente, convenzioni su formato di data e ora, rappresentazione dei numeri, codifica dei carattere in uso e altro ancora. Viene usata per specificare le impostazioni predefinite di sistema.

I valori possibili per `LANG` sono quelli elencati dal comando "locale -a", ad esempio `"it_IT.UTF-8"`, o `"C"`.

`LANG` è solo una delle variabili d'ambiente che controllano la localizzazione: ne esistono altre che trattano aspetti specifici (ad esempio `LC_MONETARY`, `LC_MESSAGES`, `LC_NUMERIC`) ed hanno precedenza su `LANG`. In particolare esiste anche `LC_ALL` che ha precedenza su tutte le altre (anche su `LANG`) e che può essere usata per specificare rapidamente l'uso di una localizzazione diversa da quella predefinita.

6.4.6 LD LIBRARY PATH

Nei sistemi che adottano `ELF`[6] come formato dei file eseguibili, `LD_LIBRARY_PATH` indica al `linker` dinamico una serie di directory separate da due punti (`":"`) in cui cercare librerie software in aggiunta a quelle predefinite nel sistema.

6.4.7 LD PRELOAD

Nei sistemi che adottano ELF come formato dei file eseguibili, `LD_PRELOAD` indica al `linker` dinamico una o più librerie software da precaricare, in aggiunta a quelle richieste dal programma.
Questo meccanismo è utile per usare librerie che ridefiniscono l'implementazione di funzioni standard, e che ad esempio possono permettere di accedere in maniera trasparente a file compressi o tenere traccia di tutte le allocazioni di memoria effettuate.

6.4.8 MANPATH

Specifica una serie di directory separate da due punti (":") in cui cercare delle pagine man, in aggiunta a quelle predefinite nel sistema.

[6] ELF(Executable and Linkable Format) è un formato file standard per eseguibili, codice oggetto, librerie condivise e core dump.

6.4.9 PATH

Specifica una serie di directory separate da due punti (":") in cui il sistema ricerca file eseguibili quando essi non sono qualificati con alcuna directory.

Un valore tipico è "/bin:/usr/bin:/usr/local/bin", ma varia da sistema a sistema.

È da sottolineare che per motivi di sicurezza, nei sistemi Unix e Unix-like la directory corrente (rappresentata con un punto ".") non viene automaticamente inclusa tra quelle in cui effettuare la ricerca: se si vuole includerla, occorre specificarla esplicitamente nel valore di PATH.

6.4.10 POSIXLY CORRECT

Se è definita (il valore non è rilevante), i programmi del progetto GNU si comportano secondo quanto previsto dallo standard POSIX anche nei casi dove normalmente essi divergono dallo standard.
Ad esempio, lo standard POSIX prevede che i comandi df e du utilizzino come unità di misura dei blocchi da 512␣byte, mentre le versioni GNU normalmente utilizzano come unità di misura dei blocchi da un KiB. Impostando la variabile d'ambiente POSIXLY_CORRECT ad un qualunque valore anch'essi useranno blocchi da 512␣byte.

6.4.11 TMPDIR

Specifica il pathname di una directory da usare per file temporanei. Normalmente non è impostata, in quanto le directory per i file temporanei sono già stabilite per convenzione (ovvero /tmp e /var/tmp), tuttavia è buona norma per i programmi che creano file temporanei di grosse dimensioni prendere in considerazione anche il valore di questa variabile prima di usare direttamente le directory standard.

6.4.12 TERM

Indica il tipo di terminale o terminale virtuale in uso, in modo che i programmi che offrono un'interfaccia a caratteri possano visualizzarla correttamente. Il valore è un identificativo in una base dati fornita dal sistema operativo (termcap o terminfo) che descrive le caratteristiche del terminale e le sequenze di caratteri necessarie a compiere azioni quali ad esempio lo spostamento del cursore o la cancellazione di una linea.
Valori tipici sono "xterm" e "vt100"

6.4.13 TZ

Specifica la regola da usare per convertire dalla data di sistema (mantenuta internamente in tempo coordinato universale) nella data e ora locali all'utente, e viceversa.

Capitolo 7

Informazioni e Configurazione di Sistema