Sistemi Operativi 2

INDICE

LINUX

Breve storia di Linux

Il sistema operativo **Multics (Multiplexed Information and Computing Service)** fu uno dei primi sistemi operativi a condivisione di tempo (time-sharing), ossia multiprocesso e multi-utente, sviluppato attivamente a partire dal 1964 da parte dei centri di ricerca delle compagnie Bell Labs (AT&T Corp., una compagnia telefonica americana) e General Electric, assieme all'università MIT.
Multics mise sul campo tutta una serie di concetti e tecniche costruttive che sono ancora oggi elementi essenziali dei moderni sistemi operativi. Sebben rivoluzionario, il progetto Multics fu presto abbandonato da Bell Labs, poiché ritenuto troppo complesso da gestire.
A seguito di ciò, Ken Thompson e Dennis Ritche, due ricercatori di Bell Labs, svilupparono tramite un microcomputer PDP-7 la prima versione di Unics (Uniplexed Information and Computing Service), scritta totalmente in Assembly. Successivamente, sotto proposta di Brian Kernighan, il nome di Unics venne cambiato definitivamente in Unix.
Il sistema operativo Unix si diffuse rapidamente nei successivi 3-5 anni, portandolo allo sviluppo di versioni scritte tramite il linguaggio B e successivamente (e definitivamente) tramite il linguaggio C. Le versioni scritte in tali linguaggio permisero di portare Unix su varie architetture.
Il linguaggio C fu sviluppato da Dennis Ritchie stesso al fine di migliorare il linguaggio B precedentemente sviluppato dal suo collega Ken Thompson. La miglioria principale rispetto al B consiste nell'aggiunta dei tipi di dato (int, float, char, ...) rispetto alle sole generiche word da 4 byte del linguaggio B.
Successivamente, il codice sorgente del sistema operativo Unix venne distribuito ad università e centri di ricerca interamente assieme al proprio codice sorgente, il quale venne anche venduto ad aziende private, portando alla nascita di molte versioni.
Negli anni 80', Richard Stallman sviluppò il sistema operativo GNU (GNU is Not Unix acronimo ricorsivo), basato su Unix ma diverso da esso in quanto non contenente codice del sistema operativo Unix, e inventò GPL (GNU General Public Licence), una licenza pubblica utilizzata per il software libero. Unix venne inoltre riscritto completamente, aggiungendo pacchetti importanti ad esso, molti presi direttamente da GNU (es: gcc, make, ...).

Negli anni 90’, Linus Torvalds sviluppa il kernel Linux, il quale verrà poi utilizzato da altri sistemi operativi basati su Unix o derivati da esso. In particolare, nel 1994 viene definito lo standard Unix, dove un sistema operativo può avere marchio UNIX solo se esso rispetta le SUS (Single Unix Specification) e paga le royalties per l’uso del marchio.

Le caratteristiche dei sistemi Unix

Le caratteristiche di un moderno sistema operativo Unix, indipendentemente dalla sua categoria, sono:

- Multi-utente e multi-processo
- File system gerarchico
- Kernel in grado di gestire la memoria principale, la memoria secondaria, i processi, le operazioni I/O e le risorse hardware in generale
- System call utilizzabili tramite funzioni C che possono essere chiamate per interfacciarsi con il kernel
- Possiedono una shell di sistema, ossia un programma che "esegue programmi" interpretando i comandi dell’utente
- Modularità, programmi di utilità e supporto ad ambienti di programmazione
- Composto da una serie di piccoli programmi che eseguono un compito specifico, limitato, ma in maniera esatta e semplice
- I programmi sono silenziosi, il loro output è minimale e ridotto a ciò che è stato esplicitamente richiesto
- Ogni lavoro complesso può essere svolto come articolazione del lavoro svolto da programmi semplici
- I programmi manipolano solo testo e mai i file binari (es: altri programmi)

Utilizzo della Shell di sistema

Shell di Sistema

Informalmente, una **shell di sistema** (spesso detta terminale) è un programma che "esegue programmi".
Più formalmente, invece, la shell è un programma interattivo e/o **batch** (ossia "a lotti") che accetta comandi da far eseguire al **kernel.**
Tali comandi non sono necessariamente dei programmi, bensì possono essere anche dei comandi definiti all'interno della shell stessa.

Prima di eseguire un comando, la shell stampa a video un prompt, ossia una stringa nel formato

```
[nome_utente@nome_macchina cwd]$
```

Ogni comando segue la seguente struttura:

```
nome_comando [argomenti_opzionali] argomenti_obbligatori
```

Ad esempio, nel comando `cp -r -i -a -u file_sorgente file_destinazione`, gli argomenti `-r, -i, -a, -u` sono opzionali, mentre i rimanenti sono obbligatori.

Tipicamente, gli argomenti opzionali possono essere utilizzati anche con sintassi alternative (es: per il comando cp gli argomenti –interactive, –recursive sono uguali agli argomenti
-i, -r).

Inoltre, eventualmente essi possono avere un valore aggiuntivo in input (es: l'argomento –key=1 assegna il valore 1 all'argomento –key) e possono essere raggruppati (es: cp -ri è equivalente a cp r -i).

Tutti i comandi lanciati nella shell vengono salvati in una **cronologia**.

Super utente (root)

Ogni sistema operativo Linux-based possiede un **super utente** detto **root**, il quale possiede tutti i privilegi di sistema.
Pertanto, tale utente possiede accesso ad ogni operazione o comando possibile all'interno del sistema stesso.
L'utente root possiede sempre
UID pari a 0.

File System

Col termine **file system** si intende una struttura dati atta all'organizzazione di un'area di memoria di massa basata sul concetto di **file** e di **directory**, dove quest'ultime
possono contenere al loro interno dei file ed altre directory, creando così una struttura
gerarchica ad albero, dove solo le directory possono avere figli e i file corrispondono alle foglie dell'albero.

Nei sistemi operativi Linux-based ogni cosa può essere rappresentata come un file o un processo. Per tanto, all'interno del file system è necessario distinguere tra **file regolari**, i quali contengono sequenze di bit dell'area di memoria sulla quale è installato il file system, e **file non regolari**, ad esempio utilizzati per l'accesso di basso livello a periferiche o dispositivi vari.
Inoltre, all'interno di una directory valgono le seguenti regole:

- Non possono esistere due file o due sotto-directory con lo stesso nome
- Non possono esistere un file ed una sotto-directory con lo stesso nome
- I nomi dei file e delle sotto-directory sono **case sensitive** (es: ciao.txt è diverso da Ciao.txt)

Nei sistemi operativi Unix-based e Linux-based vi è un solo file system principale avente una directory radice (root directory), ossia la directory **/**. Essendo la radice del file system, ogni altro file o directory è contenuto direttamente o indirettamente all'interno della root directory.

I Percorsi

Pertanto, ogni file o directory è raggiungibile dalla root directory mediante un percorso (path), il quale può essere di due tipologie:

- **Percorso assoluto**, ossia una sequenza di directory separate da uno / che specifica la posizione di un file o una directory a partire dalla root directory

`(es: /home/utente/dir/subdir/file.pdf)`

- **Percorso relativo**, ossia una sequenza di directory separate da uno / che specifica la posizione di un file o una directory a partire dalla current working directory (cwd), ossia la cartella attualmente "aperta"
  (es: se la cwd è
  `/home/utente` allora il percorso relativo `dir/subdir/file.pdf` è equivalente al percorso assoluto `/home/utente/dir/subdir/file.pdf`)

Ogni **percorso relativo** risulta essere valido solo se la **cwd attuale è corretta**, mentre ogni **percorso assoluto** risulta essere **valido indipendentemente dalla cwd.**

All'interno dei percorsi (sia assoluti che relativi) è possibile utilizzare due directory speciali presenti all'interno di ogni directory:

- La **current directory** (ossia .), corrispondente alla directory stessa in cui ci si trova
  (es: il percorso
  `~/dir/./ciao.txt` è equivalente a `~/dir/ciao.txt`)
- La **parent directory** (ossia ..), corrispondente alla directory direttamente superiore a quella in cui ci si trova
  (es: se la cwd attuale è
  `~/dir/subdir1/`, il percorso `../subdir2` è equivalente al percorso `~/dir/subdir2`)

Visualizzare i File

Per sapere la cwd attuale, è possibile utilizzare il comando `pwd`, mentre per cambiare cwd è possibile utilizzare il comando `cd [path]` (se l'argomento path viene omesso, la
cwd verrà impostata sulla home dell'utente, ossia ~/).
Per
visualizzare il contenuto di una directory, invece, è possibile utilizzare il comando `ls [path]`, restituente una lista dei file e le sotto-directory contenute in una directory (se l'argomento path viene omesso, viene utilizzata la **cwd come path**).

Se si vuole visualizzare **ricorsivamente** il contenuto della directory (dunque eseguendo ricorsivamente ls sulle sotto-directory), è possibile utilizzare l'argomento opzionale `-r`.

In una directory alcuni **file possono essere nascosti**, tipicamente file di configurazione o file usati come supporto a comandi ed applicazioni (es: il file .bash_history contiene la cronologia dei comandi eseguiti), i quali tuttavia non sono realmente invisibili, bensì essi vengono solamente omessi nell'output del comando ls, a meno che non venga utilizzato il parametro opzionale -a (dunque il comando `ls -a`). Un file può essere reso nascosto semplicemente aggiungendo un punto all'inizio del nome.

Creare Directory o File

Per **creare una directory**, è possibile utilizzare il comando `mkdir nome_dir.`

Utilizzando l'opzione -p, verranno create a catena tutte le sotto-directory del percorso indicato.
(es:
`mkdir -p dir1/dir2/dir3` crea anche le directory dir1 e dir1/dir2)
Per creare un
file vuoto, invece, è possibile utilizzare il comando `touch nome_file`, il quale potrà poi essere modificato utilizzando un editor di testo (come nano, vim, ...).
Per copiare un file, è possibile utilizzare il comando
`cp [-r] [-i] [-u] src_file dest_file`, dove:

- L'opzione **[-r]** permette di effettuare una copia ricorsiva sulle directory
- L'opzione **[-i]** avvisa l'utente nel caso in cui il file di destinazione esista gia
- L'opzione **[-u]** effettua la sovrascrittura di un file già esistente solo se l'mtime del file sorgente è più recente di quello di destinazione

Spostare o Eliminare File

Per spostare (o rinominare) un file è possibile utilizzare il comando

`mv [-i] [-u] [-f] src_file dest_file` dove:

- Le opzioni **[-i]** e **[-u]** sono identiche a quelle del comando `cp`

- L’opzione **[-f]** forza l’operazione

Per eliminare un file è possibile utilizzare il comando `rm [-r] [-i] [-f] src_file dest_file`, dove:

- Le opzioni **[-r]** e **[-i]** sono identiche a quelle del comando cp
- L’opzione **[-f]** forza l’operazione è identica a quella del comando mv

> 💡 A differenza del sistema operativo Windows, nei sistemi Linux-based non esiste il "cestino".
> Per tanto, eseguendo il comando rm il file verrà completamente eliminato dal disco.

Per convertire o copiare file in modo avanzato, è possibile utilizzare il comando `dd [opt]` dove l’argomento `[opt]` è una sequenza di entrate nel formato variabile=valore.
Le variabili principali utilizzabili sono:

- **if**, ossia il file di input (se non specificato, l’input viene letto da tastiera)
- **of**, ossia il file di output (se non specificato, l’output viene scritto sul terminale)
- **bs**, ossia la dimensione di un singolo blocco in lettura/scrittura
- **count**, ossia il numero di blocchi da copiare
- **skip**, ossia il numero di blocchi da saltare nell’input prima di leggere effettivamente
- **seek**, ossia il numero di blocchi da saltare nell’output prima di scrivere effettivamente

Mounting, Partizioni e Tipi di file system

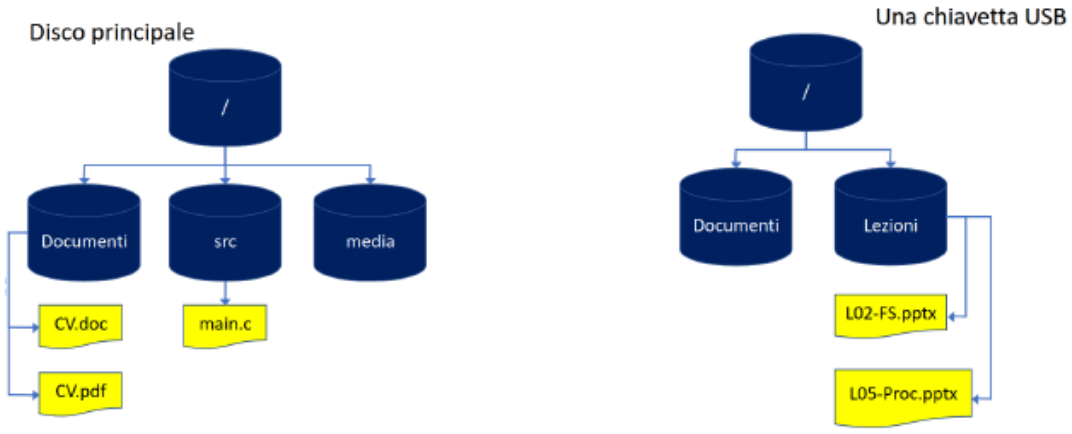
Il file system principale può contenere al suo interno **elementi** eterogenei tra loro, ad esempio:

- Dischi interni solidi o magnetici, solitamente contenenti il file system root
- File system su disco esterno
- File system di rete
- File system virtuali
- File system in memoria principale

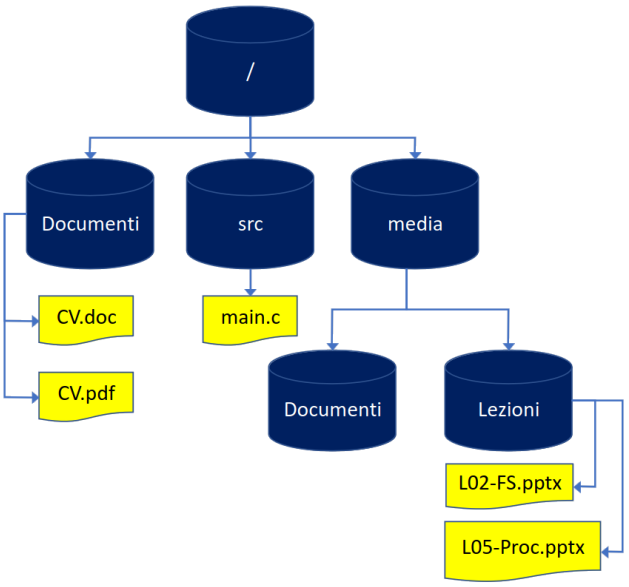
Una qualsiasi directory D all'interno del file system root può diventare il punto di **mount** per un altro file system F se e solo se la directory di root RF del file system F diventa accessibile da F.

Esempio:

- Consideriamo il seguente file system root e il seguente file system presente su una chiavetta USB



- Effettuando il **mounting** del secondo file system sulla directory `/mount` del file system root, esso diventa accessibile tramite la directory stessa



Effettuare il **mounting** di un file system F su una directory D non sovrascrive il contenuto della directory, bensì esso viene solamente temporaneamente sostituito.
Per tanto, se la directory D non è vuota, il suo contenuto originale sarà nuovamente accessibile dopo
l'**unmount** di F

Per montare un file system e visualizzare i file system montati è possibile utilizzare il comando `mount` (consultare il manuale per le varie opzioni).
Inoltre, per visualizzare i file system attualmente montati è possibile esaminare anche il contenuto del file
`/etc/mtab`, mentre per visualizzare i file system montati all'avvio (bootstrap) è possibile esaminare il contenuto del file `/etc/fstab`.

Partizione

Un disco solido o magnetico può essere **suddiviso** in due o più partizioni, le quali possono essere gestite indipendentemente, come se fossero in realtà due dischi separati.

Tipi di file system

I tipi di file system Linux si differenziano in:

Nome	Dim_{max} Part.	Dim_{max} File	Lung_{max} Nome file	Journal
ext2	32 TB	2 TB	255 B	No
ext2	32 TB	2 TB	255 B	Si
ext4	1000 TB	16 TB	255 B	Sì
ReiserFS	16 TB	8 TB	4032 B	Sì

Dove un **journaling file system** tratta ogni scrittura su disco come transazione, tenendo traccia delle operazioni svolte su un file di log. Inoltre, ogni file system differisce per il modo in cui vengono codificati i dati al suo interno.

Per **formattare** un **disco o una partizione**, ossia creare su di essi un nuovo file system da zero, può essere utilizzato il comando `mkfs [-t type] device` (consultare il manuale).
Per
visualizzare la **dimensione e l'occupazione di un file system**, è possibile utilizzare il comando `df [-h] [-l] [-i] [file]` (consultare il manuale).

Directory di primo livello

Tipicamente, la directory **/**, contiene al suo interno le seguenti **directory di primo livello**, le quali possono essere montate o non:

- **/boot**, contenente il kernel e i file per il bootstrap. Non viene montata ed è per tanto salvata direttamente all'interno della root directory.
- **/bin**, contenente i file binari (ossia i file eseguibili) di base. Non viene montata ed è per tanto salvata direttamente all'interno della root directory.
- **/sbin**, contenente i file binari (ossia i file eseguibili) di sistema. Non viene montata ed è per tanto salvata direttamente all'interno della root directory.
- **/dev**, contenente i file non regolari relativi all'uso delle periferiche hardware e virtuali. Viene montata in fase di bootstrap.
- **/proc**, contenente i file relativi a dati e statistiche dei processi e ai parametri del kernel. Viene montata in fase di bootstrap.
- **/sys**, contenente i file relativi ad informazioni e statistiche dei dispositivi di sistema. Viene montata in fase di bootstrap.
- **/media** e **/mnt**, utilizzate come punto di mount per i dispositivi I/O (es: CD, DVD, USB, ...). Vengono montate solo quando necessario.

- **/etc**, contenente i file di configurazione di sistema. Non viene montata ed è per tanto salvata direttamente all'interno della root directory.
- **/var**, contenente i file variabili. Non viene montata ed è per tanto salvata direttamente all'interno della root directory.
- **/tmp**, contenente i file temporanei. Non viene montata ed è per tanto salvata direttamente all'interno della root directory.
- **/lib**, contenente le librerie necessarie ai file binari. Non viene montata ed è per tanto salvata direttamente all'interno della root directory.

In particolare, all'interno della directory **/etc** possiamo trovare due file di fondamentale importanza all'interno del sistema operativo:

- Il file `/etc/passwd`, contenente una lista di tutti gli utenti del sistema e informazioni ad essi associate. Ogni riga della lista possiede la seguente struttura:

  ```
  username:password:uid:gid:gecos:homedir:shell
  ```

- Il file `/etc/group`, contenente una lista di tutti i gruppi del sistema e informazioni ad essi associate. Ogni riga della lista possiede la seguente struttura:

  ```
  groupname:password:gid:utente1,utente2,...
  ```

  Anche in tal caso, il campo password viene censurato da una x.

Index Node (inode)

> 💡 All'interno di un file system Linux-based, ogni file (regolare e non, directory e non) è rappresentato da una struttura dati detta **index node (inode)**.

Tra i principali attributi di un **inode** troviamo:

- **inode number**, univoco per ogni inode

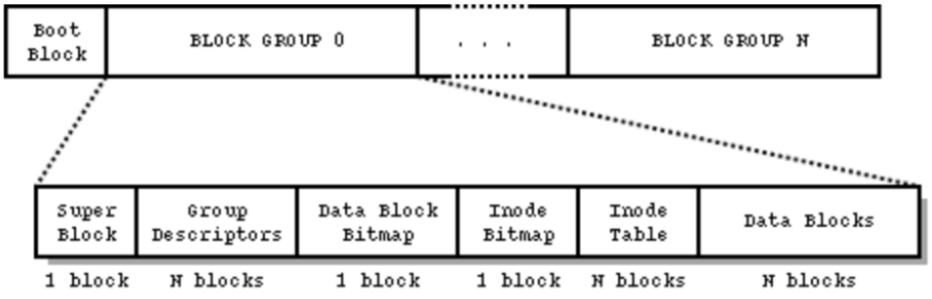
- **Type**, indicante il tipo di file
- **User ID (UID)**, ossia l'ID dell'utente proprietario del file
- **Group ID (GID)**, ossia l'ID del gruppo a cui è associato il file
- **Mode**, ossia i permessi di accesso al file per il proprietario, il gruppo associato ed ogni altro utente (vedi sezioni successive)
- **Size**, ossia la dimensione in byte del file
- **Timestamps**, ossia tre istanti di tempo:
  - **ctime** (change time), l'istante dell'ultima modifica di un attributo dell'inode
  - **mtime** (modification time), l'istante dell'ultima scrittura sul file associato
  - **atime** (access time), l'istante dell'ultima lettura del file associato
- **Link count**, ossia il numero di hard links dell'inode (vedi sezioni successive)
- **Data pointers**, ossia il puntatore alla lista dei blocchi su disco che compongono il file.

La vera funzionalità del comando `touch` risulta essere quella di **aggiornare tutti i timestamp** di un file **all'istante corrente**.

Se il file indicato non esiste, esso verrà **creato**. La creazione del file, dunque, è solo un **effetto secondario**.

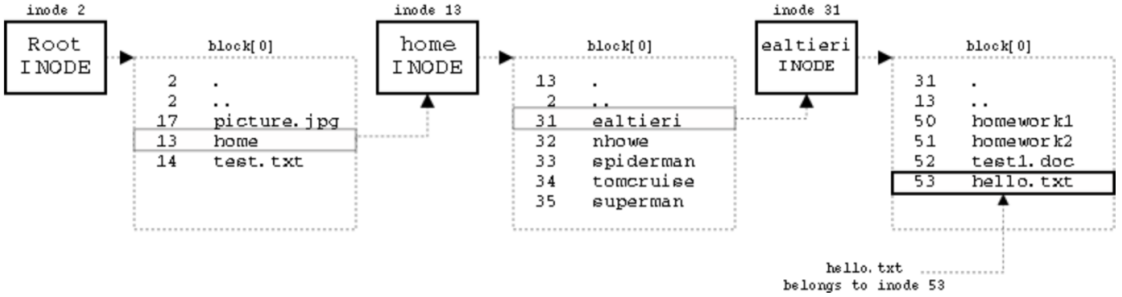
All'interno del file system (in particolare all'inizio del disco o partizione su cui è installato il file system) si trova una **tabella degli inode**.

Ad esempio, nei file system ext2 la tabella degli inode viene conservata all'interno del primo gruppo di blocchi:



Una **directory** non è altro che un **file speciale**; il cui contenuto è costituito da blocchi su disco contenenti tabelle formate da tuple nel formato (inode_file, nome_file).

Ad esempio, il path `/home/documenti/hello.txt` viene seguito esaminando uno ad uno il contenuto dei file puntati dagli inode delle directory intermedie:



Per visualizzare le informazioni contenute dell'inode di un file, il comando ls fornisce **numerevoli** opzioni:

- L'opzione **[-l]** permette di visualizzare permessi di accesso, numero di sottodirectory UID, GID, size, mtime dei file.
  Inoltre, all'inizio dell'output viene visualizzato il numero di blocchi totali occupati dalla directory (non è incluso il numero di blocchi occupati dalle sottodirectory), dove, normalmente, un blocco è grande tra 1kB e 4kB.
  - Se usato assieme a **[-c]**, viene visualizzato il **ctime** al posto dell'mtime
  - Se usato assieme a **[-u]**, viene visualizzato **l'atime** al posto dell'mtime
- L'opzione **[-i]** permette di visualizzare gli inode number dei file
- L'opzione **[-n]** permette di visualizzare i numeri associati all'UID e al GID, invece del loro nome

Hard link e Soft link

💡 Un **hard link** (o collegamento fisico) è un **collegamento che associa un nome ad un file** (e di conseguenza al suo inode). Ogni file possiede almeno un hard link.
Un
soft link (o shortcut), invece, è un **puntatore al path di un hard link** o un altro soft link.

Se un file possiede più hard link, esso sarà accessibile tramite ognuno di tali link e la dimensione e **l'inode number** di tali hard link saranno quelli del file stesso, risultando
dunque
identici.

Inoltre, eseguendo il comando `rm`, tale file verrà effettivamente rimosso dal disco solamente quando **verranno rimossi tutti i suoi hard link**

La **dimensione** di un **soft link** corrisponde al numero di **byte necessari a conservare il path puntato.**

Per creare un link è possibile utilizzare il comando `ln [-s] src_link new_link`. Se l'opzione **[-s]** non viene utilizzata, verrà creato un hard link, mentre in caso contrario
verrà creato un soft link.

Poiché un soft link è un puntatore ad un **path**, se il file relativo a tale path viene spostato, rinominato o rimosso, tale soft link non sarà più valido.

Permessi di accesso ai file

Come già accennato, all'interno di ogni inode vengono specificati i **permessi di accesso al file** associato a tale inode.
Tali permessi di accesso corrispondono ad una
terna di terne bit: tre bit per i permessi di accesso del **proprietario (user)**, tre bit per i permessi di accesso del **gruppo associato**
(group)
e tre bit per **qualsiasi altro utente (other)**.
Per ognuna delle terne di bit, ognuno di essi corrisponde ad un permesso:

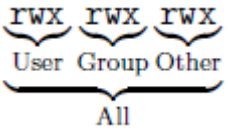
1. Il **primo bit** (partendo da destra) corrisponde al permesso di esecuzione (**execute**), indicato anche con una x. Se attivo, permette di **eseguire il file** (nel caso sia eseguibile).
2. Il **secondo bit** corrisponde al permesso di scrittura (**write**), indicato anche con una w. Se attivo, **permette di sovrascrivere, appendere in scrittura o cancellare** direttamente il file.
3. Il **terzo bit** corrisponde al permesso di lettura (**read**), indicato anche con una r. Se attivo, p**ermette di accedere al contenuto** del file.

Ognuno di tali bit può essere **impostato o non,** concedendo o meno il permesso ad esso associato.

Ad esempio, se i tre bit sono impostati su 011, vengono concessi solo i permessi di scrittura ad esecuzione. Difatti, utilizzando la **notazione alfabetica**, il permesso 011 corrisponde a -wx. Inoltre, trattandosi di terne di bit, il loro valore può essere interpretato anche in **base ottale**

Permesso	Valore binario	Valore ottale
- - -	000	0
- - x	001	1
- w -	010	2
- w x	011	3
r - -	100	4
r - x	101	5
r w -	110	6
r w x	111	7

Dunque, ogni terna di permessi associati ad un **inode** può essere interpretata come:



Permesso	Ottale	Effetto sulla directory
- - -	0	Nessuna operazione concessa
- - x	1	La directory può essere impostata come `cwd`, ma solo se tale permesso è concesso per ogni directory del path. Inoltre, è possibile "attraversarla" se il contenuto è già conosciuto
- w -	2	Nessuna operazione concessa
- w x	3	È possibile aggiungere file e directory, cancellare file contenuti in essa (anche senza avere il permesso di scrittura su tali file), cancellare directory contenute in essa (se si hanno tutti i permessi su tali directory)
r - -	4	Può essere solo elencato il contenuto della directory (senza gli attributi dei file) e non può essere "attraversata"
r - x	5	È possibile elencare il contenuto della directory (attributi compresi), impostare come `cwd` ed "attraversare". Tuttavia, non è possibile cancellare o aggiungere file alla directory
r w -	6	Come il permesso 4
r w x	7	Come il permesso 3, ma si può anche elencare il contenuto della directory (attributi compresi)

Per **modificare i permessi di accesso** di un file è possibile utilizzare il comando `chmod perms nome_file`, dove nell'argomento **perms** possono essere specificati (in più formati) i permessi da aggiungere, rimuovere o impostare per il file.

Esempi:

- Il comando `chmod 644 ciao.txt` imposta i permessi *rw–r—r—*
- Il comando `chmod +x ciao.txt` aggiunge il **permesso di esecuzione** per tutte e tre le terne
- Il comando `chmod u+r,g+w,o+x ciao.txt` aggiunge il permesso *r* per la terna User,
  il permesso w per la terna Group e il permesso x per la terna Other
-

Per **modificare il proprietario o il gruppo di appartenenza di un file**, invece, possono essere utilizzati i comandi `chown [-R] user nome_file` e `chgrp [-R] group nome_file`, dove l'opzione **[-R]** applica il comando ricorsivamente su tutte le sottodirectory nel caso in cui nome_file sia una directory.

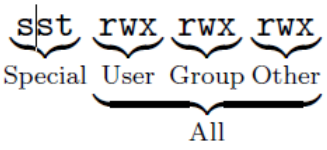
Tali comandi possono essere utilizzati solo da **root**, richiedendo quindi l'uso di `sudo`

Nel caso delle **directory** gli effetti ottenuti in base ai permessi associati non risultano del tutto intuitivi

Permessi speciali

Oltre alle tre terne di bit per i permessi di accesso, all'interno dell'inode di un file vi è un'**aggiuntiva terna di bit utilizzata per i permessi speciali**:

- Il **primo bit** (partendo da destra) corrisponde allo **sticky bit**.
  Se utilizzato sui file, esso risulta inutile. Se utilizzato sulle directory, invece, corregge
  il comportamento del permesso -wx per ogni terna, permettendo la cancellazione dei file solo se si hanno permessi di scrittura anche su di essi.
- Il **secondo bit** corrisponde al **SetGID bit**.
  Se utilizzato su un file non eseguibile, esso risulta inutile. Se utilizzato su un file
  eseguibile, invece, alla sua esecuzione i privilegi con cui opera il corrispondente
  processo vengono sostituiti con quelli del gruppo associato al file, facendo quindi prevalere i permessi della terna Group.
- Il **terzo bit** corrisponde al **SetUID bit**, analogo al SetGID bit ma con la terna User.
  Spesso, tale bit risulta essere "troppo potente", richiedendo limitazioni.



Esempi:

1.

- Siano D una directory, f un file in D e User1 e User2 due utenti tali che User1 != User2. Supponiamo inoltre che D appartenga a User1 e che D non appartenga al gruppo di User2
- Se lo sticky bit non è impostato su D, affinché User2 possa cancellare f è sufficiente che sia impostato il bit di scrittura nella terna Other di D.
- Se invece lo sticky bit è impostato su D, affinché User2 possa cancellare f è necessario che sia impostato anche il bit di scrittura nella terna Other del

file f e non solo della directory D

2.

- Il comando `passwd` (non il file /etc/passwd) ha il SetUID bit impostato, permettendo ad un utente di modificare la propria password (il proprietario dell'eseguibile è root).

Nonostante a livello effettivo sia presente un'ulteriore terna di bit, **nella notazione alfabetica tali bit non vengono rappresentati come una terna aggiuntiva**. Difatti se impostati, essi vengono visualizzati **al posto dei tre bit di esecuzione**:

- Se lo **sticky bit è impostato**, esso **rimpiazzerà il bit di esecuzione della terna Other**, visualizzando una **t minuscola** nel caso in cui anche tale bit di esecuzione sia **impostato** oppure una **T maiuscola** in caso **contrario**.
- Se il **SetGID bit è impostato**, esso **rimpiazzerà il bit di esecuzione della terna Group**, visualizzando una **s minuscola** nel caso in cui anche tale bit di esecuzione sia **impostato oppure** una **S maiuscola** in caso **contrario**.
- Se il **SetUID bit è impostato**, esso **rimpiazzerà il bit di esecuzione della terna User**, visualizzando una **s minuscola** nel caso in cui anche tale bit di esecuzione sia **impostato oppure** una **S maiuscola** in caso **contrario**.

Esempi:

- I permessi **rw–r—r—** (corrispondenti a **0644**) indicano che n**essun bit speciale è attivo**.
- I permessi **rwSr—r—** (corrispondenti a **4644**) indicano che il **SetUID bit è attivo**, ma il bit di esecuzione della terna User no.
- I permessi **rwsr—r—** (corrispondenti a **4744**) indicano che **sia il SetUID bit sia il bit di esecuzione** della terna User sono **entrambi attivi**.

User File-Creation Mask (umask)

La **User File-Creation Mask (umask)** definisce la **maschera dei file creati** dall'utente attuale, ossia i **permessi di accesso bloccati** di default per ogni file creato da tale utente stesso:

- Alla creazione di una **directory**, i suoi permessi di accesso vengono impostati automaticamente a **0777** AND NOT(umask)
- Alla creazione di un **file**, i suoi permessi di accesso vengono impostati automaticamente
  a
  0666 AND NOT(umask)

La UMASK di **default** è impostata a **0002**

Esempi:

- Supponiamo che per l'utente attuale si abbia che umask = 0022
- Se tale utente creasse una **directory**, i suoi permessi verrebbero impostati a
  0777 AND NOT(umask) = 0777 AND NOT(0022) = 0777 AND 7755 = 0755
  e dunque impostati a
  rwxr–xr–x
- Se tale utente creasse un **file**, i suoi permessi verrebbero impostati a
  0666 AND NOT(umask) = 0666 AND NOT(0022) = 0666 AND 7755 = 0644
  e dunque impostati a
  rw–r—r—

Per **modificare la umask** dell'utente attuale, è possibile utilizzare il comando `umask [mask]`. È necessario sottolineare che all'interno della **umask i primi tre bit non possano**
essere modificati
, implicando che il primo valore ottale sia **sempre 0**.

Lista Comandi Lezione 3

A seguito inserisco una lista di comandi approfonditi a lezione:

- `umask [mode]` la descrizione è nel man bash (cercate umask), serve per **modificare l'umask** di un file
- `cp [-r] [-i] [-a] [-u] {filesorgenti} filedestinazione` serve per **copiare** un file
- `mv [-i] [-u] [-f] {filesorgenti} filedestinazione` serve per **spostare** un file

- `rm [-f] [-i] [-r] {file}` serve per rimuovere un file, in linux **non è presente un cestino**, questo comando **libera l'inode del file**
- `ln [-s] sorgente [destinazione]` **crea** un **link** (senza -s è un hard link) da un file ad un altro
- `touch [-a] [-m] [-t timestamp] {file}` Serve per **modificare** i **timestamps** (timbri temporali) di un file ... e se il file non esiste lo **crea**
  - Può essere applicato anche su directory
  - -t imposta il timestamp desiderato
- `du [-c] [-s] [-a] [-h] [--exclude=PATTERN] [files...]` **Somma** tutte le **dimensioni** dei file e/o directories dati come argomento
- `df [-h] [-l] [-i] [file]` **Mostra** la **dimensione** e l'**attuale uso dei filesystem**
- `dd [opzioni]` Per **convertire** o **copiare files** in modo sofisticato

  Le opzioni sono una sequenza **variabile=valore**

  Le variabili più importanti sono:
  - **bs** dimensione di un singolo blocco in lettura/scrittura
  - **count** numero di blocchi da copiare
  - **convert**, in questo caso, il valore specifica una conversione, per esempio di codifica (da minuscolo a maiuscolo e viceversa, più svariate altre cose; vedere il man)
  - **skip** n° di bit nel file out da non sovrascrivere, di default è 0 quindi sovrascrive tutto
  - **if** file di input (se non dato, legge da tastiera (standard input))
  - **of** file di output (se non dato, scrive su schermo (standard output))

  Si usa anche per copiare file speciali che non possono essere copiati con cp
- `mkfs [-t type fsoptions] device` **Crea** un **filesystem** su **device**; a volte detto anche formattazione perché prepara i file a memorizzare file secondo un dato formato
  - **fsoption** può essere ro oppure rw
  - **device**: device file in /dev oppure file regolare, ad es. creato con dd

- Se è un **dispositivo**, non deve essere montato (/dev/hda, /dev/hda2, /dev/sdb1)

Approfondimento DD

E' possibile utilizzare dd anche come editor di file, scrivendo infatti `dd of=file.txt` è possibile scrivere in console ciò che voglio inserire nel file.txt, se esso non è presente lo crea, questo può essere utile nel caso la nostra distro (vecchia) non supporti `nano` o `touch`

Esercizio:

creare un file di testo, con il seguente contenuto:
ciao1
addio2
via3
ehila4
dove vai5
e poi copiarlo in un altro file in modo tale che quest'ultimo contenga solo i bytes che vanno dal
decimo al *ventesimo*;

- Per copiare solo i bit dal 10 al 20 uso count: `dd bs=1 count=10 if=file1.txt of=file2.txt`

rifare nuovamente la copiatura sullo stesso file destinazione, ma questa volta fare in modo che il contenuto del file destinazione sia duplicato

- Per aggiungere nuovamente i 10 bit devo usare (dopo aver fatto il comando precedente) lo stesso comando aggiungendo skip, ovvero lascio i primi 10 bit del fgile output invariati

```
dd bs=1 count=10 skip=10 if=file1.txt of=file2.txt
```

Processi

💡 Un **programma** è un file eseguibile salvato in memoria secondaria. Contiene l'insieme di istruzioni necessarie a svolgere un compito richiesto.
Un
processo è una particolare istanza attiva di un programma caricata in memoria principale, eseguendo sequenzialmente le istruzioni descritte nel programma stesso.

Non tutti i comandi avviano un processo:

- I comandi corrispondenti a programmi avviano un nuovo processo quando vengono eseguiti
- I comandi **build-in** nella shell **non avviano un nuovo processo**, venendo eseguiti all'interno del processo relativo alla shell stessa

Per avviare un **processo** è per tanto necessario eseguire un **programma**, digitando in una shell il nome del file associato. Poiché i sistemi operativi Unix sono **multi-processo**, prima di poter eseguire nuovamente tale programma, non occorre aspettare il termine dell'esecuzione del processo precedente. Per tanto, tale file eseguibile può essere eseguito più volte, dando vita ogni volta ad un nuovo processo.

Canali dei processi

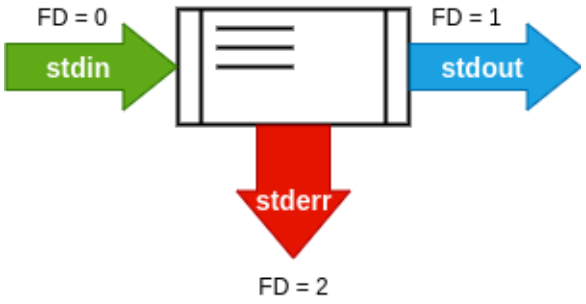
Ogni processo può avere **accesso a dei canali**, ossia dei **flussi** di **dati** in **uscita** o in **entrata** verso dispositivi o file. Ogni canale viene identificato univocamente all'interno del processo stesso tramite un valore intero detto file **descriptor**

Canali standard

Ogni processo Unix ha accesso ad almeno tre canali standard:

- **Standard Input (stdin)**, ossia il flusso dati predefinito in ingresso, corrispondente di default all'input da tastiera. Il suo file descriptor corrisponde a **0**.

- **Standard Output (stdout)**, ossia il flusso dati predefinito in uscita, corrispondente di default alla shell su cui è eseguito il processo. Il suo file descriptor corrisponde a **1**.
- **Standard Error (sterr)**, ossia il flusso dati predefinito per segnalazione di eventuali messaggi di errore e/o diagnostica, corrispondente di default alla shell su cui è eseguito il processo. Il suo file descriptor corrisponde a **2**.



All'interno della shell bash, ogni canale può essere **ridirezionato** :

- Tramite `cmd < nome_file` è possibile ridirezionare verso lo stdin del comando cmd il contenuto del file nome_file.
- Tramite `cmd N> nome_file` è possibile di ridirezionare il canale identificato dal file descriptor N del comando cmd verso il file nome_file. Se non viene specificato alcun file descriptor, viene utilizzato lo stdout come canale. Se il file possedeva già del contenuto, esso viene sovrascritto. Inoltre, se il file non esiste, esso viene automaticamente creato.
- Tramite `cmd N>>> nome_file` si ottiene lo stesso effetto di cmd N>>> nome_file, con la differenza che il contenuto venga appeso alla fine del file, invece che sovrascritto.
- Tramite `cmd N>&M` è possibile ridirezionare il canale legato al file descriptor N del comando cmd verso il canale legato al file descriptor M dello stesso comando

Esempi:

- Il programma cat nome_file restituisce sullo stdout il contenuto del file dato. Se non viene passato alcun file come argomento, il programma cat leggerà l'input direttamente da stdin fino a quando non verrà premuto CTRL+d. Per

tanto, è possibile ottenere lo stesso effetto di cat nome_file tramite cat < nome_file

- Tramite ls > out.txt, viene ridirezionato lo stdout del comando ls verso il file out.txt, sovrascrivendone il contenuto
- Tramite ls 2> out.txt, viene ridirezionato lo stderr del comando ls verso il file out.txt, sovrascrivendone il contenuto
- Tramite ls >˙> out.txt, viene ridirezionato lo stdout del comando ls verso il file out.txt, appendendo l'output alla sua fine
- Tramite ls 2>&1, viene ridirezionato lo stderr del comando ls verso il suo stdout
- Tramite ls 2> /dev/null, viene ridirezionato lo stderr del comando ls verso il dispositivo /dev/null, un dispositivo virtuale che funge da "buco nero" del sistema operativo. Per tanto, in tal modo lo stderr verrà completamente ignorato.

Le ridirezioni effettuate tramite <, > e >˙> non sono transitive

Pipelining

> 💡 Definiamo come **pipelining** il **ridirezionamento** dello **stdout** o **stderr** di un comando verso lo **stdin** di un altro comando. Il pipelining può essere effettuato ripetute volte, permettendo la **realizzazione** di **programmi complessi** tramite l'unione di **programmi più semplici**

All'interno della shell bash, il pipelining può essere effettuato tramite:

- Il simbolo **|**, il quale ridireziona lo **stdout** del comando alla sua sinistra verso lo stdin del comando alla sua destra (es: cmd1 | cmd2 | ...).
- Il simbolo **|&**, il quale ridireziona lo **stderr** del comando alla sua sinistra verso lo stdin del comando alla sua destra (es: cmd1 |& cmd2 |& ...).

Attributi e rappresentazione dei processi

Process ID (PID)

> 💡 Ogni processo è dotato di un valore intero, chiamato **Process ID (PID)**, che lo identifica **univocamente**

Nello stesso istante, all'interno della stessa macchina non possono essere presenti due processi aventi lo stesso PID. Una volta che un processo è terminato, il suo PID viene liberato, implicando che esso possa essere (prima o poi) riassegnato ad un nuovo processo.

In alcuni sistemi operativi, inclusi quelli Linux-based, i PID dei processi vengono assegnati **casualmente** tra quelli disponibili, rispetto ad un assegnamento incrementale.
In tal modo, all'riavvio di una macchina i processi avranno PID diversi rispetto alla sessione precedente, incrementando la sicurezza del sistema stesso.

Niceness

> 💡 Ogni comando è dotato di un valore intero nel range [−19, 20], detto **niceness**, che viene sommato al **valore di priorità di scheduling** dei processi di tale comando.
> Maggiore è il valore di priorità del processo,
> **minore** sarà la priorità data alla sua selezione da parte dello scheduler della CPU. Di default, la niceness di un comando è impostata a 0.

Il comando nice permette di **interagire con la niceness**:

- Se lanciato senza opzioni, permette di visualizzare la niceness di default per ogni comando
- Se lanciato con `nice [-n num] command`, viene avviato il comando command con valore di niceness impostato a num (0 se omesso)

Process Control Block (PCB)

💡 Ad ogni processo in esecuzione viene associata una struttura dati univoca detta

Processo Control Block (PCB), conservata all'interno del kernel.

All'interno del PCB di un processo, vengono conservate le seguenti informazioni:

- **PID** del processo
- **Parent PID (PPID)**, ossia il PID del processo padre tramite cui è stato avviato il processo stesso
- **Real UID (RUID)**, ossia lo UID dell'utente che ha avviato il processo
- **Real GID (RGID)**, ossia il GID dell'utente che ha avviato il processo
- **Effective UID (EUID)**, ossia lo UID attualmente assunto dal processo in esecuzione, non necessariamente uguale al RUID. Tale UID viene utilizzato come vero UID del processo, dettando i permessi attualmente ad esso concessi
- **Effective GID (EGID)**, ossia il GID attualmente assunto dal processo in esecuzione, non necessariamente uguale al RGID. Tale GID viene utilizzato come vero GID del processo, dettando i permessi attualmente ad esso concessi
- **Saved UID (SUID)**, ossia il precedente EUID assunto dal processo prima di aver assunto l'EUID attuale
- **Saved GID (SGID)**, ossia il precedente EGID assunto dal processo prima di aver assunto l'EGID attuale
- **State**, ossia lo stato del processo (vedi sezioni successive)
- **Current Working Directory (CWD)**, ossia la cwd attualmente "aperta" dal processo, corrispondente di default alla cwd in cui è stato avviato il processo
- **Root Directory**, ossia la directory utilizzata come base per i path assoluti all'interno del processo (di default è **/**)
- **Umask** dell'utente che ha avviato il processo
- **Niceness** del processo

SetUID e SetGID

> 💡 L'impostazione dei bit speciali **SetUID** e **SetGID** ha l'effetto di cambiare immediatamente l'EUID e l'EGID di un processo al suo avvio

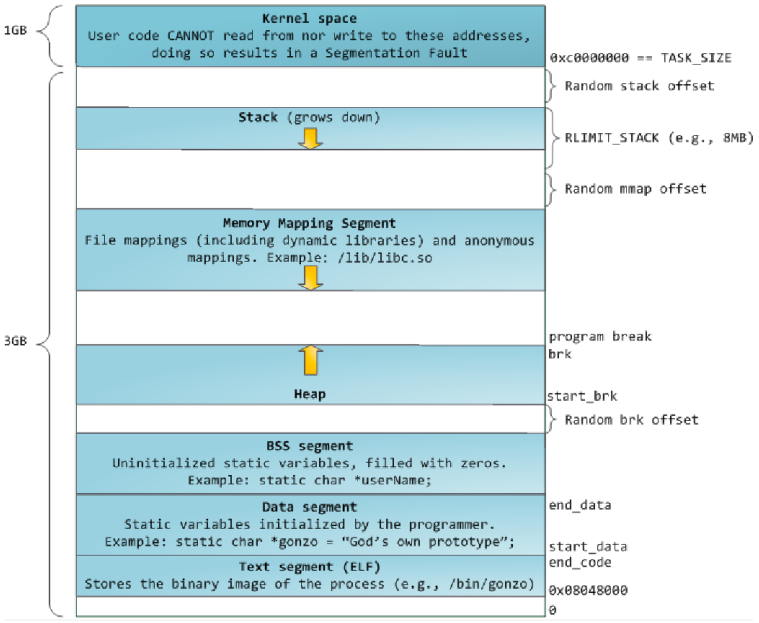
Saved UID: Saved User Identifier

> 💡 Permette ad un processo che gira come root (ad es. via setuid sull'eseguibile) di fare operazioni

Aree di Memoria

Come molti sistemi operativi, anche nei sistemi Linux-based i processi utilizzano i concetti di **suddivisione in aree e di memoria virtuale**, dunque la suddivisione della memoria in pagine.

- **Text Segment**, contenente le istruzioni in linguaggio macchina da eseguire. Viene condiviso da più istanze dello stesso processo
- **Data Segment**, contenente i dati statici (es: variabili globali, variabili locali statiche, ...)inizializzati all'avvio del processo e alcune costanti di ambiente. Potrebbe essere condiviso tra più processi.
- **BSS** (**Block Started from Symbol**), contenente i dati statici non inizializzati all'avvio del processo. Potrebbe essere condiviso tra più processi.
- **MMS** (**Memory Mapping Segment**), contenente tutto ciò che riguarda librerie esterne dinamiche utilizzate dal processo, fungendo anche da estensione dell'heap in alcuni casi. Potrebbe essere condiviso tra più processi.
- **Stack**, contenente i dati dinamici gestiti automaticamente dalle chiamate a funzioni. Mai condiviso tra processi.
- **Heap**, contenente i dati dinamici gestiti dal programmatore stesso.
- **Kernel space**, riservata esclusivamente al kernel.



Per ottenere **tutte le informazioni** inerenti ai processi attivi, in particolare le informazioni relative a tutti i processi presenti nella directory di sistema proc/, è possibile utilizzare il comando ps, dove:

- Senza opzioni aggiuntive vengono mostrati i processi dell'utente attuale lanciati nella **shell corrente**
- L'opzione **[-e]** permette di mostrare tutti i processi di tutti gli utenti lanciati in tutte le shell (ossia tutti i figli del processo 0)
- L'opzione **[-u user1,user2,...]** permette di mostrare tutti i processi degli utenti nella lista data
- L'opzione **[-p pid1,pid2,...]** permette di mostrare tutti i processi con PID interno alla lista data
- L'opzione **[-f]** mostra informazioni aggiuntive
- L'opzione **[-l]** mostra più informazioni aggiuntive rispetto all'opzione [-f]
- L'opzione **[-y]** permette di non mostrare le flag, sostituendo nell'output il campo ADDR con il campo RSS. Può essere utilizzato solo con l'opzione [-l]
- L'opzione **[-o field1,field2,...]** permette di scegliere i campi da mostrare nell'output

Tra i vari campi mostrabili dal comando **ps**, troviamo:

- **PID** e **PPID**
- **C**, ossia la parte intera della percentuale in uso della CPU da parte del processo
- **STIME** (o **START**), ossia l'ora in cui è stato avviato il comando (oppure la data se avviato da più di un giorno)
- **TIME**, ossia il tempo di CPU utilizzato finora
- **CMD**, ossia il comando e gli argomenti utilizzati per avviare il processo
- **F**, ossia varie flag associate al processo, tra cui:
  - F = 1, il processo è stato biforcato ma non ancora eseguito
  - F = 4, il processo ha utilizzato privilegi da super utente
  - F = 5, entrambi i precedenti
  - F = 0, nessuno dei precedenti
- **S**, ossia lo stato del processo, espresso con la sua lettera rappresentativa (vedi sezioni successive)
- **UID**, ossia lo UID con cui è stato avviato il processo (se il SetUID bit è impostato, potrebbe non coincidere con chi ha lanciato il comando sulla shell)
- **PRI**, ossia l'attuale priorità del processo (maggiore è il valore, minore è la priorità)
- **NI**, ossia la niceness del processo
- **ADDR**, ossia l'indirizzo di memoria del processo (mostrato solo per retro-compatibilità con versioni precedenti di ps)
- **SZ**, ossia il numero totali di pagine occupate dal processo sia nella RAM sia nel disco
- **RSS**, ossia la quantità di memoria in KB occupata dal processo all'interno della RAM (dunque escludendo la memoria all'interno delle pagine sul disco)
- **VSZ**, ossia la quantità di memoria in KB occupata dal processo, sia nella RAM sia sul disco
- **WCHAN**, ossia la funzione del kernel nella quale il processo si è fermato in attesa di un segnale (se in attesa)

Il comando `top [-b] [-n num] [-p pid1,pid2,...]` corrisponde ad una "versione interattiva" del comando ps (vedere il manuale per l'uso dettagliato), dove l'opzione **[-b]** disattiva i comandi interattivi ma aggiorna automaticamente l'output dopo pochi secondi, l'opzione **[-n num]** effettua solo num aggiornamenti e l'opzione **[-p]** risulta analoga a quella di ps.

Normalmente, per terminare il comando top è possibile premere sia il tasto q che i tasti **CTRL+c.** Utilizzando l'opzione [-b], invece, il tasto q verrà disabilitato

Infine, per visualizzare le syscall effettuate da un processo attivo, è possibile utilizzare il comando `strace [-p pid]`, mentre il comando `strace command` permette di lanciare un comando e visualizzare le syscall da esso effettuate.

Stati ed esecuzione di un processo

In ogni istante, ogni processo assume un determinato stato:

- **Running** (**R**), indicante che il processo è in esecuzione su un processore
- **Runnable** (**R**), indicante che il processo è pronto ad essere mandato in esecuzione su un processore da parte dello scheduler
- **Sleep** (**S**), indicante che il processo è in attesa di un evento e non può essere scelto dallo scheduler
- **Zombie** (**Z**), indicante che il processo è terminato e le sue 6 aree di memoria sono state disassociate, tuttavia il suo PCB è ancora presente nel kernel poiché il suo processo padre non ha ancora richiesto il suo exit status, ossia non ha "chiuso" il figlio
- **Stopped** (**T**), corrispondente ad un caso particolare di sleep, dove, a seguito della ricezione di un segnale STOP, viene atteso un segnale CONT
- **Traced** (**t**), corrispondente ad un caso particolare di sleep, dove è in esecuzione il debugging del processo
- **Uninterruptible Sleep** (**D**), corrispondente ad un caso particolare di sleep, il quale non può in alcun modo essere interrotto

Job

💡 Definiamo come **job** un qualsiasi "lavoro" svolto all'interno di una shell. In particolare, un job può essere composto anche da **più comandi** (es: tramite il pipelining) e dunque da più processi (uno per comando).

Ogni job può essere eseguito secondo **due modalità:**

- **Esecuzione in foreground** (trad: in primo piano), dove:
  - I sotto-comandi del job possono leggere l'input da tastiera e scrivere sul terminale
  - Finché esso non termina, il prompt non viene restituito e non possono essere lanciati altri job all'interno della stessa shell
  - Ogni job lanciato viene eseguito in foreground di default
- **Esecuzione in background** (trad: in sottofondo), dove:
  - I sotto-comandi del job non possono leggere l'input da tastiera, ma possono scrivere sul terminale
  - Il prompt viene immediatamente restituito al loro avvio
  - Mentre il job viene eseguito in background, possono essere eseguiti altri job sulla shell
  - Nella shell bash, un job può essere avviato in background aggiungendo il simbolo & alla fine del comando stesso

  In ogni istante, all'interno di una shell può esserci **solo un job in foreground**, ma più job in foreground possono essere eseguiti su più shell contemporaneamente.

Job number

Ogni job eseguito all'interno di una shell possiede un **job number**, partendo dal numero

Due job possono avere lo stesso job number se eseguiti all'interno di due shell diverse.

Per **interrompere temporaneamente** il processo in foreground, è possibile premere **CTRL+z**, mandando tale processo in stato di **Stopped (T)**, mentre per terminarlo direttamente è possibile premere **CTRL+c.**
Un processo in stato di Stopped (T), può essere
rimandato in esecuzione tramite due comandi:

- Il comando **fg %N** permette di mandare in esecuzione in foreground il job N
- Il comando **bg %N** permette di mandare in esecuzione in background il job N

Per **entrambi i comandi**, al posto del parametro %N è possibile specificare anche:

- %prefix, dove prefix è la parte iniziale del comando del job desiderato
- %+, %% oppure nulla, per selezionare l'ultimo job eseguito (o rieseguito)
- %-, per selezionare il penultimo job eseguito (o rieseguito)

Segnali dei processi

Un **segnale** è un evento (o interruzione software) generato dal kernel o da un processo a seguito di specifiche condizioni ed inviato verso un altro processo. Quando un processo riceve un segnale, esso reagisce eseguendo l'**azione predefinita** per tale segnale o un'**azione personalizzata** definita all'interno del programma del processo stesso.

Nei sistemi Linux-based, ogni segnale è identificato da un **numero** o da un **nome** e rientra in una **categoria** in base alla sua azione predefinita:

- **Segnale di Terminazione**, ossia viene richiesta la terminazione del processo
- **Segnale Ignorato**, ossia non viene svolta alcuna operazione
- **Segnale di Core dump**, ossia viene richiesta la terminazione del processo e viene effettuato un **core dump** (viene registrato in un file lo stato attuale della memoria e della CPU)
- **Segnale di Stop**, ossia viene messo il processo in stato di Stopped (T)

- **Segnale di Continuazione**, ossia viene richiesta la continuazione di un processo in stato di Stopped (T)

Tra i vari segnali inviabili, troviamo:

Nome	Numero	Categoria	Significato o Condizione scatenante
SIGINT	2	Terminazione	Invio di un `CTRL+c` da tastiera
SIGQUIT	3	Core dump	Uscita
SIGILL	4	Core dump	Istruzione illegale
SIGABR	6	Core dump	Abort
SIGFPE	8	Core dump	Eccezione di tipo aritmetico
SIGKILL	9	Terminazione	Terminazione forzata del processo
SIGUSR1	10	Terminazione	Definito dall'utente
SIGSEGV	11	Core dump	Segmentation Fault
SIGUSR2	12	Terminazione	Definito dall'utente
SIGPIPE	13	Terminazione	Scrittura senza lettori su pipe o socket
SIGALRM	14	Terminazione	Allarme temporizzato
SIGTERM	15	Terminazione	Terminazione del processo
SIGCHLD	17	Ignorato	Status del figlio cambiato
SIGCONT	18	Continuazione	Ripresa dell'esecuzione
SIGSTOP	19	Stop	Sospende del processo
SIGTSTP	20	Stop	Invio di un `CTRL+z` da tastiera
SIGTTIN	21	Stop	Lettura da terminale in background
SIGTTOU	22	Stop	Scrittura su terminale in background

A differenza degli altri segnali, i segnali SIGKILL e SIGSTOP **non possono essere gestiti** all'interno del programma. Per tanto, la loro azione non può essere personalizzata.

Le shortcut da tastiera CTRL+z e CTRL+c utilizzabili all'interno delle shell bash, inviano rispettivamente un segnale di SIGTSTOP e di SIGINT al job in foreground.
I comandi fg e bg inviano un segnale SIGCONT al job specificato (con la differenza che fg riporti anche tale job in primo piano).

Per inviare un segnale ad un processo in esecuzione, è possibile utilizzare il comando kill, dove:

- Il comando `kill -l [signal]` lista il numero e il nome del segnale signal (listando tutti i segnali disponibili se omesso)
- I comandi `kill -signal pid` e `kill -s signal pid` permettono di specificare (come numero, nome o abbreviativo) il segnale da inviare al processo avente pid come PID

(es: i comandi
`kill -9 pid`, `kill -s SIGKILL pid` e `kill -s KILL pid` sono equivalenti)

- All'interno di ogni comando richiedente l'argomento pid, è possibile inviare il segnale ad un job invece che ad un processo specificandone il numero con %N al posto di pid
  (es:
  `kill -9 %1` invia un segnale SIGKILL ad job 1 della shell)

Un processo considererà un segnale ricevuto solo se l'UID di chi ha inviato tale segnale corrisponde al RUID del processo

Linguaggio C

Introduzione

Come già discusso, il linguaggio di programmazione C venne sviluppato dagli AT&T Bell Labs agli inizi degli anni '70. Esso fu utilizzato per sviluppare il kernel di Unix (successivamente anche il kernel Linux) ed altri sistemi operativi. Venne standardizzato dall'American National Standards Institute (ANSI) e successivamente anche dall'International Organization for Standardization (ISO).
Ogni programma scritto tramite il linguaggio C possiede una **funzione principale obbligatoria detta main()**, la quale può anche essere semplicemente il punto da cui vengono invocate tutte le altre funzioni che compongono il programma o essere direttamente l'unica funzione del programma. Tutte le funzioni dello stesso programma, inclusa main(), possono risiedere in un unico file o essere distribuite su più file.
Ogni funzione consiste di un **'intestazione (header)**, a sua volta composta dal nome della funzione, dal tipo di valore ritornato e da una lista di parametri in input, ed un **blocco di istruzioni**:

```
<return-type> function-name (parameter-list){
        instruction 1;
        instruction 2;
}
```

Ogni **statement** (ossia un'istruzione o parte di essa) è terminato da un carattere **;**.
Solitamente, all'interno di ogni statement sono presenti delle **keyword**, ossia delle "parole riservate" ben definite direttamente all'interno del linguaggio stesso (es: if, int, ...)
Di fondamentale importanza è la keyword **return <value>**, la quale imposta il valore di ritorno della funzione al valore **<value>**, terminando immediatamente l'esecuzione della funzione stessa e tornando alla funzione chiamante.

All'interno del codice è possibile inserire **commenti**, ossia linee di testo ignorate dal compilatore, tramite due modalità:

- I caratteri **//** rendono un commento tutto ciò che è successivo ad essi fino alla fine della riga
- I caratteri **/*** rendono un commento ciò che è successivo ad essi fino ai successivi caratteri ***/**

Esempio Primo Programma, Hello Word!

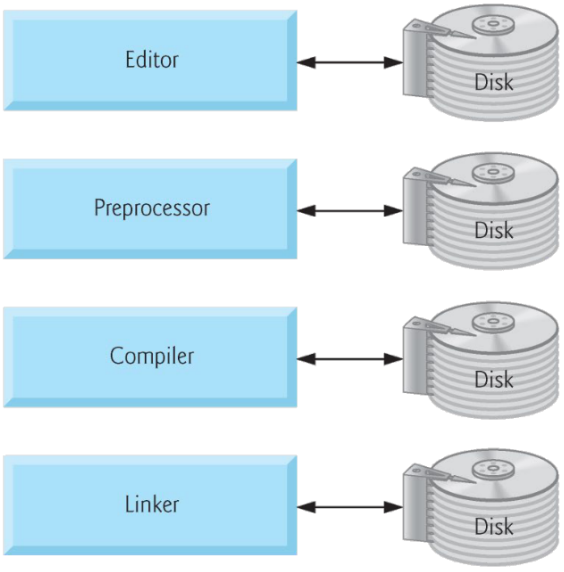
```
#include <stdio.h>
int main() {
    printf("Hello World!\n"); //stampa sul terminale
    return 0; // terminato con successo
}
```

Ambiente di sviluppo

Tra le caratteristiche principali del linguaggio C troviamo **l'indipendenza dall'hardware**, in grado di rendere le applicazioni compilabili ed eseguibili su ogni tipo di processore (con eventuali leggere modifiche al codice, se necessario).

L'**ambiente di sviluppo** relativo al linguaggio C prevedere **4 fasi** necessarie alla creazione di un programma, ognuna svolta da un programma indipendente:

1. Il programmatore **scrive il codice** del programma tramite un **editor di testo** (nano, vim, ecc)
2. Il **pre-processore** (o pre-compilatore) processa il codice, eseguendo varie direttive e preparando la compilazione
3. Il **compilatore** compila il codice, producendo un file oggetto, ossia una parte incompleta del programma finale e non eseguibile
4. Il **linker** collega tra loro i vari file oggetto, creando il file eseguibile finale



Nel sistema operativo GNU/Linux, il programma **gcc (GNU Compiler Collection)** è in grado di svolgere tutte le fasi necessarie alla creazione di un file eseguibile.
Per
precompilare un programma C, è possibile utilizzare il comando cpp program.c > precompiled.c. Tramite il carattere # vengono definite le **direttive del pre-processore**, come la **direttiva #include filename**:

Direttiva #include

La direttiva **#include** filename impone al pre-processore di inserire il contenuto del file specificato al posto della direttiva stessa.
I file inclusi seguono (non necessariamente) la seguente struttura:

- Possiedono un'estensione **.h** (es: filename.h) e vengono detti header file

- Se circondati dai caratteri **< >**, viene specificato che il header file è un file standard del linguaggio C ed è situato nella directory /usr/include (es: `#include <stdio.h>`)
- Se circondati dai caratteri **" "**, viene specificato che il header file è dell'utente e che si trova nella directory corrente o in un path specificato (es: #include "file.h"). Vedere il manuale del comando gcc per maggiori informazioni

Per **compilare** un programma C, è possibile utilizzare il comando `gcc program1.c program2.c` ..., dove uno dei solo dei file specificati contiene la funzione main() del programma.

Il programma **gcc** include molte opzioni:

- L'opzione **[-Wall]** mostra tutti i messaggi di avvertimento (warning) presenti alla compilazione
- L'opzione **[-Wextra]** mostra ulteriori warning "non-standard"
- L'opzione **[-o filename]** permette di specificare il nome del file di output. Se non utilizzata, il nome utilizzato sarà a.out
- L'opzione **[-c]** effettua solo la compilazione, dando per assunto che il file in input
  sia stato pre-compilato (vedi in seguito)
- L'opzione **[-lm]** permette di includere librerie matematiche come <math.h>

Inoltre, a seconda dei file in input specificati, il comando gcc assume comportamenti diversi:

- Tramite il comando gcc -c precompiled.c, dove precompiled.c è stato precompilato, viene eseguita solo la compilazione:
  - Viene controllato che la sintassi del codice sia corretta
  - Per ogni chiamata di funzione, viene controllato che venga rispettato il rispettivo header
  - Viene creato dell'effettivo codice macchina, ma solo per il contenuto delle funzioni
  - Ogni chiamata a funzione possiede una destinazione simbolica

- Con il comando `gcc -c file.c -o file.o`, dove file.c non è stato precompilato, viene eseguita **sia la precompilazione sia la compilazione**
- Con il comando `gcc file.o`, dove file.o è stato compilato, viene effettuato il **linking**:
  - Vengono risolte le chiamate a funzione, aggiungendo anche il blocco di ognuna di esse alla loro intestazione
  - L'implementazione di tali funzioni può essere data dal programmatore o fornita tramite librerie di sistema
  - L'inclusione delle librerie può essere automatica o specificata dall'utente
- Con il comando `gcc file.c`, dove **file.c non è precompilato**, vengono **eseguite tutte le fasi**