



DIPARTIMENTO
DI INFORMATICA
SAPIENZA
UNIVERSITÀ DI ROMA

Sistemi Operativi II

Authors

Emanuele D'Agostino
Giuseppe Borracci

GitHub

[Rurik-D](#)
[GiusTMP](#)

Indice

0 Introduzione

0.1

0.1.1

0.2

0.2.1

1

1.1

1.1.1

1.2

1.2.1

1.3

1.3.1

1.4

1.4.1

1.5

1.5.1

0 Introduzione

0.1 Unix: da Multics a Linux

Nel 1965 venne presentato alla Fall Joint Computer Conference un nuovo sistema operativo, il quale, nella sua concezione originale, avrebbe dovuto servire la necessità di calcolo dell'intera città di Boston, similmente ad un servizio di distribuzione di elettricità o acqua corrente. Stiamo parlando di Multics (Multiplexed Information and Computing Service): uno dei primi sistemi operativi a supportare l'esecuzione di applicazioni in [time-sharing](#).



Multics fu un progetto sviluppato da parte di un consorzio di imprese che comprendeva Project MAC (in seguito MIT Laboratory for Computer Science (LCS)) General Electric e Bell Laboratories.

La vera importanza storica di Multics risiede nel fatto che fu il primo sistema operativo a mettere sul campo tutta una serie di concetti e tecniche costruttive che sono ancora oggi elementi essenziali dei moderni sistemi operativi.

Il progetto di Multics venne però abbandonato ben presto dalla direzione dei Bell Laboratories in quanto ritenuto troppo complesso. Alcuni ricercatori non ritennero corretta la decisione presa e decisero nonostante tutto di continuare lo sviluppo del progetto. Tra questi ricercatori citiamo Ken Thompson e Dennis Ritchie, grazie ai quali nacque la prima versione, scritta totalmente in assembly, di Unics (*Uniplexed Information and Computing Service*), che in seguito cambiò definitivamente in **Unix**. Nome che stava a sottolineare la semplicità del progetto rispetto alla mal gestita complessità di Multics.

Una delle fasi più importanti nella storia di Unix, fu di certo l'invenzione del linguaggio di programmazione C. Sviluppato da Thompson e Ritchie tra il '69 e il '73, il C permise di portare il kernel su piattaforme diverse dall'originario PDP-7, costituendo di fatto il primo software della storia ad essere in grado di funzionare in ambienti totalmente diversi.

Assieme al kernel, Unix è stato corredato di una serie di applicazioni standard per la gestione dei file e degli utenti, che continuano ancora ad essere usate nei sistemi operativi moderni.

I Bell Laboratories erano di proprietà dell'AT&T (società che gestiva le comunicazioni telefoniche negli Stati Uniti), la quale deteneva i diritti di Unix. Proprio in quel periodo, all'inizio degli anni settanta, il sistema telefonico statunitense stava subendo una piccola rivoluzione interna: l'utilizzo di mini-computer per la gestione del traffico voce e dati. Questi erano dotati di software di tipo minimale, che permetteva operazioni di manutenzione piuttosto limitate. Ben presto, si scoprì come Unix, grazie alla sua concezione moderna e alla sua versatilità, permettesse ai mini-computer di fare operazioni molto più complesse. Per la prima volta, le operazioni di manutenzione potevano essere gestite a livello centrale, senza spedire i tecnici a investigare sul posto ad ogni singolo guasto.

Al centro di forti critiche per via della sua posizione dominante, AT&T permise che il codice sorgente di Unix venisse distribuito gratuitamente per fini di studio alle università di tutto il mondo (per una manovra politica più che per beneficenza).

Ottenere una copia del sistema operativo era piuttosto semplice e davvero poco costoso. In breve tempo, si formò quindi una comunità mondiale a livello universitario incentrata sullo

sviluppo di nuove componenti e applicazioni per Unix, seguendo le linee guida dell'open source moderno. Grazie a questo processo, nel corso degli anni settanta, videro la luce le prime versioni del sistema operativo, tra cui citiamo:

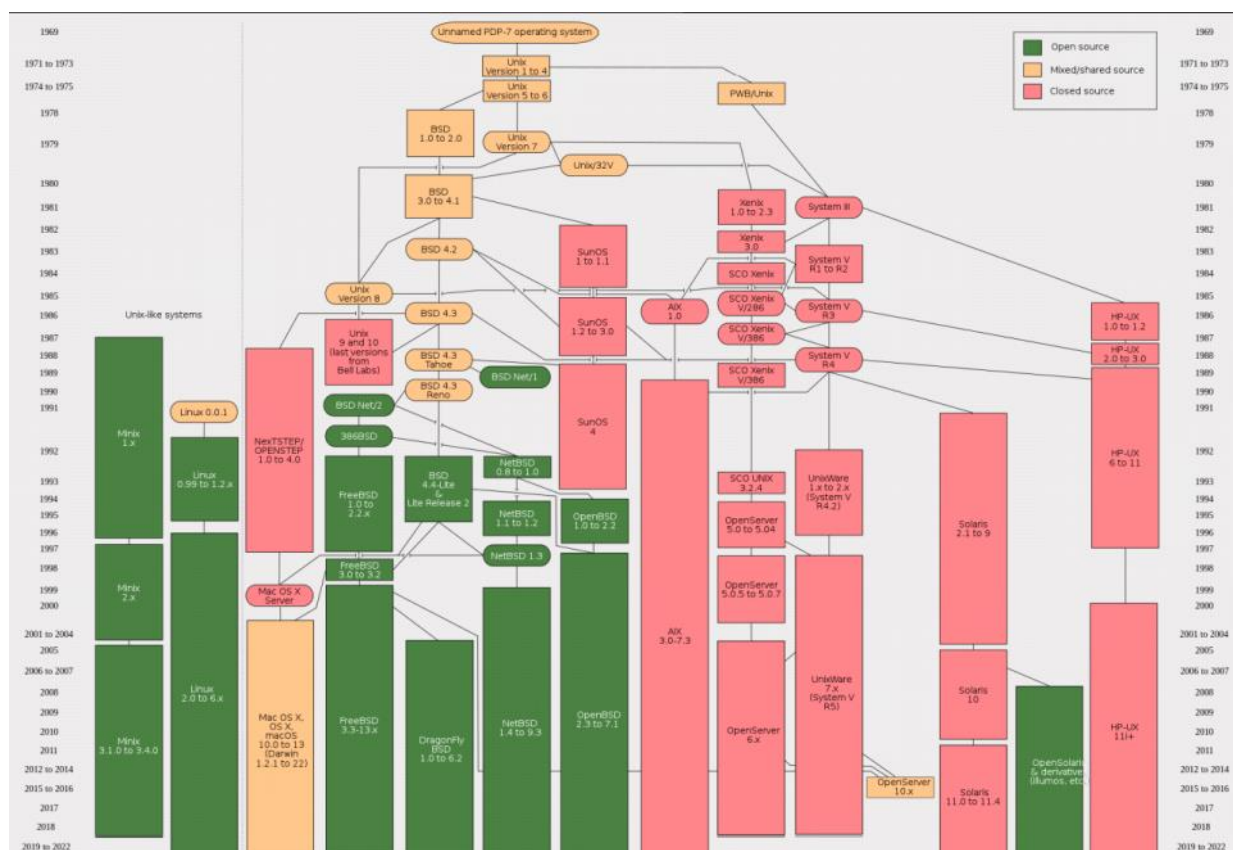
- System V AT&T (Bell Labs)
- BSD (Berkeley Software Distribution, Università di Berkeley)
- Xenix (Microsoft)
- SunOS/Solaris (Sun Microsystems, poi acquisita da Oracle)

Unix costituì un forte aggregatore per la nascente scienza dell'informazione. Di fatto, si può addirittura affermare che fu il suo sviluppo congiunto a definire, per la prima volta, l'idea di informatica come scienza.

Il primo nucleo del kernel Linux fu creato nel '91 dal giovane studente finlandese di informatica Linus Torvalds che, appassionato di programmazione, era insoddisfatto del sistema operativo Minix (sistema operativo destinato alla didattica basato su un'architettura a microkernel), poiché supportava male la nuova architettura i386 a 32 bit, all'epoca economica e popolare. Così Torvalds decise di creare un kernel unix con lo scopo di divertirsi e studiare il funzionamento del suo nuovo computer, che era un 80386.

Nel 1994 viene definito lo standard per Unix e, in quegli anni, si vengono a definire 3 categorie di sistemi Unix:

- **Generis Unix** Sistemi Unix che provengono da quello dell'AT&T o da quelli da lui derivati (BSD).
- **Trademark Unix** Sistemi Unix che pagano la royalties al SUS per essere definiti Unix.
- **Functional Unix** Sistemi operativi che si "ispirano" a Unix. Qui rientrano Linux e anche altri sistemi come Minix, anche detti impropriamente "Unix-like" o, più propriamente, Unix system-based.



0.2 Caratteristiche di un SO Unix

Tra le caratteristiche di un sistema Unix possiamo apprezzare:

- Multi-utente e multi-processo
- File system (sistema di archiviazione) gerarchico, anche su rete (NFS)
- **Kernel**
Gestisce memoria (principale e secondaria), processi, I/O, risorse hardware in generale
- **System Call**
Funzioni C che possono essere chiamate se ci si vuole interfacciare con il kernel (ad esempio, per creare un file...)
- **Shell**
Programma interattivo e/o batch («a lotti») che accetta comandi da "girare" al kernel (del tipo: mostra il contenuto di una directory)
- Ambienti di programmazione
- Programmi di utilità
- Modularità

I sistemi Unix sono composti da una serie di piccoli programmi che eseguono compiti specifici, limitati e in maniera esatta e semplice (modularità). I programmi sono "silenziosi": il loro output è minimale ed è ridotto a ciò che è stato esplicitamente richiesto, o a ciò che costituisce particolare interesse. L'esecuzione di compiti complessi può essere quindi suddivisa in più programmi che eseguono frazioni del compito originale. Tali programmi inoltre manipolano testo e non file binari.

0.3 La shell

La **shell**, in italiano *interprete dei comandi*, è la componente del sistema operativo che permette all'utente il più alto livello di interazione con lo stesso (spesso viene chiamata *terminale*). Tramite la shell è possibile impartire comandi e richiedere l'avvio di altri programmi.

Ci sono svariati tipi di shell, per citarne alcune:

- Thompson/Bourne shell sh
- Bourne-Again shell bash
- KornShell ksh
- Friendly Interactive shell fish

Nei sistemi Linux troviamo generalmente la bash: una shell testuale che tecnicamente è un clone evoluto della shell standard di Unix (sh).

0.3.1 Bash: cronologia

Con i tasti *freccia su* e *freccia giù* potete scorrere la lista dei comandi dati e, una volta visualizzato il comando selezionato, questo può essere modificato. È possibile inoltre ricercare un comando data una certa keyword con la combinazione di tasti CTRL+r.

0.3.2 Bash: prompt

Il **prompt**, termine traducibile in italiano come *richiesta*, indica una richiesta che l'elaboratore trasmette al suo utilizzatore, attraverso l'interfaccia utente, al fine di sollecitarne un'azione. La **bash scrive un prompt ed attende** che l'utente scriva un comando. Il prompt tipico su bash è così costituito:

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

Dove path è il percorso **dalla cartella** (directory o folder) **home alla cartella attuale**. Se si è semplicemente nella home, sarà visualizzato solo il simbolo ~ (tilde). Se la cartella corrente non si trova nel sottoalbero radicato nella home, allora path sarà il percorso assoluto.

0.3.3 Bash: comando

Vediamo ora la sintassi dei comandi; ogni comando verrà indicato come segue

```
comando [opzioni] argomenti_obbligatori
```

Ci dev'essere almeno un argomento (ma ce ne può essere anche più d'uno) se {argomento}

```
cp [-r] [-i] [-a] [-u] {filesorgenti} file_destinazione
```

Ci possono essere 0, 1 o più argomenti opzionali se [opzione...]

```
ps [opzioni] [pid...]
```

Argomenti vanno separati da carattere indicato se [carattere opzione...]

```
chmod mode[, mode...] file_name
```

Opzioni tipicamente composte da

```
-carattere (vecchia versione)
--parola (nuova versione)
```

Ad esempio sono equivalenti per il comando cp

```
-i      --interactive
-r      --recursive
```

Le opzioni possono avere un argomento

```
-k1, -k 1, --key=1
```

Le opzioni senza argomento sono raggruppabili

```
-b -r -c sono equivalente a -brc
```

Esistono anche le opzioni in stile BSD, senza dash '-'

```
tar xfz nomefile.tgz
```

1 Il filesystem

1.1 Introduzione

Un **file system** (in acronico **FS**), indica informalmente un meccanismo con il quale i **file** e le **directory/folder** sono posizionati e organizzati su dispositivi utilizzati per l'archiviazione dei dati, come **unità di memoria di massa** (unità a nastro magnetico, dischi rigidi, dischi ottici, SSD, ecc...) o su **dispositivi remoti** tramite protocolli di rete e dei file.

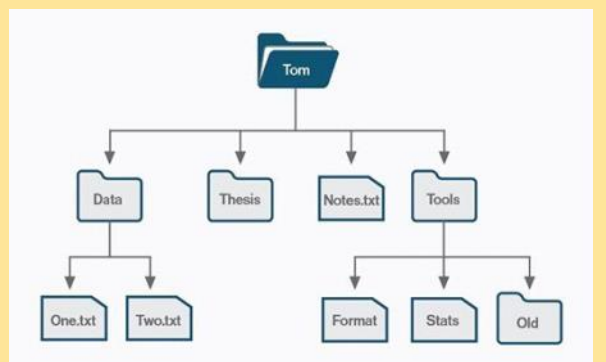
Esistono principalmente due tipi di file:

- **File regolari** Sono una **sequenza di byte** registrata su memoria di massa, visualizzabili anche tramite normali editor di testo.
- **File non regolari** Possono essere **interpretati** solo da programmi o funzioni di libreria che implementano opportuni algoritmi. Si pensi ai file compressi nei vari formati, o ai file Word, ai file JPEG, e ai tanti altri tipi di file che servono a memorizzare informazioni complesse sempre secondo una opportuna sequenza di byte: la differenza sta nella logica con cui l'informazione viene codificata, ma sono comunque delle sequenze di byte.

Linux e *Unix* hanno un solo filesystem principale che ha come directory radice / (*root*) e ogni file o directory è contenuto direttamente o indirettamente in /. In una stessa directory è vietato creare 2 file con lo stesso nome, 2 directory con lo stesso nome e un file ed una directory con lo stesso nome. Inoltre, i nomi dei file e delle directory sono *case sensitive*, quindi ad esempio, Amore.txt \neq amore.txt.

1.2 Directory

Una directory o cartella è una specifica entità del file system che elenca altre entità, tipicamente file e altre directory e, che permette di organizzarle in una struttura ad albero (dove solo le directory possono avere figli e i file sono foglie). È pertanto definibile come **un percorso (path) in cui sono presenti file o altre directory**.



Un **path** è una **sequenza di uno o più nomi** (separati da / *slash*) **di directory che terminano, eventualmente, col nome di un file**.

Un path può essere:

- **Absolute** Il path specifica una posizione nel file system a partire dalla directory principale.
- **Relativo** Il path specifica una posizione nel file system a partire dalla directory corrente (rappresentabile anche tramite ' ./ ').

1.2.1 Working directory (cwd)

Ogni processo ha una **working directory**, a volte chiamata *current working directory*. Questa è la directory da cui vengono interpretati tutti i path relativi. Un processo può cambiare la sua working directory con la funzione **cd [path]** dove:

- [Path] può essere assoluto o relativo
- cd senza path ritorna alla home

Inoltre nel path si possono usare:

- .. (directory "genitore")
- . (directory attuale)

1.2.2 Contenuto di una directory

Per visualizzare il contenuto di una directory ci basta usare il comando **ls**. Questo comando ci restituisce la *lista dei file* contenuti in una directory, per avere un'idea delle opzioni disponibili basta consultare il *man*.

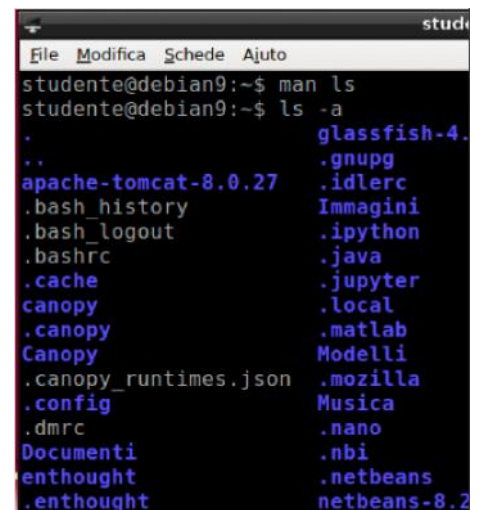
Digitando soltanto **ls**, possiamo vedere il contenuto della current working directory. Per mostrare il contenuto di *un'altra directory*, dobbiamo digitare **ls nomedir**.

In una directory ci sono dei file "nascosti". Tipicamente questi file, sono file di configurazione o file usati a supporto di comandi e applicazioni. Questi file nascosti, sono preceduti da un punto (come possiamo notare dall'immagine qui di fianco).

Con il comando **ls [-a | --all]** possiamo visualizzare questi file nascosti.

Se invece vogliamo visualizzare ricorsivamente il contenuto delle sotto directory abbiamo l'opzione

[-R | --recursive].



Per visualizzare l'albero delle directory invece, usiamo il comando

tree [-a] [-L maxdepth] [-d] [-x] [nomedir].

1.2.3 Creazione di directory e file

Se vogliamo creare una directory da terminale ci basterà usare il comando

mkdir [-p] nomedir

in questo modo andremo a creare una directory di nome **nomedir** vuota.

Con questo comando possiamo anche creare un intero path di directory ad esempio:

mkdir -p /dir11/dir12/dir13

Digitando il comando **man mkdir**, possiamo consultare la sinossi (si provi a digitare il comando **man man**).

Se invece vogliamo creare un file, abbiamo a disposizione il comando **touch**; digitando

touch nomefile

creiamo un file di nome **nomefile** vuoto.

Se vogliamo modificare il contenuto di un file, possiamo usare diversi comandi, come ad esempio: *gedit*, *nano*, *pico*, *vi*. Su *Ubuntu* ad esempio possiamo usare *nano* o *vi*.

1.3 Mounting e partizioni

Il filesystem root(/) contiene elementi eterogenei come ad esempio:

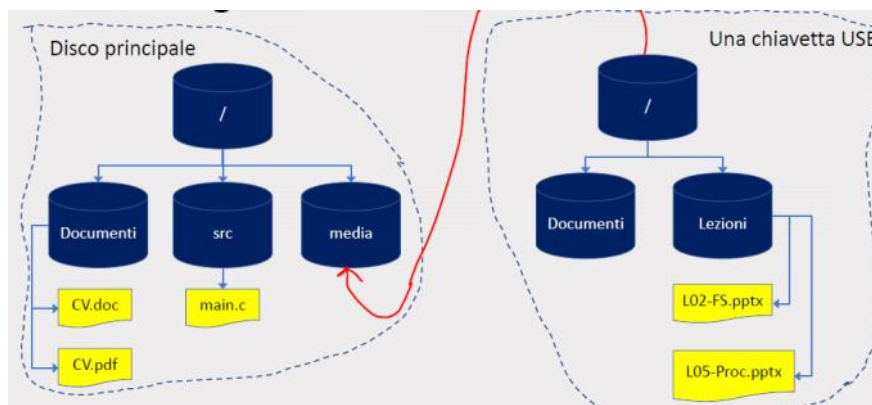
- Disco interno solido o magnetico
- Filesystem su disco esterno (e.g., usb)
- Filesystem di rete
- Filesystem virtuali (usati dal kernel per gestire risorse)
- Filesystem in memoria principale

Tutto questo grazie al **mounting**. Il comando *mount* collega il filesystem di un dispositivo esterno al filesystem di sistema.

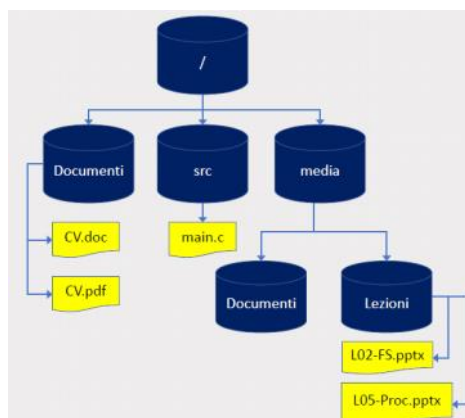
Indica al sistema operativo che il filesystem è pronto per l'uso e lo associa a un punto particolare nella gerarchia del sistema. Il *mounting* renderà disponibili agli utenti, file, directory e dispositivi.

Al contrario, il comando *umount* smonta il punto di mounting e scollega il dispositivo dal sistema.

Esempio:



il risultato che otteniamo è il seguente:



Una qualsiasi directory D dell'albero gerarchico può diventare il punto di mount per un altro (nuovo) filesystem F se e solo se la directory root di F diventa accessibile da D :

- Se D è vuota, dopo il mount conterrà F
- Se D non è vuota, dopo il mount conterrà F ma ciò non significa che i dati che vi erano dentro sono persi, ma saranno di nuovo accessibili dopo l'unmount di F

Una *partizione*, indica una suddivisione logica di un'unità di memorizzazione fisica (tipicamente una memoria di massa come un disco rigido o una chiavetta USB). Le singole unità logiche vengono viste dal sistema operativo come unità separate e possono essere formattate e gestite in maniera del tutto indipendente.

Su ogni disco rigido è sempre presente almeno una partizione per la sua operatività, al più l'intera memoria vista come singola o unica partizione.

Un singolo disco può essere suddiviso in due o più partizioni. Una partizione A può contenere il sistema operativo e la partizione B , i dati degli utenti (home directory degli utenti). In questo modo la partizione A verrà montata su $/$ e la partizione B su $/home$.

1.4 Tipi di filesystem

Dal punto di vista dell'utente, il tipo di un filesystem dipende da:

- Dimensione max partizione
- Dimensione max file
- Lunghezza max nome file
- Se Journal filesystem o no

Nome	Journal	Partiz (TB)	File (TB)	Nome file (bytes)
Ext2	No	32	2	255
Ext3	Sì	32	2	255
Ext4	Sì	1000	16	255
ReiserFS	Sì	16	8	4032

il *journaling* è una tecnica utilizzata da molti file system moderni per preservare l'integrità dei dati da eventuali cadute di tensione. Derivata dal mondo dei database, il *journaling* si basa infatti sul concetto di transazione dove ogni scrittura su disco è interpretata dal file system come tale.

Quando un applicativo invia dei dati al file system per memorizzarli su disco, questo prima memorizza le operazioni che intende fare su un file di *log* e in seguito provvede a effettuare le scritture sul disco rigido, quindi registra sul file di *log* le operazioni che sono state effettuate.

In caso di caduta di tensione durante la scrittura del disco rigido, al riavvio del sistema operativo il file system non dovrà far altro che analizzare il file di log per determinare quali sono le operazioni che non sono state terminate e quindi sarà in grado di correggere gli errori presenti nella struttura del file system.

Poiché nel file di log vengono memorizzate solo le informazioni che riguardano la struttura del disco (metadati), un'eventuale caduta di tensione elimina i dati che si stavano salvando, ma non rende incoerente il file system.

Dal punto di vista del programmatore, il tipo del filesystem definisce la codifica dei dati. ci sono diverse codifiche, come ad esempio: NTFS, MSDOS, FAT32, FAT64. FAT (16, 32, 64bit) e NTFS possono essere montati su di un fs Linux.

mount è il comando che ci permette di montare un fs e visualizzare i fs montati.

Mount inoltre ci permette di visualizzare i fs correntemente montati:

- `cat/etc/mtab` visualizza fs montati
- `cat/etc/fstab` visualizza fs montati all'avvio (bootstrap)

Cat invece, ci permette di concatenare più files e stamparne il contenuto su standard output (stdout). La forma più semplice:

- `cat nomefile`

Le tipiche directory di primo livello di un sistema *Linux*, sono le seguenti:

Table 2: Le tipiche directory di primo livello di un sistema Linux

Directory	Spiegazione	Montata
/boot	Kernel e file di boot	NO
/bin	Binari (programmi eseguibili) di base	NO
/dev	Devices (periferiche) hardware e virtuali	boot
/etc	File di configurazione di sistema	NO
/proc	Dati e statistiche dei processi e parametri del kernel	boot
/sys	Informazioni e statistiche di device di sistema	boot
/media	Mountpoint per device di I/O (es: CD, DVD, USB pen)	quando necessario
/mnt	(come /media)	quando necessario
/sbin	Binari di sistema	NO
/var	File variabili (log file, code di stampa, mail ...)	NO
/tmp	File temporanei	NO
/lib	Librerie	NO

1.5 File passwd e group

il path `/etc/passwd` contiene tutti gli utenti che sono presenti nel sistema, mentre il path `/etc/group` contiene tutti i gruppi presenti nel sistema.

Questi due tipi di file rappresentano una delle filosofie di Linux. Ovvero quella di usare file di testo (con codifica ASCII a 8 bit) con una struttura definita e conosciuta dai programmi che devono interagire con quei file.

1.8.1 Struttura

Questi tipi di file sono organizzati per righe, una riga è una sequenza di caratteri terminata con line feed LF (0x0A) (serve per avanzare verso il basso alla riga successiva).

Analizziamo la struttura di entrambi:

- `passwd` ha la seguente struttura:
 - ▶ `username:password:uid:gid:gecos:homedir:shell`

```
studente@debian9:~$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

possiamo notare come al posto della password ci sia una X. Questo ovviamente serve per preservare i dati e quindi la password viene cifrata.

- group ha la seguente struttura:

▶groupname:password:groupID:listautenti

```
studente@debian9:~$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:
tty:x:5:
disk:x:6:
lp:x:7:
```

Gli utenti all'interno della lista degli utenti sono separati da una virgola. Come possiamo ben vedere dall'immagine, anche qui la password è sostituita da una X, sempre per il motivo analogo.

Nel caso dell'immagine sopra, i gruppi sono quelli usati dal sistema, quindi non c'è nessun utente che vi appartiene.

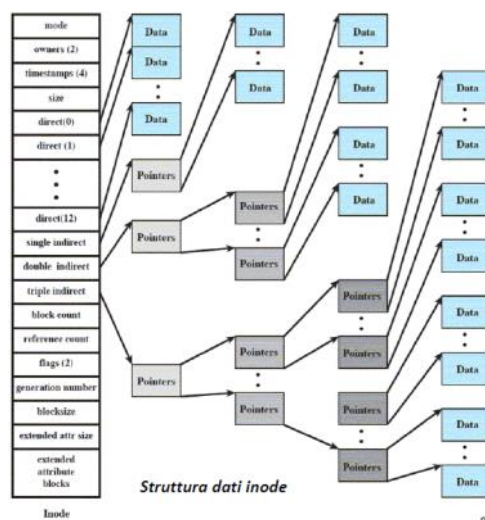
Solitamente se una riga inizia per # significa che contiene un commento o che il suo contenuto non deve essere interpretato dalle applicazioni che usano il file.

1.6 File

Ogni file nel filesystem è rappresentato da una struttura dati inode ed è univocamente identificato da un inode number. Il numero di inode si riferisce al file fisico, ovvero i dati memorizzati in una posizione particolare. Un file ha anche un device number, e la combinazione dell'inode number con il device number, è univoca in tutti i file system nel file system gerarchico.

La cancellazione di un file libera l'inode number che verrà riutilizzato quando necessario per un nuovo file.

Esempio di struttura dati inode:

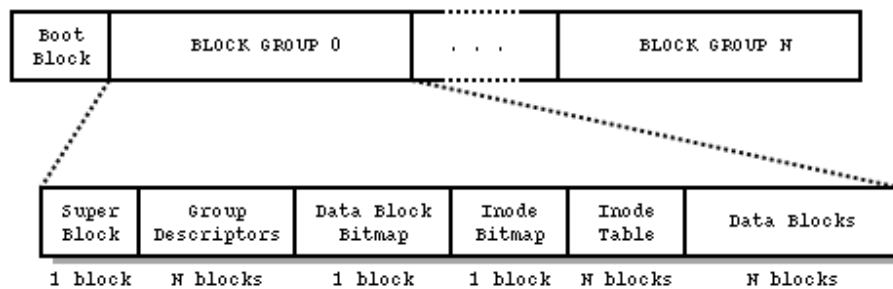


Ogni inode contiene informazioni sul file come:

- *Type*: Tipo di file (regular, block, fifo...)
- *User ID*: id dell'utente proprietario del file
- *Group ID*: id del gruppo a cui è associato il file
- *Mode*: Permessi(read, write, execute) di accesso per il proprietario, il gruppo e tutti gli altri
- *Size*: Dimensione in byte del file
- *Timestamps*: **ctime** (inode changing time: cambiamento di un attributo), **mtime** (content modification time: solo scrittura), **atime** (content access time: solo lettura)
- *Link count*: Numero di hard links (collegamenti "rigidi")
- *Data pointers*: Puntatore alla lista dei blocchi che compongono il file; se si tratta di una directory, il contenuto su disco è costituito da una tabella con 2 colonne: nome del file/directory e suo inode number

1.7 Struttura di un filesystem Ext2

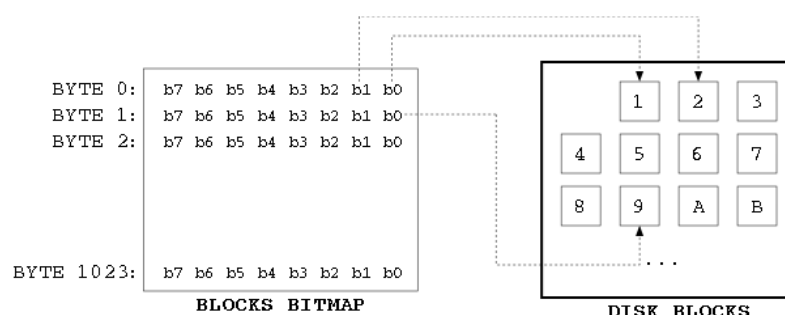
Su disco, il filesystem Ext2 è organizzato come mostrato nell'immagine seguente:



I primi 1024 byte del disco, il "boot block", sono riservati ai settori di boot partition e non vengono utilizzati dal filesystem Ext2. Il resto della partizione è suddiviso in gruppi di blocchi, ognuno dei quali ha il layout mostrato nella figura sopra. Su un floppy disk da 1,44 MB, c'è un solo gruppo di blocchi.

1.7.1 Blocchi e inodes bitmaps

Una bitmap è una sequenza di bit. Ogni bit rappresenta un blocco specifico (blocks bitmap) o un inode (inode bitmap) nel gruppo di blocchi. Un valore bit pari a 0 indica che un blocco/inode è libero, mentre un valore pari a 1 indica che il blocco/inode è in uso. Una bitmap fa sempre riferimento al gruppo di blocchi a cui appartiene e le sue dimensioni devono rientrare in un blocco.



Limitare la dimensione di una bitmap a un blocco limita anche la dimensione di un gruppo di blocchi, perché una bitmap fa sempre riferimento ai blocchi/inode nel gruppo a cui appartiene.

Consideriamo la bitmap dei blocchi:

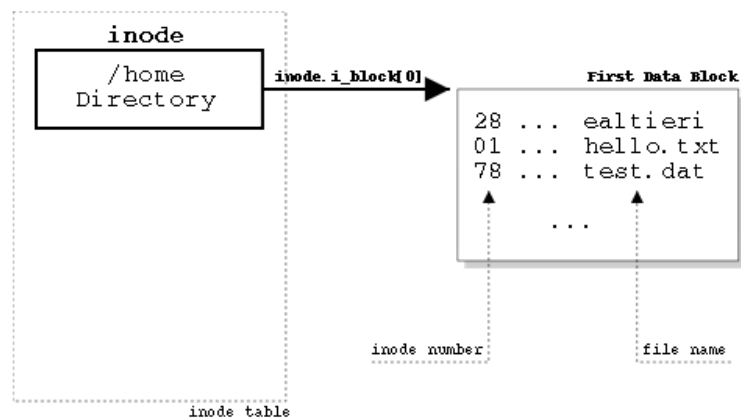
*data una dimensione del blocco di 1024 byte, e sapendo che ogni byte è composto da 8 bit, possiamo calcolare il numero massimo di blocchi che la bitmap dei blocchi può rappresentare: $8 * 1024 = 8192$ blocchi. Pertanto, 8192 blocchi è la dimensione di un gruppo di blocchi utilizzando una dimensione del blocco di 1024 byte.*

1.7.2 Directory entries nella tabella degli inode

Le directory entries nella tabella degli inode richiedono un'attenzione speciale. Per verificare se un inode fa riferimento a un file di directory, possiamo utilizzare la macro `S_ISDIR(mode)`:

```
if (S_ISDIR(inode.i_mode)) ...
```

Nel caso di directory entries, i blocchi di dati puntati da `i_block[]` contengono un elenco dei file che sono presenti all'interno della directory con i rispettivi numeri di inode.



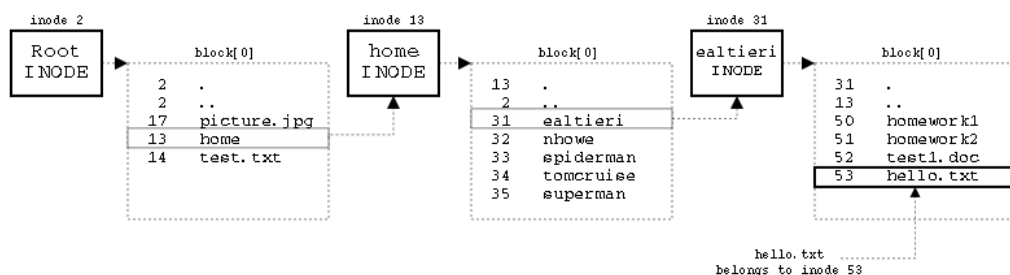
1.7.3 Individuazione di un file

Individuare i blocchi di dati appartenenti a un file implica prima individuare il suo inode nella tabella degli inode. L'inode del file desiderato non è generalmente noto al momento dell'esecuzione dell'operazione di apertura. Quello che sappiamo è il percorso del file. Per esempio:

```
int fd = open("/home/ealtieri/hello.txt", O_RDONLY);
```

Il file desiderato è `hello.txt`, mentre il suo percorso è `/home/ealtieri/hello.txt`. Per scoprire l'inode appartenente al file dobbiamo prima scendere lungo il suo path, partendo dalla directory root, fino a raggiungere la directory padre del file.

A questo punto possiamo individuare la voce `ext2_dir_entry_2` corrispondente a `hello.txt` e quindi il suo numero di inode.



Una volta che l'inode del file è noto, i blocchi di dati appartenenti a `hello.txt` sono specificati dall'array `inode.block[]`. La directory root si trova sempre nell'inode 2.

1.7.4 Come visualizzare le informazioni contenute nell'inode di un file

Il comando `ls` ha le seguenti opzioni:

- `ls [-a] [-c] [-u] [-R] [-l] [-i] [-n] [-S] [-h] [-1]`
`[nomedir1] ... [nomedirn] [nomefile1] ... [nomefilen]`

Per visualizzare l'inode number, basta usare l'opzione `-i`. Mentre con l'opzione `-l`, possiamo visualizzare: diritti, user, group, date, size, time.

```
studente@debian9:~$ ls -l .
totale 64
drwxr-xr-x 9 studente studente 4096 lug 31 2018 apache-tomcat-8.0.27
drwxr-xr-x 4 studente studente 4096 lug 31 2018 canopy
drwxr-xr-x 6 studente studente 4096 lug 31 2018 Canopy
drwxr-xr-x 4 studente studente 4096 mar 6 05:48 Documenti
drwxr-xr-x 2 studente studente 4096 lug 31 2018 entthought
-rw-r--r-- 1 studente studente 7 mar 6 06:33 filevuoto
```

come possiamo notare dall'immagine, abbiamo:

1. *Dimensione*: Ovvero quanto effettivamente occupato dal file
2. *numero di directory all'interno della directory* (vengono contate anche `.` e `..`), per i file è ovviamente 1
3. *Totale*: si tratta delle dimensione della directory in blocchi su disco (normalmente, 1 blocco ha dimensione tra 1kB e 4kB):

- riguarda solo la directory attuale, non tutto il sottoalbero
- per ogni sottodirectory, ne considera solo la dimensione, come definita sopra

Con l'opzione `-n` possiamo visualizzare l'ID utente e l'ID del gruppo. Per vedere i timestamp, dobbiamo aggiungere all'opzione `-l` anche:

- c per ctime, ad es. `ls -l -c nomefile` oppure `ls -lc nomefile`
- u per atime
- senza niente (per mtime)

Con il comando `stat filename` possiamo restituire varie informazioni. Invece digitando il comando `stat -c %B filename` possiamo vedere la dimensione del blocco su disco che coincide con la dimensione di un settore di disco.

```
studente@debian9:~$ stat filevuoto
  File: filevuoto
  Dim.: 17
Device: 801h/2040
Accesso: (0644/-rw-r--r--)  Uid: ( 1000/studente)   Gid: ( 1000/studente)
  Blocc: 8
  Blocco di IO: 4096
  file regolare
Accesso : 2020-03-07 22:36:52.880272001 +0100
Modifica : 2020-03-07 22:36:52.880272001 +0100
Cambio  : 2020-03-07 22:36:52.880272001 +0100
Creazione: -
studente@debian9:~$ stat -c %B filevuoto
512
```

2 Processi

In Unix/Linux le due entità fondamentali sono:

- **File:** descrivono/rappresentano risorse, contengono dati e programmi
- **Processi:** permettono di elaborare dati e usare le risorse di sistema

Un **processo** è un'entità dinamica caricata su memoria RAM generata da un programma: identificato da un codice univoco chiamato *PID*, più precisamente, esso è una sequenza di attività (*task*) controllata da un programma (scheduler) che si svolge su un processore in genere sotto la gestione o supervisione del rispettivo sistema operativo.

Per lanciare un processo si deve eseguire il file corrispondente e lo si può fare in due modi:

- Digitando il nome del file da shell come un normale comando
- Cliccare sull'icona corrispondente (ambienti grafici)

Esempi di processi sono quelli creati eseguendo i comandi visti precedentemente:

- `dd`, `ls`, `cat`, `cp`, `ln`, `adduser`, `groups`, `sudo`, `su`, ...

non tutti i comandi creano processi, ad esempio:

- `echo`, `cd`, ...

questi vengono eseguiti all'interno del processo di shell (la `bash` nel nostro caso)

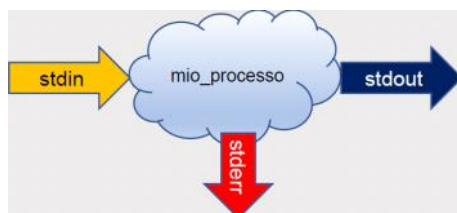
Un file eseguibile può essere eseguito più volte dando vita ogni volta ad un nuovo processo, e non occorre aspettare il termine dell'esecuzione prima di lanciarlo nuovamente.

Ad esempio, aprendo due shell e lanciando in entrambe il comando `cat/dev/random`, stiamo eseguendo due volte lo stesso processo.

2.1 Canali standard

Ogni processo Unix/Linux ha accesso ad almeno tre canali/file:

- Standard input (*stdin*) = Sta per input standard e viene utilizzato per prendere il testo come input.
- Standard output (*stdout*) = viene utilizzato per l'output di testo di qualsiasi comando digitato nel terminale, quindi tale output viene memorizzato nel flusso *stdout*.
- Standard error (*stderr*) = Viene invocato ogni volta che un comando incontra un errore, quindi quel messaggio di errore viene memorizzato in questo flusso di dati.



Va notato che in Linux tutti questi flussi vengono trattati come se fossero file. Inoltre, Linux assegna valori univoci a ciascuno di questi flussi di dati:

- 0 = stdin
- 1 = stdout
- 2 = stderr

Questi tre canali possono essere *redirezionati* indipendentemente.

2.1.1 Redirezione dei canali standard

Il lavoro di ogni comando è quello di prendere in input, dare in output oppure fare entrambe le cose. Quindi Linux ha dei comandi o caratteri speciali che servono per redirezionare queste funzionalità di input e output.

Ad esempio: supponiamo di voler eseguire un comando chiamato "date". Se lo eseguiamo stamperà l'output sullo schermo del terminale corrente. Ma il nostro requisito è diverso, non vogliamo che l'output venga visualizzato sul terminale, ma vogliamo che l'output venga salvato in un file.

Questo potrebbe essere fatto molto facilmente con il redirezionamento dell'output. Redirezionamento significa semplicemente deviare l'output o l'input.

I tipi di redirezione sono i seguenti:

1. Overwrite

- ">" standard output
- "<" standard input

2. Appends

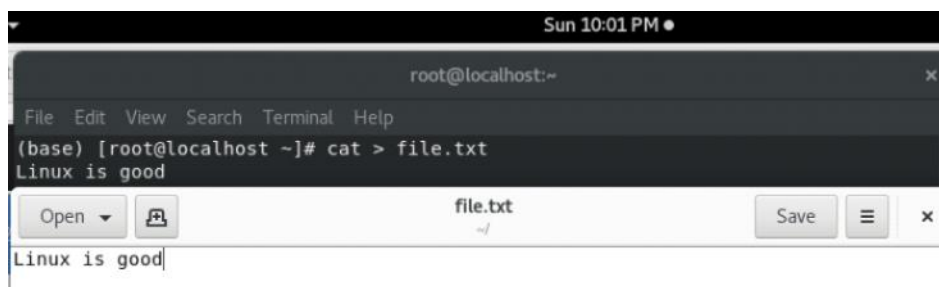
- ">>" standard output
- "<<" standard input

3. Merge

- "p>&q" Unisce l'output dello stream p con lo stream q
- "p<&q" Unisce l'input dello stream p con lo stream q

Qualunque cosa scriverai dopo aver eseguito questo comando, tutto verrà redirezionato e copiato nel "file.txt". Questo è il redirezionamento dell'output standard:

```
cat > file.txt
```



questo è il redirectionamento dell'input standard, il comando cat prenderà l'input da "file.txt" e lo stamperà sullo schermo del terminale. Questa riga di codice mostra anche il vero funzionamento e il significato del comando cat che è copia e incolla.

```
cat < file.txt
```

Molte persone hanno un'idea sbagliata che il comando cat sia usato per creare un file, ma non è vero, il lavoro principale è quello di copiare l'input e dare l'output a schermo.

Questo viene utilizzato quando vogliamo aggiungere alcune righe al contenuto esistente del file. Se si utilizza una sola parentesi angolare tutto il contenuto del file andrà perso.

```
cat >> file.txt
```

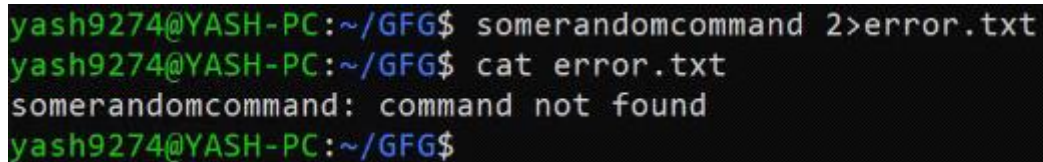
Il redirectionamento degli errori trasferisce gli errori generati da alcuni falsi comandi in un file anziché in STDOUT.

Ogni volta che un programma viene eseguito sul terminale, vengono generati 3 file: *standard input(0)*, *standard output(1)*, *standard error(2)*. Questi file vengono sempre creati ogni volta che viene eseguito un programma. Per impostazione predefinita, sullo schermo viene visualizzato un flusso di errori.

Esempi:

1. Nell'esempio riportato di seguito, il descrittore di file utilizzato sopra è 2(STDERR). L'utilizzo di "2>" reindirizza l'output dell'errore a un file denominato "error.txt" e non viene visualizzato nulla su STDOUT.

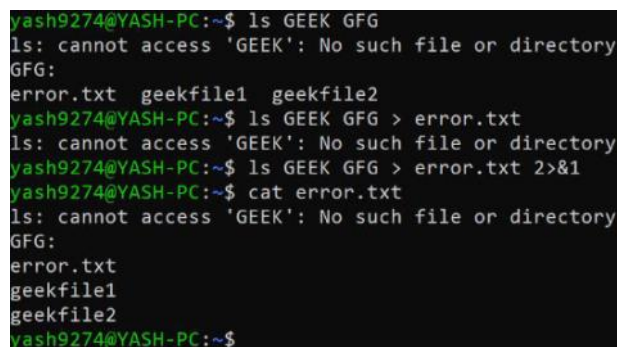
```
2>error.txt
```



```
yash9274@YASH-PC:~/GFG$ somerandomcommand 2>error.txt
yash9274@YASH-PC:~/GFG$ cat error.txt
somerandomcommand: command not found
yash9274@YASH-PC:~/GFG$
```

2. Qui, 2>&1 significa che STDERR redireziona alla destinazione di STDOUT. Più formalmente, il messaggio di errore generato da "2" viene unito all'output corrente "1".

```
ls GEEK GFG > error.txt 2>&1
```



```
yash9274@YASH-PC:~$ ls GEEK GFG
ls: cannot access 'GEEK': No such file or directory
GFG:
error.txt  geekfile1  geekfile2
yash9274@YASH-PC:~$ ls GEEK GFG > error.txt
ls: cannot access 'GEEK': No such file or directory
yash9274@YASH-PC:~$ ls GEEK GFG > error.txt 2>&1
yash9274@YASH-PC:~$ cat error.txt
ls: cannot access 'GEEK': No such file or directory
GFG:
error.txt
geekfile1
geekfile2
yash9274@YASH-PC:~$
```

Nell'esempio precedente, la directory GEEK non è presente. L'output dell'errore viene unito allo standard output che a sua volta viene redirezionato a "error.txt".

2.2 Rappresentazione dei processi

Il sistema fornisce un'istanza speciale ogni volta che viene eseguito un programma/comando. Questa istanza è costituita da tutti i servizi/risorse che possono essere utilizzati dal processo in esecuzione.

Un processo ha un suo PID, ovvero un process IDentifier, un Process control block(PCB) e viene coinvolto in sei aree di memoria che vedremo successivamente.

2.2.1 PID - Process IDentifier

Il *PID* è un identificatore (numero intero) univoco di un processo. In un dato istante, non ci possono essere 3 processi con lo stesso PID. Una volta che un processo è terminato, il suo PID viene liberato, e potrebbe essere prima o poi riusato per un altro processo.

Alcuni sistemi, Linux incluso, assegnano PID casuali ai processi per incrementare la sicurezza. (Ad esempio: in un sistema parzialmente compromesso, un attacco al processo web server dovrebbe prima di tutto scoprire il PID di tale processo.).

Nell'ultima decade aumento dell'uso della casualità (randomness) nei sistemi operativi per ridurre (mitigate) la superficie d'attacco.

2.2.2 Process Control Block

Il *process control block* o *PCB* è la struttura dati di un processo, del nucleo del sistema operativo, che contiene le informazioni essenziali per la gestione del processo stesso.

Le informazioni contenute variano a seconda delle implementazioni, ma in generale sono presenti:

- **PID:** Process Identifier
- **PPID:** Parent Process Identifier
- **Real UID:** Real User Identifier
- **Real GID:** Real Group ID
- **Effective UID:** Effective User Identifier (UID assunto dal processo in esecuzione)
- **Effective GID:** Effective Group ID (come sopra per GID)
- **Saved UID:** Saved User Identifier (UID avuto prima dell'esecuzione del SetUID)
- **Saved GID:** Saved Group Identifier (come sopra per GID)
- **Current Working Directory:** directory di lavoro corrente
- **Umask:** file mode creation mask
- **Nice:** priorità statica del processo

2.2.3 Aree di memoria

Ogni processo ha un suo spazio di indirizzamento privato e non visibile dall'esterno. Ciò vuol dire che due processi non possono accedere a una stessa zona di memoria.

Questa separazione totale degli spazi di indirizzamento è ottenuta sfruttando le caratteristiche hardware del processore (segmentazione, protezione di memoria, ecc.).

Lo spazio di indirizzamento di un process è di solito organizzato come segue:

Text Segment: istruzioni da eseguire(in linguaggio macchina)

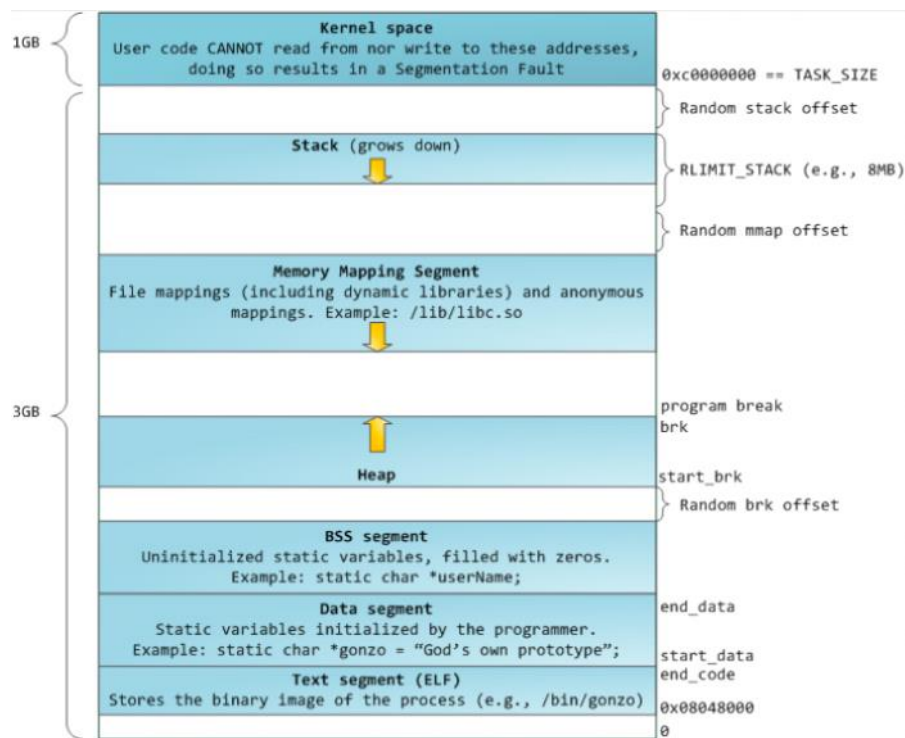
Data Segment: dati statici(ovvero, variabili globali e variabili locali static) iniziati e alcune costanti di ambiente

BSS: dati statici non iniziati (sta per block started from symbol); la distinzione dal segmento dati si fa per motivi di realizzazione hardware

Heap: dati dinamici(allocati con malloc e simili)

Stack: per le chiamate a funzioni, con i corrispondenti dati dinamici(parametri, variabili locali non static)

Memory Mapping Segment: tutto ciò che riguarda librerie esterne dinamiche usate dal processo, nonché estensione dello heap in alcuni casi



Le aree di memoria che possono essere condivise tra i processi sono il text segment oppure il BSS, il data segment o l'MMS.

Nota: lo stack NON è mai condiviso

2.3 Stati di un processo

In Linux, un processo è un'istanza di esecuzione di un programma o di un comando. Ogni istanza di esecuzione di un programma si può trovare in uno dei sette stati seguenti:

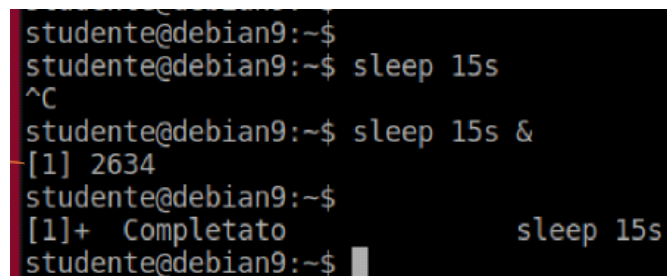
- **Running (R)**: in esecuzione su un processore.
- **Runnable (R)**: pronto per essere eseguito (non è in attesa di alcun evento); in attesa che lo scheduler lo (ri)selezioni per l'esecuzione.
- **(Interruptible) Sleep (S)**: è in attesa di un qualche evento (ad esempio, lettura di blocchi dal disco), e non può quindi essere scelto dallo scheduler.
- **Zombie (Z)**: il processo è terminato e le sue 6 aree di memoria non sono più in memoria; tuttavia, il suo PCB viene ancora mantenuto dal kernel perché il processo padre non ha ancora richiesto il suo "exit status".
- **Stopped (T)**: caso particolare di sleep: avendo ricevuto un segnale STOP, è in attesa di un segnale CONT.)
- **Traced (t)**: in esecuzione di debug, oppure in generale in attesa di un segnale (altro caso particolare di sleep).
- **Uninterruptible sleep (D)**: come sleep, ma tipicamente sta facendo operazioni di I/O su dischi lenti e non può essere interrotto o ucciso.

2.4 Modalità di Esecuzione dei processi

I processi che richiedono che un utente li avvii o interagisca con essi sono chiamati *foreground processes*. Mentre, i processi che vengono eseguiti indipendentemente da un utente vengono definiti *background processes*.

Programmi e comandi vengono eseguiti come processi in foreground per impostazione predefinita. Per eseguire un processo in background, bisogna inserire una *e commerciale* (&) alla fine del nome del comando utilizzato per avviare il processo.

Il comando `sleep` mette in pausa per un tempo specificato nell'argomento:



```
studente@debian9:~$  
studente@debian9:~$ sleep 15s  
^C  
studente@debian9:~$ sleep 15s &  
[1] 2634  
studente@debian9:~$  
[1]+  Completato sleep 15s  
studente@debian9:~$
```

Se vogliamo vedere la lista dei job in esecuzione ci basta eseguire:

```
jobs [-l] [-p]
```

Per portare un processo in background invece, usiamo il comando `bg` che possiamo interrompere con `CTRL+Z` e risvegliare con `bg`.

Con il comando `fg%n` dove `%n` è il numero del job, lo riporta in foreground. Mentre con `bg%n` porta in background il processo `%n`.

Si possono identificare dei job anche con:

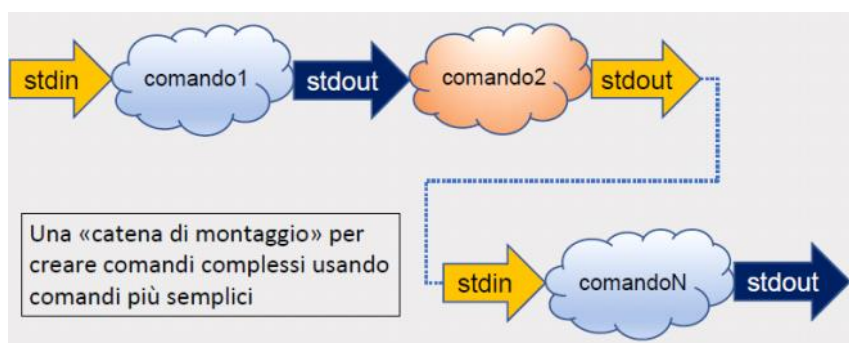
`%prefix`: dove `prefix` è la parte iniziale del comando del job desiderato

`%+` oppure `%%`: l'ultimo job mandato

`%-`: il penultimo job mandato

2.4.1 Pipelining dei comandi

Pipe è un comando in Linux che consente di utilizzare due o più comandi in modo tale che l'output di un comando funga da input per il successivo. Il simbolo `|` indica una pipe.



Per eseguire un job composto da più comandi, scriviamo:

```
comando1 | comando2 | ... comando n
```

Lo standard output di un comando i diventa l'input del comando $i + 1$.

Se usiamo `"| &"` stiamo ridirezionando lo standard error sullo standard input del comando successivo. In questo caso, il comando $i + 1$ non deve necessariamente usare l'output/stderr del comando i .

2.4.2 Comandi utili per i processi

Prima di tutto abbiamo il comando **ps**:

```
ps[opzioni] [pid...]
```

che ci permette di visualizzare le informazioni sui processi in esecuzione. `ps` senza argomenti, mostra i processi dell'utente attuale lanciati dalla shell corrente. Per ogni processo mostra: PID, TTY, TIME e CMD.

```
studente@debian9: ~
File Modifica Schede Aiuto
niente /dev/null
[3]+ Ucciso      yes niente /dev/null
studente@debian9:~$ ps
  PID TTY          TIME CMD
 1970 pts/0    00:00:00 bash
 3352 pts/0    00:00:00 ps
studente@debian9:~$ sleep 60 &
[1] 3357
studente@debian9:~$ sleep 80 &
[2] 3358
studente@debian9:~$ yes niente > /dev/null &
[3] 3360
studente@debian9:~$ yes altroniente > /dev/null &
[4] 3362
studente@debian9:~$ ps
  PID TTY          TIME CMD
 1970 pts/0    00:00:00 bash
 3357 pts/0    00:00:00 sleep
 3358 pts/0    00:00:00 sleep
 3360 pts/0    00:00:05 yes
 3362 pts/0    00:00:00 yes
 3364 pts/0    00:00:00 ps
studente@debian9:~$

studente@debian9:~$ ps -p 3357
  PID TTY          TIME CMD
 3357 pts/0    00:00:00 sleep
studente@debian9:~$ ps -t 3357
  PID TTY          TIME CMD
 3357 pts/0    00:00:00 sleep
studente@debian9:~$
```

Le opzioni che ci offre il comando `ps`, sono le seguenti:

- `-e`: tutti i processi di tutti gli utenti lanciati da tutte le shell o al boot (figli del processo 0)
- `-u{utente, }`: tutti i processi degli utenti nella lista
‣ Es: `ps -u studente`
- `-p {pid, }`: tutti i processi con i PID nella lista
- `-o {field, }`: per scegliere i campi da visualizzare; nel man ci sono gli acronimi

```
studente@debian9:~$ ps
  PID TTY          TIME CMD
 1970 pts/0    00:00:00 bash
 3582 pts/0    00:00:00 ps
studente@debian9:~$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
studente    1970   1968  0 mar20 pts/0    00:00:00 bash
studente    3584   1970  0 00:27 pts/0    00:00:00 ps -f
studente@debian9:~$ ps -l
 F S  UID      PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
 0 S  1000    1970   1968  0  80   0 -  5501 -          pts/0    00:00:00 bash
 0 R  1000    3586   1970  0  80   0 -  7467 -          pts/0    00:00:00 ps
studente@debian9:~$
```

I significati dei campi mostrati da `ps` sono i seguenti:

- **PPID**: parent pid, pid del processo che ha creato questo processo
- **C**: parte intera della percentuale di uso della CPU
- **STIME** (o **START**): l'ora in cui è stato invocato il comando, oppure la data, se è stato fatto partire da più di un giorno
- **TIME**: tempo di CPU usato finora
- **CMD**: comando con argomenti
- **F**: flags associati al processo: 1 il processo è stato “biforcato”, ma ancora non eseguito; 4, ha usato privilegi da superutente; 5, entrambi i precedenti; 0, nessuno dei precedenti
‣ `-y -l` elimina questo campo che è poco utile
- **S**: stato(modalità) del processo in una sola lettera
- **UID**: utente che ha lanciato il processo(se SetUID presente potrebbe non essere chi ha dato il comando)
- **PRI**: attuale priorità del processo (più il numero è alto, minore è la priorità)
- **NI**: valore di nice, da aggiungere alla priorità (vedere più avanti)
- **ADDR**: indirizzo in memoria del processo, ma è mostrato (senza valore) solo per compatibilità all'indietro
‣ `-y -l` toglie questo campo e lo sostituisce con **RSS** (resident set size) dimensione del processo in memoria principale in KB (non tiene conto delle pagine su disco)
- **SZ**: dimensione totale attuale del processo in numero di pagine (tutte le 6 aree di memoria del processo) RAM + disco (memoria virtuale)
- **WCHAN**: se il processo è in attesa di un qualche segnale o comunque in sleep, qui c'è la funzione del kernel all'interno della quale si è fermato

Poi abbiamo il comando **top**, la cui sintassi è la seguente:

```
top [-b] [-n num] [-p {pid,}]
```

questo comando lo si può vedere come un **ps** interattivo (provare a digitarlo nella shell) e ha le seguenti caratteristiche:

- **-b**: non accetta più comandi interattivi, ma aggiorna i dati ogni pochi secondi
- **-n**: num fa solo num aggiornamenti
- **-p**: come in **ps**

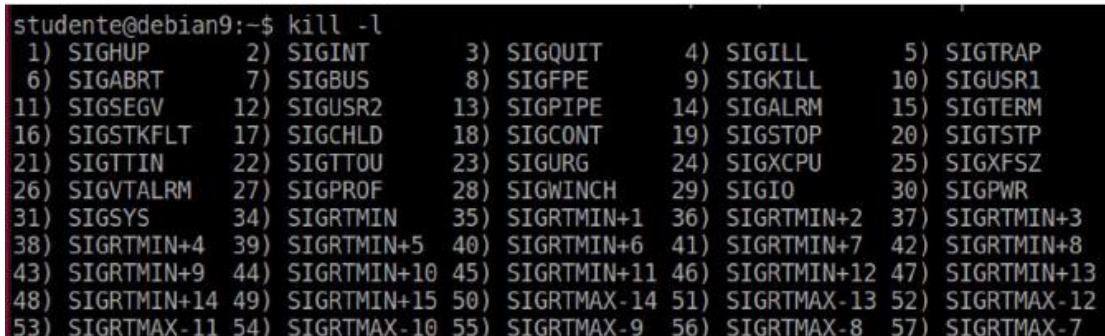
Una volta aperto in modo interattivo, basta premere **?** per avere la lista dei comandi accettati (tipicamente, sono singole lettere)

Un altro comando molto importante è **kill**:

```
kill [-l [signal]] [-signal] [pid...]
```

questo comando invia segnali ad un processo (non solo la terminazione). Con l'opzione **-l** possiamo visualizzare la lista dei segnali; un segnale **signal** è identificato dal numero oppure dal nome con **SIG** o senza **SIG**. Ad esempio:

```
kill -9 pid = kill -s SIGKILL pid = kill -s KILL pid
```



```
studente@debian9:~$ kill -l
 1) SIGHUP    2) SIGINT    3) SIGQUIT   4) SIGILL    5) SIGTRAP
 6) SIGABRT   7) SIGBUS    8) SIGFPE    9) SIGKILL   10) SIGUSR1
11) SIGSEGV  12) SIGUSR2  13) SIGPIPE  14) SIGALRM  15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD  18) SIGCONT  19) SIGSTOP  20) SIGTSTP
21) SIGTTIN  22) SIGTTOU  23) SIGURG   24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF  28) SIGWINCH 29) SIGIO    30) SIGPWR
31) SIGSYS    34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
```

I segnali vengono presi in considerazione solo se il *real user* del processo è lo stesso che invia il segnale (oppure, se lo invia un superuser). Un processo che riceve un segnale fa un'azione predefinita (vedere `man 7 signal`) oppure un'azione personalizzata.

Alcuni tipi di segnali sono:

- **SIGSTOP**, **SIGSTP**: per la sospensione di un processo(CTRL+Z invia un **SIGSTOP**)
- **SIGCONT**: per la continuazione di processi bloccati(con **bg** si invia **SIGCONT** al job indicato)
- **SIGKILL**, **SIGINT**: per la terminazione dei processi(CTRL+C invia un **SIGINT**)

Poi ci sono anche i segnali **SIGUSR1** e **SIGUSR2** che sono impostati per essere utilizzati dall'utente per le proprie necessità.

Con **kill** possiamo usare la notazione **%** (usata in **bg** e **fg**) per indicare job destinatari del messaggio.

Come abbiamo già accennato prima, abbiamo anche il comando **nice**:

```
nice [-n num] [comando]
```


`nice` senza opzioni dice quant'è il *niceness* di partenza. Il *niceness* può essere pensato come una addizione sulla priorità: se positivo, ne aumenta il valore (quindi la priorità decresce), altrimenti ne diminuisce il valore (la priorità cresce). Questo valore va da -19 a +20, con valore di default che è uguale a 0.

digitando:

```
nice [-n num] comando
```

lancia il comando (che potrebbe avere delle sue opzioni, più o meno come con `sudo`) con *niceness num* (0 se non specificato).

Il comando:

```
renice priority {pid}
```

Interviene su processi già in esecuzione infatti, richiede dei PID (ci sarebbero anche altre opzioni per fargli avere effetti sui processi di un dato utente/gruppo)

L'ultimo comando che vediamo è **strace**:

```
strace [-p pid] [comando]
```

`strace` lancia comando, visualizzando tutte le sue chiamate di sistema, oppure, visualizza le chiamate di sistema del processo `pid`.

Con l'opzione `-o filename` ridireziona l'output su di un file. Questo comando ci sarà utile per il debugging di programmi che utilizzano chiamate di sistema.