

Sistemi Operativi 2

Marco Casu



Contents

1	Introduzione	3
1.1	Breve Panoramica su Unix	3
1.2	La Shell	3
2	Il File System	5
2.1	Operazioni sulle Directory	5
2.1.1	Comandi di Base	6
2.2	Struttura del File System	7
2.2.1	Mounting e Partizioni	7
2.2.2	Inode e Metadati	8
2.3	Permessi di Accesso	9
3	I Processi in Unix	11
3.1	Canali di Input/Output	11
3.2	Struttura di un Processo	11
3.3	Pipelining e Segnali	13
4	Il Linguaggio C	14
4.1	Sintassi	15

1 Introduzione

1.1 Breve Panoramica su Unix

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

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

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

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

1.2 La Shell

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

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

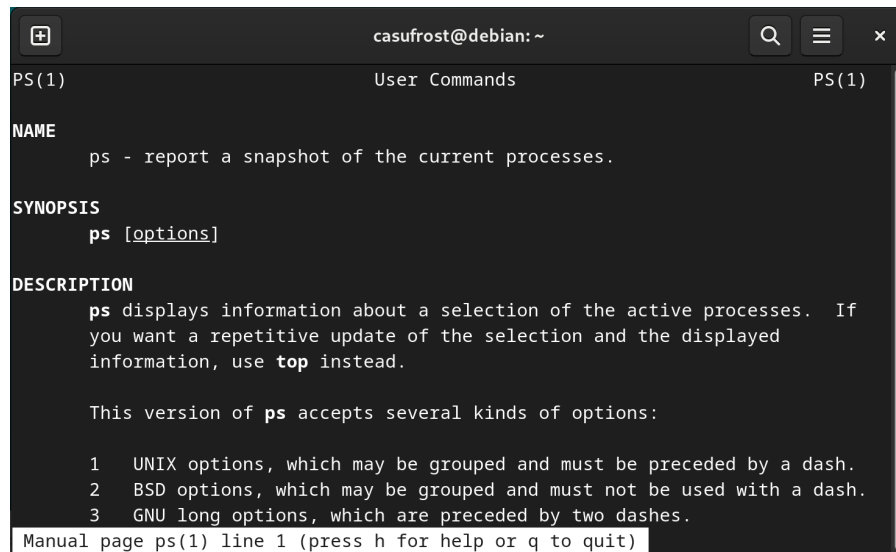
in attesa di ricevere comandi

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

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

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

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



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

NAME
    ps - report a snapshot of the current processes.

SYNOPSIS
    ps [options]

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

    This version of ps accepts several kinds of options:

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

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

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

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

Analizziamo il comando di esempio scritto in precedenza, ossia:

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

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

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

Un altro esempio :

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

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

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

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

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

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

2 Il File System

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

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

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

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

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

2.1 Operazioni sulle Directory

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

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

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

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

2.1.1 Comandi di Base

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

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

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

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

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

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

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

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

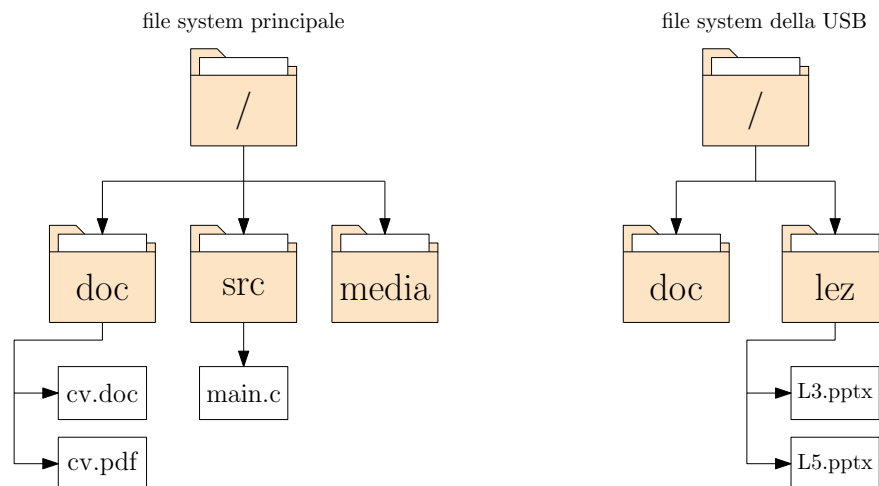
Esistono altri comandi secondari piuttosto utili : `tree` mostra l'alberazione della directory, `touch` crea un file vuoto, `pwd` stampa a schermo la directory corrente.

2.2 Struttura del File System

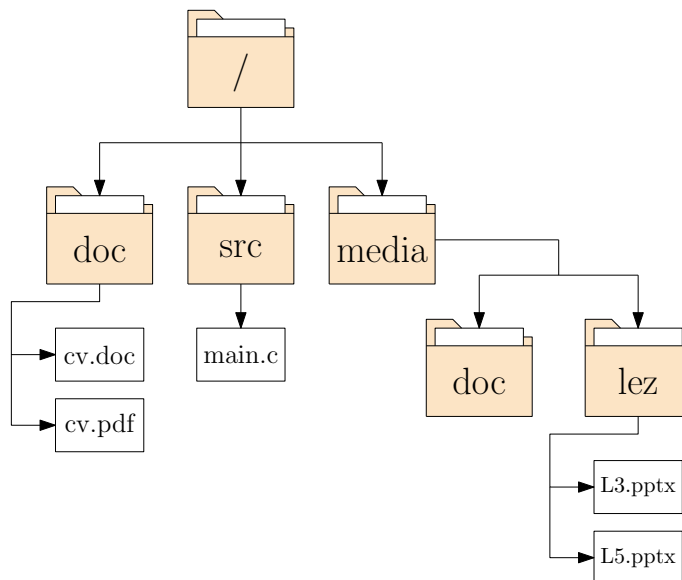
2.2.1 Mounting e Partizioni

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

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



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



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

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

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

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

Diversi file system differiscono in caratteristiche quali :

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

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

2.2.2 Inode e Metadati

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

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

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

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

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

2.3 Permessi di Accesso

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

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

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

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

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

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

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

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

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

È possibile modificare i permessi di accesso ad un file tramite il comando `chmod`, specificando prima la tripla di ottali che identificherà i nuovi permessi, e poi il file in questione, è possibile anche cambiare il proprietario ed il gruppo di un file tramite `chown` e `chgrp`, se abbinati con l'opzione `-r`, tutti i sottoalberi dell'argomento riceveranno lo stesso trattamento.

Un importante valore di un sistema operativo Unix è la *user mask*, ossia una parola di bit che identifica i permessi di default che vengono assegnati ad un file nel momento della sua creazione. La user mask è composta da quattro terne di 3 bit (rappresentanti i permessi ed i permessi speciali), e può essere modificata con il comando `umask` inserendo come parametro la nuova maschera.

I permessi di default di ogni file sono calcolati nel seguente modo :

```
111 111 111 111 AND NOT({user mask})
```

Ad esempio (escludendo i bit speciali) per impostare i diritti di default a 664, bisognerà formare il seguente comando :

```
umask 113
```

```
111 111 111 AND NOT(001 001 011) = 111 111 111 AND 110 110 100 = 110 110 100 = 664
```

Altri comandi fondamentali per manipolare i file sono

- `cp` : Permette di copiare un file in una destinazione specificata
- `mv` : Permette di spostare un file in una destinazione specificata
- `rm` : Elimina il file specificato

Abbiamo poi visto che è possibile creare dei *link o collegamenti* fra file, un *soft link*, sarà un file che conterrà il puntatore ad un altro file, un *hard link*, sarà invece un altro file, ma con il riferimento allo stesso inode number, saranno quindi due file distinti, senza che però il disco sia occupato due volte.

È possibile creare dei link con il comando `ln`, l'eliminazione del file originale nega l'accesso a tutti i soft link ad esso collegati, per eliminare un file di cui invece esistono più hard link, bisogna eliminarli tutti.

Passiamo ora ad un comando particolare, ossia `dd`, tale comando è presente anche in delle primordiali versioni di Unix, e serve a copiare i file in maniera più sofisticata, opzionalmente effettuando conversioni, i suoi parametri assumono la sintassi `variabile=valore`, diversamente dal simbolo `-` utilizzato per gli altri comandi.

Può leggere e copiare i file *byte per byte*, specificando il numero preciso di blocchi da copiare, la dimensione di tali blocchi, e dove scriverli all'interno del file di destinazione, a tal scopo ci sono le opzioni `count,bs,skip,of,if,seek`, ad esempio :

- `skip=x` - Del file da copiare, verranno ignorati i primi *x* blocchi.
- `seek=x` - Nel file di destinazione, il contenuto verrà copiato a partire dal *x*-esimo blocco.

3 I Processi in Unix

Abbiamo già accennato al fatto che in Unix ogni entità può essere vista come un file, le due componenti fondamentali di un sistema di questo tipo però, non sono esclusivamente i file, ma anche i *processi*. Un processo rappresenta l'istanza di un programma/file eseguibile che viene avviato e caricato sulla memoria principale, una volta avviato, l'OS si occuperà di tale processo, come i file, ogni processo è identificato da un numero intero univoco.

Per avviare un programma, e renderlo un processo, bisogna digitare sulla shell il codice del programma da eseguire, quando vengono richiamati i comandi `dd,ls,cat` e `cp`, vengono eseguiti dei nuovi processi, diversamente, i programmi `echo` e `cd`, che sono comandi interni, fanno parte del processo shell.

Uno stesso programma può essere istanziato più volte, dando vita a più processi copia di uno stesso programma che girano sulla macchina.

3.1 Canali di Input/Output

Ogni processo di Unix ha accesso a 3 canali/file di default, utili per il flusso di dati in input ed output, questi sono :

- `stdin` (*standard input*) : Per il flusso di dati in input, ha il codice descrittore uguale a 0, e di default è impostata la tastiera.
- `stdout` (*standard output*) : Per il flusso di dati in output, ha il codice descrittore uguale a 1, e di default è impostato lo schermo.
- `stderr` (*standard error*) : Secondo canale di output utile per eventuali errori e diagnostica, ha il codice descrittore uguale a 2, e di default è impostato lo schermo.

I 3 canali possono essere "ridirezionati" dall'utente in maniera indipendente, ad esempio, è possibile cambiare il flusso di dati in output, facendo sì che `stdout` venga impostato su un file, tutti i dati in output di un programma potranno quindi essere scritti su un file specificato, vale lo stesso principio per la lettura con `stdin`.

A tale scopo, quando si avviano i programmi dalla shell, bisogna utilizzare i simboli `<` e `>` per ridirezionare rispettivamente l'input e l'output, ad esempio :

```
casufrost@debian:~$ ls > dirlist
```

 : scriverà il risultato del comando sul file `dirlist`.

Il precedente citato *descrittore di un file* è un identificatore univoco per ogni file, similmente all'inode, ma di più alto livello.

3.2 Struttura di un Processo

Ogni processo è identificato dal sistema con un codice univoco, un numero intero detto *PID*. Oltre ciò, ogni processo ha un *Process Control Block* (PCB), e 6 aree di memoria dedicata.

Il PID 0 è assegnato ad un processo chiamato `init`, è il primo processo avviato dalla macchina e si occupa di avviare tutti gli altri processi. Quando un processo termina, il suo PID viene

liberato, alcuni sistemi per ragioni di sicurezza assegnano i PID in maniera casuale. Il PCB è una struttura dati contenente dei valori che forniscono informazioni su ogni processo, contiene:

- *PID* - Il suo codice identificativo
- *PPID* - Il PID del suo processo padre
- *Real UID* - Id dell'utente proprietario
- *Real GID* - Id del gruppo dell'utente proprietario
- *Effective UID* - Id effettivo dell'utente assunto
- *Effective GID* - Id effettivo del gruppo dell'utente assunto
- *Saved UID* - UID avuto prima dell'esecuzione del SetUID
- *Saved GID* - Analogo al precedente ma per il gruppo
- *Current Working Directory* - La directory dalla quale è stato avviato il processo

Il *Saved UID* permette ad un processo in esecuzione come root, di eseguire delle operazioni in veste di utente non privilegiato, è importante ridurre al minimo (per ragioni di sicurezza) l'utilizzo dei privilegi di root. All'avvio di ogni processo, vengono riservate le 6 aree di memoria in seguito :

1. *Text Segment* - Le istruzioni sequenziali del programma in linguaggio macchina.
2. *Data Segment* - I dati statici del programma inizializzati caricato, come le variabili globali.
3. *BSS* - I dati statici del programma non inizializzati, distinto dal Data Segment per motivi di realizzazione hardware.
4. *Heap* - I dati dinamici generati dal processo, ad esempio, la memoria che si riserva con la funzione `malloc`.
5. *Stack* - Pila necessaria per il funzionamento delle chiamate di funzione, con i corrispondenti dati dinamici, come i parametri passati.
6. *Memory Mapping Segment* - Tutto ciò che riguarda le librerie esterne necessarie al processo.

Per comunicare, più processi potrebbero condividere alcune aree di memoria, come l'heap, oppure se sono caricati due processi istanza dello stesso programma, non è necessario avere due distinti text segment, in quanto condividono le stesse istruzioni. Un processo in memoria, può trovarsi in uno dei seguenti stati :

- **Running** - Il processo è in esecuzione sulla CPU.
- **Runnable** - Il processo è pronto per essere eseguito, ed è in una coda in attesa di essere selezionato dallo scheduler.
- **Sleeping** - Il processo non può essere eseguito, in quanto è in atteso di un certo evento, come un segnale di Input da parte dell'utente.

- **Zombie** - Il processo è stato eliminato e le sue aree di memoria liberate, tuttavia il suo PCB è ancora mantenuto, in quanto il processo padre che l'ha generato non ha richiesto che il processo figlio venisse abortito.
- **Stopped** - Non è altro che un particolare caso dello stato *Sleeping*, anche se il processo potrebbe continuare la sua esecuzione, esso è fermo in quanto ha ricevuto un segnale esterno (dall'utente o da un processo) di fermarsi.
- **Traced** - in esecuzione di debug, oppure in generale in attesa di un segnale (altro caso particolare di sleep, verrà trattato in seguito).
- **Uninterruptible Sleep** - Un caso di *Sleeping* in cui il processo sta facendo operazioni di I/O molto lunghe su un disco molto lento, facendo sì che si trovi in uno stato di attesa appartenentemente interminabile.

Un processo, può essere eseguito in **foreground**, ossia sulla shell dalla quale è stato avviato, prendendone il controllo, e solamente una volta che terminerà la sua esecuzione l'utente potrà nuovamente interagire con la shell, oppure in **background**, lasciando la shell libera all'utente, e venendo eseguito comunque, a patto che non possa ricevere input da tastiera grazie alla shell, riducendone l'interattività. Si può richiedere ad un comando di venire eseguito in background con il simbolo `&` :

```
casufrost@debian:~$ ls / -l &
```

La lista di tutti i processi in esecuzione può essere visualizzata con il comando `job`, inoltre, è possibile spostare un processo da uno stato di background ad uno stato di foreground con il comando `fg`, oppure fare l'operazione analoga ma inversa con il comando `bg`, specificando il PID del processo in questione.

3.3 Pipelining e Segnali

L'idea del *pipelining* è quella di creare programmi complessi, concatenando gli output e gli input di piccoli programmi più semplici (come appunto è nella filosofia di Unix). Il simbolo `|` fra due comandi Unix permette di concatenare l'Output del comando alla sua sinistra con l'input del comando alla sua destra, ad esempio `comando1 | comando2` farà sì che l'input di `comando2` sarà l'output di `comando1`.

Le informazioni relative ai processi, quali PID, nome comando, tempo di esecuzione ed altre possono essere visualizzate con il comando `ps`, che sta per *processes*, con l'opzione `-o` è possibile scegliere quali dati visualizzare. Sono particolarmente utili le opzioni `-a` : visualizza tutti i processi, e `-x` : visualizza i processi che non hanno un terminale.

Il comando `top` è una versione "interattiva" del comando `ps`, prende il controllo della shell e funziona in tempo reale, mostrando i processi che eventualmente vengono creati o terminati.

Il comando `kill -[codice] %PID%` serve per inviare segnali ai processi, prende in ingresso il codice di uno specifico segnale ed il PID di un processo, ad esempio :

```
casufrost@debian:~$ kill -9 4562
```

Con il comando `kill -l` è possibile visualizzare tutti i segnali che possono essere inviati, i più comuni sono:

- `SIGSTP` - Per la sospensione di un processo
- `SIGCONT` - Per la continuazione di un processo
- `SIGKILL` o `SIGINT` - Per la terminazione di un processo.

Il comando `nice` serve ad assegnare ad un processo un certo grado di priorità, che lo avvantaggerà nell'essere selezionato dallo scheduler, il range della priorità va da -19 a +20.

4 Il Linguaggio C

Il linguaggio C è uno dei più antichi linguaggi mai sviluppati, è stato creato dagli autori di Unix negli anni 70 ed è ancora in uso. Il C è nato per poter scrivere il kernel di Unix in un linguaggio di più alto livello, in modo da renderlo portatile su differenti architetture, è un linguaggio *indipendente* dalla macchina e può essere eseguito su diversi sistemi con poche modifiche nel codice (previa eventuale ricompilazione).

Il *workflow* per la scrittura di un programma in C è la seguente :

1. Si scrive il programma su un editor di testo.
2. Si chiama il preprocessore, che si occuperà di "semplificare" il codice.
3. Si chiama il compilatore, che dal file sorgente genererà le istruzioni in linguaggio macchina.
4. Il Linker collegherà l'assemblatore con le eventuali librerie esterne.

I punti (2) e (4) sono spesso gestiti dal compilatore, quindi sarà necessario scrivere il programma, compilarlo per poi eseguirlo. Verrà utilizzato il compilatore open-source della GNU, ossia *gcc*.

Sappiamo che un codice scritto in Java viene compilato in bytecode, per poi venire interpretato dalla *Java Virtual Machine*, e non direttamente dal processore. La compilazione di un programma C invece, genera un codice del tutto eseguibile, in quanto sono istruzioni interpretabili direttamente dal processore, rendendolo quindi più efficiente. Necessariamente, un file eseguibile scritto in C, deve avere i permessi di esecuzione garantiti.

Gli strumenti per la scrittura di un programma in C sono il compilatore ed un editor di testo, basterà scrivere il codice per poi richiamare dalla shell il compilatore con il comando `gcc`, la "struttura" di un programma C è fissata, ci deve sempre essere una funzione chiamata `main()`, ed è la prima funzione che viene chiamata all'avvio del programma quando viene caricato in memoria.

Ovviamente possono essere incluse anche altre funzioni che implementano ulteriori operazioni, è buona regola non scrivere tutto il programma in C, ma suddividere il tutto in apposite funzioni, possono essere scritte in un unico file, o anche in più file diversi, la compilazione genererà comunque un solo eseguibile.

4.1 Sintassi

Una funzione in C ha un intestazione, un tipo di ritorno, dei parametri ed un blocco di codice associato.

```
1
2   return_type function_name(parameter list)
3   {
4       blocco di istruzioni
5   }
```

I blocchi in C vengono aperti e chiusi dalle parentesi graffe, un blocco può contenere dichiarazioni di variabili o dei comandi C da eseguire, detti *statement*, il punto e virgola viene utilizzato per identificare la fine di uno statement, un blocco può contenere 0 o più statement.

Esempio di un codice C :

```
1
2 #include <stdio.h>
3
4 int main()
5 {
6     int x = 5; //dichiarazione di una variabile
7     int y = 0;
8     y = x+5; //statement
9     printf("Ciao mondo!"); //comando per scrivere una stringa in output
10    return 0; //valore di ritorno
11 }
```

Il valore di ritorno può essere costante, una variabile, un'espressione o perfino una chiamata di funzione, l'importante è che sia del tipo specificato.

```
return 0;    return x;    return 3*x+1;    return abs(x);
```

Il simbolo `#`, serve ad indicare al preprocessore quali file vanno inclusi nel contenuto del programma, nel codice di esempio sovrastante, all'inizio del file viene scritto `#include <stdio.h>`, si sta includendo nel programma il file `stdio.h`, tale estensione viene utilizzata per i file *header*, ossia i file che contengono delle funzioni da includere nel programma ma contenute in file separati.

L'inclusione può avvenire in due modi :

- `#include <nome_file.h>` - Viene incluso il file con tale nome, contenuto nella directory dei file standard del C, nel path `/usr/include`.
- `#include "nome_file.h"` - Viene incluso il file con tale nome, contenuto nella stessa directory in cui è contenuto il file che si sta compilando.
- Al momento della compilazione, con il comando `-I`, è possibile specificare il path in cui cercare i file header.

Un programma C interagisce con l'utente grazie alle funzioni contenute nel file `stdio.h`, contenente tutte le operazioni essenziali per l'I/O.

La principale funzione per scrivere delle stringhe nel canale output di default (che è lo schermo) è `printf("format string", value-list)`, essa prende come parametri due valori, la stringa da stampare ed una lista di valori che possono essere variabili, costanti, sequenze di caratteri o espressioni logico-matematiche, e restituisce come valore di ritorno il numero effettivo di caratteri stampati a schermo.

Ci sono alcuni caratteri speciali da utilizzare nella funzione `printf`, ad esempio, scrivendo `\n` si andrà a capo nella stringa.

Nella stringa da stampare è possibile inserire dei "segnaposti" dichiarati con il carattere `%`, essi indicheranno che in quel punto ci sarà il valore di una variabile, definita nella lista di valori, il segnaposto include anche un carattere, che definisce il tipo della variabile che verrà stampata a schermo. `%d` indica che la variabile sarà un numero intero.

```
1
2  int x = 5;
3  printf("Dopo il numero 4 viene il numero %d", x);
4
```

Segnaposto	Tipo
<code>%d</code> o <code>%i</code>	intero
<code>%l</code>	intero (long)
<code>%x</code>	esadecimale
<code>%f</code>	float
<code>%e</code>	float in notazione scientifica
<code>%lf</code>	double

Definendo una variabile, non si sta facendo altro che assegnare un'etichetta ad una locazione di memoria che conterrà il contenuto di tale variabile, tale contenuto può essere modificato durante l'esecuzione del programma.

Una variabile si definisce specificandone il tipo, che ne specificherà anche lo spazio in byte che occuperà in memoria.

```
1
2  int a; //variabile non inizializzata
3  int b = 5; //variabile inizializzata
4  char c;
5  double r;
6
```

Una variabile può contenere dei modificatori opzionali, per indicare se il valore memorizzato ha o non ha il segno, oppure il range che può contenere.

Tipo	Dimensione	Intervallo di valori
signed char	1 byte	[-128,127]
unsigned char	1 byte	[0,255]
short	2 byte	[-32 768,32 767]
int	4 byte	[-2 147 483 648, 2 147 483 647]
float	4 byte	[1.2E-38,3.4E+38]
double	8 byte	[2.3E-308,1.7E+308]

È anche possibile mostrare le dimensioni in byte di una variabile (spazio che occupa in memoria) tramite la funzione `sizeof()`.

Per prendere input da tastiera si utilizza la funzione `scanf("format string", address-list)`, differentemente da `printf`, gli si deve dare l'indirizzo di memoria nella quale dovrà inserire il contenuto letto, è possibile quindi mettere come prefisso all'etichetta di una variabile il carattere `&`, che indicherà il puntatore di quella variabile.

Esempio di programma che utilizza le funzioni di input ed output :

```
1
2 #include <stdio.h>
3
4 int main(){
5
6     printf("Questo programma converte gli euro in lire.\n");
7     printf("Inserire il valore in euro : \n");
8     int euro;
9     scanf("%d", &euro);
10    printf("il valore corrispondente in lire risulta essere %d", euro*200);
11
12    return 0;
13
14 }
15
```