SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" Università di Roma
Ingegneria dell'Informazione,
Informatica e Statistica
Dipartimento di Informatica

Sistemi Operativi II

Appunti integrati con il libro "Advanced Programming in the UNIX environment", W. R. Stevens, S. A. Rago

Autore
Simone Bianco

25 marzo 2024

Indice

Informazioni e Contatti

Appunti e riassunti personali raccolti in ambito del corso di *Sistemi Operativi II* offerto dal corso di laurea in Informatica dell'Università degli Studi di Roma "La Sapienza".

Ulteriori informazioni ed appunti possono essere trovati al seguente link: **https://github.com/Exyss/university-notes**. Chiunque si senta libero di segnalare incorrettezze, migliorie o richieste tramite il sistema di Issues fornito da GitHub stesso o contattando in privato l'autore :

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

Gli appunti sono in continuo aggiornamento, pertanto, previa segnalazione, si prega di controllare se le modifiche siano già state apportate nella versione più recente.

Prerequisiti consigliati per lo studio:

Apprendimento del materiale relativo al corso *Sistemi Operativi I* e conoscenze discrete di programmazione.

Licence:

1

Introduzione all'ambiente Linux

1.1 Breve storia di Unix

Il sistema operativo **Multics (Multiplexed Information and Computing Service)** fu uno dei primi sistemi operativi a **condivisione di tempo (time-sharing)**, ossia multi-processo 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 (System V

AT&T, BSD, Xenix, SunOS/Solaris, ...), tuttavia utilizzabili solo da personale specializzato su dei mainframe. Del corso del tempo, il supporto di AT&T ad Unix iniziò a diminuire, portando le prime comunità di ricercatori e sviluppatori a prendere in mano il progetto.

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.

Attualmente, i vari sistemi Unix rientrano in tre categorie:

- **Generic Unix**, ossia sistemi Unix che provengono da quello originale o da lui derivati
- **Trademark Unix**, ossia i sistemi che rientrano nelle specifiche SUS e pagano per il marchio UNIX
- **Functional Unix**, ossia sistemi operativi che si ispirano a Unix (detti anche **Unix-based**), come GNU e GNU/Linux, ossia la versione di GNU utilizzante kernel Linux (*erroneamente* definita direttamente come sistema operativo Linux nel linguaggio comune - ricordiamo che Linux è solamente il kernel, non l'intero sistema operativo). I sistemi operativi derivanti da Linux (o meglio, da GNU/Linux) sono utilizzati ovunque.

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

- **Multi-utente** e **multi-processo**
- **File system gerarchico**
- **Kernel** in grafo 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)

Proposizione 1: File e Processi

Nei sistemi operativi Unix, **qualsiasi risorsa** può essere rappresentata come un **file**, indipendentemente dall'essere una risorsa hardware o software (es: è possibile interagire con un dispositivo hardware tramite il file che lo rappresenta), o come **processo**.

1.2 Utilizzo della Shell di sistema

Definizione 1: 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** (pertanto il suo nome, in quanto *shell* tradotto sia *guscio* e *kernel* tradotto sia *nucleo*).

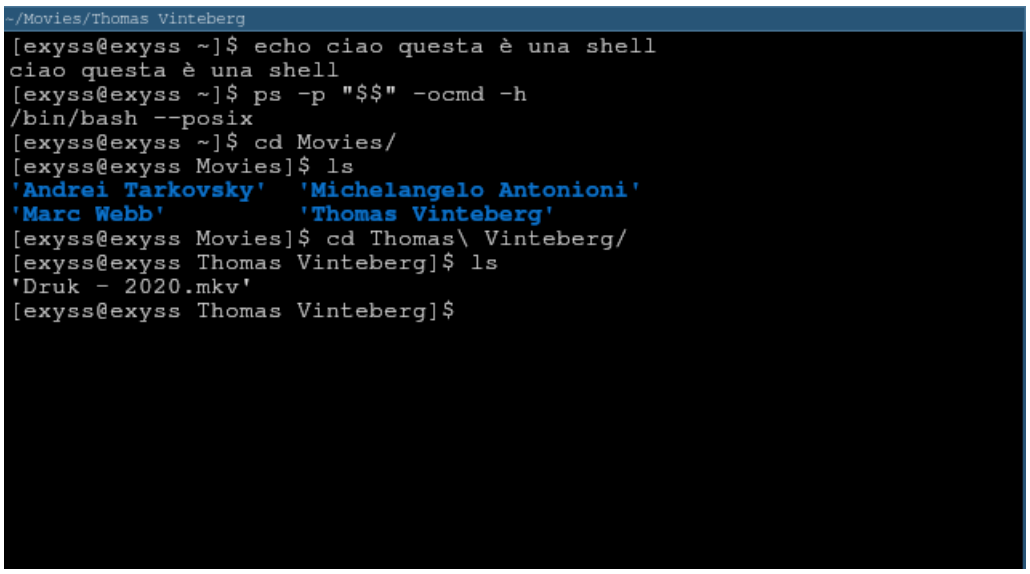
Tali comandi non sono necessariamente dei programmi, bensì possono essere anche dei comandi definiti all'interno della shell stessa.

Esistono vari tipi di shell:

- `sh`, la prima shell inventata da Thompson e Bourne
- `bash`, una versione migliorata di `sh` (il nome deriva da Bourne Again Shell, un gioco di parole tra le parole *Bourne* e *Born*)
- `ksh` (KornShell)
- `fish` (Friendly Interactive Shell)
- ...

In particolare, nei capitoli e sezioni successive considereremo l'uso della shell `bash`.

Per utilizzare `bash` (o una qualsiasi altra shell), è necessario eseguire quello che viene comunemente detto **terminale** (es: programmi come `tty`, `kitty`, `alacritty`, ...).



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

`[nome_utente@nome_macchina cwd]$`

dove `cwd` è la **current working directory**

Definizione 2: Current Working Directory

Definiamo come **current working directory di una shell (cwd)** la cartella attualmente "aperta" all'interno della shell stessa.

Se la `cwd` è impostata sulla home dell'utente attivo (ossia `/home/nome_utente`), essa verrà sostituita direttamente dal simbolo $\sim$.

Ogni **comando** segue la seguente struttura

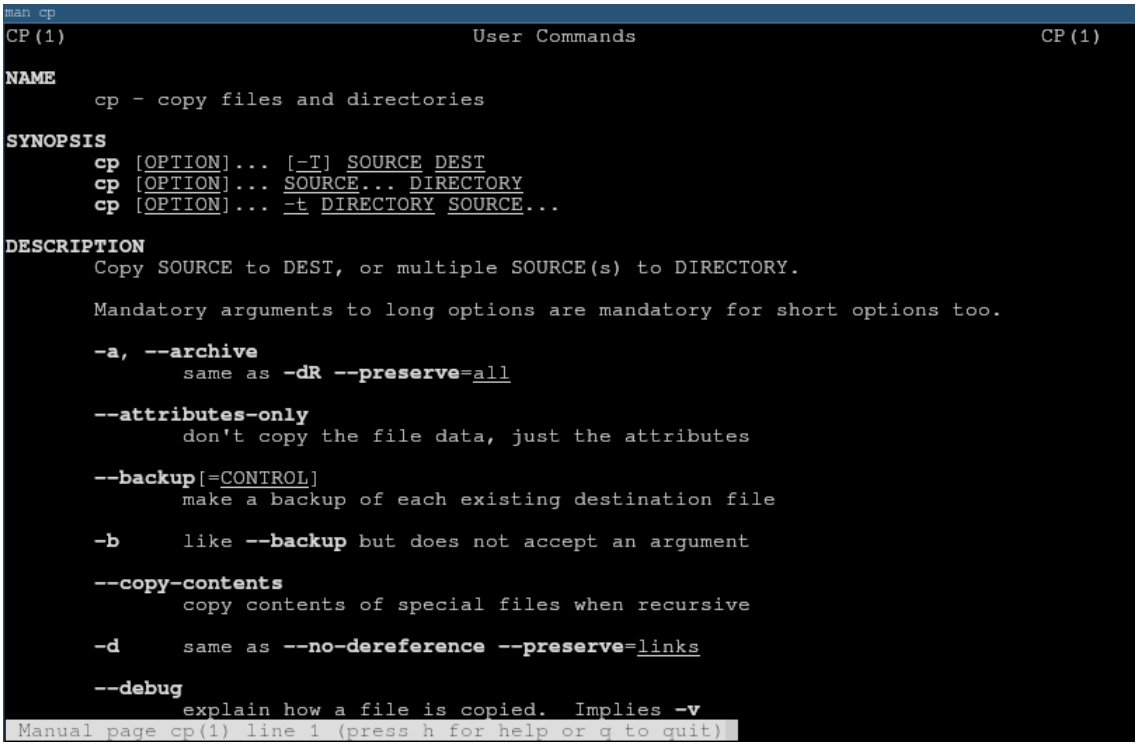
`nome_comando [argomenti_opzionali] argomenti_obbligatori`

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**. Utilizzando le frecce su e giù della tastiera, è possibile scorrere i comandi presenti nella cronologia.

Sebbene siano più o meno simili, ogni comando/programma eseguibile tramite shell segue una propria struttura per gli argomenti opzionali ed obbligatori. Tramite il comando `man nome_comando` è possibile aprire la **pagina del manuale** relativa al comando `nome_comando`.

Ad esempio, eseguendo il comando `man cp` verrà visualizzata la seguente pagina del manuale:



Il manuale possiede un totale di **9 sezioni**:

1. Programmi eseguibili e comandi shell
2. System call fornite dal kernel
3. Library calls fornite dalle librerie dei programmi
4. File speciali (solitamente situati nella cartella `/dev/`)
5. Formati dei file e convenzioni
6. Informazioni su giochi comuni
7. Varie informazioni
8. Comandi di amministrazione di sistema
9. Routine del kernel (non standard)

È possibile specificare la **sezione** della pagina che si vuole aprire aggiungendo il numero di sezione come parametro opzionale. Se tale sezione non esiste, verrà comunicato (es: non esiste una sezione 2 per la pagina del comando `cp`, dunque `man 2 cp` restituirà solamente un messaggio di avviso). Se non viene specificata la sezione da aprire, verrà aperta la sezione di default (solitamente la sezione 1).

1.3 Utenti e Gruppi del sistema

Durante l'installazione di un qualsiasi sistema operativo Linux-based è necessario specificare almeno un **utente**, il quale sarà l'**utente principale**. Alcune distribuzioni creano un utente automaticamente. Per creare un nuovo utente, è possibile utilizzare il comando `adduser nome_nuovo_utente`. Ogni utente è identificato univocamente da un valore intero detto **User ID (UID)**.

Gli utenti possono appartenere a dei **gruppi utente**. In particolare, ogni utente appartiene ad almeno un gruppo, ossia il gruppo avente lo stesso nome dell'utente principale, generato automaticamente alla creazione dell'utente stesso. Per ogni gruppo possono essere impostati **privilegi diversi**. Inoltre, come per gli utenti, ogni gruppo è identificato univocamente da un **Group ID (GID)**.

Per listare i gruppi a cui appartiene un utente, è possibile utilizzare il comando `groups [nome_utente]`, mentre per aggiungere un utente ad un gruppo è possibile utilizzare il comando `adduser nome_utente nome_gruppo`.

Definizione 3: 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.

È necessario notare che non tutti gli utenti possono effettuare il login nel sistema. Ad esempio, l'utente root non può effettuare il login, ma un utente può acquisire i diritti di root tramite i comandi `su` e `sudo`.

Gli utenti appartenenti al **gruppo sudo** (un gruppo predefinito speciale) vengono detti **sudoer** e sono in grado di eseguire il comando `sudo nome_comando`, il quale permette loro di eseguire il comando dato **impersonando root**. In alcune distribuzioni della famiglia Ubuntu, l'utente principale è già un *sudoer*.

Per **cambiare utente**, invece, è possibile utilizzare il comando `su [-l] nome_utente` (solitamente utilizzato per cambiare utente attivo in root). Inoltre, l'opzione `[-c command]` del comando `su` permette di eseguire un singolo comando impersonando l'utente dato, invece di cambiare utente. Per tanto, è possibile ottenere tramite `su -c command root` lo stesso effetto del comando `sudo command`.

2

File System

2.1 Introduzione al file system Linux

Definizione 4: 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.

Come già espresso nel capitolo precedente, 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.

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`)

Osservazione 1

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

Inoltre, è necessario puntualizzare che, come accennato nel capitolo precedente, il simbolo ∼ sia un abbreviativo del percorso `/home/utente` (se l'utente che usa tale simbolo si chiama effettivamente `utente`). Per tanto, i percorsi avente ∼ come prefisso, sebben sembrino dei percorsi relativi, risultano essere effettivamente dei percorsi assoluti.

(es: per l'utente `marco`, il percorso ∼`/ciao.txt` equivale a `/home/marco/ciao.txt`)

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

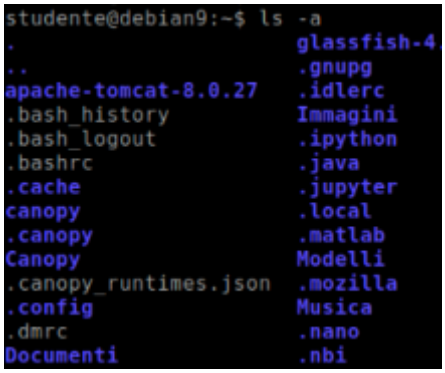
  (es: se la cwd attuale è ∼`/dir/subdir1/`, il percorso `../subdir2` è equivalente al percorso ∼`/dir/subdir2`)

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.

(es: il file `ciao.txt` può essere reso nascosto cambiando il suo nome in `.ciao.txt`)



Similmente a `ls -r`, il comando `tree [path]` permette di listare il ricorsivamente il contenuto di una directory, ma sotto forma di albero.

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

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`

Osservazione 2

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 (dunque il numero di blocchi da scartare da quelli letti in input)

Esempio:

- Il comando `dd if=filein of=fileout bs=1 skip=1 count=100` salta un carattere (1 blocco da 1 carattere), per poi leggere 100 caratteri (100 blocchi da 1 carattere)
- Il comando `dd if=filein of=fileout bs=100 skip=1 count=1` salta 100 caratteri (1 blocco da 100 caratteri), per poi leggere 100 caratteri (1 blocco da 100 caratteri)

- Il comando `dd if=filein of=fileout bs=100 seek=1 count=1` legge 100 caratteri (1 blocco da 100 caratteri), per poi scartare 100 caratteri da quelli letti (1 blocco da 100 caratteri)

2.2 Mounting, Partizioni e Tipi di file system

Il file system principale (ossia `/`) può contenere al suo interno elementi **eterogenei** tra loro, ad esempio:

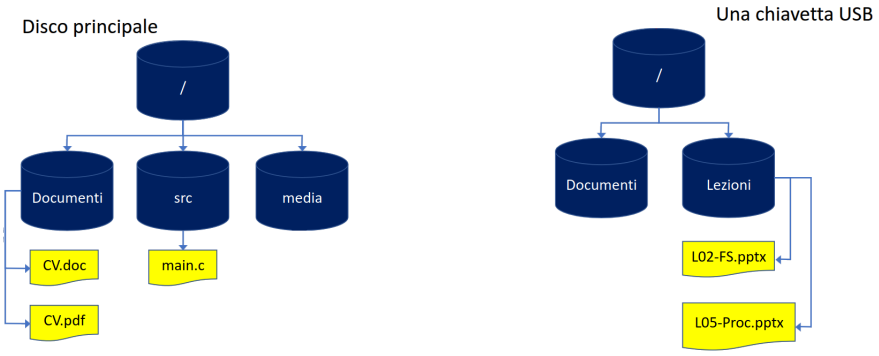
- Dischi interni solidi o magnetici, solitamente contenenti il file system root
- File system su disco esterno
- File system di rete
- File system virtuali
- File system in memoria principale

Definizione 5: Mounting

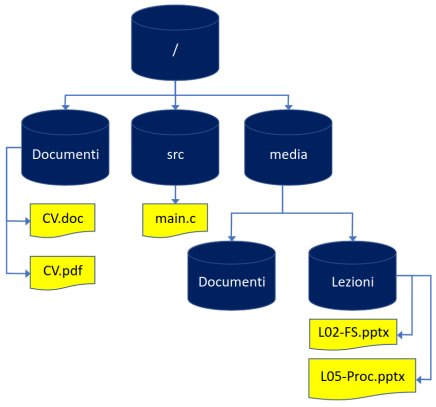
Una qualsiasi directory D all'interno del file system root può diventare il **punto di mount** per un altro file system F se e solo se la directory di root R_F del file system F diventa accessibile da F.

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



Osservazione 3

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`.

Definizione 6: 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.

Esempio:

- Supponiamo che un disco sia partizionato in due partizioni A e B
- La partizione A può contenete il sistema operativo, mentre la seconda può contenente i dati degli utenti (dunque la home directory dei vari utenti)
- Pertanto, la partizione A verrà montata sulla directory `/`, mentre la partizione B verrà montata sulla directory `/home`
- Tale partizionamento risulta vantaggioso per alcune situazioni (es: se si necessita di reinstallare il sistema operativo, è possibile farlo direttamente sulla partizione A, senza intaccare la partizione B)

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.

Tra i vari file system non Linux, invece, troviamo `NTFS`, `MSDOS`, **FAT16**, **FAT32** e **FAT64**. I file system `FAT` e `NTFS` possono essere montati anche su un file system Linux.

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).

2.3 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`

dove il campo `gid` contiene il GID del gruppo principale dell'utente, il campo `gecos` contiene una breve descrizione dell'utente, il campo `homedir` contiene il percorso verso la cartella home dell'utente e il campo `shell` contiene il percorso verso la shell predefinita per l'utente.

Solitamente, inoltre, il campo `password` di ognuna di tali righe viene mascherato dal carattere `x` Difatti, la vera password di un utente viene conservata all'interno del file `/etc/shadow`, sotto forma di hash stesso della password.

```
[exyss@exyss ~]$ cat /etc/passwd
root:x:0:0::/root:/bin/bash
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
http:x:33:33::/srv/http:/usr/bin/nologin
nobody:x:65534:65534:Kernel Overflow User:/:/usr/bin/nologin
dbus:x:81:81:System Message Bus:/:/usr/bin/nologin
systemd-coredump:x:981:981:systemd Core Dumper:/:/usr/bin/nologin
systemd-network:x:980:980:systemd Network Management:/:/usr/bin/nologin
systemd-oom:x:979:979:systemd Userspace OOM Killer:/:/usr/bin/nologin
systemd-journal-remote:x:978:978:systemd Journal Remote:/:/usr/bin/nologin
systemd-resolve:x:977:977:systemd Resolver:/:/usr/bin/nologin
systemd-timesync:x:976:976:systemd Time Synchronization:/:/usr/bin/nologin
tss:x:975:975:tss user for tpm2:/:/usr/bin/nologin
uuidd:x:68:68::/:/usr/bin/nologin
exyss:x:1000:1000::/home/exyss:/bin/bash
dhcpcd:x:974:974:dhcpcd privilege separation:/:/usr/bin/nologin
```

- 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`.

2.4 Index Node (inode)

Definizione 7: 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:

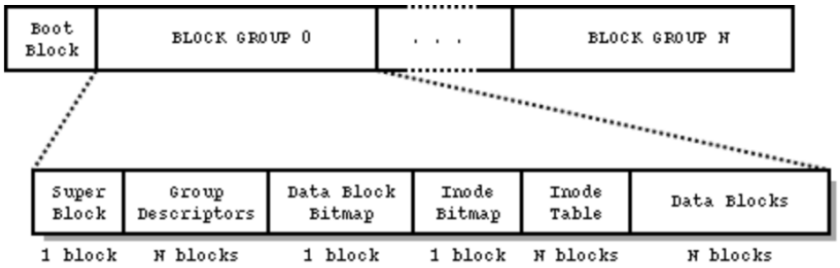
- **inode number**, univoco per ogni inode
- **Type**, indicante il tipo di file
- **User ID (UID)**, ossia l'ID dell'utente proprietario del file
- **Group ID (GID)**, ossia l'ID del gruppo a cui è associato il file
- **Mode**, ossia i permessi di accesso al file per il proprietario, il gruppo associato 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.

Osservazione 4

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.

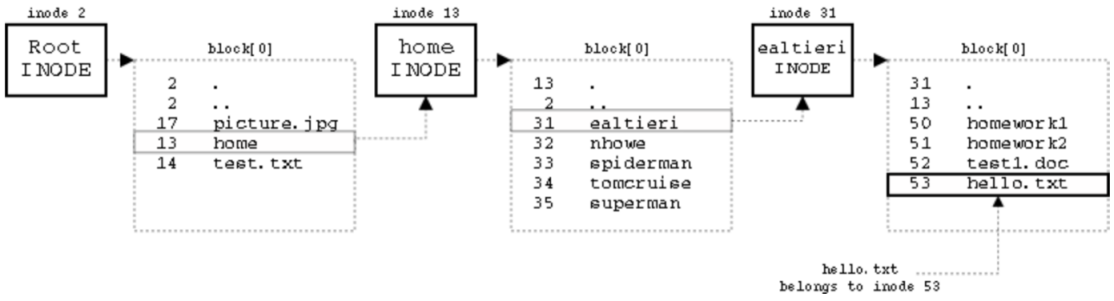
All'interno del file system (in particolare all'inizio del disco o partizione su cui è installato il file system) si trova una **tabella degli inode**. Ad esempio, nei file system `ext2` la tabella degli inode viene conservata all'interno del primo gruppo di blocchi:



Osservazione 5

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/ealtieri/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

```
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 enthought
-rw-r--r--  1 studente studente    7 mar  6 06:33 filevuoto
```

Oltre al comando `ls`, il comando `stat` permette di visualizzare in modo dettagliato tutte le informazioni dell'inode di un file.

```
[exyss@exyss ~]$ stat SO2.txt
  File: SO2.txt
  Size: 17692           Blocks: 40         IO Block: 4096   regular file
Device: 259,2   Inode: 20463063    Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   exyss)   Gid: ( 1000/   exyss)
Access: 2023-07-05 23:39:55.090536291 +0200
Modify: 2023-07-05 23:39:55.083869661 +0200
Change: 2023-07-20 23:44:23.945544918 +0200
 Birth: 2023-07-03 17:50:06.381983489 +0200
```

Per **sommare le dimensioni di vari file e/o directory**, è possibile utilizzare il comando `du [files...]` (consultare il manuale).

2.4.1 Hard link e Soft link

Definizione 8: 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.

Osservazione 6

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**.

Osservazione 7

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.

```
[exyss@exyss ~]$ ls -l ciao
-rw-r--r-- 1 exyss exyss 0 Jul 21 19:17 ciao
[exyss@exyss ~]$ ln ciao ciao1
[exyss@exyss ~]$ ln -s ciao ciao2
[exyss@exyss ~]$ ln -s ./././ciao ciao3
[exyss@exyss ~]$ ls -l ciao*
-rw-r--r-- 2 exyss exyss  0 Jul 21 19:17 ciao
-rw-r--r-- 2 exyss exyss  0 Jul 21 19:17 ciao1
lrwxrwxrwx 1 exyss exyss  4 Jul 21 19:17 ciao2 -> ciao
lrwxrwxrwx 1 exyss exyss 12 Jul 21 19:18 ciao3 -> ./././ciao
```

Osservazione 8

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.

```
[exyss@exyss ~]$ rm ciao
[exyss@exyss ~]$ ls -l ciao*
-rw-r--r-- 1 exyss exyss  0 Jul 21 19:17 ciao1
lrwxrwxrwx 1 exyss exyss  4 Jul 21 19:17 ciao2 -> ciao
lrwxrwxrwx 1 exyss exyss 12 Jul 21 19:18 ciao3 -> ./././ciao
[exyss@exyss ~]$ cat ciao2
cat: ciao2: No such file or directory
```

2.5 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)**.

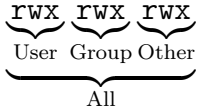
Per ognuna delle terne di bit, ognuno di essi corrisponde ad un permesso:

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, permette 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:



Esempi:

- I permessi `rw-r--r--` (corrispondente a `644`) indicano che tutti gli utenti possono leggere il contenuto del file, ma solo il proprietario possa scrivere su di esso

- I permessi `rwx-wx--w--` (corrispondente a `732`) indicano che il proprietario può svolgere qualsiasi operazione, gli utenti del gruppo associato può scrivere ed eseguire il file e tutti gli altri possano solo scrivere sul file.
- I permessi `rwxrwxrwx` (corrispondente a `777`) indicano che tutti gli utenti possano eseguire qualsiasi operazione.
- I permessi `---------` (corrispondente a `000`) indicano che nessun utente possa eseguire alcuna operazione.

Solitamente, a tale terna viene anche associato come prefisso anche il campo `type` del inode, in modo da poter distinguere se tali permessi siano relativi ad un file regolare o solitamente una directory.

Esempi:

- I permessi `-rwxrwxrwx` indicano che l'inode è relativo ad un file regolare su cui ogni utente può svolgere qualsiasi operazione
- I permessi `drwxrwxrwx` indicano che l'inode è relativo ad una directory su cui ogni utente può svolgere qualsiasi operazione,

In particolare, nel caso delle **directory** gli effetti ottenuti in base ai permessi associati non risultano del tutto intuitivi:

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 sotto-directory nel caso in cui `nome_file` sia una directory. Tali comandi possono essere utilizzati solo da `root`, richiedendo quindi l'uso di `sudo`.

2.5.1 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**:

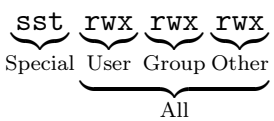
- Il primo bit (partendo da destra) corrisponde allo **sticky bit**.

  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` $\neq$ `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 nessun 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.

Definizione 9: 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)`

Esempio:

- 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.

3 Processi

Definizione 10: Programmi e 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.

Osservazione 9

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.

3.1 Canali dei processi

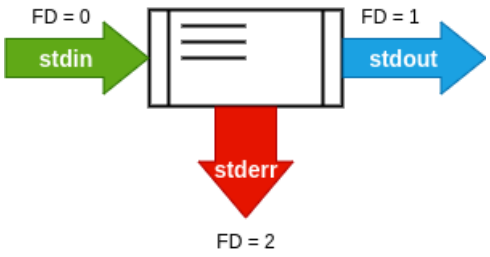
Definizione 11: Canali e File descriptor

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**.

Proposizione 2: 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.

Osservazione 10

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

Esempi:

- Tramite `ls 2>&1 > out.txt`, viene ridirezionato lo stderr del comando `ls` verso il suo stdout, mentre il suo stdout viene ridirezionato verso il file `out.txt`, sovrascrivendone il contenuto, il cui contenuto corrisponderà esclusivamente al flusso dati inserito nello stdout prima del ridirezionamento dello stderr verso lo stdout.

Definizione 12: 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 |& ...`).

3.2 Attributi e rappresentazione dei processi

Definizione 13: 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.

Osservazione 11

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.

Definizione 14: 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)

In comandi `renice priority pid1,pid2,...`, invece, permette di modificare la niceness dei processi in esecuzione aventi PID interno alla lista data.

Definizione 15: 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

Osservazione 12: 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

Osservazione 13

Sebbene molto situazionale, il SUID e il SGID possono essere utilizzati da un processo eseguito con EUID e/o EGID privilegiati (es: tramite SetUID e/o SetGID) per impostare momentaneamente l'EUID/EGID pari al RUID/RGID, per poi tornare all'EUID/EGID precedente tramite il SUID/SGID.

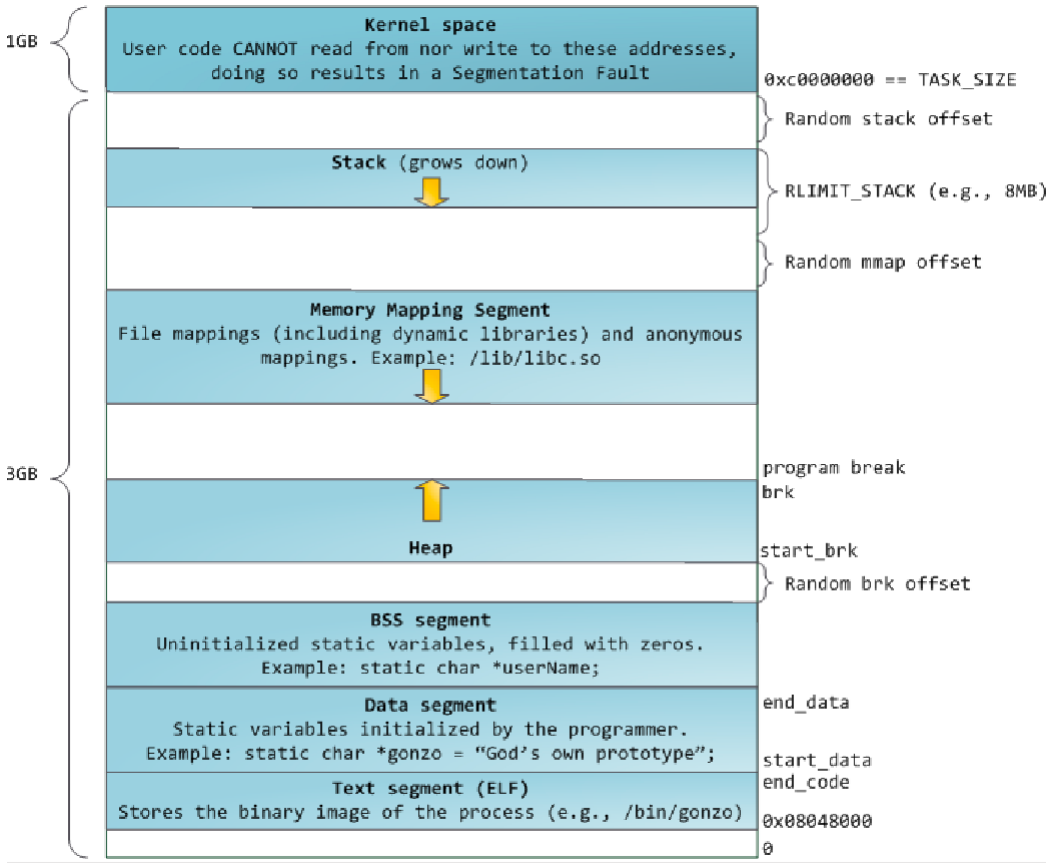
Esempio:

- Supponiamo che il programma `myProgram` abbia il bit SetUID impostato e che il suo proprietario sia `root`.
- All'avvio del programma da parte dell'utente `user`, il RUID e il SUID del processo vengono automaticamente impostati a `user`, mentre l'EUID viene impostato a `root`.
- Supponiamo che, dopo varie operazioni da super utente, il programma debba creare un file per l'utente `user`.
- Se il programma creasse tale file senza svolgere prima operazioni preliminari, il proprietario del file sarebbe `root`, poiché l'EUID del processo è impostato a `root`.
- Per tanto, il programma pone l'EUID pari al RUID, ossia `user`, prima di creare il file. In tal modo, il SUID verrà automaticamente posto all'EUID precedente, ossia `root`
- Successivamente, il programma potrà porre l'EUID pari al SUID, tornado ad avere `root` come EUID

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** (si consiglia vedere gli appunti relativi al Modulo 1 per entrambi i concetti).

Per quanto riguarda le aree di memoria, esse si suddividono in:

- **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

```
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
```

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`.

Osservazione 14

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.

3.3 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

Definizione 16: 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).

Esempio:

- Il comando `sleep 5 | sleep 5 | sleep 5` è un job composto da tre processi sleep eseguiti uno dopo l'altro

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

```
[exyss@exyss ~]$ (sleep 5; echo "ciao dal background!") &
[1] 5457
[exyss@exyss ~]$ sleep 5; echo "ciao dal foreground!"
ciao dal background!
[1]+  Done                    ( sleep 5; echo "ciao dal background!" )
ciao dal foreground!
```

Osservazione 15

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.

Osservazione 16: Job number

Ogni job eseguito all'interno di una shell possiede un **job number**, partendo dal numero 1. Due job possono avere lo stesso job number se eseguiti all'interno di due shell diverse.

Per visualizzare la lista dei job attualmente attivi, è possibile utilizzare il comando `jobs [-l] [-p]`, dove l'opzione `[-l]` permette di listare anche i PID dei processi coinvolti nel job, mentre l'opzione `[-p]` permette di listare solamente il PID del processo attualmente in esecuzione di ogni job attivo.

```
[exyss@exyss ~]$ sleep 10 | sleep 10 | sleep 10 &
[1] 2059
[exyss@exyss ~]$ sleep 15 &
[2] 2062
[exyss@exyss ~]$ jobs
[1]-  Running                 sleep 10 | sleep 10 | sleep 10 &
[2]+  Running                 sleep 15 &
[exyss@exyss ~]$ jobs -l
[1]-  2057 Running                 sleep 10
      2058                       | sleep 10
      2059                       | sleep 10 &
[2]+  2062 Running                 sleep 15 &
[exyss@exyss ~]$ jobs -p
2057
2062
```

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`

```
[exyss@exyss ~]$ sleep 50
^Z
[1]+  Stopped                 sleep 50
[exyss@exyss ~]$ bg %1
[1]+ sleep 50 &
[exyss@exyss ~]$ fg %1
sleep 50
^Z
[1]+  Stopped                 sleep 50
```

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)

3.4 Segnali dei processi

Definizione 17: Segnale

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

Osservazione 17

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.

Osservazione 18

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)

Osservazione 19

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

4

Linguaggio C

4.1 Introduzione al linguaggio

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 tutto ciò che è successivo ad essi fino ai successivi caratteri `*/`

Vediamo quindi un primo esempio di programma in linguaggio C in grado di stampare sul terminale la stringa `"Hello World!"`:

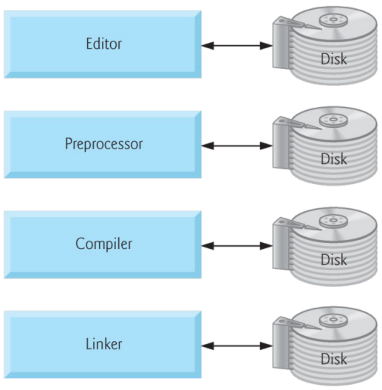
```
#include <stdio.h>

int main() {
    printf(“Hello World!\n”);    //stampa sul terminale
    return 0;   // terminato con successo
}
```

4.2 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**
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**, ossia far eseguire al pre-processore tutte le sue direttive ed eliminare i commenti, 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`:

Definizione 18: 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
- Tramite il comando `gcc -c file.c -o file.o`, dove `file.c` non è stato precompilato, viene eseguita sia la **precompilazione** sia la **compilazione**
- Tramite 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
- Tramite il comando `gcc file.c`, dove file.c non è precompilato, vengono eseguite tutte le fasi

4.3 Variabili ed tipi di dato

Come nella maggior parte dei linguaggi di programmazione, il linguaggio C fornisce il supporto per l'uso di **variabili**, ossia delle locazioni in memoria in cui può essere memorizzato un valore che verrà utilizzato dal programma stesso.

Ogni variabile deve essere **dichiarata**, ossia ne deve essere definita la struttura, prima di poter **assegnare** un valore ad essa, ossia modificarne il valore "contenuto" al suo interno.

In particolare, definiamo il primo assegnamento effettuato su una variabile o costante come **inizializzazione**. Inoltre, è possibile inizializzare una variabile anche durante la sua dichiarazione:

```
void main() {
    int x;  //dichiaro una variabile
    x = 5;  //inizializzo la variabile

    const int y = 0;    //dichiaro ed inizializzo una costante
}
```

Ogni variabile o costante possiede un **identificatore**, ossia un nome utilizzato dal programmatore e dal compilatore per tale variabile.

Gli identificatori delle variabili (o costanti) devono rispettare le seguenti regole:

1. Il primo carattere deve essere una lettera o un underscore (ossia `_`) e può essere seguito solo da lettere, numeri o underscore
2. Il nome è *case sensitive*, ossia vi è distinzione tra lettere maiuscole e minuscole
3. Un'identificatore non può coincidere con una keyword del linguaggio
4. La lunghezza massima per un identificatore è pari a 31 caratteri

Esempi validi	**Esempi non validi**
distance	x-ray
distance32	2ndGrade
milesPerHour	$amount
_voltage	two&four
goodChoice	after five
MIN_RATE	return

Proposizione 3: Dichiarazione di variabili e costanti

Per dichiarare una lista di variabili viene utilizzato il seguente statement:

```
optional_modifier data_type name_list
```

dove:

- Il campo `name_list` è una **lista di identificatori** (uno per ogni variabile che si sta dichiarando)
- Il campo `data_type` specifica il **tipo di valore** della variabile, permettendo al compilatore di sapere quali sono le operazione consentite e come esso debba esser rappresentato in memoria
- Il campo `optional_modifier` definisce delle "modifiche" opzionali al tipo di valore assunto dalla variabile. In particolare, al suo interno può essere specificata una (o nessuna) keyword dei seguenti tre insiemi:
  - I modificatori `signed` o `unsigned` indicano se nel tipo di valore debba essere considerato il segno o meno (es: per i numeri negativi). Le variabili `signed` hanno accesso ad un range di valori più alto. Se omesso, esso è impostato di default a `signed`.
  - I modificatori `short` o `long` indicano se si voglia una variabile di dimensione inferiore o superiore rispetto alla dimensione normale utilizzata dal tipo di valore. Se omesso, viene utilizzata la dimensione normale.
  - I modificatori `const` indicano se la variabile sia una costante, ossia che una volta inizializzata essa non possa mai più modificata. Se omesso, la variabile non sarà una costante.

In particolare, considerando anche i modificatori, il linguaggio C dispone dei seguenti **tipi base di variabile intera**:

Tipo	Num. Byte	Intervallo	Placeholder
`char`	1 byte	$[-2^7, 2^7-1]$ o $[0, 2^8-1]$	`%c` o `%hhi`
`signed char`	1 byte	$[-2^7, 2^7-1]$	`%c` o `%hhi`
`unsigned char`	1 byte	$[0, 2^8-1]$	`%c` o `%hhi`

`short`
`short int`
`signed short`
`signed short int`
	2 byte	$[-2^{15}, 2^{15}-1]$	`%hd` o `%hi`

`unsigned short` `unsigned short int`	2 byte	$[0, 2^{16}-1]$	`%hu`

`int`
`signed`
`signed int`
	2 o 4 byte	$[-2^{15}, 2^{15}-1]$ o $[-2^{31}, 2^{31}-1]$	`%d` o `%i`

`unsigned` `unsigned int`	2 o 4 byte	$[0, 2^{16}-1]$ o $[0, 2^{32}-1]$	`%u`

`long`
`long int`
`signed long`
`signed long int`
	4 byte	$[-2^{31}, 2^{31}-1]$	`%ld` o `%li`

`unsigned long` `unsigned long int`	4 byte	$[0, 2^{32}-1]$	`%lu`

`long long`
`long long int`
`signed long long`
`signed long long int`
	8 byte	$[-2^{63}, 2^{63}-1]$	`%lld` o `%lli`

`unsigned long long` `unsigned long long int`	8 byte	$[0, 2^{64}-1]$	`%llu`

Per quanto riguarda i **tipi base di variabile reale** (ossia in **notazione floating-point** (Standard IEEE 754)), si dispone dei seguenti tipi:

Tipo	Num. Byte	Intervallo	Cifre decimali	Placeholder
`float`	4 byte	$[1.2 \cdot 10^{-38}, 3.4 \cdot 10^{38}]$	6	`%f` o `%e`
`double`	8 byte	$[2.3 \cdot 10^{-308}, 1.7 \cdot 10^{308}]$	15	`%lf` o `%Le`
`double`	10 byte	$[3.4 \cdot 10^{-4932}, 1.1 \cdot 10^{4932}]$	19	`%Lf` o `%Le`

Osservazione 20

I **limiti massimi e minimi dell'intervallo**di ogni tipo possono facilmente essere ottenuti importando la libreria `<limits.h>` (es: la costante `INT_MAX` corrisponde al limite massimo dell'intervallo del tipo `int`)

Proposizione 4: Operatore sizeof

Il **numero di byte** occupati da un tipo di dato (non solo quelli base) può essere ottenuto tramite l'operatore `sizeof(type)`, dove `type` è il tipo di dato interessato

Osservazione 21

Al tipo `char` può essere assegnato un **carattere ASCII** circondandolo tra due caratteri `'` (es: `char x = 'A';`).

Il carattere assegnato viene automaticamente convertito dal compilatore nel suo valore decimale (es: `char x = 'A'` equivale a `char x = 65;`)

Inoltre, abbiamo due tipi di dato relativi ai **valori booleani**:

- Il tipo `_Bool`, utilizzato implicitamente dal compilatore. Può memorizzare solo il valore 0 e il valore 1. Ad esso può essere assegnato il risultato di un'**espressione logica** (se il risultato è 0, verrà assegnato 0, altrimenti verrà assegnato 1, indipendentemente dal valore)
- Il tipo `bool`, richiedente l'importazione della libreria `<stdbool.h>`. Può assumere solo i valori `true` e `false`. Se ad esso viene assegnato il valore 0, verrà interpretato come `false`, altrimenti come `true`, indipendentemente dal valore.

Esempio:

```
#include <stdbool.h>

void main(){
    bool x;
    x = 0;      //viene assegnato false
    x = 1;      //viene assegnato true
    x = 4316;   //viene assegnato true
}
```

Definizione 19: Casting

Definiamo come **casting** la conversione di un valore di un tipo ad un altro tipo. Per effettuare il casting della variabile `var` al tipo `type`, è sufficiente utilizzare l'**operatore** `(type) var`.

Osservazione 22

Il casting di un valore potrebbe generare **effetti indesiderati** (es: castando un valore float in un intero, verrà troncata la parte decimale del valore)

Esempio:

```
void main(){
    float x = 7.3;
    int y = (int) x;    //7.3 viene convertito in 7
}
```

4.4 Input e output da terminale

Tutte le funzionalità relative all'**input** e all'**output** vengono fornite dalla libreria di sistema `<stdio.h>`.

In particolare, all'interno di tale libreria viene fornita la funzione `printf(format_string, value1, value2, ...)`, la quale si occupa di **scrivere** la stringa `format_string` data in input sullo **stdout**.

La stringa `format_string` data in input può contenere al suo interno dei **placeholder** (sezione 4.3). Al posto dell'n-esimo placeholder verrà inserito il valore assunto dall'n-esimo valore dato in input alla funzione `printf()`.

Esempio:

```
int main(){
    int x = 5;

    printf("Il valore di x è: %d", x);
    //viene stampato "Il valore di x è: 5"

    printf("Il valore di x+%d è: %d", 5, x+5);
    //viene stampato "Il valore di x+5 è: 10"
}
```

All'interno delle stringhe formattate è possibile specificare il **formato di un placeholder**, ossia la modalità con cui il valore ad esso associato debba essere interpretato:

`%[parameter][flags][width][.precision][length]type`

Esempio:

- Il placeholder `%.3f` indica che il valore associato debba essere interpretato come un `float` avente 3 cifre decimali. Per tanto, l'istruzione `printf(".3f", x)` stamperà le prime 3 cifre decimali della variabile `x`

Osservazione 23

I placeholder `%b`, `%o` e `%x` indicano che i valori ad essi associati sono da interpretate rispettivamente come un intero binario, un intero ottale e un intero esadecimale.

Definizione 20: Escape sequences

All'interno delle stringhe formattate è possibile controllare la **spaziatura orizzontale e verticale** utilizzando le **escape sequences**, ossia caratteri ASCII speciali in grado di spostare il cursore di scrittura:

- Il carattere `\b` (**backspace**) sposta il cursore sul carattere precedente
- Il carattere `\r` (**carriage return**) sposta il cursore all'inizio della riga
- Il carattere `\f` (**form feed**) sposta il cursore alla riga successiva, mantenendo lo stesso allineamento orizzontale precedente
- Il carattere `\n` (**line feed**) sposta il cursore all'inizio della riga precedente (ha lo stesso effetto di `\f\r` e `\r\f`)
- Il carattere `\t` (**horizontal tab**) sposta il cursore al tab orizzontale successivo
- Il carattere `\v` (**vertical tab**) sposta il cursore al tab verticale successivo (equivalente a `\f` in molti shell moderni)

Esempio:

- L'istruzione

```
printf("aa\bkaaa\fbbbbb\f\rc\tcccc\r\fddddd\reeeee\nfffff");
```

  produce il seguente output:

```
akaaa
     bbbbb
c   ccc
eeeee
fffff
```

Molto simile alla funzione `printf()` è la funzione di libreria `scanf(format_string, address1, address2, ...)`. A differenza della precedente funzione, essa è in grado **leggere** dati dallo **stdin**.

La stringa `format_string` data in input può contenere al suo interno dei **placeholder** del tutto analoghi a quelli di `printf()`: l'n-esimo valore contenuto nell'input di stdin verrà interpretato con il formato indicato dall'n-esimo placeholder, inserendo il valore interpretato all'interno dell'n-esimo indirizzo di memoria fornito in input alla funzione `scanf()`.

Proposizione 5: Operatore di riferimento

Il simbolo `&` posto davanti all'identificatore di una variabile permette di ottenere il suo **indirizzo di memoria** (es: `&var` restituisce l'indirizzo di `var`).

Tale operatore viene detto **riferimento**.

Esempio:

- Supponiamo che venga eseguita l'istruzione `scanf("%d %f", &x, &y)`
- Supponiamo inoltre che l'input dato su stdin sia "123 123.24"
- Il contenuto delle variabili `x` e `y` dopo la lettura sarà rispettivamente 123 e 123.24

Osservazione 24

La funzione `printf()` restituisce in output il numero di caratteri stampati, mentre la funzione `scanf()` restituisce il numero di valori letti correttamente.

Se viene passato in input ad stdin un valore di un **formato diverso da quello atteso** dalla funzione `scanf()`, la funzione smetterà di interpretare l'input e di assegnare i valori, bloccandosi nel mezzo. In alcune situazioni, ad esempio se viene passato un carattere in input quando il formato atteso era un float, l'input di stdin potrebbe **rimanere bloccato fino alla chiusura del programma**. Per tanto, si consiglia di sperimentare molto con la funzione `scanf()`.

4.5 Operatori aritmetici, bit-wise e logici

Come molti altri linguaggi, il linguaggio C fornisce i seguenti operatori aritmetici:

- L'operatore `val1 + val2` restituisce la somma tra i valori `val1` e `val2`
- L'operatore `val1 - val2` restituisce la differenza tra i valori `val1` e `val2`
- L'operatore `val1 * val2` restituisce il prodotto tra i valori `val1` e `val2`
- L'operatore `val1 / val2` restituisce il quoziente tra i valori `val1` e `val2`.
- L'operatore `val1 % val2` restituisce il resto del quoziente tra i valori `val1` e `val2`

Osservazione 25

L'operatore `/` dipende dai **tipi di dato degli operandi**:

- Il risultato è dello **stesso tipo dell'operando più grande** in termini di tipi di dato (es: un float è più grande di un intero)
- Il tipo del risultato potrebbe subire delle **approssimazioni** (es: dividere un intero per un'altro intero approssima per difetto il risultato della divisione)

Oltre agli operatori aritmetici, viene fornito il supporto anche per **operatori bit-wise (bit-a-bit)**, ossia operatori binari applicati sui singoli bit che rappresentano un valore:

- L'operatore `val1 & val2` restituisce il bit-wise AND tra i bit dei valori `val1` e `val2` (es: se in binario si ha `val1 = 10111` e `val2 = 00101`, allora `val1 & val2` è uguale a `00101`)
- L'operatore `val1 | val2` restituisce il bit-wise OR tra i bit dei valori `val1` e `val2` (es: se in binario si ha `val1 = 10111` e `val2 = 00101`, allora `val1 | val2` è uguale a `10111`)
- L'operatore `val1 ^ val2` restituisce il bit-wise XOR tra i bit dei valori `val1` e `val2` (es: se in binario si ha `val1 = 10111` e `val2 = 00101`, allora `val1 ^ val2` è uguale a `10010`)
- L'operatore `!val1` restituisce il bit-wise NOT tra i bit del valore `val1` (es: se in binario si ha `val1 = 10111`, allora `!val1` è uguale a `01000`)
- L'operatore `val1 <<̇ val2` shifta a sinistra i bit del valore `val1` di `val2` posizioni (es: se in binario si ha `val1 = 10111` e in decimale `val2 = 2`, allora `val1 <<̇ val2` è uguale a `11100`)
- L'operatore `val1 >>̇ val2` shifta a destra i bit del valore `val1` di `val2` posizioni (es: se in binario si ha `val1 = 10111` e in decimale `val2 = 2`, allora `val1 <<̇ val2` è uguale a `00111`).

  Se il valore `val1` è `unsigned`, allora lo shift effettuato sarà di tipo logico, altrimenti sarà di tipo aritmetico.

Infine, il linguaggio C fornisce anche il supporto agli **operatori logici** ed **operatori di confronto**:

- L'operatore `exp1 && exp2` valuta l'AND logico tra le espressioni `exp1` e `exp2`
- L'operatore `exp1 || exp2` valuta l'OR logico tra le espressioni `exp1` e `exp2`
- L'operatore `!exp1` valuta il NOT logico dell'espressione `exp1`
- L'operatore `val1 == val2` valuta se `val1` sia uguale a `val2`
- L'operatore `val1 != val2` valuta se `val1` sia diverso da `val2`

Osservazione 26

Per quanto riguarda gli operatori logici, ricordiamo che se un valore è uguale a 0 esso viene interpretato come `false`, altrimenti viene interpretato come `true`

La maggior parte degli operatori visti gode di possibili **abbreviazioni**:

Operatore	Abbreviazione
`var = var + 1`	`var++` o `++var`
`var = var - 1`	`var--` o `--var`
`var = var + val_var`	`var += val_var`
`var = var - val_var`	`var -= val_var`
`var = var * val_var`	`var *= val_var`
`var = var / val_var`	`var /= val_var`
`var = var % val_var`	`var %= val_var`
`var = var & val_var`	`var *= val_var`
`var = var | val_var`	`var *= val_var`
`var = var ^val_var`	`var ^= val_var`
`var = var <<̇ val_var`	`var <<̇= val_var`
`var = var >>̇ val_var`	`var >>̇= val_var`

Osservazione 27: Operatori `++` e `--`

L'operatore `++` può essere utilizzato in due modalità:

- **Pre-incremento**: se l'operatore `++var` è inserito all'interno di un'espressione, l'incremento verrà effettuato prima della valutazione dell'espressione
- **Post-incremento**: se l'operatore `var++` è inserito all'interno di un'espressione, l'incremento verrà effettuato dopo la valutazione dell'espressione

Lo stesso vale per l'operatore `--`.

Esempio:

```
void main(){
    int x = 0;
    int y;

    y = 2 * (++x);
    /*
        verrà prima incrementato x, per poi valutare
        l'espressione, dunque x = 1 e y = 2
    */

    x = 0;
    y = 2 * (x++);
    /*
        verrà prima valutata l'espressione, per poi
        incrementare x, dunque x = 1 e y = 0
    */
}
```

4.5.1 Precedenza degli operatori

Nel linguaggio C, gli operatori delle espressioni vengono valutati con la seguente precedenza (più basso è il numero, maggiore è la precedenza):

Precedenza	Operatore	Descrizione
1	`++ --`	Post-incremento e Post-decremento
	`.`	Accesso attributo di uno struct
	`->`	Accesso attributo di uno struct tramite puntatore
2	`++ --`	Pre-incremento e Pre-decremento
	`!`	NOT bit-wise e logico
	`*`	De-riferimento
	`&`	Riferimento
	`sizeof`	Operatore sizeof
3	`* / %`	Moltiplicazione, divisione e resto
4	`+ -`	Addizione e sottrazione
5	`<<̇ >>̇`	Shift sinistro e destro
6	`< <=`	Minore e minore uguale
	`> >=`	Maggiore e maggiore uguale
7	`== !=`	Uguale e diverso
8	`&`	AND bit-wise
9	`^`	XOR bit-wise
10	`|`	OR bit-wise
11	`&&`	AND logico
12	`||`	OR logico
13	`?:`	Operatore ternario
14	`=`	Assegnamento
	`+= -=`	Somma e differenza con assegnamento
	`*= /= %=`	Prodotto, quoziente e resto con assegnamento
	`<<̇= >>̇=`	Shift sinistro e destro con assegnamento
	`&= ^= |=`	AND, XOR e OR bit-wise con assegnamento

Nota: all'interno della precedente tabella sono stati inseriti anche operatori ancora visti fino a questo punto. Tali operatori verranno discussi nelle sezioni successive.

Esempio:

- Se $x = 1$ e $y = 7$, l'espressione `(x & y) == (x && y)` viene valutata come `true`
- Se $x = 1$ e $y = 7$, l'espressione `x & y | x == y` viene valutata come 1, poiché le sue operazioni vengono valutate come se l'espressione fosse `(x & y) | (x == y)`

4.6 Costrutti condizionali, iterativi e funzioni

4.6.1 Blocchi di istruzioni

Definizione 21: Blocco

Definiamo come **blocco** una porzione di codice racchiusa da delle parentesi graffe (es: `{ ... }`).

Le variabili definite all'interno di un blocco sono **locali** al blocco stesso, ossia:

- Un blocco può contenere dei **sotto-blocchi**
- Le variabili dichiarate in un blocco vengono **allocate sullo stack** e sono **accessibili solo all'interno del blocco stesso e dei suoi sotto-blocchi**
- Una volta che un blocco è terminato, tutte le variabili dichiarate in esso vengono automaticamente **de-allocate** dallo stack

Esempi:

```
void ex1(){
    {
        int i = 0;  //la variabile i è accessibile
        printf("i = %d\n", i);
    }

    //la variabile i non esiste più, generando un errore
    printf("i = %d\n", i);
}

void ex2(){
    int x = -1;

    //la variabile i è accessibile anche dai sotto-blocchi
    {
        printf("i = %d\n", i);  //viene stampato -1
        i++;
    }
    printf("i = %d\n", i);  //viene stampato 0
}
```

Osservazione 28: Oscuramento nei blocchi

Se all'interno di un sotto-blocco viene dichiarata una variabile con lo **stesso nome di una variabile dichiarata in un suo sovra-blocco**, tale variabile viene **oscurata**, ossia il nome farà riferimento alla nuova variabile del sotto-blocco, venendo poi "ricollegato" alla variabile del sovra-blocco una volta che il blocco è concluso

Esempio:

```
void main(){
    int x = -1;

    {
        //la variabile x del sovra-blocco viene oscurata
        int x = 0;

        printf("i = %d\n", i);  //viene stampato 0
    }

    //il nome "x" fa di nuovo riferimento alla
    //variabile del sovra-blocco
    printf("i = %d\n", i);  //viene stampato -1
}
```

4.6.2 Costrutti condizionali

Come ogni altro linguaggio di programmazione, il linguaggio C fornisce il **costrutto condizionale** `if(condition)`. Tale costrutto contiene al suo interno un **blocco di istruzioni**, il quale viene eseguito se e solo se la condizione interna all'if viene valutata come `true`.

Esempio:

```
if(x <= y){
    //istruzioni eseguite se valutato true
}
```

Inoltre, viene fornito anche il costrutto `if/else`, il quale possiede un **aggiuntivo blocco di istruzioni** che viene eseguito se e solo se la condizione viene valutata come `false`.

Esempio:

```
if(x <= y){
    //istruzioni eseguite se valutato true
}
else{
    //istruzioni eseguite se valutato false
}
```

Osservazione 29

Se all'interno di un costrutto `if` o `if/else` non viene specificato un blocco if o un blocco else, la **prima istruzione successiva** verrà considerata come il blocco stesso.

Esempi:

```
//i seguenti costrutti sono tutti equivalenti tra loro

//-----

if(...){
    printf("true");
}
else{
    printf("false");
}

//-----

if(...){
    printf("true");
}
else printf("false");

//-----

if(...) printf("true");
else{
    printf("false");
}

//-----

if(...) printf("true");
else printf("false");

//-----

if(...)
    printf("true");
else
    printf("false");

//-----

if(...) printf("true"); else printf("false");
```

Proposizione 6: Operatore ternario

L'**operatore ternario** restituisce uno **tra due valori indicati** in base ad una **condizione**: se tale condizione viene valutata `true`, verrà restituito il primo, altrimenti il secondo

`(condition) ? value_if_true : value_if_false`

L'operatore ternario può essere utilizzato per effettuare **assegnamenti condizionati** in modo rapido.

Esempio:

```
x = (a < b) ? 1000 : -20;

//è equivalente a

if(a < b){
    x = 1000;
}
else{
    x = -20;
}
```

Osservazione 30

I costrutti condizionali possono essere **concatenati tra loro**

Esempio:

```
if(...){
    //caso 1
}
else if(...){
    //caso 2
}
else if(...){
    //caso 3
}
else{
    //tutti gli altri casi
}
```

In alcune situazioni, tale serie di casistiche dell'esempio precedente può essere realizzata anche tramite il **costrutto** `switch`, il quale, data un'espressione in input, permette di dettare le operazioni da svolgere a seconda del valore restituito da tale espressione

```
switch(value){
    case 1:
        //eseguito se value = 1
        break;
    case 2:
        //eseguito se value = 2
        break;

    ...

    case n:
        //eseguito se value = n
        break;
    default:
        //eseguito in ogni altro caso
        break;
}
```

Di fondamentale importanza all'interno dei costrutti switch risulta essere la keyword `break`, la quale permette di uscire immediatamente dallo switch.

Senza l'uso del break, il costrutto switch prosegue l'esecuzione delle istruzioni fornite in ogni **caso sottostante**.

Esempio:

```
switch(value){
    case 1:
        //eseguito se value = 1
    case 2:
        //eseguito se value = 2 o value = 1
        break;
    default:
        //eseguito in ogni altro caso
        break;
}
```

Osservazione 31

Ogni blocco interno ad un costrutto condizionale corrisponde ad un vero e proprio blocco di istruzioni. Per tanto, per esso valgono le **stesse regole**.

4.6.3 Costrutti iterativi

Il linguaggio C fornisce tre tipi di **costrutti iterativi**:

- Il costrutto `while`, il quale esegue il blocco dato finché la condizione data è vera

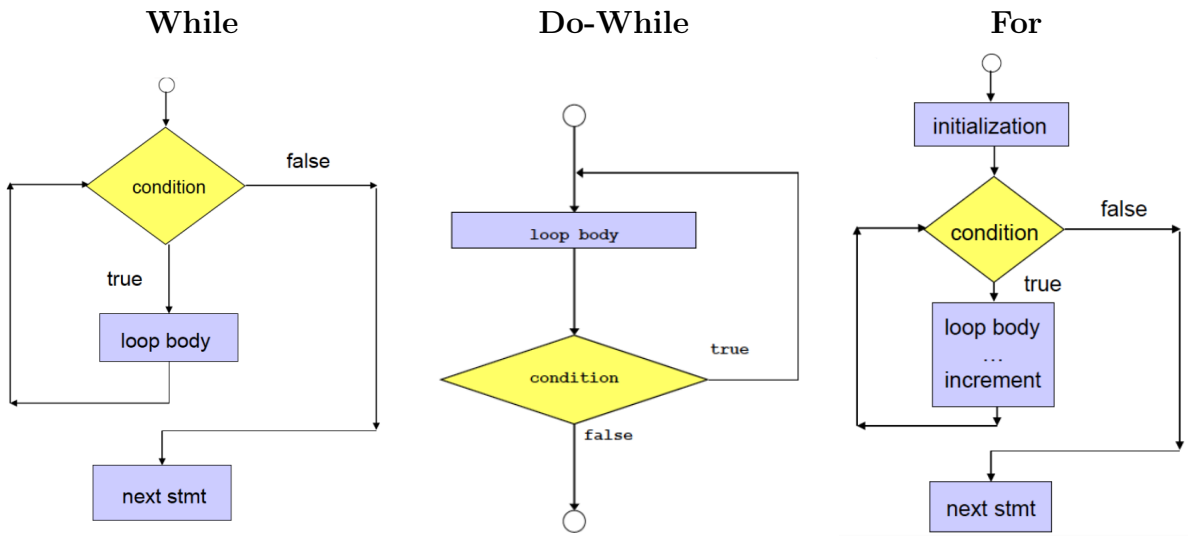
```
while(condition){
    //block
}
```

- Il costrutto `do-while`, il quale esegue almeno una volta il blocco dato, per poi rieseguirlo finché la condizione data è vera

```
do{
    //block
}
while(condition);
```

- Il costrutto `for`, il quale permette di effettuare un'assegnamento o inizializzazione, per poi eseguire il blocco dato finché la condizione data è vera, effettuando (dopo ogni iterazione) l'incremento dato

```
for(initialization; condition; increment){
    //block
}
```



```
//le seguenti funzioni sono equivalenti

void ex1(){
    for(int i=0; i <= 10; i++){
        printf("ciao");
    }
}

void ex2(){
    int i;
    for(i=0; i <= 10; i++){
        printf("ciao");
    }
}

void ex2(){
    int i=0;
    while(i <= 10){
        printf("ciao");
        i++;
    }
}
```

Come per il costrutto `switch`, anche all'interno dei costrutti iterativi può essere utilizzata la keyword `break`, uscendo immediatamente dal costrutto stesso. In aggiunta, all'interno dei costrutti iterativi è possibile utilizzare anche la keyword `continue`, la quale permette di interrompere l'esecuzione del blocco e saltare automaticamente all'iterazione successiva (effettuando sempre il controllo sulla condizione).

Esempio:

```
for(int i=0; i <= 10; i++){
    ...
    break
    //viene direttamente terminato il for
    ...
    //queste istruzioni dopo il break non vengono eseguite
}

for(int i=0; i <= 10; i++){
    ...
    continue
    //si passa all'iterazione successiva
    ...
    //queste istruzioni dopo il continue non vengono eseguite
}
```

Osservazione 32

Ogni blocco interno ad un costrutto iterativo corrisponde ad un vero e proprio blocco di istruzioni. Per tanto, per esso valgono le **stesse regole**.

Inoltre, l'assegnamento effettuato all'interno del campo `initialization` di un ciclo `for`, viene considerato come **interno al blocco**, potenzialmente oscurando le variabili omonime precedenti

4.6.4 Funzioni

Durante la fase di introduzione al linguaggio, abbiamo già discusso brevemente le **funzioni**. Ogni funzione consiste in un'**intestazione (header)**, a sua volta composta dal nome della funzione, dal tipo di valore ritornato e da una lista di parametri in input, e da un **blocco di istruzioni**:

```
<return-type> function-name (parameter-list){
    //block
}
```

In particolare, notiamo che:

- Il blocco di istruzioni di una funzione viene eseguito solo quando tale funzione viene **invocata (o chiamata)**.
- La lista dei parametri può essere anche vuota
- Il tipo di ritorno può essere un qualsiasi tipo di dato (dunque non necessariamente quelli standard del C)
- Il blocco di una funzione segue le regole di ogni altro blocco di istruzioni
- All'interno di una funzione è possibile definire un'altra funzione (la quale sarà invocabile solo all'interno del blocco della funzione stesso)

Esempio:

```
int somma(int a, int b){
    int c = a+b;
    return c;
}

int main(){
    int a = 5;
    int b = 6;
    int c = func(a, b);   //viene ritornato il valore 11
}
```

Definizione 22: Tipo `void`

Il tipo di dato `void` è un **tipo speciale** con caratteristiche diverse dagli altri tipi:

- **Non esiste** un valore di tipo `void`
- Il **tipo di ritorno di una funzione** può essere impostato a `void` in modo che tale funzione non ritorni alcun valore.
- Assegnare un "valore di tipo `void`" (es: il valore "ritornato" da una funzione di tipo `void`) ad una variabile genererà un errore
- **Non può** essere utilizzato come tipo per la **dichiarazione di variabili**
- Un valore **non può essere castato** in `void`

Osservazione 33

Come per le variabili, anche le funzioni possono essere prima **dichiarate** per poi essere `definite`. Per dichiarare una funzione, è sufficiente scriverne l'intestazione. Per definirla, invece, è necessario ri-scrivere la sua intestazione e il seguente blocco.

Esempio:

```
void func(int a, int b);     //dichiarazione della funzione

void func(int a, int b){     //definizione della funzione
    //block
}
```

Osservazione 34

Se una funzione viene invocata prima che essa sia **dichiarata**, verrà generato un errore di compilazione. Se invece una funzione è **dichiarata** e viene invocata prima che essa sia **definita**, verrà generato un errore in fase di linking.

Per tanto, una funzione può essere invocata correttamente solo se essa è **dichiarata e definita**.

Esempio:

```
void func2(){...}
void func3();

void func(int a, int b){
    func2();    //nessun errore (dichiarata e definita)
    func3();    //errore di linking (dichiarata ma non definita)
    func4();    //errore di compilazione (né dichiarata né definita)
}
```

4.7 Puntatori

Definizione 23: Puntatore

Un **puntatore** è una variabile speciale in grado di **contenere un indirizzo di memoria**. La dichiarazione di un puntatore corrisponde a:

```
type* pointer_name
```

dove il campo `type` di un puntatore indica che il valore contenuto dall'indirizzo puntato è da interpretare come una variabile di tipo `type`

Nota: solitamente, per la dichiarazione viene utilizzata anche la sintassi alternativa `type *pointer_name` (la scelta ricade sul programmatore)

Esempio:

```
void main(){
    int x;

    //supponiamo che la variabile x venga
    //salvata nell'indirizzo di memoria 0x1234

    int* pointer = &x;  //pointer = 0x1234

    //ricordiamo che &x è l'operatore di riferimento,
    //il quale  restituisce l'indirizzo di x
}
```

Proposizione 7: Operatore di deriferimento

Il simbolo `*` posto davanti all'identificatore di un puntatore permette di **accedere** il **valore contenuto all'interno dell'indirizzo di memoria puntato**.

(es: se `int x = 4` e `int* p = &x`, allora `*p` restituisce 4)

Tale operatore viene detto **deriferimento**

Definizione 24: Valore diretto o indiretto

Dato il puntatore `type* pointer`, definiamo come **valore diretto** l'indirizzo di memoria contenuto al suo interno, mentre definiamo come **valore indiretto** il valore ottenibile tramite il deriferimento del puntatore stesso.

Osservazione 35

L'accesso ad un indirizzo di memoria tramite il deriferimento permette anche di **modificare** il contenuto di tale indirizzo di memoria

Esempio:

```
void main(){
    int n = 5;
    int* ptr = &n;

    *ptr = 10;
    /*
        viene modificato il valore contenuto
        nell'indirizzo di memoria puntato,
        implicando che ora si abbia n = 10
    */
}
```

Proposizione 8: Aritmetica dei puntatori

Poiché l'indirizzo di memoria contenuto all'interno di un puntatore non è altro che un **valore intero**, su di esso possono essere applicate operazioni aritmetiche.

Dato il puntatore `type* ptr`, l'**unità additiva** utilizzata per le operazioni aritmetiche su di esso sarà `sizeof(type)`.

Esempi:

- Dato il puntatore `int* ptr` contenente l'indirizzo di memoria x, l'espressione `ptr + 1` viene valutata come $x + $ `sizeof(int)` $= x + 4$, restituendo quindi l'indirizzo successivo di 4 byte all'indirizzo x
- Dato il puntatore `int* ptr` contenente l'indirizzo di memoria x, l'espressione `ptr + 6` viene valutata come $x + 6 \cdot$ `sizeof(int)` $= x + 6 \cdot 4 = x + 24$
- Dato il puntatore `char* ptr` contenente l'indirizzo di memoria x, l'espressione `ptr + 1` viene valutata come $x + $ `sizeof(char)` $= x + 1$

4.7.1 Puntatore `void*` e valore `NULL`

Definizione 25: Puntatore `void*`

Essendo il tipo `void` un **tipo speciale** è possibile definire anche un puntatore di tipo `void*`, godendo delle seguenti caratteristiche:

- Ad un puntatore `void*` può essere assegnato l'indirizzo di memoria di una variabile di **qualsiasi tipo** (**puntatore polimorfico**)
- Un puntatore `void*` non può **mai essere dereferenziato**, necessitando prima il casting in un altro tipo

Esempio:

```
void main(){
    int x = 10;
    char y = 'A';
    char* z = &y;

    void* ptr;

    ptr = &x;
    x = *(int*) ptr + 10;   //viene prima castato e poi dereferenziato

    //NOTA: usando più parentesi risulta più comoda la lettura
    ptr = &y;
    y = *((char*) ptr) + 1;

    ptr = &z;
    printf("z = %c", **((char**) ptr));     //stampa "z = B"
}
```

Definizione 26: Macro e Direttiva `#define`

Definiamo come **macro** l'abbreviazione in una parola di un'espressione.

Il linguaggio C permette la creazione di macro tramite la **direttiva** `#define` del pre-processore:

```
#define shortcut full_expression
```

Durante la fase di pre-processo, ogni singolo uso della parola `shortcut` viene **automaticamente rimpiazzato** con `full_expression`.

Esempio:

```
#define u16 unsigned short int

void main(){
    u16 x;  //u16 viene rimpiazzato con unsigned short int
}
```

Proposizione 9: Puntatore `NULL`

Il **puntatore** `NULL` è una macro per l'espressione `(void*) 0`.

Trattandosi di un **puntatore** `void*` **all'indirizzo 0**, esso risulta assegnabile ad ogni tipo di puntatore. L'uso del puntatore `NULL` permette una comprensione agevolata del codice, in quanto tale puntatore venga interpretato come un "segnaposto" indicante che non sia effettivamente rilevante l'indirizzo assunto in quel preciso istante o che semplicemente non si sappia l'indirizzo.

Esempio:

```
void func(char* str){
    if(str == NULL){
        ...
    }
    else{
        ...
    }
}

void main(){
    func(NULL);     //otteniamo un effetto
    func("ciao");   //otteniamo un altro effetto
}
```

Osservazione 36

Nonostante il puntatore `NULL` possa essere castato, dereferenziandolo si andrà incontro ad un **segmentation fault**, ossia un errore di esecuzione dovuto all'accesso ad una zona di memoria non autorizzata al processo.

Difatti, l'indirizzo `0x0` a cui fa riferimento `NULL` è riservato al kernel

4.7.2 Puntatori a funzioni

Il linguaggio C permette anche l'uso di **puntatori a funzioni**, i quali permettono di rendere il codice più generico e flessibile (es: passare una funzione come argomento di un'altra funzione)

Per dichiarare un puntatore a funzione viene usata la seguente sintassi:

```
returned_type (*pointer_name)(arg1_type, arg2_type, ...)
```

Esempi:

```
int add(int a, int b){
    return a+b;
}
int multiply(int a, int b){
    return a*b;
}

int func(int (*f)(int, int)){
    return (*f)(10, 11);    //invoca f(10, 11)
}

void main(){
    //dichiarazione puntatore ad una
    //funzione che prende in input due interi
    int (*funptr)(int,int);

    ...

    if(var == 1){
        funptr = add;
    }
    else{
        funptr = &multiply;
        //utilizzare il riferimento non ha alcun effetto
        //in questo caso, ma potrebbe agevolare la lettura
    }

    (*funptr)(5, 11);   //invoca add(5, 11) o multiply(5, 11)
    func(funcptr);      //passa add o multiply come parametro
}
```

4.8 Array, Stringhe e Struct

4.8.1 Array

Un **array** è un tipo di dato strutturato composto da una **schiera di variabili dello stesso tipo**, le quali occupano una regione contigua di memoria. La **dimensione** di un array, ossia il numero totale di elementi al suo interno, è **statica**, ossia inalterabile.

Proposizione 10: Dichiarazione di array

Per dichiarare un array, viene utilizzata la seguente sintassi:

```
modifiers data_type array_name[number_of_elements]
```

dove i campi `modifiers` e `data_type` specificano il tipo di ogni elemento

Gli elementi di un array sono **indicizzati** da un numero intero compreso tra 0 e $n-1$, dove n è il numero di elementi dell'array.

Esempi:

```
void main(){
    unsigned int arr[3];    //un array di tre unsigned int
    char str[10];           //un array di 10 caratteri

    arr[0] = 0;     //inizializzo il primo elemento dell'array
    arr[1] = 0;     //inizializzo il secondo elemento dell'array
    arr[2] = 0;     //inizializzo il terzo elemento dell'array
}
```

Inoltre, è possibile **"auto-inizializzare"** i valori di un array in fase di dichiarazione:

```
type arr[n] = {val0, val1, ..., val(n-1)}
                \_____n valori_____/
```

Osservazione 37

Quando un array viene dichiarato, gli **elementi non inizializzati** al suo interno conterranno dei **"valori casuali"**, corrispondenti agli ultimi valori presenti nei byte di memoria precedentemente utilizzati da altri processi ed ora occupati dall'array stesso.

Esempio:

```
void main(){
    unsigned int arr[3];
    printf("%d", arr[0]);   //viene stampato un "valore casuale"
}
```

Osservazione 38

È necessario prestare **molta attenzione** all'uso degli indici per accedere agli elementi di un array:

- Se si tenta di **accedere** ad un elemento fuori dal range dei valori dell'array (**Stack Smashing**), verrà generato un errore di esecuzione
- Se si tenta di **printare** un elemento fuori dal range dei valori dell'array, tale operazione verrà concessa

Esempio:

```
void main(){
    int x;
    int arr[3];

    printf("%d", arr[100]);     //nessun errore
    arr[4] = 5;                 //errore
    x = arr[10];                //errore
}
```

Osservazione 39

Dichiarando un array `type arr[n]`, viene creata la variabile `arr`, corrispondente ad un **puntatore al primo elemento** dell'array.

Difatti, la sintassi `arr[m]` per accedere all'$(m+1)$-esimo elemento dell'array corrisponde ad una **contrazione sintattica** dell'operazione `*(arr + m)`

Esempi:

```
void main(){
    int arr[3];

    arr[0] = 0;     //viene modificato il primo elemento dell'array
    *(arr + 0) = 0; //viene modificato il primo elemento dell'array

    printf("%d", arr);
    //viene printato l'indirizzo del primo elemento dell'array

    printf("%d", arr[2]);
    //viene printato il terzo elemento dell'array

    printf("%d", *(arr + 2));
    //viene printato il terzo elemento dell'array
}
```

Osservazione 40

Per utilizzare un array come argomento di una funzione, è strettamente necessario passare tale array per **riferimento**.

Ricordiamo inoltre che la variabile tramite cui è possibile accedere agli elementi dell'array è essa stessa un puntatore. Per tanto, non è necessario utilizzare l'operatore di riferimento su di esso.

Esempio:

```
void func(int* arr){...}

void main(){
    int arr[3];

    func(&arr);
    /*
        verrebbe passato l'indirizzo di memoria in cui
        è salvato il puntatore arr, ma non l'indirizzo
        del primo elemento dell'array (doppio riferimento)
    */

    //l'utilizzo corretto dunque è
    func(arr);
}
```

Osservazione 41

Utilizzando l'operatore `sizeof` sulla variabile puntatore di un array, verrà restituita la somma delle dimensioni di tutti gli elementi.

(es: dato l'array `type arr[n]`, l'operatore `sizeof(arr)` restituirà `n * sizeof(type)`)

Osservazione 42: Array multimensionali

È possibile definire anche un **array multidimensionale**, ossia i cui elementi sono indicizzabili tramite n valori (dove n è il numero di dimensioni).

Un array multidimensionale non è altro che un array contenente altri array, i quali a loro volta contengono altri array e così via fino ad esaurire il numero di dimensioni.

Esempi:

```
void main(){
    int a[10];              //un vettore di 10 elementi
    int a[10][10];          //una matrice 10 x 10
    int a[10][10][10];      //un cubo 10 x 10 x 10

    a[0][0] = 0;    //inizializzo primo elemento della matrice
    a[0][0][0] = 0; //inizializzo primo elemento del cubo
}
```

4.8.2 Stringhe

All'interno del linguaggio C, le **stringhe** vengono rappresentate come un particolare **array di caratteri**, dove:

- Ogni elemento dell'array è un carattere della stringa
- L'ultimo elemento dell'array è un **null byte** (ossia il carattere `\0`), detto anche carattere di fine stringa

Ad esempio, la stringa `"Hello World!"` viene rappresentata come:

H	e	l	l	o		W	o	r	l	d	!	\0

e il suo **placeholder** all'interno delle stringhe formattate corrisponde a `%s`.

Per **assegnare** una stringa ad un array, dunque, è sufficiente trattare tale stringa come se fosse un normale valore:

```
void main(){
    char str[20] = "Hello World!";
    printf("%s", str);
}
```

Osservazione 43

Quando si vuole assegnare una stringa ad un array di caratteri, è necessario tenere a mente che tale stringa contenga anche **carattere di fine stringa**, implicando che, se la stringa è lunga n caratteri, l'array debba avere una **dimensione** pari ad almeno $n + 1$.

Attenzione: nel caso in cui l'array abbia una dimensione troppo piccola, non verrà generato un errore, ma verrà segnalato solamente un warning in fase di compilazione

Esempio:

```
void main(){
    char str[5] = "ciao";   //ok

    char str2[4] = "ciao";
    //l'array è troppo piccolo, dunque il carattere
    //'\0' viene tagliato durante l'assegnamento

    char str3[3] = "ciao";
    //l'array è troppo piccolo, dunque i caratteri
    //'o' e '\0' vengono tagliati durante l'assegnamento
}
```

Il linguaggio prevede, inoltre, multiple modalità per **inizializzare** una stringa:

```
void main(){
    char str[10] = "Ciao";
    //i primi 5 caratteri vengono occupati dalla stringa,
    //mentre i restanti 5 assumono un "valore casuale"

    char str2[10] = {'C', 'i', 'a', 'o', '\0'};
    //equivalente all'istruzione precedente

    char str3[] = "Ciao";
    //l'array assume automaticamente la dimensione della stringa,
    //dunque non vi sono spazi in eccesso
}
```

La libreria `<string.h>` fornisce le seguenti funzioni per lavorare con le stringhe:

- La funzione `size_t strlen(const char* str)` permette di ottenere la lunghezza di una stringa (escludendo il null byte dal conteggio). Il tipo `size_t` ritornato è una macro per il tipo `int`.
- La funzione `char* strcpy(char* dest, const char* src)` copia all'interno del buffer `dest` il contenuto del buffer `src`. Il puntatore ritornato è `dest`. Per evitare errori, è necessario che il buffer `dest` abbia una dimensione maggiore o uguale a quella di `src`.
- La funzione `char* strcat(char* dest, const char* src)` appende il contenuto del buffer `src` alla fine del buffer `dest` (usando il null byte di `dest` come punto di concatenazione). Il puntatore ritornato è `dest`. Per evitare errori, è necessario che il buffer `dest` abbia una dimensione "rimanente" maggiore o uguale a quella di `src`.
- La funzione `int strcmp (const char* str1, const char* str2 )` permette di comparare due stringhe, ritornando la differenza tra i primi due caratteri (uno di `str1` e uno di `str2`) diversi

  - Ritorna un valore < 0 se il primo carattere diverso con valore minore (ricordiamo che i caratteri ASCII sono comunque un intero) è quello di `str1`
  - Ritorna 0 se non esistono due caratteri diversi (dunque le stringhe sono uguali)
  - Ritorna un valore > 0 se il primo carattere diverso con valore minore è quello di `str2`

- La funzione `char* strncpy(char* dest, const char *src, size_t n)` risulta analoga alla funzione `strcpy()`, con la differenza che vengano copiati massimo `n` caratteri
- La funzione `char* strncat(char* dest, const char *src, size_t n)` risulta analoga alla funzione `strcat()`, con la differenza che vengano appesi massimo `n` caratteri
- La funzione `char* strncmp(char* dest, const char *src, size_t n)` risulta analoga alla funzione `strcmp()`, con la differenza che vengano comparati massimo `n` caratteri

Osservazione 44

Comparare due stringhe `str1` e `str2` tramite l'operatore `str1 == str2` risulta **errato**, poiché il tal modo stanno venendo comparati i puntatori di tali stringhe

All'interno della libreria `<stdio.h>` vengono inoltre fornite le seguenti funzioni più flessibili (e più sicure) per scrivere o leggere stringhe:

- La funzione `int putchar(int char)` scrive il carattere dato su stdout
- La funzione `int puts(const char* str)` scrive la stringa data su stdout
- La funzione `int fputs(const char* str, FILE* stream)` scrive la stringa data su file `stream` (vedere sezioni successive)
- La funzione `char* gets(char* str)` legge una stringa da stdio e la scrive nel buffer `str`, ritornando `str` stesso come puntatore. Per motivi di sicurezza, tale funzione è stata **deprecata** e non è disponibile nelle versioni moderne di C
- La funzione `char* fgets(char* str, int n, FILE* stream)` legge n caratteri dal file `stream` e scrive nel buffer `str`, ritornando `str` stesso come puntatore
- La funzione `int getchar()` legge un carattere da stdin e ritorna il valore decimale di tale carattere

Osservazione 45

Il **carattere di fine stringa** risulta fondamentale, poiché non si è in grado di poter sapere quando la stringa sia terminata, portando ad un **undefined behaviour** (trad: *comportamento sconosciuto*)

Ad esempio di ciò, possiamo analizzare il comportamento della funzione `strlen()`:

- Partendo dal primo elemento della stringa, la funzione scorre la lista contando il numero di caratteri letti e fermandosi solo quando viene letto un carattere `\0`
- Per tanto, se il carattere `\0` è assente, la funzione potrebbe continuare potenzialmente all'infinito

4.8.3 Struct

Definizione 27: Struct

Uno **struct (structure)** è un tipo di dato **definito dall'utente** e **composto da più campi di diversi tipi di dato**.

Uno struct può essere visto come un contenitore di una collezione di variabili di vari tipi, utilizzabile pienamente come un tipo di dato.

Per definire uno struct, possono essere utilizzate **tre modalità**:

- **Variabile struct**, utilizzabile per definire tutte le variabili che assumeranno la struttura definita

```
void main(){
    struct {
        type1 attr1;
        type2 attr2;
        ...
    } var1, var2, ...;
}
```

- **Tagged struct**, utilizzabile per conservare la struttura definita e utilizzarla tra più file

```
struct struct_name{
    type1 attr1;
    type2 attr2;
    ...
};

void main(){
    struct struct_name var1, var2, ...;
}
```

- **Type-defined struct** tramite l'operatore `typedef` utilizzabile per definire direttamente un nuovo tipo di dato (può essere utilizzato anche con altri tipi, non necessariamente uno struct)

```
typedef struct struct_name{
    type1 attr1;
    type2 attr2;
    ...
} type_name;

//NOTA: solitamente viene aggiunto "_t"
//alla fine del nome per indicare che si
//tratti di un tipo definito dall'utente

void main(){
    type_name var1, var2, ...;
}
```

Come per gli array, è possibile effettuare un **"auto-inizializzazione"** dei campi di uno struct:

$$\texttt{struct\_type var} = \underbrace{\texttt{\{val0, val1, ..., val(n-1)\}}}_{\texttt{n valori}}$$

Proposizione 11: Operatore punto

L'**operatore punto**

`struct_var.attr_name`

permette di **accedere** all'attributo `attr_name` dello struct `struct_var`.

Esempio:

```
typedef struct point2D{
    float x;
    float y;
} point2D_t;

void main(){
    point2D_t point;
    point.x = 43.16;
    point.y = 13.12;
}
```

Proposizione 12: Operatore freccia

L'**operatore freccia**, ossia

`struct_p->attr_name`

permette di **dereferenziare** il puntatore `struct_p` per poi **accedere** all'attributo `attr_name` dello struct da esso puntato.

Difatti, l'operatore `struct_p->attr_name` e lo statement `(*struct_p).attr_name` sono equivalenti tra loro.

Esempio:

```
typedef struct point2D{
    float x;
    float y;
} point2D_t;

void main(){
    point2D_t point;
    point2D_t* point_ptr = &point;

    point_ptr->x = 43.16;
    point_ptr->y = 13.12;
}
```

Osservazione 46

Utilizzando l'operatore `sizeof` su un type-defined struct, verrà restituita la somma delle dimensioni di tutti i campi del tipo struct definito.

Esempio:

```
typedef struct point2D{
    float x;
    float y;
} point2D_t;

void main(){
    printf("%d", sizeof(point2D_t));    //viene printato 32+32 = 64
}
```

4.9 Utilizzo della memoria dinamica

Fino ad ora, abbiamo visto come i blocchi di istruzioni allochino e deallochino automaticamente la memoria sullo **stack** durante la compilazione.

Per quanto riguarda la **memoria dinamica**, ossia la porzione di memoria riservata all'heap, l'allocazione e la deallocazione è a carico del programmatore. In particolare, è **fondamentale** che, qual'ora si sia allocata della memoria nell'heap, essa venga **liberata** (ossia deallocata), poiché altrimenti essa rimarrà "occupata" (**memory leakage**) fino a che il processo non terminerà e tutte le sue pagine verranno deallocate.

La presenza di memory leakage influisce molto sulle prestazioni di un processo e anche dell'intero sistema operativo. Per tanto, è necessario mantenere al mimino possibile (potenzialmente zero) la quantità di memoria "sprecata".

La libreria `<stdlib.h>` fornisce le seguenti funzioni per l'**allocazione e deallocazione di memoria nell'heap**:

- La funzione `void* malloc(size_t size)` alloca una porzione di memoria pari a `size` byte, restituendo un puntatore `void*` all'inizio di tale porzione.

  Come per l'allocazione sullo stack, i byte allocati conterranno i valori precedenti. Se l'allocazione fallisce, viene ritornato `NULL`.

  ```
  void main(){
      //vengono allocati 4 byte
      int* x = (int*) malloc(sizeof(int));

      //viene stampato un "valore casuale",
      //poiché la memoria non è inizializzata
      printf("%d", *x);

      //vengono allocati 8 byte
      char* str = (char*) malloc(sizeof(char) * 8);
      str = "Ciao";

      //vengono allocati 16 byte (dunque 4 interi)
      int* arr = (int*) malloc(16);
  }
  ```

- La funzione `void* calloc(size_t num, size_t size)` alloca `num` porzioni contigue di memoria, ciascuna pari a `size` byte, restituendo un puntatore `void*` all'inizio della prima porzione.

  Dopo l'allocamento, ogni byte della porzione di memoria viene **inizializzato con valore 0**. Se l'allocazione fallisce, viene ritornato `NULL`

```
void main(){
    //viene allocato un intero (dunque 4 byte)
    int* x = (int*) calloc(1, sizeof(int));

    //viene stampato 0, poiché la memoria è inizializzata
    printf("%d", *x);

    //vengono allocati 8 byte
    char* str = (char*) calloc(8, sizeof(char));
    str = "Ciao";

    //vengono allocati 16 byte
    int* arr = (int*) malloc(4, 4);
}
```

- La funzione `void* realloc(void* ptr, size_t size)` modifica la porzione di memoria precedentemente ed accessibile dal puntatore `ptr`:
  - Se il puntatore dato in input è `NULL`, la funzione avrà lo stesso comportamento di `malloc(size)`
  - Se possibile, la precedente porzione di memoria viene **direttamente estesa** fino a raggiungere la dimensione `size` e il puntatore ritornato **coincide** con quello dato in input
  - Se non è possibile estendere la porzione precedente, viene allocata una **nuova porzione di memoria** di dimensione `size`, **copiando** della porzione precedente e **deallocando** quest'ultima. Il puntatore ritornato sarà quello all'inizio della nuova area di memoria

  Come per l'allocazione sullo stack, i byte allocati conterranno i valori precedenti (esclusi i byte su cui è stato copiato il contenuto). Se l'allocazione fallisce, viene ritornato `NULL`.

```
void main(){
    int* x = (int*) malloc(sizeof(int));

    ...

    //il contenuto di x viene riallocato
    int* y = realloc(x, sizeof(int) * 3);

    int* z = realloc(NULL, 8); //equivale a malloc(8)
}
```

- La funzione `free(void* ptr)` libera la porzione di memoria puntata da `ptr`.

  Se il puntatore dato in input non è stato ottenuto tramite una chiamata alle funzioni `malloc`, `calloc` e `realloc`, la funzione andrà in **undefined behaviour**, solitamente generando un segmentation fault.

  Se viene invocata la funzione `free()` su una porzione di memoria già deallocata (es: utilizzando due volte `free()` sullo stesso puntatore), la funzione andrà in **undefined behaviour**, solitamente generando un segmentation fault (**double free**)

```
void main(){
    int* x = (int*) malloc(sizeof(int));

    ...

    free(x);    //deallocata

    free(x);    //undefined behaviour (double free)

    int y[10];
    free(y)     //undefined behaviour
}
```

Nel caso in cui si volesse **allocare memoria sullo stack** invece che l'heap, ma senza dover dichiarare necessariamente una variabile, la libreria `<alloca.h>` fornisce la funzione `void* alloca(size_t size)`, avente lo stesso funzionamento di `malloc()` (ma con allocazione sullo stack).

Osservazione 47

Essendo l'allocazione da parte di `alloca()` effettuata sullo stack, non è necessario eseguire `free()` in quanto tale memoria viene automaticamente deallocata alla chiusura del blocco

Nonostante non siano strettamente legate alle stringhe, la libreria `<string.h>` fornisce le seguenti due funzioni utili per l'assegnamento rapido in porzioni di memoria:

- La funzione `void* memset(void* dest, int value, size_t num)` assegna l'intero `value` a `num` byte contingui partendo dalla porzione puntata da `dest`
- La funzione `void* memcpy(void* dest, const void* src, size_t num)` copia `num` byte contingui dall'inizio della porzione puntata da `src` all'inizio della porzione puntata da `dest`

4.10 Utilizzo dei file

Ogni **file** su cui è possibile operare appartiene ad una delle seguenti categorie:

- **File di testo**, ossia contenente puro testo
- **File binari**, ossia file eseguibili, compilati, oggetto, ...
- **Buffer**, ossia dei file speciali corrispondenti ad aree di memorizzazione temporanee che mantengono i dati da trasferire dalla memoria principale al disco o viceversa

Ogni **riga** di un file di testo è terminata dal carattere `\n`, mentre l'intero file è terminato dal valore speciale **End-of-File**, definito come `EOF` in C. Ogni file aperto, inoltre, possiede un **cursore**, ossia un puntatore al suo contenuto che tiene traccia della posizione attuale di lettura e/o scrittura.

Per poter **utilizzare i file** all'interno del linguaggio C, la libreria `<stdio.h>` permette di eseguire (in ordine) le seguenti operazioni:

1. Dichiarare un **file pointer**, ossia `FILE*`
2. Aprire il file desiderato tramite la funzione `FILE* fopen(const char* filename, const char* mode)`. Tale funzione apre il file con in modalità `mode`, crea uno struct `FILE` e restituisce un puntatore `FILE*`. Viene ritornato `NULL` se l'apertura del file fallisce.
3. Utilizzare le funzioni di scrittura e/o lettura (vedi più avanti)
4. Chiudere il file tramite la funzione `int fclose(FILE* stream)`, la quale ritorna 0 se la chiusura è avvenuta con successo, altrimenti ritorna `EOF`
5. Liberare il puntatore `FILE*` tramite `free()`

Quando un processo effettua una lettura o scrittura su un file aperto e associato ad un `FILE*`, il contenuto utilizzato da tali operazioni inserito momentaneamente all'interno di un **buffer** presente all'interno dello struct `FILE`.

Ad ogni operazione di scrittura invocata, il contenuto da scrivere viene inserito all'interno del buffer, effettuando la vera operazione di scrittura solo quando il buffer viene riempito o il file viene chiuso **bufferized writing**. Analogamente, l'operazione di lettura ha lo stesso funzionamento, ma in ordine inverso (**bufferized reading**)

In particolare, lo struct `FILE` contiene campi relativi a:

- Il **file descriptor** associato al file
- Il **puntatore al buffer** per lo stream dati
- La dimensione del buffer
- Un contatore per il numero di caratteri attualmente nel buffer
- Una flag di errore

Tra le varie funzioni di scrittura e/o lettura fornite da `<stdio.h>` troviamo:

- La funzione `int fprintf(FILE* stream, const char* format, ...)`, del tutto analoga alla funzione `printf()` ma la scrittura avviene sul file
- La funzione `int fscanf(FILE* stream, const char* format, ...)`, del tutto analoga alla funzione `scanf()` ma la lettura avviene dal file
- La funzione `size_t fread(void* ptr, size_t size, size_t count, FILE* stream)` legge `count` elementi contigui dal file dato, ognuno di dimensione `size`, copiando il contenuto letto nel buffer `ptr`
- La funzione `size_t fwrite(void* ptr, size_t size, size_t count, FILE* stream)` scrive `count` elementi contigui dal buffer `ptr`, ognuno di dimensione `size`, copiando il contenuto letto nel file dato.
- La funzione `char* fgets(char* str, int size, FILE* stream)` legge massimo `size` caratteri dal file (fermandosi prima se viene letto `\n`) e copiando il contenuto letto nella stringa `str`
- La funzione `char* fputs(char* str, FILE* stream)` scrive la stringa `str` sul file
- La funzione `int feof(FILE* stream)` restituisce un valore diverso da 0 se è stato raggiunto `EOF`
- La funzione `int fseek(FILE* stream, long int offset, int whence)` imposta il cursore alla posizione `whence` incrementata di `offset` byte. I valori impostabili per il campo `whence` sono:
  - `SEEK_SET`, ossia l'inizio del file
  - `SEEK_CUR`, ossia la posizione attuale del cursore
  - `SEEK_END`, ossia la fine del file
- La funzione `long int ftell(FILE* stream)` restituisce la posizione attuale del cursore del file se essa è valida, -1 altrimenti
- La funzione `void rewind(FILE* stream)` riporta il cursore del file all'inizio, dunque ha lo stesso effetto di `fseek(stream, 0, SEEK_SET)`

Esempio:

```
void main(){
    FILE* file = fopen("test.txt", "w");        //apertura
    fprintf(file, "Hello World! :)");              //scrittura

    rewind(file);   //riporta il cursore all'inizio
    fprintf(file, "Ciao Mondo!");
    //viene sovrascritto parte del contenuto già scritto

    fclose(file);   //chiusura
    free(file);
}
```

Osservazione 48

All'interno della libreria `<stdio.h>` vengono forniti **tre file pointer** `stdin`, `stdout` e `stderr` relativi ai **tre canali standard** di ogni processo.

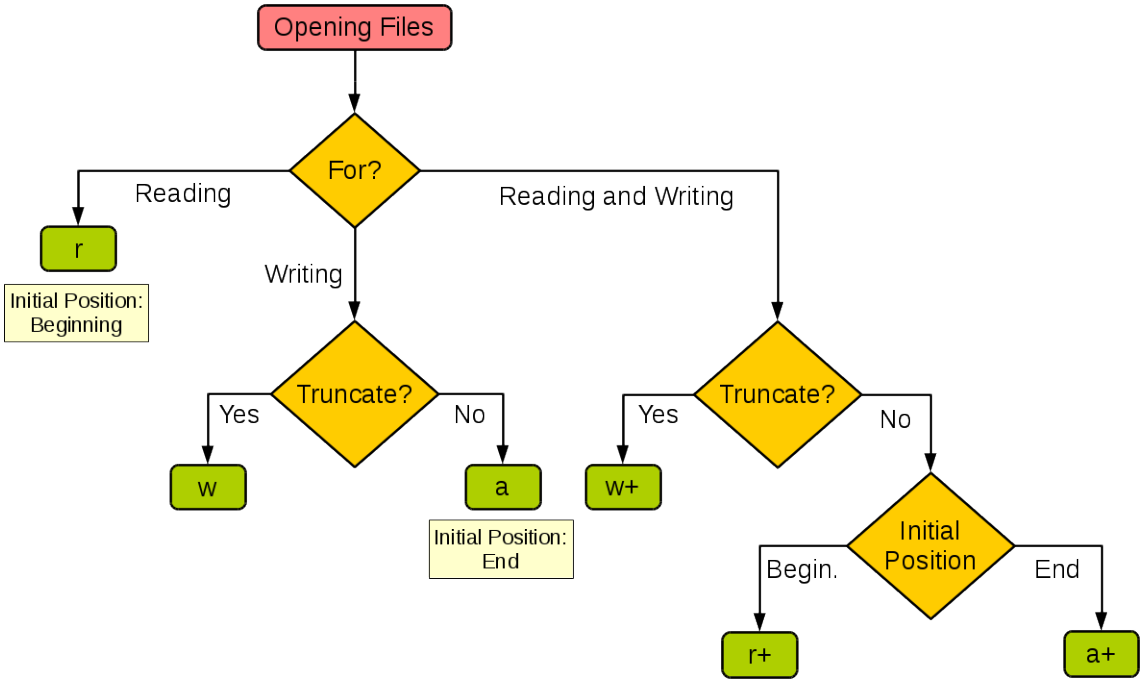
Esempio:

```
#include <stdio.h>

void main(){
    int x;
    fscanf(stdin, "%d", &x);    //è equivalente a scanf("%d", &x);

    fprintf(stdout, "Ciao");    //è equivalente a printf("Ciao");

    fprintf(stderr, "Questo è un messaggio di errore");
}
```

4.10.1 Modalità di apertura di un file

Abbiamo visto come la funzione `fopen()` possieda un parametro `mode` tramite cui è possibile definire la modalità di apertura del file. Tale modalità di apertura definisce le **operazioni eseguibili sul file**, fungendo come una sorta di "restrizione":

- Aprendo il file con la modalità `"r"`, sarà possibile effettuare solo operazioni di **lettura**. La posizione iniziale del cursore del file corrisponderà all'**inizio del file**.
- Aprendo il file con la modalità `"w"`, sarà possibile effettuare solo operazioni di **scrittura**. La posizione iniziale del cursore del file corrisponderà all'**inizio del file**. Il contenuto precedente del file verrà **troncato**. Inoltre, se il file dato non esiste, esso verrà **creato**.
- Aprendo il file con la modalità `"a"`, sarà possibile effettuare solo operazioni di **scrittura**. La posizione iniziale del cursore del file corrisponderà alla **fine del file**. Inoltre, se il file dato non esiste, esso verrà **creato**.
- Aprendo il file con la modalità `"r+"`, sarà possibile effettuare sia operazioni di **lettura** che di **scrittura**. La posizione iniziale del cursore del file corrisponderà all'**inizio del file**. Inoltre, se il file dato non esiste, esso verrà **creato**.
- Aprendo il file con la modalità `"w+"`, sarà possibile effettuare sia operazioni di **lettura** che di **scrittura**. La posizione iniziale del cursore del file corrisponderà all'**inizio del file**. Il contenuto precedente del file verrà **troncato**. Inoltre, se il file dato non esiste, esso verrà **creato**.
- Aprendo il file con la modalità `"a+"`, sarà possibile effettuare sia operazioni di lettura che di scrittura. La posizione iniziale del cursore del file corrisponderà alla **fine del file**. Inoltre, se il file dato non esiste, esso verrà **creato**.



Osservazione 49

Per poter effettuare le operazioni dettate dalla modalità di apertura, è comunque necessario possedere i corretti **privilegi di sistema** per il file richiesto.

(es: un file aperto con `"w"` richiede comunque che l'utente abbia il permesso `w` per il file dato)

Osservazione 50

Un file possiede **un solo cursore**. Per tanto, se un file viene aperto in lettura e scrittura, tali operazioni condivideranno il cursore.

4.11 Variabili esterne e statiche

Come già discusso, nel linguaggio C una variabile è **accessibile** solo all'interno del blocco in cui è dichiarata e nei suoi sotto-blocchi. La porzione di codice in cui una variabile è accessibile viene detto **scoping (o visibilità)** della variabile.

Una funzione avente uno scoping pari all'**intero file sorgente** è detta **variabile globale**, mentre qualsiasi altra variabile è detta **locale**.

Per creare una variabile globale, è sufficiente dichiarare tale variabile **prima e al di fuori di ogni funzione**. Difatti, possiamo considerare anche il file sorgente stesso come un **"enorme blocco radice"** all'interno di cui vengono definiti tutti gli altri blocchi.

Esempio:

```
int var;

//tale variabile è accessibile ovunque
//al di sotto di tale istruzione (scoping globale)

void func(){
    var = 1;
}

int var2;
//tale variabile è accessibile ovunque
//al di sotto di tale istruzione, dunque non è
//accessibile tramite func(), ma tramite main() sì

void main(){
    var = 0;
    var2 = 1;
}
```

Proposizione 13: Modificatore `extern`

Il **modificatore** `extern` permette di "dichiarare" (o meglio richiamare) una variabile non visibile all'interno di un blocco (**external linkage**).

In particolare, tramite il modificatore extern è possibile utilizzare variabili **dichiarate dopo un blocco** o anche **in un altro file**

Esempio:

```
//Supponiamo che la variabile
//  int var2 = 10;
//sia dichiarata in un altro file importato

void func1() {
    extern int var;
    //richiamo di variabile esterna
    //(var non è ancora stata dichiarata)

    var++;
    printf("var = %d\n", var);  //viene printato 1

    void nestfunc1() {
        var++;
        printf("var = %d\n", var);
    }
```

```
    nestfunc1();    //viene printato 2
    nestfunc1();    //viene printato 3
}

int var = 0;  //dichiarazione di var

void main(void) {
    printf("var = %d\n", var);      //viene printato 0
    func1();
    printf("var = %d\n", var);      //viene printato 3

    extern int var2;
    printf("var2 = %d\n", var2);    //viene printato 10
}
```

Proposizione 14: Modificatore `static` su variabili

Il **modificatore** `static` permette di dichiarare variabili il cui valore viene **mantenuto** tra una chiamata e l'altra di una funzione, costituendo una sorta di *memoria privata* della funzione (**internal linkage**).

Inoltre, le variabili statiche vengono **automticamente inizializzate a 0**.

Esempio:

```
void func(){
    static int var;     //auto-inizializzata a 0
    var++;
    printf("%d", var);
}

void main(){
    func();     //viene printato 1
    func();     //viene printato 2
}
```

Osservazione 51

Non è possibile utilizzare il modificatore `static` anche assieme al modificatore `extern`, poiché si otterrebbe un linkage sia esterno che interno.

Proposizione 15: Modificatore `static` su funzioni

Il **modificatore** `static` permette di dichiarare funzioni che possono essere accedute solo ed esclusivamente all'interno del file sorgente in cui sono dichiarate (**internal linkage**).

```
----- File: file1.c
static void func(){...}

---- File: file2.c
#include "file1.c"

void main(){
    func();
    //viene generato errore di compilazione,
    //poiché la funzione non è accedibile
}
```

Osservazione 52

L'uso del modificatore `static` su funzioni permette di creare funzioni all'interno di un file che importa un altro file contenente una funzione con lo stesso nome

Esempio:

```
----- File: file1.c
static void func(){
    printf("Vengo da file1.c")
}

---- File: file2.c
#include "file1.c"

void func(){
    printf("Vengo da file2.c")
}

void main(){
    func();     //viene printato "Vengo da file2.c"
}
```

4.12 Uso avanzato del linguaggio

4.12.1 Inserire parametri all'avvio

Nei capitoli precedenti, abbiamo visto come durante l'avvio di un programma sia possibile inserire dei **parametri** (opzionali e non). All'interno del linguaggio C, tali parametri sono ottenibili **direttamente tramite la funzione main**.

Difatti, essa è dotata di **due parametri opzionali**:

- L'intero `int argc`, il quale permette di sapere la quantità di parametri che sono stati passati alla funzione
- L'array `char* argv[]`, contenente i parametri passati (i quali vengono passati come stringhe). Il primo elemento di tale vettore è il nome del programma eseguito e tale elemento è sempre presente.

Esempio:

- Supponiamo di avviare il nostro programma tramite il comando `program 5 9 ciao`
- L'intero `argc` sarà pari a 4
- L'array `argv` sarà pari a `{"5", "9", "ciao"}`
- Per ottenere tali valori, ci basterà aggiungere gli argomenti opzionali:

```
void main(int argc, char* argv[]){
    printf("argc = %d", argc);
    printf("Il nome del programma è: %s", argv[0]);
    printf("Il primo argomento è: %s", argv[1]);
}
```

Poiché i parametri vengono passati come stringhe, se necessario si dovrà **convertire in intero** le stringhe contenenti numeri. La libreria `<stdlib.h>` fornisce la funzione `int atoi(char* str)`, la quale converte la stringa `str` nell'intero rappresentato da essa (es: `atoi("542")` restituisce `542`).

Tuttavia, è necessario notare che la funzione `atoi()` non effettui alcun controllo sull'input dato. Difatti, essa interpreta ogni carattere come valore intero, sottraendo ad esso il valore intero del carattere `'0'`.

- Il valore intero del carattere `'5'` corrisponde a 53
- Per ottenere l'intero rappresentato dal carattere, al suo valore intero viene sottratto il valore intero del carattere `'0'`, ossia 48, dunque $53 - 48 = 5$
- Per tanto, essendo il valore intero del carattere `'A'` pari a 65, passando tale carattere alla funzione `atoi` il risultato ottenuto sarebbe $65 - 48 = 17$

Per tanto, si consiglia di effettuare i necessari controlli sulla stringa data in input al fine di evitare problematiche.

4.12.2 Makefile

Per **automatizzare la compilazione** di file sorgenti, viene utilizzato il meccanismo del **Makefile (o Make)**, il quale definisce un **linguaggio proprio** per descrivere relazioni tra file sorgenti, file oggetto ed eseguibili all'interno di un file.

Oltre alla maggiore comodità rispetto alla compilazione manuale, tale meccanismo permette di ri-compilare un file sorgente solo quando esso è stato modificato, riducendo notevolmente i tempi di compilazione per programmi molto grandi.

Per compilare un programma tramite il meccanismo makefile viene utilizzato il comando `make`. Se chiamato senza opzioni aggiuntive, tale comando cerca all'interno della directory corrente un file chiamato `GNUmakefile`, `makefile` o `Makefile`, per poi avviare l'esecuzione utilizzando le relazioni dettate da tale file. Tramite l'opzione `[-y filename]`, invece, è possibile specificare un file avente un nome diverso dai tre precedenti.

Le **regole** dettate all'interno di un file make possiedono la seguente struttura:

```
Target: Prerequisites
    Recipe
    Recipe
    ...
    Recipe
```

dove:

- `Target` è tipicamente un file. Possono esserci più `Target` all'interno dello stesso makefile
- `Prerequisites` è una lista di file
- `Recipe` è un comando da eseguire
- Il carattere `<TAB>` è obbligatorio prima di ogni `Recipe` (indentazione forzata)
- Ogni `Recipe` associato ad un `Target` viene eseguito quando:
  - All'interno della directory in cui `make` viene invocato non esiste un file di nome `Target` e tutti i file in `Prerequisites` esistono
  - Il file `Target` esiste ma uno o più file in `Prerequisites` sono stati aggiornati dopo la creazione del file `Target`
- È possibile **specificare il** `Target` da eseguire dandolo come argomento al comando `make` (es: `make my_target`). Se non viene specificato alcun target, verrà eseguito **solo il primo** `Target`
- Il carattere \ alla fine di una riga permette di continuare il contenuto di tale riga anche nella riga successiva

Esempio:

```
merge_sorted_lists: merge_sorted_lists.c
    gcc -Wall -Wextra -O3 merge_sorted_lists.c -o merge_sorted_lists
```

Proposizione 16: Phony target

Un `Target` senza prerequisiti è detto **azione** o **phony target (target fasullo)**.

Ogni phony target va esplicitamente dichiarato tramite la direttiva `.PHONY target1, target2, ....`

In caso contrario, se venisse creato un file con lo stesso nome di un phony target, tale target non verrebbe mai eseguito. Difatti, un target senza prerequisiti e per cui esiste un file con quel nome viene sempre considerato come aggiornato

Esempio:

- Consideriamo il seguente makefile

```
.PHONY: clean

merge_sorted_lists: merge_sorted_lists.c
    gcc -Wall -Wextra -O3 merge_sorted_lists.c \
                          -o merge_sorted_lists

sort_file_int: sort_file_int.c
    gcc -Wall -Wextra -O3 sort_file_int.c \
                          -o sort_file_int

clean:
    rm -f *.o merge_sorted_lists
```

- Per eseguire il target `merge_sorted_list`, possiamo eseguire il comando `make merge_sorted_list` o anche solo il comando `make`, poiché si tratta del primo target
- Per eseguire i target `sort_file_int` e `clean`, possiamo eseguire rispettivamente i comandi `make sort_file_int` e `make clean`

Osservazione 53

Nel caso in cui si voglia **eseguire automaticamente tutti i target**, solitamente viene creato target chiamato `all` i cui prerequisiti sono tutti i target.

Ponendo tale target come primo della lista, inoltre, sarà possibile eseguire tutti i target utilizzando semplicemente il comando `make`

Esempio:

```
.PHONY: clean

all: merge_sorted_lists sort_file_int

merge_sorted_lists: merge_sorted_lists.c
```

```
    gcc -Wall -Wextra -O3 merge_sorted_lists.c \
                            -o merge_sorted_lists

sort_file_int: sort_file_int.c
    gcc -Wall -Wextra -O3 sort_file_int.c \
                            -o sort_file_int

clean:
    rm -f *.o merge_sorted_lists
```

Il linguaggio Make permette inoltre la definizione di **variabili**. È possibile accedere in lettura al contenuto di una variabile tramite la direttiva `$(VAR)`. Per quanto riguarda l'assegnamento, invece, esso può essere effettuato in numerevoli modi:

- La direttiva `VAR := val` prevede l'espansione completa della variabile `VAR` nel momento in cui si accede al suo contenuto in lettura
- La direttiva `VAR = val` prevede solo l'espansione testuale della variabile `VAR` nel momento in cui si accede al suo contenuto in lettura
- La direttiva `VAR ?= val` assegna alla variabile `VAR` il valore `val` solo se `VAR` è vuota
- La direttiva `VAR += val` concatena `val` al contenuto di `VAR`

Esempio:

```
CC := gcc
CFLAGS := -Wall -Wextra -O3

.PHONY: clean

all: merge_sorted_lists sort_file_int

merge_sorted_lists: merge_sorted_lists.c
    $(CC) $(CFLAGS) merge_sorted_lists.c \
                            -o merge_sorted_lists

sort_file_int: sort_file_int.c
    $(CC) $(CFLAGS) sort_file_int.c \
                            -o sort_file_int

clean:
    rm -f *.o merge_sorted_lists
```

Proposizione 17: Variabili automatiche

Ogni `Target` possiede delle **variabili automatiche** utilizzabili solo all'interno dei suoi `Recipe`:

- La variabile `$^` contiene la lista dei file in `Prerequisites` del `Target`
- La variabile `$<` contiene il primo file in `Prerequisites` del `Target`
- La variabile `$@` contiene il nome del `Target`

Esempio:

```
CC := gcc
CFLAGS := -Wall -Wextra -O3
PROGS := merge_sorted_list sort_file_int

all: $(PROGS)

merge_sorted_lists: merge_sorted_lists.c
$(CC) $(CFLAGS) $^ -o $@

sort_file_int: sort_file_int.c
$(CC) $(CFLAGS) $^ -o $@

.PHONY: clean
clean:
    rm -f *.o merge_sorted_lists
```

Osservazione 54: Catene di dipendenze

Se all'interno dei `Prerequisites` del `Target` che si vuole eseguire è presente un file per cui a sua volta è definito un altro `Target` e tale file non esiste, verrà eseguito **prima il secondo `Target`,** per poi eseguire il `Target` richiesto.

In tal modo, è possibile definire **catene di dipendenze** tra i target (es: sfruttando la precompilazione)

Esempio:

```
CC := gcc
CFLAGS := -Wall -Wextra -O3

program: file.o
    $(CC) $(CFLAGS) $^ -o $@

file.o: file.c
    $(CC) $(CFLAGS) $< -o $@
```

Inoltre, il linguaggio make prevede alcune **regole e variabili implicite**, ossia un insieme di regole e variabili che ricoprono casi comuni per la generazione di un eseguibile a partire da un sorgente C. Tali regole implicite permettono di scrivere regole senza doverne scrivere i recipe:

- La regola `program: file1.c file2.h, file3.c, ...` crea automaticamente l'eseguibile `program` utilizzando i file contenuti nei prerequisiti
- Se un target dipende solo da un prerequisito con il suo stesso nome (a meno dell'estensione), tale prerequisito è omissibile

  (es: la regola`program:` cerca un file `program.x` da utilizzare come prerequisito, dove `.x` è un'estensione qualsiasi)
- Se definite dall'utente, le regole implicite utilizzano le seguenti variabili implicite:
  - La variabile `CC` contiene il compilatore da utilizzare
  - La variabile `CFLAGS` contiene le flag da utilizzare per la compilazione
  - La variabile `LDFLAGS` contiene le flag da utilizzare per il linking (ossia l'opzione `[-L]` di `gcc`)
  - La variabile `LDLIBS` contiene le librerie da includere in fase di linking (ossia l'opzione `[-l]` di `gcc`)

Esempio:

```
CC := gcc
CFLAGS := -Wall -Wextra -O3
PROGS := merge_sorted_lists sort_file_int main

.PHONY: clean

all: $(PROGS)

main: main.c list.h list.c

clean:
    rm -f *.o $(PROGS)
```

4.12.3 Corretto uso degli header file

Abbiamo già accennato come la direttiva `#include filename` venga utilizzata per includere liberie standard o file definiti dall'utente (sezione 4.2).

In particolare, abbiamo trattato di come tale direttiva imponga al pre-processore sostanzialmente di **copiare e incollare** il contenuto del file incluso all'interno del file includente.

Esempio:

```
----- File: file1.c
void func(){...}

void main(){...}

----- File: file2.c

#include "file1.c"
/*
    al posto della direttiva precedente viene incollato
        "void func(){...}

        void main(){...}"
*/

...
```

Per via di tale funzionamento *naive*, possono verificarsi situazioni spiacevoli:

- Supponiamo che i due file `file1.c` e `file2.c` importino entrambi il file `file3.c`
- Supponiamo inoltre che il file `file4.c` importi sia il file `file1.c` e `file2.c`
- In tal caso, all'interno del file `file4.c` vi sarebbero due copie del contenuto del file `file3.c`

Sebbene tale utilizzo non generi alcuna problematica a livello di compilazione, quest'ultima risulterebbe essere **estremamente più lunga**, poiché verrebbero pre-compilate, compilate e linkate più volte le stesse cose. Di conseguenza, è necessario un'utilizzo corretto delle direttive disponibili, affinché non si vada ad impattare la performance.

Definizione 28: Header file

All'interno di un **header file** (estensione `.h`) vengono inserite tutte le istruzioni "senza logica" (es: include, definizione di macro, dichiarazione di nuovi tipi, funzioni, ...)

Ogni header file è **associato** (dal programmatore) **ad un file sorgente** `.c` all'interno del quale viene definita la vera logica legata alle istruzioni dell'header file (es: viene definita una funzione precedentemente dichiarata nell'header file).

- Consideriamo il seguente file

```
----- File: file1.c
#include <...>
#include <...>

#define ...

void func1(){...}
void func2(){...}
```

- Accorpando le sue istruzioni "senza logica" in un header file, otteniamo che

```
----- File: file1.h
#include <...>
#include <...>

#define ...

void func1();
void func2();

----- File: file1.c
#include "file1.h"

void func1(){...}
void func2(){...}
```

- In tal modo, il file sorgente `file1.c` avrà comunque accesso alle istruzioni "senza logica", poiché il contenuto del file `file1.h` viene copiato al suo interno, "ripristinando" il file originale

Osservazione 55

Ricordiamo che l'associazione della definizione di una funzione precedentemente dichiarata avviene nella **fase di linking**, dunque l'ultima fase.

Per tanto, includendo un header file all'interno di un file sorgente (ossia il `.c`), è necessario compilare solo e comunque il file `.c`.

In tal modo, tutti i file che vogliono utilizzare le funzioni definite nel `.c` possono importare direttamente l'header file, ottenendo così una corretta compilazione (poiché ogni cosa è definita nell'header) ed un corretto linking (poiché la logica viene associata solo in fase finale).

Esempio:

- Consideriamo i file dell'esempio precedente

```
----- File: file1.h
#include <...>
#include <...>

#define ...

void func1();
void func2();


----- File: file1.c
#include "file1.h"

void func1(){...}
void func2(){...}
```

- Supponiamo inoltre che esistano i seguenti file

```
----- File: file2.c
#include "file1.h"
...

----- File: file3.c
#include "file1.h"
...
```

- In fase di pre-compilazione, in entrambi tali file verrebbe copiato il contenuto dell'header file `file1.h`
- In tal modo, entrambi i file verrebbero compilati con successo, poiché tutte le funzioni utilizzate sono dichiarate
- Infine, in fase di linking ad ognuna di tali funzioni dichiarate verrebbe associata la logica definita all'intero del file `file1.c`
- **NOTA:** nell'esempio precedente, per semplicità di dimostrazione è stato omesso l'uso dei file `file2.h` e `file3.h`. In una situazione ottimale, ovviamente, andrebbero creati anche tali file, associandoli ai loro file sorgenti tramite l'inclusione (ossia in modo analogo ai file `file1.c` e `file1.h`)

Proposizione 18: Direttive `#ifndefn` e `#endif`

Le **direttive `#ifndefn VAR` e `#endif`** impongono al pre-compilatore di interpretare una porzione di codice (es: durante un `#include`) **solo se `VAR` non è definita**.

Esempio:

```
#ifdefn VAR
...
#endif
```

Osservazione 56

Le direttive `#ifndefn VAR` e `#endif` possono essere utilizzate per far si che un header file venga copiato solo ed esclusivamente una volta all'interno del codice (**include guards**), riducendo notevolmente i tempi di compilazione.

Esempio:

- Consideriamo i seguenti file:

```
----- File: file1.h
#ifndefn FILE1_H
#define FILE1_H

#include "file1.h"
...

#endif

----- File: file2.h
#include "file1.h"

----- File: file3.h
#include "file2.h"
#include "file1.h"
```

- La porzione di codice all'interno di `file1.h` circondata dagli include guards viene eseguita una sola volta, poiché durante la sua prima esecuzione viene impostata la variabile `FILE_H`, rendendo la condizione di `#ifndefn` falsa tutte le successive volte
- Di conseguenza, tramite l'uso degli include guards, il contenuto del file `file1.h` viene copiato all'interno del file `file3.h` solo una volta, "ignorando" il secondo import

4.12.4 Debugging con `gdb`

Nello sviluppo di codice è spesso utile eseguire e fermare un programma per effetture operazioni di **debugging**. In particolare, essendo un linguaggio più verso il basso livello, tale funzionalità risulta particolarmente utile per il linguaggio C.

Per effettuare debugging di programmi scritti in C, abbiamo i seguenti strumenti:

- L'opzione `[-g]` del comando `gcc` permette di inserire dei **simboli di debug** all'interno del codice (es: il codice sorgente del programma stesso, ...). L'uso di tale opzione permette di visualizzare più comodamente il contenuto delle varie variabili, chiamate di funzione, ecc. Tali simboli di debug possono essere **rimossi** dal programma utilizzanto il comando `strip`.
- Il programma `gdb` (GNU Project Debugger) permette di effettuare operazioni di debugging (funziona anche per altri linguaggi)

Osservazione 57

Senza l'opzione `[-g]` per il comando `gcc`, l'uso del programma `gdb` risulta molto più complesso in quanto dovremmo essere noi a tener traccia di quale registro della CPU o quale indirizzo di memoria contenga quale variabile.

Utilizzando assieme i due strumenti, dunque, possiamo comodamente effettuare operazioni di debugging. Difatti,

In particolare, evidenziamo i seguenti comandi eseguibili all'interno di `gdb`:

- `file program` permette di selezionare l'eseguibile `program`
- `run` permette di avviare l'esecuzione dell'eseguibile selezionato
- `list start,end` permette di visualizzare il codice del programma partendo dalla riga `start` fino alla riga `end` (solo se `[-g]` viene usato)
- `diplay var` permette di visualizzare il contenuto della variabile `var` (solo se `[-g]` viene usato)
- `break line` permette di inserire un **breakpoint** sulla linea di codice `line`, implicando che l'esecuzione del programma verrà interrotta prima di eseguire la linea `line` (solo se `[-g]` viene usato)
- `step` permette di eseguire l'istruzione successiva (solo durante una pausa di esecuzione)
- `jump line` permette riprendere l'esecuzione del programma dalla linea `line`, saltando tutte le istruzioni nel mezzo (solo durante una pausa di esecuzione)

Esempio:

```
[exyss@exyss ~]$ cat test.c
#include <stdio.h>

int main(){
    int a=0;
    a++;
    a = a + 1;
    a = a + 2;
    a += 3;
    a += 4;
    a = a + 5;
    printf("Il valore di a e': %d\n", a);
    return a+1;
}

[exyss@exyss ~]$ gcc -g test.c -o test

[exyss@exyss ~]$ gdb

GNU gdb (GDB) 13.1
[...]
[...]
For help, type "help".
Type "apropos word" to search for commands related to "word".

(gdb) file test
Reading symbols from test...

(gdb) list 1,20
1    #include <stdio.h>
2
3    int main(){
4        int a=0;
5        a++;
6        a = a + 1;
7        a = a + 2;
8        a += 3;
9        a += 4;
10        a = a + 5;
11        printf("Il valore di a e': %d\n", a);
12        return a+1;
13    }

(gdb) display a
No symbol "a" in current context.
```

```
(gdb) break 8
Breakpoint 1 at 0x1154: file test.c, line 8.

(gdb) run
Breakpoint 1, main () at test.c:8
8        a += 3;

(gdb) display a
1: a = 4

(gdb) step
9        a += 4;
1: a = 7

(gdb) jump 11
Continuing at 0x555555555160.
Il valore di a e': 7
[Inferior 1 (process 5258) exited with code 010]

(gdb) exit
```

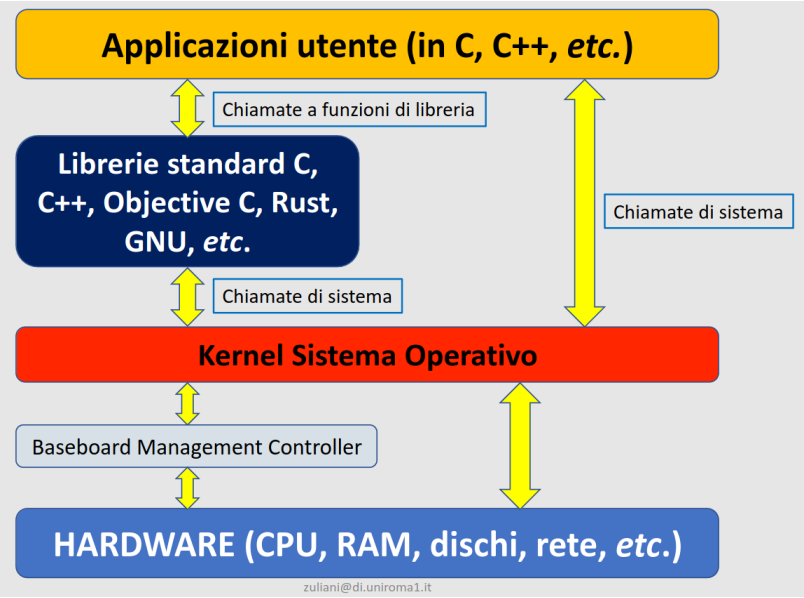
5

Programmazione di sistema

Con il termine **programmazione di sistema** intendiamo la scrittura di codice relativo all'uso di **risorse del sistema operativo** (es: CPU, RAM, Dispositivi I/O, ...). In particolare, ricordiamo che all'interno di un sistema operativo il componente che si occupa della gestione, dell'accesso e dell'utilizzo delle risorse disponibili sia il **kernel**.

Il sistema operativo mette a disposizione una serie di **syscalls (system calls)**, ossia un limitato numero di "punti di accesso" al kernel, in modo da permettere ad un programma di interfacciare con esso. Le syscall possono essere utilizzate **direttamente** o tramite **funzioni di generali**, ossia le funzioni delle librerie standard che utilizzano in modo ottimale tali syscall (es: la funzione `malloc()` usa la syscall `sbrk()`).

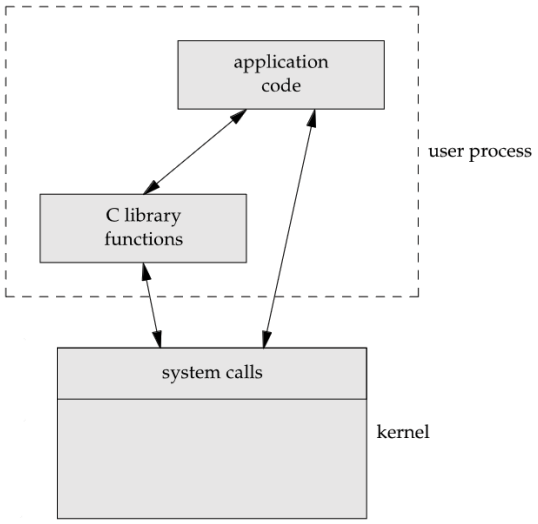
Inoltre, ricordiamo che le informazioni relative alle syscall si trovino all'interno della sezione 2 del `man`, mentre le funzioni di libreria generali si trovino nella sezione 3.



5.1 System calls

Nel linguaggio C, la definizione delle syscall è **indipendente** dalla tecnica utilizzata dallo specifico sistema operativo stesso per invocare le syscall stesse. Difatti, **per ogni syscall del sistema operativo esiste una funzione C avente lo stesso nome**, permettendo ad un processo utente di utilizzare tali syscall come se fossero normali funzioni:

- Il processo utente invoca tali funzioni, passando come argomento i dati richiesti
- Le funzioni invocano il corrispettivo servizio del kernel utilizzando la tecnica richiesta (es: mettendo gli argomenti dati in registri specifici e poi generando un interrupt)



Di conseguenza, abbiamo che:

- Nel linguaggio C, sia le syscall che le funzioni generali sono funzioni
- Entrambe forniscono servizi generali ad un programma
- Una funzione generale può essere rimpiazzata da un'altra funzione, ma una syscall no

  (es: possiamo implementare la nostra versione della funzione `malloc()`, ma dovremo sempre necessariamente utilizzare la syscall `sbrk()` al suo intero)

- Le syscall introducono una separazione dei compiti

  (es: la syscall `sbrk` alloca porzioni di memoria in kernel mode, le quali vengono gestite in user mode dal resto delle istruzioni presenti all'interno della funzione `malloc()`)

- Le funzioni di libreria semplificano l'uso delle syscall. Difatti, quest'ultime espongono un'interfaccia minimale, mentre le funzioni di libreria forniscono funzionalità più elaborate

L'esecuzione di una syscall può **interrompersi** e non andare a buon fine per diversi motivi (es: mancanza di privilegi, di risorse o argomenti invalidi). Per tale motivo, è fondamentale controllare i valori di ritorno delle syscall e segnalare all'utente l'eventuale verificarsi di errori.

Per ottenere ciò, le librerie standard forniscono i seguenti strumenti per gestire gli errori delle syscall:

- La libreria `<errno.h>` fornisce la **variabile** `errno` all'interno della quale viene conservato il **codice di errore dell'ultima syscall andata in errore** (dunque, se una syscall va a buon fine, non verrà modificato il valore di `errno`)
- La libreria `<string.h>` fornisce la funzione `char* strerror(int errnum)`, la quale restituisce in output un **messaggio di errore** ottenuto "traducendo" il codice di errore di una syscall dato in input.
- La libreria `<stdio.h>` fornisce la funzione `void perror(const char* prefix)`, la quale **scrive su stderr** la stringa `"prefix:errno_str\n"`, dove `errno_str` è il messaggio restituito da `strerror(errno)`

  Dunque, si ha che `perror("main")` è equivalente a `fprintf(stderr, "main:%s\n", stderror(errno))`

In alcuni casi, può anche essere utile **monitorare** tramite il comando `strace` le syscall invocate da un processo (vedere il manuale per le opzioni).

5.2 Gestione della memoria

Le varie funzioni `malloc`, `calloc`, `realloc`, ecc utilizzano delle syscall della liberia `<unistd.h>` per effettuare le operazioni di gestione della memoria:

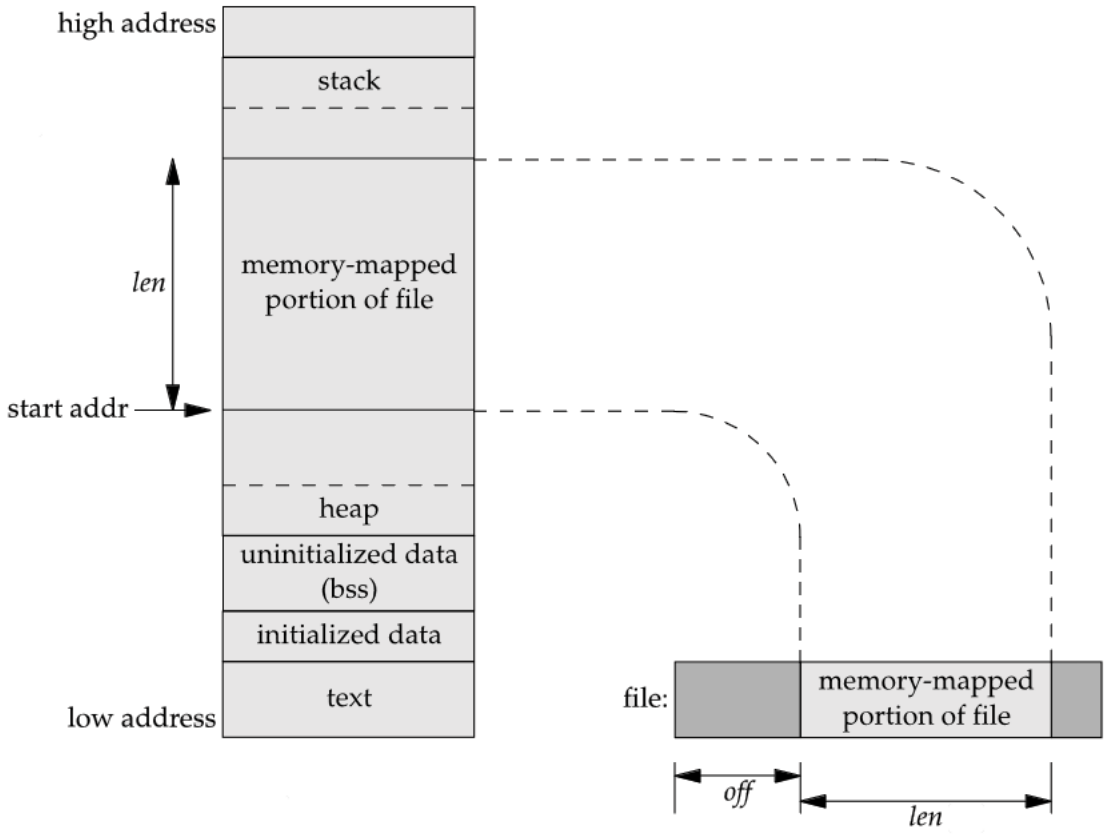
- La syscall `int brk(void* addr)` permette di definire la fine del data segment di un processo
- La syscall `void* sbrk(intptr_t increment)` permette di incrementare il data space (ossia l'intero spazio dati occupato) di un processo

L'uso da parte dell'utente di tali syscall risulta **pericoloso** (soprattutto `brk`). Per tanto, è consigliato di utilizzarle solo dopo averne studiato molto attentamente il funzionamento sul manuale.

Per quanto riguarda la gestione dei file in memoria, invece, la libreria `<sys/mman,h>` fornisce la syscall `void* mmap(void* addr, size_t len, int prot, int flags, int fd, off_t off)`, la quale permette di **mappare un file ad un'area di memoria**, dove:

- `addr` è l'indirizzo iniziale dell'area di memoria in cui effettuare la mappatura. Se `addr = NULL`, il kernel sceglierà da solo.
- `len` è il numero di byte da trasferire

- `prot` è il livello di protezione. Può essere impostato a:
  - `PROT_READ`: permette la lettura della regione di memoria
  - `PROT_WRITE`: permette la scrittura della regione di memoria
  - `PROT_EXEC`: permette l'esecuzione della regione di memoria
  - `PROT_NONE`: impedisce l'accesso alla regione di memoria
- `flag` specifica se le operazioni effettuate valgano anche per altri processi che stanno mappando la stessa regione. Può essere impostato a:
  - `MAP_SHARED`: la regione è condivisa, implicando che le modifiche siano condivise tra tutti i processi
  - `MAP_PRIVATE`: viene creata una copia privata del mapped file, implicando che le modifiche abbiano effetto solo a livello locale
- `fd` è il file descriptor del file (il quale deve essere aperto prima aperto)
- `off` è l'offset del file



Tale syscall viene utilizzata prevalentemente per gestire operazioni di **memory-mapped I/O**, permettendo alle operazioni di lettura/scrittura sul buffer creato di risultare come se siano state effettuate sul disco, senza che quest'ultimo venga realmente acceduto.

La libreria `<sys/mman.h>`, inoltre, fornisce le ulteriori seguenti due syscall relative al memory-mapped I/O:

- La syscall `int msync(void* addr, size_t len, int flags)` permette di scrivere sul disco le modifiche effettuate ad un file memory-mapped (solo se mappato usando `MAP_SHARED`)
- La syscall `int munmap(void* addr, size_t len)` permette di de-mappare una regione di memoria.

Osservazione 58

Quando un processo termina, le sue regioni mappate vengono **automaticamente de-mappate**, ma il contenuto delle regioni **non viene scritto sul disco**.

Per tanto, è sempre buona prassi scrivere manualmente con `msync()` per poi de-mappare con `munmap()`

5.3 File descriptors

Precedentemente, abbiamo discusso di come un file sia un'**astrazione** che descrive ogni cosa all'interno dell'ambiente Linux (fatta eccezione dei processi) ed abbiamo accenato come ogni file aperto possieda un **file descriptor**, ossia un intero univoco facente riferimento ad esso.

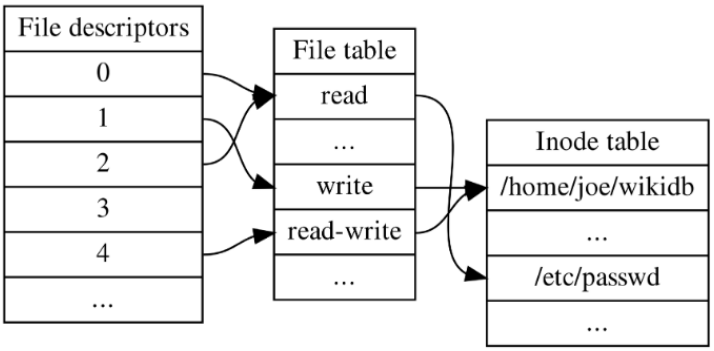
In particolare, i file descriptor corrispondono ad un **intero** e sono **relativi ad un processo**. Gli interi associati ai file descriptor sono **univoci per un processo** e vengono generati in modo **sequenziale**, partendo dall'intero 0 ed assegnando l'intero più basso disponibile. Quando un file descriptor viene **chiuso**, il suo intero potrà essere riutilizzato.

Osservazione 59

È possibile **aprire più volte lo stesso file** (anche su più processi), ottendo file descriptor diversi facenti riferimento allo stesso file.

Di default, ogni processo possiede i seguenti tre file descriptor associati ai canali standard:

- Il file descriptor `0` facente riferimento a `stdin`
- Il file descriptor `1` facente riferimento a `stdout`
- Il file descriptor `2` facente riferimento a `stderr`



Ad ogni file descriptor sono associate due categorie di flag:

- **File status flags**, le quali contengono informazioni relative allo stato del file e sono condivise tra tutti i file descriptor clonati da un altro file descriptor (torneremo su questo)
- **File descriptor flags**, le quali sono indipendenti dal contenuto e dallo stato del file, descrivendo le proprietà e il comportamento delle operazioni effettuate sul file stesso.

  Ogni file descriptor facente riferimento allo stesso file possiede le proprie file descriptor flags ed alcuni di quest'ultimi possono essere definiti solo per alcuni tipi di file speciali.

Le **flag** vengono rappresentate mediante **maschere di bit**, dove se un determinato bit associato ad un flag è impostato ad 1, tale flag verrà considerato come impostato (in modo analogo alle maschere dei privilegi e alla umask).

Per ogni maschera di una flag impostabile esiste una **macro** (es: `O_RDONLY`). Tali flag possono essere combinate tra loro mettendo in **bit-wise OR** le loro maschere.

(es: date le macro `MACRO1 = 01000000` e `MACRO2 = 00001000`, si ha che `MACRO1 | MACRO2 = 01001000` e dunque che entrambe le flag siano impostate)

Le **file status flag** si dividono a loro volta in tre categorie:

- **Flag di modalità di accesso**, le quali definiscono la modalità con cui accedere al file, ossia in lettura, in scrittura o in entrambe. Una volta aperto il file, esse non possono essere modificate.
- **Flag di modalità di apertura**, le quali definiscono le azioni eseguite durante l'apertura del file e non vengono conservate.
- **Flag delle modalità operative**, le quali definiscono il comportamento delle operazioni di lettura e scrittura. Possono essere modificate anche mentre il file è aperto.

Definizione 29: Duplicazione di un file descriptor

Definiamo come **duplicazione** di un file descriptor l'operazione tramite, all'interno del processo chiamante, viene creato un nuovo file descriptor facente riferimento allo stesso file del file descriptor dato

La libreria `<unistd.h>` fonisce:

- La syscall `int dup(int oldfd)`, la quale è in grado di duplicare il file descriptor `oldfd`. Il numero associato al nuovo file descriptor sarà il numero più basso associabile all'interno del processo chiamante. Viene ritornato il nuovo file descriptor.
- La syscall `int dup2(int oldfd, int newfd)`, la quale è in grado di duplicare il file descriptor `oldfd`.Il numero associato al nuovo file descriptor sarà `newfd`. Viene ritornato il nuovo file descriptor (ossia `newfd`).

  Inoltre, si ha che:

  - Se esiste già un file descriptor avente numero `newfd`, esso verrà chiuso
  - Se `oldfd` non è un file descriptor valido, la chiamata fallirà e `newfd` non verrà chiuso in alcun caso
  - Se `oldfd = newfd`, la chiamata non avrà effetto e verrà ritornato `newfd`

Osservazione 60

La syscall `dup2()` può essere utilizzata per effettuare le **ridirezioni dei canali**

Esempio:

- La syscall `dup2(2, 1)` fa si che il file descriptor 1 faccia riferimento ad stderr, chiudendo stdout. Di conseguenza, tutte le operazioni di scrittura effettuate sul file descriptor 1, verranno effettuate su stderr
- In particolare, ricordiamo che il file pointer `stdout` presente di default nella libreria `<stdio.h>` contenga al suo interno il file descriptor 1. Per tanto, statement come `printf(...)` e `fprintf(stdout, ...)` verranno ridirezionati su stderr

5.4 Gestione dei file

5.4.1 Operazioni sui file

Per **aprire un file**, la libreria `<fcntl.h>` (importata anche da `<stdio.h>`) fornisce le due syscall

```
int open(const char* pathname, int flags)
int open(const char* pathname, int flags, mode_t mode)
```

dove:

- `pathname` corrisponde al **nome del file** da aprire
- `flags` definisce le **flag in bit-wise OR** da impostare
- `mode` definisce i **privilegi in bit-wise OR** da impostare per il file nel caso in cui venga creato
- L'intero restituito è il **file descriptor** aperto

Le flag impostabili all'interno del parametro `flags` corrispondono a:

- `O_RDONLY`: il file viene aperto in lettura
- `O_WRONLY`: il file viene aperto in scrittura
- `O_RDWR`: il file viene aperto in lettura e scrittura
- `O_CREAT`: se non esiste, il file viene creato. Richiede la presenza del parametro `mode` (dunque l'uso della seconda syscall)
- `O_EXCL`: se utilizzato assieme a `O_CREAT`, genera un errore se il file esiste già.
- `O_APPEND`: se la modalità di accesso consente la scrittura, il contenuto scritto viene appeso alla fine del contenuto precedente (se esistente)
- `O_TRUNC`: se la modalità di accesso consente la scrittura e il file è un file regolare, il contenuto precedente del file viene troncato

A questo punto, risulta ovvia l'analogia tra le modalità di apertura di `open()` e le modalità di apertura di `fopen()`. Difatti, la funzione di libreria `fopen()` usufruisce al suo interno della syscall `open()`.

`fopen()` **mode**	`open()` **flags**
r	O_RDONLY
w	O_WRONLY | O_CREAT | O_TRUNC
a	O_WRONLY | O_CREAT | O_APPEND
r+	O_RDWR
w+	O_RDWR | O_CREAT | O_TRUNC
a+	O_RDWR | O_CREAT | O_APPEND

Per quanto riguarda il parametro `mode`, invece, il suo tipo di dato è `mode_t` (una macro per `int`), utilizzato per indicare che il valore assunto sia un **bit-wise OR** delle seguenti maschere:

- `S_IRWXU`: vengono impostati i permessi di lettura, scrittura ed esecuzione per il proprietario del file
- `S_IRUSR`: viene impostato il permesso di lettura per il proprietario del file
- `S_IWUSR`: viene impostato il permesso di scrittura per il proprietario del file
- `S_IXUSR`: viene impostato il permesso di esecuzione per il proprietario del file
- `S_IRWXG`: vengono impostati i permessi di lettura, scrittura ed esecuzione per il gruppo di appartenenza del file
- `S_IRGRP`: viene impostato il permesso di lettura per il gruppo di appartenenza del file
- `S_IWGRP`: viene impostato il permesso di scrittura per il gruppo di appartenenza del file
- `S_IXGRP`: viene impostato il permesso di esecuzione per il gruppo di appartenenza del file
- `S_IRWXO`: vengono impostati i permessi di lettura, scrittura ed esecuzione per gli altri utenti del file
- `S_IROTH`: viene impostato il permesso di lettura per gli altri utenti del file
- `S_IWOTH`: viene impostato il permesso di scrittura per gli altri utenti del file
- `S_IXOTH`: viene impostato il permesso di esecuzione per gli altri utenti del file
- `S_ISUID`: viene impostato il bit di SetUID per il file
- `S_ISGID`: viene impostato il bit di SetGID per il file
- `S_ISVTX`: viene impostato lo sticky bit per il file

Osservazione 61

I privilegi definiti dal parametro `mode` verranno applicati **solo se il file viene creato** (dunque solo se la flag `O_CREAT` è impostata).

La libreria `<unistd.h>`, invece, fornisce le syscall relative alle **operazioni su file**:

- La syscall `ssize_t read(int fd, void* buf, size_t count)` legge `count` byte dal file avente file descriptor `fd`, copiando il contenuto nel buffer `buf`. Viene ritornato il numero di byte letti dal file (-1 se errore).
- La syscall `ssize_t write(int fd, const void* buf, size_t count)` legge `count` byte dal buffer `buf`, copiando il contenuto nel file avente file descriptor `fd`. Viene ritornato il numero di byte scritti sul file (-1 se errore).

- La syscall `int close(int fd)` chiude il file descriptor `fd`. Viene ritornato 0 se la chiusura va a buon fine (-1 se errore)
- La syscall `int lseek(int fd, off_t offset, int whence)` imposta il cursore alla posizione `whence` incrementata di `offset` byte. I valori impostabili per il campo `whence` sono:
  - `SEEK_SET`, ossia l'inizio del file
  - `SEEK_CUR`, ossia la posizione attuale del cursore
  - `SEEK_END`, ossia la fine del file

Osservazione 62

Le syscall `read()`, `write()` e `lseek()` risultano analoghe alle funzioni `fread()`, `fwrite()` e `fseek()` utilizzabili su un file pointer.

Tuttavia, a differenza di esse, la lettura e la scrittura **non sono bufferizzate**, mentre, in caso di successo, `lseek()` ritorna la **nuova posizione del cursore**, a differenza di `fseek()` che ritorna 0 (entrambe ritornano -1 se la nuova posizione è invalida).

Osservazione 63

Se per un determinato file vi sono **più file descriptor aperti**, il file verrà chiuso solamente quando tutti i file descriptor saranno chiusi.

Esempio:

```
#include <unistd.h>
#include <fcntl.h>

void main(){
    //Modalità apertura: "w"
    int flags = O_WRONLY | O_CREAT | O_TRUNC;

    //Privilegi: rw-r--r--
    int mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

    int fd = open("hello.txt", flags, mode);    //apertura

    char str[14] = "Hello World!\n";    //scrittura

    write(fd, str, 13);     //chiusura

    close(fd);
}
```

5.4.2 Manipolazione delle proprietà dei file

Per **manipolare** le informazioni, le proprietà e i privilegi dei file, vengono fornite numerevoli syscall, sparse per le varie librerie standard.

La libreria `<sys/stat.h>`, ad esempio, fornisce:

- Uno struct `stat`, avente i seguenti campi:

```
struct stat {
    dev_t      st_dev;      // ID of device containing file
    ino_t      st_ino;      // Inode number
    mode_t     st_mode;     // File type and mode
    nlink_t    st_nlink;    // Number of hard links
    uid_t      st_uid;      // User ID of owner
    gid_t      st_gid;      // Group ID of owner
    dev_t      st_rdev;     // Device ID (if special file)
    off_t      st_size;     // Total size, in bytes
    blksize_t  st_blksize;  // Block size for filesystem I/O
    blkcnt_t   st_blocks;   // Number of 512 B blocks allocated

    struct timespec  st_atim;  // Time of last access
    struct timespec  st_mtim;  // Time of last modification
    struct timespec  st_ctim;  // Time of last status change
};
```

- La syscall `int stat(const char* path, struct stat* buf)`, la quale inserisce le informazioni relative al file dato all'interno del buffer `buf`
- La syscall `int fstat(int fd, struct stat* buf)`, analoga a `stat()` ma utilizzante un file descriptor
- Le macro `S_ISREG(m)`, `S_ISDIR(m)`, `S_ISCHR(m)`, `S_ISBLK(m)`, `S_ISFIFO(m)`, `S_ISLNK(m)`, `S_ISSOCK(m)` per verificare il tipo del file in base agli attributi dello struct `m` di tipo `stat`
- La syscall `int chmod(const char* path, mode_t mode)`, la quale sostituisce i privilegi del file dato con la maschera `mode`
- La syscall `int fchmod(int fd, mode_t mode)`, analoga a `stat()` ma utilizzante un file descriptor

Osservazione 64

Ricordiamo che il tipo `mode_t` utilizzato come campo dallo struct `stat` e come parametro dalle syscall `chmod()`, `fchmod()`, `mkdir()` sono di tipo `mode_t`, sia lo stesso del parametro `mode` della syscall `open()`.

Per tanto, esso corrisponde ad una **maschera in bit-wise OR** delle macro relative ai privilegi (`S_IRWXU`, `S_IRUSR`, ...)

La libreria `<unistd.h>`, invece, fornisce:

- La syscall `int chown(const char* path, uid_t owner, gid_t group)`, la quale modifica il proprietario e il gruppo di appartenenza del file.

  Se l'UID dato in input è `-1`, il proprietario rimane inalterato. Analogamente, lo stesso avviene per il GID dato.

- La syscall `int fchown(int fd, uid_t owner, gid_t group)`, analoga a `fchown()` ma utilizzante un file descriptor

- La syscall `int unlink(const char* path)`, la quale rimuove un link (hard o soft) dal file system. Se tale link è l'ultimo hardlink relativo ad un file, viene eliminato anche il file

- La syscall `int ink(const char* target, const char* linkpath)`, la quale crea un hardlink di nome `linkpath` verso il file puntato da `target`

- La syscall `int symlink(const char* target, const char* linkpath)`, la quale crea un softlink di nome `linkpath` verso il file (o link) puntato da `target`

Infine, la libreria `<stdio.h>` fornisce:

- La syscall `int rename(const char* oldpath, const char* newpath)`, la quale rinomina (o sposta) il file `oldpath` nel file `newpath`

5.4.3 Operazioni sulle directory

Per quanto riguarda le **directory** (che ricordiamo essere solo dei file speciali), vengono fornite anche le seguenti syscall e funzioni di libreria:

- La syscall `int mkdir(const char* path, mode_t mode)`, la quale crea la directory `path` con la maschera di permessi `mode` (fornita da `<sys/stat.h>`)

- La syscall `int chdir(const char* path)`, la quale imposta la `working directory` del processo su `path` (fornita da `<unistd.h>`)

- La syscall `int chroot(const char* path)`, la quale imposta la `root directory` del processo su `path` (fornita da `<unistd.h>`)

- La funzione `DIR* opendir(const char* path)`, la quale apre la directory `path` e restituisce un puntatore ad uno struct di tipo `DIR`, un tipo esclusivo restituito **solo da questa funzione** (fornita da `<sys/types.h>` e `<dirent.h>` - entrambe le librerie sono necessarie)

- La funzione `DIR* fdopendir(int fd)`, analoga a `opendir()` ma utilizzante un file descriptor, dunque richiedente prima l'uso di `open()` (fornita da `<sys/types.h>` e `<dirent.h>` - entrambe le librerie sono necessarie)

- La funzione `int closedir(DIR* dirp)`, la quale chiude la directory puntata da `dirp` (fornita da `<sys/types.h>` e `<dirent.h>` - entrambe le librerie sono necessarie)

- La funzione `int rmdir(const char* path)`, la quale rimuove la directory `path` (fornita da `<unistd.h>`)
- La funzione `struct dirent* readdir(DIR* dirp)`, la quale legge le informazioni del prossimo elemento disponibile all'interno della directory puntata da `dirp`, restituendo un oggetto di tipo `dirent` contenente tali informazioni o `NULL` se non vi sono elementi rimanenti (fornita da `<dirent.h>`).

```
struct dirent {
    ino_t          d_ino;       // Inode number
    off_t          d_off;       // Not an offset; check the man
    unsigned short d_reclen;    // Length of this record
    unsigned char  d_type;      // Type of file; not supported
                                    by all filesystem types
    char           d_name[256]; // Null-terminated filename
};
```

5.5 Sincronizzazione tra processi

Definizione 30: Lock

Un **lock** è un meccanismo che permette la **sincronizzazione tra processi**, impedendo ad essi di accedere simultaneamente alla stessa risorsa (**mutua esclusione**)

Una syscall di fondamentale importanza nell'ambito dell'uso dei lock è la syscall

```
int fcntl(int fd, int cmd, ...  /* arg */)
```

fornita dalla libreria `<fcntl.h>`. Tale syscall è in grado di:

- Duplicare il file descriptor `fd`
- Manipolare le flag del file descriptor `fd`
- Manipolare le flag di status del file riferito dal file descriptor `fd`
- Gestire i lock su `fd`

A tali operazioni sono associati un **insieme di comandi**, ai quali è associata una macro passabile come valore al parametro `cmd`. Il parametro `arg` è **opzionale** e definisce l'eventuale parametro da dare in input al comando definito tramite `cmd`.

In particolare, gli argomenti dei comandi relativi alla **gestione dei lock** sono di tipo `flock`, uno struct definito come:

```
struct flock {
    ...
    short l_type;   // Type of lock:
                    // F_RDLCK, F_WRLCK, F_UNLCK
```

```
    short l_whence; // How to interpret l_start:
                    // SEEK_SET, SEEK_CUR, SEEK_END

    off_t l_start;  // Starting offset for lock
    off_t l_len;    // Number of bytes to lock
    pid_t l_pid;    // PID of process blocking our lock
                    // (set by F_GETLK and F_OFD_GETLK)
    ...
};
```

dove:

- L'attributo `l_type` determina il tipo di operazione sul lock richiesta
- L'attributo `l_whence` determina il punto di inizio del lock e può essere impostato a:
  - `SEEK_SET`, ossia l'inizio del file
  - `SEEK_CUR`, ossia la posizione attuale del cursore
  - `SEEK_END`, ossia la fine del file
- L'attributo `l_start` determina l'offset di byte da `l_whence` per cui applicare il lock (dunque il vero punto di partenza è `l_whence + l_start`)
- L'attributo `l_len` determina quanti byte a partire da `l_whence + l_start` vengano bloccati. Se `l_len = 0`, viene bloccato l'intero file.

Per comodità, listeremo i vari comandi e i suoi parametri utilizzando la notazione `cmd(type)`, dove `type` è il tipo dell'argomento da dare. Tuttavia, ricordiamo che l'uso corretto dei comandi sia `fcntl(fd, cmd)` (o `fcntl(fd, cmd, arg)` se viene utilizzato anche tale parametro):

- Il comando `F_GETFL()` restituisce la modalità di accesso del file e le sue flag di status

  (es: `val = fcntl(fd, F_GETFL)`)
- Il comando `F_SETFL(int)` imposta le file status flag. L'argomento dato può essere `O_APPEND`, `O_ASYNC`, `O_DIRECT`, `O_NOATIME` o `O_NONBLOCK`

  (es: `fcntl(fd, F_SETFL, O_APPEND)`)
- Il comando `F_SETLK(flock)` acquisisce o rilascia un lock sul file:
  - Se l'attributo `l_type` dell'argomento dato è impostato a `F_RDLCK`, verrà acquisito un lock per la lettura sul file
  - Se l'attributo `l_type` dell'argomento dato è impostato a `F_WRLCK`, verrà acquisito un lock per la scrittura sul file
  - Se l'attributo `l_type` dell'argomento dato è impostato a `F_UNLCK`, verrà rilasciato un lock per la lettura sul file (se il lock è esistente)

  - Viene restituito -1 se un altro processo possiede il lock richiesto

- Il comando `F_SETLKW(flock)` risulta analogo a `F_SETLK(flock)`, con la differenza che il lock richiesto sia **bloccante**, ossia viene atteso che il lock sul file per venga rilasciato (se presente per il tipo di lock richiesto)
- Il comando `F_GETLKW(flock)` testa l'esistenza del lock dato come argomento. Se il lock può essere acquisito, l'attributo `l_type` viene impostato su `F_UNLCK`.

Esempio:

```
void main(){
    struct flock clock;
    memset(&lock, 0, sizeof(lock));
    lock.l_type = F_WRLCK;

    int fd = open ("file", O_WRONLY);
    fcntl (fd, F_SETLKW, &lock);    //creazione lock in scrittura
}
```

Osservazione 65: Lock advisory e mandatory

I lock impostati tramite `fcntl()` sono **advisory**, ossia richiedono la cooperazione tra processi:

- Ogni processo deve prima effettuare una chiamata `F_GETLK`, per poi utilizzare `F_SETLK/LKW` solo se il lock è acquisibile
- Cercare di leggere o scrivere su un file sul quale un processo detiene un lock **non ha l'effetto di impedire l'operazione**

Per impostare un lock di tipo **mandatory**, ossia in grado di impedire le operazioni, è necessario che il file system supporti quel tipo di lock.

Un'altra syscall fondamentale per la **sincronizzazione tra processi** è la syscall

```
int select(int nfds, fd_set* readfds, fd_set* writefds,
      fd_set* exceptfds, struct timeval* timeout)
```

fornita dalla libreria `<sys/select.h>`, dove:

- `nfds` è il numero del file descriptor con il valore più alto aumentato di 1 tra quelli da monitorare

  (es: se `5` è il file descriptor più alto, allora `nfds = 6`)
- `readfds` è l'insieme dei file descriptor per cui si richiede la lettura
- `writefds` è l'insieme dei file descriptor per cui si richiede la scrittura
- `exceptfds` è l'insieme dei file descriptor da controllare per il verificarsi di eccezioni

- `timeout` è il timeout da attendere prima di interrompere la chiamata

Tale syscall permette di **monitorare uno o più file descriptor**, rimanendo in attesa che almeno uno di essi sia disponibile per effettuare l'operazione richiesta.

In particolare, la funzione ritorna il **numero di file descriptor disponibili** per l'operazione richiesta. (viene ritornato 0 in caso di timeout e -1 in caso di errore). Al termine dell'esecuzione, la funzione **rimuove dagli insiemi** in input tutti i file descriptor non disponibili per l'operazione richiesta.

A seconda del valore impostato per il parametro `timeout`, la chiamata assume comportamenti diversi:

- **Infinito**: la chiamata è bloccante, ossia termina solo quando è disponibile almeno un file descriptor o quando si genera un errore
- **Intervallo definito**: la chiamata ritorna se è disponibile almeno un file descriptor, se scade il timeout e se si genera un errore
- **Zero**: la chiamata non è bloccante e ritorna dopo aver controllato tutti i file descriptor. Solitamente, viene utilizzata per il *polling*.

Per **gestire gli insiemi dei file descriptor**, la libreria fornisce le seguenti macro:

- La macro `void FD_ZERO(fd_set* set)` svuota l'insieme dato
- La macro `void FD_SET(int fd, fd_set* set)` aggiunge `fd` all'insieme dato
- La macro `void FD_CLR(int fd, fd_set* set)` rimuove `fd` dall'insieme dato
- La macro `int FD_ISSET(int fd, fd_set* set)` verifica se `fd` sia nell'insieme dato

5.6 Ambiente di un processo

Definizione 31: Ambiente di un processo

L'**ambiente** di un processo è un insieme variabili accessibili dal processo, dove ogni variabile è definita come una stringa `KEY=value` (es: `SHELL:/bin/bash`).

Nel linguaggio C, l'ambiente di un processo è accessibile tramite il parametro opzionale `char* envp[]` della funzione main. Tale array è terminato dall'elemento `NULL`.

Esempio:

```
void main(int argc, char* argv[], char* envp[]){
    int i = 0;
    while(envp[i] != NULL){     //stampa tutte le variabili
        printf("%s\n", envp[i++]);
    }
}
```

Tuttavia, l'uso di tale parametro non risulta compatibile con tutti i sistemi. Per tanto, è consigliato l'uso della variabile esterna `extern char** environ`.

Esempio:

```
void main(int argc, char* argv[], char* envp[]){
    extern char** environ;

    int i = 0;
    while(environ[i] != NULL){     //stampa tutte le variabili
        printf("%s\n", environ[i++]);
    }
}
```

Per gestire le variabili di ambiente, la libreria `<stdlib.h>` fornisce le funzioni:

- `char* getenv(const char* name)`, la quale ritorna il valore della variabile `name` (se esiste)
- `char* setenv(const char* name, const char* value, int overwrite)`, la quale imposta la variabile `name=value`. Se la variabile `name` esiste già, essa viene aggiornata solo se `overwrite` $\neq$ `0`
- `int putenv(char* string)`, la quale imposta (o aggiorna) la variabile definita da `string`
- `int unsetenv(const char* name)`, la quale rimuove la variabile `name`
- `int clearenv(void)`, la quale rimuove tutte le variabili

All'interno della **shell** `bash`, le variabili d'ambiente impostate possono essere visualizzate con il comando `printev` o tramite `echo $VAR`. Tutte le variabili di `bash` vengono ereditate dai comandi eseguiti da `bash` stesso.

Inoltre, la shell `bash` permette di **eseguire** un comando con delle variabili d'ambiente aggiuntive ponendo tali variabili **prima del comando**:

```
VAR1=value1 VAR2=value2 ...  cmd [options]
```

5.7 Creazione di un processo

La **creazione di un processo**, sostanzialmente, consiste nella **duplicazione** di un processo chiamante (ossia il **padre**), dove viene generata una copia identica (ossia il **figlio**) di quest'ultimo.

In questo modo, vanno a crearsi una serie di relazioni padre-figlio, dando vita ad un vero e proprio **albero geneaologico dei processi**. Alla radice di tale albero vi è il processo `init`, avente PID 1. Difatti, tramite esso vengono creati tutti i processi di base del sistema operativo, i quali a loro volta creeranno vari processi.

Tramite il comando `pstree pid` è possibile visualizzare l'albero dei discendenti del processo avente PID `pid`

Definizione 32: Exit status

Definiamo come **exit status** di un processo il valore ritornato dopo la sua chiusura al suo processo padre.

Nel linguaggio C, l'exit status di un processo corrisponde al **valore ritornato dalla funzione** `main()`

Esempio:

```
int main(){
    ...
    return 0;   //0 è l'exit status
}
```

Osservazione 66

Solitamente, se un processo è stato **eseguito con successo**, viene restituito il valore 0 come **exit status**.

Se invece si è **verificato un problema**, viene restituito un valore maggiore di 0, dove tale valore potrebbe assumere un significato specifico scelto dal programmatore

(es: 1 = errore generico, 2 = errore di tipo 1, 3 = errore di tipo 2, ...)

Ogni processo della gerarchia fa **riferimento al proprio padre**:

- Quando un figlio termina o muore, il suo **exit status** viene ritornato al genitore
- Quando un figlio viene creato, esso eredita il codice e parte dello stato del genitore

Attributi ereditati	Attributi non ereditati
RUID, RGID, EUID, EGID	PID, PPID
Working e Root directory	Timer
Ambiente del processo	Record e Memory lock
File descriptors	Contatori delle risorse
Terminale di controllo	Coda dei segnali
Memoria condivisa	Aree di memoria
Codice del programma	

Definizione 33: Processo orfano

Un processo viene detto **orfano** se il suo processo padre è terminato prima di esso.

Tutti i processi orfani vengono "adottati" da `init`, il quale, solitamente, procede col il terminarli direttamente

Durante la sua vita, un processo attraversa obbligatoriamente le seguenti tre fasi:

1. **Running**: vengono eseguite le due istruzioni, per poi terminare ed inviare il proprio exit status al padre
2. **Zombie**: rimane in stato Zombie (Z) (sezione 3.3) fino a quando il genitore non riceve il suo exit status (difatti, il PCB di un processo zombie viene conservato nel kernel per tale scopo)
3. **Terminato**: il genitore ha ricevuto l'exit status e il PCB viene eliminato

Per **duplicare un processo**, la libreria `<unistd.h>` fornisce la syscall `pid_t fork()`. Ogni volta tale syscall viene chiamata, essa viene **eseguita una volta**, ma **ritorna due volte**:

- Al processo chiamante (ossia il padre) viene ritornato il PID del processo figlio creato
- Al processo figlio viene ritornato il PID 0

In caso di errore, invece, al processo chiamante viene ritornato -1 e il figlio non viene creato.

Esempio:

```
#include <unistd.h>

int main(){
    ...
    pid\_t pid = fork();

    /*
        Nel padre, la variabile pid conterrà il PID del figlio
        Nel figlio, la variabile pid conterrà il PID 0
    */

    if(pid > 0){    //viene eseguito dal padre
        ...
    }
    else if(pid == 0){  //viene eseguito dal figlio
        ...
    }
    else{   //viene eseguito in caso di errore
        ...
    }
}
```

Poiché la syscall **fork()** crea **due processi con lo stesso codice**, nel caso in cui si voglia farsì che il processo figlio vada ad eseguire del **codice diverso dal padre**, è necessario **"rimpiazzare" il processo figlio**.

La libreria `<unistd.h>` fornisce la syscall

```
int execve(const char* path, const char* argv[], const char* envp[]);
```

dove:

- `path` è il path assoluto dell'eseguibile da eseguire
- `argv[]` contiene gli argomenti in input del comando da eseguire. Il primo elemento dell'array deve essere il nome dell'eseguibile da eseguire (ossia la parte finale di `path`) e l'ultimo elemento deve essere `NULL`

- `envp[]` contiene l'ambiente di processo della nuova immagine
- Viene sostituita parte dell'immagine del processo con quella dell'eseguibile `path`. In particolare, vengono sostituite le zone di memoria `text` (all'interno della quale viene inserito il codice del programma da eseguire), `bss` e `stack` e i seguenti attributi:

Attributi mantenuti	**Attributi non mantenuti**
PID, PPID, RUID, RGID	EUID, EGID
Session ID	Timer
Terminale di controllo	Memory mappings
Working e Root directory	Memoria condivisa
File lock	Memory lock
File descriptor	
Umask	
Maschera dei segnali	
Segnali in attesa	

Esempio:

```
#include <unistd.h>

int main(){
    pid\_t pid = fork();

    if(pid == 0){
        char* argv[] = {"ls", "-la", NULL};
        execve("/usr/bin/ls", argv, NULL);
    }
}
```

Oltre alla syscall `execve()`, la libreria `<unistd.h>` fornisce le seguenti funzioni di libreria, le quali eseguono al loro interno la `execve()` stessa:

- `int execl(const char* path, const char* arg, ...)`
- `int execlp(const char* path, const char* arg, ..., const char* envp[])`
- `int execle(const char* path, const char* arg, ..., const char* envp[])`
- `int execv(const char* path, const char* argv[])`
- `int execvp(const char* path, const char* argv[])`
- `int execvpe(const char* path, const char* argv[], const char* envp[])`

dove:

- Le funzioni con `l` nel loro suffisso permettono di specificare gli argomenti come una "lista" di stringhe opzionali date in input (`execl()`, `execlp()` e `execle()`). Il primo elemento della "lista" deve essere il nome dell'eseguibile da eseguire (ossia la parte finale di `path`) e l'ultimo elemento deve essere `NULL`

- Le funzioni con `p` nel loro suffisso interpretano `path` come un path relativo invece che assoluto (`execlp()`, `execvp()` e `execvpe()`)
- Le funzioni con `e` nel loro suffisso permettono di specificare l'ambiente di processo della nuova immagine (`execle()` e `execvpe()`)

Osservazione 67

Utilizzare una qualsiasi delle funzioni della famiglia `exec*()` (inclusa anche la syscall), **rimpiazza completamente il text segment del processo**.

Per tanto, tutte le istruzioni successive alla chiamata non verranno eseguite.

5.8 Terminazione di un processo

Per **gestire la chiusura** di un processo, ossia eseguire delle azioni aggiuntive oltre alla restituzione dell'exit status al padre, è possibile utilizzare:

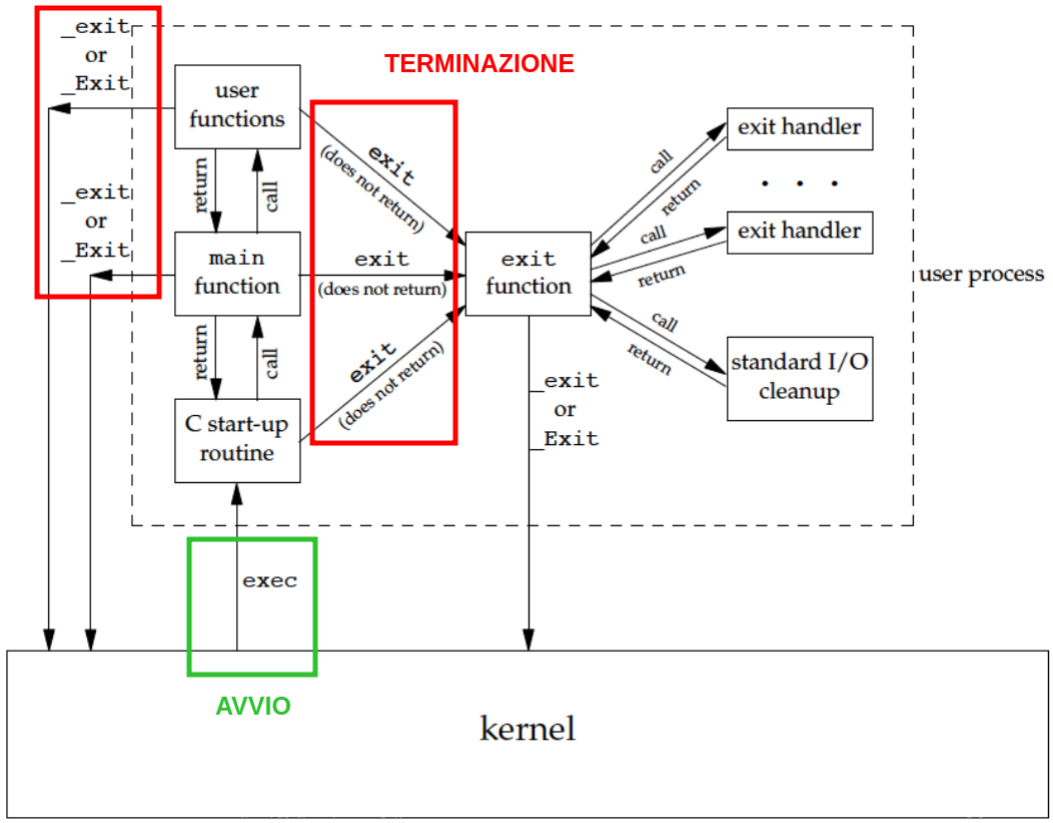
- La syscall `_exit(int status)`, fornita da `<unistd.h>`, esegue in successione le seguenti operazioni:
  1. Chiude tutti i file descriptor del processo chiamante
  2. Termina immediatamente il processo chiamante
  3. I figli non ancora terminati del processo chiamante vengono ereditati da `init`
  4. Invia il segnale `SIGCHLD` al processo padre
  5. Ritorna `status & 0x00FF` come exit status
- La funzione `void exit(int status)`, fornita da `<stdlib.h>`, esegue in successione le seguenti operazioni:
  1. Invoca tutti gli **handler** registrati tramite le seguenti funzioni (fornite sempre da `<stdlib.h>`):
     - `int atexit(void (*func)(void))`
     - `int on_exit(void (*func)(int, void*), void* arg)`
  2. Svuota gli stream standard I/O attivi e li chiude (**clean-up**)
  3. Chiama la syscall `_exit(status)` o `_Exit(status)` (`_Exit()` è una syscall esattamente equivalente ad `_exit()`, con l'unica differenza di essere fornita dalla libreria `<stdlib.h>`)
- La funzione `void abort()`, la quale invia un segnale `SIGABRT` al processo padre, per poi terminare immediatamente il processo chiamante

Osservazione 68

Per maggiore portabilità, libreria `<stdlib.h>` fornisce due macro `EXIT_SUCCESS` e `EXIT_FAILURE` come valori consigliati per il parametro `status` della funzione `exit()`, ma anche dei valori personalizzati possono essere utilizzati.

Osservazione 69

Il segnale `SIGABRT` può essere intercettato e gestito, ma la funzione `abort()` **terminerà comunque il processo**



Ogni processo padre può **attendere cambiamenti di stato** di un suo processo figlio. Un cambiamento di stato avviene quando:

- Il processo figlio è terminato
- Il processo figlio è stato arrestato da un segnale
- Il processo figlio è stato ripristinato da un segnale

Se nel momento in cui un processo padre rimane in attesa del processo figlio quest'ultimo è **terminato**, il sistema **rilascia le risorse associate al figlio** e le sue informazioni (incluso l'exit status) vengono restituite al processo padre. In tal modo, il figlio uscirà dallo stato Zombie (Z) e il suo PCB verrà eliminato.

Inoltre, se lo stato del figlio è già precedentemente cambiato, la chiamata ritornerà immediatamente. Altrimenti, il processo padre rimarrà **sospeso** finché il figlil non cambierà stato o finché la chiamata non verrà interrotta.

Le librerie `<sys/types.h>` e `<sys/wait.h>` forniscono le seguenti due syscall per effettuare l'attesa di un figlio:

- La syscall `pid_t wait(int* status)` sospende l'esecuzione del processo chiamante fino a quando uno dei suoi figli termina.

  Se la chiamata va a buon fine, viene restituito il PID del figlio terminato, altrimenti viene restituito -1

- La syscall `pid_t waitpid(pid_t pid, int* status, int options)` sospende l'esecuzione del processo chiamante fino a quando il figlio specificato da `pid` termina:
  - Se `pid < -1`, viene atteso un qualunque processo figlio il cui GID sia $|pid|$ (ossia il valore assoluto di `pid`)
  - Se `pid = -1`, viene atteso un qualsiasi processo figlio (come per `wait()`)
  - Se `pid = 0`, viene atteso un qualunque processo figlio il cui GID sia uguale al GID del processo chiamante
  - Se `pid > 0`, viene atteso il figlio con PID uguale a `pid`

  Il parametro `options` specifica il comportamento della chiamata ed è il risultato di un bit-wise OR delle seguenti macro:
  - `WNHOHANG`: la chiamata ritorna subito se il figlio richiesto non ha eseguito `exit()`
  - `WUNTRACED`: la chiamata ritorna anche se il figlio richiesto è stato arrestato
  - `WCONTINUED`: la chiamata ritorna anche se il figlio richiesto è stato arrestato

  Se `options = 0`, il processo chiamante rimane in attesa (comportamento di default).

  Inoltre, se la chiamata va a buon fine, viene restituito il PID del figlio terminato, altrimenti viene restituito -1

Entrambe le due syscall, inoltre, memorizzano in `status`, a meno che `status = NULL`, il valore dello **stato del processo figlio** (non il suo exit status).

Dando il valore di `status` in input alle seguenti macro, è possibile ottenere le informazioni inerenti al processo figlio:

- `WIFEXITED(status)`: restituisce `true` se il figlio è terminato senza problemi
- `WEXITSTATUS(status)`: restituisce l'exit status del figlio
- `WIFSIGNALED(status)`: restituisce `true` se il figlio è terminato a seguito della ricezione di un segnale
- `WTERMSIG(status)`: restituisce il numero del segnale che ha terminato il figlio
- `WCOREDUMP(status)`: restituisce `true` se la terminazione del figlio ha generato un core dump

- `WIFSTOPPED(status)`: restituisce `true` se il figlio è entrato in stato Stopped (T) a seguito della ricezione di un segnale
- `WSTOPSIG(status)`: restituisce il numero del segnale che ha messo il figlio in stato Stopped (T)
- `WIFCONTINUED(status)`: restituisce `true` se il processo ha ripreso l'esecuzione a seguito della ricezione di un segnale SIGCONT

Di seguito, viene riportato un esempio di **corretto uso** delle varie syscall relative alla creazione e terminazione di un figlio.

```
void main(){
    pid_t pid = fork();

    if(pid > 0){    // processo padre
        int status;
        int result = waitpid(pid, &status, 0);  //bloccante

        if(result == -1){
            perror("waitpid() call failed");
        }
        else if (!WIFEXITED(status)){
            perror("Child exit failed");
        }
        else{
            int exit_status = WEXITSTATUS(status);
            printf("Child terminated with status %d", exit_status);
        }
    }
    else if(pid == 0){  // processo figlio
        ...

        if(...){
            exit(EXIT_SUCCESS);
        }
        else{
            exit(EXIT_FAILED);
        }
    }
    else{
        perror("fork() failed");
    }
}
```

5.9 Gestione ID dei processi

Le librerie `<sys/types.h>` e `<unistd.h>` forniscono delle syscall in grado di ottenere e modificare gli ID del processo chiamante:

- La syscall `pid_t getpid()` ritorna il PID del processo chiamante
- La syscall `pid_t getppid()` ritorna il PPID del processo chiamante
- La syscall `uid_t getuid()` ritorna il RUID del processo chiamante
- La syscall `uid_t geteuid()` ritorna l'EUID del processo chiamante
- La syscall `int setuid(uid_t uid)` imposta a `uid` l'EUID del processo chiamante.

  Se il processo possiede sufficienti privilegi, anche il RUID e il SUID verranno impostati a `uid`
- La syscall `int seteuid(uid_t uid)` imposta a `uid` l'EUID del processo chiamante
- La syscall `int setgid(gid_t gid)` imposta a `gid` l'EGID del processo chiamante.

  Se il processo possiede sufficienti privilegi, anche il RGID e il SGID verranno impostati a `gid`
- La syscall `int setegid(gid_t gid)` imposta a `gid` l'EGID del processo chiamante

5.10 Gestione dei segnali

Come già discusso, i **segnali** sono **eventi asincroni**, ossia occorrono ad un tempo non predeterminato, che possono essere inviati e ricevuti da un processo.

Ogni processo può **gestire un segnale** associando ad essa una delle seguenti tre azioni:

- **Eseguire** l'azione di default associata al segnale
- **Ignorare** il segnale e proseguire l'esecuzione (a meno che il segnale non sia SIGKILL o SIGSTOP)
- **Catturare** il segnale e richiedere al kernel che venga eseguita una funzione definita dal programmatore stesso (a meno che il segnale non sia SIGKILL o SIGSTOP)

Definizione 34: Maschera dei segnali

Definiamo come **maschera dei segnali di un processo** l'insieme dei **segnali bloccati** all'interno del processo stesso.

I segnali bloccati non vengono scartati, ma vengono invece categorizzati come **pending signal (segnale in arrivo)** e vengono consegnati al processo quando essi vengono sbloccati.

Osservazione 70

I segnali SIGKILL e SIGSTOP non possono essere inseriti all'interno della maschera dei segnali di un processo

Osservazione 71

Ad eccezione del segnale SIGCHILD, le istanze multiple dello stesso segnale non vengono considerate dalla maschera dei segnali.

Per tanto, se un segnale è bloccato e il processo riceve più volte un segnale, verrà considerato **solo un pending signal per tale segnale**.

La libreria `<signal.h>` fornisce la funzione

```
int sigprocmask(int how, const sigset_t* set, sigset_t* oldset)
```

in grado di ottenere/impostare la **maschera dei segnali bloccati**:

- Il parametro `set` è l'insieme dei segnali per cui applicare/rimuovere la maschera
- Il parametro `how` può essere impostato su tre valori:
  - `SIG_BLOCK`: blocca i segnali definiti nel parametro `set`
  - `SIG_UNBLOCK`: sblocca i segnali definiti nel parametro `set`
  - `SIG_SETMASK`: imposta la maschera uguale a `set`, bloccando tutti i segnali definiti in essa e sbloccando tutti gli altri
- L'insieme puntato da `old_set` viene impostato alla precedente maschera

Osservazione 72

La maschera della funzione `sigprocmask()` viene applicata **solo sul processo** e non sui suoi thread, poiché ognuno di essi possiede una propria maschera che va modificata all'interno del thread stesso

Definizione 35: Signal handler

Definiamo come **signal handler** la funzione associata ad un segnale che viene eseguita al posto del normale flusso del processo dopo la ricezione del segnale stesso, per poi riprendere l'esecuzione del processo dal punto in cui è stato interrotto (a meno che l'handler non termini il processo).

Ogni segnale possiede un **default handler**, corrispondente all'azione di default del segnale stesso, ma può essere sovrascritto dal programmatore (a meno che il segnale non sia SIGKILL o SIGSTOP).

Per gestire i signal handler, la libreria `<signal.h>` fornisce:

- Le due macro `SIG_IGN` e `SIG_DFL` definiscono due signal handler:
  - `SIG_IGN` permette di ignorare il segnale
  - `SIG_DFL` assegna il default handler del segnale
- La syscall `sighandler_t signal(int signum, sighandler_t handler)`, la quale imposta il gestore del segnale `signum` alla funzione `handler`. Il parametro `handler` può essere impostato anche sulle macro `SIG_IGN` e `SIG_DFL`.

  ATTENZIONE: tale syscall è deprecata, dunque l'uso è sconsigliato
- La syscall `int sigaction(int signum, const struct sigaction* act, struct sigaction* oldact)` permette di impostare le informazioni relative alla gestione del segnale `signum` allo struct puntato da `act`, dove

```
struct sigaction {
    void (*sa_handler)(int);    /*Puntatore alla funzione signal
                                handler. Può essere SIG_IGN, SIG_DFL
                                o un puntatore a funzione */

    void (*sa_sigaction)(int, siginfo_t*, void*); /*Alternativo
                                                    a sa_handler*/

    sigset_t sa_mask;           /*Specifica la maschera dei segnali
                                che dovrebbero essere bloccati
                                durante l'esecuzione dell'handler*/

    int sa_flags;               /*Flags per modificare il
                                comportamento del segnale*/

    void (*sa_restorer)();      /*Obsoleto*/
};
```

La libreria `<signal.h>` fornisce inoltre le seguenti syscall per la gestione dei segnali:

- La syscall `int kill(pid_t pid, int sig)`, analoga al comando `kill`
- La syscall `unsigned int alarm(unsigned int seconds)` invia un SIGALARM dopo `seconds` secondi
- La syscall `int pause()`, la quale blocca il processo (o thread) chiamante finchè non viene ricevuto un segnale
- La syscall `int sigpending(sigset_t* set)` imposta l'insieme puntato da `set` all'insieme dei pending signal
- La syscall `int sigsuspend(const sigset_t* mask)` sospende il processo che invoca tale syscall e rimpiazza la sua maschera dei segnali con `mask`. Il processo rimane sospeso finchè non arriva un segnale per cui è definito un handler o finchè non arriva un segnale di terminazione

6

Inter Process Communication (IPC)

6.1 Named pipe e Unnamed pipe

I sistemi Unix-like forniscono funzionalità di **inter process communication (IPC)**, ossia modalità di scambio di messaggi e/o dati tra due processi, tramite la **scrittura e lettura sequenziale** seguendo il principio FIFO (First In, First Out), come se i dati scritti venissero inseriti all'interno di una "coda", venendo estratti man mano con la loro lettura.

Definizione 36: Named pipe

Una **named Pipe** (anche chiamata direttamente **FIFO**) è un **file speciale** che può essere utilizzato da più processi e in cui le operazioni di scrittura e lettura sono **full-duplex** (ossia possono avvenire in contemporanea).

Una volta che la named pipe è stata creata, essa può essere utilizzata come un **qualsiasi altro file**. In particolare, essa può essere acceduta in lettura e/o scrittura da **qualsiasi processo** avente i permessi richiesti.

Osservazione 73

Le operazioni di lettura e scrittura devono essere **simultanee**, poiché un processo che apre la FIFO in lettura rimane **bloccato** finchè un altro processo non aprirà la FIFO in scrittura (e viceversa).

Per creare una **named pipe**, le librerie `<sys/types.h>` e `<sys/stat.h>` forniscono la funzione

```
int mkfifo(const char* pathname, mode_t mode)
```

la quale crea una named pipe con nome `pathname` e permessi impostati su `mode`. Inoltre, esiste anche un comando `mkfifo` del tutto analogo alla funzione.

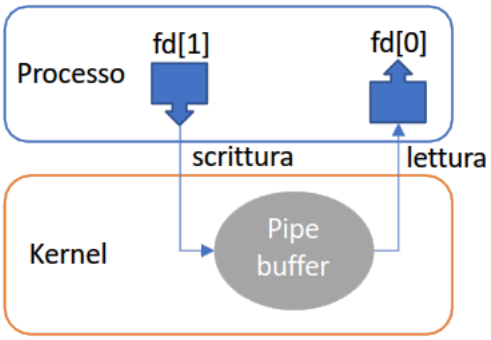
Definizione 37: Unnamed pipe

Un'**unnamed Pipe** (anche chiamata direttamente **pipe**) è una **struttura dati speciale in memoria** in cui le operazioni di scrittura e lettura sono **half-duplex** (ossia non possono avvenire in contemporanea).

Una unnamed pipe può essere creata tramite la syscall `int pipe(int fd[2])` (fornita da `<unistd.h>`), la quale **crea due file descriptor** per poter leggere e scrivere sulla pipe:

- Dopo la chiamata, `fd[0]` conterrà il numero associato al file descriptor per la **lettura** dalla pipe
- Dopo la chiamata, `fd[1]` conterrà il numero associato al file descriptor per la **scrittura** sulla pipe

In particolare, notiamo che dati scritti su una unnamed pipe sono **bufferizzati dal kernel** finchè non vengono letti:



Osservazione 74

Le unnamed pipe risultano principalmente utili nel caso di IPC tra **processi con un antenato in comune**.

Esempio:

- Supponiamo che il processo A crei la pipe `p[]`
- Supponiamo inoltre che il processo A esegua `fork()` creando il processo B, il quale erediterà i file descriptor della pipe `p[]`
- Se A volesse inviare dei dati a B, sarà sufficiente che esso esegua la syscall `write()` col file descriptor `p[1]`.
- Successivamente, per far si che il processo B riceva tali dati, sarà sufficiente che esso esegua la syscall `read()` col file descriptor `p[0]`.

- Traducendo il tutto in codice, abbiamo che:

```
void main(){
    int fd[2];
    pipe(fd);     //apertura della pipe e creazione fd

    int pid = fork();
    if(pid > 0){
        char to_send[13] = "Hello World!";

        write(fd[1], to_send, 13); //il padre scrive sulla pipe
    }
    else if(pid == 0){
        char to_read[13];
        read(fd[0], to_read, 13); //lettura dalla pipe
    }
    else{...}

    close(fd[0]);   //chiusura lettura pipe per entrambi i processi
    close(fd[1]);   //chiusura scrittura pipe per entrambi i processi
}
```

- Volendo estendere l'esempio, qualsiasi altro processo discendente da A (dunque creato da A stesso, da un figlio di A, da un figlio di un figlio di A, ...) potrà comunicare utilizzando la stessa pipe
- Ad esempio, notiamo che:
  - Se A creasse un altro processo C, tutti e tre i processi potrebbero comunicare tra di loro con la stessa pipe
  - Se B creasse un altro processo D, tutti e quattro i processi potrebbero comunicare tra di loro con la stessa pipe

Osservazione 75

Il buffer di un'unnamed pipe possiede una **dimensione massima**:

- Se un processo legge N byte da una pipe **vuota**, esso rimarrà **bloccato** finchè non verranno scritti N byte sulla pipe
- Se un processo scrive M byte su una pipe **piena**, esso rimarrà **bloccato** finchè non verranno letti M byte dalla pipe

Osservazione 76

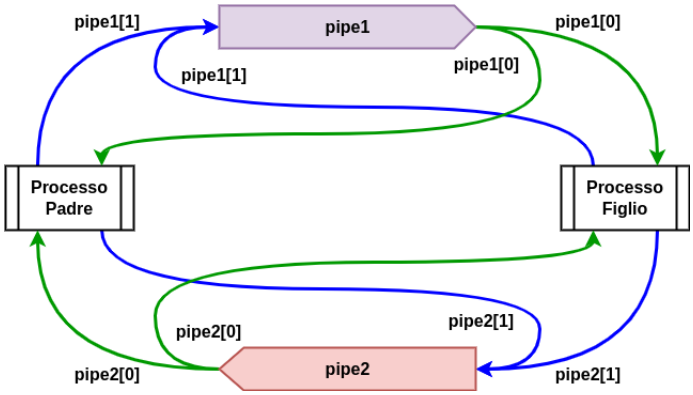
La lettura e la chiusura di una pipe vengono **chiuse** solo quando **tutti i processi** che hanno accesso alla pipe chiudono i propri file descriptor.

Per tanto, se tutti i processi hanno chiuso il file descriptor di lettura, la syscall `read()` ritornerà 0. Se invece tutti i processi hanno chiuso il file descriptor di scrittura, la syscall `write()` ritornerà -1 e verrà inviato un SIGPIPE al processo chiamante.

Essendo le unnamed pipe **half-duplex**, i dati potrebbero **collidere** nel caso in cui entrambi i processi tentino di scrivere in contemporanea o leggere in contemporanea.

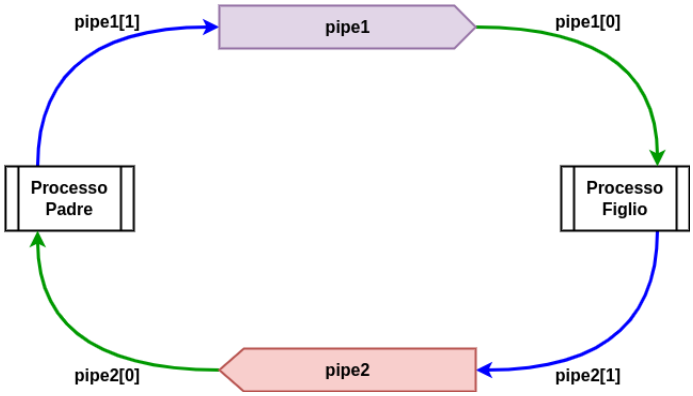
Per tanto, per scambiare dati bidirezionalmente tra due processi è consigliato creare due pipe:

- La pipe `pipe1[]` verrà utilizzata in lettura dal processo figlio e in scrittura dal processo padre.
- La pipe `pipe2[]` verrà utilizzata in lettura dal processo padre e in scrittura dal processo figlio



Inoltre, al fine di prevenire collisioni, è consigliato chiudere immediatamente i file descriptor **inutilizzati**:

- Il processo padre non necessità la lettura sulla pipe `pipe1` e la scrittura sulla pipe `pipe2`, per tanto chiuderà `pipe1[0]` e `pipe2[1]`
- Il processo figlio non necessità la scrittura sulla pipe `pipe1` e la lettura sulla pipe `pipe2`, per tanto chiuderà `pipe1[1]` e `pipe2[0]`



Esempio:

```
void main(){
    int fd[2];
    pipe(fd);     //apertura della pipe e creazione fd

    int pid = fork();
    if(pid > 0){
        close(fd[0]); //il padre non ha bisogno di leggere

        char to_send[13] = "Hello World!";

        write(fd[1], to_send, 13); //il padre scrive sulla pipe

        close(fd[1]); //il padre chiude la scrittura
    }
    else if(pid == 0){
        close(fd[1]); //il figlio non ha bisogno di scrivere

        char to_read[13];

        read(fd[0], to_read, 13); //lettura dalla pipe

        close(fd[0]); //il figlio chiude la lettura
    }
    else{...}
}
```

6.2 Socket

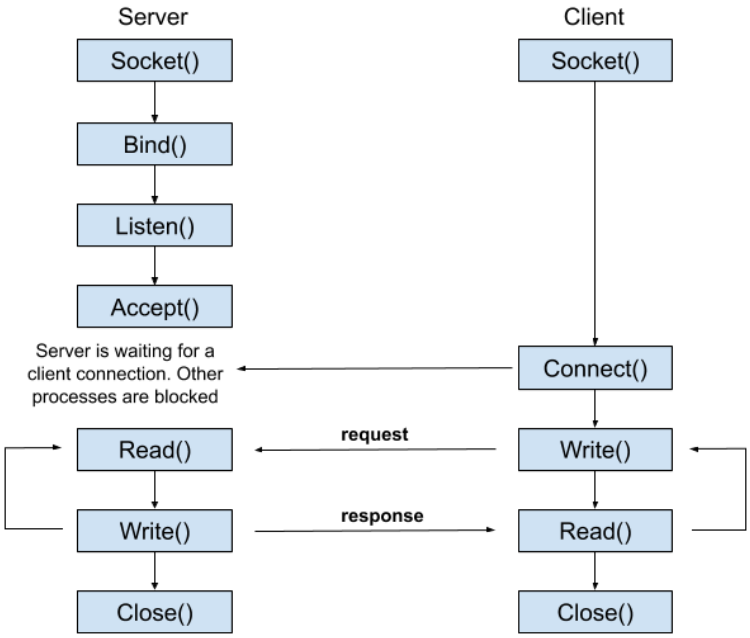
Un **socket** è un'astrazione software che consente la comunicazione tra processi **residenti anche su macchine diverse**. In particolare, l'IPC tramite socket è basata sul **paradigma client-server**:

- Il primo processo assume il ruolo di **server**, mentre tutti gli altri processi assumono il ruolo di **client**
- **Server**:
  - Definisce il socket
  - Il **riferimento** a tale socket (ossia il nome del file o l'indirizzo di rete) è noto ai client
  - Il server **accetta connessioni** sul socket da parte di uno o più client, ricavando un **file descriptor full-duplex** sulla connessione

- **Client**:
  - Definisce il socket
  - Richiede al socket del server di stabilire una connessione, ricavando un **file descriptor full-duplex** su di essa

In particolare, le operazioni svolte all'interno di una comunicazione via socket sono riassumibili in:

1. **Creazione** del socket per il processo server e per il processo client
2. **Associazione (bind)** di un nome al socket del processo server
3. **Ascolto (listen)** sul socket da parte del processo server
4. **Richiesta di connessione (connect)** al socket del processo server da parte del processo client
5. **Accettazione (accept)** della connessione in arrivo sul socket del processo server
6. **Scambio di dati** tramite operazioni di lettura e scrittura dai due socket
7. **Chiusura (close)** dei due socket quando lo scambio di dati è terminato



Osservazione 77

Durante la creazione del socket, viene generato un **socket descriptor**, ossia il file descriptor full-duplex associato al socket.

Per tanto, le operazioni di lettura e scrittura sul socket vengono effettuate tramite le syscall `read()` e `write()`

Osservazione 78

La scrittura o lettura su un **socket chiuso** genera un segnale SIGPIPE

Proposizione 19: Tipologie di socket

Esistono varie tipologie di socket, ognuna dettata dal valore impostato per i seguenti parametri:

- Il **dominio**, ossia la modalità di collegamento:
  - `AF_LOCAL` (o `AF_UNIX`): il client e il server risiedono sulla **stessa macchina**
  - `AF_INET`: il client e il server comunicano in rete tramite il **protocollo IPv4**
  - `AF_INET6`: il client e il server comunicano in rete tramite il **protocollo IPv6**
- Il **tipo**, ossia la semantica del collegamento:
  - `SOCK_STREAM`: il flusso di dati è **bidirezionale, affidabile e basato su connessione** (TCP socket). Supporta anche notifiche asincrone (out of bound)
  - `SOCK_DGRAM`: il flusso di dati è **bidirezionale, non affidabile e senza connessione** (UDP socket). Non richiede l'operazione `accept` da parte del server.
  - `SOCK_RAW`: non viene fornita alcuna proprietà (raw socket)

Definizione 38: Unnamed socket e named socket

Un **unnamed socket** è una struttura dati che rappresenta un socket ma al quale non è associato alcun nome o indirizzo.

Per tanto, eseguendo l'operazione *bind* su un unnamed socket, tale socket diviene un **named socket**

Per effettuare le varie operazioni sui socket, le librerie `<sys/types.h>` e `<sys/socket.h>` forniscono:

- La syscall `int socket(int domain, int type, int protocol)`, la quale definisce un socket, ritornando il suo socket descriptor.

  Il parametro `protocol` specifica il **protocollo** da utilizzare nella comunicazione. Se impostato a `0`, verrà utilizzato il protocollo di default associato al tipo di socket (es: TCP per `SOCK_STREAM` e UDP per `SOCK_DGRAM`)

- La syscall `int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen)`, la quale permette di associare il socket definito all'indirizzo IP o il nome definito da `addr`.

Lo struct `sockaddr` del parametro `addr` può essere di due tipi:

- `struct socaddr_un` se il tipo del socket è `AF_LOCAL`
- `struct socaddr_in` se il tipo del socket è `AF_INET` o `AF_INET6`

```
struct sockaddr_in {
    sa_family_t sin_family; /* address family: AF_INET */
    in_port_t sin_port; /* port in network format*/
    struct in_addr sin_addr; /* internet address */
};

struct in_addr {
    uint32_t s_addr; /* address in network format */
};
```

Il parametro `addrlen` specifica la dimensione dello struct `sockaddr` utilizzato

- La funzione `uint32_t htonl(uint32_t hostlong)` per convertire l'unsigned int `hostlong` in formato network (vedi sopra gli struct `in_addr` e `in_port`)
- La funzione `int inet_aton(const char* cp, struct in_addr* inp)` per convertire l'indirizzo IPv4 `cp` (dunque `cp = "X.Y.Z.W"`) in formato network
- La funzione `struct hostent* gethostbyname(const char* name)` per convertire in formato network il dominio o indirizzo IPv4 contenuto in `name`
-
- La syscall `int listen(int sockfd, int backlog)` marca il socket `sockfd` come *passive*, ossia pronto a ricevere richieste di connessione da accettare. Il parametro `backlog` indica la lunghezza della coda delle richieste in attesa di essere accettate. Restituisce 0 in caso di successo, -1 altrimenti.
- La syscall `int accept(int sockfd, struct sockaddr* addr, socklen_t* addrlen)` estrae la prima richiesta di connessione nella coda in attesa del socket `sockfd`, creando un nuovo socket con connessione associato alla richiesta in arrivo e ritornando il suo socket descriptor.

  Il nuovo socket non è in ascolto, mentre la socket `sockfd` continua ad ascoltare per tutta la durata della syscall.
- La syscall `int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen)` associa l'indirizzo `addr` ad un unnamed socket `sockfd`, ritornando il suo socket descriptor

Osservazione 79

Per chiudere un socket, è sufficiente eseguire la syscall `close()` sul suo socket descriptor

Osservazione 80

Per far si che il server possa rimanere in ascolto di più client in arrivo, è necessario creare un processo figlio (o un thread) per ogni connessione stabilita, affinché il processo padre (o il thread principale) possa continuare ad accettare le connessioni in arrivo

Esempio:

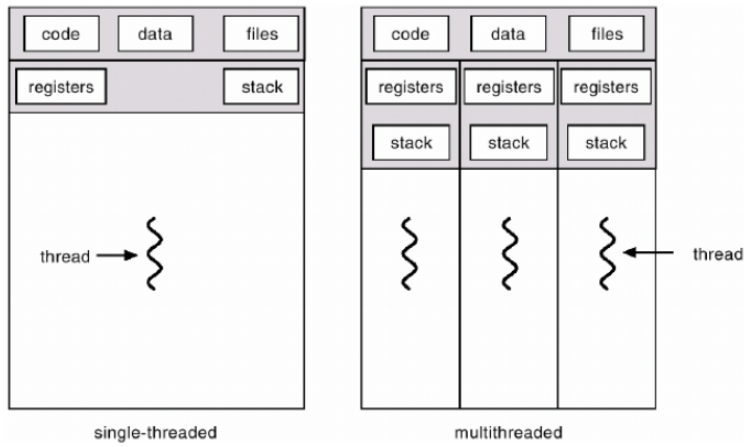
```
----- File: server.c
void main(){
    int sd = socket(AF_INET, SOCK_STREAM, 0);

    bind(sd, ...);
    listen(sd,MAX_QUEUED);

    ...     //disabilito il segnale SIGCHILD

    while (1) {
        int client_sd = accept(sd, ...);

        if (client_sd == -1){
            perror(‘‘Errore accettando connessione dal client’’);
            continue;
        }

        if (fork() == 0) { /*eseguito dal figlio */

            ...     // read/write su client_sd

            close(client_sd);
            exit(0);
        }
    }
}

----- File: client.c
void main(){
    int csd = socket(AF_INET, SOCK_STREAM,0);

    if (connect(csd, ...) != 0) {
        perror(‘‘connessione non riuscita’’);
    }

    ... /* read/write su csd

    close(csd);
}
```

7 Multi-threading

7.1 Processi e thread

In un'applicazione tradizionale, il programmatore definisce un **unico flusso di esecuzione (thread)** delle istruzioni. Le applicazioni **multi-thread**, invece, consentono al programmatore di definire diversi flussi di esecuzione, dove:

- Ciascun thread **condivide le strutture dati principali** dell'applicazione
- Ciascun thread procede in modo **concorrente ed indipendente** dagli altri thread
- Il processo termina solo quando tutti i thread vengono terminati



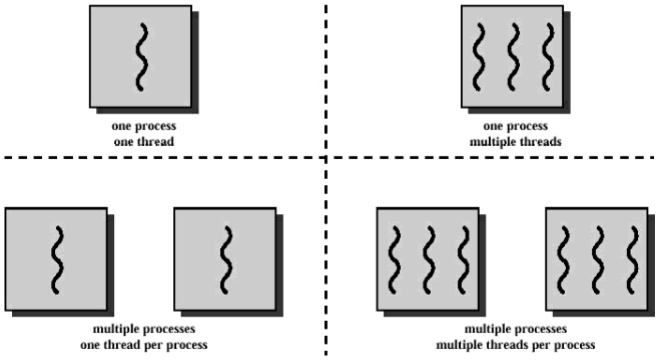
Esempio:

- Un moderno web browser potrebbe essere costituito dai seguenti thread:
  - Thread principale di controllo dell'applicazione
  - Thread per l'interazione con l'utente
  - Thread per la visualizzazionse (rendering) delle pagine in formato HTML

- Thread per la gestione dei trasferimenti di pagine e file dalla rete
- Thread per l'esecuzione dei frammenti di script integrati nelle pagine Web
- Thread per l'esecuzione dei programmi Java, Flash, ...

L'uso dei thread (se utilizzati correttamente) aumenta notevolmente le **performance** di un processo per via dell'elevata presenza di **parallelismo interno** dei calcolatori moderni:

- **Direct Memory Access (DMA)**: trasferimento dati tra memoria primaria e periferiche di I/O senza intervento della CPU
- **Hyper-threading**: supporto a diversi thread, ciascuno con un proprio insieme di registri, alternandosi sulle unità funzionali della CPU
- **Multi-core**: diversi core di calcolo integrati sullo stesso chip, condividendo alcune risorse hardware (tra cui cache di II livello, MMU, ...)
- **Multi-processori**: diverse CPU integrate sulla stessa scheda madre



Generalmente, scrivere applicazioni tradizionali (dunque **single-thread**) che sfruttino a fondo il parallelismo interno al compilatore risulta complesso. Tramite l'uso di applicazioni **multi-thread**, invece, ciascuno dei thread compie il proprio lavoro eseguendo un flusso di istruzioni indipendente tramite le proprie **risorse private** e cooperando con gli altri thread tramite le **risorse condivise**.

Risorse del processo (condivise)	**Risorse del thread (non condivise)**
Spazio d'indirizzamento	Program counter
Variabili globali	Registri
File aperti	Contenuto dello stack
Processi figli	Posizione nello stack
Segnali in arrivo	Stato
Maschera dei segnali	
Handler dei segnali	

Tra le **caratteristiche** delle applicazioni multi-thread, dunque, troviamo:

- **Lightweight**: sono la più piccola unità di lavoro eseguibile dalla CPU
- **Riduzione del tempo di risposta**: anche se un thread di un processo è bloccato in attesa di eventi, un altro thread dello stesso processo può essere eseguito dalla CPU
- **Migliore condivisione delle risorse**: tutti i thread dello stesso processo condividono parte delle risorse del processo stesso (strutture dati in memoria e file aperti) e la comunicazione tra thread è immediata
- **Migliore comunicazione**: la omunicazione tra thread è veloce in quanto i thread dello stesso processo condividono lo stesso indirizzo (e area) di memoria del processo a cui appartengono, rendendo ogni thread in grado di accedere e modificare i dati degli altri thread
- **Maggiore efficienza**: rispetto ad un'applicazione costituita da più processi cooperanti, un'applicazione multi-thread è più efficiente, poiché l'OS gestisce i thread più rapidamente

  (es: nei sistemi Linux-based, creare un thread richiede 1/10 del tempo richiesto per la creazione di un processo)
- **Maggiore scalabilità**: i thread possono sfruttare in modo implicito il parallelismo interno del calcolatore
- **Indipendenza**: il contenuto dei registri della CPU, la posizione e il contenuto dello stack sono privati ed indipendenti per ogni thread

Definizione 39: Kernel thread

Un **kernel thread** viene implementato a livello kernel (ma non necessariamente viene eseguito in kernel mode). Esso è l'unità di esecuzione più piccola in assoluto eseguibile dalla CPU e corrisponde all'astrazione definita all'interno dell'OS per **gestire un flusso di esecuzione**.

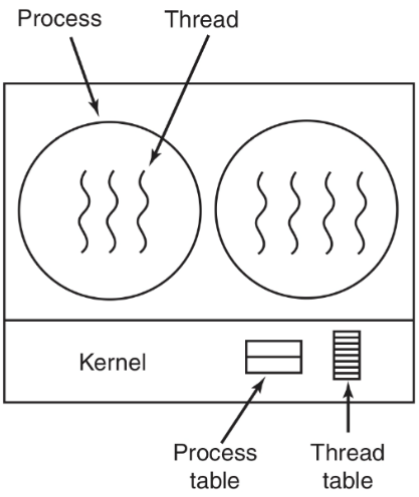
Inoltre, ogni kernel thread è dotato di un **Thread Control Block (TCB)**, contenente le risorse private del thread.

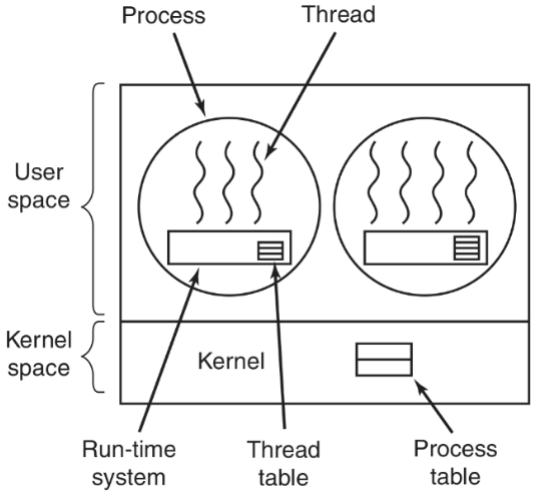
Definizione 40: User thread

Un **user thread** viene implementato a livello utente tramite librerie e gestito come se fosse vero e proprio processo single-thread indipendente. Corrisponde al vero e proprio flusso di esecuzione indipendente dell'applicazione.

Ogni user thread è **legato ad almeno un kernel thread**.

Kernel thread	**User thread**
OS consapevole dei thread	OS ignaro dei thread
Implementazione difficile	Implementazione facile
Context switch lento	Context switch veloce
Una chiamata bloccante di un thread non blocca l'intero processo	Una chiamata bloccante di un thread blocca l'intero processo





Le modalità con cui gli user thread vengono legati ai kernel thread vengono racchiuse in **tre modelli**:

- **Modello 1 ad 1**: ogni user thread è legato ad **un proprio kernel thread**, implicando che ciascun thread possa invocare chiamate bloccanti senza bloccare tutti gli altri thread dello stesso processo.

  Inoltre, poiché il **kernel** si occupa della gestione e della schedulazione dei kernel thread, esso gestisce anche i vari user thread, sfruttando implicitamente il parallelismo interno al calcolatore stesso.

- **Modello molti ad 1**: più user thread dello stesso processo sono legati allo **stesso kernel thread**, implicando che se uno di tali thread invochi una chiamata bloccante tutti gli altri verranno a loro volta bloccati.

  Inoltre, in tal modo il **kernel non è coinvolto** nella gestione dei flussi dell'applicazione, la quale gestisce autonomamente gli user thread, rendendo impossibile sfruttare in modo implicido il parallelismo del calcolatore.

- **Modello molti a molti**: n user thread dello stesso processo sono legati a m kernel thread (dove $m \leq n$), ottenendo un misto dei due precedenti modelli

7.2 POSIX Threads (pthreads)

Per realizzare un'applicazione multi-thread, generalmente il programmatore utilizza una liberia di sistema. Le API offerte da tale liberia non sono direttamente correlate con al tipologia di thread utilizzata (es: possono esistere diverse versioni di una liberia con API identiche ma thread a livello kernel o utente).

La **libreria** `<pthread.h>` è definita dallo standard POSIX, il quale definisce le API ma non stabilisce quale debba essere la loro implementazione in uno specifico OS. In particolare, nei sistemi Linux-based moderni l'ultima implementazione di tale libreria (ossia Native POSIX Threads Library - NPTL) è basata sul **modello 1 ad 1**.

Per **creare un thread**, viene fornita la funzione di liberia

```
int pthread_create(pthread_t* tid, const pthread_attr_t* attr,
          void* (*start_routine)(void*), void* arg)
```

dove:

- Nella variabile puntata da `tid` verrà inserito il Thread ID (TID) del thread creato
- Il parametro `pattr` punta ad una variabile contenente attributi e flag per la creazione del thread
- Il parametro `start_routine` punta alla funzione iniziale eseguita dal thread appena la sua creazione è terminata
- Il parametro `arg` è un puntatore passato come argomento a `start_routine()`

Per **terminare l'esecuzione di un thread**, invece, la funzione

```
void pthread_exit(void* value_ptr)
```

dove il parametro `value_ptr` viene dato come **valore di ritorno** del thread (simile all'exit status di un processo).

Tale funzione viene implicitamente invocata quando la funzione iniziale del thread termina. Inoltre, nel caso in cui venga invocata dall'**ultimo thread** di un processo, il processo stesso terminerà con `exit(0)`.

Osservazione 81

In Unix, la syscall `exit()` termina direettamente il processo chiamante. Nei sistemi Linux-based, invece, si ha che:

- La syscall `_exit()` termina solo il thread in cui viene invocata
- La syscall `exit_group()` termina tutti i thread di un processo
- La funzione `exit()` esegue in realtà `exit_group()` invece che `_exit()`
- Eseguire `return value` in `main()` equivale ad invocare `exit(value)`
- La funzione `pthread_exit()` invoca al suo interno `_exit()`

Per **attendere la terminazione di un thread** (in modo simile a come un processo padre attende un processo figlio tramite `wait()`), viene formita la funzione

```
int pthread_join(pthread_t tid, void* pret)
```

dove `tid` è il TID del thread da attendere e in `pret` viene inserito il valore di ritorno passato nella chiamata `pthread_exit()` del thread atteso.

Osservazione 82

A differenza di `wait()` per i processi figli, non è possibile specificare in `pthread_join()` di attender un thread qualsiasi

Infine, per **ottenere il TID** del thread chiamante, viene fornita la funzione

```
pthread_t pthread_self()
```

Esempio:

```
#include <pthread.h>

int start(long v){
    pthread_t tid = pthread_self();     //ottieni il TID
    printf("Thread %d", tid);
    return v+1;     //il thread viene terminato automaticamente
}

int main(){
    int x = 3;

    pthread_t tid;
    pthread_create(&tid, 0, start, x);
    pthread_join(tid, &x);
    return x;
}
```

Definizione 41: Lightweight Process

Un **Lightweight Process (LWP)** è un processo che condivide alcune risorse selezionate con il proprio processo padre.

L'implementazione dei thread nei sistemi Linux-based è basata sul concetto di LWP.

Per **creare un LWP**, viene fornita la funzione

```
int clone(int (*start)(void*), void* stack, int flags, void* arg, ...)
```

dove:

- `start` è la funzione inziale eseguita all'avvio del nuovo LWP
- `stack` è l'indirizzo della cima dello stack UM del nuovo LWP. Se `stack = NULL`, il LWP creato utilizzerà una copia dello stack del padre
- `flags` indica quale caratteristiche copiare dal processo padre. È un bit-wise OR delle seguenti macro:
  - `CLONE_FILES`: copia i file descriptor
  - `CLONE_FS`: copia le informazioni relative al file system (es: la CWD)
  - `CLONE_SIGHAND`: copia i gestori dei segnali
  - `CLONE_THREAD`: il LWP creato viene inserito nello stesso gruppo thread del chiamante (dunque farà parte dello "stesso processo")
  - `CLONE_VM`: copia lo spazio di memoria
- `arg` è l'argomento da passare in input a `start()`

Osservazione 83

Eseguire una syscall `clone()` utilizzando **nessuna flag** per il parametro `flag` e `NULL` per il parametro `stack`, avrà lo stesso effetto di eseguire la syscall `fork()`.

Eseguire, invece, una syscall `clone()` utilizzando **tutte le flag** per il parametro `flag`, avrà lo stesso effetto di eseguire la funzione `pthread_create()`.

7.3 Concorrenza tra thread

In una applicazione multi-thread, ciascun thread è un flusso di esecuzione in **concorrenza** con quelli degli altri thread. In particolare un sistema operativo multi-programma e pre-emptive, ciascun processo può essere interrotto da un altro processo e diversi processi o gestori di interruzione sono in esecuzione contemporaneamente.

La coerenza delle **strutture dati private** di ciascun flusso di esecuzione è garantita dal meccanismo del context switch, mentre lo stesso non vale per le **strutture di dati condivise**.

Per tanto, se due o più flussi di esecuzione hanno una struttura di dati in comune, la concorrenza dei flussi può determinare uno **stato della struttura non coerente** con la logica di ciascuno dei flussi.

Definizione 42: Race condition

Una **race condition** è una situazione in lo stato della memoria condivisa tra due o più flussi di esecuzione concorrenti **dipende dall'ordine esatto degli accessi** alla memoria stessa (temporizzazione)

Esempio:

- Supponiamo che il flusso #1 e il flusso #2 condividano una variabile `counter`
- Ogni incremento e decremento di tale variabile prevede l'esecuzione di tre operazioni:
  - **Load**: trasferimento del valore di `counter` dalla memoria ad un registro
  - **Update**: incremento o decremento del valore del registro
  - **Store**: trasferimento del valore del registro in `counter`
- Se non vi sono controlli sull'ordine di accesso alla variabile `counter`, i due flussi potrebbero accedervi in contemporanea, ottenendo due valori separati e incoerenti

Istante	Flusso #1	Flusso #2
1	$R_0 \leftarrow$ `counter` (2)	
2	$R_0 \leftarrow R_0 + 1$ (3)	
3		$R_1 \leftarrow$ `counter` (2)
4		$R_1 \leftarrow R_1 - 1$ (1)
5		`counter` $\leftarrow R_1$ (1)
6	`counter` $\leftarrow R_0$ (3)	

Definizione 43: Sezione critica

Una **sezione critica** è una sequenza di istruzioni di un flusso di esecuzione che accede ad una **risorsa condivisa** e che non deve essere eseguita in modo concorrente ad un'altra sezione critica

Definizione 44: Semaforo e Mutex

Un **semaforo** è una struttura dati utilizzata per consentire o impedire gli accessi ad una sezione critica (e di conseguenza alla risorsa condivisa):

- Viene inizializzata una variabile intera contatore con un valore $n > 0$.

  Se $n > 1$, allora n flussi possono accedere contemporaneamente alla risorsa condivisa.

  Se $n = 1$, allora solo un flusso alla volta può accedere alla risorsa condivisa. Inoltre, in tal caso definiamo tale semaforo come **Mutex (Mutual Exclusion)**

- Tramite la primitiva **Wait()**, un thread attende che il contatore sia positivo, per poi decrementarlo ed accedere alla risorsa condivisa.

  Se il semaforo è un mutex, tale primitiva viene anche detta **Lock()**.

- Tramite la primitica **Signal()**, viene incrementato il contatore (assumendo che la sezione critica abbia già smesso di utilizzare la risorsa condivisa).

  Se il semaforo è un mutex, tale primitiva viene anche detta **Unlock()**

La libreria `<pthread.h>` fornisce delle funzioni per l'implementazione dei mutex:

- La funzione `int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* mutexattr)` inizializza un mutex e imposta i suoi attributi pari a `mutexattr`, i quali determinano il comportamento del semaforo quando un thread invoca un lock/unlock più volte consecutivamente:
  - Se `*mutexattr = PTHREAD_MUTEX_NORMAL`, ogni operazione di lock blocca il thread finchè il lock precedente non viene rilasciato (creando eventuali stalli), mentre ogni operazione di unlock rilascia il semaforo e ritorna subito
  - Se `*mutexattr = PTHREAD_MUTEX_RECURSIVE`, viene concesso allo stesso thread di mettere più lock (un contatore tiene conto del numero di lock messi), mentre ogni unlock decrementa il contatore e rilascia il semaforo quando il contatore è 0
  - Se `*mutexattr = PTHREAD_MUTEX_ERRORCHECK`, viene generato un errore nel caso in cui un thread cerchi di mettere un lock ma il mutex detenga già un lock e se viene effettuato un unlock da parte di un thread che non aveva precedentemente effettuato un lock del mutex
  - Se `mutexattr = NULL`, viene impostato `PTHREAD_MUTEX_NORMAL` come valore di default
- La funzione `int pthread_mutexattr_settype(pthread_mutexattr_t* attr, int type)` permette di impostare `attr` pari a `PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_RECURSIVE` o `PTHREAD_MUTEX_ERRORCHECK`
- La funzione `int pthread_mutex_lock(pthread_mutex_t* mutex)` rimane in attesa che `mutex` sia disponibile ed effettua un lock su di esso

- La funzione `int pthread_mutex_lock(pthread_mutex_t* mutex)` rimane in attesa che `mutex` sia disponibile ed effettua un lock su di esso
- La funzione `int pthread_mutex_trylock(pthread_mutex_t* mutex)` ritorna subito se è già presente un lock su `mutex`, altrimenti effettua un lock su di esso
- La funzione `int pthread_mutex_unlock(pthread_mutex_t* mutex)` effettua un unlock su `mutex`
- La funzione `int pthread_mutex_destroy(pthread_mutex_t* mutex)` elimina completamente la struttura dati

Esempio:

```
struct shared_resource {
    int var;
    pthread_mutex_t mutex;
} shared_res;

void start(){
    pthread_t tid =; //ottieni tid

    //inizio sezione critica
    for(int i = 0; i < 10; i++){
        pthread_mutex_lock(&shared_res.mutex); //attesa del lock

        shared_res.var++;
        printf("Shared Var (Thread %d): %d\n", tid, shared_res.var);

        pthread_mutex_unlock(&shared_res.mutex); //rilascio lock

        sleep(1);   //assicura che l'altro thread
                    //prenda il controllo della CPU
    }
}

void main(){
    pthread_t t1, t2;

    shared_res.var = 0; //inizializzazione variabile condivisa
    pthread_mutex_init(&shared_res.mutex, NULL); //creazione mutex

    pthread_create(&t1, NULL, start, NULL);     //avvia thread 1
    pthread_create(&t2, NULL, start, NULL);     //avvia thread 2
    pthread_join(t1, NULL);     //attendi thread 1
    pthread_join(t2, NULL);     //attendi thread 2

    pthread_mutex_destroy(&shared_res.mutex);
}
```

7.4 Sincronizzazione tra thread

Definizione 45: Barriera

Una **barriera** è una struttura dati utilzzata per **bloccare momentaneamente** il flusso di esecuzione di n processi o thread **finché tutti i thread partecipanti non hanno raggiunto la barriera**:

- Viene inizializzata una variabile intera contatore con valore $n > 0$
- Tramite la primitva **Wait()**, il thread chiamante raggiunge la barriera (il contatore viene decrementato) e rimane in attesa che $n = 0$
- Se $n = 0$, la barriera viene **superata** e tutti i thread in attesa vengono sbloccati e n viene **ripristinato al valore iniziale**

Esempio:

- Supponiamo che il flusso #1, il flusso #2 e il flusso #3 condividano una barriera
- Se il flusso #1 esegue la primitiva Wait(), esso rimarrà in attesa che tutti i thread rimanenti raggiungano la barriera (ossia finché anche il flusso #2 e #3 non avranno eseguito la primiiva Wait())
- Se anche il flusso #2 esegue la primitiva, i due flussi in attesa verranno sbloccati solo quando il flusso #3 eseguirà Wait()
- Quando tutti e tre i flussi hanno eseguito Wait(), la barriera viene superata ed essi vengono tutti sbloccati

La liberia `<pthread.h>` fornisce le seguenti funzioni per l'implementazione delle barriere:

- La funzione `int pthread_barrier_init(pthread_barrier_t* barrier, const pthread_barrierattr_t* attr, unsigned int count)` crea una nuova barriera con attributi `attr` per `count` thread

  Se `attr = NULL`, vengono impostati gli attributi di default

- La funzione `int pthread_barrier_wait(pthread_barrier_t* barrier)` permette al thread chiamante di raggiungere la barriera

  Quando la barriera viene superata, viene restituito `PTHREAD_BARRIER_SERIAL_THREAD` ad un thread in attesa sulla barriera selezionato casualmente, mentre viene restituito 0 a tutti gli altri thread in attesa

- La funzione `int pthread_barrier_destroy(pthread_barrier_t* barrier)` elimina completamente la struttura dati

Esempio:

```
pthread_barrier_t brr;

void do_stuff(){
    sleep(1);
    pthread_barrier_wait(&brr);
    printf("Barriera superata (Thread %d)\n", pthread_self());
}

void do_longer_stuff_2(){
    sleep(3);
    pthread_barrier_wait(&brr);
    printf("Barriera superata (Thread %d)\n", pthread_self());
}

void main(){
    pthread_t t1, t2;

    pthread_barrier_init(&brr, NULL, 2);

    pthread_create(&t1, NULL, do_stuff, NULL);
    pthread_create(&t2, NULL, do_longer_stuff, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_barrier_destroy(&brr);
}
```

7.5 Condizioni tra thread

Definizione 46: Condizione

Una **condizione** è una struttura dati utilizzata per **bloccare momentaneamente** il flusso di esecuzione di un processo finché un **predicato non è verificato**:

- Tramite la primitiva **Wait()**, il thread chiamante viene messo in attesa
- Tramite la primitiva **Signal()** permette di risvegliare un thread specifico tra quelli in attesa
- Tramite la primitiva **Broadcast()** risveglia tutti i thread in attesa

Esempio:

- Supponiamo che il flusso #1 e il flusso #2 condividano una condizione
- Se il flusso #1 esegue la primitiva Wait(), esso rimarrà in attesa che venga eseguita la primitva Signal() su di esso (o Broadcast()) da parte del flusso #2

Osservazione 84

Una condizione deve essere **sempre associata ad un mutex**, in modo da evitare race condition sulla condizione stessa

(es: un thread esegue Wait() sulla condizione ed un altro thread esegue Signal() sulla condizione prima che il primo thread venga effettivamente messo in attesa)

La liberia `<pthread.h>` fornisce le seguenti funzioni per l'implementazione delle condizioni:

- La funzione `int pthread_cond_init(pthread_cond_t* cond, pthread_condattr_t* attr)` crea una nuova condizione con attributi `attr`.

  Se `attr = NULL`, vengono impostati gli attributi di default
- La funzione `int pthread_cond_signal(pthread_cond_t* cond)` risveglia un thread casuale tra quelli in attesa della condizione
- La funzione `int pthread_cond_broadcast(pthread_cond_t* cond)` risveglia tutti i thread in attesa della condizione
- La funzione `int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex)` esegue unlock su `mutex` e attende di essere sbloccato da un altro thread, eseguendo automaticamente lock su `mutex` dopo essere stato sbloccato.

  Attenzione: il thread chiamante deve aver prima eseguito lock su `mutex`
- La funzione `int pthread_cond_destroy(pthread_cond_t* cond)` elimina completamente la struttura dati