SISTEMI OPERATIVI

APPUNTI DI LABORATORIO

A.A. 2022/2023

Lorenzo Mancinelli

SOMMARIO

APPUNTI DELLE LEZIONI DI LABORATORIO INTEGRATI CON:

- Slide del docente e materiale didattico reperibile sul canale Teams ufficiale del corso https://teams.microsoft.com/l/channel/19%3a5pG0GY64qo5NUWzwgyDWs7Bud016KfZ2ZZ_rPukgSNo1%40thread.tacv2/Generale?groupId=504812ae-c756-4f3d-a8ec-118f6073ef1d&tenantId=c7456b31-a220-47f5-be52-473828670aa1 nella sezione
  `Generale > File > Laboratorio`

ESERCITAZIONE 1: INTRODUZIONE AI SISTEMI UNIX/LINUX

UTENTE ROOT:

- Amministratore del sistema.
- Può compiere qualsiasi tipo di operazione.

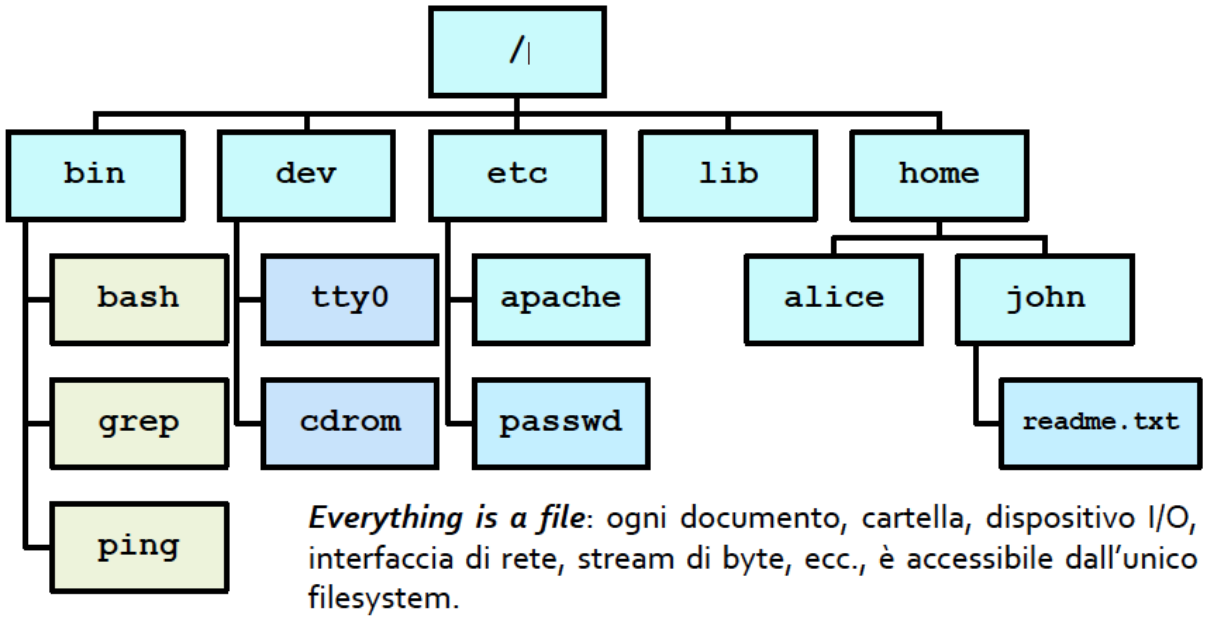
UTENTI NORMALI:

- Utilizzatori del sistema.
- Hanno privilegi limitati.

FILE SYSTEM

Tutti i dischi vengono resi accessibili (montati) tramite un unico filesystem virtuale.

/	Directory principale.
/home	Contiene le varie home directory degli utenti.
/sbin	Contiene i programmi di sistema.
/etc	Contiene i file di configurazione.
/media	Rende accessibili i supporti rimovibili. `/media/cdrom` `/media/kingston8gb`



Everything is a file: ogni documento, cartella, dispositivo I/O, interfaccia di rete, stream di byte, ecc., è accessibile dall'unico filesystem.

COME DESCRIVERE UN PERCORSO (PATH) DEL FILESYSTEM:

- **PERCORSO ASSOLUTO:** si esprime l'intero percorso partendo dalla radice.
  `/home/alice/Documents/todolist/groceries.txt`

- **PERCORSO RELATIVO:** si esprime il percorso a partire dalla directory in cui mi trovo.
  `Documents/todolist/groceries.txt`

- Unix è case-sensitive.

`~`	La nostra home directory.
`.`	La directory corrente.
`..`	La directory padre.

SHELL: BASH

Il prompt sarà:

```
alice@studio:~/Documents$
```

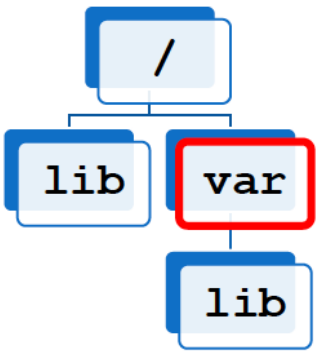
`$`	Utente normale.
`#`	Utente root.

ACCESSO AL SISTEMA	
`login`	Si accede usando username e password.
`logout (Ctrl+D)`	Per uscire dalla sessione.
`TAB`	Auto-completamento di comandi e directory.
`Frecce su/giù`	History dei comandi recenti.
`Ctrl+R`	Ricerca attraverso la storia.
`Ctrl+Alt+T`	Emulatore di terminale.
`Ctrl+Alt+F1,F2…`	Terminali virtuali (in Debian F7 è l'interfaccia grafica).

ARRESTO E RIAVVIO	
`shutdown`	Per arrestare o riavviare il sistema (solo l'utente root può invocarlo).
`#shutdown -h now`	Arresto.
`#shutdown -r now`	Riavvio.

COMANDI DI BASE

COMANDO cd: change directory, consente di passare da una directory all'altra.



`$cd /lib`	Path assoluto, vado in /lib.
`$cd lib`	Path relativo, vado in /var/lib.
`$cd ..`	Vado nella directory padre, cioè /.
`$cd ($cd ~)`	Vado nella mia home.

COMANDO pwd: print working directory, stampa il percorso assoluto della directory corrente.

Sempre facendo riferimento alla figura precedente:

COMANDO	OUTPUT
`$pwd`	/var
`$cd lib` `$pwd`	/var/lib
`$cd ../..` `$pwd`	/

COMANDO ls: list, serve per elencare il contenuto della directory specificata (se non si specifica nulla, elenca la directory corrente).

- Si possono usare percorsi assoluti o relativi.
- Si possono specificare più percorsi.
  `ls /etc /var`

- Spesso file e cartelle sono di colori diversi.

`$ls -l`	Mostra dettagli (permessi, proprietario, dimensioni, data di ultima modifica).
`$ls -a`	Mostra anche i file nascosti (cioè il cui nome inizia con `.`).
`$ls -a -l` `$ls -al`	Le opzioni sono cumulabili.

METACARATTERI: si usano per indicare insiemi di file o cartelle.

`*`	Sostituisce zero o più caratteri.
`?`	Sostituisce un singolo carattere.
`[a,b,c]` `[a-z]`	Sostituisce un carattere nell'insieme specificato (anche con cifre).

COMANDO	OUTPUT
`$ls *.c`	aa.c abc.c a.c axc.c
`$ls ?.?`	a.c a.h
`$ls a[b-t]c.c`	abc.c
`$ls a[4,f,x]c.c`	axc.c

COMANDO man: mostra cosa fa un comando e come si usa.

```
$man nome_comando
```

- Il manuale contiene la descrizione esaustiva del comando, la sintassi, le opzioni, i messaggi di errore.
- È diviso in sezioni.
- **Non è solo** per i comandi.
- Serve specificare la sezione se ci sono ambiguità:
  - `$ man printf` va al comando.
  - `$ man 3 printf` va alla funzione C.

`whatis`	Visualizzare la descrizione breve di una pagina del manuale. Indica anche le ambiguità e le sezioni giuste.
`apropos`	Cercare una parola in nomi e descrizioni.

- `whatis` si usa per sapere velocemente cosa fa un comando, `apropos` per sapere che comandi ho a disposizione per fare qualcosa.

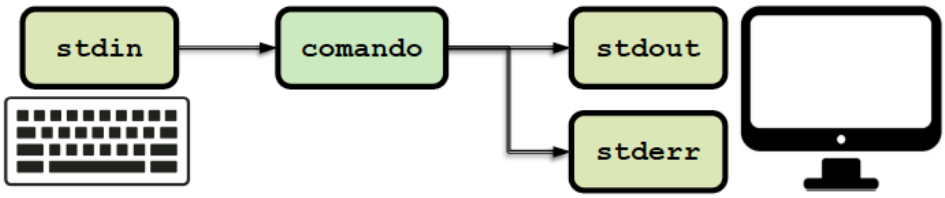
COMANDI SU FILE E DIRECTORY	
`$mkdir nome_dir`	Crea una directory.
`$rmdir nome_dir`	Rimuove una directory, solo se vuota.
`$cp src dst`	Copia un file in un nuovo file (di nome dst) o all'interno di una directory.
`$cp src1 src2 … dst_dir`	Copia più file o directory in un'unica directory.
`$mv src dst`	Rinomina un file o una directory.
`$mv src1 src2 … dst_dir`	Sposta più file o directory in un'unica directory.
`$touch nome_file`	Aggiorna il timestamp di accesso e modifica di un file. Se il file non esiste, viene creato.
`$cat file1 file2 …`	Concatena il contenuto di due o più file e li stampa nello standard output. ! Può essere utile per visualizzare velocemente file brevi.

`$rm file1 file2 …`	Rimuove file o directory. ! In mancanza di opzioni, le cartelle non vengono rimosse: per rimuovere una cartella e tutto il suo contenuto, usare -r

LETTURA DI FILE	
`less`	Visualizzare "un po' alla volta" e interattivamente. • Pagina successiva: f o barra spaziatrice. • Verso il basso per un numero specifico di righe: numero + f o barra spaziatrice. • Scorrere in avanti/dietro di una riga: freccia giù o invio/freccia su. • Pagina precedente: b. • Verso l'alto per un numero specifico di righe: numero + b. • Cercare una parola: / + parola. Premendo invio si cerca in avanti. • Per cercare all'indietro: ? + parola. • Quando si arriva alla fine del file viene visualizzato (END). • Per uscire: q.
`head` `tail`	Visualizzare la prima/ultima parte di uno o più file. Possiamo specificare il numero di byte da mostrare con -c o il numero di righe con -n (di default 10 righe).

REDIREZIONE I/O: i processi hanno 3 canali di input/output standard.

1. **stdin** – input da tastiera.
2. **stdout** – output su schermo.
3. **stderr** – messaggi di errore su schermo.



È possibile deviare l'output di un comando verso un file oppure acquisire l'input da un file.

REDIREZIONE DELL'OUTPUT	
`>`	Invia lo stdout a un file (prende l'output del comando a sinistra e lo scrive nel file a destra). • Se il file non esiste, viene creato. • Se il file esiste, viene sovrascritto. Es: `$ls -l > filelist.txt`
`2>`	Come sopra, ma per lo stderr.
`&>`	Come sopra, ma per entrambi.
`$comando > out.txt` `2> errors.txt`	Si possono inviare i due output su file diversi.

REDIREZIONE DELL'INPUT	
<	Recupera l'input da un file. Es: `$sort < list.txt` Legge uno o più file di testo (o lo standard input) e ne ordina le linee alfabeticamente oppure secondo il criterio specificato, producendo il risultato sullo standard output o su di un file. Si può usare in combinazione con > `$sort < list.txt > sortedlist.txt`

PIPELINE	
|	(pipe) collega l'output di un comando con l'input del successivo. `$ls -l mydir \| less` Si può usare più volte e in combinazione con le altre redirezioni. `$cat *.txt | sort | uniq > result-file`

`su`	Switch user, serve per accedere al terminale di un altro utente. • Se non specificato, si accede al terminale di root. • Viene chiesta la password dell'utente con cui si vuole accedere.
`sudo nome_comando`	Serve per lanciare un comando come un altro utente. • Se non specificato, si usa l'utente root. • Viene chiesta la password dell'utente corrente. • L'utente deve fare parte del gruppo sudoers.

ESERCIZI

ESERCIZIO 1

- Aprite un terminale.
- Create la directory **Esercitazione1.**

```
studenti@studenti:~$ mkdir Esercitazione1
```

- Create, *senza usare un editor*, un file **esercitazione.txt** all'interno di **Esercitazione1** che contenga la parola "**Esercizio**"
  - Per stampare parole usate `echo parola`.

```
studenti@studenti:~$ echo Esercizio > Esercitazione1/esercitazione1.txt
```

- Visualizzate il contenuto del file **esercitazione.txt** usando il comando `less` (Passate a `less` prima il path relativo e poi il path assoluto del file)

```
studenti@studenti:~$ less Esercitazione1/esercitazione1.txt
```

```
studenti@studenti:~$ less /home/studenti/Esercitazione1/esercitazione1.txt
```

- Spostatevi in **Esercitazione1** e subito dopo usate un comando per tornare nella vostra home.

```
studenti@studenti:~$ cd Esercitazione1/
studenti@studenti:~/Esercitazione1$ cd ..
```

ESERCIZIO 2

- Visualizzate il percorso della directory corrente.

```
studenti@studenti:~$ pwd
/home/studenti
```

- Spostatevi in **Esercitazione1** e create 3 file **f1.txt**, **f2.txt**, **f3.txt** contenenti rispettivamente le parole **Uno**, **Due**, e **Tre.**

```
studenti@studenti:~$ cd Esercitazione1/
studenti@studenti:~/Esercitazione1$ echo Uno > f1.txt
studenti@studenti:~/Esercitazione1$ echo Due > f2.txt
studenti@studenti:~/Esercitazione1$ echo Tre > f3.txt
```

- Con *un solo comando* create il file **f_tot.txt** partendo da **f1.txt**, **f2.txt**, **f3.txt** fatto come segue, e visualizzatene il contenuto:
  - Uno (a capo) Due (a capo) Tre.

```
studenti@studenti:~/Esercitazione1$ cat f*.txt > f_tot.txt
```

- Cancellate i file **f_tot.txt**, **f1.txt**, **f2.txt**, **f3.txt.**
- Adesso create il file **fcitta.txt** fatto come segue:
  - Milano (a capo) Perugia (a capo) Asti.

```
studenti@studenti:~/Esercitazione1$ echo Milano > fcitta.txt
studenti@studenti:~/Esercitazione1$ echo Perugia >> fcitta.txt
studenti@studenti:~/Esercitazione1$ echo Asti >> fcitta.txt
```

- Visualizzate il contenuto di **fcitta.txt** in ordine alfabetico

```
studenti@studenti:~/Esercitazione1$ sort fcitta.txt
Asti
Milano
Perugia
```

- Salvate il contenuto di **fcitta.txt** ordinato in un file **fcittaord.txt**

```
studenti@studenti:~/Esercitazione1$ sort fcitta.txt > fcittaord.txt
```

ESERCIZIO 3

- Usando la funzione di autocompletamento della shell passate **fcittaord.txt** al comando `less`. Fino a che punto riesce ad aiutarvi?

```
studenti@studenti:~/Esercitazione1$ less f
f1.txt          f2.txt          f3.txt          fcittaord.txt
fcitta.txt      f_tot.txt
studenti@studenti:~/Esercitazione1$ less fcitta
fcittaord.txt  fcitta.txt
studenti@studenti:~/Esercitazione1$ less fcittaord.txt
```

- Create un file **fcitta.c** e due cartelle **Testi** e **Sorgenti.**

```
studenti@studenti:~/Esercitazione1$ touch fcitta.c
studenti@studenti:~/Esercitazione1$ mkdir Testi
studenti@studenti:~/Esercitazione1$ mkdir Sorgenti
```

- Usando i metacaratteri copiate in **Testi** tutti i file **.txt** ed in **Sorgenti**i file **.c.**

```
studenti@studenti:~/Esercitazione1$ cp *.txt Testi/
studenti@studenti:~/Esercitazione1$ cp *.c Sorgenti/
```

- Cancellate tutti i file di testo della directory **Esercitazione.**

```
studenti@studenti:~/Esercitazione1$ rm *.txt
```

- Create 3 file chiamandoli **fa.txt**, **fb.txt**, **fc.txt.**

```
studenti@studenti:~/Esercitazione1$ touch fa.txt fb.txt fc.txt
```

- Usate un'espressione che permetta di spostare solo **fa.txt** ed **fc.txt** e non **fb.txt** nella cartella **Testi.**

```
studenti@studenti:~/Esercitazione1$ mv f[a,c].txt Testi/
```

- Eliminate **fc.txt.**

```
studenti@studenti:~/Esercitazione1$ rm Testi/fc.txt
```

ESERCIZIO 4

- Cancellate i file della cartella **Sorgenti.**

```
studenti@studenti:~/Esercitazione1$ rm Sorgenti/*
```

- Usando `rmdir` eliminate le cartelle **Testi** e Sorgenti.

- Ci riuscite?

```
studenti@studenti:~/Esercitazione1$ rmdir Sorgenti/ Testi/
rmdir: rimozione di "Testi/" non riuscita: Directory non vuota
studenti@studenti:~/Esercitazione1$ rm -r Testi/
```

- Create una cartella **sotto** e, dentro **sotto**, una cartella **sotto1**.

```
studenti@studenti:~/Esercitazione1$ mkdir -p sotto/sotto1
```

- Usate il manuale per trovare l'opzione di `rmdir` che permette di cancellare con lo stesso comando **sotto** e **sotto1.**

```
studenti@studenti:~/Esercitazione1$ rmdir -p sotto/sotto1/
```

- Create una cartella **origine** e dentro origine create la cartella **sotto_origine** ed il file **qwerty.txt.**

```
studenti@studenti:~/Esercitazione1$ mkdir -p origine/sotto_origine
studenti@studenti:~/Esercitazione1$ touch origine/sotto_origine/qwerty.txt
```

- Create la directory **destinazione** e copiate al suo interno *il contenuto* di **origine**. Se usate `cp` senza opzioni cosa succede? Come dovete fare?

```
studenti@studenti:~/Esercitazione1$ mkdir destinazione
studenti@studenti:~/Esercitazione1$ cp origine/* destinazione/
cp: directory "origine/sotto_origine" omessa
studenti@studenti:~/Esercitazione1$ cp -r origine/* destinazione/
```

- Adesso copiate non solo il contenuto ma tutta la cartella **origine** in **destinazione**.

```
studenti@studenti:~/Esercitazione1$ cp -r origine/ destinazione/
```

ESERCIZIO 5

- Visualizzate il contenuto di **destinazione.**

```
studenti@studenti:~/Esercitazione1$ ls destinazione/
```

- Adesso usate l'opzione di `ls` che visualizza anche i permessi.

```
studenti@studenti:~/Esercitazione1$ ls -l destinazione/
```

- All'interno di **destinazione** create il file **.youcantseeme.**

```
studenti@studenti:~/Esercitazione1$ touch destinazione/.youcantseeme
```

- Visualizzatelo con `ls`.

```
studenti@studenti:~/Esercitazione1$ ls -a destinazione/
```

- Salvate l'output di `ls/etc` in un file **ls_output.txt.**

```
studenti@studenti:~/Esercitazione1$ ls /etc/ > ls_output.txt
```

- Visualizzate
  - Solo la parte iniziale del file
  - Solo la parte finale
  - Solo la prima riga
  - Solo le ultime 2 righe

```
studenti@studenti:~/Esercitazione1$ head ls_output.txt
studenti@studenti:~/Esercitazione1$ tail ls_output.txt
studenti@studenti:~/Esercitazione1$ head -n1 ls_output.txt
studenti@studenti:~/Esercitazione1$ tail -n2 ls_output.txt
```

- Con *un solo comando* salvate sul file **terza.txt** solo la terza riga del file.

```
studenti@studenti:~/Esercitazione1$ head -n3 ls_output.txt | tail -n1 > terza.txt
```

ESERCITAZIONE 2: UTENTI E GRUPPI, PERMESSI DI ACCESSO AL FILESYSTEM, EDITOR DI TESTO

UTENTI E GRUPPI

Ogni **utente** è identificato da:

1. Username.
2. UID (user ID) numerico.

Ogni **gruppo** è identificato da:

1. Group name.
2. GID (group ID) numerico.

Ogni utente deve appartenere almeno ad un gruppo (primary group).

GESTIONE UTENTI

COMANDI VARI

`passwd`	Permette di cambiare password (sfrutta il permesso SUID).
`id [username]`	Visualizza UID, gruppo principale e altri gruppi dell'utente corrente o di quello selezionato.
`groups [username]`	Visualizza i nomi dei gruppi dell'utente corrente o di quello selezionato.

CREAZIONE E RIMOZIONE

È necessario avere i privilegi root.

`adduser username`	Creazione di un utente.
`deluser username`	Rimozione di un utente.

SU E SUDO

`su`	Switch user, permette di accedere al terminale di un altro utente, o dell'utente root. • **`su username`.** • Se l'utente non è specificato si richiede di accedere al terminale di root. • Viene chiesta la password dell'utente specificato.
`sudo nome_comando`	Permette di eseguire un comando con i privilegi di un altro utente. • **`sudo -u username nome_comando`.** • Se non specificato, si usa l'utente root. • Viene chiesta la password dell'utente corrente. • L'utente "destinatario" deve fare parte del gruppo sudoers.

PERMESSI DI ACCESSO AL FILESYSTEM

Il meccanismo dei permessi gestisce l'accesso ai filesystem da parte dei vari utenti del sistema.

Per ogni file (e directory) sono definiti:

1. Un **utente proprietario** (owner).
2. Un **gruppo proprietario** (group owner).

Di conseguenza, per ogni file ci sono **tre classi di utenti**:

1. Il **proprietario del file** (owner).
2. Gli **utenti appartenenti al gruppo proprietario.**
3. Gli **altri utenti** (others).

A ciascuna classe di utenti (proprietario, appartenenti al gruppo, altri) vengono applicati permessi specifici. Possono essere di accesso in:

r	Read (lettura).
w	Write (scrittura).
x	Execute (esecuzione).

Quando un utente prova ad utilizzare un file, vengono applicati i permessi:

- Relativi all'owner, se l'utente è proprietario del file.
- Relativi al group owner, se l'utente non è proprietario del file, ma appartiene al gruppo proprietario.
- Validi per tutti gli altri utenti (others), se l'utente non è proprietario e non appartiene al gruppo proprietario.

FILE

r	Permette di leggere il contenuto del file.
w	Permette di modificare il contenuto del file.
x	Permette di eseguire un file (binario o script).

Il permesso di scrittura non permette di cancellare un file: per la cancellazione di file valgono i permessi della directory.

DIRECTORY

r	Permette di leggere il contenuto (elenco dei file).
w	Permette di modificare il contenuto.
x	Permette di attraversare una cartella.

- Negare l'accesso in lettura impedisce l'esecuzione del comando ls.
- Negare l'accesso in scrittura impedisce di creare, rinominare, cancellare file.

- Negare l'accesso in esecuzione impedisce di utilizzare il comando cd sulla directory.

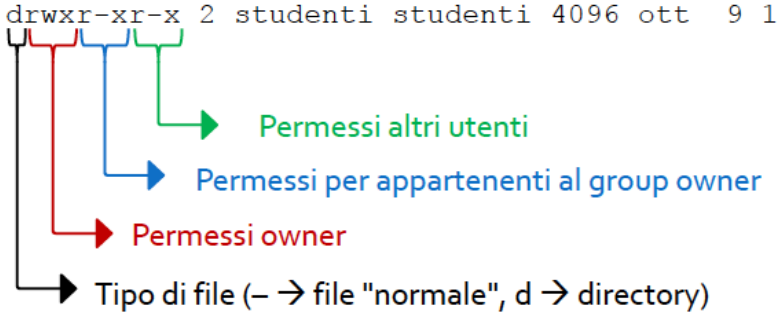
RAPPRESENTAZIONE SIMBOLICA

I permessi di un file (o directory) possono essere visualizzati con il comando

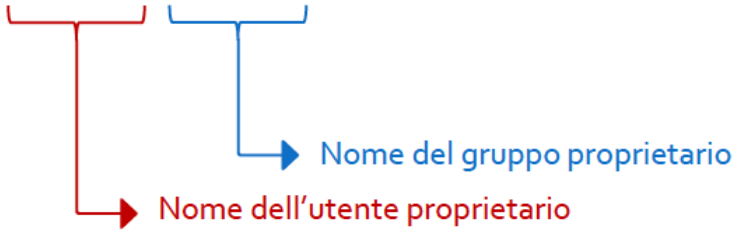
```
ls -l
```

ESEMPI DI OUTPUT:

```
-rw-r--r-- 1 studenti studenti   10 ott  9 13:17 esempio.txt
-rw-r--r-- 1 root     root        0 ott  9 13:28 root_file.txt
drwxr-xr-x 2 studenti studenti 4096 ott  9 13:19 subDir
```




```
-rw-r--r-- 1 studenti studenti   10 ott  9 13:17 esempio.txt
-rw-r--r-- 1 root     root        0 ott  9 13:28 root_file.txt
drwxr-xr-x 2 studenti studenti 4096 ott  9 13:19 subDir
```



RAPPRESENTAZIONE OTTALE

Tre cifre in base 8: rappresentano i permessi dell'owner, del group owner e degli altri utenti.

Ciascuna di queste cifre è ottenuta sommando:

- 4 se è permessa la **lettura**.
- 2 se è permessa la **scrittura**.
- 1 se è permessa l'**esecuzione**.

ESEMPI	
777	Sono garantiti tutti i permessi (4+2+1) a tutti gli utenti.
750	Il proprietario ha tutti i permessi (4+2+1), il group owner ha permesso in lettura ed esecuzione (4+1), gli altri utenti non hanno nessun permesso.

COMANDO chmod

- Permette di modificare i permessi relativi ad uno o più file.
- È possibile usare la rappresentazione simbolica o quella ottale.

- L'opzione `-R` permette di modificare in modo ricorsivo i permessi di una directory e dei file/directory in essa contenuti.
- Può essere eseguito dal proprietario o dall'utente root.

ESEMPIO CON RAPPRESENTAZIONE OTTALE: `chmod 755 file`.

SINTASSI CON RAPPRESENTAZIONE SIMBOLICA	
`chmod [who][how][which] fileName`	
who	Indica la classe di utenti per cui devono essere modificati i permessi: • `u` – owner. • `g` – group owner. • `o` – tutti gli altri.
how	Indica in che modo devono essere modificati i permessi: • `+` : aggiungere permessi. • `-` : togliere permessi. • `=` : assegnare permessi.

ESEMPIO: `chmod go-rwx file` toglie tutti i permessi di accesso a "file" a group owner e altri utenti.

PERMESSI AGGIUNTIVI – SUID, SGID

SUID	Durante l'esecuzione il processo acquisisce i privilegi del proprietario del file (normalmente un processo acquisisce i privilegi di chi lo esegue).
SGID	Durante l'esecuzione il processo acquisisce i privilegi del gruppo proprietario del file (normalmente un processo ha i privilegi del gruppo di chi lo esegue).

Rappresentazione simbolica di SUID: si utilizza il campo relativo ai permessi in esecuzione del file owner e la lettera s (invece di x) per indicare il permesso in esecuzione con SUID.

Rappresentazione simbolica di SGID: si utilizza il campo relativo ai permessi in esecuzione del group owner, allo stesso modo del SUID.

ESEMPIO: eseguendo

```
ls -l /usr/bin/passwd
```

si ottiene come rappresentazione simbolica

```
-rwsr-xr-x
```

RAPPRESENTAZIONE OTTALE: si utilizza un'ulteriore cifra prima delle 3 relative alle classi di utenti. Questa cifra ottale è ottenuta come somma di

- 4 se è attivo il permesso SUID.
- 2 se è attivo il permesso SGID.

ESEMPIO: `6754` corrisponde a `rwsr-sr--`.

COMANDI chown E chgrp

`chown username file`	Permette di impostare `username` come nuovo proprietario di file. • Può essere eseguito solo dall'utente root.
`chgrp groupname file`	Permette di impostare `groupname` come gruppo proprietario di file. • Un utente normale può eseguire il comando solo se appartiene a `groupname`, altrimenti è necessario essere root.

EDITOR DI TESTO DA TERMINALE: VI

`vi nome_file`	Per modificare un file esistente o creare un nuovo file.

Esistono due modalità di funzionamento:

1. **Modalità comandi:** permette di inserire comandi e scegliere quale azione compiere.
2. **Modalità editing:** permette di inserire e cancellare testo.

`Esc`	Passa in modalità comandi.
`i`	Passa in modalità inserimento nella posizione corrente.
`o`	Inserisce una nuova linea dopo quella corrente.
`x`	Cancella il carattere corrente.
`u`	Annulla l'ultimo comando sulla linea corrente.
`r?`	Sostituisce con ? il carattere su cui si trova il cursore.
`dd`	Cancella la riga corrente.
`ndd`	Cancella n righe.
`yy`	Copia una riga.
`nyy`	Copia n righe.
`p`	Incolla la selezione della riga sotto il cursore.
`/word`	Ricerca nel testo la parola `word`.
`n`	Si posiziona sull'occorrenza successiva (nella ricerca).
`N`	Si posiziona sull'occorrenza precedente (nella ricerca).
`:q`	Esce (solo se non si sono fatte modifiche).
`:w`	Salva.
`:wq`	Salva ed esce.
`:q!`	Esce senza salvare.
`:help`	Richiama l'aiuto in-line.

ESERCIZI

ESERCIZIO 1

- Lavorare nella propria cartella home.
- Creare una cartella con nome *visibile* e al suo interno una cartella con nome *segreta.*

```
studenti@studenti:~$ mkdir -p visibile/segreta
```

- Scrivere la stringa *vero* nel file *notizia.txt* all'interno di *visibile*.

```
studenti@studenti:~$ echo 'vero' > visibile/notizia.txt
```

- Copiare *notizia.txt* all'interno di segreta assegnandole il nome *cronaca.txt.*

```
studenti@studenti:~$ cp visibile/notizia.txt visibile/segreta/cronaca.txt
```

- Lavorare sui permessi di visibile
  - Togliere il permesso di esecuzione (proprietario) a *visibile* usando la rappresentazione simbolica.
  - Ripristinare il diritto di esecuzione (proprietario) a *visibile* usando la rappresentazione simbolica.
  - Togliere di nuovo il diritto di esecuzione (proprietario) usando la rappresentazione ottale e lasciando invariati gli altri permessi.

```
studenti@studenti:~$ chmod u-x visibile/
studenti@studenti:~$ chmod u+x visibile/
studenti@studenti:~$ ls -l
drwxr-xr-x 3 studenti studenti 4096 ott 11 15:58 visibile
studenti@studenti:~$ chmod 655 visibile/
studenti@studenti:~$ ls -l
drw-r-xr-x 3 studenti studenti 4096 ott 11 15:58 visibile
```

- A questo punto:
  - Si riesce a vedere il contenuto di visibile?
  - Si riesce a vedere il file notizia.txt dentro visibile?
  - Si riesce a vedere il contenuto di segreta?
  - Si riesce a vedere il file cronaca.txt dentro segreta?

```
studenti@studenti:~$ ls visibile/
ls: impossibile accedere a visibile/segreta: Permesso negato
ls: impossibile accedere a visibile/notizia.txt: Permesso negato
notizia.txt  segreta
studenti@studenti:~$ cat visibile/notizia.txt
cat: visibile/notizia.txt: Permesso negato
studenti@studenti:~$ ls visibile/segreta
ls: impossibile accedere a visibile/segreta: Permesso negato
studenti@studenti:~$ cat visibile/segreta/cronaca.txt
cat: visibile/segreta/cronaca.txt: Permesso negato
```

- Ripristinare il permesso di esecuzione a visibile e togliere il permesso in lettura a segreta (per l'utente proprietario).
    - Riesco a vedere il contenuto di segreta?
    - Riesco a leggere il contenuto di cronaca.txt dentro segreta?

```
studenti@studenti:~$ chmod u+x visibile/
studenti@studenti:~$ chmod u-r visibile/segreta/
studenti@studenti:~$ ls visibile/segreta/
ls: impossibile aprire la directory visibile/segreta/: Permesso negato
studenti@studenti:~$ cat visibile/segreta/cronaca.txt
vero
```

ESERCIZIO 2

- Creare un utente utente2.
    - Si riesce a vedere il contenuto della home di utente2 con le proprie credenziali utente?
    - Eventualmente cambiare i diritti in modo che gli altri utenti non riescano a vedere il contenuto della home di utente2.

```
studenti@studenti:~$ sudo adduser utente2
Aggiunta dell'utente ´utente2ª ...
Aggiunta del nuovo gruppo ´utente2ª (1001) ...
Aggiunta del nuovo utente ´utente2ª (1001) con gruppo ´utente2ª ...
Creazione della directory home ´/home/utente2ª ...
Copia dei file da ´/etc/skelª ...
Immettere nuova password UNIX:
Reimmettere la nuova password UNIX:
passwd: password aggiornata correttamente
Modifica delle informazioni relative all'utente utente2
Inserire il nuovo valore o premere INVIO per quello predefinito
     Nome completo []:
     Stanza n∞ []:
     Numero telefonico di lavoro []:
     Numero telefonico di casa []:
     Altro []:
Le informazioni sono corrette? [S/n] S

studenti@studenti:~$ ls -l /home
totale 8
drwxr-xr-x 25 studenti studenti 4096 ott 11 15:58 studenti
drwxr-xr-x  2 utente2  utente2  4096 ott 11 16:02 utente2

studenti@studenti:~$ sudo chmod o-r -R /home/utente2/
studenti@studenti:~$ ls -l /home
totale 8
drwxr-xr-x 25 studenti studenti 4096 ott 11 15:58 studenti
drwxr-x--x  2 utente2  utente2  4096 ott 11 16:02 utente2
```

- Controllare a quali gruppi appartiene l'utente root.

```
studenti@studenti:~$ groups root
root : root
```

- Creare un utente utente3.

```
studenti@studenti:~$ sudo adduser utente3
Aggiunta dell'utente ´utente3ª ...
Aggiunta del nuovo gruppo ´utente3ª (1002) ...
Aggiunta del nuovo utente ´utente3ª (1002) con gruppo ´utente3ª ...
Creazione della directory home ´/home/utente3ª ...
Copia dei file da ´/etc/skelª ...
Immettere nuova password UNIX:
Reimmettere la nuova password UNIX:
passwd: password aggiornata correttamente
Modifica delle informazioni relative all'utente utente3
Inserire il nuovo valore o premere INVIO per quello predefinito
     Nome completo []:
     Stanza n∞ []:
     Numero telefonico di lavoro []:
     Numero telefonico di casa []:
     Altro []:
Le informazioni sono corrette? [S/n] S
```

- Creare la cartella temp nella home di utente3.

```
studenti@studenti:~$ sudo mkdir /home/utente3/temp
```

- Quali sono l'utente proprietario e il gruppo proprietario di temp?

```
studenti@studenti:~$ ls -l /home/utente3/
totale 4
drwxr-xr-x 2 root root 4096 ott 11 16:05 temp
```

- Cambiare utente proprietario e gruppo proprietario di temp con utente3 e verificare che sia avvenuto l'aggiornamento di tali campi.

```
studenti@studenti:~$ sudo chown utente3 /home/utente3/temp/
studenti@studenti:~$ sudo chgrp utente3 /home/utente3/temp/
```

- Rimuovere utente2 ed utente3.

```
studenti@studenti:~$ sudo deluser utente2
Rimozione dell'utente ´utente2ª ...
Attenzione: il gruppo ´utente2ª non ha alcun membro.
Fatto.
```

```
studenti@studenti:~$ sudo deluser utente3
Rimozione dell'utente ´utente3ª ...
Attenzione: il gruppo ´utente3ª non ha alcun membro.
Fatto.
```

ESERCITAZIONE 3: UTENTI E GRUPPI (SECONDA PARTE)

IDENTIFICATORI E PERMESSI

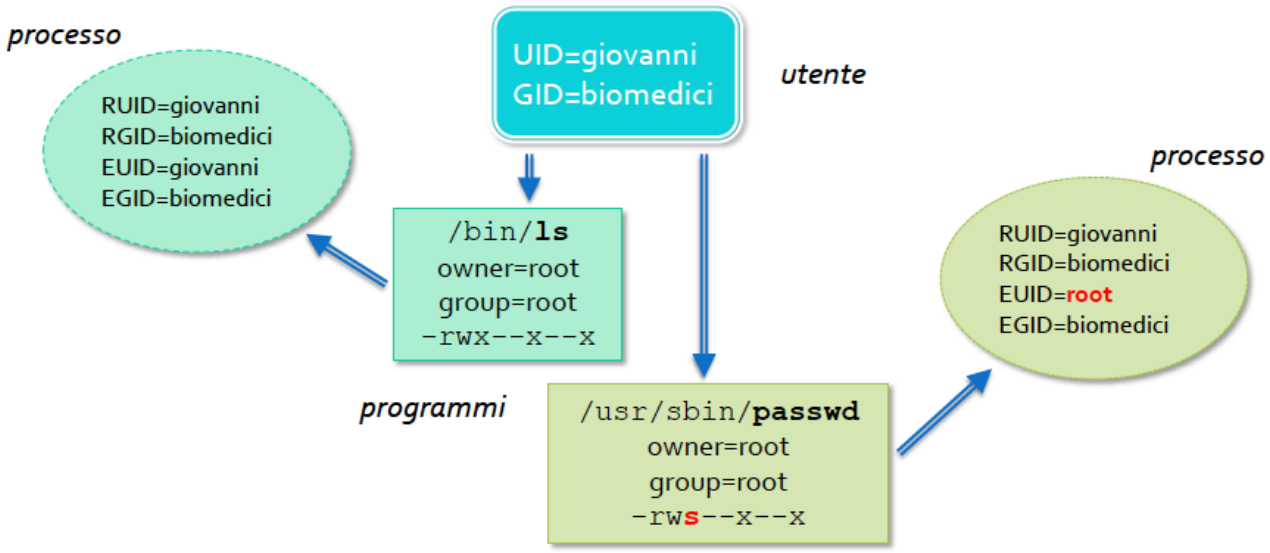
Gli identificatori che determinano i permessi del processo di dividono in:

REAL		EFFECTIVE	
`RUID`	**Real User ID**: ID dell'utente che ha mandato in esecuzione il processo.	`EUID`	**Effective User ID.**
`RGID`	**Real Group ID**: ID del gruppo primario dell'utente che ha mandato in esecuzione il processo.	`EGID`	**Effective Group ID.**

EUID/EGID possono differire da RUID/RGID se il comando eseguito ha il bit SUID o SGID attivo.

- Vengono utilizzati per definire i privilegi di accesso alle risorse e di invocazione di system call del processo.

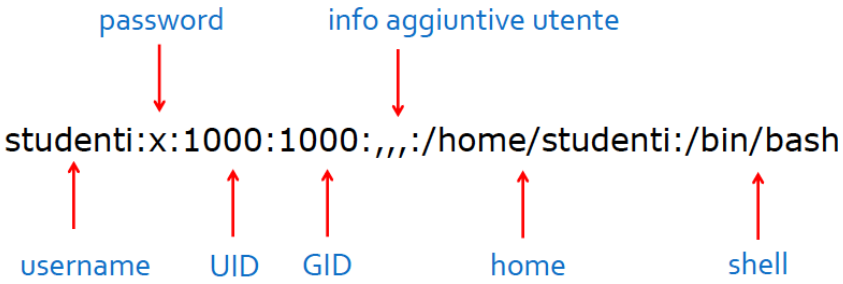
Un processo utente (non root) può inviare segnali ad un altro processo solo se il suo EUID o RUID coincide con il RUID del processo destinatario.



FILE DI CONFIGURAZIONE UTENTI

`/etc/passwd`	File con informazioni pubbliche sugli utenti.
`/etc/shadow`	File con informazioni sensibili.

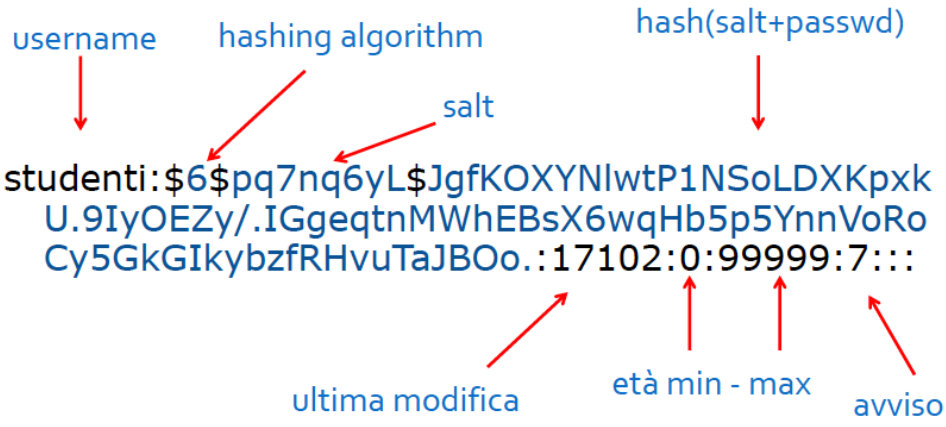
FILE /etc/passwd	
`man 5 passwd`	Informazioni del manuale.
`vipw`	Aprire il file in editing.



username	Nome utente utilizzato per il login.
password	Campo (un tempo) dedicato alla password (cifrata). 'x' indica che la password cifrata si trova in `/etc/shadow`.
UID (user id)	Identificatore numerico univoco dell'utente.
GID (group id)	Identificativo numerico univoco del gruppo primario.
Dati personali	Nome completo dell'utente e altre informazioni.
Cartella home	Percorso assoluto della cartella personale (home). Viene utilizzato per impostare la variabile d'ambiente `$HOME`.
Shell	Interprete dei comandi da utilizzare per l'utente.

La shell può essere impostata a `/sbin/nologin` o `/bin/false` per indicare che non è possibile fare login con tali utenti.

FILE /etc/shadow	
man shadow	Informazioni del manuale.
vipw -s	Aprire il file in editing.



username	Deve essere un nome valido (esistente).
password	La password cifrata dell'utente (vedere **crypt**).
Ultima modifica	Data di modifica della password (giorni dal 1970).
Età min	Durata minima della password.

`Età max`	Durata massima della password.
`Periodo di avviso`	Giorni prima della scadenza in cui l'utente viene avvisato.
`Periodo inattività`	Giorni dopo la scadenza della password in cui questa è ancora accettata.
`Scadenza`	Data scadenza account.
`Campo riservato`	Riservato per utilizzo futuro.
`salt`	Stringa di 7 caratteri.

COMANDI PER LA GESTIONE DEI GRUPPI

CREAZIONE E RIMOZIONE GRUPPI	
`addgroup gruppo`	Creazione di un gruppo.
`delgroup gruppo`	Rimozione di un gruppo.

- La creazione/rimozione di un gruppo richiede privilegi di root.

gpasswd	
`gpasswd -a utente gruppo`	Aggiungere un utente a un gruppo.
`gpasswd -d utente gruppo`	Rimuovere un utente da un gruppo.
`gpasswd -M utente1, utente2,… gruppo`	Definire i membri di un gruppo.
`gpasswd -A utente1, utente2,… gruppo`	Definire gli amministratori di un gruppo.

- Solo gli amministratori di un gruppo (oltre a root) possono aggiungere/rimuovere utenti a/da un gruppo.
- Solo root può aggiungere/rimuovere gli amministratori.

`gpasswd gruppo`	Impostare/cambiare la password di un gruppo.
`gpasswd -r gruppo`	Rimuovere la password di un gruppo.

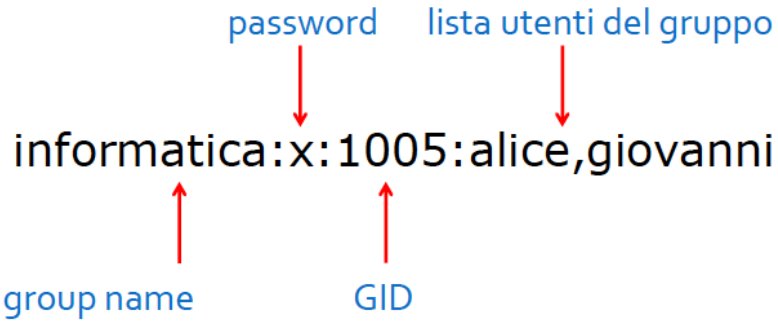
- Se la password non è impostata, solo i membri del gruppo possono averne i privilegi.
- Se la password è impostata:
  - Gli altri utenti (non membri del gruppo) possono acquisire temporaneamente i privilegi del gruppo mediante il comando `newgrp`.
  - I membri del gruppo non hanno bisogno di usare la password.
- Le password di gruppo sono intrinsecamente poco sicure, in quanto conosciute da più utenti.

`newgrp gruppo`	Il gruppo specificato diventa il nuovo gruppo primario dell'utente per la sessione di login corrente.

FILE DI CONFIGURAZIONE GRUPPI

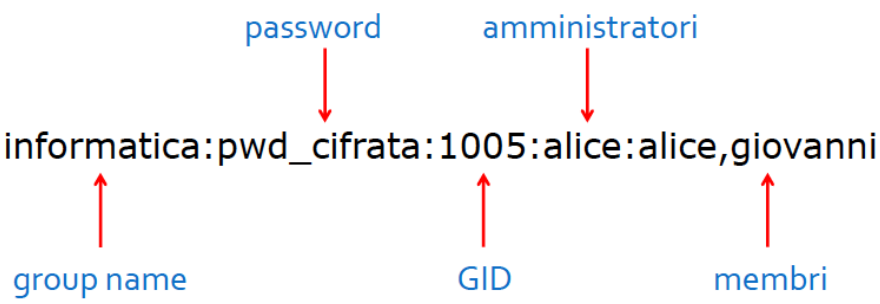
`/etc/group`	File con informazioni pubbliche sui gruppi.
`/etc/gshadow`	File con informazioni sensibili (password e amministratori).

FILE /etc/group	
`man group`	Manuale.
`vigr`	Apriamo il file.



`gruppo`	Nome del gruppo.
`password`	Password cifrata del gruppo. 'x' indica che la password cifrata si trova in `/etc/gshadow`.
`GID (group id)`	Identificatore univoco del gruppo (numero).
`Lista utenti`	Utenti appartenenti al gruppo (separati da virgole). Non contiene l'utente per il quale il gruppo è il "primary group", in quanto questa informazione è già in `/etc/passwd`.

FILE /etc/gshadow	
`man gshadow`	Manuale.
`Vigr -s`	Apriamo il file.



gruppo	Nome del gruppo.
password	Password cifrata del gruppo. Un campo vuoto oppure i caratteri '*' o '!' indicano che la password non è impostata. In tal caso solo i membri del gruppo possono avere i privilegi del gruppo.
GID (group id)	Identificatore univoco del gruppo (numero).
Amministratori	Lista degli utenti amministratori del gruppo (separati da virgole). Gli amministratori possono cambiare la password e aggiungere/rimuovere utenti al/dal gruppo.
Lista utenti	Altri utenti del gruppo (separati da virgole).

ESERCIZI

PREPARAZIONE ESERCIZIO

Eliminare eventuali utenti creati per gli esercizi precedenti:

- Visualizzare solo gli utenti che hanno la home in `/home` con il comando `cat /etc/passwd | grephome`.
- Eliminare gli utenti mostrati, lasciando solo studenti (non eliminare gli utenti di sistema!).
  - Cercare sul manuale l'opzione di `deluser` per rimuovere la home dell'utente.
  - Se necessario rimuovere manualmente la home di utenti già rimossi.

ESERCIZIO

- Aggiungere tre nuovi utenti con username alice, giovanni, simone.
  - Verificare le nuove informazioni contenute nei file di configurazione (`/etc/passwd` e `/etc/shadow`).

```
studenti@studenti:~$ sudo adduser alice
studenti@studenti:~$ sudo adduser giovanni
studenti@studenti:~$ sudo adduser simone

studenti@studenti:~$ cat /etc/passwd | less
studenti@studenti:~$ sudo cat /etc/shadow | less
```

- Fare login come alice.

```
studenti@studenti:~$ su alice
```

- Spostarsi nella home di alice.

```
alice@studenti:/home/studenti$ cd
```

- Creare un file documento dentro una cartella docs.

```
alice@studenti:~$ mkdir docs
alice@studenti:~$ touch docs/documento
```

- Scrivere la stringa «messaggio importante» dentro documento.

```
alice@studenti:~$ echo 'messaggio importante'> docs/documento
```

- Fare login come giovanni o simone.
    - giovanni e simone possono leggere documento?

```
alice@studenti:~$ su giovanni
giovanni@studenti:/home/alice$ cat docs/documento
messaggio importante
```

- Modificare i permessi di documento in modo che non possa essere letto dagli altri.
    - Verificare che gli altri non possono leggere.

```
giovanni@studenti:/home/alice$ exit
alice@studenti:~$ chmod o-r docs/documento
alice@studenti:~$ su giovanni
Password:
giovanni@studenti:/home/alice$ cat docs/documento
cat: docs/documento: Permesso negato
```

- Fare login come root.
    - Creare un gruppo informatica e aggiungere alice ai membri del gruppo.
        - Usare il comando `exit`(se necessario più volte) per tornare alla shell di alice.
    - Se si utilizza il comando groups, si vede il gruppo informatica nei gruppi di alice?
        - Provare a fare logout e login nuovamente (utilizzando i comandi `exit` e `su`), e poi lanciare di nuovo `groups`.

```
giovanni@studenti:/home/alice$ su root
Password:
root@studenti:/home/alice# addgroup informatica
Aggiunta del gruppo ´informaticaª (GID 1003) ...
Fatto.
root@studenti:/home/alice# gpasswd -a alice informatica
Aggiunta dell'utente alice al gruppo informatica
root@studenti:/home/alice# exit
giovanni@studenti:/home/alice$ exit

alice@studenti:~$ groups
alice
alice@studenti:~$ exit
studenti@studenti:~$ su alice
alice@studenti:/home/studenti$ groups
alice informatica
```

- Dal terminale di alice
    - Cambiare il group owner di documento, in modo che sia il nuovo gruppo informatica.
    - Verificare con `ls-l` che il nuovo group owner è informatica e che gli appartenenti al gruppo hanno accesso in lettura (mentre gli altri utenti non possono accedere).
    - Provare ad aggiungere giovanni al gruppo «informatica», è possibile per alice?
        - In caso negativo fare in modo che alice possa amministrare il gruppo e aggiungere giovanni.

```
alice@studenti:/home/studenti$ cd
alice@studenti:~$ ls -l docs/documento
-rw-r----- 1 alice alice 21 ott 17 18:18 docs/documento
alice@studenti:~$ chgrp informatica docs/documento
alice@studenti:~$ ls -l docs/documento
-rw-r----- 1 alice informatica 21 ott 17 18:18 docs/documento

alice@studenti:~$ gpasswd -a giovanni informatica
gpasswd: Permesso negato.
alice@studenti:~$ su root
root@studenti:/home/alice# gpasswd -A alice informatica
root@studenti:/home/alice# exit
alice@studenti:~$ gpasswd -a giovanni informatica
Aggiunta dell'utente giovanni al gruppo informatica
```

- Accedere al terminale di giovanni e verificare la possibilità di accedere al file.

```
alice@studenti:~$ su giovanni
giovanni@studenti:/home/alice$ cat docs/documento
messaggio importante
giovanni@studenti:/home/alice$ exit
```

- Controllare se simone può leggerlo.

```
alice@studenti:~$ su simone
simone@studenti:/home/alice$ cat docs/documento
cat: docs/documento: Permesso negato
simone@studenti:/home/alice$
```

- Dal terminale di alice
    - Impostare una password per il gruppo informatica.
    - giovanni e alice hanno bisogno della password per accedere a documento?

```
simone@studenti:/home/alice$ exit
exit
alice@studenti:~$ gpasswd informatica
Cambio della password del gruppo informatica
alice@studenti:~$ cat docs/documento
```

```
messaggio importante
alice@studenti:~$ su giovanni
Password:
giovanni@studenti:/home/alice$ cat docs/documento
messaggio importante
giovanni@studenti:/home/alice$ exit
```

- Accedere al terminale di simone
  - Leggere il contenuto di documento sfruttando la password del gruppo.
  - Spostarsi nella home di Simone e creare un file prova.
  - Qual è il group owner di prova?

```
alice@studenti:~$ su simone
simone@studenti:/home/alice$ cat docs/documento
cat: docs/documento: Permesso negato
simone@studenti:/home/alice$ newgrp informatica
simone@studenti:/home/alice$ cat docs/documento
messaggio importante
simone@studenti:/home/alice$ cd
simone@studenti:~$ touch prova
simone@studenti:~$ ls -l prova
-rw-r--r-- 1 simone informatica 0 ott 17 18:39 prova
```

- Accedere al terminale di root ed eliminare il gruppo informatica.

```
simone@studenti:~$ su root
root@studenti:/home/simone# delgroup informatica
Rimozione del gruppo ´informaticaª ...
Fatto.
```

- Tornare al terminale di alice e spostarsi nella home.
  - Visualizzare le informazioni di documento con `ls-l`, cosa viene visualizzato al posto del group owner?
  - Impostare alice come group ownerdel file.

```
root@studenti:/home/simone# exit
simone@studenti:~$ exit
simone@studenti:/home/alice$ exit

alice@studenti:~$ ls -l docs/
totale 4
-rw-r----- 1 alice 1003 21 ott 17 18:18 documento
alice@studenti:~$ chgrp alice docs/documento
alice@studenti:~$ ls -l
totale 4
drwxr-xr-x 2 alice alice 4096 ott 17 18:18 docs
```

ESERCITAZIONE 4: STRUMENTI PER LA GESTIONE DEI FILE

RICERCA DI FILE

FIND

Strumento molto potente per trovare file:

- La sintassi è relativamente complessa.
- Permette di effettuare la ricerca combinando dei test sulle proprietà dei file:
  - Filename.
  - File type.
  - Owner (user e/o group).
  - Permessi.
  - Timestamp.
- La ricerca non è influenzata dal contenuto del file.
- È possibile eseguire comandi (actions) sui file trovati.

```
find [path1 path2…][espressione]
```

`path`	È possibile specificare uno o più percorsi (path) separati da spazio. La ricerca verrà effettuata solo nei percorsi specificati.
`espressione`	Descrive come vengono trovati i file, e quali azioni devono essere eseguite su di essi.

ESPRESSIONI

Le espressioni sono composte da una sequenza di elementi:

1. **Test:** valutazione di una proprietà dei file, può ritornare true o false.
2. **Azioni:** azioni da effettuare sui file "trovati" (ad esempio eseguire un comando). Ritornano true se hanno successo.
3. **Opzioni globali:** influenzano l'esecuzione di test o azioni. Ritornano sempre true.
4. **Opzioni posizionali (positional options):** influenzano solo le azioni o i test che seguono. Ritornano sempre true.

Gli elementi di una espressione sono collegati da **operatori**:

- -o indica OR, -a indica AND.
- Se nessun operatore è specificato, l'utilizzo dell'operatore AND è implicito per collegare due espressioni.
- ! può essere usato per negare un'espressione (NOT).

TEST	
-name pattern	Ricerca basata sul nome del file (non sul path). • Pattern può includere i metacaratteri *, ? oppure le parentesi `[]`. o Esempio: 'm[ao]re' porta a trovare 'mare' e 'more'. • È necessario scrivere i pattern tra apici per evitare che la shell "espanda" i metacatteri: o `echo '*'` o `echo *`
-type [dfl]	Tipo di file: • d – directory. • f – regular file. • l – symbolic link.
-size [+-]n[ckMG]	Ricerca basata sulla dimensione del file. • Il prefisso `[+-]` indica se il file deve essere maggiore o minore della dimensione specificata. • `n` indica la quantità di spazio occupata dal file. • `[ckMG]` indica l'unità di misura utilizzata, rispettivamente byte, kilobyte, megabyte, gigabyte.
-user user	Permette di capire se il file appartiene all'user. • L'utente può essere specificato come username o UID.
-group group	Permette di capire se `group` è il group owner. • Il gruppo può essere specificato come group name o GID.
-perm [-/] mode	Test basato sui permessi dei file (modalità ottale o simbolica). • `mode` – i permessi devono essere esattamente quelli specificati. • `-mode` – almeno i permessi indicati devono essere presenti. • `/mode` – almeno uno dei permessi indicati devono essere presenti.

AZIONI	
-delete	I file trovati vengono eliminati. • Ritorna true in caso di successo (i file sono stati eliminati senza errori). • **Attenzione:** se scriviamo -delete **prima** dei test, verranno eliminati **tutti i file**.
-exec command ;	Esegue il comando specificato sul file considerato (se ha superato i test precedenti). • Tutti gli argomenti specificati dopo command vengono considerati come argomenti del comando, fino al carattere `';'`. • La stringa `'{}'` è utilizzata per indicare il nome del file attualmente processato. • Il comando viene eseguito a partire dal percorso di partenza: usare `-execdir` per eseguire il comando a partire dal path del file trovato.

ESEMPI

```
find path -name 'prova*' ! -type d
```

Ricerca i file il cui nome inizia con 'prova' e che NON sono directory.

- È necessario utilizzare gli apici oppure il carattere di escape `'\'` prima di `*` per "proteggerlo" dalla shell.

```
find path ! -name '*.csv' -size +50M -execdir ls -l {} \;
```

Cerca i file con:

1. Estensione diversa da ".cvs".
2. Dimensione superiore a 50 Megabyte.

Ai file trovati viene applicato il comando ls -l: è necessario usare il carattere di escape `'\'` per "proteggere" `\;` dalla shell.

```
find path -perm -664
```

Ricerca i file per cui valgono almeno questi permessi:

1. Lettura e scrittura per owner e group owner.
2. Lettura per gli altri.

Vengono trovati anche file con permessi in più rispetto a questi (ad esempio se anche gli altri utenti hanno permesso di scrittura).

```
find path -perm /u=w,g=w
```

Cerca i file che possono essere scritti da almeno uno fra owner e group owner.

LOCATE

```
locate [options] file1…
```

Ricerca il/i file specificato/i.

- Sfrutta un database periodicamente aggiornato dal sistema: l'aggiornamento del database può essere forzato con il comando `updatedb` (richiede privilegi di root).

FIND VS LOCATE

find	locate
• Comando standard presente in tutti i sistemi Unix/Linux. • Dà sempre risultati aggiornati (non dipende dall'aggiornamento di un database). • Permette di definire test e azioni.	• Più semplice da utilizzare e più veloce.

RICERCA DI TESTO NEI FILE

GREP

General regular expression print: permette di cercare in uno o più file di testo le linee (righe) che corrispondono ad espressioni regolari o stringhe letterali.

```
grep [opzioni] [-e] modello [-e modello2…] file1 [file2…]
```

! Se si vuole specificare più di un modello (stringa o espressione regolare) si deve utilizzare `-e` prima di ciascun modello (incluso il primo).

OPZIONI	
`-i`	Ignora le distinzioni tra maiuscole e minuscole.
`-v`	Mostra le linee che non contengono l'espressione.
`-n`	Mostra il numero di linea.
`-c`	Riporta solo il conteggio delle linee trovate.
`-w`	Trova solo parole intere.
`-x`	Linee intere.

ESPRESSIONI REGOLARI

È possibile specificare dove la stringa/espressione deve trovarsi all'interno di una riga.

`'^'`	L'espressione deve trovarsi ad inizio riga.
`'$'`	L'espressione deve trovarsi in fondo alla riga.

ESEMPI	
`'^stringa'`	Righe che iniziano con `'stringa'`.
`'stringa$'`	Righe che terminano con `'stringa'`.
`'^stringa$'`	Righe che contengono solo `'stringa'`.
`'^$'`	Righe vuote.

Le parentesi quadre permettono di definire set di caratteri ammessi.

ESEMPIO: `grep '1[23]:[0-5][0-9]' file`

1. Il primo carattere deve essere '1'.
2. Il secondo può essere '2' o '3'.
3. Il terzo deve essere ':'.
4. Il quarto deve essere una cifra tra '0' e '5'.
5. Il quinto tra '0' e '9'.

`'.'`	Indica qualsiasi carattere. • `'…cept'` – riconosce sia 'accept' che 'except'.
`'*'`	Indica che l'espressione può essere ripetuta. • `'[A-za-z]*'` – riconosce zero o più caratteri alfabetici.
`'\'`	Carattere di escape. • `'^[0-9].*\.$'` – riconosce righe che iniziano con una cifra e terminano con un punto.

ARCHIVIAZIONE E COMPRESSIONE

Il comando tar (Tape ARchive) permette di archiviare/estrarre una raccolta di file e cartelle.

```
tar modalità[opzioni] [file1…]
```

- La modalità specifica il modo in cui il comando deve operare (ad esempio creare un archivio o estrarre un archivio già esistente.
- Le opzioni permettono di fornire ulteriori dettagli sul comportamento di tar (ad esempio specificare la tecnica di compressione ed il nome dell'archivio).
- La lista di file/cartelle indica quali file/cartelle devono essere archiviati o estratti (in base alla modalità).

Il formato del file creato dipende dalla compressione (eventualmente) utilizzata.

`.tar`	Se non si è utilizzata compressione.
`.tar.gz`	Se l'archivio è stato compresso con gz.
`.tar.bz2`	Se l'archivio è stato compresso con bzip2.

TAR

Subito dopo il comando tar, deve essere specificata la modalità in cui operare.

AZIONI	
`A`	Aggiungi file tar all'archivio.
`c`	Crea un nuovo archivio.
`d`	Trova le differenze fra l'archivio ed il file system.
`--delete`	Cancella il file dall'archivio.
`r`	Aggiungi il file all'archivio.
`t`	Elenca i file di un archivio.
`u`	Aggiungi file all'archivio, ma solo se differiscono dalla copia eventualmente già presente.
`x`	Estrai i file dall'archivio.

Le opzioni permettono di definire meglio il modo in cui il comando tar deve operare

OPZIONI	
v	Verbose.
z	Compressione con gzip.
j	Compressione con bzip2.
f	Permette di specificare il nome dell'archivio.

- Consultare `man tar` per la lista completa delle opzioni.

ESEMPI

tar cvf archivio.tar percorso
Crea un archivio di nome "archivio.tar" con il contenuto di "percorso". ! Modalità verbose.
tar czf archivio.tar.gz percorso
Crea un archivio compresso "archivio.tar.gz". ! Usa la compressione gz.
tar tf archivio.tar
Mostra il contenuto di "archivio.tar".
tar xvf archivio.tar file
Estrae "file" da "archivio.tar". ! Modalità verbose.

gzip/gunzip e bzip2/bunzip2

Se si devono comprimere file o archivi creati precedentemente con tar, è possibile utilizzare:

gzip file1 file2…	I file elencati vengono compressi e salvati in file con lo stesso nome ed estensione .gz. I file non compressi vengono elminati.
gunzip file1.gz file2.gz…	Estrae i file compressi specificati in file con lo stesso nome (senza l'estensione relativa alla compressione). I file compressi, dopo essere stati estratti, vengono eliminati.

- `bzip2` e `bunzip2` utilizzano la stessa sintassi, ma comprimono/decomprimono con l'algoritmo `bzip2`.

ESERCIZI

ESERCIZIO 1 – FIND

- Trovare i file nella cartella `/usr/bin` che hanno il bit SUID attivo (`u=s`).

```
studenti@studenti:~$ find /usr/bin/ -perm -u=s
```

Alternativa ottale:

```
find /usr/bin -perm -4000
```

- Trovare tutte le cartelle contenute in `/etc` che hanno nel nome la stringa `'sys'`.

```
studenti@studenti:~$ find /etc -name '*sys*' -type d
```

- Trovare tutti i file con estensione `'.txt'` in `/usr/share/docutils` con dimensione superiore a 10 Kilobyte.
    - Per ogni file trovato fare in modo che venga mostrato l'output di `ls-l` eseguito dal path in cui si trovano i file.

```
studenti@studenti:~$ find /usr/share/docutils/ -name '*.txt' -size
+10k -execdir ls -l {} \;
```

ESERCIZIO 2 – GREP, ESPRESSIONI REGOLARI

- Trovare in `/etc/passwd` le righe che contengono `'studenti'`.

```
studenti@studenti:~$ grep 'studenti' /etc/passwd
```

- Trovare in `/etc/group` la riga che descrive il gruppo `'studenti'` (fare in modo che non vengano mostrate le altre righe contenenti la parola 'studenti').

```
studenti@studenti:~$ grep '^studenti:' /etc/group
```

- Cercare nel file GPL-3 (per trovarlo utilizzare locate) le righe contenenti una lettera minuscola fra parentesi tonde.

```
studenti@studenti:~$ grep '([a-z])' /usr/share/common-licenses/GPL-3
```

- Trovare in `/etc/passwd` le righe relative a UID da 102 a 105.

```
studenti@studenti:~$ grep '^[a-z][-a-z0-9_]*:x:10[2-5]:' /etc/passwd
```

ESERCIZIO 3 – ARCHIVIAZIONE

- Nella propria directory home, creare una cartella es3.

```
studenti@studenti:~$ mkdir es3
```

- Dentro es3, creare un archivio `conf.tar.bz2`, contenente tutti i file in `/etc` con estensione `.conf`, ed utilizzando la compressione bzip2.

```
studenti@studenti:~$ cd es3/
studenti@studenti:~/es3$ tar cjf archive.tar.bz2 /etc/*.conf
```

- Mostrare i file contenuti nell'archivio.

```
studenti@studenti:~/es3$ tar tf archive.tar.bz2
```

- Estrarre tutti i file dall'archivio – dove vengono salvati?

```
studenti@studenti:~/es3$ tar xvf archive.tar.bz2
```

- Cercare nel manuale l'opzione per estrarre un archivio in una directory specifica.
- Creare una sottocartella output, ed estrarre il file `etc/resolv.conf` in output.

```
studenti@studenti:~/es3$ tar xvf archive.tar.bz2 --directory
output/ etc/resolv.conf
```

ESERCITAZIONE 5: PROCESSI IN UNIX/LINUX, SYSTEM CALL PER I PROCESSI

PROCESSI IN UNIX

CARATTERISTICHE DEI PROCESSI IN UNIX

Unix è una famiglia di sistemi operativi multiprogrammati basati su processi.

Il processo Unix mantiene spazi di indirizzamento separati per dati e codice:

- **Spazio di indirizzamento dei dati privato**: comunicazione tra processi basata su scambi di messaggi.
- **Spazio di indirizzamento del codice condivisibile**: più processi possono eseguire lo stesso codice.

Unix adotta una politica di assegnamento della CPU ai processi basata sulla divisione di tempo.

- I processi attraversano vari stati.

STATI DI UN PROCESSO

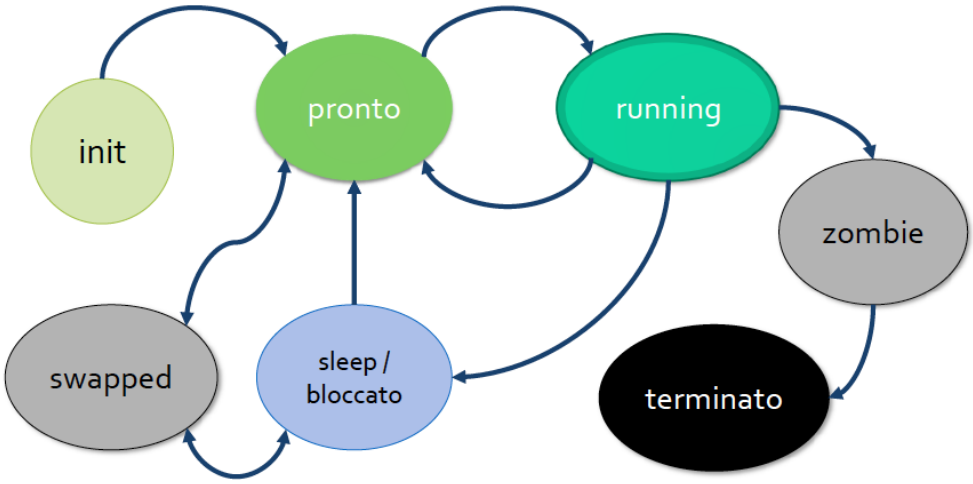
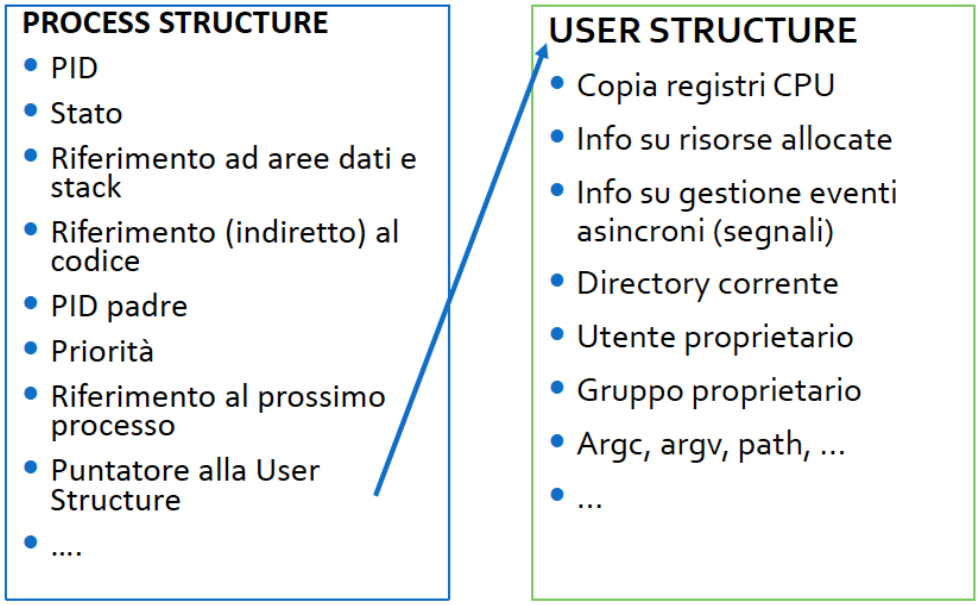


IMMAGINE DI UN PROCESSO UNIX

Il descrittore di un processo (PCB – process control block) è suddiviso in due strutture dati distinte:

1. **Process Structure:** informazioni indispensabili, sempre in memoria.
2. **User Structure:** informazioni utili solo quando il processo è residente in memoria (soggetta a swap-out).



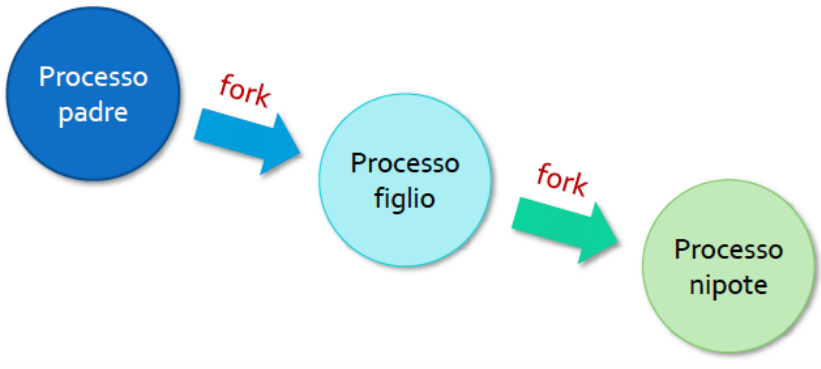
SYSTEM CALL PER I PROCESSI

`fork`	Creazione di processi.
`exit`	Terminazione.
`wait`	Sospensione in attesa della terminazione dei figli.
`exec..`	Sostituzione di codice (e di dati).

CREAZIONE DI PROCESSI – FORK

Ogni processo è in grado di creare dinamicamente processi.

- Lo strumento per la creazione è la chiamata di sistema **fork**.
- Il processo creato (**figlio**) ha uno spazio di dati separato, ma condivide con il **processo padre** il codice.
- Ogni processo figlio può a sua volta generare altri processi.



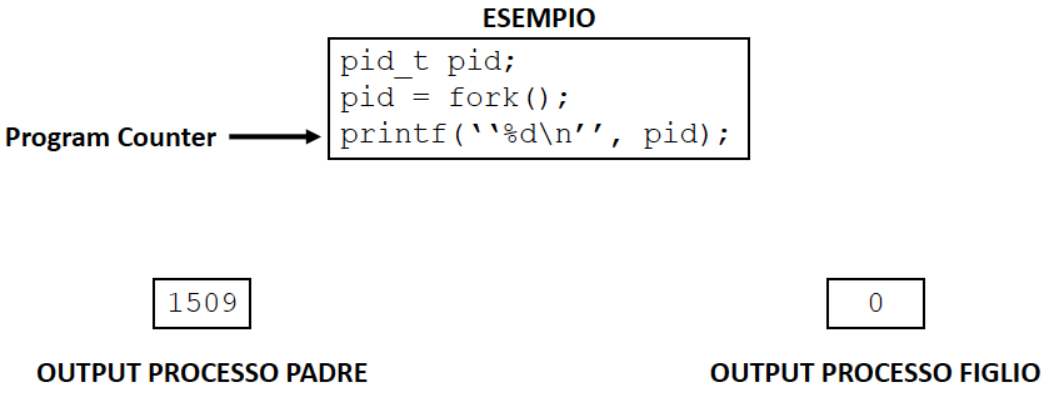
`pid_t fork(void)`	• Non richiede parametri. • Restituisce un risultato (intero) diverso a padre e figlio. ○ Al padre: PID figlio, valore negativo se fallisce. ○ Al figlio: zero.

Il processo figlio:

1. Condivide il codice con il padre.
2. Eredita una copia delle aree dati globali, stack, heap, User Structure.
    - Ogni variabile è inizializzata con il valore assegnatole dal padre.
    - Stesso valore di Program Counter del padre.

Dopo la fork, padre e figlio partono dalla stessa istruzione: quella che segue la fork.

- Il loro comportamento può essere differenziato sfruttando il valore di ritorno della fork.



PID E PPID

`pid_t getpid()`	Restituisce il PID del processo.
`pid_t getppid()`	Restituisce il PID del processo padre.

TERMINAZIONE PROCESSI

Involontaria:

- Azioni illegali (es. accesso a locazioni esterne al proprio spazio di indirizzamento).
- Interruzioni causate dalla ricezione di segnali.

Volontaria:

- Esecuzione dell'ultima istruzione.
- Chiamata della system call `exit()`.

`void exit(int status)`
• I processi che terminano volontariamente possono usare la system call **exit**. • È una chiamata senza ritorno (l'esecuzione termina). • Permette di comunicare al padre lo stato di terminazione.

```
pid_t wait(int *status)
```

- Il padre può ottenere lo stato di terminazione del figlio mediante la system call **wait**.
- `wait` ritorna il PID del figlio che è terminato.
- `status` è l'indirizzo della variabile dove verrà salvato lo stato di terminazione del figlio.

Effetto della wait sul processo padre:

1. Sospensione del padre se tutti i figli sono ancora in esecuzione.
2. Ritorno immediato con informazioni di terminazione se almeno un figlio è terminato (zombie).
3. Ritorno con valore negativo (errore) se non ci sono processi figli.

Codifica della variabile status:

- Contiene informazioni su come il figlio è terminato, oltre allo stato di terminazione eventualmente fornito dal figlio stesso.
- Se il byte meno significativo di `*status` è zero, allora la terminazione è stata volontaria: in questo caso il byte più significativo contiene lo stato di terminazione.

Macro per gestire status (in modo astratto rispetto alla reale implementazione) **definite in `<sys/wait.h>`:**

- **`WIFEXITED(status)`** – ritorna vero se terminata volontariamente.
- **`WEXITSTATUS(status)`** – ritorna lo stato di terminazione.

SOSTITUZIONE DI CODICE – exec..()

Un processo può sostituire il programma (codice e dati) che sta eseguendo utilizzando una syscall della "famiglia" `exec()`: `execl()`, `execle()`, `execclp()`, `execv()`, `execve()`, `execvp()`.

```
int execl(char* path, char* arg0, …, char* argN, (char*)0)
```

Lista di parametri di lunghezza variabile terminata dal puntatore nullo.

`path`	Percorso del comando.
`arg0`	Nome del programma da eseguire.
`arg1,…,argN`	Argomenti del comando.

Una chiamata `exec()` è senza ritorno se ha successo: solo in caso di fallimento vengono eseguite le parti di codice che seguono.

ESERCIZI

Includere le librerie necessarie:

```
#include <stdio.h>

#include <unistd.h>
```

```
#include <sys/types.h>

#include <stdlib.h>
```

ESERCIZIO 1 – FORK

Scrivere un programma C in cui:

- Viene creato un processo figlio.
- Il processo figlio stampa un messaggio del tipo
  "Sono X, figlio del processo Y"
  Dove al posto di X viene stampato l'id del processo figlio (PID) e al posto di Y viene stampato il PID del padre (PPID).
- Il padre stampa il messaggio "Sono il padre. Il PID di mio figlio è: X".
- Cosa succede se il padre termina prima del figlio o viceversa? Fare degli esperimenti utilizzando la funzione `sleep(interval)`.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) {
        // child process
        printf("Sono %d, figlio del processo %d\n", getpid(),getppid());
    }
    else if (pid > 0) {
        // parent process
        printf("Sono il padre, il PID di mio figlio e' %d\n", pid);
    }
    else
        printf("Creazione fallita!\n");
    exit(0);
}
```

ESERCIZIO 2 – WAIT

Scrivere un programma C in cui:

- Viene creato un processo figlio.
- Il processo figlio stampa un messaggio (es. "Sono il figlio Y") e poi termina fornendo come stato di terminazione il valore 1.
- Il padre attende con wait la terminazione del figlio, poi stampa a video se la terminazione è stata volontaria, ed eventualmente il valore di terminazione ottenuto.
  - Provare a fare in modo che il figlio termini in modo volontario e verificare che il padre è in grado di rilevarlo.
- Modificare il programma in modo che il padre generi N figli, e poi provveda ad attendere con la wait ciascuno dei figli generati.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NCHILDREN 3

int main()
{
    pid_t pid;
    int children = 0;

    // creation loop
    while (1) {
        children++;
        pid = fork();
        if (pid == 0) {
            // child process
            printf("Sono il figlio %d\n", getpid());
            // int a = 1/0; // uncomment this line to cause child's
            // abnormal termination.
            exit(1);
        }
        else if (pid > 0) {
            // parent process
            // generate up to NCHILDREN processes
            if (children < NCHILDREN)
                continue;

            // wait for all the children
            while (1) {
                int status;
                pid_t term_child = wait(&status); // returns pid of
                                                  // terminated child
                if (term_child < 0)
                    break;  // no more children left
                if (WIFEXITED(status)) {
                    printf("Terminazione di %d volontaria\n",
                           term_child);
                    printf("Stato di terminazione: %d\n",
                           WEXITSTATUS(status));
                }
                else
                    printf("Terminazione di %d involontaria\n",
                           term_child);
            }
            exit(0);
        }
        else {
            printf("Creazione fallita!\n");
            exit(1);
```

```
        }
    }
}
```

ESERCIZIO 3 – EXECL

Scrivere un programma C in cui:

- Viene creato un processo figlio.
  - Il figlio utilizza execl per sostituire il proprio codice con "ls -l argv[1]", dove argv[1] indica un argomento passato dalla linea di comando.
  - In caso di errore nell'esecuzione di execl, il figlio stampa un messaggio di errore e chiama la exit con stato di terminazione 1.
- Il padre attende la terminazione del figlio ed interpreta correttamente la sua terminazione, stampando a video un messaggio informativo.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char** argv)
{
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        // child
        printf("Sono il figlio %d.\n", getpid());
        char* argument = (argc > 1) ? argv[1] : NULL;
        execl("/bin/ls", "ls", "-l", argument, NULL);
        printf("Errore nella exec\n");
        exit(1);
    }
    else if (pid > 0) {
        // parent
        int status;
        pid_t term_child = wait(&status);
        printf("Terminato figlio %d\n", term_child);
        if (WIFEXITED(status)) {
            printf("Terminazione di %d volontaria\n", term_child);
            printf("Stato di terminazione: %d\n", WEXITSTATUS(status));
        }
        else
            printf("Terminazione di %d involontaria\n", term_child);

        exit(0);
    }
    else printf("Creazione fallita!\n");
}
```

ESERCITAZIONE 6: PROCESSI IN UNIX/LINUX (PARTE 2)

SINCRONIZZAZIONE BASATA SU SEGNALI

INTERAZIONE TRA PROCESSI

I processi Unix aderiscono al modello ad **ambiente locale**:

- Spazio di indirizzamento privato.
- Non c'è condivisione di variabili.

L'unica forma di interazione tra processi è la **cooperazione**:

- Sincronizzazione: imposizione di vincoli temporali.
- Comunicazione: scambio di messaggi.

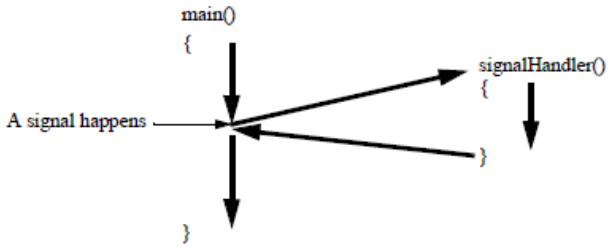
Queste interazioni si basano su astrazioni realizzate dal Kernel: i processi possono interagire mediante chiamate di sistema operativo (system calls).

SINCRONIZZAZIONE MEDIANTE SEGNALI

I segnali sono il meccanismo messo a disposizione dai sistemi Unix/Linux per la sincronizzazione di processi:

- Permettono la notifica di eventi asincroni da parte di un processo a uno o più processi.
- Possono essere utilizzati dal sistema operativo per notificare il verificarsi di eccezioni a un processo utente.

I segnali sono "interrupt software".



La ricezione di un segnale ha 3 possibili effetti sul processo:

1. Viene eseguita una funzione di gestione (handler) definita dal programmatore.
2. Viene eseguita un'azione predefinita dal sistema operativo (default handler).
3. Il segnale viene ignorato.

Nei casi 1 e 2 il processo si comporta in modo asincrono rispetto al segnale:

- L'esecuzione viene interrotta per eseguire l'handler.
- Dopo, se non è terminato, il processo riprende dall'istruzione successiva all'ultima eseguita prima dell'interruzione.

Versioni differenti di Unix possono definire segnali diversi: in Linux sono definiti 32 segnali.

La lista dei segnali è definita nel file di sistema `signal.h`.

Ciascun segnale è identificato da un intero e da un nome simbolico.

`man 7 signal`	Pagina del manuale sui segnali.

NOME SEGNALE	# SEGNALE	DESCRIZIONE SEGNALE
SIGHUP	1	Hang up (il terminale è stato chiuso).
SIGINT	2	Interruzione del processo. CTRL+C da terminale.
SIGQUIT	3	Interruzione del processo e core dump. CTRL+\ da terminale.
SIGKILL	9	Interruzione immediata. Questo segnale non può essere ignorato ed il processo che lo riceve non può eseguire operazioni
SIGTERM	15	Terminazione del programma.
SIGUSR1	10	Definito dall'utente. Default: termina processo.
SIGUSR2	12	Definito dall'utente. Default: termina processo.
SIGSEV	11	Errore di segmentazione.
SIGALRM	14	Il timer è scaduto.
SIGCHLD	17	Processo figlio terminato, fermato, o risvegliato. Ignorato di default.
SIGSTOP	19	Ferma temporaneamente l'esecuzione del processo: questo segnale non può essere ignorato.
SIGTSTP	20	Sospende l'esecuzione del processo. CTRL+Z da terminale.
SIGCONT	18	Il processo può continuare, se era stato fermato da SIGSTOP o SIGTSTP.

SYSTEM CALL PER I SEGNALI

`signal`	Permette di definire la funzione che dovrà gestire il segnale.
`kill`	Invio di segnali.
`alarm` **`sleep`**	Invio implicito di segnali.

SIGNAL

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

Permette di definire la funzione ("handler") che dovrà gestire il segnale "sig":

- La funzione handler deve prevedere un parametro intero, che al momento della ricezione del segnale conterrà il codice del segnale.
- `handler` può valere anche `SIG_IGN` (ignora il segnale) o `SIG_DFL` (ripristina azione di default).

Restituisce un puntatore al precedente handler del segnale, `SIG_ERR` in caso di errore.

`man 2 signal`	Pagina del manuale.

Il figlio eredita dal padre le informazioni relative alla gestione dei segnali: eventuali signal eseguite dal figlio non hanno effetto sul padre.

Le syscall exec non mantengono le associazioni segnale-handler.

! I segnali ignorati, però, continuano ad essere ignorati.

KILL

```
int kill(pid_t pid, int sig)
```

Invia il segnale "sig" al processo "pid":

- **`pid > 0`** → Il segnale viene inviato a pid.
- **`pid == 0`** → Il segnale viene inviato a tutti i processi nello stesso process group del chiamante.
- **`pid == -1`** → Il segnale viene inviato a tutti i processi a cui il chiamante può inviare segnali.
- **`pid < 1`** → Il segnale viene inviato ai processi il cui process group è -pid.

Ritorna zero in caso di successo.

`man 2 kill`	Pagina del manuale.

SLEEP

```
unsigned int sleep(unsigned int seconds)
```

Il processo chiamante va nello stato sleep fino a che:

1. Sono passati `seconds` secondi.
2. Arriva un segnale che non viene ignorato.

Quando è passato il tempo indicato, il processo viene svegliato dal segnale `SIGALARM`.

Ritorna zero se è passato il tempo previsto (seconds), altrimenti il tempo rimasto dopo l'arrivo di un segnale.

`man 3 sleep`	Pagina del manuale.

ALARM

```
unsigned int alarm(unsigned int seconds)
```

Provoca la ricezione di un segnale `SIGALARM` dopo `seconds` secondi.

- Un eventuale "allarme" invocato precedentemente viene cancellato.
- Se `seconds` è zero, viene eliminato un eventuale "allarme" invocato precedentemente.
- Ritorna zero se non c'era un allarme programmato, altrimenti ritorna il numero di secondi mancanti all'ultimo allarme programmato.

`man alarm`	Pagina del manuale.

COMUNICAZIONE MEDIANTE SCAMBIO DI MESSAGGI – PIPE

I processi possono comunicare sfruttando il meccanismo delle pipe.

- Comunicazione indiretta, senza naming esplicito.
- Realizza il concetto di mailbox nella quale si possono accodare messaggi in modo FIFO.
- La pipe è un canale monodirezionale: ci sono due estremi, uno per la lettura e uno per la scrittura.
- Astrazione realizzata in modo omogeneo rispetto alla gestione dei file:
  - A ciascun estremo è associato un file descriptor.
  - I problemi di sincronizzazione sono risolti dalle primitive read/write.

I figli ereditano gli stessi file descriptor e possono utilizzarli per comunicare con il padre e gli altri figli.

! Per la comunicazione di processi che non si trovano nella stessa gerarchia si utilizzano i socket.

`man pipe`	Pagina del manuale.

GESTIONE DEI PROCESSI DA TERMINALE

INVIO DI SEGNALI DA TERMINALE – KILL

Il comando `kill` permette l'invio di segnali a processi da terminale.

```
kill [options]  pid [pid2…]
```

- Il segnale di default è `SIGTERM`.

`kill -l`	Mostra l'elenco dei segnali disponibili.
`kill -SEGNALE pid`	Invia il segnale `SEGNALE` al processo `pid`.

Un utente normale può inviare segnali solo ai processi di cui è proprietario, mentre root può inviare segnali a tutti i processi.

VISUALIZZAZIONE DEI PROCESSI – PS

Il comando ps permette di visualizzare i processi in esecuzione (snapshot, informazione statica).

OPZIONI PRINCIPALI IN LINUX	
`-u utente`	Visualizza i processi dell'utente specificato.
`u`	Formato output utile all'analisi dell'utilizzo delle risorse.
`a`	Processi di tutti gli utenti.
`x`	Anche processi che non sono stati generati da terminali.
`o`	Mostra solo i campi specificati di seguito.
`-O`	Mostra i campi specificati di seguito, oltre ad alcuni campi di default.
`man ps`	Pagina del manuale.

```
studenti@studenti:~$ ps u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
studenti  1511  0.0  0.6  24288  6168 pts/0    Ss+  21:58   0:00 bash
studenti  1783  2.0  0.5  24284  6056 pts/1    Ss   23:58   0:00 bash
studenti  1788  0.0  0.2  19104  2412 pts/1    R+   23:58   0:00 ps u
```

STATI PRINCIPALI	
`S`	Sleep.
`T`	Bloccato.
`R`	Running.
`Z`	Zombie.

ESERCIZI

ESERCIZIO 1

- Realizzare un programma C che stampa un messaggio dentro un ciclo infinito.
- Eseguire il programma e, da terminale, lanciare il segnale SIGINT (CTRL+C) per terminarlo.
- Modificare il programma in modo da gestire SIGINT.
    - Quando riceve il segnale, il processo stampa un messaggio "Ricevuto segnale <codice segnale>".
    - Fare in modo che questo messaggio sia visibile per qualche secondo.
- Adesso il processo non può più essere terminato con CTRL+C.
    - Aprire un nuovo terminale e utilizzare ps per trovare il PID del processo e poi kill per terminarlo.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

void handler(int sig) {
    printf("Segnale %d ricevuto\n", sig);
    sleep(2);
}

int main() {
    signal(SIGINT, handler);

    while(1) {
        printf("hi!\n");
    }
}

/***********************************************
-- Per trovare il PID da terminale: "ps u"
-- Per terminare il processo usare: "kill PID"
***********************************************/
```

ESERCIZIO 2

Partire dall'esercizio es2.c dell'esercitazione precedente (creazione di processi figli e system call wait).

Modificare il codice come segue:

- Il padre genera 3 processi figli.
- Il primo processo figlio creato stampa il solito messaggio e poi termina con exit(0).
- Gli altri 2 processi figli entrano in un loop infinito prima di terminare.
- Il padre, dopo aver creato i processi figli:
  - Si sospende per 3 secondi.
  - Utilizza la primitiva kill per inviare a tutti i figli il segnale SIGUSR1.
- Scrivere un gestore per il segnale SIGUSR1 che stampa il messaggio "Ho ricevuto il segnale <sig>, il mio PID è <pid>" e poi termina il processo con exit(1).
  - Fare in modo che solo i figli vengano terminati in questo modo (il padre ignora il gestore). Dopo aver inviato il segnale, il padre deve continuare regolarmente la sua esecuzione, effettuando le chiamate wait() previste.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NCHILDREN 3

void handler(int sig) {
    printf("Sono il processo %d e ho ricevuto il segnale %d\n",
```

```
            getpid(), sig);
    exit(1);
}

int main()
{
    pid_t pid;
    int children = 0;

    signal(SIGUSR1, handler); // SIGUSR1 is handled by handler

    // creation loop
    while (1) {
        children++; // number of children created
        pid = fork();
        if (pid == 0) {
            // child process. System call fork() returns the child
            // process ID to the parent and returns 0 to the child
            // process.
            printf("Sono il figlio %d\n", getpid());

            // all children but the first enter this infinite loop
            if (children > 1)
                while(1);

            // first child exits
            exit(0);
        }
        else if (pid > 0) {
            // parent process
            // generate up to NCHILDREN processes
            if (children < NCHILDREN)
                continue;

            sleep(3); // sleep for 3 seconds
            signal(SIGUSR1, SIG_IGN); // ignore signal SIGUSR1
            kill(0, SIGUSR1); // send SIGUSR1 to my process group

            // wait for all the children
            while (1) {
                int status;
                pid_t term_child = wait(&status); // returns pid of the
                                                  // terminated child
                if (term_child < 0)
                    break;  // no more children left
                if (WIFEXITED(status)) {
                    printf("Terminazione di %d volontaria\n",
                           term_child);
                    printf("Stato di terminazione: %d\n",
                           WEXITSTATUS(status));
                }
                else
```

```
                    printf("Terminazione di %d involontaria\n",
                           term_child);
            }
            exit(0);
        }
        else {
            printf("Creazione fallita!\n");
            exit(1);
        }
    }
}
```

ESERCITAZIONE 7: PROCESSI IN UNIX/LINUX (PARTE 3)

GERARCHIA DI PROCESSI – INIT SYSTEM

I sistemi in Unix/Linux prevedono un **init** system:

- Processo mandato in esecuzione dal kernel durante il boot.
- È il primo processo ad andare in esecuzione (PID=1).
- Tutti gli altri processi dipendono da init: se un processo termina, gli eventuali figli vengono "adottati" da init.

In Debian/Ubuntu viene utilizzato **systemd** come init system.

`pstree`	Comando per visualizzare l'albero dei processi.

IDENTIFICATORI DI UN PROCESSO

`PID`	ID univoco del processo.
`PPID`	ID del processo padre.
`PGID`	ID del process group a cui appartiene il processo.
`RUID, RGID`	Real User/Group ID.
`EUID, EGID`	Effective User/Group ID.

IDENTIFICATORI E PERMESSI

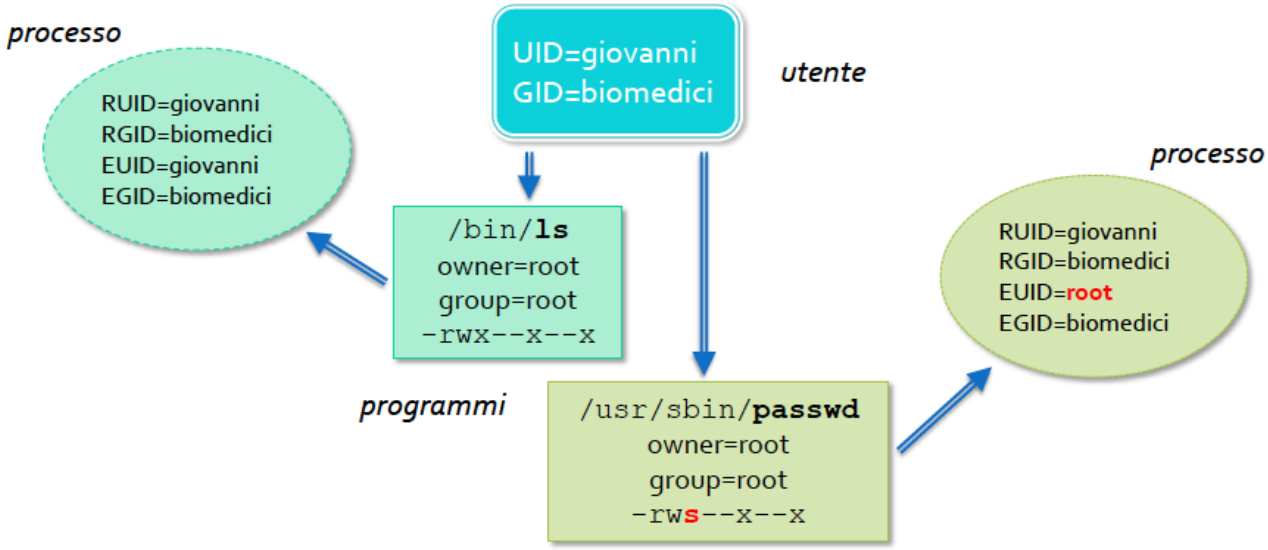
Gli identificatori che determinano i permessi del processo di dividono in:

REAL		EFFECTIVE	
`RUID`	**Real User ID**: ID dell'utente che ha mandato in esecuzione il processo.	`EUID`	**Effective User ID.**
`RGID`	**Real Group ID**: ID del gruppo primario dell'utente che ha mandato in esecuzione il processo.	`EGID`	**Effective Group ID.**

EUID/EGID possono differire da RUID/RGID se il comando eseguito ha il bit SUID o SGID attivo.

- Vengono utilizzati per definire i privilegi di accesso alle risorse e di invocazione di system call del processo.

Un processo utente (non root) può inviare segnali ad un altro processo solo se il suo EUID o RUID coincide con il RUID del processo destinatario.



FUNZIONI GET PER GLI IDENTIFICATORI

- `pid_t getpid()`
- `pid_t getppid()`
- `pid_t getpgrp(void)`
- `uid_t getuid()`
- `uid_t getgid()`
- `uid_t geteuid()`
- `uid_t getegid()`

GRUPPI DI PROCESSI

I processi sono organizzati in gruppi.

- Quando viene mandato in esecuzione un nuovo processo da terminale, al processo viene associato un nuovo process group.
- Se il processo genera dei figli, questi appartengono allo stesso gruppo.
- Il gruppo viene preservato anche dalla syscall exec.

I gruppi permettono di mandare segnali ad una gerarchia di processi e sono alla base del job-control offerto dalla shell.

PROPRIETA' DEI PROCESSI – nice

Lo scheduler Linux assegna la CPU ai processi tenendo conto di un livello di priorità assegnato a ciascun processo.

! La priorità dipende principalmente dalla classe di scheduling del processo ("real-time" o "normale).

La priorità dei processi "normali" può essere in parte controllata mediante il concetto di niceness e la relativa system call `nice`.

- Ad ogni processo è associato un valore di niceness nell'intervallo [-20, 19].
- Un valore di niceness più alto porta ad avere meno priorità di esecuzione.

- In questo modo un processo eseguito in background (non interattivo) può lasciare più tempo di elaborazione agli altri processi.

Solo root può ridurre la niceness di un processo.

! Un utente può solo aumentare la niceness dei suoi processi.

GESTIONE DEI PROCESSI DA TERMINALE (PARTE 2)

JOB-CONTROL

Con job-control si intende la possibilità di sospendere e riattivare gruppi di processi ("jobs") offerta dalla shell mediante opportuni comandi:

- La shell associa un job ID distinto ad ogni comando eseguito.
- Anche una pipeline di comandi (es. `cat file | grep 'text'`) è associato a un solo job.
- I job sono salvati in una tabella specifica, visualizzabile mediante il comando `jobs`.

```
$ ls | sleep 5 &
> [1] 620844
```

```
$ jobs
> [1]+ 620843 Running ls --color=auto
       620844 Running | sleep 5 &
```

```
$ ps -u margherita
> 620844 pts/0 00:00:00 sleep
```

FOREGROUND: un job in esecuzione in **foreground** ha il controllo di standard input, standard output e standard error → Di fatto il processo "prende il controllo del terminale" e lo restituisce alla shell alla sua terminazione.

BACKGROUND: la shell permette anche di eseguire job in **background**.

```
comando &
```

- Il processo non ha più accesso allo standard input.
- L'utente può tornare a utilizzare la shell mentre il job viene completato.

OPERAZIONI SUI PROCESSI FERMATIB: un processo in foreground può essere fermato inviando il segnale `SIGSTP (CTRL+Z)`. Il processo viene messo in background in stato STOPPED.

- È possibile intervenire sui job che sono stati fermati in questo modo:
  - Si utilizza `jobs` per ottenere l'identificatore del job (`JOB_ID`).
  - **`fg JOB_ID`** fa partire `JOB_ID` in foreground.
  - **`bg JOB_ID`** fa partire `JOB_ID` in background.

KILL: si può usare il comando kill anche con i job.

`kill %JOB_ID`	Invia SIGTERM al job specificato.
`kill -n SIG %JOB`	Invia il segnale SIG
`help nomecomando`	Per informazioni su questi comandi di shell.

JOB CONTROL – DISOWN E NOHUP

Se il terminale viene chiuso, i job in esecuzione ricevono il segnale SIGHUP e, di default, vengono terminati.

! Ad esempio, se mi connetto in remoto ad un server e lancio dei comandi di background, i rispettivi processi vengono terminati quando mi disconnetto.

Per fare in modo che SIGHUP non porti alla terminazione di un job si possono usare due strumenti:

- `nohup`
- `disown`

NOHUP

```
nohup comando
```

Il job eseguito in questo modo è immune a SIGHUP.

- Il job non ha più accesso allo stdin: in caso di lettura ottiene EOF.
- Lo stdout viene rediretto su un file chiamato `nohup.out`.

DISOWN

```
disown %JOB_ID
```

Può essere utilizzato per rendere immune a SIGHUP un job già in esecuzione.

- Il job viene rimosso dalla tabella dei job, quindi la shell non invierà più il segnale SIGHUP quando viene chiusa.
- In questo caso è opportuno fare in modo che il job non legga dallo stdin e che l'eventuale output venga rediretto su file per evitare errori durante l'esecuzione.

COMANDI nice E renice

`nice`	Permette di mandare in esecuzione un processo con un valore di niceness specificato. `nice -n valore_nice bzip2 file &`
`renice`	Permette di modificare la niceness di un processo già in esecuzione. `renice valore_nice PID`

MONITOR DI SISTEMA – top

`top`	Permette di visualizzare i processi e di effettuare operazioni su di essi in modo interattivo. • I processi sono ordinati in ordine di utilizzo decrescente della CPU. • È possibile inviare segnali ai processi e cambiarne il valore di niceness. • Vengono visualizzate anche informazioni complessive sul sistema (carico CPU, utilizzo della memoria).

```
top - 00:08:45 up  2:05,  4 users,  load average: 2,05, 2,04, 2,00
Tasks: 139 total,   3 running, 136 sleeping,   0 stopped,   0 zombie
%Cpu(s):100,0 us,  0,0 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   1024372 total,   837496 used,   186876 free,    47764 buffers
KiB Swap:   392188 total,        0 used,   392188 free.   290112 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 2596 studenti  20   0    4076    672    592 R 49,3  0,1  55:50.86 a.out
 2597 studenti  20   0    4076     84      0 R 48,9  0,0  55:50.85 a.out
 1257 studenti  20   0 1377272 221144  68112 S  1,3 21,6   0:48.99 gnome-shell
  538 root      20   0  299784  84508  19300 S  0,3  8,2   0:13.51 Xorg
 1768 www-data  20   0  373400   4488   2856 S  0,3  0,4   0:01.11 apache2
    1 root      20   0  110616   4824   3036 S  0,0  0,5   0:00.53 systemd
    2 root      20   0       0      0      0 S  0,0  0,0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S  0,0  0,0   0:00.03 ksoftirqd/0
```

ESEMPI DI COMANDI INTERATTIVI	
`h`	Help.
`d`	Intervallo di aggiornamento (delay).
`k`	Invio di un segnale.
`n`	Numero di processi da visualizzare.
`r`	Renice.
`u`	Utenti da visualizzare.
`q`	Quit.

ESERCIZI

ESERCIZIO 1

- Scrivere un programma C in cui:
  - Viene creato un processo figlio.
  - Padre e figlio entrano in un loop infinito.
- Mandare in esecuzione il programma in background.
- Utilizzare i comandi del job-control per far tornare il job in foreground.
- Fermare (SIGTSTP) il job da tastiera.
- Usare il comando "`ps o comm,user,pid,ppid,pgid`":
  - Chi è il padre del processo padre?
  - Padre e figlio hanno lo stesso PGID?
- Utilizzare i comandi del job-control per far ripartire il job in background.

- Utilizzare il comando `disown` per fare in modo che il job diventi immune a SIGHUP.
  - Usare il comando `jobs` e verificare che il job non è più nella lista.
  - Chiudere il terminale e aprire un nuovo terminale.
  - Utilizzare il comando `ps` come prima. I due processi (padre e figlio) vengono visualizzati? Quale opzione è necessario aggiungere al comando ps per vederli?
  - Qual è adesso il PPID del processo padre?
- Aprire il manager di sistema `top`:
  - Impostare l'intervallo di aggiornamento (delay) a un secondo.
  - Inviare il segnale SIGTERM ai due processi per terminarli.

```
// Il file programma.c deve contenere una fork(), poi i processi padre e
// figlio eseguono while(1);.

studenti@studenti:~/es7$ gcc programma.c   // l'eseguibile è in a.out

studenti@studenti:~/es7$ ./a.out &              // esecuzione in
                                                // background
[1] 1517

studenti@studenti:~/es7$ jobs
[1]+  In esecuzione           ./a.out &

studenti@studenti:~/es7$ fg 1                   // il job viene portato
                                                // in foreground
./a.out
^Z                                              // utilizzo Ctrl+z per
                                                // stopparlo
[1]+  Fermato                 ./a.out

studenti@studenti:~/es7$ ps o comm,user,pid,ppid,pgid
COMMAND         USER       PID  PPID  PGID
bash            studenti  1482  1478  1482
a.out           studenti  1517  1482  1517
a.out           studenti  1518  1517  1517
ps              studenti  1519  1482  1519

// Guardando PID e PPID è possibile individuare il processo padre (in
// questo caso è 1517).
// Il padre di 1517 è 1482, ossia bash.
// Padre e figlio appartengono allo stesso process group, PGID=1517.

studenti@studenti:~/es7$ bg 1                   // il job riparte in
                                                // background
[1]+ ./a.out &

studenti@studenti:~/es7$ disown %1              // 1 viene rimosso dai
                                                // job
studenti@studenti:~/es7$ jobs                   // verifico che è stato
                                                // rimosso

// A questo punto chiudo il terminale e ne apro uno nuovo
studenti@studenti:~/es7$ ps o comm,user,pid,ppid,pgid
COMMAND         USER       PID  PPID  PGID
bash            studenti  1539  1535  1539
ps              studenti  1546  1539  1546
```

```
// I processi non sono visibili perché non sono più associati ad un
// terminale.
// Per visualizzarli devo aggiungere l'opzione 'x'. Sfrutto anche grep
// per fare in modo che vengano visualizzati i processi che mi
// interessano (e la prima riga).
studenti@studenti:~/es7$ ps xo comm,user,pid,ppid,pgid | grep -e 'a.out'
-e 'USER'
COMMAND         USER       PID  PPID  PGID
a.out           studenti  1517     1  1517
a.out           studenti  1518  1517  1517

// Il PPID del processo padre adesso è 1: è stato "adottato" dal
// processo init (systemd). Perché il il processo è stato disassociato
// dal terminale (bash), quindi i suoi figli vengono adottati da 'init'.

// Aprire top.
// Utilizzare il comando d e settare il delay a 1.
// Utilizzare il comando k per inviare segnali ai processi.
// Inviare SIGTERM ai due processi utilizzando i relativi PID.
```

```c
// programma.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>


int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) {
        // child process. System call fork() returns the child process
        // ID to the parent and returns 0 to the child process.
        printf("Sono il figlio %d\n", getpid());
        while(1);
    }
    else if (pid > 0) {
        printf("Sono il padre %d\n", getpid());
        while(1);
    }
    else {
        printf("Creazione fallita!\n");
        exit(1);
    }
}
```

ESERCIZIO 2

- Mandare in esecuzione (in background) il programma C realizzato nell'esercizio precedente.
- Eseguire nuovamente `gzip` con `nice -n 19`. Cosa succede?

    - Aprire un nuovo terminale e lanciare il comando `top` per controllare quanta CPU viene assegnata al processo.
- Terminare `gzip` (se non ha ancora finito) e stavolta eseguire il comando con `nice -n 15`.
    - Controllare su `top` come viene suddivisa la CPU tra i vari processi.

```
// Creo l'archivio
studenti@studenti:~/es7$ tar cf archivio.tar /usr/bin/
tar: Rimozione di "/" iniziale dai nomi dei membri
tar: Rimozione di "/" iniziale dagli obiettivi dei collegamenti fisici
// opzione 'c': create an archive
// opzione 'f': use archive file ARCHIVE

// Eseguo gzip e misuro il tempo.
studenti@studenti:~/es7$ time gzip -k archivio.tar

real  0m9.111s   // questo è il tempo da considerare (real)
user  0m8.320s
sys   0m0.064s

// Adesso eseguo gzip con nice=19
studenti@studenti:~/es7$ time nice -n 19 gzip -k archivio.tar
gzip: archivio.tar.gz already exists; do you wish to overwrite (y or n)?
y

real  0m11.011s  // questo è il nuovo tempo.
user  0m8.276s
sys   0m0.052s

// In questo caso il tempo è simile a prima. Evidentemente non ci sono
// altri processi "pesanti" in esecuzione, e quindi la CPU viene
// comunque
// assegnata al processo gzip per la maggior parte del tempo (nonostante
// il nice alto).

// Ora mando in esecuzione due processi "pesanti", sfruttando il
// programma
// C dell'esercizio precedente.
studenti@studenti:~/es7$ ./a.out &
[1] 1613
studenti@studenti:~/es7$ time nice -n 19 gzip -k archivio.tar
gzip: archivio.tar.gz already exists; do you wish to overwrite (y or n)?
y

// Stavolta gzip impiega molto più tempo. Aprire un altro terminale
// e lanciare top per controllare come viene ripartita la CPU.

// gzip non ha ancora finito -- lo termino inviando un segnale o con
Ctrl+C.

^C

real  2m39.731s  // dopo quasi 3 minuti non aveva ancora terminato...
user  0m1.104s
sys   0m0.012s
```

```
// Provo a lanciare gzip con nice=5 (mentre è in esecuzione, posso
// aprire
// un altro terminale e controllare la situazione con top).
studenti@studenti:~/es7$ time nice -n 5 gzip -k archivio.tar

real  1m4.495s   // dopo 1 minuto ha finito.
user  0m8.524s
sys   0m0.072s
```

ESERCITAZIONE 8: THREAD POSIX NEI SISTEMI LINUX (PARTE 1)

I THREAD (PROCESSI LEGGERI)

Il thread è un flusso di esecuzione indipendente all'interno di un processo.

- Ad un singolo processo possono essere associati più thread.
- I thread coinvolgono le risorse e lo spazio di indirizzi (o parte di esso) con gli altri thread del processo.
- I thread sono anche detti "processi leggeri" in quanto:
  - La creazione/distruzione di thread è meno onerosa rispetto alla creazione/distruzione di un processo.
  - Il cambio di contesto fra thread dello stesso processo è meno oneroso rispetto al cambio di contesto fra processi.

VANTAGGI DELL'APPROCCIO MULTITHREAD	SVANTAGGI DELL'APPROCCIO MULTITHREAD
1. Interazioni più semplici ed efficienti basate su risorse comuni. 2. Passaggio di contesto fra thread meno oneroso.	Va gestita la concorrenza fra thread: il codice utilizzato deve essere thread safe. • Il codice è scritto in modo da garantire il corretto comportamento del programma e l'assenza di interazioni non volute fra i thread. • Le risorse condivise devono essere accedute in mutua esclusione, o implementazione codice rientrante (no variabili globali condivise).

I THREAD IN LINUX

Linux supporta nativamente, a livello di kernel, il concetto di thread.

- Il thread è l'unità di scheduling e può essere eseguito in parallelo con altri thread.
- Il "processo tradizionale" dei sistemi Unix può essere visto come un thread che non condivide risorse con altri thread.

LIBRERIA PTHREADS

Lo standard POSIX definisce la libreria pthreads per la programmazione di applicazioni multithread portabili.

Utilizzo:

1. Includere la libreria `#include <pthreads.h>`.
2. Compilare specificando l'uso della libreria: `gcc <opzioni> file.c -lpthread`.
   a. Su Debian: `gcc <opzioni> file.c -lpthread -std = c99`.
3. Pagine del manuale sulla libreria (per installare manuale libreria, se non presente, `sudo apt install glibc-doc`):
   a. `man pthreads`.
   b. `man nomefunzione`.

IDENTIFICATORI DEL THREAD

Un thread è identificato da un ID di tipo **`pthread_t`**.

`pthread_t pthread_self(void)`	Funzione per conoscere l'ID del thread corrente.

È un tipo "opaco", che può essere utilizzato solo mediante apposite funzioni:

- Non ha senso stamparlo a video.
- Per fare un confronto tra due ID thread è necessario usare la funzione

`pthread_equals(tid1, tid2)`

Su Linux c'è anche la funzione **`gettid()`**, che ritorna un thread ID (TID) analogo del process ID (PID).

- Se il thread è l'unico thread del processo, il suo TID è uguale al PID.
- `gettid()` è Linux-specific, quindi non deve essere usata se si vuole ottenere del codice portabile su sistemi Unix tradizionali.
- Il valore riportato da `gettid()` è lo stesso usato dai comandi `top`, `ps` ecc…

CREAZIONE DI UN THREAD

In Linux l'esecuzione di un programma determina la creazione di un primo thread che esegue il codice del main.

Il thread iniziale può generare una gerarchia di thread utilizzando:

```
int pthread_create(pthread_t* thread, const pthread_attr_t* attr,
                   void* (*start_routine)(void *), void* arg);
```

`pthread_t thread`	Puntatore ad identificatore di thread, dove verrà scritto l'ID del thread creato.
`const pthread_attr_t* attr`	Attributi del thread. ! NULL per utilzzare valori di default.
`void* (*start_routine)(void *)`	Puntatore alla funzione che contiene il codice del nuovo thread.
`void* arg`	Puntatore che viene passato come argomento a `start_routine`.

Il valore di ritorno è zero in assenza di errore, diverso da zero altrimenti.

TERMINAZIONE E JOIN

Un thread può terminare la sua esecuzione con

```
void pthread_exit(void* retval);
```

Ha i seguenti effetti:

1. L'esecuzione del thread termina e il sistema libera le risorse allocate.
2. Quando un thread "padre" (ad esempio il main) termina prima dei thread figli:
   a. Se non chiama la pthread_exit → I figli vengono terminati.
   b. Se chiama la pthread_exit → I figli continuano la loro esecuzione.

void* retval	Valore di ritorno del thread (exit status) consultabile da altri thread che utilizzano la `pthread_join`.

Un thread può bloccarsi in attesa della terminazione di un thread specifico:

```
int pthread_join(pthread_t thread, void** retval)
```

pthread_t thread	ID del thread di cui attendere la terminazione.
void retval**	Puntatore ad un puntatore dove verrà salvato l'indirizzo restituito dal thread con la `pthread_exit`. Può essere impostato a NULL (in questo caso viene ignorato).

Ritorna zero in caso di successo, altrimenti un codice di errore.

- Esempio: se un altro thread ha già fatto join sullo stesso thread, o se c'è rischio di deadlock.

ESEMPIO CREAZIONE THREAD

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* Corpo del thread */
void* tr_code(void* arg) {
      printf("Hello World! My arg is %d \n", *(int*)arg);
      free(arg);
      pthread_exit(NULL);
}

/* Main function */
int main() {
      pthread_t tr1, tr2;
      int* arg1 = (int*)malloc(sizeof(int));
      int *arg2 = (int*)malloc(sizeof(int));
      *arg1 = 1;
      *arg2 = 2;

      int ret;

      ret = pthread_create(&tr1, NULL, tr_code, arg1);
      if(ret) {
            printf("Error: return code from pthread_create is %d \n",ret);
            exit(-1);
      }

      ret = pthread_create(&tr2, NULL, tr_code, arg2);
      if(ret) {
            printf("Error: return code from pthread_create is %d \n",ret);
            exit(-1);
      }
```

```
      pthread_exit(NULL);
}
```

ESEMPIO CREAZIONE E PASSAGGIO DI PARAMETRI CON NTHREADS

```
…headers
# define NTHREADS 10
…codice thread

int main() {
      pthread_t tr[NTHREADS];
      int* args[NTHREADS];
      int ret;

      for(int i = 0; i < NTHREADS; i++) {
            args[i] = (int*)malloc(sizeof(int));
            *args[i] = i;
            ret = pthread_create(&tr[i], NULL, tr_code, args[i]);
            …gestione ret value
      }

      pthread_exit(NULL);
}
```

MUTUA ESCLUSIONE

Per risolvere problemi di mutua esclusione, la libreria pthread mette a disposizione l'astrazione della variabile di tipo **mutex**, analoga all'astrazione di semaforo binario: una variabile mutex permette di proteggere l'accesso a variabili condivise su cui operano più thread.

MUTEX

Nella libreria pthread è definito il tipo `pthread_mutex_t` che rappresenta implicitamente:

- Lo stato del mutex.
- La coda dove verranno sospesi i processi in attesa che il mutex sia libero.

È un semaforo binario, quindi il suo stato può assumere due valori: libero o occupato.

INIZIALIZZAZIONE

`pthread_mutex_t M`	Definizione di una variabile mutex.

Per inizializzare una variabile mutex, si utilizza la funzione

```
int pthread_mutex_init(pthread_mutex_t* M,
                       const pthread_mutexattr_t* mattr)
```

`pthread_mutex_t* M`	Puntatore al mutex da inizializzare.

`const pthread_mutexattr_t* mattr`	Puntatore a una struttura con attributi di inizializzazione. Con NULL vengono utilizzati i valori di default (mutex libero).

LOCK E UNLOCK

La wait sulla variabile mutex è realizzata con la primitiva

```
int pthread_mutex_lock(pthread_mutex_t* M)
```

La signal sulla variabile mutex è realizzata con la primitiva

```
int pthread_mutex_unlock(pthread_mutex_t* M)
```

Ritornano zero in caso di successo, altrimenti un codice di errore.

UTILIZZO

`pthread_mutex_t M;` `pthread_mutex_init(&M, NULL);`	Definizione e inizializzazione.
`pthread_mutex_lock(&M);`	Lock sulla variabile mutex prima di accedere alla risorsa condivisa.
`pthread_mutex_unlock(&M);`	Unlock sulla variabile mutex dopo aver utilizzato la risorsa condivisa.

Se più thread provano ad accedere alla risorsa (lock), solo uno di essi potrà accedere, mentre gli altri rimarranno bloccati.

- Dopo aver occupato e utilizzato la risorsa, il thread provvederà a "liberarla" con la primitiva unlock: in questo modo uno dei thread (eventualmente) bloccati sulla variabile mutex potrà accedere alla risorsa.

ESERCIZI

ESERCIZIO 1.1

Scrivere un programma C in cui il main genera un numero NTHREADS = 4 di thread. A ciascun thread figlio viene passato un intero <arg> che parte da 1 e arriva a NTHREADS.

I thread eseguono tutti lo stesso codice (stessa funzione), un ciclo for con 4 iterazioni in cui:

- Viene stampato un messaggio del tipo "Sono il thread <arg>".
- Il thread va in sleep per <arg> secondi.

Il padre, dopo aver creato i thread figli, chiama la pthread_exit(NULL): cosa succede se commentiamo questa chiamata?

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define NTHREADS 4
```

```c
/* Corpo del thread */
void* tr_code(void * arg)
{
    int id = *(int*)arg;
    for (int i=0; i<4; i++) {
        printf("Hello World! My arg is %d\n", id);
        sleep(id);
    }
    free(arg);
    pthread_exit(NULL);
}

int main ()
{
    pthread_t tr[NTHREADS];
    int* args[NTHREADS];
    int ret;
    for (int i=0; i<NTHREADS; i++) {
        args[i] = (int*)malloc(sizeof(int));
        *args[i] = i+1; // l'esercizio chiede di usare un ID da 1 a
                        // NTHREADS
        ret = pthread_create(&tr[i], NULL, tr_code, args[i]);
        if (ret){
           printf("Error: return code from pthread_create is %d\n",
                  ret);
           exit(-1);
        }
    }

    pthread_exit(NULL);
}
```

ESERCIZIO 1.2

Modificare il codice in modo che il thread padre faccia join in attesa del secondo figlio prima di terminare (controllare la riuscita dell'operazione).

A questo punto, modificare in modo che il secondo thread figlio creato faccia join in attesa del padre prima di terminare.

- Suggerimento: prima di creare i thread, salvare il thread ID del padre in una variabile globale.
- Cosa succede? Le join hanno successo?

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define NTHREADS 4

// variabile globale per "condividere" l'ID del padre con i thread figli
```

```c
pthread_t father;

/* Corpo del thread */
void* tr_code(void * arg)
{
    int id = *(int*)arg;
    for (int i=0; i<4; i++) {
        printf("Hello World! My arg is %d\n", id);
        sleep(id);
    }
    // Il secondo figlio deve chiamare la funzione join utilizzando l'ID
    // del padre.

    if (id == 2) {
        printf("I'm child %d. Waiting for father to exit. \n", id);
        int retjoin = pthread_join(father, NULL);
        if (retjoin != 0)
            printf("I'm child %d, join error, code %d\n", id, retjoin);
        else
            printf("I'm child %d, Join OK\n", id);
    }
    free(arg);
    pthread_exit(NULL);
}

int main ()
{
    pthread_t tr[NTHREADS];
    int* args[NTHREADS];
    father = pthread_self();
    int ret;
    for (int i=0; i<NTHREADS; i++) {

        args[i] = (int*)malloc(sizeof(int));
        *args[i] = i+1; // l'esercizio chiede di usare un ID da 1 a
                        // NTHREADS
       ret = pthread_create(&tr[i], NULL, tr_code, args[i]);
       if (ret){
           printf("Error: return code from pthread_create is %d\n",
                  ret);
           exit(-1);
       }
    }
    printf("I'm the father. Created %d threads\n", NTHREADS);
    // Il padre chiama la funzione join "in attesa" del secondo figlio.
    /*
    printf("I'm the father. Waiting for child %d to exit\n", *args[1]);
    int retjoin = pthread_join(tr[1], NULL);
    if (retjoin != 0)
        printf("I'm the father, join error, codice %d\n", retjoin);
    else
        printf("I'm the father. Join OK\n");
```

```
    */

    printf("I'm the father. Calling pthread_exit()\n");
    pthread_exit(NULL);
}
```

ESERCIZIO 1.3

- Rimuovere le pthread_join e tornare al punto 1.1.
- Rimuovere la funzione sleep dal ciclo for dei thread.
- Aumentare a 100'000 il numero di iterazioni e a 12 il numero di threads (NTHREADS = 12).
- Dentro il ciclo for dei thread:
  - Incrementare cont (cont++).
  - Aggiungere al messaggio stampato anche il valore di cont:
    "Sono il thread <arg>, il valore di cont è <cont>"
  - Eseguire il programma più volte e controllare il valore finale di cont.
- Il risultato è sempre quello atteso?

```
// Be sure to assign more than one CPU to the VM on VirtualBox, as
// Virtualbox will treat logical CPU cores as independent cores that can
// be assigned.
// See: https://www.youtube.com/watch?v=42769_AGbx8
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define NTHREADS 12

int cont = 0;

/* Corpo del thread */
void* tr_code(void * arg)
{
    int id = *(int*)arg;
    for (int i=0; i<100000; i++) {
        cont++;
        printf("Hello World! My arg is %d and cont is %d\n", id, cont);
    }
    free(arg);
    pthread_exit(NULL);
}

int main ()
{
    pthread_t tr[NTHREADS];
    int* args[NTHREADS];
    int ret;
    for (int i=0; i<NTHREADS; i++) {
        args[i] = (int*)malloc(sizeof(int));
```

```
        *args[i] = i+1; // l'esercizio chiede di usare un ID da 1 a
                        // NTHREADS
       ret = pthread_create(&tr[i], NULL, tr_code, args[i]);
       if (ret){
           printf("Error: return code from pthread_create is %d\n",
                  ret);
           exit(-1);
       }
    }

    pthread_exit(NULL);
}
// risultato atteso: 12*100'000 = 1'200'000
// Valori finali:
// esecuzione 1: Hello World! My arg is 10 and cont is 1199964
// esecuzione 2: Hello World! My arg is 9 and cont is 1199961
// esecuzione 3: Hello World! My arg is 10 and cont is 1199969
```

ESERCIZIO 2

Modificare il codice dell'esercizio 1.3 in modo che l'accesso alla risorsa condivisa (contatore cont) avvenga in modo corretto.

- Sfruttare una variabile globale mutex, inizializzata nel main e utilizzata dai thread per accedere alla "sezione critica".

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define NTHREADS 12

int cont = 0;
pthread_mutex_t M;

/* Corpo del thread */
void* tr_code(void * arg)
{
    int id = *(int*)arg;
    for (int i=0; i<100000; i++) {
        pthread_mutex_lock(&M);
        cont++;
        printf("Hello World! My arg is %d and cont is %d\n", id, cont);
        pthread_mutex_unlock(&M);

        // Nota: la printf e' all'interno della "sezione critica"
        // perche' utilizza cont e vogliamo essere sicuri che il valore
        // stampato sia effettivamente cont++ (se mettiamo la prinft
        // fuori dai lock, un altro processo potrebbe "inserirsi" tra la
        // cont++ e printf).
    }
```

```
    free(arg);
    pthread_exit(NULL);
}

int main ()
{
    pthread_mutex_init(&M, NULL);
    pthread_t tr[NTHREADS];
    int* args[NTHREADS];
    int ret;
    for (int i=0; i<NTHREADS; i++) {
        args[i] = (int*)malloc(sizeof(int));
        *args[i] = i+1; // l'esercizio chiede di usare un ID da 1 a
                        // NTHREADS
        ret = pthread_create(&tr[i], NULL, tr_code, args[i]);
        if (ret){
            printf("Error: return code from pthread_create is %d\n",
                   ret);
            exit(-1);
        }
    }

    pthread_exit(NULL);
}

// risultato finale: Hello World! My arg is 3 and cont is 1200000
```

ESERCITAZIONE 9: THREAD POSIX NEI SISTEMI LINUX (PARTE 2)

SINCRONIZZAZIONE DEI THREAD

Il mutex è lo strumento messo a disposizione dalla libreria pthread per la sincronizzazione **indiretta** dei thread: permette di garantire l'accesso in mutua esclusione a una risorsa condivisa.

Per la sincronizzazione **diretta** dei thread la libreria definisce le variabili condizione (condition variables).

- Un thread può sospendersi in attesa del verificarsi di una determinata condizione.
- Permette di realizzare politiche avanzate di accesso alle risorse condivise e di sincronizzare i thread.

VARIABILI CONDIZIONE

Una variabile condizione è di fatto una coda nella quale i thread possono sospendersi volontariamente in attesa di una condizione.

Definizione e funzione di inizializzazione:

```
pthread_cond_t C

int pthread_cond_init(pthread_cond_t* C, pthread_cond_attr_t* attr)
```

pthread_cond_t* C	Puntatore alla variabile da inizializzare.
pthread_cond_attr_t* attr	Attributi specificati per la condizione, inizializzata a default se attr = NULL.

Un thread può effettuare due "operazioni" su una variabile condition:

1. Sospendersi (wait) sulla variabile condition. Il thread, dopo aver verificato una determinata condizione logica, si sospende sulla variabile condition, in attesa di essere risvegliato da un altro thread.
2. Risvegliare uno dei thread (signal) o tutti i thread (broadcast) sospesi sulla variabile condition.

WAIT

La sospensione (wait) viene utilizzata al verificarsi di una particolare condizione logica.

- Esempio: un thread produttore ha verificato che il buffer condiviso è pieno; quindi, si blocca sulla condition "pieno" in attesa che un thread consumatore lo risvegli (dopo avere liberato il buffer).

Schema generico di utilizzo della wait:

```
while(condizione logica)
      wait(condition_variable);
```

Perché while e non if?

- Quando il thread viene risvegliato non va subito in esecuzione (la signal della libreria pthread è di tipo **signal&continue**): altri thread potrebbero inserirsi e alterare la condizione (nell'esempio del produttore/consumatore, un altro thread potrebbe riempire nuovamente il buffer).
- **Spurious wake-ups** dovuti a scheduling (prioriry, round-robin…) dei thread, a signal da parte del sistema operativo o comunque non dovuti al programmatore.

È quindi necessario ricontrollare la condizione dopo essere stati svegliati.

WAIT E MUTUA ESCLUSIONE

La "condizione logica" (esempio: `elementi_nel_buffer == MAX`) è basata su una risorsa condivisa (esempio: `elementi_nel_buffer`): la verifica della condizione deve essere eseguita in mutua esclusione.

Tenendo conto di questo aspetto, la primitiva di wait offerta dalla libreria pthread permette di "associare" una variabile mutex a una variabile condition. In questo modo:

- L'accesso in mutua esclusione sulla `condizione logica` viene rilasciato quando il thread si sospende con la wait.
- Il thread, dopo essere stato risvegliato, prova automaticamente ad eseguire di nuovo il lock sulla variabile mutex (se il mutex è occupato, il thread si blocca in attesa che venga liberato prima di proseguire).

PRIMITIVA WAIT

```
int pthread_cond_wait(pthread_cont_t* C, pthread_mutex__t* M)
```

pthread_cond_t* C	Variabile condizione su cui sospendersi.
pthread_mutex_t* M	Mutex associato alla variabile condizione: viene liberato automaticamente quando il thread si sospende. • Viene eseguito un nuovo lock quando il thread viene risvegliato.

La chiamata ha due effetti:

1. Il thread viene sospeso nella coda associata a C.
2. Il mutex M viene liberato (quando il thread verrà risvegliato, proverà nuovamente a fare lock su M).

RISVEGLIO – PRIMITIVA SIGNAL

Il risveglio di un thread sospeso su una variabile condition C avviene mediante la primitiva signal:

```
int pthread_cond_signal(pthread_cont_t* C)
```

Come conseguenza della signal:

- Se esistono thread in coda sulla condition, **almeno uno** viene risvegliato.
  - Almeno uno a causa degli spurious wake-ups.

- Se non vi sono thread sospesi, non ha alcun effetto.

La politica della signal della libreria pthread è di tipo *signal&continue*:

- Il thread che esegue la signal continua la sua esecuzione e mantiene il controllo del mutex fino al suo esplicito rilascio.
- Il thread che aveva effettuato la wait ed è stato risvegliato deve verificare nuovamente la condizione.

"ALMENO UN THREAD DEVE ESSERE RISVEGLIATO"

```
while(condizione logica)
      pthread_cond_wait(&mutex, &condition_variable);
```

```
pthread_cond_wait(mutex, cond):
      value = cond->value;                          /* 1 */
      pthread_mutex_unlock(mutex);                  /* 2 */
      pthread_mutex_lock(cond->mutex);              /* 10 */
      if (value == cond->value) {                   /* 11 */
            me->next_cond = cond->waiter;
            cond->waiter = me;
            pthread_mutex_unlock(cond->mutex);
            unable_to_run(me);
      } else
            pthread_mutex_unlock(cond->mutex);      /* 12 */
      pthread _mutex_lock(mutex);                   /* 13 */

pthread_cond_signal(cond):
      pthread_mutex_lock(cond->mutex);              /* 3 */
      cond->value++;                                /* 4 */
      if(cond->waiter) {                            /* 5 */
            sleeper = cond->waiter;                 /* 6 */
            cond->waiter = sleeper->next_cond;      /* 7 */
            able_to_run(sleeper);                   /* 8 */
      }
      pthread_mutex_unlock(cond->mutex);            /* 9 */
```

Thread A sta eseguendo la `pthread_cond_wait()`, mentre thread C è dormiente sulla coda della `condition_variable`.

In questo istante, il thread B esegue pthread_cond_signal:

1. Il thread C si risveglia.
2. Il thread A si accorge che `value != cond->value`, e quindi non si addormenta sulla `condition_variable`.

Adesso sia il thread A che il thread C sono svegli e devono:

1. Acquisire il mutex associato alla condition_variable.
2. Ricontrollare la condizione logica.

Solamente uno dei due thread andrà avanti.

RISVEGLIO – PRIMITIVA BROADCAST

Per risvegliare tutti i thread in coda su una condition, è possibile utilizzare la funzione

```
int pthread_cond_broadcast(pthread_cond_t* C)
```

ESEMPIO 1 – PRODUTTORI E CONSUMATORI

Dei thread accedono a una risorsa condivisa, ad esempio un buffer di interi gestito in modo circolare (ring buffer).

- I thread consumatori prelevano valori (leggono) dal buffer.
- I thread produttori inseriscono nuovi valori (scrivono) nel buffer.

La gestione del buffer ha due vincoli:

1. Non si può prelevare dal buffer vuoto.
2. Non si può inserire nel buffer pieno.

Si può realizzare la risorsa condivisa in questo modo:

```
typedef struct {
      int buffer[BUFFER_SIZE];
      int readInd, writeInd; // indici di read/write nel buffer
      int cont; // elementi nel buffer

      pthread_mutex_t M; // per garantire accesso esclusivo alle risorse

      pthread_cond_t FULL; // condition var. buffer pieno
      pthread_cond_t EMPTY; // condition var. buffer vuoto
} risorsa;
```

La risorsa deve essere inizializzata:

```
risorsa r; // variabile globale, condivisa da tutti i thread

int main() {
      pthread_mutex_init(&r.M, NULL);

      pthread_cond_init(&r.FULL, NULL);
      pthread_cond_init(&r.EMPTY, NULL);

      r.readInd = r.writeInd = r.cont = 0;
      …
}
```

CONSUMATORE

Il consumatore deve:

1. Assicurarsi che il buffer non sia vuoto prima di prelevare un dato.
2. Risvegliare un produttore, se c'è, dopo aver prelevato un dato.

```
…
int val; // per il dato che verrà prelevato dal buffer

pthread_mutex_lock(&r.M);
while (r.cont == 0) // buffer vuoto?
     pthread_cond_wait(&r.EMPTY, &r.M); // buffer vuoto, attendi…

// Preleva un dato e aggiorna lo stato del ring buffer
val = r.buffer[r.readInd];
r.cont--;
r.readInd = (r.readInd+1) % BUFFER_SIZE; // gestione circolare

// Risveglia un eventuale thread produttore
pthread_cond_signal(&r.FULL);
pthread_mutex_unlock(&r.M);
…
```

PRODUTTORE

Il produttore deve:

1. Assicurarsi che il buffer non sia pieno prima di inserire un dato.
2. Risvegliare un consumatore eventualmente sospeso.

```
…
pthread_mutex_lock(&r.M);
while (r.cont == BUFFER_SIZE) // buffer pieno?
     pthread_cond_wait(&r.FULL, &r.M); // buffer pieno, attendi…

// Inserisci un dato e aggiorna lo stato del ring buffer
r.buffer[r.writeInd] = val;
r.cont++;
r.writeInd = (r.writeInd+1) % BUFFER_SIZE; // gestione circolare

// Risveglia un eventuale thread consumatore
pthread_cond_signal(r.EMPTY);
pthread_mutex_unlock(&r.M);
…
```

ESEMPIO 2 – ACCESSO LIMITATO A RISORSA

L'utilizzo delle variabili condition permette di realizzare politiche di accesso a una risorsa più avanzate.

Ad esempio, si immagini uno scenario in cui:

- `NTHREADS` utilizzano periodicamente una risorsa.
- La risorsa può essere utilizzata contemporaneamente da un numero massimo `MAX_T` di thread.
- `NTHREADS > MAX_T`.

In questo caso, si può utilizzare una variabile condition PIENO associata alla condizione

```
#thread che usano la risorsa == MAX_T
```

Un thread esegue una "fase di ingresso" prima di usare la risorsa e una "fase di uscita" dopo aver usato la risorsa.

VARIABILI GLOBALI

```
#DEFINE MAX_T 10

int n_users = 0; // numero di thread che stanno usando la risorsa
pthread_cond_t FULL; // condition var. limite di utilizzo raggiunto
pthread_mutex_t M; // Mutex per l'accesso esclusivo a n_users
```

FASE DI INGRESSO

Il thread controlla se è stato raggiunto il numero massimo di utilizzatori, ed eventualmente si sospende.

```
…

pthread_mutex_lock(&M);
while (n_users == MAX_T) // massimo numero di users raggiunto
      pthread_cond_wait(&FULL, &M); // attendi…
n_users++;
pthread_mutex_unlock(&M); // rilascio il lock

…
// Uso della risorsa
…
```

FASE DI USCITA

Il thread, dopo aver usato la risorsa, aggiorna lo stato della risorsa e risveglia un thread eventualmente sospeso.

```
…
// Uso della risorsa
…

pthread_mutex_lock(&M);
n_users--;
pthread_cond_signal(&FULL);
pthread_mutex_unlock(&M);
```

ESERCIZI

ESERCIZIO 1

Completare il codice in es1.c in modo che le funzioni deposit e withdraw operino correttamente sul saldo (balance) di un conto corrente.

--Le parti che andavano completate sono evidenziate in rosso--

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define N_PRODUCERS 20
#define N_CONSUMERS 20

int balance;
pthread_mutex_t M;
pthread_cond_t LOW_BALANCE;

// Prelievo di amount dal credito (balance).
// Stampa un messaggio di debug:
//      "Consumer <tid>. Prelevato <amount>. Credito: <balance>"
void withdraw(int amount, int tid){
    pthread_mutex_lock(&M);
    while (balance - amount < 0)
        pthread_cond_wait(&LOW_BALANCE, &M);
    balance = balance - amount;
    printf("[ Consumer %d ]: \tPrelevato %d. Credito residuo: %d.\n",
           tid, amount, balance);
    pthread_mutex_unlock(&M);
}

// Aumenta balance con amount;
// Stampa un messaggio di debug:
//      "Producer <tid>. Depositato <amount>. Credito: <balance>"
void deposit(int amount, int tid){
    pthread_mutex_lock(&M);
    balance = balance + amount;
    printf("[ Producer %d ]: \tDepositato %d. Credito residuo: %d.\n",
           tid, amount, balance);
    // In questo esempio devo usare la broadcast e non la signal.
    // Infatti, dopo il deposito non so quanti (e quali) thread
    // consumatori potranno fare il prelievo (in questo esempio i valori
    // di prelievo/deposito sono delle costanti, ma in realta' non sara'
    // cosi').
    // Quindi la soluzione e' risvegliare tutti i consumer in attesa
    // (broadcast).
    pthread_cond_broadcast(&LOW_BALANCE);
    pthread_mutex_unlock(&M);
}

void* producer(void * id){
   int tid = *(int*)id;
   for (int i = 0; i < 4; i++)
      deposit(25, tid);
   free(id);
   pthread_exit(NULL);
}
```

```c
void* consumer(void * id){
    int tid = *(int*)id;
    for (int i = 0; i < 10; i++)
        withdraw(10, tid);
    free(id);
    pthread_exit(NULL);
}

int main(){
    pthread_t consumers[N_CONSUMERS];
    pthread_t producers[N_PRODUCERS];
    pthread_mutex_init(&M, NULL);
    pthread_cond_init(&LOW_BALANCE, NULL);
    balance = 0;
    // IDs
    int* consumerID[N_CONSUMERS];
    int* producerID[N_PRODUCERS];

    // Create prod/cons threads.
    for (int i = 0; i < N_CONSUMERS; i++){
        consumerID[i] = (int*)malloc(sizeof(int));
        *consumerID[i] = i+1;
        pthread_create(&consumers[i], NULL, consumer, consumerID[i]);
    }
    for (int i = 0; i < N_PRODUCERS; i++){
        producerID[i] = (int*)malloc(sizeof(int));
        *producerID[i] = i+1;
        pthread_create(&producers[i], NULL, producer, producerID[i]);
    }

    // Wait for termination of all threads.
    for (int i = 0; i < N_PRODUCERS; i++)
        pthread_join(producers[i], NULL);
    for (int i = 0; i < N_CONSUMERS; i++)
        pthread_join(consumers[i], NULL);

    // Print final balance.
    printf("[ Main ]: \tCredito finale: %d.\n", balance);
    return 0;
}
```

ESERCIZIO 2

Completare il file es2.c in modo da ottenere questa sincronizzazione fra i thread: i thread, prima di terminare, "aspettano" che anche gli altri abbiano terminato la sleep.

- Utilizzare un intero condiviso "checked_threads" per contare i thread che hanno terminato la sleep.
- Utilizzare una variabile condition per sospendere i thread in attesa degli altri.
- L'ultimo thread a raggiungere il "checkpoint" dopo la sleep provvede a svegliare gli altri, in modo che tutti possano terminare.

```c
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

#define NTHREADS    4

pthread_mutex_t M;
pthread_cond_t CHECKPOINT; // condition variable
int checked_threads = 0; // conto dei thread che hanno eseguito la
                         // sleep() e si sono "registrati" al
                         // checkpoint.

void* thread_function(void * arg)
{
    int delay = rand()%15;
    int id = *(int*)arg;
    printf("[Thread %d] Waiting for %d secs...\n", id, delay);
    sleep(delay);
    printf("[Thread %d] ...awake, waiting for the other
           threads...\n",id);

    // Checkpoint -- assicurare che i thread "si aspettino a vicenda"
    // prima di terminare e stampare "OK".
    pthread_mutex_lock(&M);
    checked_threads++;
    if (checked_threads < NTHREADS)
        pthread_cond_wait(&CHECKPOINT, &M);
    else {
        pthread_cond_broadcast(&CHECKPOINT);
    }
    pthread_mutex_unlock(&M);

    printf("[Thread %d] ...ok!\n",id);
    free(arg);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_mutex_init(&M, NULL);
    pthread_cond_init(&CHECKPOINT, NULL);
    pthread_t threads[NTHREADS];

    int* params[NTHREADS];
    int i, rc;

    srand ( time(NULL) );

    printf("[Main] Starting...\n");
    for (i=0; i<NTHREADS; i++) {
```

```
        params[i] = (int*)malloc(sizeof(int));
        *params[i] = i+1;
        rc = pthread_create(&threads[i], NULL, thread_function,
                            params[i]);
        if (rc){
            perror("[Main] ERROR from pthread_create()\n");
            exit(-1);
        }
    }

    pthread_exit(NULL);
}
```

ESERCIZIO 3

Il codice in es3.c simula il comportamento di alcuni giocatori che provano a indovinare un numero sorteggiato dal main:

- Il main sceglie il numero.
- I thread figli provano a indovinare.
- Se nessuno ha indovinato, il main fornisce un suggerimento (intervallo) e la scommessa viene ripetuta.

Completare il codice thread dei "giocatori" in modo da garantire la corretta sincronizzazione fra thread "giocatori" e thread main:

- Il thread giocatore deve inserire la sua "scommessa" nel buffer e risvegliare il thread main se tutti i giocatori hanno già fatto la loro scommessa.
- Il thread giocatore deve attendere che i risultati siano pronti prima di poter giocare di nuovo (se è necessario un altro turno per stabilire il vincitore).

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define NUM_PLAYERS 7

int bet[NUM_PLAYERS]; // vettore con le ultime scommesse. La scommessa
                      // del player i è in posizione bet[i].
// intervallo min-max usato come suggerimento per facilitare i
// giocatori.
int max;
int min;

pthread_mutex_t m;
pthread_cond_t BETTING; // condizione "SCOMMESSE IN CORSO"
pthread_cond_t RESULTS; // condizione "RISULTATI NON ANCORA PRONTI"
int remaining;          // #giocatori che ancora devono fare la loro
                        // scommessa
int winner;             // vincitore

void * player(void * arg){
```

```
    int id = *(int*)arg;
    while(1) {
        // Suggerimento: per fare la scommessa usare: rand()%(max+1-min)
        // + min
        // Il giocatore, dopo aver fatto la scommessa, deve svegliare il
        // padre (main) se necessario. Poi deve attendere che i
        // risultati siano pronti.
        pthread_mutex_lock(&m);
        bet[id] = rand()%(max+1-min) + min;
        printf("[Thread %d]: Penso sia %d\n", id, bet[id]);
        remaining--;
        if (remaining == 0)
            pthread_cond_signal(&BETTING);
        pthread_cond_wait(&RESULTS, &m); //una volta risvegliato, il
        // thread può continuare dall'istruzione successiva solamente se
        // ha riottenuto nuovamente il lock su m (per questo poi è
        // necessario eseguire l'unlock su m)
        pthread_mutex_unlock(&m);

        if (winner >= 0){
            if (winner == id){
                printf("[Thread %d]: Ho vinto\n", id);
            }
            else {
                printf("[Thread %d]: Ho perso\n", id);
            }
            free(arg);
            pthread_exit(NULL);
        }
    }
}

int main(){
    int x; // numero da indovinare
    min = 1;
    max = 2000;
    winner = -1;
    remaining = NUM_PLAYERS;
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&RESULTS, NULL);
    pthread_cond_init(&BETTING, NULL);

    pthread_t players[NUM_PLAYERS];
    int* playerIDs[NUM_PLAYERS];

    srand(time(NULL));
    x = rand()%(max+1-min) + min; // inizializzazione numero da
                                  // indovinare
    printf("[Main]: Il numero da indovinare e' compreso tra %d e %d\n",
           min,max);

    for (int i = 0; i < NUM_PLAYERS; i++){
```

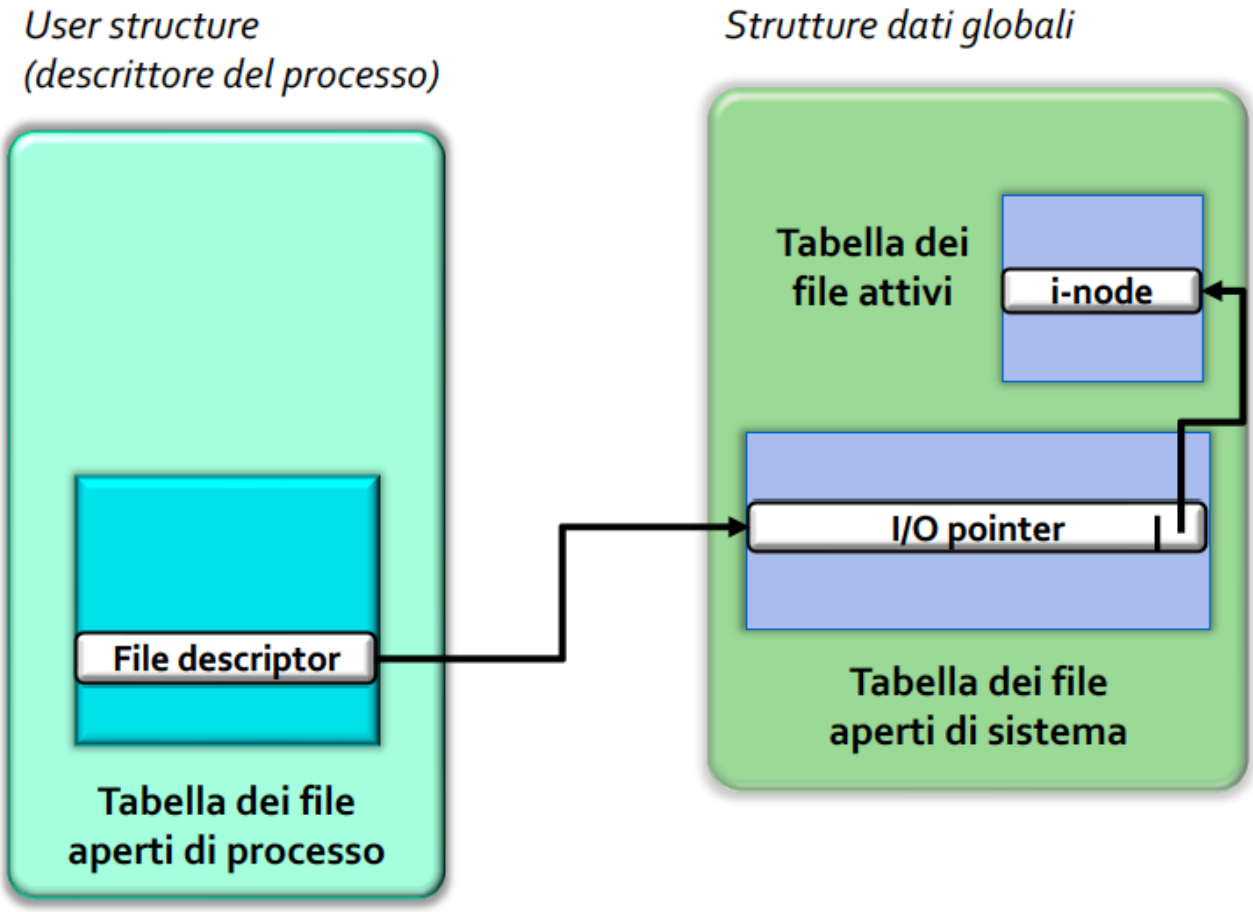
```
        playerIDs[i] = (int*)malloc(sizeof(int));
        *playerIDs[i] = i;
        pthread_create(&players[i], NULL, player, playerIDs[i]);
    }

    while(1) {
            pthread_mutex_lock(&m);
            // Qui ho usato if anziche' while... perche' funzioni devo
            // essere sicuro che al risveglio remaining sia 0.
            if (remaining > 0){
                pthread_cond_wait(&BETTING, &m);
            }

            for (int i = 0; i < NUM_PLAYERS; i++){
                if (x == bet[i]){
                    winner = i;
                    pthread_cond_broadcast(&RESULTS);
                    break;
                }
                if (bet[i]>min && bet[i]<x){
                    min = bet[i];
                }
                if (bet[i]>x && bet[i]<max){
                    max = bet[i];
                }
            }

            if (winner >=0){
                printf("[Main]: Vince il giocatore %d\n", winner);
                pthread_mutex_unlock(&m);
                break;
            }
            printf("[Main]: Nessun vincitore a questo giro. Il numero e'
                   compreso tra %d e %d\n", min,max);
            remaining = NUM_PLAYERS;
            sleep(3);
            pthread_cond_broadcast(&RESULTS);
            pthread_mutex_unlock(&m);
    }

    pthread_exit(NULL);
    return 0;
}
```

ESERCITAZIONE 10: FILE DESCRIPTOR E FORK, COMUNICAZIONE TRA PROCESSI MEDIANTE PIPE

STRUTTURE DATI PER L'ACCESSO AI FILE

Il meccanismo adottato per l'accesso ai file è di tipo sequenziale:

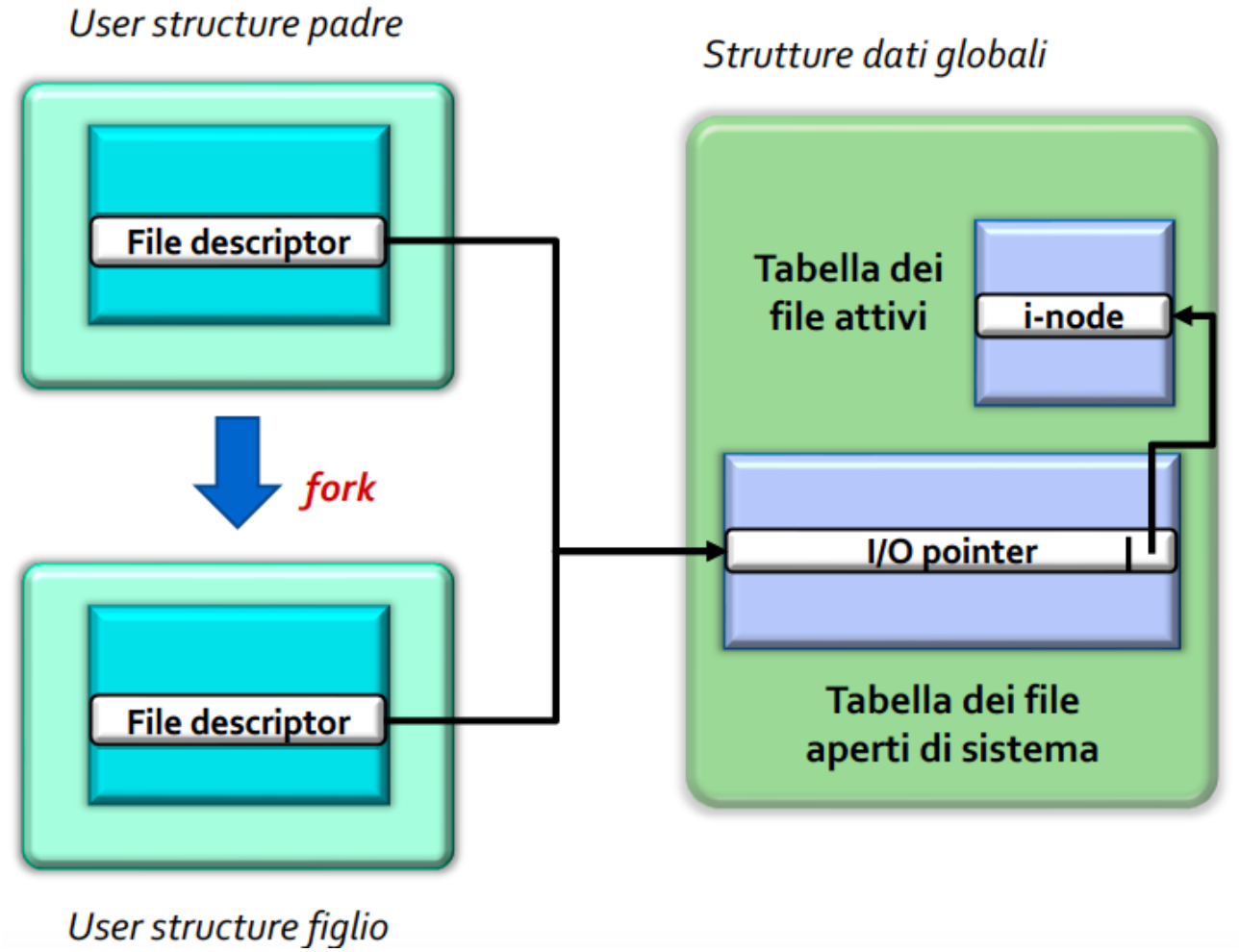
- Ad ogni file aperto è associato un I/O pointer → Riferimento per la lettura/scrittura sequenziale dei file.
- Le operazioni di lettura/scrittura provocano l'avanzamento del riferimento.



Le strutture dati per l'accesso ai file sono gestite dal kernel:

- **Tabella dei File Aperti di Processo:** è nella User Structure del processo e ogni elemento (file descriptor) è un riferimento all'elemento corrispondente nella Tabella dei File Aperti di Sistema.
- **Tabella dei File Aperti di Sistema:** contiene un elemento per ciascun file aperto dal sistema.
    - Se due processi aprono lo stesso file, si hanno due entry separate.
    - Ogni elemento contiene un I/O pointer al file e un riferimento all'i-node del File (che viene tenuto in memoria principale, nella Tabella dei File Attivi).
- **I/O pointer e i-node:** permettono di trovare l'indirizzo fisico in cui effettuare la prossima lettura/scrittura sequenziale.

- **STDIN, STDOUT e STDERR:** descrittori di default. Vengono generati automaticamente al momento dell'esecuzione del programma.



Il processo figlio eredita dal padre una copia della User Structure, quindi anche una copia del file descriptor. In questo caso, i due processi hanno descrittori che puntano allo stesso elemento della Tabella di File di Sistema, e quindi condividono l'I/O pointer nell'accesso sequenziale al file.

PRIMITIVE PER L'ACCESSO AI FILE

APERTURA DI UN FILE DESCRIPTOR

```
int open(const char* path, int flags)
```

`const char* path`	Path del file da "aprire".
`int flags`	Modalità di accesso. Ci sono varie macro definite `<fcntl.h>` per descrivere le possibili modalità. Se compatibili tra di loro, più macro possono essere messe in OR. Esempi: `O_RDONLY`, `O_WRONLY`, `O_RDWR`. Per la lista completa, leggere "man 2 open".

- Ritorna il file descriptor.

Dopo l'apertura, l'I/O pointer viene posizionato all'inizio del file se non è utilizzata la modalità `O_APPEND` (in tal caso, I/O parte dalla fine del file).

LETTURA DA FILE

```
ssize_t read(int fd, void* buf, size_t count)
```

`int fd`	Descrittore del file da cui leggere.
`void* buf`	Puntatore al buffer in cui scrivere i dati letti.
`size_t count`	Numero di byte da leggere (intero positivo).

Ritorna il numero di byte letti (valore negativo in caso di errore).

SCRITTURA SU FILE

```
ssize_t write(int fd, const void* buf, size_t count)
```

`int fd`	Descrittore del file in cui scrivere.
`const void* buf`	Puntatore al buffer da cui leggere i dati da scrivere nel file.
`size_t count`	Numero di byte da scrivere (intero positivo).

Ritorna il numero di byte scritti (valore negativo in caso di errore). Potrebbero essere meno di count, ad esempio, se è terminato lo spazio disponibile.

ESEMPIO

Esempio di lettura testo da file e stampa a video con buffer di dimensioni fisse:

```
#include<fcntl.h>
#include<stdlib.h>
#include<stdio.h>
#define BUF_SIZE 64

int main(int argc, char** argv) {
     if (argc < 2) {
          printf("Usage: %s FILENAME\n", argv[0]);
          exit(-1);
     }
     int fd = open(argv[1], O_RDONLY);
     if (fd < 0) {
          perror("Errore nella open\n");
          exit(-1);
     }
     char buffer[BUF_SIZE];
     ssize_t nread;
     while((nread=read(fd, buffer,BUF_SIZE-1)) > 0){
          buffer[nread] = '\0';
          printf("%s", buffer);
     }
```

```
        close(fd);
        if (nread < 0) {
                perror("Errore nella read\n"); exit(1);
        }
        exit(0);
}
```

COMUNICAZIONE MEDIANTE SCAMBIO DI MESSAGGI – PIPE

I processi possono comunicare sfruttando il meccanismo delle pipe.

- Comunicazione indiretta, senza naming esplicito.
- Realizza il concetto di mailbox nella quale si possono accodare messaggi in modo FIFO.
- La pipe è un canale monodirezionale: ci sono due estremi, uno per la lettura e uno per la scrittura.

Astrazione realizzata in modo omogeneo rispetto alla gestione dei file:

- A ciascun estremo è associato un file descriptor.
- I problemi di sincronizzazione sono risolti dalle primitive read/write.
  - Un lettore si blocca se la pipe è vuota.
  - Uno scrittore si blocca se la pipe è piena.

I figli ereditano gli stessi file descriptor e possono usarli per comunicare con il padre e gli altri figli.

- Per la comunicazione di processi che non si trovano nella stessa gerarchia si utilizzano fifo o socket.

`man pipe`	Pagina del manuale.

CREAZIONE DEI DESCRITTORI DELLA PIPE

```
int pipe(int fd[2])
```

`int fd[2]`	Vettore di due interi: conterrà i descrittori della pipe. Infatti, la funzione salva in `fd[0]` l'estremo (il descrittore) della pipe per la lettura, in `fd[1]` l'estremo da usare per la scrittura.

Ritorna zero se ha successo, -1 altrimenti.

ESERCIZI

ESERCIZIO 1 – FILE DESCRIPTOR

Scrivere un programma in C in cui il main:

- Apre in lettura "leggi.txt" e salva il relativo descrittore in fd.
- Crea un processo figlio con la fork.
- Si sospende con la sleep per 3 secondi.

Il processo figlio legge due caratteri dal file usando il descrittore fd e li stampa a video.

Il padre, dopo la sleep, legge da fd fino alla fine del file e stampa a video quello che ha letto (notare gli effetti della condivisione dello stesso descrittore).

Adesso provare a chiudere il descrittore fd nel processo figlio, e ad aprirlo nuovamente. Come cambiano le cose?

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

#define BUF_SIZE 128

int main() {
    int fd = open("leggi.txt", O_RDONLY);
    char buffer[BUF_SIZE];
    pid_t pid;
    pid = fork();
    if (pid == 0) {
        // Figlio.
        // modifica "close": prima della chiusura di 'fd' il figlio
        // condivide con il padre la stessa entry nella tabella dei file
        // aperti.
        // Quando il figlio chiude e poi riapre 'fd', viene creata una
        // nuova entry nella tabella dei file aperti. Il figlio poi
        // legge i primi due caratteri.
        // A questo punto, il file descriptor 'fd' del padre punta
        // sempre alla entry vecchia, in cui l'I/O pointer punta sempre
        // all'inizio del file. Per questo il processo padre scrive
        // "Hello World!" per intero (e non solo "llo World!" come nel
        // caso in cui il figlio non chiude e riapre 'fd')
        close(fd);
        fd = open("leggi.txt", O_RDONLY);
        ssize_t nread = read(fd, buffer, 2);
        close(fd);
        if (nread != 2) {
            perror("Errore nella read\n");
            exit(1);
        }
        buffer[2] = '\0'; // terminate string.
        printf("Figlio:%s\n", buffer);
        exit(0);
    }
    // Padre.
    sleep(3);
    ssize_t nread;
    while((nread=read(fd, buffer,sizeof(buffer)-1)) > 0){
        buffer[nread] = '\0';
        printf("Padre:%s\n", buffer);
    }
    close(fd);
    if (nread < 0) {
```

```
        perror("Errore nella read\n");
        exit(1);
    }
    exit(0);
}
```

ESERCIZIO 2 – PIPE

Scrivere un programma C in cui il main:

- Crea una pipe.
- Crea un processo figlio.
- Si sospende per 3 secondi con la sleep.
- Scrive "Hello world\n" nella pipe (suggerimento: usare la funzione strlen definita in string.h per trovare la lunghezza della stringa al momento della write).

Il processo figlio legge dalla pipe e stampa il messaggio che ha letto.

- Verificare il comportamento bloccante della read su pipe.

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUF_SIZE 128

int main() {
    int fd[2];
    if (pipe(fd) != 0) {
        perror("Errore nella pipe\n");
        exit(-1);
    }

    char buffer[BUF_SIZE];
    pid_t pid;
    pid = fork();
    if (pid == 0) {
        // Figlio.
        close(fd[1]); // non devo scrivere nella pipe
        ssize_t nread;
        printf("Figlio: sto per leggere...\n");
        while((nread = read(fd[0], buffer, sizeof(buffer)-1)) > 0) {
            buffer[nread] = '\0';
            printf("%s", buffer);
        }
        close(fd[0]);
        if (nread < 0) {
            perror("Errore nella read\n");
            exit(-1);
        }
```

```
        printf("Figlio: ho terminato\n");
        exit(0);
    }
    // Padre.
    close(fd[0]); // non devo leggere dalla pipe.
    printf("Padre: sto per dormire per 3 sec\n");
    sleep(3);
    char msg[] = "Hello world!\n";
    printf("Padre: sono sveglio, sto per scrivere\n");
    ssize_t written = write(fd[1], msg, strlen(msg));
    close(fd[1]);
    if (written != strlen(msg)) {
        perror("Errore nella write!\n");
        exit(-1);
    }
    exit(0);
}
```