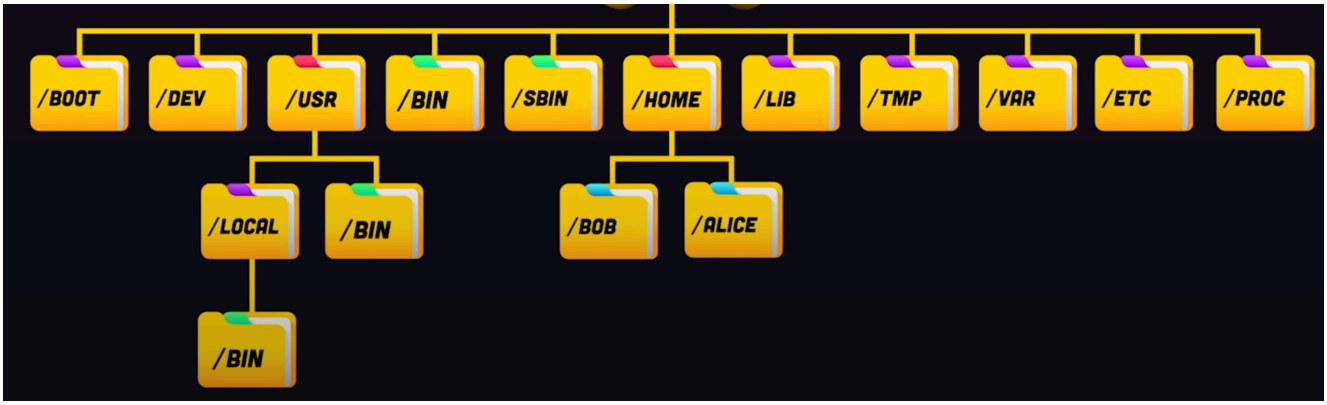
Laboratorio di Sistemi operativi

1. Introduzione



`Control + R` per cercare nella history
`Ctrl + Alt + T` apre un terminale
`Ctrl + Alt + FNx` apre la shell x (7 è quella grafica su debian)

```
exec < stdin.txt
exec > stdout.txt
exec 2> stderr.txt
exec &> both.txt

# > sovrascrive
# >> appende
# si possono combinare tutti

exec1 | exec2   # [Pipe] Collega l'output di exec1 all'input di exec2
```

```
# wildcards: [a-b] [a,c] a?c.txt *.txt
man     # Anche per funzioni del kernel, funzioni di lib C, file di conf
whatis
apropos
echo
pwd
cd
touch   # Aggiorna timestamo di accesso e modifica di un file, o lo crea
mkdir
rmdir
rm -rf
cp src dst  # Anche più src
mv src dst  # Anche più src
sort
head -n
tail -n
less    # Visualizza un po' per volta
ls -lah
cat     # Anche più src
which
```

2. Utenti e gruppi `Parte I`

Ogni utente e gruppo è identificato da un nome e da un ID numerico, ogni utente deve appartenere ad almeno un gruppo

```
id username         # Se non specificato quello corrente
groups username
!adduser username   # Lo inserisce anche in un gruppo con lo stesso nome
!deluser username
!chown username file
chgrp groupname file    # L'utente chiamante deve essere proprietario
                        #  del file ed appartenere a groupname, oppure
                        #  essere root
su user # Accede al terminale di user, o root se non specificato
exit    # Torna alla shell precedente
sudo -u username comando
    # Lancia un comando per conto di un altro utente
    #  (root se non specificato), chi esegue sudo deve
    #  fare parte del gruppo sudo(ers) e deve
    #  autenticarsi con password
```

> ! prima di un comando significa che richiede i permessi di root

Ad ogni file e directory è associato un utente proprietario ed un gruppo proprietario, di conseguenza, dal punto di vista di un file, gli utenti si dividono in:

- Il proprietario
- Gli appartenenti al gruppo
- Tutti gli altri

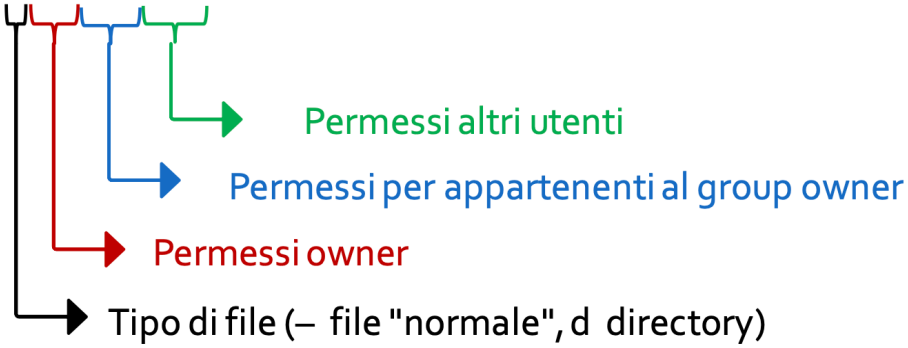
Per ognuna di queste categorie è possibile specificare i seguenti permessi:

- r: read
- w: write
- x: execute

> Per la cancellazione e la creazione di un file non valgono i permessi di scrittura su tale file ma quelli sulla directory in cui voglio compiere le operazioni

> Il permesso di esecuzione su una directory impedisce di attraversarla in qualunque modo, mentre il permesso di lettura impedisce solamente di leggerne l'elenco dei file presenti, ma non i singoli file

```
-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 ottale rappresentano i bit per le tre classi di utenti (es: 777 garantisce tutto a tutti)

```
chmod ottale file
chmod [who][how][which] file
#   [who]:
#       u -> user (owner)
#       g -> group
#       o -> others
#   [how]:
#       + -> aggiunge
#       - -> toglie
#       = -> imposta
#   [which]: r, w, x

# L'utente chiamante deve essere proprietario del file, oppure root
# L'opzione -R applica lo stesso permesso a tutte
#  le sottodirectory ed i file in esse contenuti
```

Normalmente, quando un utente esegue un programma, il processo creato acquisice gli stessi privilegi dell'utente chiamante, è tuttavia possibile fare in modo che certi programmi (file) eseguano sempre con i privilegi dell'utente o del gruppo proprietari, è proprio questo il significato dei bit di protezione **SUID** e **SGID**

Questi permessi hanno senso solo su un file eseguibili, nella visualizzazione simbolica con `ls` vengono dunque rappresentati come delle `s` al posto delle `x` rispettivamente nei

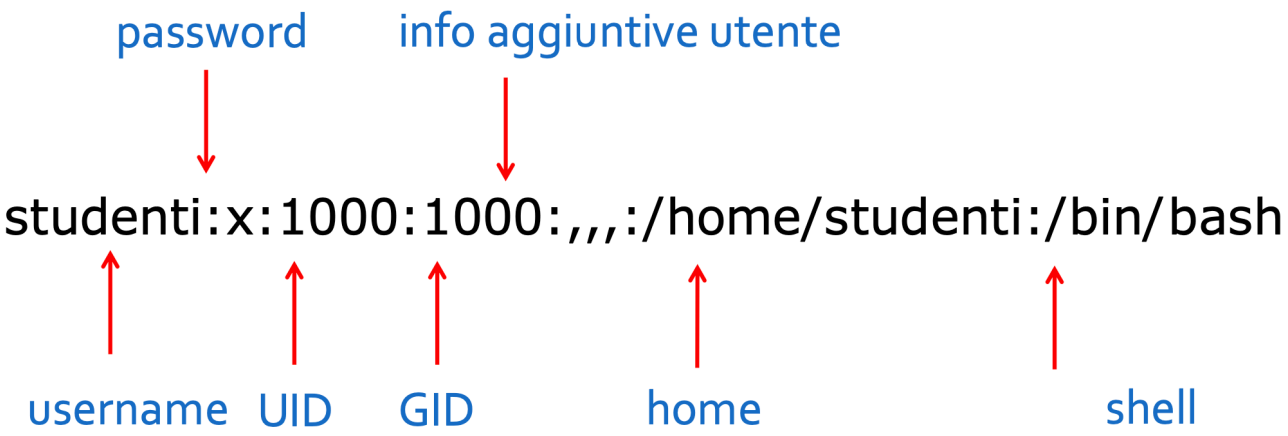
permessi del proprietario (SUID) e del gruppo (SGID)

Per quanto riguarda la rappresentazione ottale si utilizza un'ulteriore cifra (i bit di protezioni sono quindi 12) posta in testa, ad esempio `6777` ha sia attivo SUID che SGID, mentre `4777` ha attivo solo SUID

```
passwd  # Cambia password dell'utente, sfrutta il permesso SUID
```

3. Utenti e gruppi `Parte II`

```
# Informazioni pubbliche riguardo agli utenti
/ect/passwd
# Per modificarlo bisogna eseguire il comando "vipw"
```

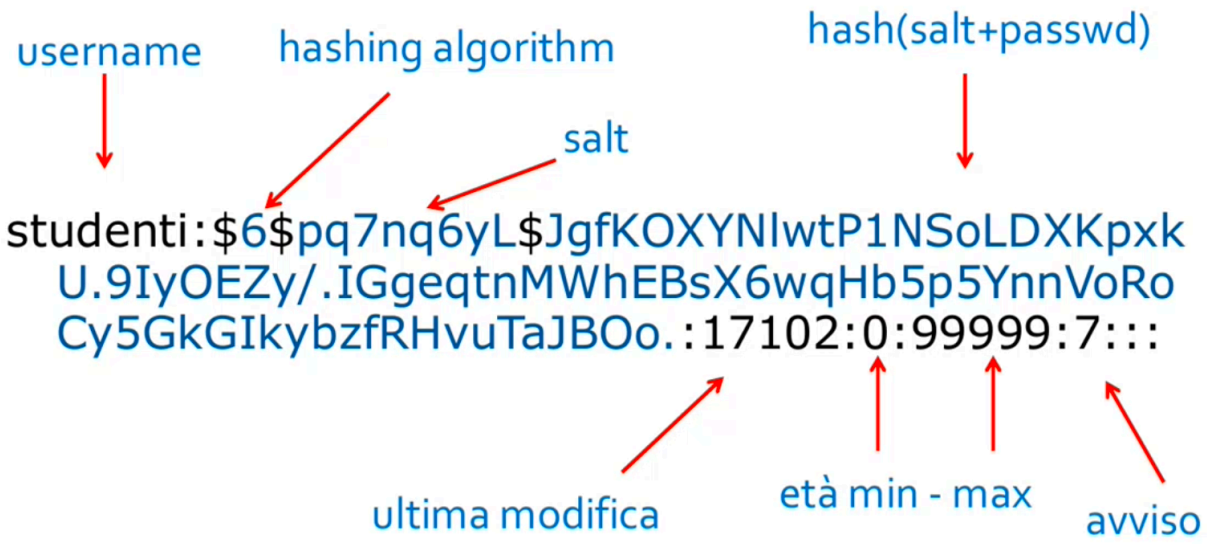


> Un tempo la password era memorizzata qui, *x* vuol dire che la password è stata spostata in `/ect/shadow`

> GID è l'ID numerico del gruppo primario, ogni utente ne deve avere uno, oltre ad un massimo di 15 gruppi secondari

> Shell può essere *false* o *nologin* per indicare che con tali utenti non è possibile autenticarsi (sono ad esempio gli utenti che eseguono i daemon, ossia processi in background, per evitare di farli eseguire tutti da root)

```
# Informazioni private riguardo agli utenti
/ect/shadow # RW solamente dall'utente root
# Per modificarlo bisogna eseguire il comando "!vipw -s"
```

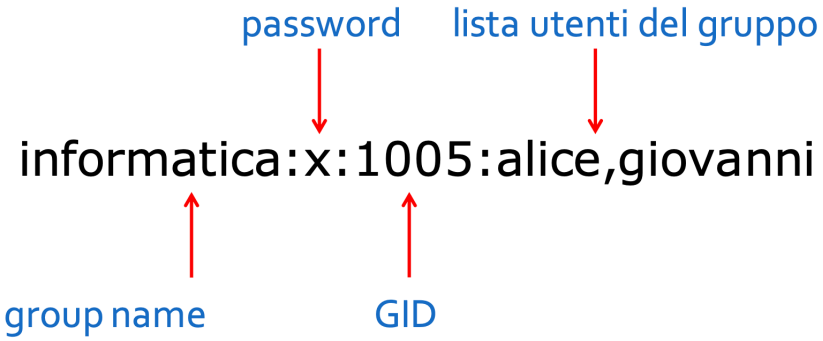


> Min, max e avviso si riferiscono ai giorni di durata minima, massima e preavviso della password

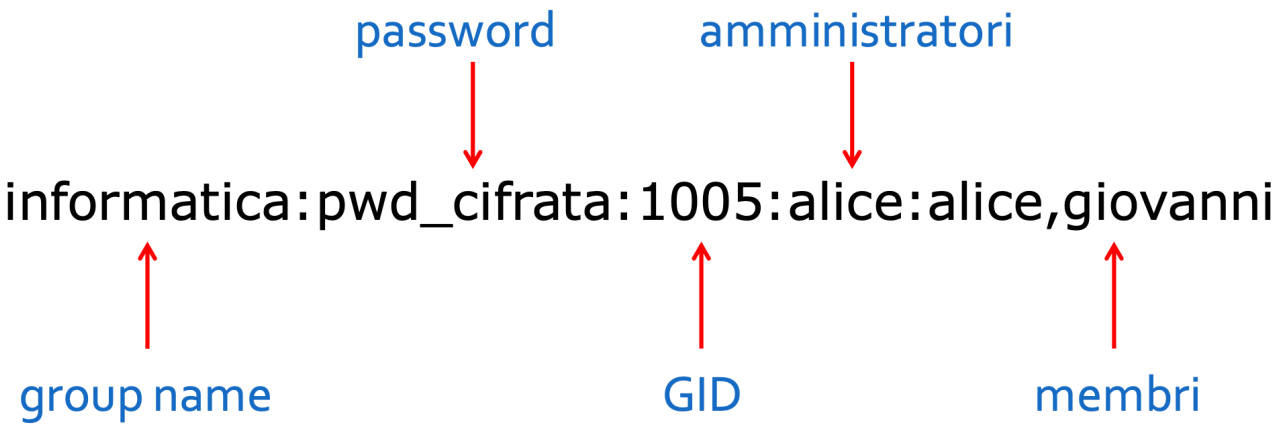
---

```
# Comandi per la gestione dei gruppi
!addgroup gruppo
!delgroup gruppo

gpasswd -a utente gruppo    # add
gpasswd -d utente gruppo    # delete
gpasswd -M utente_1, ..., utente_n gruppo   # definisce i membri
# È necessario essere amministratori del gruppo o root

!gpasswd -A utente_1, ..., utente_n gruppo  # definisce gli admin,
                                            # devono già essere membri

gpasswd gruppo      # imposta la password del gruppo
gpasswd -r gruppo   # rimuove la password del gruppo
# È necessario essere amministratori del gruppo o root

# Se la password di gruppo è impostata, un utente esterno può assumere
#  temporaneamente il gruppo come primario conoscendone la password
# Se l'utente appartiene già al gruppo può assumerlo temporaneamente
#  come primario senza bisogno della password
newgrp gruppo
usermod -g gruppo utente    # Modifica permanente
```

```
# Informazioni pubbliche riguardo ai gruppi
/ect/group
# Per modificarlo bisogna eseguire il comando "!vigr"
```




```
# Informazioni private riguardo ai gruppi
!/ect/gshadow
# Per modificarlo bisogna eseguire il comando "!vigr -s"
```



4. Gestione dei file

Find

Ricerca file sul filesystem **tramite metadati** (nome, tipo, owner, timestamp, ...), ed effettua azioni sui file trovati

```
find path_1 ... path_n espressione
# I path, separati da spazi, indicano le cartelle in cui cercare i file
# Le espressioni sono formate da:
# - Test sui metadati: [True o False]
#       -name "pattern"
#           pattern può contenere metadati, gli
#           apici evitano che la shell li espanda
#       -type [dfl]
#           d=directory, f=file, l=symlink
#       -size [+-]n[ckMG]
#       -user utente
#       -group gruppo
#       -perm [-/]permessi
#           permessi: esattamente
#           -permessi: almeno tutti
#           /permessi: almeno uno
#          i permessi sono in notazione ottale o simbolica
# - Azioni: [True se hanno successo]
#       -delete
#       -exec comando \;
#           viene eseguito ogni volta che un file supera tutti i test
#           precedenti, con {} si può indicare il nome (relativo) del
#           file attualmente processato, a partire dal percorso dove ho
#           chiamato find, in alternativa con -execdir il comando viene
#           eseguito dal path del file trovato
# - Opzioni globali [True]
# - Opzioni posizionali [True]
#       Influenzano solo l'esecuzione delle espressioni che seguono
#
# Le espressioni sono collegati da operatori logici, di default AND -a,
#  ma è anche possibile usare OR -o oppure NOT !
```

Locate

Cerca uno o più file all'interno di un database (indice) aggiornato periodicamente: è più veloce di find ma non è standard, in più non permette di definire test ed azioni e ritorna risultati che potrebbero non essere aggiornati

> L'indice può essere aggiornato tramite `updatedb`

```
locate options file1...
```

GREP

General Regular Expression Print è un comando che 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_1" (–e "modello_2"...) file_1 (file_2...)

# Modello è un'espressione regolare:
#   ^ –> inizio riga
#   $ –> fine riga
#   [0–9]
#   . –> qualisasi carattere
#   * –> zero o più ripetizioni del token precedente
```

Opzione	Significato
-i	Ignora le distinzioni fra 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

Archiviazione e compressione

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

```
tar modalità file_1 ... file_n

# Esempio di modalità:
#   c crea
#   r aggiungi file
#   t elenca i file presenti
#   z comprimi con gzip (gunzip, estensione .gz)
#   j comprimi con bzip2 (bunzip2, estensione .bz2)
#   x estrai
#   f specifica il nome dell'archivio
```

5. Processi `Parte I`

System calls per gestire i processi

```
pid_t fork()
/*
Crea un processo con lo stesso segmento codice, ma un diverso segmento
 dati (una copia di quello del padre), questo evidenzia come in Unix i
 processi comunichino attraverso messaggi e non memoria condivisa

Restituisce il PID del filgio al padre, restituisce 0 al figlio
*/
pid_t getpid()  // Ritorna il PID del processo chiamante
pid_t getppid() // Ritorna il PID del padre
```

```
void exit(int status)
/*
Provoca la terminazione volontaria del processo (una terminazione
 involontaria era ad esempio un accesso in una zona di memoria vietata
 o la ricezione segnale di terminazione)

Lo stato di terminazione viene ritornato al padre attraverso il
 meccanismo descritto dalla primitiva successiva
*/
```

```c
pid_t wait(int *status)
/*
Recupera lo stato di terminazione di un processo figlio , in particolare
 se almeno un figlio ha già terminato ritorna immediatamente il suo
 stato di terminazione, altrimenti blocca il padre, infine se il padre
 non ha più figli in esecuzione ritorna un valore negativo

L'implementazione di status non è standard, sono definite in <sys/wait>
 le seguenti macro per interpretarne il contenuto:
*/
WIFEXITED(status)   // Controlla se la terminazoione è stata volontaria
WEXITSTATUS(status) // Ritorna lo stato di terminazione
```

```c
int exec(...)
/*
È una famiglia di funzioni che permettono ad un processo di sostituire
 il proprio codice (ed i propri dati)
La chiamata è senza ritorno se ha successo

Sono disponibili due versioni principali di exec:
*/

// l: gli argomenti sono una serie di stringhe
int execl(char *path, char *arg0, ..., char *argn, (char*) 0)

// v: gli argomenti sono un vettore di stringhe
int execv(char *path, char *argv[])

/*
In ogni caso:
 - path è il file da usare come nuovo codice
 - arg0, o argv[0] è il nome del comando che si sta eseguendo,
    (in teoria dovrebbe coincidere con path)
*/
```

6. Processi `Parte II`

I segnali sono interruzioni software, utilizzati tra processi ma anche tra processi e kernel per scambiare informazioni ai fini della sincronizzazione

Le varie tipologie di segnale e le primitive per gestirli sono dichiarate in `signal.h` e sono le seguenti:

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 di chiusura
SIGTERM	15	Terminazione del programma.
SIGUSR1	10	Definito dall'utente. Default: termina processo.
SIGUSR2	12	Definito dall'utente. Default: termina processo.
SIGSEGV	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.

```c
/*
Quando un processo riceve un segnale possono succedere tre cose:
- Ignora il segnale (non valido per tutti)
- Esegue un handler di default
- Esegue un handler ridefinito dal programmatore

Per realizzare quest'ultima opzione esiste la primitiva signal:
*/

typedef void (*sighandler_t)(int)
/*
Praticamente sighandler_t è un puntatore a funzione void che riceve
 un parametro intero (al momento della ricezione del segnale conterrà
 il codice del segnale da gestire)
*/

sighandler_t signal(int sig, sighandler_t handler)
/*
Definisce la funzione handler per il segnale sig, handler può anche
valere SIG_IGN (ignora il segnale) o SIG_DFL (azione di default)

Restituisce un puntatore al precedente handler del segnale (il
 vecchio codice praticamente), oppure SIG_ERR in caso di errore
*/
```

> In caso di fork il padre passa al figlio le informazioni sulla gestione dei segnali, in caso di exec però queste si perdono

```c
int kill(pid_t pid, int sig)
/*
Invia il segnale SIG:
- Se pid > 0 al processo PID
- Se pid == 0 a tutti i processi dello stesso gruppo del chiamante
- Se pid == -1 a tutti i processi a cui può inviare segnali
- Se pid < -1 all gruppo di processi identificato da -pid

Ritorna 0 in caso di successo
*/
```

```c
int pause()
/*
Mette in pausa il processo in attesa di un segnale qualsiasi, ritorna
 -1 se riceve un segnale e l'handler non termina il processo
*/
```

```c
unsigned int sleep(unsigned int seconds)
/*
Il processo si sospende fino a quando non arriva un segnale che non
 viene ignorato (arriva sicuramente un SIGALARM dopo i secondi
 specificati)

Ritorna il tempo che mancava all'arrivo di SIGALARM (0 se va tutto bene)
*/

unsigned int alarm(unsigned int seconds)
/*
Imposta un SIGALARM allo scadere dei secondi specificati, senza
 sospendersi nel mentre

Ritorna zero se non c'era un allarme già impostato, altrimenti ritorna
 il numero di secondi che mancavano tale allarme, e lo elimina (ogni
 processo può averne uno solo)
*/
```

La primitiva `kill` può anche essere invocata da terminale:

```
kill -SEGNALE pid_1, ..., pid_n
# Il segnale di default è SIGTERM, in ogni caso bisogna essere
#  root oppure proprietari dei processi sui quali si invoca
```

Un'altro comando utile è `ps`:

```
ps opzioni
# Mostra uno snapshot dei processi in esecuzione:
#   -u utente: mostra solo i processi dell'utente specificato
#   a: mostra i processi di tutti gli utenti
#   u: fornisce una formattazione utile all'analisi delle risorse
#   x: mostra anche i processi non spawnati da terminale
```

7. Processi `Parte III`

I sistemi unix prevedono un init system (in debian **systemd**, con `PID=1`), ossia un processo, mandato in esecuzione dal kernel durante il boot, da cui discendono tutti i processi del sistema (comando `postree` per vederne l'albero)

> Se il padre di un processo termina prima di lui (dunque senza fare wait), allora questo viene "adottato" dal processo init!

Oltre al `PID` e al `PPID` ci sono anche altri identificatori legati ad un particolare processo:

- `PGID`: Process Group ID
- `RUID`, `RGID`: Real User/Group ID, ossia il `PID` e `GID` dell'utente che ha mandato in esecuzione il processo
- `EUID`, `EGID`: Effective User/Group ID, ossia `PID` e `GID` dell'utente o del gruppo che rappresenta attualmente i diritti di accesso del processo (ad esempio se il processo ha eseguito un file con bit di protezione `SUID` o `GUID`)

> Un processo non root può inviare segnali ad un altro processo solo se il suo RUID o il suo EUID equivale al RUID del destinatario

> `PGID` suggerisce che i processi siano organizzati in gruppi, quando viene eseguito un processo da terminale gli viene infatti associato un nuovo `PGID`, ai quali apparterranno tutti i suoi eventuali figli (anche in seguito ad exec)

Lo scheduler dei sistemi Unix ha due livelli di priorità principali, *real-time* o *normale*, la priorità dei processi *normali* può essere in parte controllata mediante il concetto di **niceness**: ad ogni processo è associata una niceness da -20 a 19, più alta è meno priorità ha

Un utente può solo aumentare la niceness dei propri processi, mentre root può anche diminuirla, per farlo si può utilizzare la primitiva `nice`, per la quale sono anche definiti comandi di shell:

```
nice -n valore comando
# Manda in esecuzione il comando con la niceness specificata

renice valore PID
# Modifica la niceness del processo PID secondo quanto detto prima
```

La shell Unix mette a disposizione le alcune operazioni di **job control**, ossia la possibilità di sospendere e riattivare processi o gruppi di processi, la shell infatti associa un **job id** ad ogni comando eseguito, visibile con `jobs`

Un job in esecuzione può essere in:

- **foreground**: ha il controllo di *stdin*, *stdout* e *stderr*, l'utente può posizionare un job arbitrario in foreground tramite il comando `fg %JOB`, oppure fermarlo con `CONTROL+Z` (SIGSTP)
- **background**: se viene eseguito con il carattere `&` in fondo al comando, non ha chiaramente accessi agli stream standard, l'utente può posizionare un job arbitrario in background tramite il comando `bg %JOB`

```
# Si può inoltre usare il comando kill anche con i job, ad esempio:
kill -n SIG %JOB
```

Infine c'è da notare che quando il terminale viene chiuso, i job in esecuzione ricevono il segnale `SIGHUP` e, di default, terminano, per fare in modo che questo non avvenga posso utilizzare due strumenti:

```
nohup comando
# Rende il job eseguito immune a SIGHUP, lo stdin è semplicemente
#  un eof e lo stdout viene rediretto su un file nohup.out
```

```
disown %JOB
# Rende il job già in esecuzione immune a SIGHUP, è buona norma
#  eseguirlo dopo che il processo abbia utilizzato lo stdin e fare
#  in modo che lo stdout venga rediretto
```

8. Thread POSIX `Parte I`

I thread beneficiano di tutti i vantaggi che abbiamo già visto al solo costo di scrivere codice thread safe, linux tra l'altro supporta i thread a livello di kernel, rendendo possibile esecuzione parallela su sistemi multiprocessore, di fatto anche un "normale processo" è un singolo thread, senza figli

> Per scrivere programmi multithread in linux è possibile utilizzare la libreria pthreads `<pthread.h>` , serve però specificare a gcc che la stiamo utilizzando con `-lpthread -std=c99`

```
/*
Ogni thread ha un ID di tipo pthread_t, che però è un tipo opaco, non
 ha dunque senso stamparlo a video ma va maneggiato con particolari
 funzioni:
*/
pthread_t pthread_self()
int pthread_equals(pthread_t, pthread_t)
```

```
int pthread_create(
    // Puntatore a dove verrà salvato il TID del thread creato
    pthread_t* thread,
    // Attributi del thread, NULL per usare quelli di default
    const pthread_attr_t* attr,
    // Codice del nuovo thread, parametro e ritorno sono puntatori
    void* (*start_routine)(void *),
    // Puntatore agli argomenti passati alla routine
    void* arg
)
// Ritorna 0 in caso di successo
```

```
void pthread_exit(void* retval)
/*
L'esecuzione del thread termina e il sistema libera le risorse allocate
Se un thread padre termina prima dei thread figli non chiamando la
 pthread_exit, allora i thread figli terminano immediatamente
*/
```

```
int pthread_join(pthread_t thread, void** retval)
/*
Attende il completamento del thread specificato, retval è un puntatore
 al puntatore dove verrà salvato l'indirizzo restituito dal thread con
 la pthread_exit (se NULL viene ignorato)
Ritorna 0 in caso di successo (un errore può essere ad esempio un
 altro thread che fa join sullo stesso thread)
*/
```

Per scrivere codice thread safe serve mutua esclusione, la libreria `pthreads` mette a disposizione **mutex**, astrazione analoga a quella di semaforo binario

```
pthread_mutex_t
// Tipo mutex

int pthread_mutex_init(
    // Puntatore al mutex da inizializzare
    pthread_mutex_t* M,
    // Attributi di init, se NULL quelli di default (mutex libero)
    const pthread_mutexattr_t* mattr
)

int pthread_mutex_lock(pthread_mutex_t* M)
// È equivalente alla wait()
int pthread_mutex_unlock(pthread_mutex_t* M)
// È equivalente alla signal()
// Entrambe ritornano 0 in caso di successo
```

9. Thread POSIX `Parte II`

Il meccanismo dei mutex illustrato precedentemente risolve il problema della sincronizzazione indiretta, ossia della mutua esclusione (non mi interessa l'ordine), per eseguire operazioni di sincronizzazzione diretta più avanzate la libreria `pthreads` mette a disposizione le cosiddette **variabili condizione**, ossia condizioni sulle quali i thread posso sospendersi (in sostanza dei semafori non binari)

```
pthread_cond_t
// Tipo variabile condizione

int pthread_cond_init(
    // Puntatore alla variabile condizione da inizializzare
    pthread_cond_t* C,
    // Attributi di init, se NULL quelli di default
    pthread_cond_attr_t* attr
)
```

La variabile condizione chiaramente non è che si verifica da sola e, dato che la primitiva signal è di tipo *signal and continue* (altri thread potrebbero inserirsi dopo la wait e alterare la condizione), lo schema generico di utilizzo della wait è il seguente:

```
while (condizione logica)
    wait(variable condizione)
```

La verifica della condizione logica deve però essere eseguita in mutua esclusione, la primitiva di wait offerta dalla libreria `pthreads` tiene conto di questo aspetto e permette di **associare un mutex ad una variabile condizione**, che si comporta come segue:

- *Prima del while bisogna chiamare esplicitamente l'operazione di lock sul mutex*
- Il mutex viene rilasciato automaticamente 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
- *Dopo aver attraversato la sezione critica bisogna chiamare esplicitamente l'operazione di unlock sul mutex*

```c
int pthread_cond_wait(pthread_cond_t* C, pthread_mutex_t* M)

int pthread_cond_signal(pthread_cond_t* C)
// Almeno un thread viene risvegliato
int pthread_cond_broadcast(pthread_cond_t* C)
// Tutti i thread vengono risvegliati
```

10. File system e PIPE

File system

Quando viene creato un file, nella directory in cui risiede viene aggiunta la voce del tipo `<Nome file, i-number>`, nota come **hard link**, un file può avere sia più hard link che più **soft link**, questi ultimi non sono altro che nomi alternativi per hard link

```
# Crea un hard link
ln target linkname
# Crea un soft link
ln -s target linkname
```

> Dato che in caso di fork i due processi hanno la stessa user structure copiata, allora hanno anche lo stesso I/O pointer

```
int open(const char* path, int flags)
/*
In caso di successo restituisce l'indice della tabella dei file
 descriptor, altrimenti -1
Esempi di flag, definiti in <fcntl.h>, possono essere:
 - O_RDONLY
 - O_WRONLY
 - O_RDWR
 - O_APPEND
*/
```

```
ssize_t read(int fd, void* buf, size_t count)
/*
In caso di successo sono stati scritti sul vettore buf tanti byte di
 fd, a partire dall'I/O pointer, quanto è il valore di ritorno (al
 più count)
*/
```

```
ssize_t write(int fd, const void* buf, size_t count)
/*
In caso di successo sono stati scritti sul file fd, a partire dall'I/O
 pointer, tanti byte di buf quanto è il valore di ritorno (al più count)
*/
```

```
int close(int fd)
/*
In caso di successo il file viene salvato sul disco, le tabelle dei
 file vengono pulite e restituisce 0, altrimenti < 0
*/
```

```
int lseek(...)
/*
Forza l'accesso diretto ad un file, in altre parole sposta l'I/O pointer
 a piacimento
*/
```

> Esiste la versione bufferizzata delle primitive viste in precedenza: `fopen`, `fread`, `fwrite`, `fclose`, e anche quella formattata: `fscanf`, `fprintf`

PIPE

Una pipe è un **canale monodirezionale** (un processo legge sempre ed uno scrive sempre) che realizza il concetto di mailbox (FIFO)

L'astrazione è realizzata in modo omogeneo rispetto alla gestione dei file: a ciascun estremo è associato un file descriptor e i problemi di sincronizzazione sono risolti dalle primitive read/write

> Dato che i figli ereditano gli stessi file descriptor il meccanismo delle pipe è utile per comunicare tra processi della stessa gerarchia, altrimenti si utilizzano altri strumenti, ad esempio i socket

```c
int pipe(int fd[2])
/*
fd è un vettore di due interi: fd[0] è il file descriptor dell'estremo
 ricevitore, fd[1] è il file descriptor del trasmettitore (creati
 entrambi dalla funzione)
Ritorna 0 se ha successo, -1 altrimenti
*/
```

11. Pilotare applicazioni

Primitiva dup2

```
int dup2(int target, int newfd)
/*
Duplica il file descriptor target in newfd, chiudendolo quest'ultimo se
 era già aperto
Ritorno un valore negativo in caso di errore
*/
```

Sapendo che i file descriptor degli stream di default (STDIN, STDOUT e STDERR) possono essere acceduti dalle macro `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`, allora questo, insieme alla primitiva `dup2`, permette di redirigere il flusso dei dati standard verso altri file descriptor

---

Pilotare applicazioni

Il comportamento menzionato sopra può essere particolarmente utile, con l'aggiunta del meccanismo delle pipe, per pilotare con un programma C un applicazione che interagisce tramite STDIN e STDOUT di cui non abbiamo il sorgente:

1. Creare due pipe
2. Forkare il processo
3. Redirigere STDIN e STDOUT del figlio nelle pipe
4. Trasformare il figlio nell’applicazione con exec