**INTRODUZIONE A UNIX** *24-09-24*

**FILE SYSTEM**

In UNIX ogni documento, directory (cartella), dispositivo di I/O… è accessibile attraverso il file system.   
Questo è strutturato a forma di albero, dove ‘/’ è la root (radice). Per indicare un elemento all’interno   
del file system si usa il suo path (percorso), che può essere espresso in due modi:

* **Path assoluto**: si esprime tutto il path partendo dalla root.

*es. /home/tizio/Documents/file.txt* è il path assoluto di *file.txt*;

* **Path relativo**: si esprime il path partendo dalla directory corrente.

*es.* se la directory corrente è */home/tizio, Documents/file.txt* è il path relativo di *file.txt*.

I path più importanti sono:

* **/home**: qui si hanno le home di ogni utente (UNIX è un sistema operativo multiutente,   
  e ogni utente ha una sua home);
* **/sbin**: qui si hanno i programmi di sistema;
* **/etc**: qui si hanno file di configurazione;
* **/media**: qui si può accedere ai supporti rimovibili, che vengono “montati” sul file system.

Esistono poi dei caratteri speciali:

* ***~*** : indica la home dell’utente corrente (e cioè, l’utente che attualmente sta usando il sistema operativo);
* ***.*** : indica la directory corrente;
* ***..*** : indica la directory padre della directory corrente.

**SHELL**

La **shell (terminale)** è un programma che consente di richiedere informazioni e servizi   
al sistema operativo. La shell continuamente:

1. Mostra un *prompt*, e cioè una stringa con scritte varie informazioni + un campo dove l’utente può scrivere un comando;
2. Attende che l’utente scriva un comando terminato con “*invio*”;
3. Esegue il comando:
   * Se previsto, mostra a video l’output del comando;
   * In caso di errore (*es.* se l’utente scrive che un comando che non esiste o se la shell non riesce a portare a termine il comando digitato dall’utente), lo segnala a video;
4. Torna al punto 1.

Un esempio di prompt è questo:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

In questo caso:

* *amy* è l’username dell’utente corrente;
* *tardis* è il nome dell’host;
* *~/Documents* è la directory corrente;
* Il carattere successivo indica il tipo dell’utente corrente: ‘*$’* se è un utente normale; ‘*#’* se è un utente root (ha maggiori privilegi rispetto ad un utente normale).

**COMANDI BASE SHELL**

Vediamo ora alcuni comandi base della shell. Questi comandi possono essere arricchiti con le cosiddette **opzioni,** precedute da ‘-‘. Le opzioni sono cumulabili:

* **cd *directory***: cambia la directory corrente in *directory*. Se non si indica nulla,   
  cambia la directory corrente nella home dell’utente corrente.

*es.*

Immagine che contiene schermata, testo, simbolo, Carattere

Descrizione generata automaticamente

La directory corrente è */var*:

* + Se digito “cd */lib”*, avendo indicato un path assoluto, passerò a *lib* di sinistra;
  + Se invece digito “cd *lib”*, avendo indicato un path relativo, passerò a *lib* di destra;
  + Se invece digito “cd *..”*, passerò alla directory padre, e quindi a */*;
  + Se invece digito “cd *~”* (oppure semplicemente “cd*”*), passerò alla home   
    dell’utente corrente.
* **pwd:** stampa il path assoluto della directory corrente.

*es.* se la directory corrente è */var*, digitando “pwd” verrà stampato */var*.

* **ls *directory*:** elenca il contenuto di *directory*. Se non si indica *directory*, elenca il contenuto della directory corrente. I file e le directory vengono mostrati con colore diverso (in questo modo si può capire quali sono file e quali sono directory).

Alcune opzioni utili sono:

Long

* + -l: mostra per ogni file/directory alcuni dettagli (dimensioni, data ultima modifica…);
  + -a: elenca anche i file/directory nascosti.

All

* **man *nome*:** permette di accedere al manuale di UNIX e ottenere una descrizione esaustiva   
  di *nome*. Questo *nome* può essere il nome di un comando della shell, ma può anche altro:   
  il manuale di UNIX è infatti diviso in varie sezioni e di queste le più importanti sono:
  + La 1°, relativa ai comandi della shell;
  + La 2°, relativa alle funzioni del kernel;
  + La 3°, relativa alle funzioni delle librerie C;
  + La 5°, relativa ai file di configurazione.

Nel caso in cui ci potessero essere ambiguità, occorre specificare la sezione del manuale cui si vuole accedere subito dopo man (es. *printf* è il nome sia di un comando della shell, che di una funzione in C. Se si scrive semplicemente “manprintf*”*, la shell mostra la prima descrizione di *printf* che trova sul manuale, e quindi la descrizione del comando della shell che ha tale nome. Per avere la descrizione della funzione della libreria C printf, occorre specificare la sezione del manuale interessata, scrivendo “man 3 printf*”*).

* **mkdir *dir*:** crea una nuova directory *dir*.

*es.* digitando *“*mkdir *~/Scrivania/ciao”*, creerò una nuova directory di nome *ciao*   
al path indicato.

* **rmdir *dir*:** rimuove la directory *dir*, solo se vuota.

*es.* digitando “rmdir *~/Scrivania/ciao*”, rimuoverò la directory di nome *ciao* che si trova   
al path indicato, solo se vuota.

* **cp *src1 src2 … srcN dst:*** copia i file/directory *src1, src2, … srcN*in *dst*.  
  Se si copia un singolo file/directory, è possibile anche rinominare la copia, specificando   
  alla fine di *dst* il nuovo nome della copia (o, se si sta copiando il file/directory   
  nella directory corrente, semplicemente scrivendo come *dst* il nuovo nome della copia).

*es.* digitando “cp *~/Scrivania/ciao* *~/Documents/buongiorno*”, copierò il file di nome *ciao*   
che si trova al path indicato nella directory *Documents* che trova al path indicato,   
e la copia avrà come nome *buongiorno*.

* **mv *src1 src2 … srcN* *dst:*** sposta i file/directory *src1, src2, … srcN*in *dst*.   
  Se si sposta un singolo file/directory, è possibile anche rinominare il file/directory spostato, specificando alla fine di *dst* il nuovo nome (o, se si sta spostando il file/directory   
  nella directory corrente, semplicemente scrivendo come *dst* il nuovo nome del file/directory).

*es1* digitando “mv *~/Scrivania/ciao ~/Documents/buongiorno*”, sposterò la directory di nome *ciao* che si trova al path indicato nella directory *Documents* che trova al path indicato,   
e la directory verrà rinominata in *buongiorno*.

*es2* se mi trovo nella directory *~/Scrivania*, digitando “mv *ciao buongiorno” rinominerò* la directory di nome *ciao* in *buongiorno* (corrisponde a spostare il file dalla directory corrente   
alla directory corrente cambiandogli il nome, e quindi semplicemente lo sto rinominando).

* **touch *file:*** aggiorna il timestamp di accesso di *file*, impostando il timestamp corrente.   
  Se *file* non esiste, viene creato.
* **rm *file1 file2 … fileN directory1, directory2 … directoryM*:** rimuove *file1, file2, … fileN, directory1, directory2 … directoryM*. Le directory vengono eliminate (insieme al loro contenuto) solo se si aggiunge l’opzione “-r”.

Recursive

* **cat *file1 file2 … fileN*:** stampa la concatenazione del contenuto di *file1, file2, … fileN*.   
  Se *file1 file2, … fileN* non sono indicati, il contenuto da stampare viene letto dallo stdin.
* **less *file*:** stampa il contenuto di *file* un po’ alla volta, con la possibilità di muoversi all’interno   
  di esso in modo interattivo. Se *file* non è indicato, il contenuto da stampare   
  viene letto dallo stdin.
* **head/tail *file:*** stampa le prime/ultime linee di testo di *file*. Se *file* non è indicato, le linee da stampare vengono lette dallo stdin. Di default stampa 10 linee di testo,   
  ma con l’opzione “-n *numero”* si può modificare il numero di linee da stampare.

**COMANDI PER L’I/O**

Ogni processo ha tre canali di input/output standard (cioè, tre canali di comunicazione che collegano tale processo con l’ambiente operativo in cui viene eseguito):

* stdin: canale da cui il processo può leggere l’input;
* stdout: canale su cui il processo manda l’output;
* stderr: canale su cui il processo manda messaggi di errore;

Di default, lo stdin è collegato alla tastiera, mentre lo stdout e lo stderr sono collegati alla shell   
(cioè, un processo può leggere l’input da tastiera, e mostra l’output e i messaggi di errore sulla shell). E’ possibile però ridirezionare i canali verso dei file (e cioè, fare in modo un processo possa acquisire l’input da file, oppure che l’output e/o i messaggi di errore di un processo vengano stampati su file).   
Nel caso dei comandi della shell, si fa in questo modo:

* ***comando* > *file* :** ridireziona lo stdout di *comando* a *file*, sovrascrivendolo. Se *file* non esiste, viene creato.

*es.* digitando *“*ls-l *> filelist.txt”,* l’output di *“*ls *-*l*”* verrà stampato sul file *filelist.txt*

* ***comando* 2> *file* :** ridireziona lo stderr di *comando* a *file*, sovrascrivendolo. Se *file* non esiste, viene creato.
* *es.* digitando *“*ls -l 2> *log.txt”,* i messaggi di errore di *“*ls -l*”* verranno stampati sul file *log.txt*
* ***comando* &> *file* :** ridireziona lo stdout e lo stderr di *comando* a *file*, sovrascrivendolo.   
  Se *file* non esiste, viene creato.
* Usando ***>>, 2>>*** e **&>>**si ha lo stesso effetto rispettivamente di ‘>’, “2>” e “&>”, con la differenza che viene scritto sul file in append, invece di sovrascriverlo.
* ***comando* < *file :*** ridireziona lo stdin di *comando* a *file*. E’ possibile combinarlo con ‘>’.

*es1* il comando “sort *file*” legge le linee di testo di *file*, le ordina in un certo modo (lasciando *file* inalterato) e stampa il risultato dell’ordinamento. Se *file* non è indicato, le linee da ordinare sono lette dallo stdin. Senza alcuna opzione, le linee vengono ordinate alfabeticamente   
in base al primo carattere.

Digitando allora *“*sort< *file.txt”*, il comando legge dallo stdin, e quindi da *file.txt*,   
le linee di testo in esso contenute, le riordina alfabeticamente in base al primo carattere   
e stampa il risultato sulla shell.

es2 digitando “sort< *file1.txt* > *file2.txt*”, il comando legge dallo stdin, e quindi da *file1.txt*,   
le linee di testo in esso contenute, le riordina alfabeticamente in base al primo carattere,   
e stampa il risultato sullo stdout, e quindi su *file2.txt*.

* ***comando1* | *comando2* :** ridireziona lo stdout di *comando1* allo stdin di *comando2.*   
  E’ possibile usarlo più volte e in combinazione con le altre ridirezioni viste.

*es1* digitando *“*ls *-l mydir |* less*”* viene eseguito innanzitutto “ls *-l mydir*”.   
L’output di “ls *-l mydir*” viene mandato allo stdin del successivo comando *“*less*”*.   
Questo comando, dunque, stamperà sulla shell l’output del comando precedente un po’   
per volta, con la possibilità di muoversi all’interno di esso in modo interattivo.

*es2* il comando “uniq *file*” legge le linee di testo di *file*, elimina le linee di testo ripetute   
(lasciando *file* inalterato) e stampa il risultato dell’eliminazione. Se *file* non è indicato,   
le linee da eliminare sono lette dallo stdin.

Digitando allora *“*cat *file1.txt file2.txt |* sort *|* uniq *> result-file”* viene eseguito innanzitutto   
*“*cat *file1.txt file2.txt”*.

L’output di “cat *file1.txt file2.txt*” viene mandato allo stdin del successivo comando “sort”.   
Questo comando, dunque, leggerà le linee di testo contenute nell’output del comando precedente, le riordinerà alfabeticamente in base al primo carattere e stamperà   
il risultato dell’ordinamento.

L’output di “sort” viene mandato allo stdin del successivo comando “uniq”. Questo comando, dunque, leggerà le linee di testo contenute nell’output del comando precedente, eliminerà le linee di testo ripetute e stamperà il risultato dell’eliminazione.

L’output di “uniq” viene infine stampato sul file *result-file*.

**METACARATTERI (WILD CARD)**

I metacaratteri sono dei simboli che permettono di utilizzare comandi in maniera più efficiente, indicando insiemi di file e directory. I più importanti sono tre:

* **\*** : indica zero o più caratteri;
* **?** : indica un carattere;
* **[abc]** **o [a – c]** : indica un carattere appartenente al set indicato (e il set si può indicare   
  o elencando carattere per carattere, o indicando un intervallo di caratteri se questi   
  sono contigui).

La shell poi implicitamente “espanderà” i metacaratteri, sostituendo l’insieme di file e directory   
indicati attraverso i metacaratteri nei singoli file e directory di tale insieme effettivamente presenti, separati tra loro da spazio.

*es.* il comando “ls *file1 file2* … *fileN*” elenca *file1, file2, …, fileN*. Possiamo dunque usare i metacaratteri per indicare l’insieme di *file* che si vogliono elencare. Supponiamo di essere in una certa directory   
che contiene i file *aa.c*, *abc.c*, *a.c, a.h, axc.c*:

* digitando “ls *\*.c*”, la shell implicitamente espanderà *\*.c* in *aa.c* *abc.c* *a.c* *axc.c*,   
  per cui verranno elencati *aa.c, abc.c, a.c* e *axc.c*;
* digitando “ls *a\*.c”*, la shell implicitamente espanderà *a\*.c* in *aa.c abc.c a.c axc.c*,   
  per cui verranno elencati *aa.c, abc.c, a.c* e *axc.c*;
* digitando “ls *?.?”*, la shell implicitamente espanderà *?.?* in *a.c a.h*, per cui verranno elencati   
  *a.c* e *a.h*;
* digitando “ls *a??.c”*, la shell implicitamente espanderà *a??.*c in *abc.c axc.c*,   
  per cui verranno elencati *abc.c* e *axc.c*;
* digitando “ls *a[b-t]c.c”*, la shell implicitamente espanderà *a[b-t]c.c* in *abc.c*,   
  per cuiverrà elencato *abc.c*;
* digitando “ls *a[4fx]c.c”*, la shell implicitamente espanderà *a[4fx]c.c* in *axc.c*,   
  per cui verrà elencato *axc.c*

**UTENTI E GRUPPI**  *01-10-22*

Un **utente** è un utilizzatore del sistema operativo. Ogni utente ha un **username** (cioè, un nome),   
ed è identificato univocamente attraverso un numero naturale detto **UID** (User ID).

Esistono poi i **gruppi**, cioè insiemi di utenti. Ogni gruppo ha un **groupname** (cioè, un nome),   
ed è identificato univocamente attraverso un numero naturale detto **GID** (Group ID).

Ogni utente deve appartenere sempre ad almeno un gruppo, e quando viene creato un nuovo utente, se non diversamente specificato, viene creato anche un gruppo che ha, come groupname,   
l’username dell’utente appena creato, e a cui tale utente apparterrà. Ogni utente ha poi   
un **primary group** (e cioè, un gruppo principale).

Vediamo ora un po’ di comandi:

* **passwd:** permette di cambiare la password dell’utente corrente;
* **id *username:*** mostra l’UID e i gruppi dell’utente che ha l’*username* indicato.   
  Se non viene indicato *username*, mostra l’UID e i gruppi dell’utente corrente;
* **groups *username:*** mostra i gruppi dell’utente che ha l’*username* indicato.   
  Se non viene indicato *username*, mostra i gruppi dell’utente corrente;
* **adduser *username:*** crea un nuovo utente che ha l’*username* indicato.   
  Può essere usato solo dall’utente root;
* **deluser *username*:** elimina l’utente che ha l’*username* indicato. Può essere usato solo dall’utente root.
* **addgroup *groupname***: crea un gruppo che ha il *groupname* indicato. Può essere usato   
  solo dall’utente root;
* **delgroup *groupname***: rimuove il gruppo che ha il *groupname* indicato. Può essere usato   
  solo dall’utente root;
* **Su *username*:** permette di accedere alla shell dell’utente che ha l’*username* indicato   
  (e quindi tutti i comandi eseguiti su tale shell vengono eseguiti come l’utente   
  che ha l’*username* indicato). Se non viene specificato *username*, permette di accedere   
  alla shell dell’utente root.
* **Sudo *username nome\_comando*:** permette di eseguire *nome\_comando* come l’utente   
  che ha l’*username* indicato. Se non viene specificato *username*, *nome\_comando*   
  viene eseguito come l’utente root. Può essere usato solo da un utente che fa parte   
  del gruppo *sudo*.

**PERMESSI DI ACCESSO**

Ogni file/directory ha:

* Un **owner** (un utente proprietario);
* Un **group owner** (un gruppo proprietario).

Di conseguenza, dato un qualsiasi file/directory, si possono distinguere tre classi di utenti:

* L’owner del file/directory;
* Gli utenti appartenenti al group owner del file/directory;
* Gli altri utenti.

Ciascuna classe di utenti ha dei **permessi di accesso** specifici su tale file/directory (cioè, ogni classe di utenti può fare o non fare certe cose specifiche su tale file/directory), e quando un utente prova   
ad accedere a un file/directory, vengono applicati i permessi:

* Relativi all’owner se l’utente è l’owner di tale file/directory;
* Relativi agli utenti appartenenti al group owner, se l’utente non è l’owner di tale file/directory, ma appartiene al group owner di tale file/directory;
* Relativi agli altri utenti, se l’utente non è né l’owner di tale file/directory, né appartiene   
  al group owner di tale file/directory.

Dato un file/directory, un utente può avere il permesso di accesso in:

* **r** : Read (lettura);
  + Se un utente ha il permesso di accesso in lettura ad un file, allora può leggerne   
    il contenuto;
  + Se un utente ha il permesso di accesso in lettura a una directory, allora può leggere l’elenco del suo contenuto;
* **w** : Write (scrittura);
  + Se un utente ha il permesso di accesso in scrittura ad un file, allora può modificarne   
    il contenuto;
  + Se un utente ha il permesso di accesso in scrittura ad una directory,   
    allora può modificare l’elenco del suo contenuto (quindi può rinominare, creare   
    e cancellare file/directory al suo interno);
* **x** : eXecute (esecuzione);
  + Se un utente ha il permesso di accesso in esecuzione ad un file, allora può eseguirlo;
  + Se un utente ha il permesso di accesso in esecuzione ad una directory,   
    allora può attraversarla;

Per avere la **rappresentazione simbolica** (e cioè, con i caratteri visti sopra) dei permessi relativi   
a un file/directory e vedere l’owner e il group owner di tale file/directory, si usa il comando “ls -l”.

*Esempio di output:*

*Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente*

I permessi relativi a un file/directory vengono quindi rappresentati come tre triplette:

* La prima tripletta rappresenta i permessi dell’owner su tale file/directory;
* La seconda tripletta rappresenta i permessi degli utenti appartenenti al group owner   
  su tale file/directory;
* La terza tripletta rappresenta i permessi degli altri utenti su tale file/directory;

Di ogni tripletta:

* Il primo carattere rappresenta il permesso di accesso in lettura (*r* se si ha, - altrimenti);
* Il secondo carattere rappresenta il permesso di accesso in scrittura (*w* se si ha, - altrimenti);
* Il terzo carattere rappresenta il permesso di accesso in esecuzione (*x* se si ha, - altrimenti).

*Es.* nell’immagine sopra, per quanto riguarda *esempio.txt,* l’owner può leggerlo e modificarlo (compare ‘r’ e ‘w’ nella prima tripletta) mentre gli utenti appartenenti al group owner e gli altri utenti possono solo leggerlo (compare solo ‘r’ nella seconda e nella terza tripletta).

Esiste anche la cosiddetta **rappresentazione ottale** dei permessi relativi a un file/directory,   
costituita da tre cifre in base 8 che rappresentano, in ordine, i permessi dell’owner, i permessi   
degli utenti appartenenti al group owner e i permessi degli altri utenti su tale file/directory.   
Ogni cifra è ottenuta come somma di valori, in base ai permessi della classe di utenti corrispondente. In particolare, si somma:

* 4 se tale classe di utenti ha il permesso di accesso in lettura;
* 2 se tale classe di utenti ha il permesso di accesso in scrittura;
* 1 se tale classe di utenti ha il permesso di accesso in esecuzione.

*es.* *777* indica che tutte le classi di utenti possono leggere, scrivere ed eseguire   
il file/directory considerato (4 + 2 + 1);

*750* indica che l’owner può leggere, scrivere ed eseguire il file/directory considerato (4 + 2 + 1);   
gli utenti appartenenti al group owner possono leggere ed eseguire il file/directory considerato (4 + 1);   
gli altri utenti non hanno alcun permesso di accesso sul file/directory considerato (0).

**PERMESSI AGGIUNTIVI**

Per determinare i permessi di accesso di un processo, il meccanismo è un po’ diverso.   
Innanzitutto, ogni processo, tra i vari identificativi che ha, presenta questi quattro:

* **RUID (Real UID)**: è l’UID dell’utente che esegue il processo;
* **RGID (Real GID)**: è il GID del primary group dell’utente che esegue il processo;
* **EUID (Effective UID)**: è l’UID dell’utente di cui il processo assume i permessi   
  (e cioè, il processo accederà a tutti i file/directory che hanno, come owner, tale utente,   
  come se anche lui fosse l’owner di tali file/directory);
* **EGID (Effective GID)**: è il GID del gruppo di cui il processo assume i permessi da appartenente (e cioè, il processo accederà a tutti i file/directory che hanno, come group owner, tale gruppo, come se anche lui appartenesse a tale gruppo, a meno che non abbia i permessi da owner   
  su tale file/directory).

Di default, EUID = RUID e EUID = RGID (dunque un processo assume i permessi dell’utente   
che lo esegue e i permessi da appartenente al primary group di tale utente).   
Esistono però dei permessi aggiuntivi relativi ai file eseguibili che modificano questo:

* **SUID (Set UID)**: se un file eseguibile ha il permesso SUID abilitato, allora eseguendo tale file,   
  il processo risultante avrà EUID = *UID\_dell’owner\_del\_file\_eseguibile* (e quindi, assume   
  i permessi dell’owner del file eseguibile. Un esempio è il file eseguibile del comando *passwd*: solo l’utente root può accedere al file delle password, e quindi il file eseguibile ha come owner l’utente root e il permesso SUID abilitato, così che, eseguendo tale file, il processo risultante assuma i permessi dell’utente root, e quindi possa effettivamente modificare il file   
  delle password);
* **SGID (Set GID)**: se un file eseguibile ha il permesso SGID abilitato, allora eseguendo tale file,   
  il processo risultante avrà EGID = G*ID\_del\_group\_owner\_del\_file\_eseguibile* (e quindi, assume i permessi da appartenente al group owner del file eseguibile).

La rappresentazione simbolica di questi permessi aggiuntivi è la seguente:

* Se un file eseguibile ha il permesso SUID abilitato, allora nel campo relativo   
  al permesso di accesso in esecuzione dell’owner è presente la lettera ‘*s*’;
* Se un file eseguibile ha il permesso SGID abilitato, allora nel campo relativo al permesso   
  di accesso in esecuzione degli utenti appartenenti al group owner è presente la lettera ‘s’.

Per la rappresentazione ottale di questi permessi aggiuntivi si usa un’ulteriore cifra prima delle 3 relative ai permessi di accesso delle varie classi di utenti. Questa cifra è ottenuta come somma di:

* 4 se è abilitato il permesso SUID;
* 2 se è abilitato il permesso SGID.

*es.* 6754 indica che per il file eseguibile considerato sono abilitati sia il permesso SUID   
che il permesso SGID (4+2). La rappresentazione simbolica corrispondente è rwsr-sr--.

**CHMOD**

E’ possibile modificare i permessi relativi a un file/directory attraverso il comando **chmod.**Può essere usato solo dall’owner del file/directory e, se applicato ad una directory, con l’opzione “-R”   
è possibile modificare in modo ricorsivo anche i permessi di tutti i file/directory in essa contenuti   
(quindi dei file e directory in essa contenuti, dei file e delle directory contenuti nelle directory, dei file e directory contenuti nelle directory contenute nelle directory…).

Recursive

La sintassi di chmod per modificare i permessi relativi ad un *file/directory* usando la loro rappresentazione simbolica è la seguente:

chmod whohowwhich *file/directory*

dove:

* who indica, scritte una attaccata all’altra, le classi di utenti per cui devono essere modificati   
  i permessi:
  + *u* per l’owner;
  + *g* per gli utenti appartenenti al group owner;
  + o per gli altri utenti;
* how indica in che modo devono essere modificati i permessi:
  + + per aggiungere permessi;
  + – per togliere permessi;
  + = per assegnare permessi;
* which indica, scritti uno attaccato all’altro, quali sono i permessi interessati, usando la loro rappresentazione simbolica.

*es.* “chmod *ug-rx* *compiti*”, alle classi di utenti indicate, toglie i permessi indicati relativi a *compiti*.

La sintassi di chmod per assegnare i permessi relativi ad un *file/directory* usando la loro rappresentazione ottale è la seguente:

chmod *rappresentazione\_ottale\_permessi file/directory*

*es.* “chmod *6754 compiti*” assegna, nel modo indicato, i permessi relativi a *compiti*.

**CHOWN E CHGRP**

E’ possibile modificare l’owner e il group owner di un file/directory attraverso i seguenti comandi:

CHangeOWNer

* **chown *username* *file/directory***: imposta l’utente che ha l’*username* indicato   
  come nuovo owner di *file/directory*. Può essere usato solo dall’utente root;

CHangeGRPoupowner

* **chgrp *groupname* *file/directory*:** imposta il gruppo che ha il *groupname* indicato   
  come nuovo group owner di *file/directory*. Può essere usato solo da un utente appartenente   
  al gruppo *groupname*.

**COMANDI PER LA GESTIONE DEI GRUPPI** *16-10-24*

Gli utenti che fanno parte di un gruppo possono essere di due tipi:

* **Membri**;
* **Amministratori**.

Gli amministratori hanno più privilegi rispetto ai membri nella gestione del gruppo. Vediamo infatti alcuni comandi:

* **gpasswd -a *username* *groupname*:** aggiunge l’utente che ha l’*username* indicato   
  al gruppo che ha il *groupname* indicato, come membro. Può essere usato   
  solo da un amministratore del gruppo;

Add

* **gpasswd -d *username* *groupname*:** rimuove l’utente che ha l’*username* indicato   
  dal gruppo che ha il *groupname* indicato. Può essere usato solo da un amministratore   
  del gruppo;

Delete

* **gpasswd -M *username1, username2, … , usernameN groupname*:** rende gli utenti che hanno l’*username* indicato membri del gruppo che ha il *groupname* indicato (se questi non fanno parte del gruppo, ne diventano parte). Può essere usato solo da un amministratore del gruppo;

Member

* **gpasswd -A *username1, username2, … , usernameN groupname*:** rende gli utenti che hanno l’*username* indicato amministratori del gruppo che ha il *groupname* indicato (se questi   
  non fanno parte del gruppo, ne diventano parte). Può essere usato solo da un amministratore del gruppo;

Admin

* **gpasswd *groupname***:imposta/cambia la password del gruppo che ha il *groupname* indicato. Può essere usato solo da un amministratore del gruppo;
* **gpasswd -r *groupname***:rimuove la password dal gruppo che ha il *groupname* indicato.   
  Può essere usato solo da un amministratore del gruppo;

Remove

* **newgrp *groupname*:** rende il gruppo che ha il *groupname* indicato il nuovo primary group dell’utente corrente per la sessione di login corrente (se l’utente non fa parte del gruppo,   
  ne diventa parte fino alla fine della sessione). E’ necessario che il gruppo abbia una password, che verrà chiesta all’utente se questo non fa parte del gruppo.

**FILE DI CONFIGURAZIONE DEGLI UTENTI**

Su UNIX ci sono due file di configurazione degli utenti:

* ***/etc/passwd***: contiene informazioni pubbliche sugli utenti;
* ***/etc/shadow***: contiene informazioni sensibili sugli utenti (come la password).

Il file */etc/passwd* contiene, per ogni utente, una linea di testo costituita da vari campi separati da ‘:’, che sono in ordine:

* Username dell’utente;
* Una ‘x’. Un tempo questo campo conteneva la password dell’utente;
* UID dell’utente;
* GID del primary group dell’utente;
* Dati personali dell’utente;
* Path assoluto della home dell’utente;
* Path assoluto della shell predefinita dell’utente.

Impostando l’ultimo campo a *“/sbin/nologin”* o *“/bin/false”* si impossibilita a fare il login   
con tale utente.

E’ possibile modificare il file */etc/passwd* - oltre a come si fa per un qualsiasi file - anche attraverso   
il comando **vipw**, che può essere usato solo dall’utente root.

VIPassWord

Il file */etc/shadow* contiene per ogni utente una linea di testo costituita da vari campi separati da ‘:’,   
che sono in ordine:

* Username dell’utente;
* Password cifrata dell’utente
* Data di ultima modifica della password (espressa in numero di giorni dal 1° gennaio 1970);
* Durata minima della password (espressa in numero di giorni);
* Durata massima della password (espressa in numero di giorni);
* Tempo prima della scadenza della password in cui l’utente viene avvisato (espresso in numero   
  di giorni);
* Tempo dopo la scadenza della password in cui questa è ancora accettata (espresso   
  in numero di giorni);
* Data di scadenza dell’account (espressa in numero di giorni dal 1° gennaio 1970);
* Campi riservati per un eventuale utilizzo futuro.

Shadow

E’ possibile modificare il file */etc/shadow* - oltre a come si fa per un qualsiasi file – anche attraverso   
il comando **vipw -s**, che può essere usato solo dall’utente root.

**FILE DI CONFIGURAZIONE DEI GRUPPI**

Su UNIX ci sono due file di configurazione dei gruppi:

* ***/etc/group***: contiene informazioni pubbliche sui gruppi;
* ***/etc/gshadow***: contiene informazioni sensibili sui gruppi (come la password e l’elenco   
  degli amministratori).

Il file */etc/group* contiene, per ogni gruppo, una linea di testo costituita da vari campi separati da ‘:’,   
che sono in ordine:

* Groupname del gruppo;
* Una ‘x’. Un tempo questo campo conteneva la password del gruppo;
* GID del gruppo;
* Lista degli utenti appartenenti al gruppo (separati da virgola)*.* Non è presente l’utente   
  che ha come primary group tale gruppo.

E’ possibile modificare il file */etc/group* - oltre a come si fa per un qualsiasi file – anche attraverso   
il comando **vigr**, che può essere usato solo dall’utente root.

VIGRoup

Il file */etc/gshadow* contiene, per ogni gruppo, una linea di testo costituita dai seguenti campi:

* Groupname del gruppo;
* Password cifrata del gruppo. Un campo vuoto oppure i caratteri ‘\*’ o ‘!’ indicano   
  che la password non è impostata.
* Lista degli utenti amministratori del gruppo (separati da virgola);
* Lista degli utenti appartenenti al gruppo (separati da virgola). Non è presente l’utente   
  che ha come primary group tale gruppo.

E’ possibile modificare il file */etc/gshadow* - oltre a come si fa per un qualsiasi file – anche attraverso   
il comando **vigr -s**, che può essere usato solo dall’utente root.

Shadow

**RICERCA DI FILE**

Vediamo alcuni comandi per la ricerca di file (intesi come file regolari, directory…).

**FIND**

**find** è un comando che permette di ricercare file sulla base delle loro proprietà, elencandone i path   
o eseguendo azioni su di essi.

La sintassi è la seguente:

find [*dir1 dir2 … dirN*] [*espressione*]

dove:

* *dir1 dir2 … dirN* sono le directory a partire dal quale verrà effettuata la ricerca   
  in maniera ricorsiva (e quindi, nelle directory indicate, nelle directory contenute   
  nelle directory indicate, nelle directory contenute nelle directory contenute nelle directory indicate…);
* *espressione* specifica come devono essere ricercati i file e quali azioni devono essere eseguite   
  su di essi.

*espressione* è composta da una sequenza di elementi, che possono essere:

* **Test**: valutano una proprietà del file, restituendo true o false;
* **Azioni**: eseguono un’azione sui file trovati.

In *espressione* si possono usare gli operatori logici:

* -a indica AND. Se tra due elementi di *espressione* non è indicato alcun operatore,   
  questi sono implicitamente collegati attraverso AND;
* -o indica OR;
* ! indica NOT.

Alla fine verranno elencati i path o eseguite azioni sui file per cui *espressione* restituisce true.

Alcuni test sono i seguenti:

* **-name *nome*:** restituisce true se il file si chiama *nome*. Si possono usare i metacaratteri,   
  e in tal caso, per evitare che la shell li espanda (portando così ad una sintassi non corretta), *nome* va scritto tra apici;
* **-type *tipo*:** restituisce true se il file è del *tipo* indicato. *tipo* può essere:
  + f: file regolare;

File

* + d: directory;

Link

Directory

* + l: soft link.
* **-size [*prefix*]*n*[*suffix*]:** restituisce true sulla base della dimensione del file:
  + *n* indica la dimensione con cui effettuare il confronto;
  + *suffix* indica l’unità di misura di *n*:

Char (byte)

* + - c: byte;

Kilobyte

* + - k: kilobyte;

Megabyte

* + - M: megabyte;

Gigabyte

* + - G: gigabyte.
  + *prefix* indica come deve essere la dimensione del file rispetto a *n*:
    - +: maggiore;
    - -: minore;
    - se non si specifica nulla: uguale.
* **-perm [*prefix*]*mode*:** restituisce true sulla base dei permessi del file:
  + *mode* indica i permessi con cui effettuare il confronto. Questi possono essere indicati:
    - Usando la loro rappresentazione simbolica, specificando *who*=*which*   
      per ogni classe di utenti interessata e separando ogni blocco relativo ad una classe di utenti dall’altro attraverso una virgola;
    - Usando la loro rappresentazione ottale.
  + *prefix* indica come devono essere i permessi del file rispetto a *mode*:
    - /: il file deve avere almeno uno dei permessi specificati in *mode*;

alMENO

* + - -: il file deve avere almeno i permessi specificati in *mode*;
    - se non si specifica nulla: il file deve avere esattamente i permessi specificati   
      in *mode*.
* **-user *user*:** restituisce true se il file ha come owner *user*, specificato come username o UID;
* **-group *group*:** restituisce true se il file ha come group owner *group*, specificato come groupname o GID.

Alcune azioni sono le seguenti:

* **-delete**: elimina i file che hanno superato i test precedenti;
* -**exec *command* ;** : esegue *command* per ogni file che ha superato i test precedenti.   
  E’ possibile usare la stringa “{}” per indicare il nome del file attualmente processato.   
  *command* viene eseguito a partire dalla directory corrente. Per eseguire *command*a partire dalla directory del file attualmente processato bisogna usare -execdir al posto di -exec.  
  Se si usa il comando find dalla shell, ‘;’ va scritto tra apici.

*es.*

* “find *path* -name ‘*prova*\*’ !-type d” ricerca a partire da *path* i file il cui nome inizia con *prova*   
  e che non sono directory, elencandone i path;
* “find *path* !-name ‘\*.csv’ -size +50M -execdir ls -l {} ‘;’ ” ricerca a partire da *path* i file   
  con estensione diversa da *.csv* e di dimensione superiore a 50 MB, eseguendo per ciascuno   
  di questi il comando “ls -l {}” a partire dalla directory del file attualmente processato,   
  mostrando così dettagli su tale file.
* “find *path* -perm -664” ricerca a partire da *path* i file che hanno almeno i permessi indicati, elencandone i path;
* “find *path* -perm /u=w,g=w” ricerca a partire da *path* i file che hanno almeno uno   
  dei permessi indicati, elencandone i path.

**LOCATE**

**locate** è un comando che permette di ricercare file sulla base del loro path, elencandone i path.

La sintassi è la seguente:

locate *stringa*

Dove *stringa* è tale che:

* Se non si specifica alcun metacarattere, verranno elencati tutti i file che contengono   
  nel loro path *stringa;*
* Se si specificano metacaratteri, per evitare che la shell li espanda, *stringa* va scritta tra apici,   
  e verranno elencati i file che hanno come path *stringa*.

I file vengono ricercati sfruttando un database aggiornato periodicamente. E’ possibile forzare l’aggiornamento di questo database attraverso il comando **updatedb**, che può essere usato   
solo dall’utente root.

Confrontiamo find con locate:

* locate è semplice da usare, mentre find ha una sintassi molto complessa;
* locate, sfruttando un database, è veloce, mentre find, effettuando la ricerca in tutto   
  il file system, è più lento;
* find, effettuando la ricerca in tutto il file system, dà risultati sempre aggiornati, mentre locate, sfruttando un database, può dare risultati non aggiornati;
* find permette di definire test e azioni, mentre locate permette solo di ricercare file   
  sulla base del loro path, elencandone i path;
* find è un comando già installato in tutti i sistemi UNIX, mentre locate tipicamente   
  va installato successivamente.

**RICERCA DI TESTO NEI FILE DI TESTO**

**GREP**

**grep** è un comando che permette di ricercare linee di testo in uno o più file di testo, elencandole.

La sintassi è la seguente:

grep [*opzioni*] [-e] *modello1* [*-e modello2 … -e modelloN*] *file1* [file2 … *fileN*]

Dove:

* *modello1*, *modello2*, …, *modelloN* sono delle espressioni regolari. Le regole che ci interessano per definire un’espressione sono:
  + **^*subexp***: la linea di testo deve iniziare con *subexp*;
  + ***subexp*$**: la linea di testo deve finire con *subexp*;
  + **[abc]**: trova un singolo carattere tra quelli contenuti tra parentesi;
  + **[a-z]**: trova un singolo carattere nel range contenuto tra parentesi;
  + **.** : trova un qualsiasi carattere;
  + ***subexp*\***: indica che *subexp* deve essere ripetuta zero o più volte.

Se si vuole usare un metacarattere (un carattere usato di norma per definire una regola)   
come un carattere normale, bisogna precederlo da ‘\’. Le parentesi tonde “()” non sono usate   
per definire una sottoespressione, ma sono dei caratteri normali. Se si vuole specificare   
più di un *modello*, ciascun *modello* (anche il primo) deve essere preceduto da “-e”;

* *file1* [*file2* … *fileN*] sono i file in cui effettuare la ricerca;
* *opzioni* permettono di cambiare il comportamento di grep. Senza alcuna opzione,   
  verranno elencate le linee di testo che rispettano almeno uno dei *modelli* indicati.   
  Alcune opzioni utili sono:
  + -i: ignora le distinzione tra maiuscole e minuscole;

Ignore

* + -v: elenca le linee che non rispettano alcun *modello*;

inVerse

* + -n: precede ogni linea trovata con il numero di linea all’interno del *file*   
    in cui è stata trovata;

Count

Number

* + -c: mostra solo il conteggio delle linee trovate in ogni *file*;

**ARCHIVIAZIONE E COMPRESSIONE**

Un **archivio** è un file regolare ottenuto dalla concatenazione di più file (intesi come file regolari, directory…).

La **compressione di un file** (inteso come file regolare, directory… anche non un archivio)   
consiste nel sostituire il file con un altro file regolare contenente le stesse informazioni, ma tipicamente   
di dimensione minore.

Vediamo ora alcuni comandi che permettono l’archiviazione e la compressione di file.

**TAR**  
Il comando **tar** permette di lavorare con gli archivi.

La sintassi è la seguente:

tar *modalità*[*opzioni*] *file1* [*file2* … *fileN*]

Dove:

* *file1* [file2 … *fileN*] sono i file da aggiungere a un archivio, estrarre da un archivio…   
  (a seconda delle *modalità* specificate);
* *opzioni* e *modalità* permettono di specificare il modo in cui il comando deve operare:
  + Alcune *opzioni* sono:

File

* + - f *archive*: permette di specificare l’archivio *archive* da creare,   
      a cui aggiungere file, da cui estrarre file… (a seconda delle *modalità* specificate);

gZip

* + - z: comprime *archive* con gzip;

b(lowJob)zip

* + - j: comprime *archive* con bzip2.
    - --directory *dir*: cambia la directory corrente in *dir* prima di eseguire   
      qualunque operazione. Tutti i path successivi a questa opzione vanno espressi considerando *dir* come directory corrente.
  + Alcune *modalità* sono:

Create

* + - c: crea *archive* contenente *file1* [*file2* … *fileN*];

addeR

* + - r: aggiunge *file1* [*file2* … *fileN*] ad *archive*;

eXtract

* + - x: estrae *file1* [file2 … *fileN*] da *archive* nella directory corrente.   
      Se non si specifica alcun *file* da estrarre, vengono estratti tutti i file di *archive*.
    - --delete cancella *file1* [*file2* … *fileN*] da *archive*;

lisT

* + - t: elenca i file di *archive*;

Una volta creato un archivio, questo verrà salvato come un file con estensione:

* .tar se l’archivio non è stato compresso;
* .tar.gz se l’archivio è stato compresso con gzip;
* .tar.bz2 se l’archivio è stato compresso con bzip2;

**GZIP / GUNZIP E BZIP2/BUNZIP2**

E’ possibile comprimere o decomprimere un file attraverso i seguenti comandi:

* **gzip *file1* [*file2 … fileN*]**: comprime con gzip ciascun *file*, salvandolo   
  come un file con lo stesso nome del file di partenza ed estensione .gz. Il file di partenza   
  viene successivamente eliminato.
* **bzip2 *file1* [*file2 … fileN*]**: comprime con bzip2 ciascun *file*, salvandolo   
  come un file con lo stesso nome del file di partenza ed estensione .bz2. Il file di partenza   
  viene successivamente eliminato.
* **gunzip *file1* [*file2 … fileN*]**: decomprime ciascun *file* precedentemente compresso con gzip, salvandolo come un file con lo stesso nome del file di partenza (esclusa l’estensione .gz).   
  Il *file* di partenza viene successivamente eliminato.
* **bunzip2 *file1* [*file2 … fileN*]**: decomprime ciascun *file* precedentemente compresso con bzip2, salvandolo come un file con lo stesso nome del file di partenza (esclusa l’estensione .bz2).   
  Il file di partenza viene successivamente eliminato.

**PROCESSI IN UNIX** *24-10-24*

Un processo in UNIX ha una memoria contenente i dati separata dalla memoria contenente il codice:

* La memoria contenente i dati è privata, e cioè un processo non può accedere ai dati dell’altro;
* La memoria contenente il codice è condivisa, e cioè più processi possono eseguire   
  lo stesso codice.

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

Durante la sua “vita”, un processo UNIX passa per vari stati:

Immagine che contiene testo, cerchio, diagramma, schermata

Descrizione generata automaticamente

Simile a quanto già visto varie volte, con l’aggiunta di due stati:

* Un processo *P* può passare allo stato *swapped* se il processo di sistema *swapper*   
  sposta in memoria secondaria tutte le pagine di *P* attualmente presenti in memoria fisica.   
  *P* uscirà poi da questo stato quando verranno ricaricate in memoria fisica tutte le sue pagine precedentemente presenti;
* Quando un processo termina, questo passa allo stato *zombie*, in cui le sue risorse   
  ancora non vengono distrutte dal sistema operativo, in modo che altri processi   
  possano usufruirne.

Il descrittore di ogni processo è detto Process Control Block (PCB). Questo è diviso in due strutture dati distinte:

* Process Structure: contiene informazioni indispensabili. Rimarranno sempre in memoria fintantoché il processo non è terminato. Alcuni campi sono:
  + PID del processo;
  + PID del processo padre;
  + Priorità del processo;
  + Stato del processo;
  + Riferimento alla memoria del processo contenente i dati;
  + Riferimento alla memoria del processo contenente il codice;
  + Riferimento al processo successivo nella coda in cui il processo considerato si trova;
* User Structure: contiene informazioni utili solo quando il processo è in memoria. Sono soggette a swap-out. Alcuni campi sono:
  + Copia del contesto del processo;
  + Informazioni sulle risorse utilizzate dal processo;
  + Informazioni sulla gestione dei segnali;
  + Informazioni sull’*ambiente* del processo: directory corrente, RUID, RGID, EUID, EGID…

**PRIMITIVE VARIE IN C PER I PROCESSI**

**FORK**

La primitiva **pid\_t** **fork()** permette al processo che la invoca di creare un processo figlio.  
Il processo figlio condivide con il padre la memoria contenente il codice, ed eredita dal padre   
una copia della memoria contenente i dati e della User Structure, contenente tra le varie cose   
la copia del contesto del padre aggiornata al momento dell’invocazione della fork(), e quindi anche   
il valore del PC. Il processo figlio, dunque, eseguirà lo stesso codice del padre partendo dall’istruzione immediatamente successiva alla fork(), con ogni variabile inizializzata con il valore datole dal padre.

La primitiva restituisce:

* Nel processo padre, il PID del processo figlio creato, un valore negativo in caso di errore;
* Nel processo figlio, 0.

Il valore di ritorno di fork(), dunque, può essere usato per differenziare il comportamento   
del processo padre da quello del processo figlio.

**EXIT()**

Un processo può terminare in maniera:

* Involontaria, se esegue azioni illegali o a seguito della ricezione di un qualche segnale;
* Volontaria, se esegue l’ultima istruzione o se invoca exit().

La primitiva **void** **exit(int *status*)** permette al processo che la invoca di terminare volontariamente,   
con stato di terminazione *status*.

**WAIT()**

La primitiva **pid\_t** **wait(int\* *status*)** sospende il processo che la invoca fintantoché non termina   
un suo processo figlio. Nella variabile puntata da *status* verranno salvate informazioni su come   
è terminato tale processo figlio.

Un processo può invocare più volte wait(), e ogni volta si sospenderà fintantoché non terminerà   
un figlio diverso.

La primitiva restituisce:

* Il PID del figlio terminato;
* Un valore negativo se il processo che l’ha invocata non ha avuto alcun processo figlio   
  o se ne ha avuti ed ha già invocato wait() per ciascuno di questi.

Ci sono alcune macro che, passando come argomento la variabile *stat* su cui wait() ha salvato informazioni su come è terminato il processo figlio, permettono di ottenere tale informazioni:

* WIFEXITED(*stat*): restituisce true se il processo figlio è terminato volontariamente;

WaitEXITSTATUS

WaitIFEXITED

* WEXITSTATUS(*stat*): restituisce lo stato di terminazione del processo figlio.   
  Va usata solo se il processo è terminato volontariamente.

**EXECL()**

La primitiva **int execl(char\* *path*, char\* *arg0*, …, char\* *argN*, NULL)** sostituisce la memoria contenente i dati e la memoria contenente il codice del processo che la invoca con quelle dovute all’esecuzione del programma avente il path assoluto *path*, di nome *arg0*, con gli argomenti *arg1*, … *argN*.   
In caso di successo, dunque, il processo passerà ad eseguire il nuovo programma,   
terminando al termine di esso. In caso di errore, invece, execl() restituisce un valore negativo,   
e il processo continuerà ad eseguire le istruzioni immediatamente successive alla primitiva.

**COMANDI VARI PER I PROCESSI**

ProcesseS

Il comando **ps** elenca i processi attivi al momento dell’esecuzione del comando. Di default, elenca solo i processi dell’utente corrente, attivati dallo stesso terminale su cui è stato eseguito il comando.

Alcune opzioni utili sono:

All

* a: elenca i processi di tutti gli utenti;

User

* -u *utente*: elenca solo i processi dell’utente che ha l’UID o l’username *utente;*

Utile

* u: elenca i processi in un formato utile all'analisi dell'utilizzo delle risorse;

eXtended

* x: elenca anche i processi attivati da un terminale diverso da quello su cui è stato eseguito   
  il comando o che non sono stati attivati da un terminale.

**PROCESS GROUP**

Un **process group** è un insieme di processi. Ogni process group è identificato univocamente   
attraverso un numero naturale detto **PGID** (Process Group ID).

Ogni processo appartiene ad uno e un solo process group, e quando viene attivato un nuovo processo, viene creato anche un process group che ha, come PGID, il PID del processo appena attivato,   
e a cui tale processo apparterrà.

Quando un processo crea un processo figlio, questo apparterrà allo stesso process group del padre.   
Il process group viene preservato anche invocando la primitiva execl().

**PRIMITIVE IN C PER OTTENERE GLI IDENTIFICATORI DI UN PROCESSO**

Abbiamo visto finora che ad ogni processo sono legati vari identificatori. Esistono delle primitive   
che permettono di ottenere questi identificatori:

* **pid\_t** **getpid():** restituisce il PID del processo che la invoca;
* **pid\_t getppid():** restituisce il PID del padre del processo che la invoca;
* **pid\_t getpgrp():** restituisce il PGID del process group cui appartiene il processo che la invoca;
* **uid\_t getuid():** restituisce il RUID del processo che la invoca;
* **uid\_t getgid():** restituisce il RGID del processo che la invoca;
* **uid\_t geteuid():** restituisce l’EUID del processo che la invoca;
* **uid\_t getegid():** restituisce l’EGID del processo che la invoca.

**SINCRONIZZAZIONE BASATA SUI SEGNALI** 30-10-24

I processi in UNIX seguono il modello di interazione a memoria locale: ogni processo ha solo   
una propria memoria contenente i dati privata. Serve dunque un qualche meccanismo che permetta   
ad un processo di scambiare messaggi con gli altri processi, e uno di questi è il meccanismo dei segnali.Un **segnale** è una “interruzione software” che viene mandata da un processo o dal sistema operativo   
a uno o più processi per comunicare il verificarsi di un certo evento.

Quando un processo riceve un segnale:

* Se non è stato specificato diversamente, viene eseguito un default handler definito dal sistema operativo;
* Altrimenti, può essere eseguito un handler definito dal programmatore, oppure il segnale   
  può essere ignorato.

Se a seguito della ricezione di un segnale il processo non è terminato, questo continua l’esecuzione   
a partire dall’istruzione immediatamente successiva all’ultima eseguita prima della ricezione   
del segnale.

Ciascun segnale è identificato da un nome simbolico e un intero. I più importanti sono:

SIGnalHangUP

* **SIGHUP (1)**: viene mandato a un processo quando il terminale da cui è stato attivato   
  viene chiuso. Il default handler termina il processo;

SIGnalINTerrupt

* **SIGINT (2):** viene mandato a un processo in foreground quando si preme CTRL + C sul terminale   
  da cui è stato attivato. Il default handler termina il processo;
* **SIGQUIT (3):** viene mandato a un processo in foreground quando si preme CTRL + \   
  sul terminale da cui è stato attivato. Il default handler termina il processo e crea un core dump;
* **SIGKILL (9):** viene mandato a un processo per terminarlo (comportamento del default handler). Non è possibile ignorare questo segnale, né il programmatore può definire un proprio handler per gestirlo;
* **SIGTERM (15):** viene mandato a un processo per terminarlo (comportamento del default handler). A differenza di SIGKILL, è possibile ignorare questo segnale, oppure il programmatore può definire un proprio handler per gestirlo;
* **SIGUSR1(10) e SIGUSR2(12):** sono segnali lasciati liberi al programmatore per comunicare   
  il verificarsi di un evento specifico definito dal programmatore stesso. Il default handler   
  termina il processo.

SIGnalUSeR

* **SIGSEGV (11):** viene mandato a un processo quando questo causa un errore di segmentazione. Il default handler termina il processo;

SIGnalALaRM

* **SIGALRM (14):** viene mandato a un processo quando il timer che ha definito precedentemente è scaduto. Il default handler termina il processo;
* **SIGCHLD (17):** viene mandato a un processo quando un suo processo figlio è terminato,   
  si è sospeso o si è risvegliato. Il default handler non fa nulla;
* **SIGSTOP (19):** viene mandato a un processo per sospenderlo (comportamento del default handler). Non è possibile ignorare questo segnale, né il programmatore può definire   
  un proprio handler per gestirlo;

SIGnalTemporarylSToP

* **SIGTSTP (20):** viene mandato a un processo in foreground quando si preme CTRL + Z   
  sul terminale da cui è stato attivato. Il default handler sospende il processo.
* **SIGCONT (18):** viene mandato a un processo per risvegliarlo dopo che si è sospeso a seguito della ricezione di un segnale SIGSTOP, SIGTSTP o SIGCONT (comportamento   
  del default handler).

Nel file signal.h sono presenti delle costanti che associano, al nome simbolico di ogni segnale,   
l’intero corrispondente.

**PRIMITIVE IN C PER I SEGNALI**

**SIGNAL()**

La primitiva **sighandler\_t** **signal(int *sig*, sighandler\_t *func*)** permette al processo che la invoca di impostare la funzione *func*, dichiarata come una funzione con un solo argomento int   
e tipo di ritorno void, come handler per gestire il segnale identificato dall’intero *sig*.   
Quando verrà eseguita *func*, l’argomento int conterrà l’intero che identifica il segnale ricevuto.   
Questo valore può essere dunque usato per differenziare il comportamento dell’handler a seconda   
del segnale ricevuto, nel caso in cui una stessa funzione *func* venisse usata come handler   
per gestire più segnali diversi.

Come *func* si può passare anche:

SIGnal\_IGNore

* SIG\_IGN: ignora il segnale;

SIGnal\_DeFauLt

* SIG\_DFL:reimposta il default handler come handler.

La primitiva restituisce:

* Un puntatore al precedente handler del segnale;

SIGnal\_ERRor

* SIG\_ERR in caso di errore.

Quando un processo crea un processo figlio, questo eredita dal padre la User Structure,   
e dunque la gestione dei segnali. Invocando la primitiva execl(), ogni segnale torna ad essere gestito   
dal default handler, fatta eccezione per i segnali ignorati, che continuano ad essere ignorati.

**KILL()**

La primitiva **int kill(pid\_t pid, int sig)** permette al processo che la invoca di mandare il segnale identificato dall’intero *sig* al processo identificato dal PID *pid*.

Come *pid* si può passare anche:

* *pid* *==* 0: il segnale viene inviato a tutti i processi appartenenti allo stesso process group   
  del processo che invoca la primitiva;
* *pid* ==-1: il segnale viene mandato a tutti i processi a cui il processo chiamante   
  può mandare segnali;
* *pid < -1*: il segnale viene mandato a tutti i processi appartenenti al process group il cui PGID   
  è -*pid*.

Un processo di un utente normale può mandare un segnale solo ad altri processi dello stesso utente.

Un processo dell’utente root, invece, può mandare un segnale a qualsiasi processo.

La primitiva restituisce:

* 0 in caso di successo;
* Un valore negativo in caso di errore.

**PAUSE()**

La primitiva **int pause()** sospende il processo che la invoca fintantoché questo non riceve un segnale.  
Se l’handler del segnale ricevuto non termina il processo che la invoca, la primitiva restituisce   
sempre un valore negativo.

**SLEEP()**

La primitiva **unsigned int sleep(unsigned int seconds)** sospende il processo che la invoca fintantoché non sono passati *seconds* secondi o fintantoché il processo non riceve un segnale da non ignorare.  
La primitiva restituisce:

* 0 se il processo è rimasto sospeso per il tempo indicato;
* Il numero di secondi per cui il processo sarebbe ancora dovuto rimanere sospeso   
  se questo è stato risvegliato a seguito della ricezione di un segnale da non ignorare   
  e l’handler del segnale ricevuto non ha terminato il processo.

**ALARM()**

La primitiva **unsigned int alarm(unsigned int seconds)** permette al processo che la invoca   
di ricevere un segnale SIGALRM dopo *seconds* secondi.

Se un processo, avendo invocato precedentemente la primitiva, ha già in programma la ricezione   
di un segnale SIGALRM, invocando nuovamente la primitiva cancella la ricezione del segnale SIGALRM programmata dall’invocazione precedente.

Passando *seconds* == 0, il processo che la invoca non riceve alcun segnale. Questo serve dunque   
a cancellare un’eventuale ricezione del segnale SIGALRM programmata da una precedente invocazione della primitiva.

La primitiva restituisce:

* 0 se il processo non ha in programma nessun’altra ricezione di un segnale SIGALRM;
* Il numero di secondi dopo il quale il processo avrebbe ricevuto un segnale SIGALRM   
  se ne era stata programmata la ricezione da una precedente invocazione della primitiva.

**COMANDI PER I SEGNALI**

**KILL**

Il comando **kill -*segnale* *pid1* [*pid2*…]** manda il segnale identificato dal nome simbolico *segnale*   
ai processi identificati dal PID *pid1*, *pid2*, …, *pidN*.. Un utente normale che esegue questo comando   
può mandare un segnale solo ad un proprio processo. L’utente root, invece, può mandare un segnale a un qualunque processo.

**NICENESS DEI PROCESSI**

L’algoritmo di scheduling usato dallo scheduler UNIX è con priorità: viene mandato in esecuzione   
il processo con la priorità maggiore tra quelli presenti in coda pronti. Ad ogni processo è associato inoltre un valore compreso tra [-20, 19], detto **niceness**, che influenza in parte la priorità:   
maggiore è la niceness di un processo, minore è la sua priorità.

E’ possibile agire sulla niceness di un processo attraverso i seguenti comandi:

* **nice –n *niceness* *programma***: attiva un processo che esegue *programma* con il valore  
  di *niceness* indicato. Un utente normale può specificare solo un valore di *niceness* positivo. L’utente root, invece, può specificare un qualsiasi valore di *niceness*.
* **renice *niceness* *PID***: imposta la *niceness* indicata al processo avente il *PID* indicato.   
  Un utente normale può modificare la niceness solo dei propri processi, e può solo aumentarla. L’utente root, invece, può modificare la niceness di un qualunque processo, aumentandola   
  o diminuendola.

**JOB CONTROL**

Un **job** è un insieme di processi dal punto di vista del terminale su cui tali processi sono stati attivati.

Ogni job è identificato univocamente attraverso un numero naturale detto **Job ID**.

Ogni volta che si esegue un programma da un terminale, il terminale crea un nuovo job, a cui appartiene l’insieme di processi attivati a seguito dell’esecuzione del programma.

Tutti i job creati da un terminale sono salvati in una *tabella dei job*, contenente, tra le varie cose,   
il job ID di ciascun job. E’ possibile visualizzare la tabella dei job di un terminale eseguendo il comando **jobs** su tale terminale.

Di default, quando si esegue un programma da terminale, il job associato viene attivato in foreground.   
Un job in foreground ha il controllo dello stdin, stdout e stderr, per cui, se non c’è stata alcuna redirezione di questi canali, questo job ha il controllo del terminale da cui è stato attivato.

E’ possibile anche attivare un job in background, eseguendo il programma da terminale in questo modo:

*programma* &

Un job in background non ha il controllo dello stdin, permettendo così all’utente di usare il terminale mentre il job viene completato.

Dato un job in foreground, premendo CTRL+Z sul terminale da cui è stato attivato verrà mandato   
il segnale SIGTSTP al job, sospendendolo di default. E’ possibile spostare un job da foreground   
a background o viceversa, risvegliandolo nel caso in cui fosse stato sospeso a seguito della ricezione di un segnale SIGTSTP, attraverso i seguenti comandi:

* **fg %*JOB\_ID*:** sposta in foreground il job identificato dal *JOB\_ID* indicato, risvegliandolo   
  nel caso in cui fosse stato sospeso a seguito della ricezione di un segnale SIGTSTP;
* **bg %*JOB\_ID*:** sposta in background il job identificato dal *JOB\_ID* indicato, risvegliandolo   
  nel caso in cui fosse stato sospeso a seguito della ricezione di un segnale SIGTSTP.

E’ possibile usare il comando kill anche con i job. In particolare, il comando **kill -*segnale* %*JOB\_ID*** manda il segnale identificato dal nome simbolico *segnale* al job identificato dal *JOB\_ID* indicato.

Dato un job (in foreground o background), chiudendo il terminale da cui è stato attivato   
verrà mandato il segnale SIGHUP al job, terminandolo di default. E’ possibile fare in modo   
che il segnale SIGHUP non porti alla terminazione di un job, attraverso i seguenti comandi:

* **nohup *programma***: attiva il job associato all’esecuzione di *programma* in modo   
  che sia immune al segnale SIGHUP. Il job viene attivato in background, per cui   
  non ha accesso allo stdin. Inoltre, lo stdout viene ridirezionato su un file di nome *nohup.out*.

disown (*disconosco* il job, togliendolo dalla tabella dei job)

* **disown %*JOB\_ID*:** rende immune al segnale SIGHUP il job identificato dal *JOB\_ID* indicato,  
  rimuovendolo dalla tabella dei job.

**THREAD** *12-11-24*

Un **thread** è un flusso di esecuzione indipendente all’interno di un processo.

Un processo può avere più thread, ognuno dei quali esegue il proprio codice, ha il proprio stack  
e condivide le risorse del processo (tra cui parte della memoria, come lo heap e la sezione contenente le variabili globali), in modo da poter comunicare con gli altri thread del processo.

Quando viene mandato in esecuzione un programma, viene creato un processo con un unico thread, detto main thread, che esegue il codice “principale” del processo (nel caso di un programma C,   
il main). Questo thread può poi creare altri thread dello stesso processo, che possono creare a loro volta altri thread dello stesso processo, e così via. Dato un thread che ne crea un altro,   
chiameremo *thread padre* il thread creante e *thread figlio* il thread creato.

Unix supporta i thread a livello di kernel: è a un thread, dunque, che viene assegnato il controllo   
di un core, e non a un processo.

**LIBRERIA PTHREAD**

La **libreria pthread** permette di creare applicazioni multithread portatili.

Usando questa libreria, ogni thread è identificato da un ID di tipo *pthread\_t*, un tipo che può essere utilizzato solo attraverso apposite funzioni.

Alcune funzioni della libreria utili per poter lavorare con i thread sono:

* **pthread\_t pthread\_self()**: restituisce l’ID del thread che la invoca;
* **int pthread\_create(pthread\_t\* *thread*, const pthread\_attr\_t\* *attr*, void\*(\**start\_routine*)(void\*), void\* *arg*):** permette a un thread di creare un nuovo thread figlio.

Nella variabile puntata da *thread* verrà salvato l’ID del thread creato.

Il thread creato avrà gli attributi specificati nella variabile puntata da *attr.*Passando NULL come *attr*, il thread creato avrà gli attributi di default.

Il thread creato eseguirà la funzione *start\_routine*, dichiarata come una funzione   
con un solo argomento di tipo void\* e tipo di ritorno void\*, con *arg* passato come argomento.  
La variabile puntata da *arg* dovrà essere acceduta dal thread creato, per cui   
non va definita come una variabile dello stack del thread che ha invocato pthread\_create() (altrimenti il thread creato non potrebbe accederci).

La funzione restituisce 0 in caso di successo, un valore diverso da 0 in caso di errore.

* **void pthread\_exit(void\* *retval*)**: termina l’esecuzione del thread che la invoca.   
  Nella variabile puntata da *retval* verranno salvate informazioni sulla terminazione del thread.   
  Queste informazioni possono essere accedute da altri thread dello stesso processo, per cui   
  non vanno salvate in una variabile dello stack del thread che ha invocato pthread\_exit() (altrimenti gli altri thread non potrebbero accederci). Passando NULL come *retval*,   
  queste informazioni non verranno salvate.

Quando un thread termina invocando pthread\_exit(), i suoi thread figli continuano   
con la loro esecuzione. Se invece un thread termina in un altro modo, i suoi thread figli terminano a loro volta.

* **int pthread\_join(pthread\_t *thread*, void\*\* *retval*)**: il thread che la invoca si sospende fintantoché il thread che ha l’ID *thread* non termina.

Nel puntatore puntato da *retval* verrà salvato l’indirizzo della variabile in cui il thread terminato ha eventualmente salvato informazioni sulla sua terminazione attraverso pthread\_exit(). Passando NULL come *retval*, questo indirizzo non verrà salvato.

La funzione restituisce 0 in caso di successo, un valore diverso da 0 in caso di errore.

*Es.*

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NTHREADS 10

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]);

if(ret){

printf(“Errore! il valore restituito da pthread\_create è %d\n, ret);

exit(-1);

}

}

pthread\_exit(NULL);

}

void\* tr\_code(void\* arg){

printf("Ciao! Il valore della variabile puntata da arg è %d\n", \*(int\*)arg);

free(arg);

pthread\_exit(NULL);

}

Definisco la costante NTHREADS che ha, come valore, il numero di thread figli che il main thread creerà.

Nel main dichiaro innanzitutto:

* Un array *tr* di *NTHREADS* elementi di tipo pthread\_t, dove in *tr[i*] verrà salvato l’ID dell’*i*-esimo thread figlio   
  del main thread;
* Un array *args* di *NTHREADS* elementi di tipo int\*, dove *args[i]* sarà l’argomento da passare alla funzione   
  che l’*i*-esimo thread figlio del main thread dovrà eseguire.

A questo punto, per ogni *i* che va da 0 incluso a NTHREADS escluso, alloco nello heap una variabile di tipo int   
e assegno il suo indirizzo ad *args[i],* assegno il valore *i* alla variabile puntata da *args[i]*, e creo un nuovo thread figlio,   
salvando in *tr[i]* l’ID del thread creato. Questo thread avrà gli attributi di default, ed eseguirà tr\_code()   
con *args[i]* passato come argomento.

In caso di errore nella creazione di un nuovo thread figlio, termino il main thread invocando exit(), causando dunque   
la terminazione di tutti i suoi eventuali thread figli già creati. Se invece tutto va bene, termino il main thread   
invocando pthread\_exit(), per cui i suoi thread figli continueranno con la loro esecuzione.

Ogni thread figlio del main thread stampa il valore della variabile puntata da *arg*, dopodiché dealloca questa variabile   
e termina invocando pthread\_exit().

**SEMAFORI DI MUTUA ESCLUSIONE**

Per garantire l’accesso in mutua esclusione ad una risorsa condivisa, la libreria pthread   
permette di creare semafori di mutua esclusione come variabili di tipo *pthread\_mutex\_t*.

Alcune funzioni della libreria utili per poter lavorare su un semaforo di mutua esclusione sono:

* **int pthread\_mutex\_init(pthread\_mutex\_t\* *M*, const pthread\_mutex\_attr\_t\* *attr*)**: inizializza il semaforo di mutua esclusione puntato da *M.*

Il semaforo di mutua esclusione verrà inizializzato con gli attributi specificati   
nella variabile puntata da *attr*. Passando NULL come *attr*, il semaforo di mutua esclusione verrà inizializzato con gli attributi di default.

* **int pthread\_mutex\_lock(pthread\_mutex\_t\* *M*):** effettua la wait() sul semaforo   
  di mutua esclusione puntato da *M*;
* **int pthread\_mutex\_unlock(pthread\_mutex\_t\* *M*):** effettua la signal() sul semaforo   
  di mutua esclusione puntato da *M*.

**VARIABILI CONDITION**

Per garantire la comunicazione tra thread di uno stesso processo, la libreria pthread permette   
di creare variabili condition come variabili di tipo *pthread\_cond\_t*.

Alcune funzioni della libreria utili per poter lavorare su una variabile condition sono:

* **int pthread\_cond\_init(pthread\_cond\_t\* *C*, const pthread\_cond\_attr\_t\* *attr*)**: inizializza   
  la variabile condition puntata da *C.*

La variabile condition verrà inizializzata con gli attributi specificati nella variabile   
puntata da *attr*. Passando NULL come *attr*, la variabile condition verrà inizializzata   
con gli attributi di default.

* **int pthread\_cond\_wait(pthread\_cond\_t\* *C,* pthread\_mutex\_t*\* M*):** il thread che la invoca   
  si sospende sulla variabile condition puntata da *C* ed effettua la signal() sul semaforo   
  di mutua esclusione puntato da *M*. Quando verrà risvegliato, prima di proseguire   
  con la sua esecuzione, il thread rieffettua la wait() sul semaforo di mutua esclusione   
  puntato da M.
* **int pthread\_cond\_signal(pthread\_cond\_t\* *C*):** risveglia un thread a caso tra quelli sospesi sulla variabile condition puntata da *C*.
* **int pthread\_cond\_broadcast(pthread\_cond\_t\* *C*):** risveglia tutti i thread sospesi   
  sulla variabile condition puntata da *C*.

Tipicamente, data una risorsa condivisa, per ogni condizione della risorsa per cui un thread   
può sospendersi viene creata una variabile condition associata.

Le funzioni che usano pthread\_cond\_wait(), pthread\_cond\_signal() o pthread\_cond\_broadcast() tipicamente si dividono in due categorie (spesso coincidenti):

* Quelle che eseguono delle operazioni solo dopo che una certa condizione su una risorsa condivisa è soddisfatta;
* Quelle che cambiano una delle condizioni della risorsa condivisa controllate   
  dalle funzioni sopra.

Queste funzioni, dunque, hanno tipicamente una struttura del genere:

void faiQuandoCondizione(){

pthread\_mutex\_lock(&M);

while(*condizione\_non\_soddisfatta*){

pthread\_mutex\_wait(&C, &M);

}

*operazioni da eseguire*

pthread\_mutex\_unlock(&M);

}

void cambiaCondizione(){

pthread\_mutex\_lock(&M);

*cambio condizione*

pthread\_cond\_broadcast(&C);

pthread\_mutex\_unlock(&M);

}

In una funzione che esegue delle operazioni solo dopo che una certa condizione su una risorsa condivisa è soddisfatta,   
il thread all’inizio si sospende fintantoché non acquisisce l’accesso esclusivo alla risorsa condivisa.   
Fatto ciò, fintantoché la condizione non è soddisfatta, il thread invoca pthread\_mutex\_wait(*&C*, *&M*), per cui si sospende   
sulla variabile condition *C* associata alla condizione in cui si trova attualmente la risorsa condivisa, effettuando la signal() sul semaforo di mutua esclusione *M* associato a tale risorsa condivisa (permettendo così ad altri thread di poter invocare altre funzioni di questa categoria o di quelle che cambiano una condizione).

Questa condizione verrà ricontrollata ogni volta che il thread verrà risvegliato e avrà rieffettuato la wait() sul semaforo   
di mutua esclusione *M*, e quando finalmente sarà soddisfatta, il thread passerà ad eseguire le operazioni della funzione,   
segnalando infine che la risorsa è libera.

**NB:** il controllo della condizione va fatto in un while, e non in un if, perché nella finestra temporale che va   
da quando un thread viene risvegliato a quando questo andrà effettivamente in esecuzione, altri thread   
potrebbero andare in esecuzione, alterando la condizione. Questa condizione, dunque, deve essere ricontrollata   
ogni volta che il thread verrà risvegliato.

In una funzione che cambia una delle condizioni della risorsa condivisa controllate dalle funzioni della categoria sopra,   
il thread all’inizio si sospende fintantoché non acquisisce l’accesso esclusivo alla risorsa condivisa.   
Fatto ciò, il thread cambia la condizione, invoca pthread\_cond\_broadcast(*&C*), risvegliando così tutti i thread sospesi   
sulla variabile condition *C* associata alla condizione in cui la risorsa condivisa poteva trovarsi prima,   
e segnala che la risorsa condivisa è libera.

**NB:** è conveniente invocare pthread\_cond\_signal() o pthread\_cond\_broadcast() prima di segnalare che la risorsa condivisa è libera, così che sia maggiore la finestra temporale in cui la condizione è soddisfatta per i thread che sono stati risvegliati.

**ESEMPIO 1 - PRODUTTORI E CONSUMATORI**

Nel **modello di comunicazione produttori-consumatori** si hanno *n* thread *produttori*,   
che producono informazioni, e *m* thread *consumatori*, che consumano queste informazioni.   
Il passaggio delle informazioni avviene attraverso un buffer.

Ogni produttore, ciclicamente:

* Produce un nuovo valore;
* Attende fintantoché il buffer è pieno per inserirci il nuovo valore.

Ogni consumatore, ciclicamente:

* Attende fintantoché il buffer è vuoto per estrarci un valore;
* Consuma il valore.

Il buffer:

* E’ in grado di contenere un numero limitato di valori;
* E’ condiviso tra produttori e consumatori.

Il buffer condiviso e tutto ciò che lo riguarda può essere realizzato in questo modo:

Definisco una struct *buffer* che ha, come campi:

* Un array *vett* di tipo int di dimensione *BUFFER\_SIZE*, che costituisce   
  il buffer vero e proprio;
* Tre variabili *testa*, *coda*, e *quanti* di tipo int, che conterranno rispettivamente l’indice di testa, l’indice di coda e il numero   
  di elementi presenti nel buffer, per gestirlo in maniera circolare.
* Un semaforo di mutua esclusione *M* associato al buffer;
* Una variabile condition *FULL* associata alla condizione *buffer pieno*.
* Una variabile condition *EMPTY* associata alla condizione *buffer vuoto*.

Dichiaro poi una variabile globale *b* di tipo *buffer*.

struct buffer {

int vett[BUFFER\_SIZE];

int testa, coda;

int quanti;

pthread\_mutex\_t M;

pthread\_cond\_t FULL;

pthread\_cond\_t EMPTY;

};

buffer b;

Il main thread inizializzerà opportunamente il buffer condiviso e tutto ciò che lo riguarda:

int main(){

Inizializzo il semaforo di mutua esclusione *b.M*   
e le variabili condition *b.FULL* e *b.EMPTY*. All’inizio, il buffer è vuoto, dunque assegno 0   
a *b.testa*, *b.coda* e *b.quanti*.

pthread\_mutex\_init(&b.M, NULL);

pthread\_cond\_init(&b.FULL, NULL);

pthread\_cond\_init(&b.EMPTY, NULL);

b.testa = b.coda = b.quanti = 0;

…

}

Ogni thread produttore si comporterà in questo modo:

Il thread produttore, all’inizio, si sospende fintantoché   
non acquisisce l’accesso esclusivo al buffer.   
Fatto ciò, fintantoché il buffer è pieno, il thread invoca pthread\_cond\_wait*(&b.FULL*, *&b.M*). Il thread, dunque,   
si sospende sulla variabile condition *b.FULL*,   
segnalando che il buffer è libero.

Questa condizione verrà ricontrollata ogni volta che il thread   
verrà risvegliato e avrà riacquisito l’accesso esclusivo al buffer,   
e quando finalmente sarà soddisfatta, il thread effettua l’inserimento nel buffer.

Avendo reso il buffer sicuramente non vuoto, il thread invoca pthread\_cond\_signal*(&b.EMPTY*), così da risvegliare un eventuale thread consumatore sospeso sulla variabile condition *b.EMPTY* perché, provando ad estrarre un valore dal buffer,   
questo era vuoto.

Infine, il thread produttore segnala che il buffer è libero.

…

int val = *valore\_da\_inserire*

pthread\_mutex\_lock(&b.M);

while(b.quanti == BUFFER\_SIZE){

pthread\_cond\_wait(&b.FULL, &b.M);

}

b.vett[b.coda] = val;

b.quanti++;

b.coda = (b.coda + 1) % BUFFER\_SIZE;

pthread\_cond\_signal(&b.EMPTY);

pthread\_mutex\_unlock(&b.M);

…

Ogni thread consumatore si comporterà in questo modo:

Il thread consumatore, all’inizio, si sospende fintantoché   
non acquisisce l’accesso esclusivo al buffer.   
Fatto ciò, fintantoché il buffer è vuoto, il thread invoca pthread\_cond\_wait*(&b.EMPTY*, *&b.M*). Il thread, dunque,   
si sospende sulla variabile condition *b.EMPTY*,   
segnalando che il buffer è libero.

Questa condizione verrà ricontrollata ogni volta che il thread   
verrà risvegliato e avrà riacquisito l’accesso esclusivo al buffer,   
e quando finalmente sarà soddisfatta, il thread effettua l’estrazione dal buffer.

Avendo reso il buffer sicuramente non pieno, il thread invoca pthread\_cond\_signal*(&b.FULL*), così da risvegliare un eventuale thread produttore sospeso sulla variabile condition *b.FULL* perché, provando ad inserire un valore nel buffer,   
questo era pieno.

Infine, il thread produttore segnala che il buffer è libero.

…

int val;

pthread\_mutex\_lock(&b.M);

while(b.quanti == 0){

pthread\_cond\_wait(&b.EMPTY, &b.M);

}

val = b.vett[b.testa];

b.quanti--;

b.testa = (b.testa + 1) % BUFFER\_SIZE;

pthread\_cond\_signal(&b.FULL);

pthread\_mutex\_unlock(&b.M);

…

**ESEMPIO 2 - ACCESSO LIMITATO A RISORSA**

Le variabili condition permettono di realizzare un **accesso limitato a una risorsa condivisa**.

Supponiamo di avere una risorsa condivisa che può essere acceduta contemporaneamente da   
un numero massimo di thread.

Per fare ciò, sono necessarie le seguenti costanti e variabili globali:

Definisco:

* Una costante MAX\_T, contenente il numero massimo di thread   
  che possono accedere contemporaneamente alla risorsa condivisa;
* Una variabile globale *n\_users* di tipo int, che conterrà il numero   
  di thread che stanno accedendo contemporaneamente alla risorsa condivisa;
* Un semaforo di mutua esclusione *M* associato alla variabile condivisa *n\_users*;
* Una variabile condition *FULL* associata alla condizione *risorsa acceduta contemporaneamente dal numero massimo di thread possibile.*

#DEFINE MAX\_T 10

int n\_users = 0;

pthread\_mutex\_t M;

pthread\_cond\_t FULL;

Ogni thread si comporterà in questo modo:

Il thread si sospende fintantoché non acquisisce l’accesso esclusivo a *n\_users*, e fintantoché la risorsa condivisa è acceduta contemporaneamente dal numero massimo di thread possibile,   
invoca pthread\_cond\_wait*(&FULL*, *&M*). Il thread, dunque,   
si sospende sulla variabile condition *FULL*, segnalando che *n\_users* è libera.

Questa condizione verrà ricontrollata ogni volta che il thread   
verrà risvegliato e avrà riacquisito l’accesso esclusivo a *n\_users*,   
e quando finalmente sarà soddisfatta, il thread incrementa *n\_users*,   
visto che a breve si aggiungerà ai thread che stanno accedendo contemporaneamente alla risorsa condivisa.

Fatto ciò, il thread segnala che *n\_users* è libera, e accede alla risorsa.

Una volta finito di accedere alla risorsa condivisa, il thread   
si sospende nuovamente fintantoché non acquisisce l’accesso esclusivo a *n\_users*, la decrementa, visto che ha smesso di accedere alla risorsa, e avendo reso il numero di thread che stanno accedendo contemporaneamente alla risorsa sicuramente non quello massimo, il thread invoca pthread\_cond\_signal*(&FULL*), così da risvegliare   
un eventuale thread sospeso sulla variabile condition *FULL* perché, provando ad accedere alla risorsa condivisa, questa era acceduta contemporaneamente dal numero massimo di thread possibile.

Infine, il thread segnala nuovamente che *n\_users* è libera.

…

pthread\_mutex\_lock(&M);

while(n\_users== MAX\_T){

pthread\_cond\_wait(&FULL, &M);

}

n\_users++;

pthread\_mutex\_unlock(&M);

*Accedo alla risorsa*

pthread\_mutex\_lock(&M);

n\_users--;

pthread\_cond\_signal(&FULL);

pthread\_mutex\_unlock(&M);

…

**FILE SYSTEM IN UNIX**

Su UNIX il descrittore di file è detto *i-node*. Tutti gli i-node sono memorizzati in una lista, chiamata *i-list*.

In memoria RAM sono mantenute due tabelle:

* Una **tabella dei file attivi**, contenente, per ogni file per cui esiste almeno una sessione   
  di accesso ancora in corso, una copia del suo i-node;
* Una **tabella dei file aperti di sistema**, contenente, per ogni sessione di accesso   
  ancora in corso a un file:
  + L’I/O pointer associato alla sessione di accesso;
  + Un riferimento nella tabella dei file attivi all’entrata corrispondente al file cui si sta accedendo.

Nella User Structure di ogni processo è salvata poi una cosiddetta *tabella dei file aperti* del processo, contenente, per ogni sessione di accesso ancora in corso a un file da parte del processo,   
un riferimento nella tabella dei file aperti di sistema all’entrata corrispondente a tale sessione.   
L’indice di ogni entrata nella tabella dei file aperti di un processo è detto *file descriptor,*ed è usato per identificare univocamente la sessione di accesso corrispondente a tale entrata   
tra le altre sessioni di accesso del processo.

Ogni sessione di accesso a un file da parte di un processo inizia con la cosiddetta operazione   
di *apertura* del file da parte del processo. Questa comporta:

* Se non esiste una sessione di accesso ancora in corso al file, l’inserimento di un nuovo elemento nella tabella dei file attivi;
* L’inserimento di un nuovo elemento nella tabella dei file aperti di sistema;
* L’inserimento di un nuovo elemento nella tabella dei file aperti del processo.

Ogni sessione di accesso a un file da parte di un processo termina con la cosiddetta operazione   
di *chiusura* del file da parte del processo.

Quando un processo crea un processo figlio, il figlio eredita dal padre una copia della User Structure,   
e quindi anche una copia della tabella dei file aperti. Data dunque una qualunque sessione di accesso ancora in corso a un file da parte del padre, sessione a cui corrisponde una certa entrata nella tabella   
dei file aperti del padre, il figlio eredita una copia dell’entrata, e quindi eredita   
una copia della sessione di accesso. La copia dell’entrata verrà inserita nella tabella dei file aperti   
del figlio allo stesso indice in cui l’entrata originale si trova nella tabella dei file aperti del padre,   
per cui la copia della sessione sarà identificata nel figlio usando lo stesso file descriptor   
usato nel padre per identificare la sessione originale. L’entrata del padre e quella del figlio,   
essendo identiche, punteranno alla stessa entrata della tabella dei file aperti di sistema: i due processi, dunque accederanno al file condividendo lo stesso I/O pointer.

Vediamo alcune primitive per accedere a un file:

* **int open(const char\* *path*, int *flags*):** permette al processo che la invoca di aprire il file   
  avente il *path* indicato, nel modo indicato da *flags*. Restituisce il file descriptor che identifica   
  la sessione di accesso del processo appena iniziata.
* **int close(int fd):** permette al processo che la invoca di chiudere il file, terminando   
  la sessione di accesso del processo identificata dal file descriptor *fd*.
* **ssize\_t read(int fd, void\* buf, size\_t count):** permette al processo che la invoca di leggere   
  al massimo *count* byte nella sua sessione di accesso identificata dal file descriptor *fd*, copiandoli nel buffer puntato da *buf*. Restituisce il numero di byte effettivamente letti   
  in caso di successo, -1 in caso di errore.
* **ssize\_t write(int *fd*, const void\* *buf*, size\_t *count*):** permette al processo che la invoca di scrivere al massimo *count* byte nella sua sessione di accesso identificata dal file descriptor *fd*, copiandoli dal buffer puntato da *buf*. Restituisce il numero di byte effettivamente scritti   
  in caso di successo, -1 in caso di errore.

**PIPE**

I processi possono comunicare tra loro attraverso il meccanismo delle *pipe*.

Una **pipe** è di fatto un file costituito da un buffer in cui è possibile accodare dati, gestiti in modo FIFO. Questo file è costituito da due estremi, dove uno permette solo la lettura, e l’altro permette   
solo la scrittura.

La primitiva **int pipe(int *fd*[])** permette al processo che la invoca di creare una pipe, salvando   
in *fd*[0] il file descriptor che identifica la sessione di accesso del processo all’estremo di lettura   
e in *fd*[1] il file descriptor che identifica la sessione di accesso del processo all’estremo di scrittura.

Abbiamo visto prima che, quando un processo crea un processo figlio, il figlio eredita   
una copia di ogni sessione di accesso ancora in corso ai file da parte del processo padre,   
copia che sarà identificata usando lo stesso file descriptor usato nel padre per identificare   
la sessione originale. Prima di creare un figlio, dunque, se il padre ha creato una pipe,   
il figlio potrà usare la pipe per comunicare con il padre e con gli altri figli, usando gli stessi file descriptor ottenuti dall’invocazione della primitiva pipe() da parte del padre.

La primitiva read(), invocata sull’estremità di lettura di una pipe:

* E’ bloccante se il buffer della pipe è vuoto e c’è almeno un processo che ha aperta   
  l’estremità di scrittura della pipe;
* Restituisce 0 se il buffer della pipe è vuoto e non c’è alcun processo che ha aperta   
  l’estremità di scrittura della pipe.

La primitiva write(), invocata sull’estremità di scrittura di una pipe:

* E’ bloccante se il buffer della pipe è pieno;
* Restituisce un valore negativo se non c’è alcun processo che ha aperta l’estremità di lettura della pipe.

Per evitare deadlock e comportamenti inattesi, dunque, è bene che ogni processo chiuda   
le estremità della pipe delle quali non ha bisogno.

**DUP2**

La primitiva **int dup2(int *target*, int *newfd*)**: duplica il file descriptor *target* copiandolo   
nel file descriptor *newfd*, che viene chiuso se precedentemente aperto.