DA UNIX A LINUX

Inizialmente non vi erano veri e propri **sistemi operativi**, si programmava direttamente l'hardware.
I sistemi operativi permettono all'utente di comunicare con la CPU.
i primi SO erano scritti nel linguaggio nativo(***assembly***) della macchina sulla quale giravano.
Unix fu il primo sistema operativo, venne sviluppato nel *1969* presso i laboratori Bell e fu il primo sistema operativo ad essere scritto in un *linguaggio d'alto livello* (**C**).
Il codice sorgente era disponibile per tutti gli sviluppatori, e si diffuse in maniera rapidissima. Si creò una comunità di sviluppatori volontari che contribuì in maniera collettiva alla crescita e diffusione di Unix.
Un folto gruppo di costruttori informativi costituì la **OSF** (*Open Software Foundation*) con lo scopo di farla diventare la proprietaria del software di base sul quale ognuno avrebbe sviluppato il suo sistema Unix. La conseguenza fu che quasi tutte le workstation avevano sistemi operativi proprietari derivati da Unix.

CARATTERISTICHE PRINCIPALI:

- ***multitasking***: è possibile l'esecuzione di più processi contemporaneamente;
- ***multiutenza***: utilizzo contemporaneo del sistema da parte di più utenti con differenti privilegi;
- ***portabilità:*** grazie all'impiego del linguaggio c che permette di far funzionare Unix su qualsiasi computer;
- ***modularità***: possibilità di aggiungere o rimuovere dinamicamente alcune parti del **kernel**.

Possiamo individuare due componenti principali:

1. il **kernel** che consente di interagire con l'hardware;
2. le **applicazioni** che si interfacciano con il kernel per svolgere determinate funzioni. Esse si possono suddividere in **interpreti dei comandi** *(shell)* **programmi di sistema** e **programmi utente.**

Nel 1985 Richard Stallman fondò la **FSF** (*Free Software Foundation)* con l'obiettivo di creare un sistema operativo simile ad Unix composto interamente da software libero, il sistema creato prese il nome di **GNU**.
La mancanza di un kernel efficiente porto all'abbandono del progetto.
Nel 1991 venne sviluppato un altro kernel come hobby, da uno studente universitario finlandese, **Linus Torvalds**; il quale inizialmente lo chiamò "*Freax*" per indicare che era libero e strano successivamente venne chiamato **Linux** da Ari Lemmke, perché Linux era il nome della cartella del progetto prima che fosse disponibile per il download.

RAGIONI DEL SOFTWARE LIBERO:

1. *è più affidabile ed efficiente*: in quanto è di sviluppato da uno e corretto da tanti e non deve seguire le tempistiche commerciali;
2. *è più sicuro:* in quanto è possibile verificare interamente il codice sorgente;
3. *permette di scegliere il miglior fornitore*: in quanto il cliente non è facilmente ricattabile e i fornitori di software sono favoriti da un mercato aperto;
4. *avvantaggia il mercato*: in quanto gli utenti spendono poco o nulla ei fornitori guadagnano sui servizi.

Rispetto ad un file system di un sistema operativo della famiglia Microsoft Windows nel file system Linux:

- la denominazione dei file/directory può essere effettuata utilizzando 256 caratteri ma in Linux si opera in un contesto **case sensitive**;
- il carattere di separazione delle directory è lo **slash**;
- la gestione dei permessi è più **granulare**;
- **il meccanismo delle estensioni** è differente sia per *modalità* (non è spesso indispensabile il loro utilizzo per definire il tipo di file) sia per *impiego* (è possibile adoperare più di tre lettere).

Differentemente da altre famiglie di sistemi operativi quella Linux rende disponibile una notevole varietà di file system, i più noti sono: ***EXT2, EXT3, EXT4***.
Linux si può utilizzare mediante l'installazione con CD/DVD, mediante l'utilizzo di partizioni o con l'utilizzo della macchina virtuale.

GESTIONE DI BASE DEI SISTEMI LINUX

Possiamo individuare in Linux due parti principali: il **kernel** (il codice che gestisce le risorse presenti sul sistema e le rende disponibili alle applicazioni) e i **programmi di sistema**, uno dei principali programmi di sistema **è l'interprete dei comandi**, che in Linux viene chiamato **shell** (terminale).
La shell è un'interfaccia per gli utenti, cioè un programma che accetta i comandi digitati, li trasforma in istruzioni eseguibili dal computer e li invia al kernel per l'esecuzione.
Noi utilizzeremo la **Bash** (*Bourne Again Shell*).
I **comandi** sono dei programmi che sono contenuti all'interno di alcune directory speciali come le directory */bin e /sbin*. Quando l'utente digita un comando e preme il tasto invio la shell ricerca il comando nelle directory speciali e se lo trova lo esegue, in caso contrario produce un errore del tipo: "***Command not found".***
Mediante il tasto ***<Tab>*** la shell tenta di completare ciò che stiamo digitando nel prompt dei comandi, se non è univoco da un suono di errore, alla seconda pressione visualizza una lista delle possibilità.
Se non ci si ricorda la **sintassi** di un comando si utilizza ***"man"***, se non ci si ricorda **il nome del comando** si può provare ***"apropos"*** con un termine correlato; se vogliamo sapere **cosa fa un comando** si utilizza ***"whatis".***
Esistono diversi tipi di file, i principali sono: i **file regolari**, **le directory**, i **file a caratteri** e i **file a blocchi**.
Un file system composto da directory può essere paragonato ad un albero dove la directory principale (*root o radice*) rappresenta il **tronco** mentre le sottodirectory i **rami**.
Per specificare una directory che si trova all'interno di un'altra directory occorre indicare il **path** cioè il percorso completo, separando le due directory con il simbolo slash.
La "***path***" è un elenco di directory separato da due punti che configura l'insieme dei percorsi di ricerca dei comandi.

LISTA COMANDI:

- ***pwd*** (*print working directory*): visualizza il percorso corrente;
- ***ls*** *(list directory*): stampa il contenuto della cartella in cui ci troviamo;
- ***cd*** *(change directory*): consente di cambiare cartella, bisogna specificare il percorso. Possiamo anche utilizzare "***cd …"*** per andare nella cartella immediatamente superiore;
- ***touch***: crea un file;
- ***mv*** "*nomefile*" "*nuovonomefile*": rinomina il file;
- ***mv*** "*file sorgente*" "*/directory*": sposta il file in una directory;
- **cp**(*copy*): permette di copiare un file, se ci si trova nella cartella del file bisogna solo mettere il nome del file sennò bisogna specificare il percorso di destinazione;
- ***rm*** *(remove*): serve per rimuovere i file;
- ***mkdir*** (*make directory*): crea una nuova cartella;
- ***rmdir***: rimuove directory vuote;
- ***cat***: visualizza il contenuto di un file di testo;
- ***less***: visualizza il file di testo pagina per pagina;
- ***more***: mostra il primo pezzo di testo che sta in terminale premendo spazio va avanti;
- ***tail***: visualizza l'ultima parte di un file di testo si può specificare anche il numero di righe che vogliamo vedere utilizzando -n2 (visualizza le ultime due righe);
- ***head:*** stessa cosa di tail ma visualizza la prima parte;
- ***file***: analizza e mostra il tipo di un file;
- ***strings***: visualizza stringhe di testo all'interno di un file binario.

In Linux il nome dei file può contenere caratteri alfabetici, numerici, caratteri di sottolineatura o di punteggiatura e può essere lungo al massimo 256 caratteri.
Nell'ambito Shell Linux il **prompt** rappresenta un carattere o un insieme di caratteri che possono essere personalizzati dall'utente. Il "." identifica la directory corrente .

Per visualizzare a video **l'elenco** dei file dentro una cartella, come detto prima, si utilizza il comando *ls* , quest'ultimo accetta varie opzioni come :

- ***-a***: mostra anche le cartelle nascoste ;
- ***-l***: permette di visualizzare alcuni dettagli importanti di ciascun file;

Con *-l* visualizziamo una stringa del tipo : ***"-rw-r--r-x 1 lau utenti 207 Feb 20 11:55 file"***

Il nome del file è l'ultima informazione presente , il primo carattere indica il **tipo di file** ne abbiamo 8 possibili:

- **-**: File regolare ;
- ***d:*** directory ;
- **l**: link ;
- ***b***: Periferica a blocchi con buffer;
- ***c:*** periferica caratteri con buffer ;
- ***u:***periferica a caratteri senza buffer ;
- ***p***: pipe FIFO;
- ***s***: socket ;

Gli altri 9 caratteri successivi al primo indicano i **permessi di accesso ai file** , segue **il numero di link**, il **nome del proprietario del file**, **il nome del gruppo** a cui appartiene il proprietario, le **dimensioni in byte**, **la data e l'ora** dell'ultima modifica .

I permessi sono riferiti a tre categorie di utenti: il **proprietario del file**, il **gruppo di utenti** a cui appartiene il proprietario ed infine **tutti gli altri utenti** che posseggono un account sul sistema.

Un file può essere letto o visualizzato se ne si possiede il **permesso in lettura** (***r***), può essere sovrascritto cancellato o modificato se ne si possiede il **permesso in scrittura** (***w***), può essere eseguito se ne si possiede il **permesso in esecuzione** (**x**). I permessi sono divisi in gruppetti da tre, i primi tre sono relativi al proprietario, i successivi tre al gruppo, e infine gli ultimi tre agli altri utenti. Ogni terna può contenere i caratteri ***"-","r","w","x"***, **-** disabilita un permesso .

Nell'esempio di prima il proprietario può leggere e scrivere ma non eseguire, il gruppo può solo leggere e gli altri utenti possono leggere ed eseguire , le terne sono sempre composte così "***rwx***" se non vi è il permesso si mette il ***"-"***.

Per *modificare i permessi di accesso ai file* occorre usare il comando ***chmod***, tale comando accetta due argomenti : un elenco di variazione di permessi ed un *elenco di file sui quali agire*. Per abilitare un permesso occorre usare il simbolo "**+**" affianco al tipo di permesso che si desidera modificare.

Per specificare la tipologia di utenti ai quali abilitare o disabilitare dei permessi esistono tre simboli : ***"u", "g"*** e "***o***", che corrispondono rispettivamente al user cioè il proprietario al gruppo e a tutti gli altri utenti.

*Esempio: "**chmod g+w-x dati**"*

Quando non viene specificata una categoria di utenti, i permessi vengono modificati per tutti, la stessa cosa accade se si mette la "***a***" che indica all.

Oltre questo metodo per modificare i permessi esiste quello delle **maschere binarie**: il suo vantaggio è la possibilità di cambiare i permessi utilizzando un solo comando.

In una maschera binaria le tre categorie di utenti ossia le tre terne di simboli diventano tre cifre in **formato ottale** , nel senso che ciascuna categoria è rappresentata da un numero in base 8. Il numero 0 corrisponde ad un permesso disabilitato numero 1 invece lo abilita :

- 0 - 000 - ---
- 1 - 001 - --x
- 2 - 010 - -w-
- 3 - 011 - -wx⁄
- 4 - 100 - r—
- 5 - 101 - r-x⁄
- 6 - 110 - rw-
- 7 - 111 – rwx

Esempio: per abilitare tutti i permessi al proprietario del file "dati" e permettere a tutti gli altri di poterlo solo leggere ed eseguire si scrive: ***"chmod 755 dati"***

Come già detto le directory sono esse stesse dei file anche se speciali, conseguentemente è possibile specificare i permessi anche per esse, ma sono diversi.
Il **permesso di lettura** su una **directory** consente di elencare tutti i file in essa contenuti , il **permesso di scrittura** consente di aggiungere nuovi file al suo interno, il **permesso di esecuzione** consente di accedere all'interno della directory. Inoltre, inizialmente un file creato all'interno di una directory eredita i permessi che possiede la directory.
Se un file leggibile, scrivibile ed eseguibile viene posto all'interno di una directory che non è accessibile (cioè non ha il permesso di esecuzione) il file stesso diventa inaccessibile.

Per accedere ad un file *non è sufficiente abilitarne i permessi* ma occorre che la directory che lo contiene, così come la directory genitrice e tutte le altre directory siano abilitate in esecuzione.
Dato che solo il proprietario del file può *modificare i permessi* solo lui può cedere il controllo del file ad un' altro utente utilizzando il comando ***"chown***".
Esempio: ***"chown franca guida.html"***
Allo stesso modo è possibile cambiare il gruppo del file con ***"chgrp"***.
Questi comandi non hanno effetto su file system di tipo **FAT**, in quanto non supporta i permessi sui file come Linux.

LINK FISICI E SIMBOLICI

Il collegamento tra il nome del file e l'indirizzo fisico dove si trova il file viene definito il **link** ,è possibile creare un link ad un file mediante il comando ***"ln"***, questo comando quindi permette di creare ulteriori link ad un file cioè creare ulteriori nomi per referenziarlo.
Esempio : abbiamo il file uno che ha un solo link:
"***rwrwrw 1 lau utenti 553 Jul 18 1994 /home/lau/file1***"
Uso il comando ln:
"ln file1 data1" (creo il secondo nome data1 per file1), quindi adesso avremo:
"***rwrwrw 2 lau utenti 553 Jul 18 1994 /home/lau/file1"***
"***rwrwrw 2 lau utenti 553 Jul 18 1994 /home/lau/data1***"
Bisogna specificare che data1 non è la copia di file1, infatti se si modifica uno viene modificato anche l'altro invece se si cancellasse file1sarà sempre possibile far riferimento al file usando data1.
Un file viene cancellato definitivamente solo quando vengono eliminati tutti i suoi link.
Questo appena visto è un **link fisico** e può ad esempio essere utile per far riferimento ad un file da una directory diversa da quella all'interno della quale si trova il file.
Ad esempio, file1 potrebbe risiedere nella directory **/home/lau** mentre il link data1 potrebbe risiedere nella directory **/home/franca**. Si potrà accedere al file1 dalla directory /home/lau, oppure dalla directory /home/franca usando il riferimento data1.
Se è possibile creare un numero arbitrario di nomi per un unico file, non è possibile farlo con le directory. Non è possibile creare un link fisico ad una directory, ma possiamo crearne uno **simbolico**.
Un **link simbolico** non è un nome aggiuntivo ma è un file speciale che contiene al suo interno il **puntatore al file**, cioè il percorso da fare per raggiungere il file. Per creare un link simbolico usiamo il comando "**ln"** con l'opzione ***"-s"***:
"***ln -s nome_file nome_link***"
Questo comando crea un file nome_link che punta al file (o alla directory) nome_file.

CONFIGURAZIONE DELLA SHELL

La shell BASH ha delle impostazioni standard generali per tutti gli utenti, ma ogni utente può avere una versione personalizzata della propria shell modificando opportunamente alcuni file di configurazione. All'interno di questi file sono presenti delle variabili che contengono dei valori predefiniti.
Una **variabile** è un'area di memoria alla quale viene assegnato uno specifico valore. Tale area viene creata al momento del login e viene distrutta al momento del logout.

L'utente può creare delle variabili personali e può modificare alcune di quelle predefinite dal sistema. Per convenzione le variabili della shell sono definite **con nomi costituiti da lettere maiuscole**. Ad esempio, la variabile HOME definisce il percorso della directory home dell'utente.

Nella documentazione originale di bash si utilizza il termine "**parametro**" per identificare diversi tipi di entità:

- parametri posizionali;
- parametri speciali;
- variabili di shell.

Ci riferiremo con il termine "parametro" solo ai primi due tipi di entità.

L'elemento *comune tra i parametri e le variabili è il modo con cui questi devono essere identificati* quando si vuole leggere il loro contenuto occorre il simbolo "**$**" davanti al nome (o al simbolo) dell'entità in questione, mentre per assegnare un valore all'entità (sempre che ciò sia possibile), questo prefisso non deve essere indicato.

Una variabile è definita quando **contiene un valore**, compresa la stringa vuota. L'assegnazione di un valore si ottiene con una dichiarazione del tipo seguente:

"<nome-di-variabile>=[<valore>]"

Il nome di una variabile può contenere lettere, cifre numeriche e il segno di sottolineatura, ma il primo carattere non può essere un numero. Se non viene fornito il valore da assegnare, si intende la stringa vuota. Una variabile ha vita solo all'interno della shell nella quale viene creata, pertanto cambiando shell, il sistema non riconosce più tale variabile. Affinché una variabile sia visibile all'interno di qualsiasi shell, occorre '**esportarla**' mediante il comando "***export***". Ad esempio:

MYVAR="testo di prova" ***export*** MYVAR (ora la variabile MYVAR è visibile da qualsiasi shell)

Per visualizzare le variabili definite si può usare il comando "***set***", mentre per visualizzare le variabili di ambiente occorre usare il comando "***env***". Le variabili di ambiente sono tutte quelle variabili predefinite all'interno della shell che troviamo normalmente all'interno di un sistema Linux. Per visualizzare il contenuto di una variabile invece, occorre usare il comando "***echo***" insieme all'operatore '$'. L'operatore $ serve per far riferimento al contenuto di una variabile. Ad esempio il comando:

"***echo $MYVAR***" visualizza a video il contenuto della variabile MYVAR.

GESTIRE IL SISTEMA MEDIANTE SHELL

Il comando "***tar***" è stato creato originariamente per archiviare file e directory su nastro (tar, infatti, sta per "*Tape Archive*" cioè archivio su nastro) ma in realtà con tale comando è possibile creare degli archivi su qualsiasi dispositivo. Ecco perché ancora oggi è molto usato anche su dispositivi diversi dai nastri. Con il comando tar è possibile creare degli archivi che contengono file ma anche intere directory. È possibile modificare un archivio creato con il comando tar aggiungendo nuovi file o eliminandone alcuni già presenti. Il comando tar, perciò, è molto utile per *effettuare dei backup*. Un archivio creato con il comando tar ha solitamente come estensione '**.tar'.**

La sintassi del comando tar è: tar opzioni nomearchivio.tar file e/o directory. Il comando:

"***tar cf mioarchivio.tar mydir***" crea un archivio chiamato mioarchivio.tar aggiungendo all'interno tutti i file presenti sotto la directory mydir. Per estrarre tutti i file e le directory contenute nell'archivio mioarchivio.tar occorre usare lo stesso comando tar ma con le **opzioni di estrazione:**

"***tar xf mioarchivio.tar***"

Il comando tar non esegue alcuna compressione sui file aggiunti nell'archivio, tuttavia è possibile creare un archivio contenente file e directory e comprimerlo in modo da occupare meno spazio. Per far ciò occorre usare l'opzione "**z**". L'opzione z chiama il comando "***gzip***" che effettua la compressione dei file:

"***tar czf mioarchivio.tar.gz mydir***"

Il file così creato verrà chiamato mioarchivio.tar.gz. Saranno cioè presenti 2 estensioni: **.tar e .gz**. Un file con estensione .gz è un file compresso e lo si può decomprimere usando il comando "**gzip**" con l'estensione "***-d***" oppure si può usare il comando "***gunzip***".

Per estrarre i file e le directory contenute nel file mioarchivio.tar.gz occorre prima decomprimere l'archivio con il **comando gzip** e successivamente estrarre i file dall'archivio con **il comando tar.**

Per ricercare un file si utilizza il comando "***find***". Ad esempio il comando:
"***find . name nomefile print***" Ricerca il file nomefile a partire dalla directory corrente (.) in tutte le sottodirectory e stampa sul display tutti i risultati. Si possono usare i caratteri jolly.
Ad esempio il comando: "***find . name "*.txt" print"*** Ricerca tutti i file con estensione .txt file nomefile a partire dalla directory corrente (.) in tutte le sottodirectory e stampa sul display tutti i risultati.
Digitare frequentemente dei comandi complessi contenenti numerosi opzioni può essere alla lunga fastidioso, perciò può essere conveniente utilizzare dei sinonimi più facili da ricordare e da digitare punto si utilizza il comando "***alias***", la cui sintassi è:
"***alias montaWindows='mount t vfat /dev/sda1 /mnt/mydir'***" Questo comando monta la prima partizione presente sul terzo disco alla directory /mnt/mydir (/dev contiene i file che identificano i vari device).
Un alias può essere utile anche per modificare il comportamento predefinito di alcuni comandi. Ad esempio, il comando "***rm***" cancella i file irreversibilmente senza chiedere conferma, a meno che non venga usata l'opzione **-i**. possiamo creare un alias di rm che include tale opzione:
"***alias rm=' rm -I'***"
Per eliminare un alias è possibile usare il comando "***unalias rm***" dove rm e il nome dell'alias.
I caratteri '*******', '****?***' vengono definiti anche **caratteri jolly** o anche **wildcards**. Si tratta di caratteri che vengono usati dalla shell per espandere il nome di un file. Il carattere '******* significa qualsiasi carattere o qualsiasi sequenza di caratteri. Digitando una serie di caratteri e terminando la serie con il carattere *, la shell individuerà tutti i file all'interno della directory corrente i cui nomi iniziano con la serie di caratteri digitati e terminano con una sequenza qualsiasi di caratteri.
Digitando solo il carattere *, la shell individuerà semplicemente tutti i file all'interno della directory corrente.
Con il termine **quoting** si vuole fare riferimento all'azione di racchiudere parti di testo all'interno di delimitatori (virgolette o apici) per evitare confusione nei comandi, o per poter utilizzare un simbolo che altrimenti avrebbe un significato speciale. Il metodo del quoting viene quindi usato per togliere il significato speciale che può avere un carattere o una parola per la shell.
Ci sono tre meccanismi di quoting:

1. il carattere di escape (rappresentato dalla barra rovescia);
2. gli apici semplici ;
3. gli apici doppi o virgolette ;

Il metodo utilizzato più comunemente per "*neutralizzare*" i caratteri speciali consiste nel racchiudere la stringa che li contiene tra apici doppi.
Ad esempio: lo spazio è un carattere problematico nei nomi delle cartelle e quando si vuole accedere a una cartella il cui nome contiene degli spazi, occorre mettere il nome della cartella tra apici doppi.
Esempio: "***cd la mia cartella***" restituisce errore, mentre "***cd "la mia cartella"***" ci permette di raggiungere la cartella desiderata. Non sempre le virgolette risolvono il problema.
Ad esempio, provando a scrivere "***echo "$PATH"***", invece di "***echo $PATH***" (senza virgolette), ci si aspetta di avere due risultati diversi, invece il risultato è lo stesso. Se si volesse stampare la stringa $PATH è necessario effettuare un ulteriore escape del carattere speciale $ e scrivere quindi echo ***"\ $PATH***", un po' come succede nel linguaggio C.
In generale il **backslash** effettua l'escape di un singolo carattere speciale, quindi nell'esempio precedente avremmo potuto scrivere anche ***"cd la\ mia\ cartella***".
A volte può essere necessario utilizzare l'output di un comando come input per altri comandi. In questo caso può essere utile la **command substitution** che permette di istruire la shell a interpretare una stringa come fosse un comando, ad eseguirlo e ad utilizzarne il risultato.
La command substitution si effettua racchiudendo la stringa tra ` ` (Alt Gr + ').
Esempio: supponiamo di voler creare una cartella con il nome "**backup data_odierna**"; per ottenere la data in questione automaticamente possiamo utilizzare il comando: "***mkdir "backup `date `"***".

STANDARD INPUT, STANDARD OUTPUT E STANDART ERROR

Quando Unix, ed a seguire Linux, vennero sviluppati, si volle mantenere **indipendente la struttura logica dei file dalla loro conformazione fisica**. Per tale ragione venne adottato un approccio, secondo il quale qualsiasi dispositivo fisico può essere logicamente considerato un file, cioè una sequenza continua di byte. Tale concetto logico si estende anche alle operazioni di input/output, infatti in tali operazioni i dati sono organizzati come sequenze continue di byte.
I dati introdotti in input con la tastiera vengono inseriti all'interno di un canale costituito da una sequenza continua di byte; allo stesso modo i dati in output, cioè quelli visualizzati a video, vengono inseriti all'interno di un altro canale. Tali canali sono definiti, rispettivamente, **standard input** e **standard output**. Tipicamente, la tastiera costituisce il canale standard input, mentre lo schermo del monitor rappresenta il canale standard output. Per cui, qualsiasi comando immesso introduce nello standard input dei valori e produce dei risultati attraverso lo standard output.
Considerando che un comando immesso mediante lo standard input *può generare un messaggio di errore*, è stato previso un ulteriore canale; esso prende il nome di **standard error**. Essendo un canale che produce qualcosa in output, per impostazione predefinita *il suo output viene visualizzato insieme a quello dello standard output*. In altre parole, se non si modifica tale impostazione, sia il risultato di un comando, sia i suoi eventuali messaggi di errore, vengono visualizzati a video.
Linux ci permette di redigere uno di questi canali standard da o verso un file, questo permette, ad esempio, di *dirigere l'output prodotto da un comando non a video ma all'interno di un file*. Oppure al posto di immetterli con la tastiera *è possibile inviare in input ad un comando dei valori che vengono presi da un file*.
Tali operazioni di redirezione vengono effettuate utilizzando gli operatori di redirezione **'<','>'** e '**>>**'. Dove '**<**' è l'operatore di **redirezione dell'input standard**, e **'>**' e '**>>**' sono gli operatori di **redirezione dell'output standard**.
Esempio: se voglio salvare all'interno di un file la lista prodotta dal comando "***ls***" basta utilizzare il comando "***ls > disney.txt***", in questo caso la lista prodotta da ls verrà salvata nel file disney.txt.
Eseguendo nuovamente questo comando il contenuto precedente del file viene azzerato e vengono scritti solo i valori prodotti dalla lista , per aggiornare il file senza distruggere il contenuto precedente si può usare l'operatore di redirezione dello standard output '**>>**', in quanto questo operatore non distrugge il file ma accoda alla fine del file il risultato prodotto dal comando.
Anche lo standard error può essere ridiretto così come lo standard input e lo standard output. Il carattere che viene utilizzato per questo tipo di redirezione è **'2>**'.
Supponiamo di *voler visualizzare l'elenco di file contenuto all'interno di una directory e che questa visualizzazione produca un elenco interminabile di file*, è possibile in questo caso utilizzare il comando "***more***" che accetta in input dei dati e visualizza a video solo una pagina per volta , premendo un tasto qualsiasi il comando more visualizza la pagina successiva. Come è possibile collegare l'output prodotto dal comando ls al comando more? È possibile utilizzare l'operatore di **pipe** '|'.
Usando l'operatore di pipe '|' comunichiamo alla shell che è necessario collegare l'output standard del primo comando all'input standard del secondo comando.

PROCESSI E JOB

Un *singolo programma, nel momento in cui viene eseguito*, **diventa un processo**. La nascita di un processo può avvenire solo tramite una richiesta da parte di un altro processo già esistente, si forma quindi una sorta di gerarchia fra processi, organizzata ad albero. Il processo root, cioè quello che genera tutti gli altri è **init**, il processo init è a sua volta attivato direttamente dal kernel.
Un comando impartito attraverso una shell può generare più di un processo, quando si vuole fare riferimento ai processi generati da un singolo comando, dato attraverso una shell, si parla di **job** non di singoli processi. Attraverso alcune shell è possibile gestire i job.
Non si deve *confondere un job di shell con un processo*. **Un processo** *è un singolo eseguibile messo in funzione*. **Un job di shell** *rappresenta tutti i processi che vengono generati da un singolo comando* impartito tramite la shell stessa.

L'attività di un job può avvenire in **foreground** (in primo piano) o in **background** (in sottofondo). Nel primo caso, il job *impegna la shell e quindi anche il terminale*, mentre nel secondo la shell è libera da impegni e così anche il terminale.
Un programma viene avviato esplicitamente come job in background quando alla fine della riga di comando viene aggiunto il simbolo '**&**'. *Per esempio*: "***find / name "*.txt" > output.txt 2> error.txt &"***
Dopo l'avvio di un programma come job in background, la shell *restituisce una riga contenente* il **numero del job e il numero del primo processo** generato da questo job (PID).
Per esempio: ***"[1] 173***" Rappresenta il job numero 1 che inizia con il processo 173.
Se viene avviato un job in background e questo a un certo punto ha la necessità di visualizzare dati attraverso lo standard output o lo standard error e questi non sono stati ridiretti, si ottiene una segnalazione simile alla seguente:
"[1]+ Stopped (tty output) emettidati"
Nell'esempio, il job avviato con il comando emettidati si è bloccato in attesa di poter emettere dell'output.
Stessa cosa accade se viene avviato un job in background e questo ad un certo punto ha la necessità di ricevere dati dallo standard input e questo non è stato ridiretto, si ottiene una segnalazione simile alla seguente:
"[1]+ Stopped (tty input) ricevidati"
Se è stato avviato un job in foreground e si desidera sospenderne l'esecuzione, si può inviare attraverso la tastiera, il carattere '***susp***' che di solito si ottiene con la combinazione [Ctrl+z]. Il job viene sospeso e posto in background. Quando un job viene sospeso, la shell genera una riga come nell'esempio seguente:
"***[1]+ Stopped pippo***" dove il job pippo è stato sospeso.
Il comando di shell "***jobs***" permette di conoscere l'elenco dei job esistenti e il loro stato. Per poter utilizzare il comando jobs occorre *che non ci siano altri job in esecuzione in foreground*, di conseguenza, quello che si ottiene è **solo l'elenco dei job in background**.
L'opzione ***"-l***" del comando jobs permette di conoscere anche il numero **PID** del processo leader di ogni job. L'elenco di job ottenuto attraverso il comando jobs mostra in particolare il simbolo '+' a fianco del numero del job attuale, ed eventualmente il simbolo '-' a fianco di quello che diventerebbe il job attuale se il primo termina o viene comunque eliminato. Il job attuale è quello a cui si fa riferimento in modo predefinito tutte le volte che un comando richiede l'indicazione di un job e questo non viene fornito.
Di norma si indica un job con il suo numero preceduto dal simbolo '**%**', ma si possono anche utilizzare i metodi elencati di seguito:

Simbolo	Descrizione
%n	Il job con il numero indicato dalla lettera n
%<stringa>	Il job il cui comando inizia con la stringa indicata
%?<stringa>	Il job il cui comando contiene la stringa indicata
%%	Il job attuale
%+	Il job attuale
%-	Il job precedente a quello attuale

Il comando "***fg***" porta in foreground un job che prima era in background. Se non viene specificato il job su cui agire, si intende quello attuale. Il comando "***bg***" permette di fare riprendere (in background) l'esecuzione di un job sospeso. Ciò è possibile solo se il job in questione non è in attesa di un input o di poter emettere l'output. Se non si specifica il job, si intende quello attuale.
Il comando "***kill***" termina il job, eventualmente inviando un segnale al processo.
"***kill [-s <segnale> | -<segnale>] [job]***"

Signal name	Signal value	Effect
SIGHUP	1	Hangup
SIGINT	2	Interrupt from keyboard
SIGKILL	9	Kill signal
SIGTERM	15	Termination signal
SIGSTOP	17,19,23	Stop the process

"***user$ ps***" Fornisce i processi dell'utente associati al terminale corrente:

[studente@louey studente]$ ps
PID TTY TIME CMD
1909 pts/2 00:00:00 bash
1962 pts/2 00:00:00 ps

PID è il PID del processo; **TTY** è il terminale (virtuale); **TIME** è il tempo di CPU utilizzato; **CMD** è il comando che ha generato il processo.

Per ottenere il nome del terminale corrente si usa il comando "***tty***".

MANIPOLARE FILE DI TESTO

Il termine **espressione regolare** (*regular expression*), spesso abbreviato con **regexp, regex** o **RE** indica una sequenza di simboli (stringa). Una espressione regolare può essere definita come una funzione che prende in input una stringa, e restituisce in output un valore del tipo booleano, a seconda che la stringa segua (true) o meno (false) un certo pattern.

In pratica, tramite le RE, è possibile definire delle regole rivolte alla creazione di pattern di ricerca. Le RE vengono utilizzate da diversi comandi (o linguaggi) che effettuano il cosiddetto **pattern matching** (grep, sed, awk, perl) e si avvalgono di **metacaratteri e modificator**.

Metacarattere	Definizione
.	Singolo carattere
^	Identifica l'inizio della riga
\	Neutralizza il significato del metacarattere seguente
$	Identifica la fine della stringa
[]	Identifica qualsiasi carattere indicato tra parentesi
[^]	Identifica tutti i caratteri non specificati
[0-9]	Identifica ogni carattere compreso nell'intervallo

Modificatore	Definizione
?	1 volta o mai
+	Da 1 a N volte
*	Da 0 a N volte

Il comando "***grep***"(general regular expression print) è comunemente utilizzato per ricercare le occorrenze di una o più parole in una serie di file.

Alcune importanti opzioni di grep sono:

- ***-i***: ignora la distinzione fra lettere maiuscole e minuscole;
- ***-l***: fornisce la lista dei file che contengono il pattern;
- ***-n***: le linee in output sono precedute dal numero di linea;
- ***-v***: restituisce le linee che non contengono il pattern;
- ***-w***: restituisce le linee contenenti il pattern come parola completa;
- ***-x***: restituisce le linee coincidenti esattamente con il pattern;

Esempi:

1. "***grep '^1' list.txt***": ricerca in list.txt le righe che iniziano con 1;
2. "***grep '1$' list.txt***": ricerca in list.txt le righe che finiscono con 1;
3. "***grep '^2[234]' list.txt***": ricerca le righe che iniziano con 22,23,24;
4. "***grep '^Linux$' list.txt***": ricerca le righe che contengono SOLO la parola Linux;
5. "***grep -c '^$' list.txt***": visualizza il numero di righe vuote in list.txt;
6. "***grep '^[^0-9]' list.txt***": visualizza le righe che NON iniziano con un numero;
7. "***grep '\<[Ll]inux>' list.txt***": visualizza le righe che contengono la parola singola Linux o linux, ma non visualizza quelle con, per esempio, LinuxOS (la sequenza \< indica il punto d'inizio di una parola, così come \> quello in cui termina una parola).

"***sort***" Tratta ogni linea come una collezione di vari campi separati da delimitatori (default: spazi, tab ecc.). Per default avviene in base al primo campo e alfabetico.

Opzioni:

- ***-b***: ignora eventuali spazi nelle chiavi di ordinamento;
- ***-f:*** ignora le distinzioni fra maiuscole e minuscole;
- ***-n***: considera numerica la chiave di ordinamento;
- ***-r***: ordina in modo decrescente;
- ***-o file***: invia l'output al file file;
- ***-t s:*** usa s come separatore di campo;
- ***-k s1, s2***: usa i campi da s1 a s2-1 come chiavi di ordinamento.

ORDINE NUMERICO CON CHIAVE:

$ cat data.txt	***$ sort -k 2 -n data.txt***
Gamma 10	***Gamma 10***
Alfa 300	***Box 60***
Cigno 200	***Cigno 200***
Box 60	***Alfa 300***

Il comando "**tr**" (*character translation*) permette di modificare o cancellare caratteri da uno stream di input. La sua sintassi è la seguente: "***tr [OPTION]... SET1 [SET2]***".

Esempi:

1. ***"tr a-z A-Z"***: converte minuscole in maiuscole;
2. ***"tr -c A-Za-z0-9 ' '"***: sostituisce i caratteri non alfanumerici (***-c*** = complement) con spazi;
3. "***tr -cs A-Za-z0-9 ' '***": opera come il precedente, comprimendo però gli spazi adiacenti in un unico spazio (***-s***=squeeze);
4. "***tr -d mystring***": cancella i caratteri contenuti nella stringa mystring;

L'utilizzo di molti comandi, come per esempio "***tr***", si avvale solitamente del meccanismo di **pipe** | descritto in precedenza. Nel caso della conversione da minuscolo a maiuscolo di tutti i caratteri di un file di testo denominato testo.txt, il comando da impiegare sarà il seguente:

"***cat testo.txt | tr a-z A-Z***": l'effetto del comando sarà la visualizzazione a video (standard output) del contenuto del file testo.txt, dopo che questo è stato processato da tr.

Qualora volessimo rendere permanenti le modifiche apportate al file, possiamo redirezionare l'output del comando su un file: "***cat testo.txt | tr a-z A-Z > nuovofile.txt***".

Il comando "***wc***" fornisce delle informazioni circa il contenuto di un file (o di uno stream in input) specificato come argomento. Esso è in grado di conteggiare la *dimensione in byte del file, il numero di parole contenute ed il numero di linee che lo compongono*. Eseguito senza specificare alcun argomento oltre al nome del file, esso restituisce tutte le precedenti informazioni sul file, nel seguente ordine: RIGHE | PAROLE | BYTE.

Esempi:

1. ***"wc -c nomefile"***: visualizza la dimensione in byte di nomefile;
2. ***"wc -w nomefile***": visualizza il numero delle parole in nomefile;
3. "***wc -l nomefile***": visualizza il numero delle righe di nomefile;
4. "***wc nomefile***": visualizza tutte le informazioni (righe, parole e dimensione in Byte) di nomefile.

Il comando "***cut***" permette di estrarre una parte di un file che potrà essere utilizzata per un altro scopo.

Opzioni generali:

- ***-c lista_caratteri***: posizioni dei caratteri da selezionare (il primo carattere è in posizione 1);
- ***-d delimitatore:*** indica delimitatore di campo (tab di default);
- ***-f lista_campi:*** campi da selezionare (il primo campo è 1).

Esempi:

1. "***cut -c3 nomefile***": visualizza il terzo carattere di ogni riga di nomefile;
2. "***cut -c3-6 nomefile***": visualizza i caratteri dal terzo al sesto di ogni riga di nomefile;
3. "***cut –f3 nomefile***": visualizza il terzo campo di ogni riga di nomefile;
4. "***cut –f3-5 nomefile***": visualizza dal terzo al quinto campo di nomefile.

Il comando "***head***" restituisce la parte iniziale del contenuto di un file fornito in input. Qualora non venissero specificate delle opzioni, esso considera il file in input come un file di testo, e restituisce le sue prime 10 righe.

Esempi:

1. "***head nomefile***": visualizza le prime 10 righe di nomefile;
2. "***head -n2 nomefile***": visualizza le prime 2 righe di nomefile;
3. "***head -c100 nomefile***": visualizza i primi 100 byte di nomefile;
4. "***head -vc100 nomefile***": visualizza i primi 100 byte di nomefile, mostrando all'inizio (header) il nome del file processato (***v***=verbose).

Il comando "***tail***" restituisce la parte finale del contenuto di un file fornito in input. Qualora non venissero specificate delle opzioni, esso considera il file in input come un file di testo, e restituisce le sue ultime 10 righe.

"***uniq***" Rimuove le righe duplicate consecutive da un file, "***cmp***" compara due file di qualunque tipo, manda in out il byte ed il numero di riga della prima differenza riscontrata.

Il simbolo '**~**' identifica la home dell'utente.

COMPILAZIONE C IN AMBIENTE LINUX

C è un linguaggio di programmazione di alto livello. Inizialmente venne sviluppato da **Dennis Ritchie** presso i *Bell Labs* alla fine degli **anni 60** nel contesto dei laboratori della AT&T. L'idea era quella di impiegarlo *per la scrittura di UNIX*, un sistema operativo sviluppato precedentemente da Ritchie insieme a Ken Thompson, per il quale era stato adoperato il linguaggio Assembly.

Unix interamente sviluppato in C venne rilasciato nel 1972, mentre la pubblicazione di un libro sul linguaggio C avvenne nel 1978, seguita da una sua rapida diffusione in molti ambienti, spesso con differenti dialetti. Alla luce di questo grande numero di dialetti originati dal C, è nata la necessità di definire uno standard, il primo dei quali è stato l'ANSI nel 1989, dal quale si giunge alle attuali versioni.

Il C è un linguaggio di programmazione **compilato**, occorre quindi impiegare un compilatore che trasformi il *codice sorgente del programma realizzato in un formato eseguibile* dal sistema operativo.

È un linguaggio relativamente a basso livello, in quanto consente di generare dei programmi molto efficienti in termini di performance.

La gestione della memoria *rappresenta un punto a favore* di questo linguaggio, in quanto esso consente al programmatore di **allocare/deallocare** blocchi di memoria.

Dispone di numerosi tipi di dati, i quali consentono di definire strutture dati anche molto sofisticate. Differentemente da altri linguaggi (di più alto livello) come C++ e Java, non dispone di elementi come classi, oggetti, eventi, ecc. In quanto esso basa il suo funzionamento sulle funzioni *(linguaggio procedurale)*.

Iniziamo col dire che ogni programma C si compone di variabili e funzioni e fra queste ne esiste una speciale definita **main**, dalla quale inizia l'esecuzione e che quindi deve essere presente in ogni programma. Le due parantesi (senza nulla all'interno) dopo il nome della funzione main ( ), indicano che la funzione non prende alcun parametro in input.

Per produrre un eseguibile da un programma C sono necessari **tre passi**, compiuti dai seguenti moduli: *compilatore, assembler, linker*:

1. **compilatore**(fase preceduta *dall'invocazione del preprocessore*): converte il codice sorgente in linguaggio assembly (linguaggio di basso livello);
2. **assembler**: converte il codice in linguaggio assembly in linguaggio macchina (direttamente eseguibile dal processore);
3. linker: "collega" il codice macchina prodotto dalla fase precedente a quello delle funzioni di libreria utilizzate nel programma (e.g., printf).

Questi 3 passi sono spesso eseguiti in unico passo, mediante un comando.

Il codice sorgente in linguaggio C viene convenzionalmente salvato in file con estensione **.c**, supponendo di avere un codice nel file "ciao_mondo.c", per compilarlo useremo il comando "***gcc***" (GNU C compiler): "***gcc ciao_mondo.c***"

Il risultato del precedente comando sarà un file binario "**a.out**" che rappresenta l'eseguibile creato a partire dal codice sorgente.

Per eseguirlo scriviamo ***"./a.out***".

Per scegliere il nome del file eseguibile è possibile ricorrere all'opzione ***"-o***":

"gcc nome_sorgente.c -o nome_eseguibile"

Il compilatore C in ambiente Linux è quindi **gcc**, esso appartiene al progetto GNU, fondato da Richard Stallman nel 1984 al fine di definire e sviluppare un sistema operativo Unix-like interamente "free" (quindi ben prima dell'avvento di Linux, che è del 1991).

Alcune delle opzioni disponibili sono:

- ***"-c***": produce solo file oggetto (da linkare), non eseguibili;
- ***"-D":*** macro=value assegna i valori alle macro del codice (es. -D macro=2222 sostituisce il valore in #define macro value);
- ***"-O***": ottimizza il codice (vari livelli);
- "-***Wall***": produce messaggi di warning;

DEBUGGING

Un **debugger** è un particolare software utilizzato per analizzare il comportamento di un altro programma allo scopo di individuare ed eliminare gli errori. Attraverso il debugger è possibile *eseguire all'interno del programma un'istruzione alla volta, ispezionare e modificare il valore delle variabili, bloccare l'esecuzione del programma in determinati punti, controllare l'ordine in cui vengono chiamate le funzioni ed analizzare un file di tipo "core"*.

Noi utilizzeremo il debugger **GDB**, un debugger disponibile per i sistemi Linux. GDB è l'abbreviazione di GNU debugger, un programma open source facente parte del progetto GNU, divenuto il debugger di default nei sistemi operativi basati su Linux, in quanto in grado di operare con differenti linguaggi di programmazione (tra i quali anche C e C++).

Supponiamo di voler effettuare il debug di un programma, affinché il debugger possa funzionare è necessario compilare il programma con l'opzione **"-g**". In questo modo vengono inserite all'interno dell'eseguibile le informazioni necessarie per il debug.

Supponiamo di voler effettuare il debug del programma "esempio.c":

"gcc g esempio.c -o esempio"

Ora è possibile lanciare il debugger specificando il nome dell'eseguibile ed entrando nella shell del debugger stesso, ovvero:

"***gdb esempio"***

Il debugger mostra il proprio prompt, (gdb), ed è possibile scorrere la lista di tutti i comandi inseriti mediante i tasti "cursore", oltre a *completare automaticamente* il nome del comando digitato premendo il tasto "**Tab**".

Per eseguire il programma si usa il comando "***run***" seguito da uno o più argomenti. In questo modo il programma viene eseguito senza essere interrotto e dunque senza poter controllare passo dopo passo cosa succede al suo interno. Per ovviare a tale problema occorre inserire un **breakpoint**, ovvero scegliere un punto in cui il programma deve terminare l'esecuzione. È possibile inserire un breakpoint in un qualsiasi punto del programma: su un determinato numero di riga, su una funzione...

Quando **gdb** incontra un breakpoint mette in pausa il programma e aspetta l'inserimento di un nuovo comando sulla linea di comando.

Un breakpoint può essere specificato con il comando "***breakpoint***", "***break***", o semplicemente con la lettera "***b***", seguiti dal numero di riga o dal nome della funzione. Ad ogni breakpoint viene assegnato un numero intero progressivo e per visualizzare la lista dei break attivi nel programma si usa il comando "***info break***" o "***info breakpoints***". Per cancellarli si usa il comando "***delete***" seguito dal numero corrispondente al breakpoint che vogliamo eliminare.

Una volta che il programma si è fermato, i principali comandi disponibili per continuare a testarlo sono i seguenti:

- "***n", "next***": per avanzare alla successiva linea di programma senza entrare nelle sotto funzioni;
- ***"s", "step***": per avanzare alla successiva linea di programma entrando nelle sotto funzioni;
- ***"p", "print+ [variabile]"*** oppure "***display+[variabile]":*** per visualizzare o assegnare il contenuto di una variabile;
- "***l", "list***": per visualizzare il codice sorgente del programma sotto esame;

- "***finish***": per completare l'esecuzione del programma fino alla fine della funzione corrente;
- ***"c", "continue***": per continuare l'esecuzione del programma fino al prossimo breakpoint;
- "***quit***": per uscire dal gdb.

Per capire meglio il funzionamento di tale software prendiamo in esame un *programma contenente errori* e, tramite i comandi del debbuger, cercheremo di risalire alla loro causa e quindi correggerli.
Esercizio: Calcoliamo la radice quadrata di un numero reale positivo attraverso il metodo iterativo di Newton.
Ricordiamo brevemente il suddetto metodo: in sostanza si utilizza la seguente formula iterativa: "***x_i+1 = 1/2(x_i + a/x_i) con i=0,1,2,…*** ".
La formula di Newton enuncia che il limite di x_i, per i che tende ad infinito, è uguale alla radice quadrata di a. Generalmente, il valore iniziale x_0 deve essere scelto opportunamente approssimato alla radice di a.
Il programma scritto nel linguaggio C sarà il seguente:

```
#include <stdio.h>
main (){
 float x0=1, x1, a;
 int i;
 printf("Inserisci un numero reale positivo: ");
 scanf("%f", &a);

 for (i=1; i<4; i++){
   x1=0.5*(x0+a/x0);
   x0=x1;
 }
 printf("La radice quadrata di %f e' %f\n",a,x1);
}
```

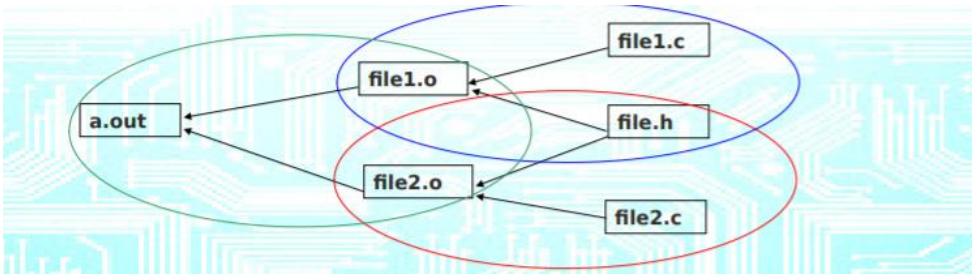
Per prima cosa compiliamo il codice sorgente con l'opzione di debug:
"***gcc g newton.c -o newton***"
Quindi lo eseguiamo:
"./newton"
Inserisci un numero reale positivo: 81
La radice quadrata di 81.000000 è 12.628693
Ci accorgiamo che il programma non funziona bene, in quanto il risultato è palesemente errato; lanciamo adesso il debugger:
"***gdb newton"***
Mettiamo un breakpoint sul main con il comando break e successivamente mandiamo in esecuzione il programma con il comando run, il programma si ferma senza eseguire neppure una linea di codice.
Ora con il comando ***display*** visualizziamo il valore di x1 per vedere quale potrebbe essere l'errore.
"(gdb) display x1"
Ora con il comando "***n***", ripetuto più volte, continueremo l'esecuzione del programma, verificando i valori intermedi assunti da x1, prima della stampa a video. Osserviamo che x1 tende a convergere: il malfunzionamento, individuato tramite il debugger, è dato (banalmente) dal fatto che il numero di iterazioni è insufficiente.

AUTOMATIZZARE IL PROCESSO DI COMPILAZIONE

Sviluppando programmi complessi, si è spesso portati a suddividere il codice sorgente in diversi file. La fase di compilazione quindi richiede maggior tempo, anche se le modifiche apportate riguardano soltanto una piccola parte dell'intero progetto.
Il comando "***make***" è uno strumento che aiuta il programmatore nella fase di sviluppo, tenendo traccia delle modifiche apportate e delle dipendenze fra i vari file, ricompilando soltanto quando necessario.
Supponiamo di avere un programma suddiviso in 3 file:

- *2 file per il codice: file1.c e file2.c;*
- *1 file per le intestazioni: file.h*

La compilazione viene svolta in 3 passi:
"***cc -c file1.c"*** crea il file oggetto (in assembly) file1.o
"***cc -c file2.c***" crea il file oggetto (in assembly) file2.o
"cc file1.o file2.o" crea il file eseguibile a.out
Il grafo delle dipendenze è il seguente:



Il grafo delle dipendenze viene codificato in un file di testo chiamato "**Makefile"**, che risiede nella stessa directory del sorgente.
Un **makefile** è composto da regole specificate dalla seguente sintassi:
target: source file(s)
command
command deve essere sempre preceduto da un TAB.
Il comando make controlla le date di ultima modifica dei file. Ogni volta che un file (***source***) ha una data di modifica più recente di quella dei file che da essa dipendono (**target)**, questi ultimi vengono aggiornati seguendo i comandi specificati nelle regole del makefile.
Esempio di makefile:

```
a.out: file1.o file2.o
        cc file1.o file2.o
file1.o: file.h file1.c
        cc -c file1.c
file2.o: file.h file2.c
        cc -c file2.c
```

Supponiamo di avere, ora, il seguente makefile:
Target: source
Azione
Supponiamo che esista il file "*source*": digitando "**make Target**", make controlla l'esistenza del file "Target":

- se Target esiste, make controlla le date di ultima modifica dei file **Target** e **source**;
- se Target è stato modificato dopo il file source, allora la shell scrive a video che il file **Target è stato aggiornato**;
- se Target è stato modificato prima del file source, allora la shell esegue solo **l'azione corrispondente al Target**;
- se Target non esiste, make fa eseguire alla shell solo **l'azione corrispondente a Target**

Supponiamo che non esista il file "source": sia che il file Target esista oppure no, make fa eseguire alla shell l'azione corrispondente al Target.
Se aggiungiamo al makefile un'altra regola che ha come "Target" il precedente "source", makefile verifica prima le dipendenze ed esegue la regola source prima della regola Target.
È possibile specificare target specifici privi di dipendenze(**make clean**).
Esempi di makefile completo:

```
a.out: file1.o file2.o
    cc file1.o file2.o

file1.o: file.h file1.c
    cc -c file1.c

file2.o: file.h file2.c
    cc -c file2.c

clean: rm -f *.o
```

La prima esecuzione di make lancia i seguenti comandi:

"cc c file1.c"

"cc c file2.c"

"cc file1.o file2.o"

Il comando **make clean** provoca sempre l'esecuzione di "***rm -f * .o"***, in quanto è un target senza dipendenze.

Possono definire *valori da utilizzare varie volte*.

Esempio**: OBJECTS** = file1.o file2.o, per riferirsi ad essa si scrive ***$(OBJECTS***).

Esempio:

```
OBJECTS = file1.o file2.o
a.out: $(OBJECTS)
  cc $(OBJECTS)
file1.o: file.h file1.c
  cc -c file1.c
file2.o: file.h file2.c
  cc -c file2.c
```

L'assegnamento da command line:

"***make 'OBJECTS=file1.o file2.o***"' sovrascrive il valore definito nel Makefile.

PROGRAMMAZIONE DI SISTEMA

Consiste *nell'utilizzare l'interfaccia di system call fra il kernel e le applicazioni che girano sotto Unix*. Il kernel è la parte di Linux che corrisponde al sistema operativo vero e proprio, gestisce sia i processi che l'I/O.

System call: chiamate al sistema che fanno eseguire i metodi che richiamano le funzioni in kernel space.

Per i programmi C, le system call sono come funzioni, ogni system call ha un prototipo, ad esempio: "***pid_t fork(void)***";

Si usano normalmente pid_t pid; pid = fork(); e restituiscono un valore negativo (tipicamente -1) per indicare errore.

Tipi di system call relative a:

- controllo di processi;
- gestione di file;
- comunicazione tra processi;
- segnali.

Alcune system call per **il controllo dei processi:**

- *getpid, getppid, getgrp ecc.: forniscono degli attributi dei processi (PID, PPID, gruppo ecc.);*
- *fork: crea un processo figlio duplicando il chiamante;*
- *exec: trasforma un processo sostituendo un nuovo programma nello spazio di memoria del chiamante;*
- *wait: permette la sincronizzazione fra processi; il chiamante "attende" la terminazione di un processo correlato;*
- *exit: termina un processo.*

INTRODUZIONE ALLE COMUNICAZIONI INTERPROCESSO

L'accesso concorrente a dati condivisi può provocare l'inconsistenza dei dati stessi, per garantire la consistenza occorre implementare meccanismi in grado di coordinare l'esecuzione dei processi cooperanti.

Race Condition è una situazione nella quale due o più processi leggono o scrivono dei dati condivisi, ed in tale contesto il risultato finale dipende dalle loro tempistiche, per prevenire problemi, tali processi concorrenti devono essere opportunamente sincronizzati.

Nel noto problema della sezione critica: n processi competono per usare dati condivisi e ogni processo ha un segmento di codice (sezione critica) in cui i dati condivisi sono utilizzati. Il problema consiste nel garantire che quando un processo *sta eseguendo la sua sezione critica, nessun altro processo possa accedervi.*

Prevedere appropriate operazioni primitive per ottenere la mutua esclusione è uno dei problemi più importanti quando si progetta un qualunque sistema operativo.

In tale contesto occorre assicurarsi che:

- *due processi non devono mai simultaneamente accedere ad una sezione critica;*
- *nessuna assunzione a priori deve essere fatta a proposito della velocità o del numero di CPU;*
- *nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi;*
- *nessun processo deve aspettare indefinitamente per entrare nella sua sezione critica.*

CREAZIONE DI PROCESSI

Il prototipo è: "pid_t fork()"; dove pid_t è definito negli include di sistema e di solito corrisponde ad un tipo intero, un tipico utilizzo è:

"#include <unistd.h>
pid_t pid; ...
pid = fork();"

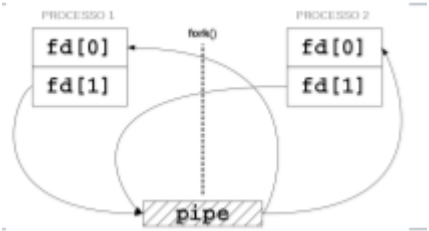
Il valore restituito da fork viene usato per distinguere tra **processo genitore e processo figlio**, infatti al genitore viene restituito il PID del figlio, mentre al figlio viene restituito 0.

Quindi restituisce 0 al figlio e pid al padre, oppure -1 (solo al padre) in caso di errore (per esempio, la tabella dei processi non ha più spazio).

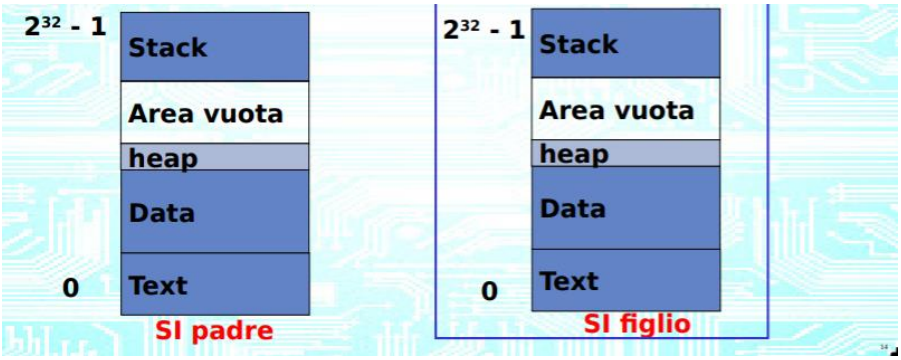
Lo spazio di indirizzamento del nuovo processo è *un duplicato di quello del padre*.

Padre e figlio hanno due tabelle dei descrittori di file diverse (il figlio ha una copia di quella del padre) ma condividono la tabella dei file aperti (e quindi anche il puntatore alla locazione corrente di ogni file).

Convenzionalmente i File Descriptor, 0, 1 e 2, cioè gli indici dell'array, identificano rispettivamente lo standard input, lo standard output e lo standard error



Spazio di indirizzamento di padre e figlio dopo una fork terminata con successo:



Entrambi i processi iniziano la loro esecuzione dopo la system call **fork()**, entrambi i processi hanno un **identico** ma **separato** spazio di indirizzamento.
Tutte le variabili inizializzate prima di invocare la fork() hanno lo stesso valore in entrambi gli spazi di indirizzamento, dato che ciascun spazio di indirizzamento è separato, ogni successiva modifica sarà indipendente per ciascun processo.

esempio 1

Unix standard library

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main()
{
pid_t pid;
printf("Unico processo con PID %d.\n",(int)getpid());
pid=fork();
if(pid == 0)
  printf("Sono il processo figlio (PID: %d).\n",
(int)getpid());
else if(pid>0)
  printf("Sono il genitore del processo con PID   %d.\n",pid);
  else
  printf("Si e' verificato un errore nella    chiamata a
fork.\n");
}
```

Trascinare la corretta istruzione all'interno nel seguente codice:

```
main()
{
 pid_t pid;
 pid = fork();
 switch(pid) {
 case -1:
 printf("fork failed");
 break;

 case 0:
 execl("/bin/ls", "-l", (char *)0);
 break;

 default:
 wait((int *)0);
 _exit(0);
}}
```

wait(0, NULL); | wait(-1, 0, 0); | waitpid((int *)0);

TERMINAZIONE DEI PROCESSI

Terminazione dei processi mediante exit(): "***void exit(int status)***". Chiude tutti i descrittori di file, libera lo spazio di indirizzamento, invia un **segnale SIGCHLD** al padre (un segnale inviato da ogni processo al processo padre al momento della sua terminazione), salva il primo byte (0-255) di status nella tabella dei processi in attesa che il padre lo accetti con una ***wait()*** o ***waitpid()*.**
Se il processo padre è terminato, il processo 'orfano' viene adottato da **init** (il suo pid viene settato a 1).
Se eseguita nel main() è equivalente ad una return.
Attesa terminazione del processo figlio:
"pid_t wait(int *status)
pid_t waitpid(pid_t pid,i nt *status, int options)"
Restituiscono informazioni sul tipo di terminazione del figlio e il byte meno significativo restituito dalla exit(), informazioni alle quali è possibile accedere attraverso opportune maschere come:

- ***WIFEXITED(status***): True se il processo termina normalmente;
- ***WEXITSTATUS(status***): se WIFEXITED(status) è True, consente di accedere al valore restituito dalla exit().

In caso di errore (ad esempio, non ci sono figli) viene restituito il valore -1.

In sintesi:

- **wait():** permette ad un processo di attendere la terminazione di uno dei suoi processi figli, consentendo di accedere al valore ritornato dalla funzione exit();
- **waitpid()**: sospende il processo corrente in attesa che termini il figlio specificato tramite pid.

La system call wait() consente ad un processo di attendere che termini uno dei suoi processi figli, ottenendo il valore ritornato dalla sua funzione exit():

- *se il processo chiamante non ha figli, la chiamata fallisce e viene ritornato il codice di errore, il quale è un numero minore di zero;*
- *se il processo chiamante ha generato processi figli mediante fork(), ma non è terminato ancora nessuno di questi, la system call si blocca;*
- *se il processo chiamante ha figli già terminati, viene ritornato pid il valore di uscita del primo dei processi figlio terminati;*

I file header da includere ed il prototipo della system call wait() sono riportati di seguito:
#include<sys/types.h>
#include<sys/wait.h>
pid_t wait(int *status)

La funzione **waitpid()** consente ad un processo di attendere che termini un determinato processo figlio, specificato mediante pid come argomento.

Il processo corrente termina anche se viene ricevuto un segnale di terminazione o un segnale che coinvolga la gestione da parte di una funzione, se il processo figlio specificato tramite pid si trova nello stato di **zombie**, la funzione relativa al processo corrente ritorna immediatamente, liberando tutte le risorse dei processi figli.

Il parametro pid di waitpid() accetta i seguenti valori:

- ***<-1***: attende la terminazione dei figli con process group ID uguale al valore assoluto del pid specificato (esempio pid=-2, attende figli con process group ID=2);
- ***-1***: attende la terminazione di qualunque figlio, operando esattamente come una wait();
- ***0***: attende la terminazione dei figli con process group ID uguale a quello del processo corrente;
- ***> 0***: attende il processo figlio con il pid specificato.

I file header da includere ed il prototipo della system call waitpid() sono riportati di seguito:
#include<sys/types.h>
#include>sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);

Sulla base di quanto detto in precedenza, queste due system call sono quindi equivalenti: ***wait(&status) e waitpid(-1, &status, 0).***

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

main(){

pid_t pid;
int status ; // conterrà lo stato

pid = fork();

if (pid) { // siamo nel padre (0=figlio, >0=padre, -1=errore)
    sleep(20); // attendiamo 20 secondi
    pid = wait(&status); // restituisce il PID del processo completato

    if (WIFEXITED(status)) { // !=0 se il figlio termina normalmente (exit o return), cioè non terminato da signal
        printf("stato %d\n", WEXITSTATUS(status));
    }
}
else { // siamo nel figlio
    printf("Processo %d, figlio.\n", getpid());
    _exit(17); // terminiamo con un valore di ritorno (exit status) di 17
    }
}
```

NOTA: la **exit()** prima di uscire ripristina lo stato del processo (per esempio, chiude file e descrittori aperti), mentre la **_exit()** esce senza effettuare queste operazioni, modalità da impiegare in un contesto operativo basato su impiego di **fork()**

Lanciamo il programma dell'esempio precedente e mandiamo il processo in background dopo la stampa del pid del figlio [CTRL+z], verifichiamo quindi con il comando "***ps***" lo stato del figlio.
Lo stato del processo figlio sarà indicato come **zombie** (o *defunct*), un processo diventa zombie dopo essere terminato e prima che il processo padre abbia eseguito una wait su quel figlio.
La wait ha lo scopo di recuperare le informazioni sul processo figlio, non avvenendo questo il processo non termina correttamente, rimanendo zombie, finché la wait non viene chiamata.
Un processo zombie non consuma memoria, ma un *numero elevato di essi potrebbe portare all'impossibilità di assegnare nuovi PID*, saturando la tabella dei processi, e quindi rendendo impossibile la creazione di nuovi processi.
Se il padre muore prima di poter fare la wait al processo figlio, quest'ultimo viene ereditato dal processo init, che si occuperà di effettuare la wait.
Se **fork** fosse l'unico modo per creare nuovi processi la *programmazione in ambiente Linux risulterebbe alquanto ostica, i*n quanto si potrebbero creare soltanto copie dello stesso processo.
La famiglia di primitive **exec** ci consente di superare tale limitazione, in quanto le system call di questa famiglia consentono di avviare un nuovo programma sovrascrivendo la memoria di un processo già esistente.
Alcune di queste system call sono ***execl, execle, execlp, execv, execvp e execve***. Tutte richiamano comunque, seppur in modo diverso, execve che rappresenta la principale system call della famiglia, cambia solo il modo in cui vengono passati i parametri, cioè:

- ***p***: richiede un "nome programma";
- **l**: richiede una "argument list";
- **e**: richiede una "array environnement variable".

In assenza di errori questa system call non effettua un return.
La system call execl elimina il programma originale, sovrascrivendolo, le istruzioni che seguono una chiamata a execl verranno eseguite soltanto in caso si verifichi un errore nella chiamata execl.
Il prototipo ***exec()*** è: " ***int execl(char *pathname, char *arg0, ...);".***

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

main()
{
  printf("Esecuzione di ls\n");

  execl("/bin/ls", "-l", (char *)0);

  perror("execl ha generato un errore\n");

  _exit(1);
}
```

Il **pathname** rappresenta il comando da eseguire con specificato il suo percorso completo, esso è seguito da una lista di variabili che contengono un puntatore ad una stringa di caratteri, essi rappresentano i valori da passare ad argv nel nuovo programma eseguito. Tale lista deve essere terminata con un puntatore di tipo NULL.
L'utilizzo combinato di fork per creare un nuovo processo e di exec per eseguire nel processo figlio un nuovo programma costituisce un potente strumento di programmazione in ambiente Linux

PROGRAMMARE CON LA LIBRERIA NCURSES

Ncurses (contrazione di "*new curses*") è una libreria di funzioni che consente di gestire una shell alfanumerica Linux (terminale) come se fosse uno schermo grafico.
Attraverso le **API** disponibile è quindi possibile svolgere funzionalità grafiche e impiegare il mouse su un terminale alfanumerico, a prescindere dalle caratteristiche di questo, essa è un'implementazione open source di una precedente libreria curses per sistemi Unix, dove sono state introdotte ulteriori funzionalità.

Qualora il relativo pacchetto non fosse già installato nella propria distribuzione Linux, è possibile installarlo eseguendo (nel caso della versione 5): "***sudo apt -get install libncurses5 -dev libncursesw5dev".***
Per impiegare la libreria occorre includere un file header contenente alcune definizioni da essa impiegate, cioè:
"#include <curses.h>"
Occorre inoltre specificare l'impiego di tale libreria durante il processo di compilazione tramite gcc, aggiungendo la seguente opzione: "***lncurses***".
Quindi invocare il compilatore in questo modo: "***gcc sorgente.c o nome_eseguibile lncurses***".
Una volta installata la libreria, è possibile utilizzarla in accordo al seguente schema:

```
#include <curses.h>

Main(){

initscr();      /* Inizializza schermo */
noecho();       /* Imposta modalità della tastiera */
curs_set(0);    /* Nasconde il cursore */

/* CODICE PROGRAMMA */

endwin();       /* Ripristina la modalità standard */

}
```

"#include" ci consente di utilizzare questa libreria ed esso può variare a seconda della distribuzione Linux (*curses.h* oppure *ncurses.h*).
"***initscr()"*** inizializza la libreria per l'impiego, significando che le funzioni di libreria potranno essere utilizzate solo dopo l'invocazione di initscr().
Dopo l'invocazione di initscr() *non saranno più disponibili le funzioni di input/output standard come, per esempio, printf()*.
"***endwin()***" termina l'utilizzo della libreria ncurses, per riprendere l'impiego di ncurses è necessario rieseguire la libreria con l'istruzione "***refresh()***" o re inizializzarla con "***initscr()***".
Tipicamente, tutte le funzioni di ncurses restituiscono un valore intero positivo per indicare che l'operazione si è conclusa con successo, mentre restituiscono un valore intero negativo per indicare che si è verificato un errore.
Appena invocato initscr() lo schermo (l'area della shell/terminale) viene cancellato.
In ambito ncurses lo schermo viene gestito come *una matrice di caratteri identificati mediante coordinate cartesiane*, dove l'angolo in alto a sinistra ha coordinate (x=0, y=0).
La dimensione dello schermo è modificabile mediante le variabili intere LINES e COLS che stabiliscono, rispettivamente, il numero di caratteri che caratterizza linee (righe) e colonne dell'area. Per cui l'angolo in basso a destra avrà coordinate (x=COLS, y=LINES).
Come accennato in precedenza, mediante la funzione "***curs_set()***" possiamo impostare la visibilità del cursore, utilizzando la seguente sintassi: "***int curs_set(int visibilità);".***
Essa ci consente di impostare un cursore **invisibile, visibile, oppure molto visibile,** utilizzando come parametro, rispettivamente, 0, 1 o 2.
I comandi di output non sono visualizzati immediatamente da ncurses ma soltanto memorizzati fino all'invocazione della funzione refresh(), una modalità volta a velocizzare le operazioni di output. Ne deriva che l'utilizzo di ncurses si compone di uno o più comandi di output seguiti dall'istruzione refresh().
Considerando che in ambito ncurses *non sono disponibili le funzioni standard di input/outpu*t, occorre utilizzare altre funzioni, invece della funzione standard *printf()* è possibile impiegare "***printw***", anche se con quest'ultima non è necessario inserire un ritorno a capo a fine riga.
L'output viene visualizzato a partire dalla posizione corrente dello schermo, il suo prototipo è il seguente:
"***int printw(char *fmt [, arg] ...)";***

Analogamente, esiste una funzione di input che sostituisce la standard *scanf(),* essa è ***"scanw***", il suo prototipo è il seguente: "***int scanw(char *fmt [, arg] ...)".***

Dove, per esempio, con scanw("%c", &i) chiediamo in input un valore alla posizione corrente e lo memorizziamo nella variabile i.

Una funzione equivalente a printw, dove è però possibile specificare le coordinate dove posizionare l'output è "***mvprintw()***", il cui prototipo è di seguito riportato:

"***int mvprintw(int y, int x, char *fmt [, arg] ...);"***

Da tenere conto che i valori delle coordinate orizzontali e verticali sono invertite rispetto alla loro usuale posizione, cioè (y,x) invece che (x, y).

Ad esempio, se vogliamo scrivere il contenuto della variabile intera i nella posizione x=6, y=2, quindi alla sesta colonna e seconda linea, dobbiamo eseguire: "***mvprintw(2, 6,"Contenuto della variabile: %d", i);"***

Esiste l'analogo per la funzione di input scanw(), esso è "***mvscanw()",*** il cui prototipo è di seguito riportato: "***int mvscanw(int y, int x, char *fmt [, arg] ...);"***

Anche in questo caso valgono le medesime considerazioni sulle coordinate fatte in precedenza per mvprintw().

Per comportamento predefinito, la libreria ncurses visualizza sullo schermo i caratteri digitati in fase di input, per fare in modo che ciò non accada occorre invocare la seguente funzione: "***noecho()***". Per ritornare al comportamento predefinito, occorre invece richiamare: ***"echo()***";.

Per fare in modo che quel che digitiamo venga immediatamente inviato al programma, senza che sia necessaria la pressione del tasto INVIO, possiamo invocare la funzione "***cbreak()***", il cui prototipo è: "***int cbreak(void)".*** Per ritornare al comportamento predefinito, occorrerà poi invocare la funzione inversa "***nocbreak()***", il cui prototipo è: "***int nocbreak(void);***

La funzione "***timeout()***", il cui prototipo è di seguito riportato: "***int timeout(int delay);"*** Se utilizziamo 0 (zero) come parametro, invocando la funzione "***getch()***" *non attende che l'utente prema un qualunque tasto, proseguendo nell'esecuzione del programma*. Se utilizziamo un valore positivo di delay, *il programma attende per i millisecondi specificati e poi prosegue*. Se utilizziamo un valore negativo di delay, *il programma opera secondo il comportamento standard, cioè attende l'input dell'utente per un tempo indefinito*.

Qualora si intendesse inibire l'esecuzione del programma per un numero specificato di millisecondi (delay), è possibile ricorrere alla funzione "***napms()***", il cui prototipo è di seguito riportato: "***int napms(int delay);"***

Il concetto di **colore** è visto in ambito ncurses come una coppia di valori che specifica il colore del carattere da visualizzare e quello del relativo sfondo, secondo lo schema (***colore_carattere, colore_sfondo***).

Per utilizzare i colori, dopo l'invocazione della funzione initscr() è necessario invocare la seguente funzione: ***"int start_color(void);"***

Mentre con la successiva funzione definiamo le coppie di colori che utilizzeremo: "***int init_pair(short pair, short f, short b);"***

Il parametro **pair** è un intero da 1 a 7 (0 è riservato a bianco su nero), il parametro **f** è il colore del carattere, mentre il parametro **b** è il colore dello sfondo.

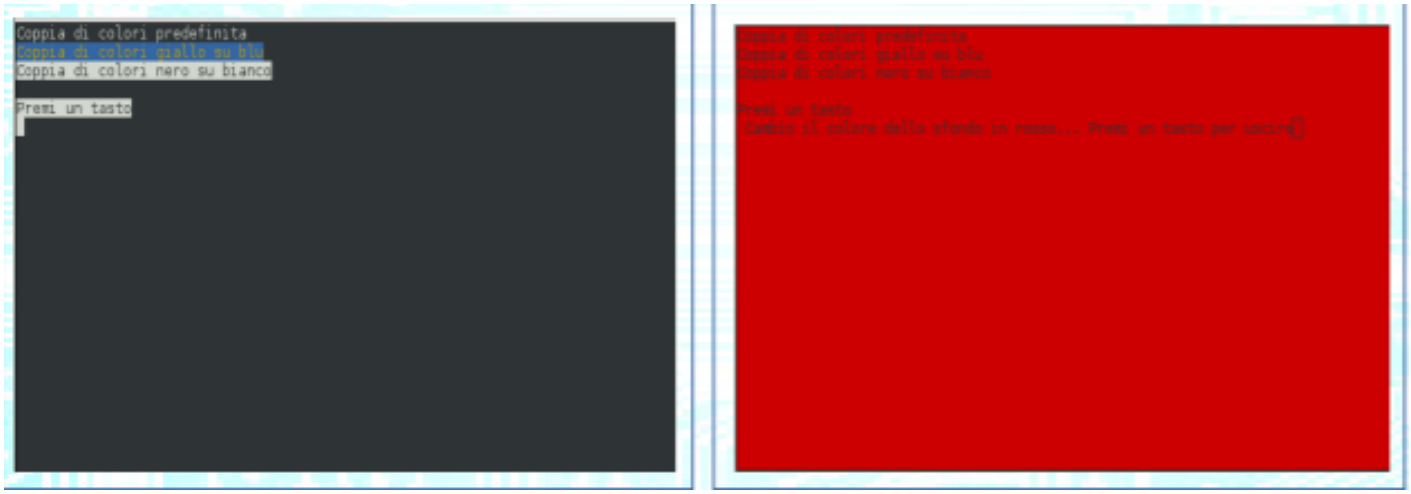
Sebbene sia possibile creare in modo arbitrario i colori sulla base delle componenti RGB che li definiscono (funzione "***init_color"***), nella maggior parte dei casi è sufficiente impiegare quelli predefiniti, cioè:

- *COLOR_BLACK;*
- *COLOR_RED;*
- *COLOR_GREEN;*
- *COLOR_YELLOW;*
- *COLOR_BLUE;*
- *COLOR_MAGENTA;*
- *COLOR_CYAN;*
- *COLOR_WHITE.*

Utilizzando la seguente sintassi per definire, per esempio, dei caratteri gialli su sfondo nero: "***init_pair(1,COLOR_YELLOW,COLOR_BLACK);***".
La funzione "***attron()***" consente di selezionare la coppia di attributi colore/sfondo prima definiti, il suo prototipo è il seguente: "***int attron(int attributi);***"
È possibile invocarla in combinazione con la macro ***COLOR_PAIR(n***), dove n è il numero che identifica la coppia di colori prima definita, quindi scrivere: ***attron(COLOR_PAIR(n));***
Un comando utile per poter *modificare il colore di sfondo dell'intera area di output* è il seguente: "***bkgd(int attributi);".***
Un esempio di codice che riepiloga quanto detto è il seguente:

```c
#include <stdio.h>
#include <ncurses.h>

int main()
{
  initscr();
  start_color();
  init_pair(1,COLOR_YELLOW,COLOR_BLUE);
  init_pair(2,COLOR_BLACK,COLOR_WHITE);
  init_pair(3,COLOR_BLACK,COLOR_RED);

  printw("Coppia di colori predefinita\n");

  attron(COLOR_PAIR(1));
  printw("Coppia di colori giallo su blu\n");

  attron(COLOR_PAIR(2));
  printw("Coppia di colori nero su bianco\n\n");
  printw("Premi un tasto\n");

  refresh();
  getch();

  printw("Cambiato il colore dello sfondo in rosso... Premi un tasto per uscire");
  bkgd(COLOR_PAIR(3));

  refresh();
  getch();
  endwin();

  return 0;
}
```

L'output sarà il seguente:



La libreria ncurses mette a disposizione differenti funzioni per la **cancellazione**:

- ***int delch(void***): cancella il carattere sotto il cursore e sposta i caratteri seguenti ad esso verso sinistra;
- ***int deleteln(void***): cancella l'intera riga su cui si trova il cursore, muovendo in alto le linee che seguono;
- ***int insertln():*** inserisce una riga vuota nella posizione corrente del cursore, muovendo in basso le altre, quindi cancellando l'ultima riga in basso;
- ***int clrtoeol(void***): cancella tutti i caratteri alla destra del cursore presenti sulla riga;

- ***int erase(void):*** sovrascrive con spazi bianchi ogni posizione dello schermo, cancellandolo (una funzione molto lenta)

THREADS

Nei sistemi Linux, le funzioni dei thread ricevono in ingresso un solo parametro, di tipo void*, e restituiscono un valore tipo void*. Tale parametro viene definito **thread argument**. Il programma può usare questo parametro per inviare dati al thread creato.

Allo stesso modo il programma può utilizzare il valore di ritorno per ricevere dati dal thread.

La funzione **pthread_create** crea un nuovo thread, essa viene invocata utilizzando:

1) Un puntatore ad una variabile **pthread_t,** destinata a contenere il thread ID del nuovo thread;
2) Un puntatore ad un oggetto **thread attribute**, il quale definisce la maniera nella quale il thread interagisce col resto del programma (se si passa un puntatore a NULL, esso viene creato con attributi di default);
3) Un puntatore alla **thread function**, in pratica una normale funzione di tipo void
4) Un **thread argument** di tipo void* che viene passato al thread al momento della sua creazione;

"#include int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), v oid *arg)"

Dal punto di vista del processo (o thread) chiamante, una chiamata a pthread_create effettua immediatamente una return, e il processo (o thread) chiamante continua l'esecuzione con l'istruzione successiva alla chiamata. Nel frattempo, il nuovo thread inizia l'esecuzione, Linux schedula entrambi i thread in maniera concorrente ed asincrona, asincrona significa che la sequenza di esecuzione delle istruzioni dei 2 thread non è deterministica.

Un programma con i thread deve essere compilato cosi: ***gcc o thread_create thread_create.c lpthread.***

In circostanze normali, un thread può terminare in 2 modi:

1) Utilizzando una **return** dall'interno della thread function: il valore di ritorno del thread è in questo caso il valore di ritorno della thread function;
2) Con una uscita esplicita mediante funzione **pthread_exit**, invocata dall'interno della thread function o da un'altra funzione chiamata direttamente o indirettamente da questa; l'argomento della pthread_exit sarà il valore di ritorno del thread.

La funzione **pthread_join** forza il main ad aspettare fino a che i thread presenti nella join queue non sono terminati, cioè qualcosa di simile al wait() vista in precedenza nel fork dei processi.

Un **mutex** è una sorta di blocco (lock) che solo un thread alla volta può inserire o rimuovere. Se un thread esegue un lock con un mutex ed un secondo thread tenta di eseguire un lock sullo stesso mutex, il secondo thread viene posto in attesa. Solo quando il primo thread esegue un **unlock** sul mutex, viene riattivata l'esecuzione del secondo thread.

Linux guarantisce che non si verifichino race conditions fra thread che effettuano il lock di un mutex: solo un thread alla volta può eseguire un lock, gli altri thread saranno bloccati. Per creare un mutex è necessario creare una variabile di tipo **pthread_mutex_t** e passare un puntatore a questa variabile alla funzione **pthread_mutex_init**. Il secondo argomento da passare alla funzione pthread_mutex_init è un puntatore ad un oggetto mutex_attribute, che specifica gli attributi del mutex. Se tale puntatore è null, vengono utilizzati degli attributi di default. Le variabili mutex devono essere inizializzate una sola volta.

Un'altra possibilità per inizializzare un mutex con attributi di default è inizializzarlo con il valore speciale PTHREAD_MUTEX_INITIALIZER***: pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;***

Gli accessi ad una zona critica del codice dovrebbero avvenire tra la chiamata a **pthread_mutex_lock** e la chiamata a **pthread_mutex_unlock**.

Invece per utilizzare i veri e propri semafori è necessario includere <semaphore.h>, un semaforo è rappresentato da una variabile di tipo **sem_t.**

Prima di essere utilizzata, una variabile sem_t va inizializzata utilizzando la funzione **sem_init**, alla funzione sem_init vanno passati 3 parametri:

1. il primo parametro deve essere **un puntatore** alla variabile sem_t;
2. il secondo parametro deve essere 0;
3. il terzo parametro è il **valore iniziale del contatore** del semafore.

Se non si ha più necessità del semaforo, è buona norma deallocarlo utilizzando la funzione **sem_destroy**. La funzione di wait sul semaforo viene implementata tramite **sem_wait**, la funzione di post (la signal dei mutex) sul semaforo viene implementata tramite **sem_post.**

Linux fornisce anche una funzione di wait non bloccante, essa è **sem_trywait**. Ciò significa che se il semaforo non può essere bloccato, il thread non resta in attesa. Linux fornisce anche la funzione **sem_getvalue**, la quale restituisce il valore corrente del semaforo nella variabile int puntata dal suo secondo parametro.

PROGRAMMAZIONE IN LINGUAGGIO BASH

È possibile programmare la shell Bash, si possono creare files che contengono sequenze di comandi e strutture di controllo del flusso. Salvate come file di testo, possono essere invocati con ***bash nomescript.sh***, oppure automaticamente interpretati se si posseggono I diritti di esecuzione ***chmod 755 nomescript.s.***
Prima riga ***#!/bin/bash***
Il pro è che gli **script** vengono utilizzati per gestire il sistema, permettono in modo semplice di provare procedure complesse.
Il contro è che è meglio non utilizzarla se servono velocità o portabilità.
Uno **shabang** (chiamato anche shebang, hashbang, hashpling, o pound bang), è una sequenza particolare di caratteri che inizia con ***#!***, è posizionato all'inizio di uno script Unix/linux al fine di indicare al sistema quale interprete utilizzare per eseguire le istruzioni contenute. In maggior dettaglio, esso si compone di un **magic number** (i caratteri cancelletto e il punto esclamativo #!), seguito dal pathname assoluto dell'interprete da utilizzare (nel caso di Bash: **#!/bin/bash**). Considerando che nei linguaggi di scripting il carattere # è usato per i commenti, l'interprete ignorerà il contenuto della riga dello shabang, che indicherà soltanto l'interprete da adoperare.
#!/bin/bash echo Hello World
#!/bin/bash tar -czf /var/my-backup.tgz /home/utente/
Le variabili sono **non tipizzate**, si può avere un assegnamento semplice *variabile = valore*, *x=variabile*, *y='pippo'*. Per accedere al valore di una variabile si utilizza il simbolo **$**: *echo il valore di x: $x*, stampa il valore di x che è variabile, invece *echo il valore di y: $y* stampa il valore di y: pippo, echo y, stampa y.
$1, $2, ... sono variabili speciali associate ai vari argomenti passati allo script dalla linea di comando (parametri posizionali)

```
#!/bin/sh
mkdir $1
touch $2
mv $2 $1/$2
```

- **$*** rappresenta in un'unica parola tutti i parametri posizionali passati allo script;
- **$@** rappresenta tutti i parametri posizionali passati allo script, ciascuno sotto forma di singola parola (utile, per esempio, nell'ambito di un ciclo for);
- **$#** numero dei parametri;
- **$?** exit status della più recente pipeline eseguita in foreground;
- **$!** status del più recente processo in background;
- **$0** nome della shell in uso

La funzione **read** consente di assegnare variabili da standard input è utile per semplici interfacce:

```
echo "inserisci il valore della variabile e premi Invio: "
read variabile
echo "variabile= $variabile"
```

Esempio di script bash:

```
#!/bin/bash
# crea un nuovo file con il nome scelto
clear
echo "Verra creato un file con il nome scelto"
echo "inserisci il nome"
read nomefile
touch $nomefile.sh
clear
echo "Verifica dell'avvenuta creazione del
file"
ls -l
#fine script
```

Crea un nuovo file usando come nome il valore della variabile **nomefile,** aggiungendo ad esso l'estensione **.sh**

Per le operazioni matematiche si usa **expr**, **let** o **doppia parentesi tonda**: let a=2+3; echo $a, stampa 5, invece a=`expr 5 + 3`; echo "5 + 3 = $a", stampa 5 + 3 = 8; invece ((a = 4+1)); echo $a, stampa 5.

Operatori:

- ***id++, id***--: incremento e decremento;
- ***++id, --id***: pre-incremento e decremento;
- ***! ~ :*** logical and bitwise negation;
- ***** :*** exponentiation;
- **** / % :*** moltiplicazione, divisione e modulo;
- **+ -** : somma e sottrazione;
- **<< >>** : bitwise shifts;
- **<= >= < >** : comparison;
- **== !=** : equality and inequality;

- **& ^ | :** bitwise AND, XOR, OR;
- **&& | |:** AND, OR.

exit termina uno script e restituisce un valore al processo genitore, ogni comando restituisce un **exit status**. Il successo restituisce 0, il fallimento un codice d'errore. Le funzioni all'interno di uno script restituiscono un exit status, quando uno script termina con exit senza alcun parametro, l'exit status dello script è quello dell'ultimo comando eseguito.

```
#!/bin/bash

echo hello
echo $?       # Exit status 0
lskdf         # Comando sconosciuto.
echo $?       # Exit status diverso da zero perché
              # il comando non ha avuto successo
exit 113      # Restituisce 113 alla shell
```

Il comando condizionale: **if** condition_command **then** true_commands instructions **else** false_commands instructions **fi** . Esegue il comando condition_command e utilizza il suo exit status per decidere se eseguire le istruzioni true_commands (exit status 0) oppure le istruzioni false_commands (exit status non zero).

```
#!/bin/bash

if grep "^$1:" /etc/passwd >/dev/null 2>/dev/null
 then
  echo $1 is a valid login name
 else
  echo $1 is not a valid login name
Fi
```

Lo script prende come argomento un nome di login e stampa a video un messaggio diverso a seconda se il parametro fornito compaia all'inizio di una linea del file /etc/passwd oppure no.

```
# passa dalla directory corrente ad una subdirectory
# e se questa non esiste fornisce un messaggio di errore

#!/bin/bash
clear

echo "Inserisci il nome della subdirectory"
read nomedir
echo ""

if ! cd $nomedir2> /dev/null
then
 echo "Spiacente! la subdirectory $nomedir non esiste!"
else
 echo "La subdirectory $nomedir contiene:"
 ls -l
 echo ""
fi
```

I test condizionali vengono adoperati se non si può usare un exit status di una funzione per la condizione, la sintassi è: ***test –opzione argomento,*** un'alternativa è la chiusura tra parentesi quadre []: # exe_test.sh; #!/bin/bash; test -x /bin/ls verifica se "ls" è un file eseguibile.

Test condizionali sui file:

-e	il file esiste
-f	il file è un file regolare
-s	il file ha dimensione superiore a zero
-d	il file è una directory
-b	il file è un dispositivo a blocchi
-c	il file è un dispositivo a caratteri
-r	il file ha il permesso di lettura
-w	il file ha il permesso di scrittura
-x	il file ha il permesso di esecuzione
-O	siete il proprietario del file
-N	il file è stato modificato dall'ultima lettura
f1 -nt f2	il file f1 è più recente del file f2
f1 -ot f2	il file f1 è meno recente del file f2
f1 -ef f2	il file f1 e f2 sono hard link allo stesso file

Test condizionali su interi:

Test condizionali su stringhe:

-eq	è uguale a
-ne	è diverso (non uguale) da
-gt	è maggiore di
-ge	è maggiore di o uguale a
-lt	è minore di
-le	è minore di o uguale a
<	è minore di (tra doppie parentesi)
<=	è minore di o uguale a (tra doppie parentesi)
>	è maggiore di (tra doppie parentesi)
>=	è maggiore di o uguale a (tra doppie parentesi)

=	è uguale a
==	è uguale a
!=	è diverso (non uguale) da
<	è inferiore a, in ordine alfabetico ASCII
>	è maggiore di, in ordine alfabetico ASCII
-z	la stringa è "null", cioè ha lunghezza zero
-n	la stringa non è "null"

Esempio:

```
if test -z "$1"
then
  echo "Non è stato inserito nessun argomento nella riga di comando"
else
  echo "Il primo argomento inserito nella riga di comando è $1"
fi
```

```
# verifica che non sia stato passato un terzo parametro allo script

if [ [! -z $3 ] ]; then
echo "Inserisci due argomenti, prego"
[exit]
fi
```

Il **case selection** è un'alternativa al if else e ha la stessa sintassi del C con lo switch case:

```
#!/bin/bash;

echo "Premi un tasto e poi invio."
read Tasto

case "$Tasto" in
  [a-z,A-Z] ) echo "Lettera";;
  [0-9] ) echo "Cifra";;
  * ) echo "Punteggiatura, spaziatura, o
  altro";;
esac
```

Per implementare un esecuzione ciclica, si utilizza un **for** che ad ogni passo arg assume il valore di ognuna delle variabili elencate in lista:

```
for arg in [lista]
do
 comando(i)...
done
```

```
#!/bin/bash
for i in 1 2 3 4 5
  do
    echo the value of i is $i
  done
  exit 0
```

Si può anche implementarlo in maniera molto simile al C:

ESEMPIO:

```
#!/bin/bash
LIMITE=10
for ((i=1; i <= LIMITE; i++))
do
   echo "Conteggio: $i"
done
```

In alternativa al for possiamo utilizzare il **while**, il quale è utile in quelle situazioni in cui il numero delle iterazioni non è noto in anticipo:

```
while [condizione]
do
 comando...
done
```

```
#!/bin/bash
while test -e $1
   do
     sleep 2
   done
   echo file $1 does not exist
   exit 0
```

Lo script esegue un ciclo che dura finché il file fornito come argomento non viene cancellato. Il comando che viene eseguito come corpo del while è una pausa di 2 secondi.
Un altro modo per usare il while è l'**until**: lo script legge continuamente dallo standard input e visualizza quanto letto sullo standard output, finché l'utente non inserisce la stringa end.

```
#!/bin/bash
until false
do
   read firstword restofline
   if test $firstword = end
   then
         exit 0
   else
         echo $firstword $restofline
   fi
done
```

Per implementare gli array uso ***array[n],*** la dereferenziazione si fa con ***${array[n]}.*** val[4]=13, val[2]="pippo", se facciamo echo ${val[4]} stampiamo 13, se facciamo echo ${val[2]} stampiamo pippo, definiamo array=(zero uno due tre quattro cinque) facendo echo ${array[2]} stampiamo due.
Per quanto riguarda l'implementazione delle funzioni è un po' limitata. Si utilizza: function nome_funzione { comando... } oppure nome_funzione () { comando... }. Si possono anche passare dei parametri semplicemente facendo: funzione arg1 arg2.

```
#!/bin/bash

myfunction1()
{
  i=0
  RIPETIZIONI=10
  echo
  echo "Inizio"
  echo
  Sleep 1

  while [ $i -lt $RIPETIZIONI ]
  do
    echo "--------TEST--------->"
    echo
    let "i+=1"
   Done
}
```

```
MyFunction2 ()
{  echo "Questa è la seconda funzione"
 }

# La dichiarazione della funzione deve
# precedere la sua chiamata.

# Adesso richiamiamo le due funzioni

  MyFunction1
  MyFunction2

exit 0
```

DOMANDE TEST:

1. Valutare la correttezza dell'affermazione "l'accesso concorrente a dati condivisi può provocare l'inconsistenza dei dati stessi, in mancanza di strumenti di gestione opportuni".
   L'affermazione è corretta.
2. Il linguaggio C dispone di elementi come classi e oggetti, mediante i quali consente una stesura strutturata e funzionale del codice.
   Falso.

3. Dire quale funzione svolge il seguente codice C:

```
close(p[0]);
write(p[1], msg, msize);
```

Chiude il descrittore di lettura del file descriptor "p" e ne utilizza uno in scrittura per inviare il messaggio “msg" di dimensione "msize" tramite la "pipe".

4. Alcuni vantaggi principali dei thread sono: tempi di risposta migliori, risparmio di risorse, parallelismo reale anziché emulato.
5. L’operazione di post su un semaforo: incrementa il valore del semaforo di 1 (è come la signal).
6. Dire se la seguente affermazione è corretta: "Se il mutex è già in stato di lock da parte di un altro thread, la funzione pthread_mutex_lock blocca la propria esecuzione e ritorna al thread chiamante solo quando il mutex viene rilasciato (unlock) dal thread che ne ha eseguito il lock".
   L’affermazione è corretta.
7. Il meccanismo delle "pipe": nasce con l'avvento di Unix, si basa su una coppia di "file descriptor", si adopera per creare un canale di comunicazione.
8. Un "semaforo contatore": è un intero che può assumere valori in un dominio non limitato.
9. I thread argument: vengono restituiti dalla funzione di creazione dei thread.
10. In ambiente Linux, i file core rappresentano il risultato di: una segmention fault.
11. Thread e processi condividono le risorse in modo uguale: falso.
12. Utilizzando i threads è possibile il verificarsi di una race condition: si;
13. Durante l’esecuzione di più threads vengono condivise: le medesime informazioni di stato, la memoria ed altre risorse di sistema.
14. Il meccanismo di attivazione di un thread rappresenta: un operazione poco onerosa.
15. La chiamata di sistema fork() che copia tutti i sottocontesti senza condividerne alcuno, rappresenta un caso particolare di clone().
16. Il cosiddetto “starvation” rappresenta: l’impossibilità continua, da parte di un processo pronto all’esecuzione, di ottenere le risorse di cui necessita per essere eseguito.
17. Il thread principale può creare nuovi thread, i quali eseguono lo stesso programma in: modo concorrente all’interno dello stesso processo.
18. Dopo una chiamata pthread_create il nuovo thread inizia la sua esecuzione in modo concorrente ed asincrono: vero.
19. Un thread viene identificato all’interno di un processo tramite: un thread ID.
20. Se un thread esegue un lock con un mutex ed un secondo thread tenta di eseguire un lock sullo stesso mutex: il secondo thread viene messo in attesa.
21. Se un thread del processo esegue una exec: anche lo spazio di indirizzamento di tutti gli altri thread viene sovrascritto dal nuovo processo invocato.
22. Dire se la seguente affermazione è corretta: "Se il mutex è già in stato di lock da parte di un altro thread, la funzione pthread_mutex_lock blocca la propria esecuzione e ritorna al thread chiamante solo quando il mutex viene rilasciato (unlock) dal thread che ne ha eseguito il lock: è corretta.
23. I thread possono essere considerati: parti di un processo che vengono eseguite in maniera asincrona e concorrente.
24. La system call Clone ci consente di: definire quante e quali risorse del processo genitore debbano essere condivise e generare thread.
25. Se un thread chiude un file descriptor, gli altri thread: non sono più in grado di accedere al file in lettura o scrittura.
26. Un thread può essere definito come: uno dei possibili sottoprocessi che è possibili eseguire all’interno di un processo.
27. Il thread che crea e quello creato condividono: gli stessi file descpritor, le risorse di sistema del processo originale, lo stesso spazio di memoria.
28. Si può accedere ad un semaforo S solo attraverso: due operazioni indivisibili.

29. Dire se la seguente affermazione è corretta: "L'operazione di post incrementa il valore del semaforo di 1, e se il semaforo assumeva precedentemente al post il valore zero e altri thread erano bloccati su di esso con una operazione di wait, uno dei thread viene sbloccato, lasua operazione di wait viene completata e il semaforo ritorna a zero" : si è corretta.
30.