

Sistemi operativi

Docente:

Vittorio Ghini

Contatti:

vittorio.ghini@unibo.it

Siti utili:

1. <https://virtuale.unibo.it/course/view.php?id=58257>
2. https://www.cs.unibo.it/~ghini/didattica/sistemioperativi/sistemioperativi_index.html

Realizzato da Rebecca Scarcelli **completato in data** 10 dic 2024

Indice

Argomento	Pagina
File system	03
Comandi utili	04
Comando echo	05
Quoting	05
Caratteri speciali	06
Expansion	06
Variabili	08
File	09
Subshell	10
Valutazione Aritmetica	12
Exit status	12
Sequenze di comandi condizionali e non	13
Loop e flussi	14
Stream I/O	14
Espressioni condizionali	15
Caratteri non stampabili in una stringa	17
Lettura da standard input	18
Apertura di file	19
Redirezionamenti di Stream di I/O	20
Raggruppamenti di comandi	22
GNU Coreutils	23
Variabile random	26
Terminazione di un terminale di controllo	26
Processi in foreground e background	26
Comando wait	27
Processi zombie, processi Orfani e processo init	27
Manipolazione di stringhe	28
Comando find	29
Installazione di pacchetti	30

File system

In **windows**: le partizioni sono separate e indicate con la lettera maiuscola (es. D: C:). Si utilizza come **separatore** il carattere \.

In **Linux/Mac**: le partizioni sono tutte collegate alla **root** / del file system. In questo caso si utilizza come **separatore** il carattere /.

In entrambi i sistemi è possibile specificare due percorsi

→ Percorso **assoluto** che **parte** dalla **root**

es. /home/user (linux)

es. D:\home\user (windows)

→ Percorso **relativo** che **parte** dalla **cartella in cui mi trovo** utilizzando il comando **..** che indica la directory padre e **.** che indica quella corrente

es. ../../mnt/win (linux)

es. ../user (windows)

Per vedere in che **directory ci si trova** si usa il comando **pwd** mentre per **cambiare directory** si utilizza il comando **cd <percorso assoluto o relativo>**

In \ ci sono delle **directory predefinite**:

1. **bin** Essential command binaries
2. **boot** Static files of the boot loader
3. **dev** Device files
4. **etc** Host-specific system configuration
5. **lib** Essential shared libraries and kernel modules
6. **media** Mount point for removable media
7. **mnt** Mount point for mounting a filesystem temporarily
8. **opt** Add-on application software packages
9. **sbin** Essential system binaries
10. **srv** Data for services provided by this system
11. **tmp** Temporary files user Secondary hierarchy

Per creare un **nuovo file directory** (cartella) si usa il comando **mkdir <Nome Directory>** tale nome non deve contenere caratteri speciali o spazi

es. `mkdir EserciziBash` (crea correttamente la cartella)

es. `mkdir Esercizi Bash` (crea una cartella Esercizi e una cartella Bash)

Per creare invece un **nuovo file** viene usato il comando `touch <Nome File.espansione>`

es. `touch nuovo.sh` (crea un nuovo file di espansione .sh)

es. `touch myFile.txt` (crea un nuovo file di testo)

Comandi utili

- `pwd` mostra directory di lavoro corrente .
- `cd percorso_directory` cambia la directory di lavoro corrente .
- `mkdir percorso_directory` crea una nuova directory nel percorso specificato
- `rmdir percorso_directory` elimina la directory specificata, se è vuota
- `ls -alh percorso` stampa informazioni su tutti i files contenuti nel percorso
- `rm percorso_file` elimina il file specificato
- `echo` sequenza di caratteri visualizza in output la sequenza di caratteri specificata
- `cat percorso_file` visualizza in output il contenuto del file specificato
- `env` visualizza le variabili ed il loro valore
- `which nomefileeseguibile` visualizza il percorso in cui si trova (solo se nella PATH) l'eseguibile
- `mv percorso_file percorso_nuovo` sposta il file specificato in una nuova posizione
- `ps` aux stampa informazioni sui processi in esecuzione
- `du percorso_directory` visualizza l'occupazione del disco.
- `kill -9 pid_processo` elimina processo avente identificativo pid_processo
- `killall nome_processo` elimina tutti i processi con nome nome_processo
- `bg` ripristina un job fermato e messo in sottofondo
- `fg` porta il job più recente in primo piano
- `df` mostra spazio libero dei filesystem montati
- `touch percorso_file` crea il file specificato se non esiste, oppure ne aggiorna data.

- **more** `percorso_file` mostra il file specificato un poco alla volta
- **head** `percorso_file` mostra le prime 10 linee del file specificato
- **tail** `percorso_file` mostra le ultime 10 linee del file specificato
- **man** `nomecomando` è il manuale, fornisce informazioni sul comando specificato
- **find** cercare dei files
- **grep** cerca tra le righe di file quelle che contengono alcune parole
- **read** `nomevariabile` legge input da standard input e lo inserisce nella variabile specificata
- **wc** conta il numero di parole o di caratteri di un file
- **true** restituisce exit status 0 (vero)
- **false** restituisce exit status 1 (non vero)

Comando echo

Il comando **echo** permette di **stampare a video** la parte che segue il comando

es. `echo gatto cane ciao` (stampa in output gatto cane ciao)

Tuttavia il testo che segue echo **non deve contenere caratteri speciali** se non quotati.

es. `echo gatto ; cane` (stampa gatto e cane: command not found)

Quoting

Il **quoting** viene fatto attraverso le `" "` o ``` in generale il loro utilizzo è indistinguibile ma c'è una differenza sostanziale:

- `" "` impedisce wildcards e del `;` ma **permette le expansion**

es. `"${VAR}"` (viene sostituita con il valore di VAR)

- `' '` **impedisce** ogni tipo di **expansion** e **interpretazione** dei metacaratteri

es. `'${VAR}'` (non viene sostituita con il valore di VAR)

Caratteri speciali

1. `>` `>>` `<` redirectione I/O
2. `|` pipe

3. ***** **?** [...] wildcards
4. **`command`** command substitution
5. **;** esecuzione sequenziale
6. **||** **&&** esecuzione condizionale
7. **(...)** raggruppamento comandi
8. **&** esecuzione in background
9. **"** **"** **'** **'** quoting
10. **#** commento (tranne un caso speciale)
11. **\$** espansione di variabile
12. **** carattere di escape *
13. **<<** "here document"

Expansion

→ History expansion

Memorizza i comandi già utilizzati in particolare si usa `set +o history` per disabilitare l'espansione mentre `-o history` la abilita.

Possiamo anche **rilanciare i comandi usati** tramite il comando `!NumeroComando`

→ Brace expansion

Preambolo{parola1, parola2, parola3, ...}Post-scritto è una brace expansion che **genera stringhe** (anche con **nomi di variabile**) combinando preambolo, parole e post-scritto.

È possibile annidare più brace expansion o specificare un intervallo {estremo..estremo}

es. Mio{no, gno, "gre, re"}re produce le stringhe

Mionore Miognore Miogre, rere

→ Tilde expansion

È un'espansione realizzata quando viene riconosciuto il **carattere ~**, la tilde viene **sostituita dal percorso assoluto della home directory** dell'utente corrente, se la tilde

è seguita da altri caratteri allora tali caratteri vengono concatenati al percorso della home directory.

→ Parameter expansion

Variabili che **memorizzano le informazioni sugli argomenti passati** alla shell:

- ♦ **\$#** viene sostituito con il numero di caratteri passati alla shell
- ♦ **\$Nome** viene sostituito con il **nome del processo**
- ♦ **\$n** viene **sostituito dall'n-esimo argomento** passato alla shell
- ♦ **\$*** viene sostituito da tutti gli argomenti passati alla shell concatenati e separati da spazi
- ♦ **\$@** è uguale al precedente solo che **mantiene protetti i singoli argomenti**. È utilizzato quando uno script deve eseguire un altro comando passandogli tutti i suoi argomenti.

→ Variable expansion

Sostituisce al nome delle variabili il proprio contenuto.

→ Arithmetic expansion

((espressione da valutare)) racchiude **tutta la riga di comando**, tutta valutabile aritmeticamente, **valuta l'espressione** all'interno **e la esegue**

\$((espressione da valutare)) racchiude **solo una parte di una riga di comando**, non tutta valutabile, **valuta l'espressione** all'interno e **il risultato calcolato va a sostituire l'operatore** stesso nella riga di comando che poi viene eseguita.

→ Command substitution effettuata da sinistra verso destra

Sostituisce a run-time un comando con l'output da lui prodotto.

`riga di comando` o **\$(riga di comando)** viene **eseguita** ma il suo **output viene sostituito** al posto del comando poi tale linea viene ulteriormente eseguita. Non è possibile annidare più command substitution.

→ Word splitting

→ Pathname expansion

È una **sostituzione di stringhe** che contengono metacaratteri:

- ♦ **?** sostituito da un **singolo carattere**
- ♦ ***** sostituito da una qualsiasi **sequenza** di **caratteri**
- ♦ **[elenco]** contiene vari caratteri di cui solo uno può andare a sostituire la stringa.
 - **[[:digit:]]** che può essere sostituito da **una cifra**,
 - **[[:upper:]]** che può essere sostituito da un **carattere maiuscolo**
 - **[[:lower:]]** che può essere sostituito da un **carattere minuscolo**

Possono essere utilizzate delle sequenze che devono essere contenute in parentesi graffe

→ Quote removal

Variabili

\${nomeVar} è la sintassi per la **dichiarazione di una variabile**, nel caso la variabile sia abbastanza isolata è possibile omettere le parentesi, per eliminarla si utilizza il comando **unset nomeVar**.

nomeVariabile=valore serve per **assegnare un valore** ad una variabile ed è essenziale che l'assegnamento avvenga senza nessuno spazio

es. VAR=5 oppure \${VAR}=5

- **nomeVariabile =valore** fa vedere tutta la parte prima dell'uguale come un comando
- **nomeVariabile= valore** la parte dopo l'uguale viene vista come un comando a cui viene passata la parte prima dell'uguale nell'ambiente di esecuzione

\${!Variabile} permette, quando si ha una variabile che contiene il valore di un'altra, di **accedere al valore della prima** variabile sfruttando la seconda

es VAR=5 e NEW=VAR allora echo \${!NEW} stampa 5

`${#Var}` espande il nome della variabile in una **stringa composta da cifre** che rappresentano il numero di caratteri che contiene la variabile

es. `VAR="ciao"; echo ${#VAR};` → produce in output 4

`${VAR:offset}` **sottostringa** che parte dal **offset-esimo carattere** del contenuto della variabile di nome VAR

es. `VAR="CIAO"; echo ${VAR:1};` → stampa "IAO"

`${VAR:offset:length}` **sottostringa lunga length** che **parte dal offset-esimo carattere** del contenuto della variabile di nome VAR

es `VAR="CIAO"; echo ${VAR:0:1};` → stampa "C"

Variabili d'ambiente e variabili locali

Variabili d'ambiente: sono variabili ereditate come copia dai processi figli

→ La variabile **PATH** contiene una serie di percorsi su file system concatenati in modo ordinato. Tale variabile viene utilizzata per **cercare gli eseguibili** e **comandi** bash

Variabili locali: sono variabili che non vengono mai ereditate automaticamente da un processo figlio.

→ Per fare in modo che **vengano ereditate** si usa il comando **export**

File

Permessi su file

Tutti i file hanno permessi diversi per proprietario, utenti del gruppo e tutti gli altri utenti:

user			<u>group</u>			<u>others</u>		
R	W	X	R	W	X	R	W	X
4	2	1	4	2	1	4	2	1

Per cambiarli si utilizza il comando **chmod +<permesso> nomeFile** oppure **chmod <somma valori> nomeFile**, se si utilizza la notazione numerica.

Per cambiare proprietario e gruppo invece si utilizzano **chown** e **chgrp**

Permessi sulle directory

Il significato dei permessi sulle directory è diverso da quello che si applica sui normali file:

Il permesso di lettura **R** **consente di fare il listing**, quindi usare il comando `ls`, sulla directory per vedere cosa contiene

Il permesso di scrittura **W** consente di **modificare, creare o eliminare i file**

- **rmdir** elimina la directory ma non se al suo interno è presente del contenuto
- **rm -r -f** percorso che va ad eliminare tutto quello che c'è dentro directory e figli

Il permesso di esecuzione **X** consente di **far diventare tale directory quella corrente**, quindi è possibile utilizzare il comando `cd` per entrare in quella directory.

File Nascosti

I **file nascosti** sono file il cui nome inizia con **.nome.tipo** e sono file che non vengono visualizzati tramite le normali `ls` a occorre utilizzare l'opzione **ls -a**. È da notare la differenza del primo carattere:

- **-** nel caso dei **file**
- **d** nel caso di **directory**
- **l** nel caso di **link**
- **p** nel caso delle **pipe**

Subshell

Una **subshell** è una **shell figlia creata da un'altra shell detta padre** alla quale viene passata una copia priva di variabili locali dell'ambiente di esecuzione del padre.

- possibile vedere **quali variabili ne fanno parte** con il comando **env**
- trasformare una **variabile in una variabile d'ambiente** si utilizza il comando **export**.

Quando una shell deve eseguire uno script esegue in ordine una serie di operazioni:

1. Legge la **prima riga** dove può essere **indicato** con la sintassi **#!percorsoInterprete** chi è **l'interprete che deve eseguire** lo script. In una qualsiasi altra riga tale sintassi **#** è un commento.
2. Viene **creata una subshell** in cui il nome dello script viene passato come argomento

Comando set

Comando `set -a` successivamente al quale **tutte le variabili saranno d'ambiente**

→ per **creare variabili locali successive** si utilizza il comando `export -n`

Comando `set +a` successivamente al quale le **variabili create sono locali**.

Esecuzione senza subshell

Una **shell** può eseguire uno script senza creare una subshell tramite i comandi **source nomeScript** oppure **./nomeScript** i quali non fanno **eseguire** lo script nella sua interezza ma **solo il suo contenuto saltando la prima riga speciale**.

Tipi di shell

La shell `bash` si comporta in maniera diversa a seconda di quali argomenti a riga di comando le vengono passati nel momento in cui ne viene lanciata l'esecuzione.

Shell non interattiva è una shell figlia che esegue script

→ lanciata con argomenti `-c percorso_script_da_eseguire`

Shell interattiva di login è la shell che vediamo all'inizio nella finestra di terminale

→ lanciata senza nessuno degli argomenti `-c -l --login`

Shell interattiva non di login è come la shell non di login, ma inizia chiedendo user e password

→ lanciata con argomenti `-l` oppure `--login`

Valutazione aritmetica

E' possibile valutare una stringa come se fosse un'espressione costituita da operazioni aritmetiche tra soli numeri interi.

→ `(())` esegue **tutta una riga** di comando che racchiude **valutando aritmeticamente gli operandi**

es. `((NUM=3+2))` → `NUM=5`

→ `$(())` esegue **solo una parte** di una riga di comando, che deve essere una espressione

es. `echo $((12+2))` → stampa 14

Le valutazioni aritmetiche possono essere utilizzate con:

- degli operatori aritmetici `+` `-` `*` `/` `%`
- degli assegnamenti
- delle parentesi tonde `()` per accorpare operazioni e modificare precedenze

Exit status

Ogni programma o comando **restituisce un valore numerico** compreso **tra 0 e 255** per indicare se c'è stato un errore durante l'esecuzione oppure se tutto è andato bene

→ **0** indica che è **andato tutto bene**

→ Un risultato **diverso da zero** indica **errore**

Altrimenti fornisce **risultati diversi a seconda di ciò che sto valutando**:

- Indica 0 se il risultato logico è vero
- Numero diverso da zero se il risultato logico è falso
- Se un'operazione aritmetica da un risultato diverso da zero indica 0
- Diverso da zero altrimenti

Per restituire il risultato in uno script bash si usa il comando **exit**, tale risultato viene catturato utilizzando la variabile **\$?** modificata ogni volta che un programma o un comando termina

es. `exit 9` → fa terminare lo script e restituisce 9

`echo {$?}` → stampa 9

Exit Code Number	Meaning	Example	Comments
1	catchall for general errors	let "var1 = 1/0"	miscellaneous errors, such as "divide by zero"
2	misuse of shell builtins, according to Bash documentation		Seldom seen, usually defaults to exit code 1
126	command invoked cannot execute		permission problem or command is not an executable
127	"command not found"		possible problem with \$PATH or a typo
128	invalid argument to <code>exit</code>	exit 3.14159	<code>exit</code> takes only integer args in the range 0 – 255
128+n	fatal error signal "n"	kill -9 \$PPID of script	\$? returns 137 (128 + 9)
130	script terminated by Control-C		Control-C is fatal error signal 2, (130 = 128 + 2, see above)
255*	exit status out of range	exit -1	<code>exit</code> takes only integer args in the range 0 – 255

Sequenze di comandi condizionali e non

Sono un **elenco di comandi da lanciare** in esecuzione **in successione**:

- Comando semplice (o chiamata di script o di eseguibile binario)
es. `cd ./script.sh primo.exe`
- Espressione valutata aritmeticamente
es. `((VAR=(5+3)*(2+$VAR)))`
- Sequenza di comandi connessi da | pipe
es. `cat file.txt | grep stringa`
- Sequenza di comandi condizionali
es. `gcc file.c && ./file.exe`
- Raggruppamento di comandi
es. `(cat file1.c ; cat file 2,c)`
- Espressioni condizionale
es. `[[-e file.txt && $1 -gt 13]]`

Il risultato **Exit Status** restituito da una lista di comandi è l'Exit Status restituito dall'ultimo comando che è stato lanciato dalla lista di comandi stessa, può inoltre capitare che l'**ultimo comando eseguito non sia l'ultimo** della lista

Sequenze non condizionali

Il metacarattere ";" viene **utilizzato per eseguire due o più comandi in sequenza** ed indica la fine degli argomenti passati a ciascun comando riga di comando

es. `date ; ls /usr/vittorio/ ; pwd`

Sequenze condizionali

"||" viene utilizzato per eseguire in sequenza ma il **secondo comando** viene **eseguito solo se il primo termina con un exit code diverso da 0** (failure)

"&&" viene utilizzato per eseguire in sequenza, ma il **secondo comando** viene **eseguito solo se il primo termina con un exit code uguale a 0** (success)

es. Eseguire il secondo comando in caso di successo del primo

`$ gcc prog.c -o prog && prog`

es. Eseguire il secondo comando in caso di fallimento del primo

`$ gcc prog.c echo Compilazione fallita`

Loop e flussi

For

Dentro il for (()) sono sempre valutate aritmeticamente:

- I. `for varname in elencoword ; do list ; done`
- II. `for ((expr1 ; expr2 ; expr3)) ; do list ; done`

If

```
if listA ; then listB ;  
[ elif listC ; then listD ; ] ... [ else listZ ; ]  
fi
```

While

```
while list ; do list ; done
```

Stream I/O

Il file descriptor è un numero intero utilizzato per accedere ad un file. Il sistema operativo mantiene una **tabella dei file aperti**, non visibile ai processi che ne hanno una propria, in cui per ogni indice intero sono contenute **informazioni sul file aperto**.

Quando un processo inizia l'esecuzione vengono inizializzati dei file descriptor standard che fanno **riferimento ai flussi** predefiniti di I/O:

- **Standard INPUT** (stdin) con file descriptor 0
- **Standard OUTPUT** (stdout) con file descriptor 1
- **Standard ERROR** (stderr) con file descriptor 2

Quando viene creato un processo figlio, esso **ottiene una copia della tabella dei file** aperti **del padre**, quindi **padre e figlio possono scrivere sugli stessi stream** e potrebbe risultare un problema generando una competizione per scritture e letture.

Espressioni condizionali

Le espressioni condizionali sono dei **comandi** che restituiscono un **exit status a seconda della condizione valutata**.

Si riconoscono perché utilizzano la sintassi `[[espressione]]` mettendo uno spazio tra le condizioni e le parentesi

es. `[[4<5]]` → sintassi corretta

es. `[[4<5]]` → sintassi sbagliata

Al cui interno sono presenti particolari operatori per la verifica delle condizioni e operatori logici per poter comporre logicamente:

→ **!** not

→ **&&** and

Occorre distinguere l'**&&** come and **logico** e l'**&&** come **operatore in sequenze di comandi**

es. `[[cond1 && cond2]]` → operatore logico

es. `[[]] && altro` → operatore in sequenze di comandi

→ **||** or

Sono permesse:

- Variable expansion
- Valutazioni aritmetiche con `$(())`
- command substitution
- Process substitution
- Quote removal ma solo negli operandi

Nelle **versioni più vecchie** di bash non funzionano le espressioni con `[[]]` e veniva utilizzata la sintassi **`[condizioni]`** o **`test condizioni`** in cui è permesso **comporre espressioni ma solamente con operatori specifici**:

- **`-a`** (and)
- **`-o`** (or)
- **`!`** (negazione)

Operatori per confronti aritmetici

- `-eq`** (equal) uguale
- `-ne`** (not-equal) diverso
- `-le`** (less or equal) minore uguale
- `-lt`** minore stretto
- `-ge`** (greater or equal) maggiore uguale
- `-gt`** maggiore stretto

Condizioni sui file

- `-e`** verifica se un certo file **esiste o non esiste**
- `-d`** verifica se un percorso esiste ed è una directory
- `-f`** verifica se un percorso **esiste ed è un file normale**
- `-h`** verifica se un percorso esiste ed è un link
- `-r`** verifica se un percorso esiste e che abbia **permessi di lettura**
- `-w`** verifica se un file esiste e che abbia permessi di scrittura
- `-x`** verifica che un file abbia **permessi di esecuzione**
- `-s`** verifica se non è vuoto
- `-t`** verifica se un numero è un **file descriptor attivo**
- `-O`** verifica se il file esiste ed è proprietà dell'effective user
- `-G`** verifica se il file esiste ed è **proprietà dell'effective group**
- `-o parametro`** verifica se il parametro specificato è stato abilitato dal comando set

`file1 -nt file2`: verifica se il **file alla sua sinistra è stato modificato più recentemente** del file alla sua destra. Oppure se il file alla sua sinistra esiste e l'altro no.

file1 -ot file2: verifica che il **file alla sua sinistra sia stato modificato meno recentemente**. Oppure se il file alla sua destra esiste e l'altro no.

Operatori su stringhe

-z verifica se la stringa ha **lunghezza zero**

-n verifica se la stringa ha lunghezza diversa da zero

Negli altri operatori viene effettuato un confronto lessicografico, carattere per carattere, di due stringhe.

stringa1 == stringa2 (analogo `stringa1 = stringa2`)

stringa1 != stringa2

stringa1 > stringa2

stringa1 < stringa2

Caratteri non stampabili in una stringa

Parole aventi forma `$'charsequence'` sono **trattate in modo speciale** e possono contenere backslash-escaped characters.

Le backslash-escaped characters sono poi **sostituite come specificato nello standard**

ANSI C:

\a alert (bell)

\b backspace

\e \E an escape character **\f** form feed

\n new line

\r carriage return

\t horizontal tab

\v vertical tab

**** backslash

\' single quote

\" double quote

\nnn the eight-bit character whose value is the octal value

\xHH the eight-bit character whose value is the hexadecimal value

\cx a control-x character, as if the dollar sign had not been present.

Variabile IFS

La variabile **IFS** contiene i **caratteri** che fungono **da separatori delle parole** negli elenchi.

OLDIFS=\${IFS} → salvo il valore di default della IFS

IFS=\$'\n\' → cambio la IFS e faccio le mie operazioni

IFS=\${OLDIFS} → ripristino il valore originale

Lettura da standard input

Utilizzando il comando **read** è possibile **leggere dallo standard input** e mettere il risultato in una variabile (se tale variabile non esiste la crea).

La `read` restituisce un risultato che indica se la lettura è andata a buon fine, cioè restituisce:

→ **0** se **non si arriva a fine** file e viene **letto qualcosa**

→ **>0** se **si arriva a fine file**

Controllare se nella variabile letta c'è qualcosa dentro:

```
while read RIGA; if (( $?==0 )); then true; elif (( ${#RIGA} != 0 )); then true; else false; fi ;  
do      echo  read "${RIGA}"; done
```

OR Logico dentro espressione condizionale

```
while read RIGA; [[ $? == 0 || ${RIGA} != "" ]] ; do echo "read ${RIGA}"; done
```

```
while read RIGA; [[ $? -eq 0 || ${#RIGA} > 0 ]] ; do echo "read ${RIGA}"; done
```

```
while read RIGA; [[ $? == 0 ]] || [[ -n ${RIGA} ]] ; do echo "read ${RIGA}"; done
```

Sequenza di comandi condizionale, prosegue se exit status != 0

Con **read variabili** la IFS usa i **separatori per separare le parole** e assegnarle alle diverse variabili

es. `read A B C` e scrivo prima seconda terza

→ `A="prima" B="seconda" C="terza"`

es `read A B C D` e scrivo prima seconda terza

→ `A="prima" B="seconda" C="terza" D=""`

es. `read A B C` e scrivo prima seconda terza quarta

→ `A="prima" B="seconda" C="terza quarta"`

-n permette alla **read di leggere al massimo n caratteri**

es. `read -n 4 STRINGA` → `STRINGA` conterrà al massimo 4 caratteri (ma può contenerne di meno)

-N permette alla **read** di leggere esattamente **n** caratteri

es. `read -N 4 STRINGA` → `STRINGA` conterrà esattamente 4 caratteri

read -u <file descriptor> per indicare al comando `read` da quale file aperto deve essere effettuata la lettura

Apertura di un file

Con il comando **exec** viene effettuata l'apertura di un file di cui poi andrà specificata la modalità di apertura

→ Può essere specificato un FD

→ Si può far determinare l'FD dal sistema

In questo modo posso decidere di avere **standard input/output da un file invece che dalla tastiera** specificando il file descriptor 0 o 1 per quel file.

Modo Apertura	Utente sceglie fd (n è il numero scelto dall'utente)	Sistema sceglie fd libero e lo inserisce in variabile
Solo Lettura	<code>exec n< PercorsoFile</code>	<code>exec {NomeVar}< PercorsoFile</code>
Scrittura	<code>exec n> PercorsoFile</code>	<code>exec {NomeVar}> PercorsoFile</code>
Aggiunta in coda	<code>exec n>> PercorsoFile</code>	<code>exec {NomeVar}>> PercorsoFile</code>
Lettura e Scrittura	<code>exec n<> PercorsoFile</code>	<code>exec {NomeVar}<> PercorsoFile</code>

Una volta effettuata l'apertura è possibile **leggere o scrivere su tale file**:

es. `exec {FD}< /home/usr/mioinput.txt` → apro il file in lettura

`while read -u ${FD} StringaLetta ;` → input da tale file

`do`

`echo "ho letto: ${StringaLetta}"`

`done`

es. `exec {FD}> /home/usr/miooutput.txt` → apro il file in scrittura

`for name in pippo pippa pippi ;` → output in tale file

`do`

`echo "inserisco ${name}" 1>&${FD}`

`done`

Qualunque sia il modo di apertura (lettura scrittura o entrambi), la chiusura di un file è effettuata con `exec {FD}>&-`

es. `exec 10< /home/usr/mioinput.txt` → apro il file in lettura

`exec 10>&-` → Chiudo il file

Directory /proc/

Quando ho una shell interattiva aperta `$$` mi dice il PID della shell corrente.

In `/proc/` esiste una **sotto-directory per ciascun sotto-processo in esecuzione**, per visualizzarne il contenuto uso `ls /proc/$$/`

Nella sotto-directory propria di ciascun processo, esiste una sotto-directory `fd` in cui sono presenti dei file speciali che sono **i file aperti da quel processo**

Redirezionamenti di Stream di I/O

Redirezionamento a livello di file descriptor di processi figli

Un processo figlio ottiene una copia dei file aperti dal padre che però può decidere di **cambiare gli stream** da far utilizzare al figlio.

Il **FD** viene comunque **ereditato a livello di process ID** ma **cambia il canale**. Questo redirezionamento viene effettuato solo quando viene passata la tabella fra due shell.

Auto-ridirezionamento

Avviene quando un FD viene associato ad un altro file, il **file descriptor** rimane **lo stesso** del file originale ma **gli stream cambiano** diventando quelli del nuovo file.

Ridirezionamenti:

<code><</code>	ricevere input da file.
<code>></code>	mandare std output verso file eliminando il vecchio contenuto del file
<code>>></code>	mandare std output verso file aggiungendolo al vecchio contenuto del file
<code> </code>	ridirigere output di un programma nell' input di un altro programma

Si possono **ridirezionare** assieme **standard output e standard error** su uno stesso file o su file diversi sovrascrivendo il vecchio contenuto:

```
program &> nome_file_error_and_output
program 2> nome_file_error > nome_file_output
```

I redirezionamenti **input ed output** possono essere fatti **contemporaneamente**:

```
program < nome_file_input > nome_file_output
```

Redirezionamenti con < >

N> NomeFileTarget → ridireziona il file descriptor N sul file Target.

Viene usato >> per **append**

<N NomeFileSource → ridireziona il file con nome NomeFileSource sul file descriptor N del programma specificato alla sinistra dell'operatore.

Redirezionamenti con |

program1 ; program2 ; program3

→ programmi eseguiti **uno dopo l'altro**

program1 | program2 | program3

→ programmi **partono assieme** e l'output di un programma viene ridirezionato nell'input del programma successivo.

Redirezionamento per blocchi di comandi

```
NUM=1
echo "${NUM}"
if (( "${NUM}" <= "3" )) ;
    then ((NUM=${NUM}+1))
echo "${NUM}"
else ((NUM=${NUM}+2))
    echo "${NUM}"
fi > pippo.txt echo "${NUM}"
```

→ redirezionamento di **tutto il blocco**
dei comandi **tra if e fi**

```
NUM=1
echo "${NUM}"
if (( "${NUM}" <= "3" )) ; then
    ((NUM=${NUM}+1))
    echo "${NUM}"
    echo "Nuovo" > mio.txt
else ((NUM=${NUM}+2))
    echo "${NUM}"
fi > pippo.txt echo "${NUM}"
```

→ redirezionamento di **tutto il blocco**
dei comandi **tra if e fi** **tranne** che per
il redirezionamento sul **file mio.txt**

Here documents

<<word fa ridirezionare in input tutto quello che compare **dopo word** **fino a** dove **word** compare ancora all'**inizio di una riga**.

```

es. while read A B C ; do echo $B ; done <<FINE
uno due tre quattro
alfa beta gamma          → da in output: due
gatto cane                beta
FINE                      cane
echo ciao                 ciao

```

Here strings

<<<word ridireziona nell'input la **prima parola** che compare **subito dopo <<<**.

```

es. read A B C <<< alfa
echo 1 $A 2 $B 3 $C → produce in output 1 alfa 2 3

```

Raggruppamenti di comandi

cmd1 ; cmd2 ; cmd3 >out.txt → il **ridirezionamento** in questo modo viene fatto solo sull'**ultimo dei comandi**

```

es. ls; pwd; whoami > out.txt
→ Visualizzo nomi files in directory corrente /home/vittorio
→ Dentro il file out.txt trovo vittorio

```

(cmd1 ; cmd2 ; cmd3) >out.txt → i tre comandi vengono eseguiti in una bash figlia e il **ridirezionamento** in questo modo viene applicato **a tale shell figlia**

```

es. (ls ; pwd ; whoami) >out.txt
→ Non visualizzo nulla
→ Dentro out.txt: a1B a2B aB akB akmB akmtB /home/vittorio vittorio

```

Concatenazione stdout

(cat file1.txt ; cat file2.txt) | grep stringa → L'output del **comando tra parentesi** è la **concatenazione dei singoli comandi** che poi va in input al terzo comando

Concatenazione stdout e stderr

(`cat file1.txt ; cat file2.txt`) `& grep stringa` → vengono concatenati sia standard error che standard output dei due comandi tra parentesi e poi passati in input al terzo comando

concatenazione stdin

`cat file.txt | (read RIGA1 ; usa RIGA1 ; read RIGA2 ; usa RIGA2)` → l'**output del primo** comando viene **rediretto** e usato in sequenza dai comandi della shell figlia

GNU Coreutils

Ci sono una serie di comandi che lavorano su righe di testo forniti dal pacchetto **coreutils**, tutti i comandi hanno la particolarità di **accettare input sia da file che stdin** (se non specificato nulla).

Tali programmi sono:

- head e tail
- sed
- cut
- cat
- grep
- tee

Comando grep

grep stringa nameFile legge delle righe e **cerca la stringa** se la trova manda in output le righe che la contengono (se non specificato nulla legge da stdin).

es. `grep gatto`

→ scrivo: ciao

cane

flavio gatto merda

in output da grep ho flavio gatto merda

Comandi tail e head

tail -n k file.txt manda sullo **standard output le ultime k righe di un file** (se si digita da stdin specificare la fine con control D)

es. tail -n 2

→ Scrivo

mio

tuo

suo

→ Output da tail

tuo

suo

tail -f file.txt → il comando tail rimane in attesa controllando il file. Se al file vengono **aggiunte righe** allora il comando **le renderà visibili** (per terminare posso utilizzare control C).

Il comando **head** è analogo ma **riguarda le prime righe** di un file

Comando tee

cmd | tee file.txt duplica l'output del comando che ne produce uno salvandolo su file e permettendo al contempo di visualizzarlo a video

Comando sed

Il comando permette di **editare** delle **linee di testo**

sed -i <modifica> file.txt o **sed --in-place <modifica>**

→ **modifica** l'**interno del file** che passo per nome

sed 's/str1/str2/g'

→ **s** è il **comando di sostituzione** a cui segue **str1** da sostituire e **str2** che deve prendere il suo posto mentre **g** sta a dire che vanno modificate tutte le occorrenze della stringa.

sed 's/word1/word2/' file.txt

→ **Sostituisce** la prima occorrenza di **word1 con word2** in ciascuna riga del file

sed 's/char//' file.txt

→ **Rimuove il primo tra i caratteri char** che trova in **ciascuna riga** del file

sed 's/^./' file.txt

→ **Rimuove il carattere in prima posizione** di ogni linea.

^ significa inizio linea, . significa un carattere qualunque

sed 's/.\$/' file.txt

→ **Rimuove l'ultimo carattere** di ogni linea.

\$ significa fine linea

sed 's/./;/s/.\$/' file.txt

→ Eseguo due **rimozioni insieme** (;)

sed 's/.../'

→ **Rimuove i primi 3 caratteri** ad inizio linea.

sed -r 's/.{k}'

→ **Rimuove i primi k caratteri** ad inizio linea

sed -r 's/(.{3}).*/\1/'

→ **Rimuove tutto tranne i primi n caratteri** in una linea

sed -r 's/.*(.{3})/\1/'

→ **Rimuove tutto tranne gli ultimi n caratteri** di un file

's/[char1char2char3]/g'

→ **Rimuove tutte le occorrenze di più caratteri**

sed 's/char/k'

→ **Rimuove le k occorrenze di un carattere** in tutte le linee

sed 's/char.*/'

→ **Rimuove tutta la linea dopo un carattere**

sed 's/[a-zA-Z0-9]/g'

→ **Rimuove tutti i caratteri alfanumerici** in ogni linea

Comando cut

Il comando **cut** viene utilizzato per **eliminare un certo sottoinsieme di caratteri**, per specificare più di uno vengono separati da virgola.

cut -b k

→ -b consente di mandare in output **solo il k-esimo carattere**

es. cut -b 2

→ scrivo: abc

→ in output ho b

cut -b k-n

→ manda in output i caratteri **dal k-esimo fino all'n-esimo**

es. cut -b 1-2

→ scrivo: abc

→ in output ho ab

cut -b n-

→ manda in output i caratteri **dall'n-esimo in poi**

es. cut -b 2-

→ scrivo: amaca

→ in output ho maca

cut -b -n

→ manda in output i caratteri fino **al'n-esimo compreso**

es. cut -b -3

→ scrivo: amaca

→ in output ho ama

Variabile random

La variabile **\$RANDOM** genera dei numeri casuali

→ Usare **$\$((\$RANDOM \% k+1))$** per ottenere i numeri nell'**intervallo 0,...,k**

→ Usare **$\$((n + ($RANDOM \% k+1)))$** per ottenere i numeri nell'**intervallo n,...,n+k**

→ **RANDOM=num** assume tutte le volte la **stessa sequenza di numeri casuali**

Terminazione di un terminale di controllo

I processi vengono uccisi una volta chiuso un terminale, per fare in modo che un processo sopravviva alla chiusura occorre sganciare il processo dal terminale:

→ **nohup creare** un **processo sganciato** fin da subito dal terminale di controllo

→ **disown -[ar] jobs** dopo che un processo è creato normalmente lo **sgancia dal gruppo di processi del terminale**

Processi in foreground e background

Processo in foreground: prende controllo della shell fino al termine della sua esecuzione

Processo in background: viene eseguito in parallelo rispetto all'esecuzione della bash, utilizzano comunque stdin, stdout e stderr del terminale ma **è possibile l'interazione**.

→ Si dicono **job** i processi in background o sospesi solo figli di quella shell

Comandi per job control

- **&** lancia un processo direttamente in background (in **\$!** trovo il pid) prova &
es. (cmd1 | cmd2) & → esegue la shell figlia in bg
- **ctrl Z** **sospende** un processo in foreground
- **ctrl C** **termina** un processo in foreground
- **bg** riprende l'esecuzione in background di un **processo sospeso**
- **jobs** produce una **lista numerata dei processi in background o sospesi** il numero tra parentesi è un **indice del job** che si usa per gestirlo usando il carattere %
- **fg %n** **porta in foreground** un processo sospeso
- **\$!** contiene il **pid** del processo
- **kill** **elimina il processo** specificato dal proprio identificatore pid oppure specificato dal numero del job
es. kill 6152 → dove 6152 è il pid del processo
kill %2 → dove 2 è il numero del job

Comando wait

wait \${PID1} \${PID2} → attende la fine dell'esecuzione dei processi direttamente figli, restituisce l'**exit status dell'ultimo processo**

wait → attende la **terminazione di tutti i processi** figli, non restituisce exit status

Processi zombie, processi Orfani e processo init

La **wait** serve al sistema operativo per sapere se è possibile rilasciare tutte le risorse relative al processo.

- Processo **zombie**: **processo figlio morto** di cui il **padre** non ha ancora fatto la wait
- Processi **Orfani**: il cui processo **padre** termina senza aver fatto la wait

→ I processi orfani vengono **adottati dal processo init** che **fa la wait** una volta terminati.

Manipolazione di stringhe

`${VAR%%pattern}` **rimuove il più lungo suffisso** che fa match con pattern

es. VAR="[13] qualcosa con [o] fine"

echo `${VAR%%*}` → stampa "[13"

`${VAR%pattern}` **rimuove il più corto suffisso** che fa match con pattern

es. VAR="[13] qualcosa con [o] fine"

echo `${VAR%}` → stampa "[13] qualcosa con [o"

`${VAR##pattern}` **rimuove il più lungo prefisso** che fa match con pattern

es. VAR="[13] qualcosa con [o] fine"

echo `${VAR##[]}` → stampa "o] fine"

`${VAR#pattern}` **rimuove il più corto prefisso** che fa match con pattern

es. VAR="[13] qualcosa con [o] fine"

echo `${VAR#[]}` → stampa "[13] qualcosa con [o] fine"

`${VAR/pattern/string}` **sostituisce** la **sottostringa piu' lunga** che fa match con il pattern **con string**

es. VAR="alfabetagamma"

echo `${VAR/beta/SOST}` → stampa "alfaSOSTgamma"

`${VAR:offset}` **sottostringa** che parte dal **offset-esimo carattere** del contenuto della variabile di nome VAR

es. VAR="CIAO"; echo `${VAR:1}`; → stampa "IAO""

`${VAR:offset:length}` **sottostringa lunga length** che **parte dal offset-esimo carattere** del contenuto della variabile di nome VAR

es VAR="CIAO"; echo `${VAR:0:1}`; → stampa "C"

`${VAR//pattern/string}` **sostituisce tutte le occorrenze** di pattern

`${VAR/#pattern/string}` quando dopo il primo / c'è un # allora il pattern viene **sostituito solo se si trova all' inizio** della variabile.

`${VAR/%pattern/string}` quando dopo il primo / c'è un % allora il pattern viene **sostituito solo se si trova alla fine** della variabile.

Comando find

find percorso è usato per **cercare file o directory** che corrispondono ad un nome **iniziando da percorso**

es. `find /usr/` → cerca tutti i file e directory della forma `/usr/*`

Si possono usare varie opzioni:

- **-type** quando voglio **specificare il tipo di file** che cerco
 - ◆ **-d** se cerchiamo una directory
 - ◆ **-f** se si cerca un file vero e proprio
- **-iname "string"** cerca i **file il cui nome è string** (case insensitive)
es. `find /usr/ -iname "*STD*" == find /usr/ -iname "*std"`
- **-name "string"** cerca i **file il cui nome è string** (case sensitive)
es. `find /usr/ -name "*STD*" != find /usr/ -name "*std"`
- **-maxdepth n** cerca **al massimo fino all'n-esimo livello** del sottoalbero
es. `find /usr/ -maxdepth 2` → cerca al massimo fino al secondo livello
- **-mindepth n** cerca **a partire dall'n-esimo livello** del sottoalbero in poi
es. `find /usr/ -mindepth 2` → cerca dal secondo livello in poi
- **-exec comando '{ }' \;** usato per **eseguire dei comandi** sui file cercati
es. `find /usr/ -maxdepth 2 -exec head -n 1 '{ }' \;` → di tutti i file trovati
stampa la prima riga
- **-print** **stampa il nome** del file su stdout

Installazione dei pacchetti

L'installazione di pacchetti viene fatta con:

- **sudo** che permette di **eseguire il comando come amministratore**
- **apt-get** installa e disinstalla i pacchetti

Dobbiamo utilizzare il comando **sudo apt-get update** per **cercare in locale i pacchetti installabili** prima di poter procedere all'**installazione vera e propria** dei pacchetti fatta con **sudo apt-get install namepkg**.

Con apt-get possiamo anche:

- **disinstallare un pacchetto** e tutti i files di configurazione **sudo apt-get purge nomepkg**
- **reinstallare un pacchetto** sovrascrivendo la vecchia installazione **sudo apt-get install --reinstall nomepkg**
- **rimuovere pacchetti** inutilizzati **sudo apt-get autoremove**

Comando sort

Il comando **sort** viene utilizzato per **ordinare file**, l'opzione **-k n** ordina in base all'n-esimo argomento

Comando uniq

Il comando **uniq** serve per **filtrare linee adiacenti uguali** dall'input scrivendo in output, **senza opzioni** le ricorrenze non vengono considerate.

Dato che viene **applicato su linee adiacenti** prima si deve **fare un sort**.