MICRO TEST: Nome:_____ Cognome:_____

1) Quali di questi indirizzi IP sono di classe A:

a) 128.34.17.230
b) 120.56.117.3
c) 192.186.13.4
d) 191.255.255.47
e) non so

2) Quali di questi indirizzi IP sono di classe B:

a) 128.34.17.230
b) 120.56.117.3
c) 192.186.13.4
d) 191.255.255.47
e) non so

3) Quali di questi indirizzi IP sono di classe C:

a) 128.34.17.230
b) 120.56.117.3
c) 192.186.13.4
d) 191.255.255.47
e) non so

4) Qual'è il numero massimo di Hosts usando la classe A:

a) 65,536
b) 16,777,216
c) 256
d) illimitato
e) 0
f) non so

5) Quale di questi indirizzi identifica una rete:

a) 10.0.0.0
b) 192.168.1.0
c) 192.168.1.1
d) 10.75.10.1
e) 127.0.0.1
f) non so

6) Con l'indirizzo IP nel formato 192.168.1.72/24 cosa definisco oltre all'indirizzo dell'host:

a) una rete di classe A

b) una rete di classe B
c) una rete di classe C
d) una maschera di rete 255.255.255.0
e) una maschera di rete 255.255.0.0
f) una maschera di rete 255.0.0.0
g) la forma non è corretta
h) non so

7) Che cos'è il DNS:

a) Un sistema per l'implementazione della crittografia
b) Un database di nomi di dominio associati ai relativi IP
c) Un sistema per l'implementazione delle regole di accesso
d) Un database di nomi di hosts associati ai relativi IP
e) non so

8) Cosa si intende quando si definisce un sistema con il termine multitasking:

a) Che può gestire più utenti tramite autenticazione
b) Che può gestire diverse porte allo stesso tempo
c) Che può controllare in tempo reale gli accessi degli utenti remoti
d) Che può gestire più operazioni contemporaneamente
e) non so

9) Cosa fa il comando ping?:

a) Associato ad un indirizzo IP o ad un nome di un host valido stabilisce se esiste un collegamento di rete con quest'ultimo
b) Associato ad un indirizzo IP o ad un nome di host valido identifica gli utenti autenticati con quest'ultimo
c) Associato ad un indirizzo IP o ad un nome di un host valido stabilisce se esiste un collegamento ipertestuale con quest'ultimo
d) non so

10) Cosa fa il comando telnet 192.168.1.1?:

a) Stabilisce una sessione di collegamento con il server telnet 192.168.1.1
b) Stabilisce una sessione di collegamento con il client telnet 192.168.1.1
c) Chiude una sessione di collegamento con il server telnet 192.168.1.1
d) Chiude una sessione di collegamento con il client telnet 192.168.1.1
e) non so

L'accensione

Terminato il caricamento il sistema linux ci presenta la schermata di login; a questo punto siamo pronti per autenticarci utilizzando l'utente "support" e la password "password". Durante il login il sistema ci fornisce indicazioni sulla distribuzione "Red Hat", sulla release "7.1", sul nickname "Seawolf" e sulla versione del kernel "2.4.2-2". Il secondo numero, il "4", identifica che si tratta di una release stabile, se fosse stato dispari saremo di fronte ad una beta release, cioè una versione sperimentale con nuove caratteristiche non completamente testate.

```
Red Hat Linux release 7.1 (Seawolf)
Kernel 2.4.2-2 on an i686
login:
```

una volta effettuato il login (sperando che tutti quanti avete digitato correttamente l'username e la password) vi trovate di fronte alla shell bash di sistema. Dopo il login il sistema visualizza una serie di informazoni che indicano l'ultimo login effettuato per la macchina che state utilizzando e se avete eventuali messaggi di posta.

La Shell

La shell è molto importante perchè consente di metterci nelle condizioni per poter impartire i comandi al kernel, che è il cuore del sistema operativo (esattamente la stessa cosa succedeva qualche anno fa con l'MS-DOS - Chi di voi non ha mai utilizzato l'MS-DOS?).

In Linux vengono principalmente utilizzate quattro shell:

- BASH (Bourne Again SHell)
- PDKSH (Public Domain Korn SHell)
- TCSH
- Z

Se vogliamo visualizzare tutte le shell supportate dalla nostra distribuzione, possiamo impartire il comando:

```
[roberto@diamondhead roberto]$ more /etc/shells
```

Vedremo tutte le shells disponibili del nostro sistema operativo.

Con le ultime distribuzioni di Linux abbiamo anche la BASH2 che è una BASH un pò più evoluta.
Può essere utilizzata una qualsiasi di queste shell ma inizialmente, nel sistema operativo Linux (almeno su questa distibuzione), viene installata la shell BASH, ed è proprio questa che tratteremo.

In qualsiasi momento possiamo cambiare shell digitando uno dei nomi di shell che troviamo nel file /etc/shells. Proviamo a digitare:

```
[roberto@diamondhead roberto]$ bash2
```

adesso ci troviamo nella shell bash2. Per uscire da una shell basta digitare:

```
[roberto@diamondhead roberto]$ exit
```

E si torna alla shell originale. Proviamo ora ad entrare nella shell sh:

```
[roberto@diamondhead roberto]$ sh
sh-2.04$
```

vediamo che il prompt dei comandi si presenta leggermente diverso da quello precedente. Possiamo uscire da questa shell anche premendo contemporaneamente CTRL + D.

```
sh-2.04$ [CTRL +D]
```

Una volta fatto il login sul sistema, come prompt di comandi abbiamo una serie di informazioni che ci identifica:
"`[roberto@diamondhead roberto]$`", dove roberto è il nome dell'utente separato da una chiocciola, segue poi il nome dell'host e il nome della directory dove attualmente ci troviamo "/home/roberto".
Il dollaro finale sta ad indicare che ci troviamo in modalità utente normale, se si fosse fatto il login come root, cioè come amministratore, il simbolo sarebbe un # (diesis).
In questo momento possiamo impartire tutti i comandi che vogliamo. Prima però di proseguire dovremo far partire il server grafico in modo che voi possiate vedere cosa sto digitando nel mio terminale. Questo per comodità, in ogni caso utilizzeremo i tools grafici di sistema il meno possibile:

```
[roberto@diamondhead roberto]$ startx
```

Una volta caricato il desktop apriremo la console di sistema cliccando sul terminale (la terza icona in basso partendo da sinistra). Bene, adesso proviamo a digitare ls:

```
[roberto@diamondhead roberto]$ ls
```

vedremo tutti i file contenuti nella directory corrente. Se vogliamo vedere tutti i file, anche quelli nascosti, incolonnati con i relativi permessi digitiamo:

```
[roberto@diamondhead roberto]$ ls -la
```

l'opzione "l" subito dopo il trattino mi fa vedere i dettagli sui file, come permessi, nome dell'utente, nome del gruppo, data di creazione e così via, mentre il flag "a" mi visualizza i file nascosti, quelli che iniziano con il punto (sia se si tratta di file normali - con il colore vere e bianco, sia se si tratta di directory - in blu). Nel caso in qui vogliamo visualizzare le directory seguite da uno slash (questo è utile se NON sia ha la possibilità di vedere i colori), possiamo utilizzare il flag "F:

```
[roberto@diamondhead roberto]$ ls -F
```

Ovviamente si può utilizzare questi flag assieme:

```
[roberto@diamondhead roberto]$ ls -laF
```

Adesso vediamo com'è organizzato il file system di Linux. Proviamo a dare il comando:

```
[roberto@diamondhead roberto]$ pwd
/home/roberto
```

vediamo in questo momento in quale directory ci troviamo, cioè in /home/roberto. Infatti pwd sta per print work directory. Proviamo a risalire di directory:

```
[roberto@diamondhead roberto]$ cd ..
[roberto@diamondhead /home]$
```

adesso, come dice anche il nostro prompt di comandi, ci troviamo nella directory /home. Quello slash prima di home sta ad indicare che il path è assoluto. Se non ci fosse lo slash sarebbe un

riferimento relativo, cioè riferito alla directory dove attualmente ci troviamo. Quindi, se noi vogliamo ritornare alla nostra home directory faremo:

```
[roberto@diamondhead /home]$ cd roberto/
```

in questo modo non c'è lo slash prima del nome di directory e quindi facciamo riferimento alla directory roberto in modo relativo, cioè rispetto a dove siamo.
Risaliamo un attimo di livello:

```
[roberto@diamondhead roberto]$ cd ..
[roberto@diamondhead /home]$
```

ora ci troviamo dentro la directory home. Se vogliamo entrare nella nostra directory roberto specificando un path assoluto possiamo scrivere:

```
[roberto@diamondhead /home]$ cd /home/roberto/
```

con lo slash iniziale si specifica un percorso assoluto. Se vogliamo andare alla radice, cioè al padre di tutte le directory scriviamo:

```
[roberto@diamondhead /home]$ cd /
[roberto@diamondhead /]$
```

diamo un:

```
[roberto@diamondhead /]$ ls
bin   dev  home  lost+found  mnt  proc  sbin      tmp  var
boot  etc  lib   misc        opt  root  tftpboot  usr
```

e vedremo tutte le principali directory del nostro file system. In sostanza è come se ci trovassimo in C: di windows.
Adesso proviamo ad andare dentro:

```
[roberto@diamondhead /]$ cd bin/
[roberto@diamondhead /bin]$ ls
```

qui troviamo parte dei comandi di linux. Vediamo che c'è il comando "ls", il comando "pwd", etc.
Risaliamo di directory e torniamo indietro alla directory radice, chiamata anche ROOT DIRECTORY:

```
[roberto@diamondhead /bin]$ cd ..
```

```
[roberto@diamondhead /]$ ls
```

andiamo dentro la directory boot:

```
[roberto@diamondhead /]$ cd boot/
[roberto@diamondhead /boot]$ ls
```

qui troviamo i programmi del kernel di linux. Questi programmi vengono caricati al momento del boot. In particolare il file "vmlinux-2.4.2-2" è il primo programma che viene caricato quando selezioniamo la voce "Linux" dal dual boot. Senza questo file Linux con partirebbe.
Torniamo ancora indietro:

```
[roberto@diamondhead /boot]$ cd ..
[roberto@diamondhead /]$ ls
```

entriamo dentro:

```
[roberto@diamondhead /]$ cd var/
[roberto@diamondhead /var]$ ls
```

qui c'è un pò di tutto. Una directory molto importante è "log". Qui dentro ci sono tutti i file di log del sistema. Se un domani vogliamo controllare gli accessi al sistema dovremo andare qui dentro. Proviamo a farlo:

```
[roberto@diamondhead /var]$ cd log/
[roberto@diamondhead log]$ ls
```

eccoli qua, organizzati per tipologia di servizi. Proviamo ora ad anadare dentro la directory /mnt/ però usando un path relativo. Intanto vediamo dove ci troviamo:

```
[roberto@diamondhead log]$ pwd
/var/log
```

ora la directory "mnt" si trova allo stesso livello di var, quindi faremo:

```
[roberto@diamondhead log]$ cd ../../mnt/
[roberto@diamondhead /mnt]$ ls
```

chiaro?

Qui dentro ci sono i dispositivi di lettura/scrittura come floppy disk, cdrom, masterizzatori etc. Più avanti vedremo anche come utilizzarli. Per ora ci basta sapere che sono qui dentro. Adesso ritorniamo alla nostra home directory (provare a chiedere come si ritorna....):

```
[roberto@diamondhead /mnt]$ cd /home/roberto/
[roberto@diamondhead roberto]$ ls
```

Se vogliamo creare in file possiamo scrivere:

```
[roberto@diamondhead roberto]$ touch pippo
```

verrà creato un file vuoto. Se vogliamo cancellarlo si usa:

```
[roberto@diamondhead roberto]$ rm pippo
```

per creare una nuova directory si scrive:

```
[roberto@diamondhead roberto]$ mkdir newdir
```

per cancellare una directory vuota:

```
[roberto@diamondhead roberto]$ rmdir newdir
```

se invece la directory NON è vuota, cioè se al suo interno ha dei file o altre directory si usa il comando:

```
[roberto@diamondhead roberto]$ rm -r newdir/
```

La shell BASH ci da anche la possibilità di correggere ciò che abbiamo digitato.

Caratteri speciali: *, ? e []

proviamo nella nostra home directory a creare dei file vuoti con il comando touch:

```
[roberto@diamondhead roberto]$ touch doc1
[roberto@diamondhead roberto]$ touch doc2
[roberto@diamondhead roberto]$ touch documento
[roberto@diamondhead roberto]$ touch doc_s
[roberto@diamondhead roberto]$ touch miodoc
[roberto@diamondhead roberto]$ touch lunedi
[roberto@diamondhead roberto]$ touch martedi
```

per visualizzare tutti i file che iniziano con "doc" digitiamo:

```
[roberto@diamondhead roberto]$ ls doc*
```

Il carattere "*" serve appunto per espandere nomi di file con 0 o più caratteri. Se si vuole visualizzare tutti i file che iniziano con doc e che sono preceduti da un solo carattere si usa il "?":

```
[roberto@diamondhead roberto]$ ls doc?
```

Bisogna prestare attenzione quando si usano questi caratteri speciali, soprattutto quando si usano con comandi come rm (remove):

```
[roberto@diamondhead roberto]$ rm doc*
```

se non è attivata la modalità interattiva questo comando ci cancella nella directory corrente, senza conferme, tutti i file che iniziano con "doc".
Per dovere di cronaca se ci scappa un:

```
[root@diamondhead roberto]$ rm -rf /
```

come amministratore di sistema (root), cancelleremo senza tanti complimenti l'intero sistema operativo. Ci resterebbe solo qualche directory dedicata al mantenimento dei processi in corso, ma il resto verrebbe cancellato senza possibilità di tornare indietro.

Ora cancelliamo i file che abbiamo appena creato:

```
[roberto@diamondhead roberto]$ rm lunedi
[roberto@diamondhead roberto]$ rm martedi
[roberto@diamondhead roberto]$ rm miodoc
```

e creiamo:

```
[roberto@diamondhead roberto]$ touch doc1
[roberto@diamondhead roberto]$ touch doc2
[roberto@diamondhead roberto]$ touch doc3
[roberto@diamondhead roberto]$ touch docA
[roberto@diamondhead roberto]$ touch docB
[roberto@diamondhead roberto]$ touch docC
[roberto@diamondhead roberto]$ touch documento
[roberto@diamondhead roberto]$ touch documentoA
```

se vogliamo visualizzare tutti i file che iniziano con "doc" e come terzo carattere l'"1" o il carattere "A" digiteremo:

```
[roberto@diamondhead roberto]$ ls doc[1A]
doc1  docA
```

per visualizzare tutti i file che terminano con "1" o con "A":

```
[roberto@diamondhead roberto]$ ls doc*[1A]
doc1  docA  documentoA
```

se vogliamo includere un range utilizzeremo all'interno delle parentesi quadre un trattino:

```
[roberto@diamondhead roberto]$ ls doc[1-C]
doc1  doc2  doc3  docA  docB  docC
```

Abbiamo poi un'altro operatore, che non si può definire di espansione dei file, che è la parentesi graffa. La parentesi graffa è spesso utile per generare nomi utilizzabili per creare o modificare file e directory. Per esempio, vogliamo creare tre directory e sappiamo che tutte iniziano con "doc":

```
[roberto@diamondhead roberto]$ mkdir doc{umentoB,finale,bozza}
```

questi particolari che ho appena spiegato possono sembrare a prima vista delle sciocchezze. In realtà se in un futuro vi capiterà di amministrare un sistema linux sarete presi molto spesso con la creazione di script shell che dovranno svolgere determinati compiti. Con queste "dritte" sarete in grado di scrivere codice più compatto e più adatto ad una futura manutenzione.

Standard input, standard output, standard error e redirezione

Tutto in linux viene visto come file. Anche i dispositivi di output come video e stampanti, i dispositivi di input, come mouse e tastiere; i dischi rigidi, i floppy disk, i cd-rom, i dvd, etc.
Difatti se facciamo un giro per la directory "dev":

```
[roberto@diamondhead /dev]$ ls /dev/
```

vedremo una serie interminabile di file che identificano un dispositivo. Non tutti i file che vedete qui visualizzati sono associati ad un dispositivo fisico, ma solo una minima parte.
Se vogliamo renderci conto che effettivamente linux vede anche il video come un file, possiamo fare in questo modo: bisogna prendere i privilegi di superutente:

```
[roberto@diamondhead roberto]$ su -
```

"su" sta per super user e il "-" ve lo spiegherò dopo. Attenzione, adesso tutto ciò che si ha può essere potenzialmente pericoloso. Fatto questo si cerca di identificare qual'è il nostro terminale con il comando "w":

```
[root@diamondhead /root]# w
 10:59am  up  2:27,  1 user,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
roberto  pts/0    geelong           9:46am  0.00s  0.14s  0.01s  w
```

Con il comando "w" posso sapere quali utenti attualmente sono presenti nel sistema e altre informazioni come il terminale attualmente in uso per quell'utente, cioè "pts/0". Poi si redirige lo standard input del mouse, nello standard output del terminale:

```
[root@diamondhead /root]# cat /dev/mouse > /dev/pts/0
```

cliccando i pulsanti del mouse, o muovendolo, vedremo a video una sequenza di caratteri illeggibili (molti di questi non sono visualizzati perchè sono caratteri speciali) e servono per l'interazione tra il dispositivo ed il sistema. Si possono fare altre prove, come quella di vedere i dati che passano attraverso un modem (che in norma viene identificato con /dev/ttyS0 se viene collegato alla seriale 1 oppure /dev/ttyS1 per la seriale 2 - l'equivalente di COM1 e COM2 di windows), oppure trasferire informazioni direttamente al modem da linea di comando.

Quando si esegue un comando Linux, l'output prodotto viene inserito nel canale standard output. Se scriviamo:

```
[root@diamondhead /root]# echo "ciao"
```

questa è la maniera implicita per dire al sistema di scrivere "ciao" nello standard output, cioè il video; sarebbe esattamente la stessa cosa se in modo esplicito scrivessimo:

```
[root@diamondhead /root]# echo "ciao" > /dev/pts/0
```

Se vogliamo redirezionare l'output in un file possiamo scrivere:

```
[root@diamondhead /root]# echo "ciao" > testo.txt
[root@diamondhead /root]# more testo.txt
ciao
```

Praticamente l'operazione di redirezione, in questo caso, crea il file di destinazione. Se il file esiste già verrà cancellato e solo successivamente verranno passati i dati al file (se questi esistono).

Un secondo operatore di redirezione ">>" ci consente di accodare i dati in un file:

```
[root@diamondhead /root]# echo "a te" >> testo.txt
[root@diamondhead /root]# more testo.txt
```

Quando si esegue un comando può verificarsi un errore. Per esempio, se digitiamo il comando:

```
[root@diamondhead /root]# cat pippo.c
```

e il file pippo.c non esiste, come in questo caso, avremo un messaggio d'errore:

```
cat: pippo.c: File o directory inesistente
```

questo messaggio è stato generato dallo standard error rediretto allo standard output, cioè il video. Se vogliamo catturare questo messaggio d'errore e redirigerlo su di un file possiamo digitare:

```
[root@diamondhead /root]# cat pippo.c 2> error.txt
[root@diamondhead /root]# more error.txt
cat: pippo.c: File o directory inesistente
```

in questo esempio solo lo standard error (identificato con il numero "2") viene rediretto nel file error.txt, eventuali altri messaggi diretti sullo standard output vengono regolarmente visualizzati a video.

Definizione1: A tutti i canali standard è associato un numero: i numeri 0, 1 e 2 fanno riferimento, rispettivamente, allo standard input, allo standard output e allo standard error.
Anche in questo caso utilizzando l'operatore ">>" possiamo accodare i messaggi d'errore in un file:

```
[root@diamondhead /root]# cat pippo.o 2>> error.txt
[root@diamondhead /root]# more error.txt
cat: pippo.c: File o directory inesistente
cat: pippo.o: File o directory inesistente
```

Ora digitiamo:

```
[root@diamondhead /root]# vi lettera
```

e scriviamo nell'editor:

```
ciao come stai?
```

Possiamo redirigere lo standard input in modo da ricevere i dati dal file lettera invece che dalla tastiera:

```
[root@diamondhead /root]# cat < lettera
ciao come stai?
```

in questo modo viene copiato il contenuto del file lettera nello standard input di cat. Quindi il comando cat legge lo standard input e visualizza il contenuto del file lettera.

In alcuni casi può essere necessario passare i dati da un comando all'altro. In altre parole, si deve inviare lo standard output di un comando a un altro comando e non a un file di destinazione. Per esempio vogliamo visualizzare la lista del processi attivi con il comando ps:

```
[root@diamondhead /root]# ps -ef
```

vedremo una lunga lista di processi. Questi in parole povere sono i processi che attualmente stanno girando silenziosamente nel nostro sistema. Se vogliamo vedere se per esempio sendmail è in esecuzione (sendmail è il server di posta) dovremo sfogliare ogni riga della lista alla ricerca di questa voce. Per evitare ciò potremo filtrare il comando ps con un grep (che ci consente di cercare una stringa):

```
[root@diamondhead /root]# ps -ef | grep "sendmail"
root       891     1  0 08:32 ?        00:00:00 sendmail: accepting connections
```

in questo modo abbiamo utilizzato come input del comando grep l'output generato dal comando ps. Altro esempio:

```
[root@diamondhead /root]# ls | lpr
```

invia la lista dei file della directory corrente alla stampante.

Una delle principali comodità della shell bash è il completamento automatico dei comandi e dei nomi di file.
Per esempio, proviamo a digitare per due volte il tasto Tab da terminale:

```
[root@geelong local]#
Display all 3049 possibilities? (y or n)
```

il sistema ci chiederà se vogliamo visualizzare tutti i 3049 comandi disponibili. Ma attenzione, non sono gli unici, quelli visualizzati sono quelli che effettivamente abbiamo nel nostro path, ma ce ne sono altri.

Il comando history

Definizione2: Nella shell BASH il comando history mantiene una registrazione degli ultimi comandi eseguiti, che in questo caso vengono considerati eventi. I comandi vengono numerati a partire da 1 e viene memorizzato un determinato numero di comandi (normalmente 500 o 1000).

Digitando history si hanno gli ultimi comandi impartiti preceduti da un numero.
Per scorrere i comandi in sequenza si utilizzano i tasti cursore su e giù.
Come per il completamento automatico, già visto, dei comandi e dei nomi di file con la doppia pressione del tasto TAB, con la pressione della combinazione di ESC e TAB è possibile completare gli eventi memorizzati nell'history.

Esempio:

```
[roberto@geelong roberto]$ m [ESC][TAB][ESC][TAB]

mio_archivio-2002050611*            miodoc
mio_archivio-20020506120*           mkdir
mio_archivio-200205061204           more
```

Un'altro modo per far riferimento ad un evento dell'history e quello di far precedere il numero di riferimento dell'evento con il punto esclamativo in questo modo:

```
[roberto@geelong roberto]$ !1000
su -
Password:
```

oppure quello di digitare il punto esclamativo seguito da uno spazio e da una o più lettere di riferimento all'evento:

```
[roberto@geelong roberto]$ ! e[ESC][TAB][ESC][TAB]
echo   eject  exit
```

Per vedere com'è configurato il comando history sul nostro sistema basa vedere a quale valore corrisponde la variabile HISTSIZE (che contiene un numero riferito alla quantità di comandi memorizzabili nell'history) e HISTFILE, cioè a quale NOME del file history fa riferimento. Se digitiamo:

```
[roberto@geelong roberto]$ echo $HISTFILE
/home/roberto/.bash_history
```

vedremo il path ed il file di riferimento. Con:

```
[roberto@geelong roberto]$ echo $HISTSIZE
1000
```

Il comando alias

Un alias consente di definire un nuovo nome per un comando. Esempio:

```
alias pippo='ls -la'
```

definisce l'alias pippo per il comando ls -la. Per vedere tutti gli alias utilizzati dal proprio utente digitiamo:

```
[roberto@geelong roberto]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias pippo='ls -la'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-
tilde'
```

E' possibile utilizzare anche le espressioni regolari per definire un'alias. Per definire un'alias per elencare i file di codice sorgente e di codice oggetto, ovvero i file che terminano con .c o .o, si scrive:

```
[roberto@geelong roberto]$ alias lsc='ls *.[co]'
```

per cancellare un alias:

```
[roberto@geelong roberto]$ unalias pippo
[roberto@geelong roberto]$ alias
```

I flag noclobber, ignoreeof e noglob

Adesso parliamo di alcuni flag molto utili.
Definizione3: Il flag noclobber, se abilitato, attiva il controllo che impedisce che l'output rediretto possa inavvertitamente cancellare un file.
Per esempio, creiamo un file che contiene il testo "salve":

```
[roberto@diamondhead roberto]$ echo "salve" > testo1.txt
[roberto@diamondhead roberto]$ more testo1.txt
salve
```

ora creiamo un'altro file con il testo "mondo!":

```
[roberto@diamondhead roberto]$ echo "mondo" > testo2.txt
[roberto@diamondhead roberto]$ more testo2.txt
mondo
```

adesso il file testo1.txt contiene la stringa "salve", ed il file testo2.txt contiene la stringa "mondo". Come possiamo fare (con

una operazione di redirezione) per fare in modo che il file testo1.txt prenda il contenuto del file testo2.txt, cioè la stringha "mondo"?:

```
[roberto@diamondhead roberto]$ cat testo2.txt > testo1.txt
[roberto@diamondhead roberto]$ more testo1.txt
mondo
```

in questo modo però abbiamo perso la stringa "salve". Se il comando fosse stato lanciato accidentalmente, avremo perso tutto il contenuto di testo1.txt. Se vogliamo che questo NON accade possiamo abilitare il flag noclobber. Rimettiamo a posto il file testo1.txt:

```
[roberto@diamondhead roberto]$ echo "salve" > testo1.txt
[roberto@diamondhead roberto]$ more testo1.txt
salve
```

abilitiamo noclobber:

```
[roberto@geelong roberto]$ set -o noclobber
```

adesso proviamo a rifare l'operazione di prima:

```
[roberto@diamondhead roberto]$ cat testo2.txt > testo1.txt
bash: testo1.txt: cannot overwrite existing file
```

Il file non verrà sovrascritto. Quindi l'opzione -o abilita questo controllo. Per disabilitarlo si utilizza l'opzione +o:

```
[roberto@diamondhead roberto]$ set +o noclobber
[roberto@diamondhead roberto]$ cat testo2.txt > testo1.txt
```

Per ignorare il flag noclobber e fare in modo di redirigere forzatamente il contenuto di un file in un'altro possiamo utilizzare il punto esclamativo. Allora ricostruiamo il file:

```
[roberto@diamondhead roberto]$ echo "salve" > testo1.txt
[roberto@diamondhead roberto]$ more testo1.txt
salve
[roberto@diamondhead roberto]$ more testo2.txt
mondo
```

abilitiamo noclobber in questo modo:

```
[roberto@diamondhead roberto]$ set -o noclobber
[roberto@diamondhead roberto]$ cat testo2.txt > testo1.txt
bash: testo1.txt: cannot overwrite existing file
```

quindi adesso non possiamo fare la redirezione. Forziamo ora la redirezione:

```
[roberto@diamondhead roberto]$ cat testo2.txt >! testo1.txt   (....non funziona)
```

Un'altro flag utile è ignoreeof, che consente di attivare una funzionalità che evita l'uscita della shell con la combinazione di caratteri CTRL-D:

```
[roberto@geelong roberto]$ set -o ignoreeof [INVIO]
[roberto@geelong roberto]$ [CTRL] + D
[roberto@geelong roberto]$ Use "exit" to leave the shell.
```

L'editor VI

L'editor VI è l'editor di default sia per sistemi Linux che Unix in genere. Se volete fare l'amministratore di sistema con molta probabilità vi capiterà prima o poi di collegarvi con un sistema remoto per le operazioni di amministrazione. Quasi sicuramente, se si tratta di sistemi Unix, l'unico editor disponibile è il VI. Da questo si capisce la necessità di impadronirsi almeno dei comandi base.

Ora proviamo a creare un file di testo con il VI:

```
[roberto@diamondhead roberto]$ vi elenco.txt
```

ora ci troviamo in modalità comandi. Tutti i tasti, o le sequenze di tasti, che in questo momento verranno digitati, l'editor li tratterà come comandi. Per iniziare a scrivere qualcosa basta premere il tasto i di insert:

```
prova1 prova2 prova3
pluto pippo paperino
eccetera
```

ora premiamo il testo ESC e ci ritroviamo in modalità comandi. Possiamo fare anche delle ricerche o delle sostituzioni, per esempio possiamo sostituire tutte le lettere p del nostro file con la lettera X, nel modo seguente:

come prima cosa posizioniamoci all'inizio della riga del file, premiamo ESC per assicurarci di essere in modalità comandi e poi digitiamo:

:.,$s/p/X/g

aspettate di premere INVIO. Con questa riga impartiamo all'editor VI che si tratta di un comando composto (:), che vogliamo partire dalla linea corrente (.) e proseguire fino all'ultima linea del buffer ($), la virgola (,) serve per separare l'inizio e la fine dell'area da sostituire. La s sta per substitute, quindi è la parola chiave che mi permette di fare questo tipo di operazione, segue la lettera da cercare (p) e la lettera da sostituire (X) separate da uno slash. Per finire vogliamo dire all'editor di applicare le modifiche in maniera globale (g).
Una volta modificato il nostro file possiamo salvare il tutto premendo ESC :wq!. La "w" sta per write, la "q" sta per quit, ed il punto esclamativo mi sembra che serva per non effettuare nessun altro controllo sul file, tipo se è già in uso e così via.

Variabili d'ambiente

Penso che ormai tutti voi sanno che cosè una shell: praticamente è il nostro ambiente di lavoro. Adesso cerchiamo di capire che cos'è una sottoshell.

Definizione4: Quando si esegue il login nel proprio account, Linux genera una shell per l'utente. All'interno di questa shell si possono eseguire comandi e dichiarare variabili. Inoltre è possibile creare ed eseguire file script realizzati con i comandi della shell, quindi dei file che raggruppano un insieme di comandi di sistema pronti per essere lanciati in esecuzione (un pò simili se vogliamo ai vecchi file bach dell'MS-DOS). Quando si esegue un file script della shell, il sistema genera una sottoshell. Dunque a questo punto vi sono due shell, quella alla quale si è connessi e quella generata per lo script. All'interno della shell dello script, si potrebbe eseguire un altro script (quindi uno script nidificato all'interno di un'altro script) per il quale verrà attivata una terza shell. Quando uno script termina la

propria esecuzione, si conclude anche l'esecuzione della shell relativa e l'utente torna alla shell dalla quale era partito (in sostanza la shell padre). In questo senso si può dire che esistono contemporaneamente più shell, una all'interno dell'altra.
Con un esempio definiamo uno script shell:

```
[roberto@geelong roberto]$ vi prova.sh
#!/bin/bash

COUNTER=0
clear
while [ $COUNTER = 0 ]; do
  tput cup 0 0
  echo "ciao"
done

[roberto@geelong roberto]$ chmod +x prova.sh
```

con questo comando diamo i permessi di esecuzione (x) al file prova.sh (comunque più avanti avremo modo di parlerne). Apriamo un'altro terminale (cliccate sull'icona del terminale in basso a sinistra) e da questo digitiamo:

```
[roberto@geelong roberto]$ w
  3:24pm  up  6:11,  5 users,  load average: 1.73, 1.92, 1.06
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
root     tty1     -                 9:50am  4:02m  0.40s  0.01s  /bin/sh /usr/X1
root     pts/0    -                11:22am  4:02m  0.00s  0.00s  /bin/cat
root     pts/1    -                11:22am  3:52m  0.08s  0.08s  /bin/bash
roberto  pts/3    192.168.1.2       3:14pm  0.00s  0.05s  0.01s  w
roberto  pts/4    192.168.1.2       3:15pm  9.00s  0.02s  0.01s  -bash
```

il comando w ci da un elenco degli utenti collegati al sistema in questo momento. Adesso abbiamo il primo terminale su pts/3 ed il secondo su pts/4. Dall'ultimo terminale aperto mandiamo in esecuzione lo script:

```
[roberto@geelong roberto]$ ./prova.sh
```

e vediamo che lo script va in loop. Dal terminale pts/3 digitiamo:

```
[roberto@geelong roberto]$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
....
root     23076   628  0 15:15 ?        00:00:03 in.telnetd: 192.168.1.2
root     23101 23076  0 15:15 pts/4    00:00:00 login -- roberto
roberto  24399 23101  0 15:15 pts/4    00:00:00 -bash
roberto  30747 24399  7 15:33 pts/4    00:00:00 /bin/bash ./prova.sh
roberto   1815 30747  0 15:33 pts/4    00:00:00 tput cup 0 0
```

```
roberto   1819 18123  0 15:33 pts/3    00:00:00 ps -ef
```

il comando ps ci da una lista dei processi attivi in questo momento, cioè di tutti i programmi che attualmente sono in funzione nel nostro sistema. La maggior parte di questi programmi gira in background, cioè silenziosamente senza che noi ce ne accorgiamo. Si possono vedere nella prima colonna il nome dell'utente che ha lanciato il processo, poi segue il PID del processo (process ID) cioè quel numero che lo identifica in modo univoco , poi abbiamo il PPID (parent process ID) cioè il PID che l'ha generato, poi seguono altre informazioni come lo start time (STIME) del processo, su quale terminale è in esecuzione (TTY), il tempo di esecuzione (TIME), ed il nome del comando (CMD). Alla linea dove appare il comando "tput cup 0 0" vediamo che il parent ID di 1815 (prima colonna) è proprio il PID 30747 (seconda colonna) dove è stato lanciato lo script prova.sh. La sottoshell con il PID 1815 (prima colonna) sta attualmente eseguendo la linea dello script tput cup 0 0. Se in questo momento chiudiamo lo script prova.sh, chiuderemo implicitamente anche la sottoshell generata da questo script, cioè "tput cup 0 0". Questo vuol dire che se stoppiamo il processo padre, a cadere stopperemo tutti i processi figli.

Definizione5: Le variabili definite all'interno di una shell sono locali all'interno della shell stessa. Le variabili d'ambiente nella shell BASH devono essere esportate, a differenza dalla altre shell. Questo significa che per ogni sottoshell viene creata una copia di una variabile d'ambiente. Con il comando export si richiede al sistema di definire una copia della variabile per tutte le sottoshell generate. Si può pensare che le variabili esportino il proprio valore in una shell. Per chi conosce le strutture di programmazione, il meccanismo può essere considerato una forma di chiamata per valore.

Per esempio:

```
[root@diamondhead /home]# PILLO="ciao";
```

non la esportiamo subito, ma lanciamo:

```
[root@diamondhead /home]# echo $PILLO
ciao
```

come si vede il valore della variabile viene stampato, ma se cerchiamo di visualizzare questo valore in uno script, generando quindi una sottoshell, il valore non verrà visualizzato:

```
[root@diamondhead /home]# vi prova.sh
#!/bin/bash

echo $PILLO

[root@diamondhead /home]# chmod 777 prova.sh
[root@diamondhead /home]# ./prova.sh
```

il 777 dopo il comando chmod è un'altra forma per dare i permessi di esecuzione (anche di questo ne parleremo più avanti). Se vogliamo che questo valore venga visto anche dalle sottoshell dobbiamo esportare la variabile pippo:

```
[root@diamondhead /home]# export PILLO
[root@diamondhead /home]# ./prova.sh
ciao
```

Nella shell utente abbiamo diverse variabili d'ambiante già preimpostate. Difatti se lanciamo il comando:

```
[roberto@diamondhead roberto]$ set
```

avremo come risposta una serie di variabili d'ambiente preimpostate. Come si vede dall'output ci sono variabili scritte in maiuscolo ed in minuscolo. Quelle in maiuscolo sono dette variabili speciali mentre quelle scritte in minuscolo sono dette variabili locali. Per esempio la variabile speciale HOME definisce il percorso della directory home dell'utente, mentre la variabile locale noclobber, come già abbiamo visto, impedisce la cancellazione accidentale con un'operazione di redirezione. Alcune delle variabili speciali sono liberamente modificabili, come ad esempio la variabile PATH, altre invece è meglio non farlo, come la variabile HOME.
Per ottenere un elenco delle variabili speciali attualmente definite si può usare il comando env:

```
[roberto@geelong roberto]$ env
```

env si comporta come il comando set ma elenca solo le variabili speciali. Per definire automaticamente tutte le variabili speciali esiste il file .bash_profile posto nella directory dell'utente:

```
[roberto@diamondhead roberto]$ more .bash_profile
```

non appena si esegue il login nel sistema, tramite shell, la prima cosa che il sistema fa è quella di leggere il contenuto di questo file per il nostro utente. Nella shell BASH, abbiamo detto, per trasformare una variabile normale in una d'ambiente dobbiamo esportarla con il comando export. Come si può notare, al termine del file .bash_profile vengono esportate le variabili definite nello script, cioè le variabili BASH_ENV e PATH.
Proviamo ora a visualizzare con il comando echo alcune di queste variabili:

```
[roberto@diamondhead roberto]$ echo $HOME
/home/roberto
```

viene visualizzato il percorso della home del nostro utente.
Proviamo ora con:

```
[roberto@diamondhead roberto]$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/roberto/bin
```

abbiamo i vari percorsi relativi ai comandi di sistema. Proviamo adesso a creare una directory personale dove mettere tutti i nostri script eseguibili. Possiamo creare questa directory con il nome myscript. Al posto di specificare per esteso /home/roberto/myscript, si può utilizzare la variabile speciale HOME in questo modo:

```
[roberto@diamondhead roberto]$ vi .bash_profile
PATH=$PATH:$HOME/myscript
export PATH
```

usciamo dalla shell e rientriamo. (questo comporta l'uscita da VNC!, si potrebbe usare source .bash_profile).

Se facciamo un:

```
[roberto@diamondhead roberto]$ echo $PATH
```

vedremo effettivamente che nel PATH c'è la nostra directory myscript dove potremo mettere i nostri script personali. Per fare in modo che nel PATH ci sia anche la directory di lavoro attuale, per eseguire direttamente dove ci troviamo degli script o comunque

degli eseguibili, possiamo aggiungere alla variabile PATH un ":" in questo modo:

```
[roberto@diamondhead roberto]$ vi .bash_profile
PATH=$PATH:$HOME/myscript:
```

il ":" finale sta ad indicare che si deve prendere in considerazione, per la ricerca degli eseguibili, anche la directory attuale di lavoro. Proviamo a create un file script:

```
[roberto@diamondhead roberto]$ vi pippo.sh
#!/bin/bash

ls -la

[roberto@diamondhead roberto]$ chmod 777 pippo.sh
[roberto@diamondhead roberto]$ pippo.sh
[roberto@diamondhead roberto]$ source .bash_profile
```

Adesso proviamo a lanciare pippo.sh:

```
[roberto@diamondhead roberto]$ pippo.sh
```

noteremo che lo script pippo.sh verrà eseguito dalla directory corrente. Se vogliamo trasferire lo script pippo.sh all'interno della directory che abbiamo creato, cioè "myscript". Possiamo fare un:

```
[roberto@diamondhead roberto]$ mv pippo.sh myscript/
[roberto@diamondhead roberto]$ ls myscript/
```

Ora proviamo dalla nostra home directory a lanciare pippo.sh senza specificare il percorso dove si trova:

```
[roberto@diamondhead roberto]$ pippo.sh
```

Lo script verrà trovato grazie alla variabile PATH e dunque lanciato. Adesso proviamo a visualizzare il contenuto di PS1 e PS2:

```
[roberto@diamondhead roberto]$ echo $PS1
[\u@\h \W]\$
[roberto@diamondhead roberto]$ echo $PS2
>
```

La variabile PS1 contiene il valore del prompt del sistema, mentre PS2 contiene il prompt secondario, cioè il simbolo che appare quando il comando viene completato su più righe.

Si possono mettere informazioni particolari come ora, data, nome utente, etc.

Definizione6: Il file .bash_profile è lo script di inizializzazione della shell e viene eseguito automaticamente al momento del login dell'utente. Mentre .bash_profile inizializza la shell dell'utente, c'è un'altro file all'interno della directory /etc che si chiama profile. (andiamo a vedere un attimo: more /etc/profile). In questo script vengono inizializzate le variabili di sistema ancora prima che lo script .bash_profile del relativo utente viene mandato in esecuzione.

Per verificare questo possiamo definire la stessa variabile con valori diversi nei due file /etc/profile e /$HOME/.bash_profile esportandola correttamente e vedere con un echo quale dei due valori viene stampato. Siete in grado di verificarlo da soli? (ricordare che per leggere le nuove variabili definite bisogna lanciare: source nome_file - es: source /etc/profile)

Definizione7: Lo script .bashrc è un file che si trova nella directory di ogni utente e che viene eseguito anch'esso ogni volta che si accede alla shell o ad una sottoshell. Se si utilizza la shell BASH come shell di login, il file .bashrc viene eseguito subito dopo il file .bash_profile durante la procedura di login. Se la shell di default fosse stata la TCSH il file eseguito sarebbe stato .tcshrc.

I job: background, annullamento e sospensione

Prima ho parlato di processi in background che girano silenziosamente senza che noi ce ne accorgiamo. Per spiegare come funziona lo stato di background creiamo un script che ci consente di calcolare un valore tramite un'espressione. Apriamo un nuovo file che chiameremo prova2.sh:

```
[roberto@diamondhead roberto]$ vi prova2.sh

#!/bin/bash

LIMIT=900000
for (( a=1; a<=LIMIT; a++ ))
do
  z=$(($z+3-$a))
done
echo $z > result.txt
```

diamo i permessi di esecuzione e mandiamo in esecuzione lo script:

```
[roberto@diamondhead roberto]$ chmod +x prova2.sh
[roberto@diamondhead roberto]$ ./prova2.sh
```

notiamo che il nostro terminale è attualmente occupato dall'esecuzione di questo script. Se il valore di LIMIT fosse ancora più grande il tempo che dovremo aspettare per avere il terminale libero sarebbe ancora più elevato. Per ovviare a questo potremo aprire un'altro terminale, oppure mettere il nostro script shell (il nostro processo) in background in questo modo:

```
[roberto@diamondhead roberto]$ ./prova2.sh &
[1] 1912
```

il sistema ci da una serie di informazioni che indicano il numero del job (tra parentesi quadre) e il PID del processo. Difatti se noi digitiamo:

```
[roberto@diamondhead roberto]$ ps -ef | grep prova2.sh
roberto   1916  1333 99 10:59 pts/0    00:00:08 /bin/bash ./prova2.sh
```

vedremo che appunto, il PID del processo corrisponde con quello che il sistema ci ha dato quando abbiamo lanciato lo script.
Se ora vogliamo vedere quali sono i nostri jobs in esecuzione possiamo digitare:

```
[roberto@diamondhead roberto]$ jobs
[1]+  Running                 ./prova2.sh &
```

Il numero del job è 1, è in esecuzione, e il suo nome è prova2.sh.
A questo punto possiamo lavorare con altre cose intanto che il nostro script è in esecuzione. Oppure possiamo ucciderlo, perchè magari non ci interessa più.
In questo modo:

```
[roberto@diamondhead roberto]$ ./prova2.sh &
[2] 1940
[1]   Done                    ./prova2.sh
[roberto@diamondhead roberto]$ ps -ef | grep prova2.sh
roberto   1940  1333 99 11:07 pts/0    00:00:03 /bin/bash ./prova2.sh
[roberto@diamondhead roberto]$ kill -9 1940
[roberto@diamondhead roberto]$ ps -ef | grep prova2.sh
[2]+  Killed                  ./prova2.sh
```

il comando kill -9 uccide un processo immediatamente, si potrebbe anche aver terminato il processo facendo riferimento ai jobs in esecuzione:

```
[roberto@diamondhead roberto]$ ./prova2.sh &
[1] 1947
[roberto@diamondhead roberto]$ jobs
[1]+  Running                 ./prova2.sh &
[roberto@diamondhead roberto]$ jobs
[1]+  Running                 ./prova2.sh &
[roberto@diamondhead roberto]$ kill %1
[roberto@diamondhead roberto]$ jobs
[1]+  Terminated              ./prova2.sh
```

Un piccolo programma per il backup automatico

Proviamo adesso ad utilizzare i comandi utili per la compattazione e la compressione di file e di intere directory. Il comando tar ci consente di compattare un'intera directory in un singolo file. Per esempio, creiamo una directory:

```
[roberto@diamondhead roberto]$ mkdir mio_archivio
```

e creiamo all'interno di questa dei file:

```
[roberto@diamondhead roberto]$ touch mio_archivio/edit1.txt
[roberto@diamondhead roberto]$ touch mio_archivio/edit2.txt
[roberto@diamondhead roberto]$ touch mio_archivio/edit3.txt
```

ora con il comando tar compattiamo la directory mio_archivio in modo da avere un singolo file:

```
[roberto@diamondhead roberto]$ tar cvf mio_archivio.tar mio_archivio
```

ora abbiamo compattato in un solo file il contenuto della directory mio_archivio, ma non l'abbiamo ancora compresso. Per fare questa operazione diamo questo comando:

```
[roberto@diamondhead roberto]$ gzip mio_archivio.tar
```

il file ottenuto con estensione .gz è praticamente il risultato di queste operazioni: abbiamo ottenuto un archivio compresso.
Per ritornare alla directory originale si deve unzippare il file compresso mio_archivio.tar.gz in questo modo:

```
[roberto@diamondhead roberto]$ gzip -dvr mio_archivio.tar.gz
```

ed espandere il file così ottenuto:

```
[roberto@diamondhead roberto]$ tar xvf mio_archivio.tar
```

Un'altro comando molto utile (che abbiamo già visto) è l'mv che sta per move. Possiamo cambiare nome ad un file o ad una directory o trasferirla in un'altra zona del file system:

```
[roberto@diamondhead roberto]$ mv mio_archivio mio_archivio_new
```

altro comando utile è il cp, che copia un file o una directory:

```
[roberto@diamondhead roberto]$ touch pippo
[roberto@diamondhead roberto]$ cp pippo pippo_bis
```

TEST1: Provate ora a creare uno script shell in grado di prendere una directory prestabilita (facciamo mio_archivio), dove supponiamo che in essa sia contenuto tutto il nostro lavoro, e di eseguire un backup di quest'ultima nella stessa directory dove ci troviamo. Sareste in grado di farlo?

Risposta -> Lo script avrà una serie di istruzioni come queste:

```
#!/bin/bash

echo "Inizio archiviazione del file : mio_archivio"
echo "Nella directory               : /home/roberto/"

# archiviazione e compressione file
tar cvf /home/roberto/mio_archivio.tar /home/roberto/mio_archivio
gzip /home/roberto/mio_archivio.tar
```

Questo script shell esegue un tar di mio_archivio, dopodichè lo zippa nella directory prestabilita.
Ho preparato uno script che faremo assieme.
Proviamo ora a modificare lo script appena fatto in modo che le directory siano contenute in delle variabili, e le operazioni devono essere fatte su di esse. Stabiliamo anche che i vecchi backup non devono essere cancellati, ma numerati per esempio dalla data odierna. Inoltre, se durante queste operazioni si manifestano degli errori, redirezionare lo standard error in un file che chiameremo

error.txt ed un eventuale standard output (tipico dei comandi come gzip e tar) nel device null:

```
#!/bin/bash

# path assoluto dove si trova l'archivio
DIR=/home/roberto/

# nome dell'archivio
ARCHIVIO=mio_archivio

# path + nome archivio
OF=$DIR$ARCHIVIO

# nome del file di errori
ERROR=error.txt

# definizione della data corrente
APPEND=$(date +%Y%m%d%H%M%S)

echo "Inizio archiviazione del file :"$ARCHIVIO
echo "Nella directory               :"$DIR

# archiviazione e compressione file
tar cvf $OF-$APPEND.tar $OF 1> /dev/null 2> $DIR$ERROR
gzip $OF-$APPEND.tar 1> /dev/null 2>> $DIR$ERROR
```

TEST2: proviamo adesso a modificare lo script in modo da copiare l'archivio ottenuto in una directory prestabilita, che chiameremo backup, e gli errori eventuali sulla directory error:

```
#!/bin/bash

# path assoluto dove si trova l'archivio
DIR=/home/roberto/

# nome dell'archivio
ARCHIVIO=mio_archivio

# path + nome archivio
OF=$DIR$ARCHIVIO

# nome della directory degli errori
ERROR_DIR=error/

# nome del file di errori
ERROR=error.txt

# nome della directory di backup
BACKUP=backup/

# definizione della data corrente
```

```
APPEND=$(date +%Y%m%d%H%M%S)

echo "Inizio archiviazione del file :"$ARCHIVIO
echo "Nella directory               :"$DIR$BACKUP

# archiviazione e compressione file
tar cvf $DIR$BACKUP$ARCHIVIO-$APPEND.tar $OF 1> /dev/null 2>
$DIR$ERROR_DIR$ERROR
gzip $DIR$BACKUP$ARCHIVIO-$APPEND.tar 1> /dev/null 2>> $DIR$ERROR_DIR$ERROR
```

Il nostro programma di backup non è abbastanza solido. Se per esempio la directory backup viene cancellata per errore il programma si ferma.

(provare a mandarlo in esecuzione senza la directory di backup e vedere gli errori riportati nel file error/error.txt).

Per renderlo più robusto dovremo controllare se esistono le directory dell'archivio mio_archivio, backup ed error e crearle in caso contrario:

```
#!/bin/bash

# path assoluto dove si trova l'archivio
DIR=/home/roberto/

# nome dell'archivio
ARCHIVIO=mio_archivio

# path + nome archivio
OF=$DIR$ARCHIVIO

# nome della directory degli errori
ERROR_DIR=error/

# nome del file di errori
ERROR=error.txt

# nome della directory di backup
BACKUP=backup/

# definizione della data corrente
APPEND=$(date +%Y%m%d%H%M%S)

# controllo se esiste le directory mio_archivio
# se non esistone procedi con la creazione
if [ -x $OF ]
then
        echo "La directory $OF esiste, procedo con la compressione"
else
        echo "La directory $OF non esiste!"
        exit
fi
```

```
echo "------------------------------------------------"

# controllo se esiste le directory error e backup
# se non esistono procedi con la creazione
if [ -x $DIR$ERROR_DIR ]
then
        echo "La directory $ERROR_DIR esiste, procedo con la scrittura di eventu
ali errori"
else
        mkdir $DIR$ERROR_DIR
        echo "Directory degli errori creata"
fi

if [ -x $DIR$BACKUP ]
then
        echo "La directory $BACKUP esiste, procedo con la scrittura dell'archivi
o"
else
        mkdir $DIR$BACKUP
        echo "Directory dei backup creata"
fi

echo "Inizio archiviazione del file :"$ARCHIVIO
echo "nella directory               :"$DIR$BACKUP

# archiviazione e compressione file
tar cvf $DIR$BACKUP$ARCHIVIO-$APPEND.tar $OF 1> /dev/null 2> $DIR$ERROR_DIR$ERRO
R
gzip $DIR$BACKUP$ARCHIVIO-$APPEND.tar 1> /dev/null 2>> $DIR$ERROR_DIR$ERROR
```

ora il programma è abbastanza robusto e può reggere bene da situazioni impreviste, come la mancanza delle directory di riferimento. Possiamo utilizzare adesso il programma crontab per schedulare il nostro script ad intervalli regolari in modo da creare un backup della nostra cartella di lavoro, diciamo ogni ora. Prima di tutto dobbiamo preoccuparci che il demone crond sia in esecuzione:

```
[roberto@diamondhead roberto]$ ps -ef | grep crond
root      1008     1  0 08:28 ?        00:00:00 crond
roberto   1731  1656  0 10:39 pts/2    00:00:00 grep crond
```

poi modifichiamo (se non esiste lo creiamo) il file cron.allow dentro /etc come amministratore di sistema, e ci aggiungiamo gli utenti abilitati al suo utilizzo:

```
roberto
root
```

Effettuiamo adesso l'editazione del crontab del nostro utente:

```
[roberto@diamondhead roberto]$ crontab -e
```

e aggiungiamo la riga:

```
0 * * * * /home/roberto/script.sh
```

il primo campo definisce i minuti (0-59), il secondo specifica l'ora (0-23), il terzo al giorno del mese (1-31), il quarto è il mese (1-12), il quinto campo è il giorno della settimana (0-6) dove 0 corrisponde alla domenica.
Se per esempio volessimo eseguire lo script alle 2 di ogni notte dal lunedi al venerdi dovremo impostare una riga del tipo:

```
0 2 * * 1-5 /home/roberto/script.sh
```

Per vedere quali jobs sono in elenco nel crontab basta digitare:

```
[roberto@diamondhead roberto]$ crontab -l
```

se si è root e si vuole editare il crontab di un utente basta specificare quest'ultimo preceduto dall'opzione -u:

```
[root@diamondhead /root]# crontab -u roberto -e
```

Tutti i file che contengono la lista di cose da fare di crontab vengono archiviati nella directory:

```
[root@diamondhead /root]# cd /var/spool/cron/
```

dove ogni file ha il nome dell'utente che lo ha creato. Se si vuole editare il file di crontab si deve utilizzare esclusivamente crontab -e; non si deve editare direttamente il file.
Se vogliamo cancellare il nostro file crontab dobbiamo utilizzare:

```
[roberto@diamondhead roberto]$ crontab -r
```

Un'altra funzione interessante della shell bash è quella che permettere di specificare a linea di comando dei parametri. Se per esempio vogliamo stampare un qualcosa specificato subito dopo il nome dello script, possiamo fare:

```
#!/bin/bash
if [ -z "$1" ]; then
        echo "uso: $0 file"
```

```
        exit
fi
echo "Nome del File:$1"
```

il flag -z indica se la variabile è vuota, ed in questo caso ne mostra il funzionamento (notare che viene stampato il nome dello script grazie alla variabile $0 che è il primo parametro). Alla fine viene stampato il nome specificato subito dopo lo script, se questo esiste.

TEST3: In base a quello che avete appreso fino a questo punto, provate a realizzare un sistema di backup simile a quello che abbiamo creato, ma deve creare copie di backup con directory specificate a linea di comando. Il programma registra sempre gli errori su error/error.txt e il backup sulla directory backup/. Prestare particolare attenzione al percorso assoluto che in questo caso diventa relativo visto che si specifica il file direttamente.

Esempio già pronto:

```
#!/bin/bash

if [ -z "$1" ]; then
        echo "uso: $0 directory di backup"
        exit
fi

# nome della directory degli errori
ERROR_DIR=error/

# nome del file di errori
ERROR=error.txt

# nome della directory di backup
BACKUP=backup/

# definizione della data corrente
APPEND=$(date +%Y%m%d%H%M%S)

# controllo se esiste le directory $1
# se non esistone procedi con la creazione
if [ -x $1 ]
then
        echo "La directory esiste, procedo con la compressione"
else
        echo "La directory $1 non esiste!"
        exit
fi
echo "------------------------------------------------"

# controllo se esiste le directory error e backup
# se non esistono procedi con la creazione
```

```
if [ -x $ERROR_DIR ]
then
        echo "La directory $ERROR_DIR esiste, procedo con la scrittura di eventu
ali errori"
else
        mkdir $ERROR_DIR
        echo "Directory degli errori creata"
fi

if [ -x $BACKUP ]
then
        echo "La directory $BACKUP esiste, procedo con la scrittura dell'archivi
o"
else
        mkdir $BACKUP
        echo "Directory dei backup creata"
fi

echo "Inizio archiviazione del file :"$1
echo "nella directory               :"$BACKUP

# archiviazione e compressione file
tar cvf $BACKUP$1-$APPEND.tar $1 1> /dev/null 2> $ERROR_DIR$ERROR
gzip $BACKUP$1-$APPEND.tar 1> /dev/null 2>> $ERROR_DIR$ERROR
```

Permessi di accesso ai file e alle directory, utenti e gruppi

Digitando nella propria home directory il comando:

```
[roberto@diamondhead roberto]$ ls -l
totale 48
drwxr-xr-x    2 roberto  roberto      4096 mag  6 15:26 backup
drwxr-xr-x    3 roberto  roberto      4096 mag  2 17:26 Desktop
drwxr-xr-x    2 roberto  roberto      4096 mag  6 15:25 error
-rw-------    1 roberto  roberto       409 mag  2 17:41 mbox
drwxrwxr-x    2 roberto  roberto      4096 mag  5 19:40 mio_archivio
drwx------    2 roberto  roberto      4096 mag  2 15:34 nsmail
-rwxrwxr-x    1 roberto  roberto       662 mag  5 20:07 script1a.sh
....
```

oltre ai nomi di file e directory si vedranno tutta una serie di permessi nella prima colonna, il nome del proprietario nella terza (roberto), ed il gruppo nella quarta (roberto). Seguono poi altre informazioni che riguardano la data di creazione del file.
Valutiamo la prima riga. Abbiamo una serie di valori `drwxr-xr-x` disposti su 4 colonne: la prima è formata da una lettera, in questo caso la "d" sta a significare che si tratta di una directory. Segue poi una colonna composta da tre elementi rwx. Questi sono i permessi del proprietario della directory che in questo caso sono di lettura

(read), scrittura (write) e di accesso (execute). Nella colonna che segue abbiamo altre tre lettere che indicano i permessi del gruppo, che in questo caso può solo leggere e accedere alla directory. L'ultima colonna, anche questa composta da tre lettere, riguardano i permessi per gli altri utenti del sistema.
Allora ricapitolando: la prima colonna, composta da 1 carattere, indica se il file è una directory (allora avremo una "d"), oppure un file normale (in questo caso avremo un "-"). Poi l'altra colonna è composta da 3 caratteri, che identifica i permessi del proprietario e può essere di lettura (r), scrittura (w) o di esecuzione (x) per i file, o di accesso per le directory. L'altra colonna, sempre composta da tre caratteri, sono i permessi del gruppo, e l'ultima colonna si riferisce ai permessi di tutti gli altri utenti.

Ho parlato di utenti di sistema e gruppi. Gli utenti di sistema generalmente hanno accesso al sistema e questi si possono organizzare in gruppi. Per esempio creiamo un nuovo utente da root:

```
[root@diamondhead /root]# adduser marco
```

e stabiliamo la password:

```
[root@diamondhead /root]# passwd marco
Changing password for user marco
New UNIX password:pippo
Retype new UNIX password:pippo
passwd: all authentication tokens updated successfully
```

ora se andiamo a controllare il contenuto del file /etc/passwd vedremo che il nuovo utente è nell'elenco assieme al nostro:

```
[root@diamondhead /root]# more /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
...
squid:x:23:23::/var/spool/squid:/dev/null
roberto:x:500:500::/home/roberto:/bin/bash
paola:x:501:501::/home/paola:/bin/bash
marco:x:502:502::/home/marco:/bin/bash
```

in questo file il primo campo identifica l'utente (massimo 8 caratteri), il secondo identifica la password (in questo sistema linux è installata l'utility shadow che permette di nascondere la password

in un'altro file dentro /etc/shadow), poi segue l'userID che identifica univocalmente l'utente grazie ad un numero (non deve essere 0 perchè è associato all'utente root), poi abbiamo il groupID che ci identifica in un gruppo di appartenenza, poi ancora abbiamo la possibilità di creare per ogni utente informazioni personali, come il vero nome dell'utente, il numero di telefono e così via (in questo caso non abbiamo nessuna informazione per i nostri utenti), poi abbiamo la directory home dell'utente e per finire la shell utilizzata per l'utente, che come abbiamo visto possono essere scelte tra quelle disponibili dal nostro sistema e si possono consultare dentro la directory /etc/shells.
Ora, se vogliamo far appartenere l'utente marco ad un gruppo, magari assieme ad altri utenti, visto che magari marco fa parte del team di lavoro di un progetto, possiamo creare questo gruppo (ovviamente da utente root) con il comando:

```
[root@diamondhead /root]# groupadd works
```

adesso facendo un:

```
[root@diamondhead /root]# more /etc/group
```

verranno visualizzati tutti i gruppi del sistema compreso quello appena creato: il gruppo works con GID 504.
Ora se vogliamo far parte l'utente marco del gruppo works non dovremo fare altro che editare il file:

```
[root@diamondhead /root]# vi /etc/group
```

e aggiungere subito dopo al ":" del group ID gli utenti relativi:

```
works:x:504:marco,roberto  (non mettere support lo faranno loro....)
```

adesso salviamo e usciamo dal terminale. Facciamo nuovamente il login però con l'utente marco e vedremo che digitando:

```
[marco@diamondhead marco]$ id
uid=502(marco) gid=502(marco) gruppi=502(marco),504(works)
```

l'UID di marco è 502, fa parte dello stesso gruppo del suo UID, ma fa anche parte del gruppo works con GID 504.

TEST4: Provate ora a far appartenere il vostro utente support al gruppo works.

_ · _ · _ · _

(si presuppone che gli allievi siano riusciti a far appartenere l'utente support al gruppo works)

Loggatevi tutti come support e andate nella vostra home directory. Digitare:

```
[roberto@diamondhead roberto]$ ls -l
```

qui abbiamo i vostri file con i relativi permessi. Troviamo in particolare il file prova2.sh che ha questi permessi:

```
-rwxrwxrwx    1 roberto  roberto       101 mag  8 10:59 prova2.sh
```

in questo caso il file script prova2.sh è effettivamente un file, notale il trattino sulla prima colonna, poi ha i permessi di lettura-scrittura-esecuzione per il proprietario, i permessi di lettura-scrittura-esecuzione per il gruppo, che in questo caso è sempre lo stesso del proprietario, e i permessi di lettura-scrittura-esecuzione per tutti gli altri utenti del sistema.
Adesso vogliamo fare in modo che solo il nostro utente possa mandare in esecuzione lo script. Come facciamo?
Scriviamo:

```
[roberto@diamondhead roberto]$ chmod 766 prova2.sh
```

diamo un:

```
[roberto@diamondhead roberto]$ ls -la
```

e vedremo che la lettera x di esecuzione c'è solo nel proprietario del file:

```
...
-rwxrw-rw-    1 roberto  roberto       101 mag  8 10:59 prova2.sh
...
```

Che cosa vuol dire quel 766 subito dopo chmod? Chi è che lo sa?

Allora, considerate che il numero ottale 1 sta per esecuzione, il numero ottale 2 sta per scrittura, ed il numero ottale 4 sta per lettura. Se combiniamo questi numeri tra di loro possiamo avere tutte le combinazioni per i permessi relativi al proprietario, al gruppo e a tutti gli altri.
Per esempio se vogliamo negare anche la possibilità di scrivere il file prova2.sh per tutti gli altri utenti e dare solo a questi i permessi di lettura dovremo scrivere:

```
[roberto@diamondhead roberto]$ chmod 764 prova2.sh
```

noteremo con un ls -l che il file prova2.sh ora ha solo i permessi di lettura per tutti gli altri utenti (vedere le ultime 3 lettere).

Consideriamo il primo numero di chmod 764 e cioè 7. Per il proprietario abbiamo i permessi di lettura = 4 + i permessi di scrittura = 2 + i permessi di esecuzione = 1 che in totale fa appunto 7. Per il secondo numero 6 (riferito ai permessi del gruppo) abbiamo i permessi di lettura = 4 + i permessi di scrittura = 2 che in totale fa appunto 6. Continuiamo poi con il numero 4 relativi ai permessi di tutti gli altri utenti: e si riferisce al solo permesso di lettura. Per quel che riguarda le directory, visto che non sono eseguibili, la x di execute sta ad indicare l'accesso a essa.
Quindi se vogliamo abilitare o disabilitare ad un utente o ad altri l'accesso ad una directory, non dovremo fare altro che mettere o togliere la x, dai permessi di quella directory.
C'è un'altro sistema per attribuire i permessi ai file, cioè quello con le lettere.
Se digitiamo:

```
[roberto@diamondhead roberto]$ chmod g-w prova2.sh
```

toglieremo al gruppo i permessi di scrittura del file prova2.sh. Provate a fare un ls -l per controllare che effettivamente questo è avvenuto. Per aggiungere nuovamente i permessi di scrittura al gruppo scriveremo:

```
[roberto@diamondhead roberto]$ chmod g+w prova2.sh
```

quindi quel - e quel + sta per togliere e aggiungere rispettivamente i permessi specificati subito dopo. Se si volesse modificare i

permessi all'utente, al posto della g si utilizzerebbe la lettera u mentre per gli altri utenti la lettera è la o (la o sta per other). Possiamo anche dare i permessi di scrittura sia per il proprietario, per il gruppo e per gli altri utenti con un comando solo:

```
[roberto@diamondhead roberto]$ chmod a+x prova2.sh
```

la lettera a sta ad indicare tutti (il proprietario, il gruppo e gli altri). Possiamo anche combinare le cose facendo un:

```
[roberto@diamondhead roberto]$ chmod ug-x prova2.sh
```

in questo caso abbiamo tolto il permesso di esecuzione al proprietario e al gruppo lasciando invariati i permessi per gli altri utenti.

TEST5: Provate ora a creare 3 file vuoti + una directory.
Il primo file deve avere i permessi di lettura, scrittura ed esecuzione per il proprietario, e nessun altro permesso per il gruppo e gli altri.
Il secondo file deve avere i permessi di scrittura per il proprietario e lettura per il gruppo. Nessun permesso per gli altri.
Il terzo file deve avere tutti i permessi di lettura scrittura ed esecuzione solo per gli altri utenti.
La directory dovrà avere i permessi di lettura, scrittura ed accesso al proprietario, lettura ed accesso per il gruppo e solo accesso per gli altri utenti.

Gestione del server: standalone e xinetd

Parliamo adesso di programmi server, cioè i demoni che girano nel nostro sistema.
Definizione8: Affinché gli utenti possano accedere ad un servizio come quello web nel nostro sistema, il programma che gestisce queste richieste (il demone httpd di apache) deve essere attivo e funzionate. Se nel nostro sistema è presente un sito web, questo sarà visibile alla rete internet (o alla intranet) solo se esiste appunto questo server web. Lo stesso vale per gli altri servizi come quello della telnet, dell'ftp, dell'nfs etc. Per vedere i servizi attualmente in esecuzione nel nostro sistema digitiamo:

```
[roberto@diamondhead roberto]$ ps -aux
```

Il comando ps, come abbiamo già visto mostra un'istantanea dei processi correnti. l'opzione a (subito dopo il trattino) sta ad indicare che vogliamo visualizzare anche i processi degli altri utenti, l'opzione u visualizza il nome dell'utente e l'ora di start del processo. La x finale mostra i processi senza il terminale di controllo. Se volete conoscere le altre opzione basta fare un:

```
[roberto@diamondhead roberto]$ man ps
```

TEST6: se vogliamo visualizzare in tempo reale lo stato e le varie caratteristiche dei processi attualmente in esecuzione, come possiamo fare? Considerate che basterebbe avviare un ciclo infinito con all'interno il comando ps. Provate a farlo.

Si possono visualizzare i processi attualmente in esecuzione anche con il comando top. A differenza del comando ps, top aggiorna costantemente lo stato dei processi in modo da monitorarli in tempo reale. Digitiamo top:

```
[roberto@diamondhead roberto]$ top
```

Vediamo un momento i server attivi nella nostra linux box:

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.1   1368   544 ?        S    10:27   0:04 init [3]
root         2  0.0  0.0      0     0 ?        SW   10:27   0:00 [keventd]
root         3  0.0  0.0      0     0 ?        SW   10:27   0:00 [kapm-idled]
root       969  0.0  0.5   5344  2004 ?        S    10:28   0:00 sendmail:
accepting connections
....
```

(spiegare brevemente i vari servizi attivi come l'httpd, il sendmail, etc.)

Ci sono due modi (anzi tre) per avviare un programma server come il server web apache o sendmail. Il primo è quello di avviare il demone direttamente a linea di comando da superutente, per esempio:

```
[root@diamondhead init.d]# su -
```

```
[root@diamondhead /root]# /etc/init.d/httpd start
```

per fermarlo scriveremo:

```
[root@diamondhead /root]# /etc/init.d/httpd stop
```

per stopparlo e riavviarlo in una sola volta:

```
[root@diamondhead /root]# /etc/init.d/httpd restart
```

se scriviamo:

```
[root@diamondhead init.d]# ps -ef | grep httpd
```

vedremo che il server httpd è partito. Proviamo a lanciare netscape da linea di comando con:

```
[root@diamondhead init.d]# netscape &
```

e nell'url digitiamo l'indirizzo di loopback 127.0.0.1 che identifica la nostra macchina (è possibile anche digitare localhost).
Vedremo la pagina di index.html di apache che ci conferma che il web server è attivo e pronto per ricevere richieste di connessione.
Se ora proviamo a stoppare apache con il comando:

```
[root@diamondhead init.d]# /etc/init.d/httpd stop
```

prima di tutto non vedremo più il demone che gira. Proviamo a fare:

```
[root@diamondhead init.d]# ps -ef | grep httpd
```

difatti non troveremo l'httpd che gira. Se andiamo nel browser netscape e ricarichiamo la pagina vedremo che effettivamente la richiesta della prima pagina index.html che non verrà soddisfatta.

Abbiamo visto il primo modo per far partire o fermare un servizio. Il secondo modo consiste nel modificare gli script di avvio automatici in modo da attivare il servizio all'accensione della macchina. Se visualizziamo il contenuto del file:

```
[root@diamondhead xinetd.d]# more /etc/inittab
```

e andiamo alla linea dove c'è scritto "Default runlevel". Vediamo una serie di numeri che vanno da 0 a 6. Ognuno di questi numeri identifica il runlevel di avvio della macchina. Il runlevel 0 porta la nostra linux box allo spegnimento (halt). C'è anche scritto di non utilizzare questo numero per l'initdefaults, altrimenti la macchina non partirebbe mai.
Il numero 1 porta la macchina nella modalita single user, disabilitando la multiutenza e i servizi di rete (questo runlevel è usato generalmente per l'amministrazione del sistema escludendo interferenze dagli altri utenti).
Il numero 2 porta la macchina in uno stato di multiutenza senza la rete, quindi anche senza il servizio NFS.
Il numero 3 porta la macchina alla piena operatività: multiutenza, servizi di rete, etc. Questa è l'opzione di default, infatti se si vede un pò più giù si vedrà che l'initdefault è settato al runlevel 3.
Il numero 4 non è utilizzato.
Poi abbiamo il numero 5 che è la stessa identica cosa del 3 solo che il sistema si porta in modalità grafica: quindi piena operatività ma con l'interfaccia grafica. Bisogna fare attenzione che la modalità grafica sia settata correttamente altrimenti il sistema quando tenta di passare in modalità grafica si blocca.
Per finire abbiamo il numero 6 che riavvia il sistema e si porta nella modalità specificata in initdafault.
Se vogliamo fare come S. Tommaso proviamo a verificare quanto detto. Proviamo a riavviare la nostra macchina. Prima di specificare il runlevel dobbiamo usare il comando init o telinit:

```
[root@diamondhead xinetd.d]# telinit 6
```

Il sistema si riavvierà portandoci nel runlevel specificato da initdefault.

(aspettare che il sistema si porta nello stato operativo)

Proviamo ora a portare la nostra macchina al runlevel 1.

```
[root@diamondhead xinetd.d]# telinit 1
```

Definizione9: Il runlevel 1 viene usato spesso per ripristinare le inconsistenze create nelle varie partizioni causate da continui spegnimenti brutali. (arresti macchina senza la normale procedura di spegnimento. Faremo anche questo: simuleremo uno stato simile

portandoci ad eseguire manualmente i controlli necessari per il ripristino del file system danneggiato).

Proviamo ora a portare la nostra macchina al runlevel 5 (piena funzionalità ma in modalità grafica):

```
[root@diamondhead xinetd.d]# telinit 5
```

Portiamo adesso il sistema nel runlevel 3:

```
[root@diamondhead xinetd.d]# telinit 3
```

Ok, visto questo, eravamo rimasti al modo automatico di avvio dei servizi, come l'httpd.
Se vogliamo che questo servizio parta in automatico, quando il sistema si avvia al runlevel 3, dobbiamo entrare su:

```
[root@diamondhead xinetd.d]# cd /etc/rc.d/rc3.d/
```

diamo in ls per visualizzare i file, e troviamo tutti i demoni che girano silenziosamente in background al runlevel 3, ossia quello nel quale attualmente ci troviamo. Questi file che vediamo non sono altro che link simbolici agli script contenuti nella directory **/etc/rc.d/init.d/**. Se infatti scriviamo:

```
[root@diamondhead rc3.d]# ls -l
```

vediamo che effettivamente sono file che si riferiscono ad altri dentro **init.d**.
Proviamo ad entrarci un attimino:

```
[root@diamondhead rc3.d]# cd /etc/rc.d/init.d/
```

visualizziamo il contenuto della directory con un ls. Questi file sono gli script di avvio e di chiusura dei demoni installati nel server. Ogni volta che installiamo un processo o demone, lo script di avvio e di chiusura viene creato qui dentro. Proviamo a visualizzare lo script di gestione dei apache:

```
[root@diamondhead init.d]# more httpd
```

Vediamo il codice scritto nella shell bash che gestisce questo servizio. Ora torniamo nella directory rc3.d:

```
[root@diamondhead rc3.d]# cd /etc/rc.d/rc3.d/
```

diamo il comando:

```
[root@diamondhead rc3.d]# ls S*
```

vediamo tutti i processi che partono a questo runlevel (S sta per start) e la loro sequenza di avvio (il numero indica la sequenza). Inversamente, se facciamo un:

```
[root@diamondhead rc3.d]# ls K*
```

vediamo tutti i processi che si fermano quando usciamo dal runlevel 3. Anche in questo caso il numero identifica la sequenza di chiusura del runlevel.
Diamo un'occhiata al demone httpd, ossia al web server apache, digitando il comando:

```
[root@diamondhead rc3.d]# ls *httpd
```

vediamo che è collocato come **S85httpd** nella sequenza di avvio, ma non c'è nessun riferimento per la chiusura - stranamente - (vabbè, si vede che non ne ha bisogno).
Per fare in modo che il web server apache non parta più al runlevel 3, non dobbiamo fare altro che cancellare il link **S85httpd**:

```
[root@diamondhead rc3.d]# rm S85httpd
```

In questo modo non cancelliamo lo script vero e proprio che si trova dentro **/etc/rc.d/init.d/**, ma solo il link di riferimento. Proviamo a riavviare il server al runlevel 3 sempre per fare il S. Tommaso della situazione: per vedere che effettivamente il server httpd non verrà caricato.

```
[root@diamondhead rc3.d]# telinit 3
```

Se un domani volessimo ripristinarlo possiamo riscrivere il link con:

```
[root@diamondhead rc3.d]# ln -s /etc/rc.d/init.d/httpd S85httpd
```

il comando ln scrive un link simbolico (con l'opzione s) che si chiama S85httpd nella directory corrente, del file httpd situato su

/etc/rc.d/init.d/. Un link praticamente è un riferimento virtuale ad un'altro file reale. Se si cancella un link il file reale rimane inviolato, mentre se si cancella il file reale un eventuale link a questo rimarrà irrisolto. Cerco di chiarire subito la relazione tra questi: editiamo con il VI un file:

```
[roberto@diamondhead roberto]$ vi pippo
ciao al mondo
```

ora abbiamo un file reale. Se vogliamo creare un link a questo faremo:

```
[roberto@diamondhead roberto]$ ln -s pippo pippoln
```

ora abbiamo un file reale che si chiama pippo e un link a questo che si chiama pippoln. Proviamo adesso a editare il link pippoln con l'editor VI:

```
[roberto@diamondhead roberto]$ vi pippoln
ciao al mondo infame
```

ora visualizziamo il file reale pippo:

```
[roberto@diamondhead roberto]$ more pippo
```

Le modifiche sono state apportate anche se abbiamo editato il link. Se cancelliamo il link pippoln, al file pippo non succede nulla, ma se cancelliamo il file pippo:

```
[roberto@diamondhead roberto]$ rm pippo
```

e vediamo il contenuto di pippoln con:

```
[roberto@diamondhead roberto]$ more pippoln
pippoln: File o directory inesistente
```

il sistema ci dà un messaggio d'errore. Ora se editiamo il link pippoln con il VI:

```
[roberto@diamondhead roberto]$ vi pippoln
sono ancora io
```

proviamo a fare:

```
[roberto@diamondhead roberto]$ ls
```

il file pippo è risorto e conterrà la frase appena scritta.

TEST7: Per ritornare al discorso di prima, cioè quello dei vari processi che partono a determinati runlevel, potremo far eseguire solo al momento dell'avvio del runlevel 3 lo script di backup che abbiamo già fatto. Non si tratta di un processo che deve partire, ma in ogni caso sarebbe uno script che verrebbe eseguito. In sostanza il backup verrà avviato solo quando il sistema viene acceso al runlevel 3. Cercate di dare un numero piuttosto alto, tipo S87 o S88. Dai, provate a farlo.

Esempio risolto:

```
[root@diamondhead roberto]# cp script2.sh /etc/rc.d/init.d/.

[root@diamondhead roberto]# cd /etc/rc.d/rc3.d/

[root@diamondhead rc3.d]# ln -s ../init.d/script2.sh S87backup
```

Siamo arrivati al terzo sistema per avviare un demone.

Definizione10: Utilizzando il comando chkconfig possiamo specificare il servizio che si vuole avviare e il runlevel in cui deve essere avviato.
Proviamo a far partire il web server apache al runlevel 3:

```
[root@diamondhead rc3.d]# chkconfig --level 3 httpd on
```

se andiamo a controllare dentro /etc/rc.d/rc3.d vedremo che il link è stato creato dal comando chkconfig:

```
[root@diamondhead rc3.d]# cd /etc/rc.d/rc3.d/
[root@diamondhead rc3.d]# ls

...
K16rarpd       K65identd   S06reconfig  S55sshd        S85httpd
...
```

per disabilitarlo scriviamo:

```
[root@diamondhead rc3.d]# chkconfig --level 3 httpd off
```

meglio abilitarlo:

```
[root@diamondhead rc3.d]# chkconfig --level 3 httpd on
```

per conoscere le informazioni di avvio di un servizio scriviamo:

```
[root@diamondhead rc3.d]# chkconfig --list httpd
httpd           0:off   1:off   2:off   3:on    4:off   5:off   6:off
```

vediamo che il web server apache è disattivato al runlevel 0, 1 e 2, attivato al 3, disattivato al 4, 5 e 6.
Un modo per vedere lo stato di tutti i servizi del nostro sistema è digitare:

```
[root@diamondhead rc3.d]# chkconfig --list
```

qui abbiamo un elenco di tutti gli stati dei servizi. Verso la fine dell'output vediamo una dicitura "servizi basati su xinetd". Questi sono servizi che vengono mandati in esecuzione SOLO quando c'è una richiesta da soddisfare. Non sono perennemente in ascolto, come gli altri, ma in uno stato di riposo. Sul mio sistema ho l'rsh su on, se provo a vederlo con un ps non lo troverò:

```
[root@diamondhead rc3.d]# ps -ef | grep rsh
```

C'è qualcuno di voi che ha un servizio xinetd su on? Provate a visualizzarlo come ho fatto io con rsh. Non lo troverete.
Questo perchè il servizio parte solo quando serve, quindi è fermo, e non appena serve parte per l'occasione per spegnersi subito dopo.
Un server come httpd che abbiamo visto è detto standalone, mentre un server che parte quando serve è detto "extended internet services daemon" o xinetd. In pratica, un server di questo tipo non ha tante richieste da soddisfare: qui troveremo il server telnet, l'ftp etc. Mentre se abbiamo un sito molto trafficato ci conviene fare in modo di tenere l'httpd sia perennemente in ascolto.

Per dimostrare questo abilitiamo il servizio telnet dentro la directory /etc/xinet.d:

```
[root@diamondhead /root]# vi /etc/xinetd.d/telnet
disable = no
```

qui ci sono tutti i servizi xinetd. Adesso per fare in modo di aggiornare l'xinetd dobbiamo riavviarlo:

```
[root@diamondhead /root]# /etc/rc.d/init.d/xinetd restart
```

Come si vede lo stesso xinetd è un servizio che resta in ascolto in background, e che gestisce però tutti i servizi xinetd che si trovano dentro la directory /etc/xinetd.d/. Riavviando il servizio xinetd aggiorneremo tutte le impostazioni dei server dentro la directory /etc/xinetd.d compreso il disable=no appena impostato del servizio telnetd.

proviamo a fare un:

```
[root@diamondhead rc3.d]# ps -ef | grep telnetd
```

non troveremo nulla, anche se il telnetd è stato abilitato. Questo perchè parte il demone telnetd parte solo su richiesta. Ora proviamo ad aprire un'altra console e a fare un:

```
[root@diamondhead /root]# telnet 192.168.1.1
```

ovviamente sostituite l'ip con quello del vostro computer. Autenticativi con la vostra username e password. Una volta autenticati proviamo dall'altra console a fare un:

```
root      1605  1522  0 11:27 ?        00:00:00 in.telnetd: geelong.r3d.it
```

vediamo appunto che il demone telnetd è partito e resterà attivo fino alla chiusura della sessione.

Il comando chkconfig può gestire anche i servizi xinetd in questo modo:

```
[root@diamondhead /root]# chkconfig telnet on
[root@diamondhead /root]# chkconfig --list telnet
telnet          on
```

ora proviamo a disabilitarlo:

```
[root@diamondhead /root]# chkconfig telnet off
[root@diamondhead /root]# chkconfig --list telnet
telnet          off
```

Gestione dei pacchetti RPM e installazione di un'applicazione in codice sorgente

Adesso daremo un'occhiata a come si installa un programma sotto linux. Inizieremo con i pacchetti RPM.
Definizione11: Un pacchetto RPM è un programma di installazione di un'applicazione software. Spesso un'applicazione Linux è costituita da numerosi file che devono essere installati in varie directory: il programma viene inserito nella directory /usr/bin, i file di documentazione in un'altra directory e i file di libreria in una terza directory. Inoltre l'installazione può dover modificare alcuni file di configurazione del sistema. Il pacchetto software RPM svolge automaticamente tutte queste operazioni. Se successivamente si deciderà di non volere una determinata applicazione, si potranno anche disinstallare i pacchetti per eliminare dal sistema tutti i file e le informazioni di configurazione (più o meno come accade su windows).

Proviamo a installare il pacchetto. Digitando:

```
[root@diamondhead /root]# rpm
```

si ha la lista delle opzioni per questo comando. Procuriamoci innanzitutto il pacchetto da installare, per esempio possiamo installare xlispstat, il linguaggio lisp con un modulo supplementare per le statistiche.
Intanto accertiamoci che xlispstat non sia già installato:

```
[root@diamondhead /root]# rpm -qi xlispstat
package xlispstat is not installed
```

fatto questo installiamo il pacchetto:

```
[root@diamondhead /root]# rpm -ivh xlispstat-3.52.18-2.i386.rpm
Preparing...                ########################################### [100%]
   1:xlispstat              ########################################### [100%]
```

il pacchetto è stato installato. Il parametro i sta per install, la v sta per verify (pronuncia: verifai), e la h sta per hash (pronuncia: hash), cioè fa in modo che si veda lo stato di avanzamento dell'installazione. A questo punto proviamo a verificare i file installati:

```
[root@diamondhead /root]# rpm -ql xlispstat
/usr/bin/xlispstat
/usr/lib/xlispstat
/usr/lib/xlispstat/Autoload
/usr/lib/xlispstat/Autoload/_autoidx.lsp
/usr/lib/xlispstat/Autoload/bayes.fsl
/usr/lib/xlispstat/Autoload/glim.fsl
/usr/lib/xlispstat/Autoload/maximize.fsl
/usr/lib/xlispstat/Autoload/nonlin.fsl
/usr/lib/xlispstat/Autoload/oneway.fsl
/usr/lib/xlispstat/Autoload/stepper.fsl
/usr/lib/xlispstat/Data
/usr/lib/xlispstat/Data/absorbtion.lsp
...
```

si vedono i file installati e le relative directory. Proviamo ora a vedere le informazioni relative al pacchetto RPM:

```
[root@diamondhead /root]# rpm -qi xlispstat
```

Se si ha un pacchetto RPM da aggiornare su uno già installato basta usare l'opzione U al posto della lettera i.
Ora controlliamo se effettivamente il linguaggio LISP è disponibile nel nostro sistema:

```
[root@diamondhead /root]# xlispstat
```

vediamo che siamo già dentro l'interprete. Ora usciamo con Ctrl+D. Per sapere in quale directory xlispstat è stato installato basta fare un:

```
[root@diamondhead /root]# which xlispstat
/usr/bin/xlispstat
```

questo vale ovviamente per qualsiasi programma che si trova nella variabile PATH del nostro ambiente. Se facciamo un:

```
[root@diamondhead /root]# which httpd
/usr/sbin/httpd
```

vediamo che il programma apache si trova dentro /usr/sbin/ anche se però lo script di gestione (avvio/arresto) si trova dentro /etc/rc.d/init.d/httpd.
Proviamo ora a cancellare un pacchetto RPM. Proviamo con xlispstat appena installato.

```
[root@diamondhead /root]# rpm -e xlispstat
```

come si vede non è obbligatorio specificare la versione e quant'altro. Controlliamo se l'opzione -e ha fatto il suo dovere:

```
[root@diamondhead /root]# rpm -qi xlispstat
package xlispstat is not installed
```

come volevasi dimostrare.

Ora proviamo a compilare un codice sorgente. Andiamo a vedere su internet se troviamo qualcosa. Andiamo su google e proviamo a fare una ricerca per "chkrootkit", un applicativo che controlla eventuali virus e comandi sostituiti al nostro sistema.

(se non si trova niente)
Vi copio dentro la home directory di root l'archivio compresso chkrootkit.tar.gz.

```
[root@diamondhead /root]# scp chkrootkit.tar.gz 192.168.1.2:/root/.
```

(per tutti gi host da 2 a 13)

ora che abbiamo il nostro codice da compilare, creaimo una directory:

```
[root@diamondhead /root]# mkdir chkrootkit
```

e spostiamo l'archivio dentro la directory appena creata:

```
[root@diamondhead /root]# mv chkrootkit.tar.gz chkrootkit/.
```

ora entriamo nella directory:

```
[root@diamondhead /root]# cd chkrootkit/
[root@diamondhead chkrootkit]# ls
```

scompattiamo il tutto:

```
[root@diamondhead chkrootkit]# gzip -dvr chkrootkit.tar.gz
chkrootkit.tar.gz:       79.0% -- replaced with chkrootkit.tar
[root@diamondhead chkrootkit]# tar xvf chkrootkit.tar
```

ora entriamo nella directory appena creata:

```
[root@diamondhead chkrootkit]# cd chkrootkit-0.35/
```

e diamo un'occhiata al file README:

```
[root@diamondhead chkrootkit-0.35]# more README
```

nel punto 5 c'è la fase di installazione, basta fare un make sense per compilare il programma in C e poi possiamo utilizzare l'applicativo semplicemente mandando in esecuzione chkrootkit:

```
[root@diamondhead chkrootkit-0.35]# make sense
[root@diamondhead chkrootkit-0.35]# ./chkrootkit
```

il programma inizierà a controllare il filesystem alla ricerca di eventuali virus; allo stesso tempo il programma controlla anche possibili comandi modificati da un eventuale rootkit.

Roberto Terzo

roberto@r3d-engineering.net