Corso di Laboratorio di Sistemi Operativi

Lezione 1

Ivan Scagnetto
ivan.scagnetto@uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Informazioni generali

- Orario del corso:
  - Martedì 16:30–18:30, Venerdì 15:30–17:30
- Materiali del corso: `http://elearning.uniud.it/`
- Orario di ricevimento:
  - Su appuntamento
  - (e-mail: `ivan.scagnetto@uniud.it`)
- Modalità di superamento del corso:
  - il corso fa parte dell'esame di Sistemi Operativi e si supera svolgendo un'apposita prova scritta (in appelli regolari).
  - il voto ottenuto concorre alla determinazione del voto finale dell'esame:
    - media pesata dei voti di teoria e laboratorio:

$$\frac{9 \times \text{voto teoria} + 3 \times \text{voto lab.}}{12}$$

Motivazioni

- Uso della **shell** e **programmazione della shell** (scripting):
  - amministrazione di sistema (soprattutto remota);
  - automazione di task che coinvolgono molti elementi/dati.
- Linguaggio **C** e **programmazione di sistema**:
  - utilizzo di un linguaggio efficiente e "vicino" alla macchina (gestione diretta della memoria);
  - programmazione di sistemi embedded o con risorse limitate;
  - utilizzo delle system call di un sistema operativo per sfruttare in modo efficiente e diretto i servizi di quest'ultimo;
  - utilizzo efficace di game engine, physics engine ecc.

Programma e Bibliografia

Il programma del corso si suddivide in due parti:

- la shell di UNIX e GNU/Linux;
- programmazione di sistema, con introduzione al linguaggio C.

Testi di riferimento:

- R. Blum, C. Bresnahan, "Linux Command Line and Shell Scripting Bible", 2nd/3rd Edition, Wiley, 2011.
- G. Glass, K. Ables, "UNIX for Programmers and Users", Prentice Hall, 2a edizione, 1999.
- B.W. Kernighan, D.M. Ritchie. "Linguaggio C", 2a edizione, 1989.
- A. Kelley, I. Pohl. "C, Didattica e Programmazione", Pearson, 2a edizione, 2004.
- K. Haviland, D. Gray, B. Salama. "UNIX System Programming", Addison Wesley, 2a edizione, 1989.

Organizzazione delle lezioni

- Introduzione e spiegazione dei nuovi argomenti (variabile: circa metà lezione);
- consegna degli esercizi (5–10 min);
- svolgimento individuale degli esercizi;
- pubblicazione delle soluzioni ed eventuale discussione collettiva degli esercizi (in genere negli ultimi minuti della lezione o all'inizio della lezione successiva).

La Shell di UNIX e GNU/Linux

- La parte del sistema operativo UNIX dedita alla gestione dell'interazione con l'utente è la **shell**, ovvero, un'**interfaccia a carattere**:
  - l'utente impartisce i comandi al sistema digitandoli ad un apposito **prompt**;
  - il sistema stampa sullo schermo del terminale eventuali messaggi all'utente in seguito all'esecuzione dei comandi, facendo poi riapparire il prompt, in modo da continuare l'interazione.
- Versioni moderne di UNIX forniscono **X-Windows**, un gestore grafico (per interfacce a finestre), che consente di inviare comandi tramite menu, utilizzando un mouse.
- **X-Term** è un emulatore di terminale che gira sotto X-Windows, fornendo localmente un'interfaccia a carattere.
- Gestori di ambienti desktop (che girano su X-Windows) come **KDE**, **Gnome** ecc. forniscono altre versioni di emulatori di terminale più avanzati.

Per fare gli esercizi...

- In laboratorio: lanciare la **macchina virtuale Linux** disponibile sui PC ed aprire l'emulatore di terminale.
- A casa/sul proprio portatile:
  - installare una **distribuzione Linux** su una **partizione** del proprio disco (non serve installare l'ambiente grafico, basta la shell a carattere);
  - installare una distribuzione Linux su una macchina virtuale (gestita tramite **Virtual Box**, **VMware** ecc.) nel sistema operativo ospite (Windows, macOS ecc.);
  - i computer con macOS sono sostanzialmente delle macchine UNIX (quindi basta aprire l'**emulatore di terminale**, anche se tavolta c'è qualche problema di compatibilità relativo ai comandi della shell...);
  - per i PC con Windows si può installare **MinGW** (`http://www.mingw.org/`) o **Cygwin** (`https://www.cygwin.com/`), ovvero, ambienti che emulano UNIX direttamente su Windows oppure installare il **Windows Subsystem for Linux** di Windows 10/11.

Tipi di Shell

sh	Bourne shell
bash	Bourne again shell
csh	C shell
tcsh	Teach C shell
ksh	Korn shell

Quando viene invocata una shell, automaticamente al login o esplicitamente:

1. viene letto un file speciale nella home directory dello user, contenente informazioni per l'inizializzazione;
2. viene visualizzato un **prompt**, in attesa che l'utente invii un comando;
3. se l'utente invia un comando, la shell lo esegue e ritorna al punto 2; ad esempio, `echo $SHELL` stampa sullo schermo del terminale il percorso della shell di login, mentre il comando `bash` invoca la shell bash.

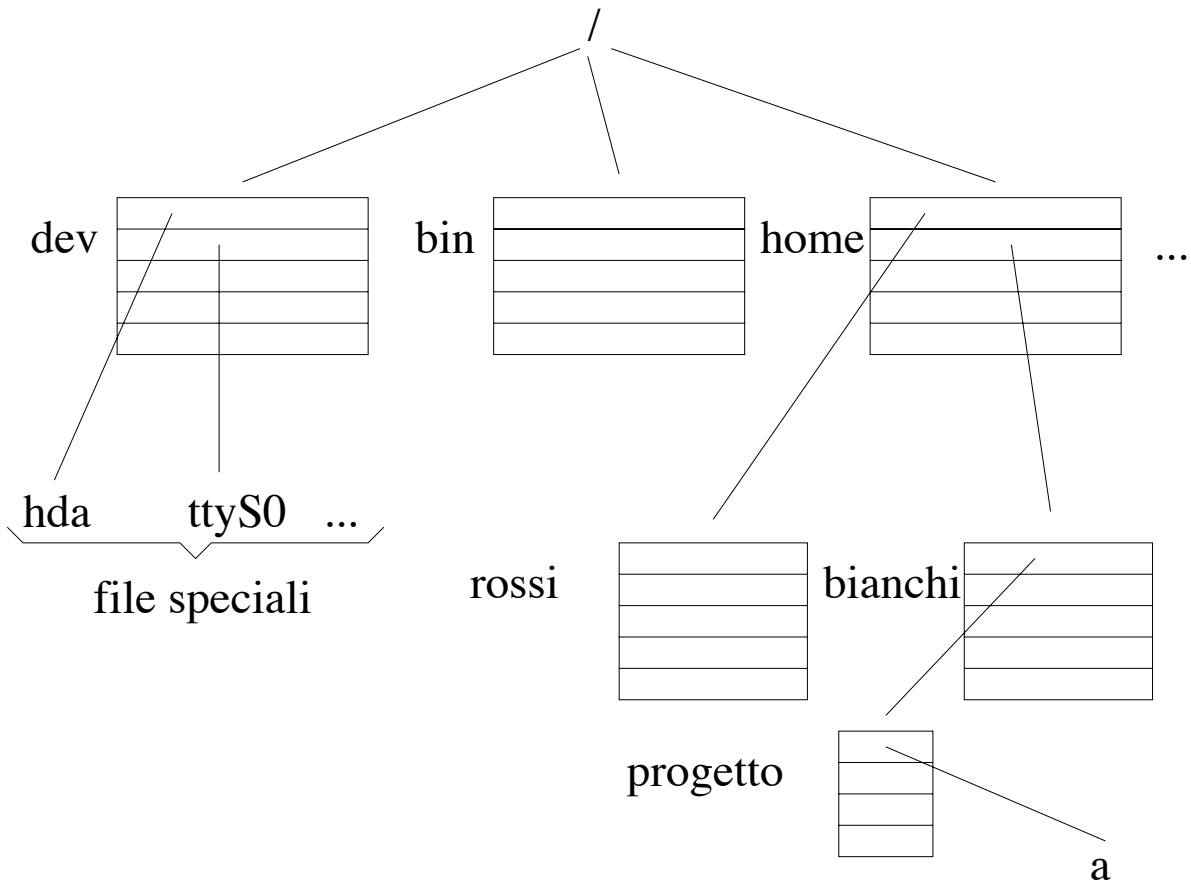
Per terminare la shell si possono usare i seguenti metodi:

- premere Ctrl-D (quando la linea di comando è vuota);
- digitare i comandi `logout` o `exit`.

File in UNIX

- Ordinari
- Directory
- Speciali

I file sono organizzati in una struttura gerarchica ad albero:



Il pathname

Ci si riferisce ai file con il **pathname** $\begin{cases} \text{assoluto} & \text{(rispetto a root /)} \\ \text{relativo} & \text{(rispetto alla dir. corrente)} \end{cases}$

Esempio:

(assoluto) `/home/bianchi/progetto/a`

(relativo) `progetto/a` (supponendo di trovarsi nella directory `/home/bianchi`)

- Present working directory:
  `> pwd`
  `/home/bianchi`
- Change directory:
  `> cd /bin` (cd senza argomenti sposta l'utente nella sua home directory)
- Per spostarsi nella directory "madre":
  `> cd ..` (dove `..` è l'alias per la directory "madre")
- `> pwd`
  `/home/bianchi`
  `> cd ./progetto` (dove `.` è l'alias per la directory corrente)
  `> pwd`
  `/home/bianchi/progetto`

Comandi per manipolare file e directory

- Listing dei file:
  ```
  > ls
  > ls -l
  > ls -a
  > ls -al
  > ls -l /bin
  > ...
  ```
- Creazione/rimozione di directory:
  ```
  > mkdir d1
  > rmdir d1
  ```
- Copia il file `f1` in `f2`:
  ```
  > cp f1 f2
  ```
- Sposta/rinomina il file `f1` in `f2`:
  ```
  > mv f1 f2
  ```
- `cp` e `mv` come primo argomento possono prendere una lista di file; in tal caso il secondo argomento deve essere una directory:
  `> cp f1 f2 f3 d1` (copia `f1`, `f2`, `f3` nella directory `d1`)

Un esempio d'uso del comando `ls`

Eseguendo il comando `ls -l /bin` si ottiene il seguente output:

```
...
lrwxrwxrwx    1 root     root            4 Dec  5  2000 awk -> gawk
-rwxr-xr-x    1 root     root         5780 Jul 13  2000 basename
-rwxr-xr-x    1 root     root       512540 Aug 22  2000 bash
...
```

da sinistra a destra abbiamo:

1. tipo di file (`-` file normale, `d` directory, `l` link, `b` block device, `c` character device),
2. permessi,
3. numero di hard link al file,
4. nome del proprietario del file,
5. nome dell'insieme di utenti che possono accedere al file come gruppo,
6. grandezza del file in byte,
7. data di ultima modifica,
8. nome del file.

I permessi dei file

Linux è un sistema **multiutente**. Per ogni file ci sono 4 categorie di utenti:
root, **owner**, **group**, **world**
L'amministratore del sistema (root) ha tutti i permessi (lettura, scrittura, esecuzione) su tutti i file. Per le altre categorie di utenti l'accesso ai file è regolato dai permessi:

```
> ls -l /etc/passwd
-rw-r--r--   1 root    root    981 Sep 20 16:32 /etc/passwd
```

Il blocco di caratteri `rw-r--r--` rappresenta i permessi di accesso al file. I primi 3 (`rw-`) sono riferiti all'owner. Il secondo blocco di 3 caratteri (`r--`) è riferito al group e l'ultimo blocco (`r--`) è riferito alla categoria world.
La prima posizione di ogni blocco rappresenta il permesso di **lettura** (`r`), la seconda il permesso di **scrittura** (`w`) e la terza il permesso di **esecuzione** (`x`). Un trattino (`-`) in una qualsiasi posizione indica l'assenza del permesso corrispondente.

N.B.: per "attraversare" una directory, bisogna avere il permesso di esecuzione su di essa.

Il comando `chmod`

L'owner di un file può cambiarne i permessi tramite il comando `chmod`:

- `> chmod 744 f1` (imposta i permessi del file `f1` a `rwxr--r--`)
  Infatti: `rwxr--r--` $\rightsquigarrow$ `111 100 100` $\rightsquigarrow$ `7 4 4` (leggendo ogni gruppo in ottale)
- `> chmod u=rwx,go=r f1` (produce lo stesso effetto del comando precedente)
  dove `u` rappresenta l'owner, `g` il gruppo e `o` il resto degli utenti (world)
  Inoltre:
  `+` aggiunge i permessi che lo seguono,
  `-` toglie i permessi che lo seguono,
  `=` imposta esattamente i permessi che lo seguono.
  Quindi l'effetto di `chmod g+r f1` è in generale diverso da `chmod g=r f1`.

Ulteriori comandi

- Visualizzazione del contenuto di un file:

```
> cat f1
> more f1
> tail f1
> head f1
```

- Consultazione del manuale on-line:
  * Sezione 1 : comandi
  * Sezione 2 : system call
  * Sezione 3 : funzioni di libreria
  * Sezione 4 : file speciali e driver
  * Sezione 5 : formati di file e convenzioni
  * Sezione 6 : giochi e screensaver
  * Sezione 7 : miscellanea
  * Sezione 8 : amministrazione di sistema e demoni

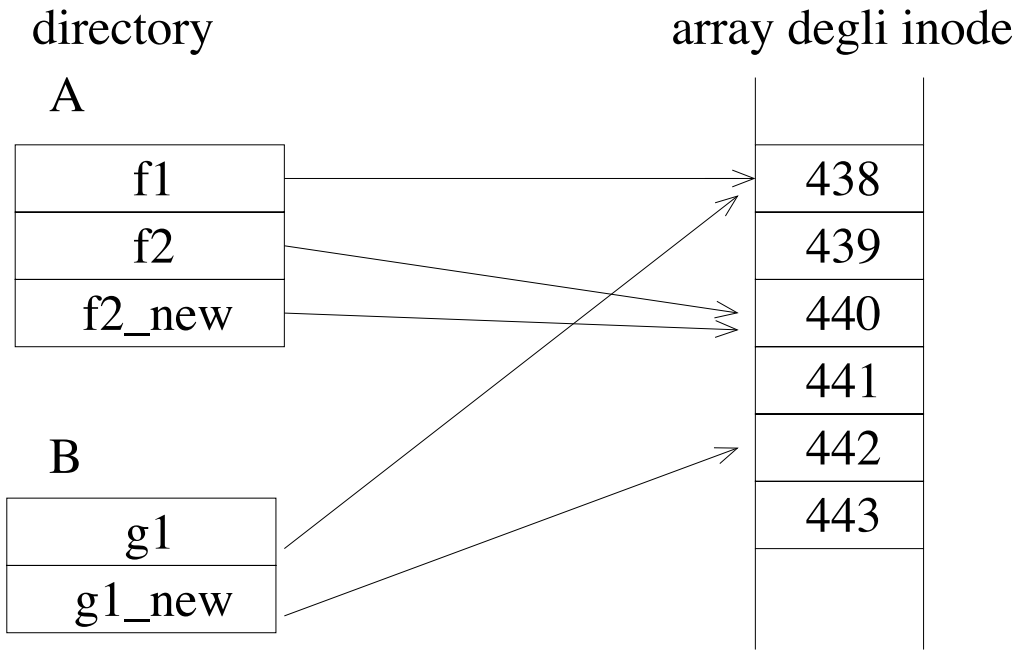
```
> man passwd
> man -a passwd
> man -s2 mkdir
> man man
```

- Pulizia dello schermo:

```
> clear
```

Inode e link

In UNIX, ad ogni file corrisponde un numero di **inode**, che è l'indice in un array memorizzato su disco.
Ogni elemento dell'array contiene le informazioni relative al file (data di creazione, proprietario, dove si trova il contenuto del file su disco, . . .).
Le directory sono tabelle che associano nomi di files a numeri di inode.
Ogni entry di una directory è un **link**.



Link e link simbolici

- Creazione di **link (hard)**, supponendo che la directory corrente sia `A`:
  `> ln f2 f2_new`
  il file f2_new ha lo stesso inode di f2
  `> ln f1 path-to-B/g1`
- Creazione di un **link simbolico**, supponendo che la directory corrente sia `B`:
  `> ln -s g1 g1_new`
  un link simbolico è un tipo di file speciale in UNIX; g1_new è un file di testo che contiene il pathname di g1

Esercizi

- Esplorate il vostro file system. Qual è il pathname della vostra home directory?
- Visualizzate i file della vostra home directory ordinati in base alla data di ultima modifica.
- Che differenza c'è tra i comandi `cat`, `more`, `tail`?
- Un link simbolico può puntare ad un altro link che a sua volta punta ad un file?
  Se è possibile, c'è un limite al numero di link simbolici che si possono avere in catena?
  Qual è lo svantaggio dei link simbolici rispetto ai link hard?
- Trovate un modo per ottenere l'elenco delle subdirectory contenute ricorsivamente nella vostra home.
- Trovate due modi diversi per creare un file (suggerimento: consultare la man page del comando `touch`).
- I seguenti comandi che effetto producono? Perché?

```
> cd
> mkdir d1
> chmod 444 d1
> cd d1
```

Corso di Laboratorio di Sistemi Operativi

Lezione 2

Ivan Scagnetto
ivan.scagnetto@uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

I Metacaratteri della Shell Unix

La shell Unix riconosce alcuni caratteri speciali, chiamati **metacaratteri**, che possono comparire nei comandi.
Quando l'utente invia un comando, la shell lo scandisce alla ricerca di eventuali metacaratteri, che processa in modo speciale.
Una volta processati tutti i metacaratteri, viene eseguito il comando.
Esempio:

```
> ls *.java
 Albero.java           div.java          ProvaAlbero.java
 AreaTriangolo.java    EasyIn.java       ProvaAlbero1.java
 AreaTriangolo1.java   IntQueue.java
```

Il **metacarattere** `*` all'interno di un pathname è un'**abbreviazione** per un nome di file. Il pathname `*.java` viene espanso dalla shell con tutti i nomi di file che terminano con l'estensione `.java`. Il comando `ls` fornisce quindi la lista di tutti e soli i file con tale estensione.

Abbreviazione del Pahtname

I seguenti metacaratteri, chiamati **wildcard** sono usati per **abbreviare** il nome di un file in un pathname:

Metacarattere	**Significato**
*	stringa di 0 o più caratteri
?	singolo carattere
[ ]	singolo carattere tra quelli elencati
{ }	sequenza di stringhe

Esempi:

```
> cp /JAVA/Area*.java /JAVA_backup
```

copia tutti i file il cui nome inizia con la stringa `Area` e termina con l'estensione `.java` nella directory `JAVA_backup`.

```
> ls /dev/tty?
/dev/ttya /dev/ttyb
```

... esempi

```
> ls /dev/tty?[234]
/dev/ttyp2 /dev/ttyp4 /dev/ttyq3 /dev/ttyr2
/dev/ttyr4
/dev/ttyp3 /dev/ttyq2 /dev/ttyq4 /dev/ttyr3
> ls /dev/tty?[2-4]
/dev/ttyp2 /dev/ttyp4 /dev/ttyq3 /dev/ttyr2
/dev/ttyr4
/dev/ttyp3 /dev/ttyq2 /dev/ttyq4 /dev/ttyr3
> mkdir /user/studenti/rossi/{bin,doc,lib}
```

crea le directory `bin`, `doc`, `lib` .

Il "quoting"

Il meccanismo del **quoting** è utilizzato per inibire l'effetto dei metacaratteri. I metacaratteri a cui è applicato il quoting perdono il loro significato speciale e la shell li tratta come caratteri ordinari. Ci sono tre meccanismi di quoting:

- il metacarattere di **escape** \ inibisce l'effetto speciale del metacarattere che lo segue:

```
> cp file file\?
> ls file*
file     file?
```

- tutti i metacaratteri presenti in una stringa racchiusa tra **singoli apici** perdono l'effetto speciale:

```
> cat 'file*?'
...
```

- i metacaratteri per l'abbreviazione del pathname presenti in una stringa racchiusa tra **doppi apici** perdono l'effetto speciale (ma non tutti i metacaratteri della shell):

```
> cat "file*?"
...
```

Redirezione dell'I/O

Di default i comandi Unix prendono l'input da **tastiera** (**standard input**) e mandano l'**output** ed eventuali **messaggi di errore** su video (**standard output, error**).
L'input/output in Unix può essere **rediretto** da/verso **file**, utilizzando opportuni metacaratteri:

Metacarattere	**Significato**
>	redirezione dell'output
>>	redirezione dell'output (append)
<	redirezione dell'input
<<	redirezione dell'input dalla linea di comando ("here document")
2>	redirezione dei messaggi di errore (bash Linux)

Esempi:

```
> ls LABSO > temp
> more temp
lezione1.aux lezione1.log lezione1.tex lezione2.dvi
lezione2.tex
lezione1.dvi lezione1.ps lezione2.aux lezione2.log
lezione2.tex~
```

... esempi

```
> echo ciao a tutti >file     # redirezione dell'output
> more file
ciao a tutti
> echo ciao a tutti >>file      # redirezione dell'output
(append)
> more file
ciao a tutti
ciao a tutti
```

Il comando `wc` (**word counter**) fornisce numero di linee, parole, caratteri di un file:

```
> wc <progetto.txt
21 42 77
> wc <<delim      # here document
?  queste linee formano il contenuto
?  del testo
?  delim
   2    7    44
> man -s2 passwd      # redirezione dei messaggi di errore
No entry for passwd in section(s) 2 of the manual.
> man -s2 passwd 2>temp
```

Pipe

Il metacarattere | (**pipe**) serve per comporre n comandi "in cascata" in modo che l'output di ciascuno sia fornito in input al successivo. L'output dell'ultimo comando è l'output della pipeline.
La sequenza di comandi

```
> ls /usr/bin > temp
> wc -w < temp
459
```

ha lo stesso effetto della pipeline:

```
> ls /usr/bin | wc -w
459
```

I comandi `ls` e `wc` sono eseguiti in parallelo: l'output di `ls` è letto da `wc` mano a mano che viene prodotto.
Per mandare in stampa la lista dei file in `/usr/bin`:

```
> ls /usr/bin | lpr
```

Per visualizzare l'output di `ls` pagina per pagina

```
> ls | more
```

Esercizi

- Scrivete un unico comando (pipeline) per ognuno dei seguenti compiti:
  - copiare il contenuto della directory `dir1` nella directory `dir2`;
  - fornire il numero di file (e directory) a cui avete accesso, contenuti ricorsivamente nella directory `/home` (si può utilizzare `ls -R`? e con il comando `find`?);
  - fornire la lista dei file della home directory il cui nome è una stringa di 3 caratteri seguita da una cifra.
- Qual è la differenza tra i seguenti comandi?

  ```
  ls
  ls | cat
  ls | more
  ```
- Quale effetto producono i seguenti comandi?
  - `uniq < file`, dove `file` è il nome di un file;
  - `who | wc -l` ;
  - `ps -e | wc -l` .

Corso di Laboratorio di Sistemi Operativi

Lezione 3

Ivan Scagnetto
ivan.scagnetto@uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Bash: history list (I)

L'**history list** è un tool fornito dalla shell bash che consente di evitare all'utente di digitare più volte gli stessi comandi:

- bash memorizza nell'history list gli ultimi **500 comandi** (1000 nelle distribuzioni più recenti) inseriti dall'utente;
- l'history list viene memorizzata nel file `.bash_history` nell'home directory dell'utente al momento del logout (e riletta al momento del login);
- il comando `history` consente di visualizzare la lista dei comandi:

```
$ history | tail -5
    511 pwd
    512 ls -al
    513 cd /etc
    514 more passwd
    515 history | tail -5
```

- ogni riga prodotta dal comando history è detta **evento** ed è preceduta dal **numero dell'evento**.

Bash: history list (II)

Conoscendo il numero dell'evento corrispondente al comando che vogliamo ripetere, possiamo eseguirlo, usando il metacarattere `!`:

```
$ !515
history | tail -5
    512 ls -al
    513 cd /etc
    514 more passwd
    515 history | tail -5
    516 history | tail -5
```

Se l'evento che vogliamo ripetere è l'ultimo della lista è sufficiente usare `!!`:

```
$ !!
history | tail -5
    513 cd /etc
    514 more passwd
    515 history | tail -5
    516 history | tail -5
    517 history | tail -5
```

Bash: history list (III)

Oltre a riferirsi agli eventi tramite i loro numeri, è possibile eseguire delle `ricerche testuali` per individuare quello a cui siamo interessati:

```
$ !ls
ls -al
total 491
drwxr-xr-x  16 root    root       0 Oct 15 21:35 .
drwxr-xr-x  16 root    root       0 Oct 15 21:35 ..
-rw-r--r--   1 root    root   87515 Jul 10 04:28 Muttrc
drwxr-xr-x   2 root    root       0 Oct 15 21:27 WindowMaker
...
```

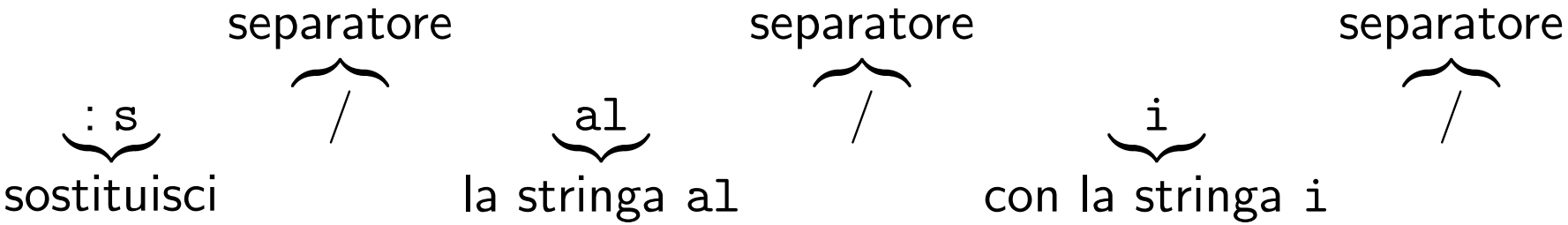
In questo modo la shell comincia a cercare a partire dall'ultimo evento, procedendo a ritroso, nell'history list un comando che inizi con `ls`. Racchiudendo con due caratteri `?` la stringa da ricercare (e.g. `$ !?ls?`), la shell controllerà che quest'ultima appaia in un punto qualsiasi del comando (non necessariamente all'inizio).

Bash: history list (IV)

Talvolta può capitare di voler ripetere un comando eseguito precedentemente dopo aver operato qualche modifica:

```
$ !ls:s/al/i/
ls -i
1067566292 Muttrc            123123 mib.txt
 204714 WindowMaker          123127 mime.conf
...
```

In questo modo la shell, dopo aver trovato l'evento cercato (`ls -al`), sostituisce la stringa `al` con `i` (`:s/al/i/`), producendo il comando `ls -i`.



Bash: command line editing

La shell bash mette a disposizione dell'utente dei semplici **comandi di editing** per facilitare la ripetizione degli eventi:

- utilizzando i **tasti cursore**:
  - con la **freccia verso l'alto** si scorre l'history list a ritroso (un passo alla volta) facendo apparire al prompt il comando corrispondente all'evento;
  - analogamente con la **freccia verso il basso** si scorre l'history list nella direzione degli eventi più recenti.
  - le frecce sinistra e destra consentono di spostare il cursore sulla linea di comando verso il punto che si vuole editare;
- le combinazioni di tasti `Ctrl-A` e `Ctrl-E` spostano il cursore, rispettivamente all'inizio ed alla fine della linea di comando;
- il tasto `Backspace` consente di cancellare il carattere alla sinistra del cursore;
- il tasto invio (`enter`) esegue il comando.

Bash: command completion (I)

Una caratteristica molto utile della shell bash è la sua abilità di **tentare di completare** ciò che stiamo digitando al prompt dei comandi (nel seguito <Tab> indica la pressione del tasto Tab).

```
$ pass<Tab>
```

La pressione del tasto <Tab> fa in modo che la shell, sapendo che vogliamo impartire un comando, cerchi quelli che iniziano con la stringa pass. Siccome l'unica scelta possibile è data da passwd, questo sarà il comando che ritroveremo automaticamente nel prompt.
Se il numero di caratteri digitati è insufficiente per la shell al fine di determinare univocamente il comando, avviene quanto segue:

- viene prodotto un suono di avvertimento al momento della pressione del tasto Tab;
- alla seconda pressione del tasto Tab la shell visualizza una lista delle possibili alternative.
- digitando ulteriori caratteri, alla successiva pressione del tasto Tab, la lunghezza della lista diminuirà fino ad individuare un unico comando.

Bash: command completion (II)

Oltre a poter completare i comandi, la shell bash può anche completare i nomi dei file usati come argomento:

```
$ tail -2 /etc/p<Tab><Tab>
passwd  printcap  profile
$tail -2 /etc/pa<Tab><Invio>
bianchi:fjKppCZxEvouc:500:500::/home/bianchi:/bin/bash
rossi:Yt1a4ffkGr02:501:500::/home/rossi:/bin/bash
```

In questo caso alla prima doppia pressione del tasto Tab, la shell presenta tre possibili alternative; digitando una a e premendo il tasto Tab, la shell ha una quantità di informazione sufficiente per determinare in modo univoco il completamento del nome di file.

Alias

Alias già visti:

1. `.` (directory corrente)
2. `..` (directory madre)

Esiste anche l'alias `~user` che sta per la directory home dell'utente `user`:

```
user> cd ~user          # equivale a cd /home/user
user> cd ~user/doc      # equivale a cd /home/user/doc
```

Gli alias sono trattati dalla shell come i metacaratteri, nel senso che la shell scandisce la linea di comando impartita dall'utente processando i caratteri alias prima di eseguire i comandi.

Il comando alias

Il comando `alias` serve per creare nuovi alias:

```
user> alias dir='ls -a'
user> dir
.   ..   .bash_history
user> alias ls='ls -l'
user> ls *.java
-rw-r--r--   1 user    users    0 Oct 16 17:24 Figura.java
-rw-r--r--   1 user    users    0 Oct 16 17:24 Quadrato.java
-rw-r--r--   1 user    users    0 Oct 16 17:24 Triangolo.java
```

Per rimuovere uno o più alias:

```
user> unalias dir ls
```

All'uscita dalla shell gli alias creati con il comando `alias` sono automaticamente rimossi.

Metacaratteri comuni a tutte le shell (I)

Simbolo	Significato	Esempio d'uso
`>`	Ridirezione dell'output	`ls >temp`
`>>`	Ridirezione dell'output (append)	`ls >>temp`
`<`	Ridirezione dell'input	`wc -l <text`
`<<delim`	ridirezione dell'input da linea di comando (here document)	`wc -l <<delim`
`*`	Wildcard: stringa di 0 o più caratteri, ad eccezione del punto (`.`) e di stringhe che iniziano con il punto	`ls *.c`
`?`	Wildcard: un singolo carattere, ad eccezione del punto (`.`) se a inizio stringa	`ls ?.c`
`[...]`	Wildcard: un singolo carattere tra quelli elencati	`ls [a-zA-Z].bak`
`{...}`	Wildcard: le stringhe specificate all'interno delle parentesi	`ls {prog,doc}*.txt`

Metacaratteri comuni a tutte le shell (II)

Simbolo	Significato	Esempio d'uso
`|`	Pipe	`ls | more`
`;`	Sequenza di comandi	`pwd;ls;cd`
`||`	Esecuzione condizionale. Esegue un comando se il precedente fallisce.	`cc prog.c || echo errore`
`&&`	Esecuzione condizionale. Esegue un comando se il precedente termina con successo.	`cc prog.c && a.out`
`(...)`	Raggruppamento di comandi	`(date;ls;pwd)>out.txt`
`#`	Introduce un commento	`ls # lista di file`
`\`	Fa in modo che la shell non interpreti in modo speciale il carattere che segue.	`ls file.\*`
`!`	Ripetizione di comandi memorizzati nell'history list	`!ls`

Controllo di processi

Ogni processo del sistema ha un **PID** (**Process Identity Number**).
Ogni processo può generare nuovi processi (figli).
La radice della gerarchia di processi è il processo **init** con PID=1.
init è il primo processo che parte al boot di sistema.
Il comando `ps` fornisce i processi presenti nel sistema:

```
user> ps   # fornisce i processi dell'utente associati
           # al terminale corrente
```

```
   PID          TTY          TIME     CMD
 23228          pts/15       0:00     xdvi.bin
  9796          pts/15       0:01     bash
 23216          pts/15       0:04     xemacs-2
  9547          pts/15       0:00     csh
```

Legenda: `PID` = PID; `TTY` = terminale (virtuale); `TIME` = tempo di CPU utilizzato; `CMD` = comando che ha generato il processo.
Per ottenere il nome del terminale corrente:

```
user> tty
 /dev/pts/15
```

Il comando ps e sue varianti, I

Per ottenere tutti i processi nel sistema associati ad un terminale (-a), full listing (-f):

```
user> ps -af

  UID   PID  PPID   C     STIME TTY          TIME CMD
    0  1699  1697   0   0:00.02 ttys000    0:00.03 login -fp ivan
  501  1700  1699   0   0:00.02 ttys000    0:00.08 -bash
  501  1740  1700   0   0:00.14 ttys000    0:00.58 coqtop -opt
    0  3259  1697   0   0:00.01 ttys001    0:00.02 login -fp ivan
  501  3260  3259   0   0:00.01 ttys001    0:00.06 -bash
    0  3313  3260   0   0:00.00 ttys001    0:00.00 ps -af
                ...
```

Legenda: UID = User Identifier; PPID = Parent PID; C = informazione obsoleta sullo scheduling; STIME = data di inizio del processo.

Il comando ps e sue varianti, II

Per ottenere tutti i processi nel sistema, anche non associati ad un terminale (-e), long listing (-l):

```
user> ps -el

 F S   UID   PID  PPID  C PRI NI    ADDR     SZ   WCHAN  TTY       TIME CMD
19 T     0     0     0  0   0 SY       ?      0          ?         0:12 sched
 8 S     0     1     0  0  41 20       ?    100        ?  ?         0:03 init
 8 S   140 12999 12997  0  56 20       ?    278        ?  pts/12   0:00 tcsh
 8 R   159  9563  9446  0  50 20       ?   2110           ?         0:30 acroread
         ...
```

Legenda: F = flag obsoleti; S = stato del processo (T=stopped); PRI = priorità; NI = nice value (usato per modificare la priorità); ADDR = indirizzo in memoria; SZ = memoria virtuale usata; WCHAN = evento su cui il processo è sleeping.

Terminazione di un processo

Per arrestare un processo in esecuzione si può utilizzare

- la sequenza `Ctrl-c` dal terminale stesso su cui il processo è in esecuzione;
- il comando `kill` seguito dal `PID` del processo (da qualsiasi terminale):

```
user> ps
  PID    TTY      TIME CMD
      ...
  28015  pts/14   0:01 xemacs
      ...

user> kill 28015
```

- il comando `kill` con il segnale SIGKILL

```
user> kill -9 28015
user> kill -s kill 28015
```

Processi in background

Un comando (pipeline, sequenza) seguito da & dà luogo ad uno o più **processi in background**. I processi in background sono eseguiti in una **sottoshell**, **in parallelo** al processo padre (la shell) e **non** sono controllati da tastiera.
I processi in background sono quindi utili per eseguire task in parallelo che non richiedono controllo da tastiera.

```
user> xemacs &
  [1] 24760
```

[1] è il numero del job, 24760 il PID del processo

```
user> xemacs &
user> ls -R / >temp 2>err &
```

Il comando `jobs` mostra la lista dei job in esecuzione:

```
user> jobs
  [1]   Running                 xemacs &
  [2]-  Running                 xemacs &
  [3]+  Running                 ls -R / >tmp 2>err
```

Controllo di job

Un job si può **sospendere** e poi **rimandare in esecuzione**

```
user> cat >temp    # job in foreground

   Ctrl-z   # sospende il job

  [1]+ Stopped

user> jobs
  [1]+ Stopped     cat >temp

user> fg    # fa il resume del job in foreground

   Ctrl-z   # sospende il job

user> bg    # fa il resume del job in background

user> kill %1   # termina il job 1
  [1]+ Terminated
```

Monitoraggio della memoria

Il comando top fornisce informazioni sulla memoria utilizzata dai processi, che vengono aggiornate ad intervalli di qualche secondo. I processi sono elencati secondo la quantità di tempo di CPU utilizzata.

```
user> top
load averages:  0.68,  0.39,  0.27                                  14:34:55
245 processes: 235 sleeping, 9 zombie, 1 on cpu
CPU states: 91.9% idle,  5.8% user,  2.4% kernel,  0.0% iowait,  0.0% swap
Memory: 768M real, 17M free, 937M swap in use, 759M swap free

   PID USERNAME THR PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
 12887 root       1  59    0   65M   56M sleep  105:00  3.71% Xsun
  4210 lenisa     1  48    0 2856K 2312K cpu      0:00  1.50% top
  9241 root       1  59    0   35M   26M sleep   15:58  1.47% Xsun
 24389 pietro     4  47    0   28M   25M sleep   16:30  0.74% opera
   ......
```

Legenda: la prima riga indica il carico del sistema nell'ultimo minuto, negli ultimi 5 minuti, negli ultimi 15 minuti, risp.; il carico è espresso come numero di processori necessari per far girare tutti i processi a velocità massima; alla fine della prima riga c'è l'ora; la seconda contiene numero e stato dei processi nel sistema; la terza l'utilizzo della CPU; la quarta informazioni sulla memoria; le restanti righe contengono informazioni sui processi (THR=thread, RES=resident).

Esercizi (I)

- Ridefinire il comando `rm` in modo tale che non sia chiesta conferma prima della cancellazione dei file.
- Definire il comando `rmi` (`rm` interattivo) che chiede conferma prima di rimuovere un file.
- Sapendo che il comando `ps` serve ad elencare i processi del sistema, scrivere una pipeline che fornisca in output il numero di tutti i processi in esecuzione.
- Salvare in un file di testo l'output dell'ultimo evento contenente il comando `ls`.
- Scrivere un comando che fornisce il numero dei comandi contenuti nella history list.

Esercizi (II)

- Scrivere un comando che fornisce i primi 15 comandi della history list.
- Quali sono i comandi Unix disponibili nel sistema che iniziano con `lo`?
- Fornire almeno due modi diversi per ottenere la lista dei file della vostra home directory il cui nome inizia con `al`.
- Qual è l'effetto dei seguenti comandi?
  - `ls -R || (echo file non accessibili > tmp)`
  - `(who | grep rossi) && cd ~rossi`
  - `(cd / ; pwd ;  ls | wc -l )`

Esercizi (III)

- Qual è la differenza tra **programma** e **processo**?
- Qual è la differenza tra **processo** e **job**?
- Scrivere una pipeline che fornisca in output il numero di processi appartenenti all'utente `root`.
- Il comando

  ```
  > emacs &
  ```

  provoca l'avvio di un processo in **background**. Invece il comando

  ```
  > emacs
  ```

  provoca l'avvio di un processo in **foreground**. Come si può mandare tale processo in esecuzione in background in modo da rendere il terminale nuovamente disponibile per l'invio di ulteriori comandi?

Corso di Laboratorio di Sistemi Operativi

Lezione 4

Ivan Scagnetto
ivan.scagnetto@uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

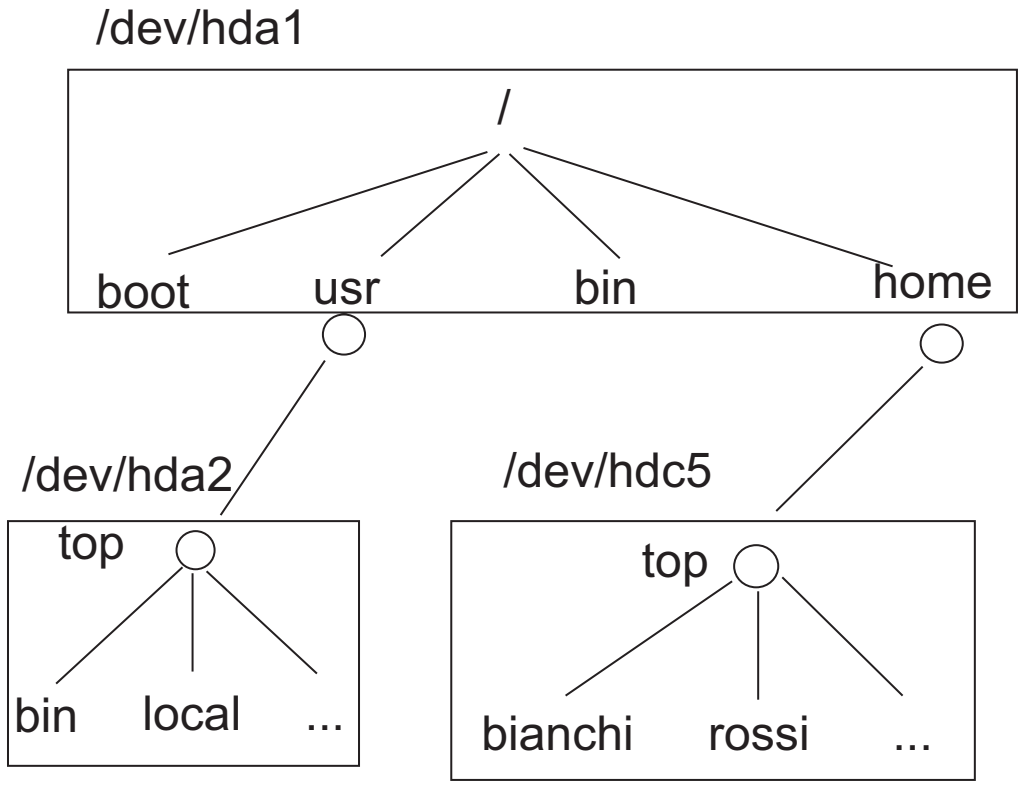
Il filesystem di Unix/Linux (I)

- Comunemente, in un elaboratore l'informazione è memorizzata in modo permanente nei **dischi fissi**.
- Ogni disco fisso può essere suddiviso in **partizioni**.
- Ogni partizione può contenere un filesystem con una propria **top level directory**.

Sorge quindi il problema di come permettere agli utenti di accedere ai vari filesystem contenuti nelle differenti partizioni:

- la prima possibilità consiste nell'avere **root directory distinte** (e.g. in Windows: `C:\`, `D:\` ecc.): una per ogni partizione; quindi per riferirsi ad un file, bisogna usare un pathname che parta dalla root directory giusta (e.g. `D:\Doc\bilancio.xls`).
- Unix/Linux invece fa in modo che i diversi filesystem vengano combinati in un'**unica struttura gerarchica**, "montando" la top level directory di una partizione come foglia del filesystem di un'altra partizione.

Il filesystem di Unix/Linux (II)



dove `/dev/hda1`, `/dev/hda2` e `/dev/hdc5` sono partizioni differenti. Le informazioni su quali filesystem montare al boot ed in che modo sono contenute nel file `/etc/fstab`. Il comando per montare i filesystem è `mount <file speciale> <mount point>` e, solitamente, solo l'utente `root` può utilizzarlo. `mount` senza argomenti elenca i filesystem in uso nel sistema.

Controllo dello spazio su disco

Per controllare la quantità di spazio su disco in uso:

```
user> df
Filesystem      1K-blocks       Used  Available Use% Mounted on
/dev/sda2       472332568  389526168   58790172  87% /
udev                10240          0      10240   0% /dev
tmpfs              814184       1300     812884   1% /run
tmpfs                5120          4       5116   1% /run/lock
tmpfs             3306080          4    3306076   1% /run/shm
/dev/sdb1      7752327768  497103336 6864506724   7% /mnt/disk8T
/dev/sdc1      2884152988 1366180464 1371442864  50% /mnt/disk3T
tmpfs              814184          4     814180   1% /run/user/1000
```

Legenda: il primo campo contiene il device corrispondente (eventualmente virtuale); il secondo il numero di blocchi totale; il terzo il numero di blocchi occupati; il quarto il numero di blocchi liberi; il quinto la percentuale in uso ed il sesto il punto di mount.

Per controllare la quantità di spazio su disco utilizzata da una directory (in blocchi):

```
user> du LABORATORIO_SO
  8       LABORATORIO_SO/LABSO/CVS
  16884   LABORATORIO_SO/LABSO
  16      LABORATORIO_SO/scriptColonne
  14      LABORATORIO_SO/linguaggio_c
  17342   LABORATORIO_SO
```

Ulteriori comandi sui file

- Confronto tra file:
  1. `> cmp file1 file2`
     restituisce il primo byte ed il numero di linea in cui `file1` e `file2` differiscono (se sono uguali, non viene stampato nulla a video).
  2. `> diff file1 file2`
     restituisce la lista di cambiamenti da apportare a `file1` per renderlo uguale a `file2`.
- Ricerca di file:

  `> find` <*pathnames*> <*expression*>
  attraversa ricorsivamente le directory specificate in <*pathnames*> applicando le regole specificate in <*expression*> a tutti i file e sottodirectory trovati.
  <*expression*> può essere una fra le seguenti:
  1. opzione,
  2. condizione,
  3. azione.

Esempi d'uso di `find`

- `> find . -name '*.c' -print`
  cerca ricorsivamente a partire dalla directory corrente tutti i file con estensione c e li stampa a video.
- `> find . -name '*.bak' -ls -exec rm {} \;`
  cerca ricorsivamente a partire dalla directory corrente tutti i file con estensione bak, li stampa a video con i relativi attributi (`-ls`) e li cancella (`-exec rm {} \;` Il carattere `\` serve per fare il "quote" del `;`).
- `> find /etc -type d -print`
  cerca ricorsivamente a partire dalla directory `/etc` tutte e solo le sottodirectory, stampandole a video.

I comandi filtro

I **filtri** sono una particolare classe di comandi che possiedono i seguenti requisiti:

- prendono l'input dallo **standard input device**,
- effettuano delle operazioni sull'input ricevuto,
- inviano il risultato delle operazioni allo **standard output device**.

Tali comandi risultano quindi degli ottimi strumenti per costruire pipeline che svolgano compiti complessi.
Ad esempio:

```
> uniq file
```

restituisce in output il contenuto del file `file`, sostituendo le linee adiacenti uguali con un'unica linea.

Comandi filtro: grep, fgrep, egrep

I comandi:

- grep: General Regular Expression Parser,
- fgrep: Fixed General Regular Expression Parser,
- egrep: Extended General Regular Expression Parser,

restistuiscono solo le linee dell'input fornito che contengono un **pattern** specificato tramite espressione regolare o stringa fissata.

Sintassi:

grep [options] pattern [filename...]
fgrep [options] string [filename...]
egrep [options] pattern [filename...]

Opzioni:

-i: ignora la distinzione fra lettere maiuscole e minuscole,
-l: fornisce la lista dei file che contengono il pattern/string,
-n: le linee in output sono precedute dal numero di linea,
-v: vengono restituite solo le linee che **non** contengono il pattern/string,
-w: vengono restituite solo le linee che contengono il pattern/string come parola completa,
-x: vengono restituite solo le linee che coincidono esattamente con pattern/string.

I metacaratteri delle espressioni regolari

metacarattere	tipo	significato
`^`	B	inizio della linea
`$`	B	fine della linea
`\<`	B	inizio di una parola
`\>`	B	fine di una parola
`.`	B	un singolo carattere (qualsiasi)
`[str]`	B	un qualunque carattere in `str`
`[^str]`	B	un qualunque carattere non in `str`
`[a-z]`	B	un qualunque carattere tra `a` e `z`
`\`	B	inibisce l'interpretazione del metacarattere che segue
`*`	B	zero o più ripetizioni dell'elemento precedente
`+`	E	una o più ripetizioni dell'elemento precedente
`?`	E	zero od una ripetizione dell'elemento precedente
`{j,k}`	E	un numero di ripetizioni compreso tra `j` e `k` dell'elemento precedente
`s|t`	E	l'elemento `s` oppure l'elemento `t`
`(exp)`	E	raggruppamento di `exp` come singolo elemento

dove B (basic) indica che la sequenza di caratteri è utilizzabile sia in `grep` che in `egrep`, mentre E (extended) indica che la sequenza di caratteri è utilizzabile solo in `egrep` (o in `grep` usando l'opzione `-E`).

Esempi d'uso di grep, fgrep, egrep

- `> fgrep rossi /etc/passwd`
  fornisce in output le linee del file `/etc/passwd` che contengono la stringa **fissata** `rossi`.
- `> egrep -nv '[agt]+' relazione.txt`
  fornisce in output le linee del file `relazione.txt` che **non** contengono stringhe composte dai caratteri `a`, `g`, `t` (ogni linea è preceduta dal suo numero).
- `> grep -w print *.c`
  fornisce in output le linee di tutti i file con estensione `c` che contengono la parola **intera** `print`.
- `> ls -al . | grep '^d.......w.'`
  fornisce in output le sottodirectory della directory corrente che sono modificabili dagli utenti ordinari.
- `> egrep '[a-c]+z' doc.txt`
  fornisce in output le linee del file `doc.txt` che contengono una stringa che ha un prefisso di lunghezza non nulla, costituito solo da lettere `a`, `b`, `c`, seguito da una `z`.

Comandi filtro: `sort`

Il comando `sort` prende in input delle linee di testo, le **ordina** (tenendo conto delle opzioni specificate dall'utente) e le invia in output.

- `sort` tratta ogni linea come una collezione di vari campi separati da delimitatori (default: spazi, tab ecc.).
- l'ordinamento di default avviene in base a tutta la linea (i.e., in base a tutti i suoi campi) ed è **alfabetico**.

Il comportamento di default si può cambiare tramite le opzioni:

`-b`	ignora eventuali spazi presenti nelle chiavi di ordinamento,
`-f`	ignora le distinzioni fra lettere maiuscole e minuscole,
`-n`	considera numerica (invece che testuale) la chiave di ordinamento
`-r`	ordina in modo decrescente,
`-o` *file*	invia l'output al file *file* invece che allo standard output,
`-t` *s*	usa *s* come separatore di campo,
`-k` *s1*`,`*s2*	usa i campi da *s1* a *s2* come chiavi di ordinamento ed eventualmente i successivi (fino a fine linea) in caso di "pareggio",
`-s`	rende "stabile" il confronto, impedendo di ricorrere a campi ulteriori, se il confronto risulta in un "pareggio" sulle chiavi di ordinamento.

Esempi d'uso di sort

Volendo ordinare le righe del file /etc/passwd in base al terzo campo (user ID), il comando

```
> sort -t: -k3,3 /etc/passwd  # in questo caso basta -k3
root:x:0:1:Super-User:/:/sbin/sh
guest:x:1001:120:Guest User:/home/guest:/usr/local/bin/bash
daemon:x:1:1::/:
...
```

non dà il risultato voluto in quanto di default l'ordinamento è alfabetico, mentre il campo user ID è un numero; quindi si rende necessaria l'opzione -n:

```
> sort -t: -k3,3 -n /etc/passwd # in questo caso basta -k3
root:x:0:1:Super-User:/:/sbin/sh
daemon:x:1:1::/:
bin:x:2:2::/usr/bin:
...
guest:x:1001:120:Guest User:/home/guest:/usr/local/bin/bash
...
```

Si noti che in entrambi gli esempi il separatore (:) è stato impostato con l'opzione -t:.

Comandi filtro: `tr`

Character translation: `tr` è un semplice comando che permette di eseguire operazioni come la conversione di lettere minuscole in maiuscole, cancellazione della punteggiatura ecc. Siccome può prendere input soltanto dallo standard input e stampare soltanto sullo standard output, bisogna usare delle pipe o delle ridirezioni di input/output per farlo leggere/scrivere su file.
Sintassi di base: `> tr str1 str2`
(i caratteri in `str1` vengono sostituiti con i caratteri in posizione corrispondente della stringa `str2`)
Esempi:

- `> tr a-z A-Z`
  converte le minuscole in maiuscole.
- `> tr -c A-Za-z0-9 ' '`
  sostituisce tutti i caratteri **non** (opzione `-c`: complemento) alfanumerici con degli spazi.
- `> tr -cs A-Za-z0-9 ' '`
  come nell'esempio precedente, ma comprime gli spazi adiacenti in un'unico spazio (opzione `-s`: *squeeze*).
- `> tr -d str`
  cancella i caratteri contenuti nella stringa `str`.

Cut and paste

- Il comando `cut` serve ad estrarre delle colonne specifiche dalle linee di testo che riceve in input:

```
> cut -d: -f1 /etc/passwd
root
daemon
bin
...
```

  il separatore si specifica con l'opzione `-d` (*delimiter*), il campo da estrarre con l'opzione `-f` (*field*).

- Il comando `paste` combina le righe corrispondenti di due file, inserendo un delimitatore fra esse (default: `<Tab>`):

```
> cd; cut -d: -f1 /etc/passwd > p1.txt; cut -d: -f6 /etc/passwd > p6.txt
> paste p1.txt p6.txt
root    /
daemon  /
bin     /usr/bin
...
```

Esercizi

- Scoprire quanto spazio occupa il contenuto della propria home directory. Esiste un modo per ottenere in output soltanto il numero di blocchi (evitando di visualizzare informazioni ulteriori)?
- Qual è l'effetto del comando `sort file >file`, dove `file` è il nome di un file?
- Fare alcuni esperimenti per scoprire qual è l'effetto del comando `tr str1 str2` se le stringhe `str1` e `str2` hanno lunghezze diverse.
- Scrivere un comando per sostituire tutti i caratteri alfanumerici nell'input con un carattere <Tab>, in modo che non compaiano più <Tab> consecutivi.
- Il comando `date` fornisce data e ora su standard output. Scrivere una pipeline per estrarre soltanto i minuti.
- Scrivere una pipeline che permetta di scoprire se ci sono linee ripetute in un file.
- Visualizzare su standard output, senza ripetizioni, lo user ID di tutti gli utenti che hanno almeno un processo attivo nel sistema.

Corso di Laboratorio di Sistemi Operativi

Lezione 5

Ivan Scagnetto
ivan.scagnetto@uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Comandi filtro: `sed`

Il nome del comando `sed` sta per **S**tream **ED**itor e la sua funzione è quella di permettere di editare il testo passato da un comando ad un altro in una pipeline. Ciò è molto utile perché tale testo non risiede fisicamente in un file su disco e quindi non è editabile con un editor tradizionale (e.g. `vi`). Tuttavia `sed` può anche prendere in input dei file, quindi la sua sintassi è la seguente:

```
sed actions files
```

- Se non si specificano azioni, `sed` stampa sullo standard output le linee in input, lasciandole inalterate.
- Se vi è più di un'azione, esse possono essere specificate sulla riga di comando precedendo ognuna con l'opzione `-e`, oppure possono essere lette da un file esterno specificato sulla linea di comando con l'opzione `-f`.
- Se non viene specificato un **indirizzo** o un intervallo di indirizzi di linea su cui eseguire l'azione, quest'ultima viene applicata a tutte le linee in input. Gli indirizzi di linea si possono specificare come **numeri** o **espressioni regolari**.

Esempi d'uso di sed

- `> sed '4,$d' /etc/passwd`
  stampa a video soltanto le prime 3 righe del file `/etc/passwd`: `d` è il comando di cancellazione che elimina dall'output tutte le righe a partire dalla quarta (`$` sta per l'ultima riga del file).
- `> sed 3q /etc/passwd`
  stesso effetto del precedente comando: in questo caso `sed` esce dopo aver elaborato la terza riga (`3q`).
- `> sed /sh/y/:0/_%/ /etc/passwd`
  sostituisce in tutte le righe che contengono la stringa `sh` il carattere `:` con il carattere `_` ed il carattere `0` con il carattere `%`.
- `> sed '/sh/!y/:0/_%/' /etc/passwd`
  sostituisce in tutte le righe che **non** contengono la stringa `sh` il carattere `:` con il carattere `_` ed il carattere `0` con il carattere `%`. Si noti l'uso del quoting per impedire che la shell interpreti il metacarattere `!`.

Sostituzione del testo con `sed`

Il formato dell'azione di sostituzione in `sed` è il seguente:
`s/expr/new/flags`
dove:

- `expr` è l'espressione da cercare,
- `new` è la stringa da sostituire al posto di `expr`,
- `flags` è uno degli elementi seguenti:
  - *num*: un numero da 1 a 9 che specifica quale occorrenza di `expr` deve essere sostituita (di default è la prima),
  - `g`: ogni occorrenza di `expr` viene sostituita,
  - `p`: la linea corrente viene stampata sullo standard output nel caso vi sia stata una sostituzione (utile in congiunzione con l'opzione `-n` che esegue `sed` in **silent mode**, ovvero, senza emettere output);
  - `w` *file*: la linea corrente viene accodata nel file *file* nel caso vi sia stata una sostituzione.

Esempi di sostituzioni con sed

- `sed '/^root/,/^bin/s/:x:/::/w disabled.txt' /etc/passwd`
  sostituisce la password criptata (rappresentata dalla `x`) con la stringa vuota nelle righe in input comprese fra quella che inizia con `root` e quella che inizia con `bin`; tali righe sono poi accodate nel file `disabled.txt`.
- `cat /etc/passwd | sed 's?/bin/.*sh$?/usr/local&?'`
  cerca tutte le righe in input in cui compare la stringa corrispondente all'espressione regolare `/bin/.*sh$` (ad esempio `/bin/bash`) e sostituisce quest'ultima con la stringa corrispondente a `/usr/local/bin/.*sh$` (ad esempio `/usr/local/bin/bash`). Si noti che, siccome il carattere separatore di `sed` compare nella stringa da cercare, si è usato il carattere `?` come separatore. Inoltre il carattere `&` viene rimpiazzato automaticamente da `sed` con la stringa cercata (corrispondente a `/bin/.*sh$`).

Shell Script

Gli **shell script** sono **programmi** interpretati dalla shell, scritti in un linguaggio i cui costrutti atomici sono i **comandi Unix**. I comandi possono essere combinati in sequenza o mediante i costrutti usuali di un linguaggio di programmazione. La sintassi varia da shell a shell. Faremo riferimento alla shell `bash`.
Gli shell script sono la base degli *scripting languages*, come *Perl*.
Uno shell script va scritto in un file utilizzando per esempio il comando `cat` o un editor (`vi`, `emacs`, etc). Per poter eseguire lo script, il file deve essere reso **eseguibile**. Lo script viene eseguito invocando il nome del file.

Esempio

```
> cat >dirsize              # il file dirsize viene editato
ls /usr/bin | wc -w
Ctrl-d                      # fine dell'editing

> chmod 700 dirsize         # dirsize viene reso
                            # (in particolare) eseguibile

> ./dirsize                 # viene invocato il comando dirsize
   459                      # risultato dell'esecuzione
```

Esecuzione di script

Lo script viene eseguito in una **sottoshell** della shell corrente.
Il comando `set -v`/`set -x` fa sì che durante l'esecuzione di uno script la shell visualizzi i comandi nel momento in cui li legge/esegue (`set -` annulla l'effetto di `set -v`/`set -x`). Ciò è utile per il debugging.

```
> cat >data
set -x                                  # contenuto dello script data
echo the date today is:                 # ...
date                                    # ...
Ctrl-d                                  # fine dell'editing

> chmod u+x data
> ./data                                # lo script viene invocato
++ echo the date today is:              # ... la shell visualizza
the date today is:                      # i comandi mentre
++ date                                 # li esegue
Tue Oct 25 17:37:52 CEST 2005           # ...
```

... esecuzione di script

```
> cat >sost
set -v                                  # contenuto dello script sost
cd TEXT                                 # ...
ls *.txt                                # ...
sed s/'#'/';;;'/  file.txt              # ...
Ctrl-d                                  # fine dell'editing

> more TEXT/file.txt
# questo e‘ un commento
# in un programma

> chmod u+x sost
> ./sost                                # lo script viene invocato
cd TEXT                                 # ... la shell visualizza
ls *.txt                                # i comandi
file.txt                                # mentre
sed s/'#'/';;;'/ file.txt               # li legge
;;; questo e‘ un commento               # output del comando sed
;;; in un programma                    # ...
```

Variabili

Le **variabili** della shell sono stringhe di caratteri a cui è associato un certo spazio in memoria. Il **valore** di una variabile è una **stringa di caratteri**. Le variabili della shell possono essere utilizzate sia sulla linea di comando che negli script.
Non c'è dichiarazione esplicita delle variabili.
Assegnamento di una variabile (eventualmente nuova): `variabile=valore`
(Importante: non lasciare spazi a sinistra ed a destra dell'operatore =)

```
> x=variabile
> y='y e‘ una variabile'
```

Per **accedere** al valore di una variabile si utilizza il $:

```
> echo il valore di x: $x
il valore di x: variabile
> echo il valore di y: $y
il valore di y: y e‘ una variabile
> echo y
y
```

Le variabili sono **locali** alla shell o allo script in cui sono definite. Per rendere globale una variabile (**variabile d'ambiente**) si usa il comando `export`:

```
> export x          # promuove x a variabile di ambiente
```

Variabili di ambiente

Le **variabili di ambiente** sono variabili globali. Esiste un insieme di variabili di ambiente speciali riconosciute dalla shell e definite al momento del login:

VARIABILE	SIGNIFICATO
`PS1`	prompt della shell
`PS2`	secondo prompt della shell; utilizzato per esempio in caso di ridirezione dell'input dalla linea di comando
`PWD`	pathname assoluto della directory corrente
`UID`	ID dello user corrente
`PATH`	lista di pathname di directory in cui la shell cerca i comandi
`HOME`	pathanme assoluto della home directory

Esempi d'uso:

```
> echo il valore di PATH:  $PATH
il valore di PATH: /usr/bin:/usr/openwin/bin:/usr/local/bin
```

Le variabili di ambiente possono essere modificate:

```
> PS1='salve: '
salve:
> PATH=./bin:$PATH
> echo il valore di PATH:  $PATH
il valore di PATH: ./bin:/usr/bin:/usr/openwin/bin:/usr/local/bin
```

Sintassi del prompt: `http://ss64.com/bash/syntax-prompt.html`

Parametri

Le variabili `$1, $2, ..., $9` sono variabili speciali associate al primo, secondo, ..., nono parametro passato sulla linea di comando quando viene invocato uno script:

```
> cat >copia
mkdir $1
mv $2 $1/$2
Ctrl-d

> ./copia nuovadir testo
> ls nuovadir
testo
```

Se uno script ha più di 9 parametri, si utilizza il comando `shift` per fare lo shift a sinistra dei parametri e poter accedere ai parametri oltre il nono:

```
> cat >stampa_decimo
shift
echo decimo parametro: $9
Ctrl-d
> ./stampa_decimo 1 2 3 4 5 6 7 8 9 10
decimo parametro: 10
```

Variabili di stato automatiche (I)

Sono variabili speciali che servono per gestire lo **stato** e sono aggiornate **automaticamente** dalla shell. L'utente può accedervi solo in lettura. Al termine dell'esecuzione di ogni comando unix, viene restituito un valore di uscita, **exit status**, uguale a 0, se l'esecuzione è terminata con successo, diverso da 0, altrimenti (codice di errore).
La variabile speciale `$?` contiene il valore di uscita dell'ultimo comando eseguito.

```
> cd
> echo $?
0
> ./copia nuovadir testo
> echo $?
0
> ./copia nuovadir testo
mkdir: Failed to make directory "nuovadir"; File exists
mv: cannot access testo
> echo $?
2
```

Il comando `exit n`, dove `n` è un numero, usato all'interno di uno script, serve per terminare l'esecuzione e assegnare alla variabile di stato il valore `n`.

Variabili di stato automatiche (II)

La shell `bash` mette a disposizione numerose variabili di stato; le principali sono:

Variabile	**Contenuto**
`$?`	*exit status* dell'ultimo comando eseguito dalla shell
`$$`	`PID` della shell corrente
`$!`	il `PID` dell'ultimo comando eseguito in background
`$-`	le opzioni della shell corrente
`$#`	numero dei parametri forniti allo script sulla linea di comando
`$*`, `$@`	lista di tutti i parametri passati allo script sulla linea di comando

In particolare `$$` viene usata per generare nomi di file temporanei che siano unici fra utenti diversi e sessioni di shell diverse, e.g., `/tmp/tmp$$`.

Login script

Il **login script** è uno script speciale eseguito **automaticamente** al momento del login. In (alcune versioni di) Unix/Linux tale script è contenuto in uno dei file `.bash_profile`, `.bash_login`, `.bashrc`, `.profile`, memorizzati nella home directory degli utenti.
Il login script contiene alcuni comandi che è utile eseguire al momento del login, come la definizione di alcune variabili di ambiente.
Ciascun utente può modificare il proprio login script, ad esempio (ri)definendo variabili di ambiente e alias "permanenti".
Esiste anche uno script di login *globale* contenuto nel file `/etc/profile` in cui l'amministratore di sistema può memorizzare dei comandi di configurazione che valgano per tutti (tale script è infatti eseguito prima di quelli dei singoli utenti).
Lo script di logout eseguito al momento dell'uscita dalla shell (di login), si chiama solitamente `.bash_logout`.

Esercizi (I)

- Creare una sottodirectory `bin` all'interno della propria home directory in cui mettere gli script. Fare in modo che gli script contenuti in `bin` possano essere invocati da qualunque directory con il nome del file, senza dover specificare l'intero pathname.
- Qual è l'effetto della seguente sequenza di comandi? Perché?

```
> cat >chdir
cd ..
Ctrl-d
> chmod 700 chdir
> ./chdir
> pwd
```

- Creare un alias permanente `lo` per il comando `exit`.
- Progettare uno script che prende come parametro una stringa e un file di testo e controlla se la stringa compare nel file.

Esercizi (II)

- Il comando `read` assegna alla variabile speciale `REPLY` un testo acquisito da standard input. Qual è l'effetto dello script `words` contenente i seguenti comandi?

```
echo -n 'Enter some text: '
read one two restofline
echo 'The first word was: $one'
echo 'The second word was: $two'
echo 'The rest of the line was: $restofline'
exit 0
```

- Qual è l'effetto della seguente sequenza di comandi? Perché?

```
> cat >data
echo -n the date today is:
date
Ctrl-d
> chmod 700 data
> ./data
```

- Scrivere uno script che estragga soltanto i commenti dal file con estensione `java` fornito come primo argomento, sostituendo `//` con la stringa `linea di commento del file` <nome del file>`:`. Inoltre i commenti estratti devono essere salvati nel file fornito come secondo argomento.

Corso di Laboratorio di Sistemi Operativi

Lezione 6

Ivan Scagnetto
ivan.scagnetto@uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

Controllo di flusso negli script: if-then-else

Il comando condizionale

```
if condition_command
then
    true_commands
else
    false_commands
fi
```

esegue il comando `condition_command` e utilizza il suo **exit status** per decidere se eseguire i comandi `true_commands` (exit status 0) od i comandi `false_commands` (exit status diverso da zero).
Ad esempio lo script seguente prende come argomento un login name e stampa a video un messaggio diverso a seconda se il parametro fornito compaia all'inizio di una linea del file `/etc/passwd` oppure no:

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

exit 0
```

Condizioni: exit status e comando test (I)

Se la condizione che si vuole specificare non è esprimibile tramite l'exit status di un "normale" comando, si può utilizzare l'apposito comando `test`:
`test expression`
che restituisce un exit status pari a `0` se `expression` è vera, pari a `1` altrimenti. Si possono costruire vari tipi di espressioni:

- espressioni che controllano se un file possiede certi attributi:
  `-e` *f* restituisce vero se *f* esiste;
  `-f` *f* restituisce vero se *f* esiste ed è un file ordinario;
  `-d` *f* restituisce vero se *f* esiste ed è una directory;
  `-r` *f* restituisce vero se *f* esiste ed è leggibile dall'utente;
  `-w` *f* restituisce vero se *f* esiste ed è scrivibile dall'utente;
  `-x` *f* restituisce vero se *f* esiste ed è eseguibile dall'utente;
- espressioni su stringhe:
  `-z` *str* restituisce vero se *str* è di lunghezza zero;
  `-n` *str* restituisce vero se *str* non è di lunghezza zero;
  str1 `=` *str2* restituisce vero se *str1* è uguale a *str2*;
  str1 `!=` *str2* restituisce vero se *str1* è diversa da *str2*;

Condizioni: exit status e comando test (II)

- espressioni su valori numerici:
  num1 `-eq` *num2* restituisce vero se *num1* è uguale a *num2*;
  num1 `-ne` *num2* restituisce vero se *num1* non è uguale a *num2*;
  num1 `-lt` *num2* restituisce vero se *num1* è minore di *num2*;
  num1 `-gt` *num2* restituisce vero se *num1* è maggiore di *num2*;
  num1 `-le` *num2* restituisce vero se *num1* è minore o uguale a *num2*;
  num1 `-ge` *num2* restituisce vero se *num1* è maggiore o uguale a *num2*
- espressioni composte:
  exp1 `-a` *exp2* restituisce vero se sono vere sia *exp1* che *exp2*
  exp1 `-o` *exp2* restituisce vero se è vera *exp1* o *exp2*
  `!` *exp* restituisce vero se non è vera *exp*
  `(` *exp* `)` le parentesi possono essere usate per cambiare l'ordine di valutazione degli operatori (è necessario farne il quoting).

La shell fornisce anche la possibilità di costruire espressioni numeriche complesse, da utilizzare con il comando di test, tramite la sintassi seguente:
`$[expression]`
Ad esempio:

```
> num1=2
> num1=$[$num1*3+1]
> echo $num1
7
```

Controllo di flusso negli script: cicli while

Sintassi:

```
while condition_command
do
    commands
done
```

L'effetto risultante è che vengono eseguiti i comandi `commands` finché la condizione `condition_command` è vera. Esempio:

```
while test -e $1
do
    sleep 2
done

echo file $1 does not exist
exit 0
```

Lo script precedente esegue un ciclo che dura finché il file fornito come argomento non viene cancellato. Il comando che viene eseguito come corpo del `while` è una pausa di 2 secondi.

Controllo di flusso negli script: cicli until

Sintassi:

```
until condition_command
do
    commands
done
```

L'effetto risultante è che vengono eseguiti i comandi `commands` finché la condizione `condition_command` è **falsa**. Esempio:

```
until false
do
    read firstword restofline
    if test $firstword = end
    then
        exit 0
    else
        echo $firstword $restofline
    fi
done
```

Lo script precedente legge continuamente dallo standard input e visualizza quanto letto sullo standard output, finché l'utente non inserisce la stringa `end`.

Controllo di flusso negli script: cicli for

Sintassi:

```
for var in wordlist
do
    commands
done
```

L'effetto risultante è che vengono eseguiti i comandi `commands` per tutti gli elementi contenuti in `wordlist` (l'elemento corrente è memorizzato nella variabile `var`). Esempio:

```
for i in 1 2 3 4 5
do
    echo the value of i is $i
done


exit 0
```

L'output dello script precedente è:

```
the value of i is 1
the value of i is 2
the value of i is 3
the value of i is 4
the value of i is 5
```

Controllo di flusso negli script: case selection

Sintassi:

```
case string in
expression_1)
    commands_1
    ;;
expression_2)
    commands_2
    ;;
...
*)
    default_commands
    ;;
esac
```

L'effetto risultante è che vengono eseguiti i comandi `commands_1`, `commands_2`,... a seconda del fatto che `string` sia uguale a `expression_1`, `expression_2`,...
I comandi `default_commands` vengono eseguiti soltanto se il valore di `string` non coincide con nessuno fra `expression_1`, `expression_2`,...

I valori `expression_1`, `expression_2`,... possono essere specificati usando le solite regole per l'espansione del percorso (caratteri jolly).

Esempio d'uso del costrutto di case selection

Supponiamo di avere il seguente script memorizzato nel file `append`:

```
case $# in
1)
    cat >>$1
    ;;
2)
    cat >>$1 <$2
    ;;
*)
    echo "usage: append out_file [in_file]"
    ;;
esac

exit 0
```

Lo script precedente controlla che il numero degli argomenti forniti (variabile $#) sia 1 o 2 (a seconda se l'input da accodare al primo argomento debba provenire dallo standard input o da un altro file specificato sulla linea di comando), altrimenti stampa un messaggio che illustra l'utilizzo dello script.

Command substitution

Il meccanismo di **command substitution** permette di sostituire ad un comando o pipeline quanto stampato sullo standard output da quest'ultimo.
Esempi:

```
> date
Tue Nov 19 17:50:10  2002
> vardata=‘date‘
> echo $vardata
Tue Nov 19 17:51:28  2002
```

Un comando molto usato con le command susbstitution è `basename` (restituisce il nome di un file, senza il path):

```
> basefile=‘basename /usr/bin/man‘
> echo $basefile
man
```

Importante: per operare una command substitution si devono usare gli "apici rovesciati" o backquote (‘), non gli apici normali (’) che si usano come meccanismo di quoting.

Esempio (I)

Progettare uno script, chiamato `listfiles`, che prende due parametri, una directory e la dimensione di un file in byte. Lo script deve fornire il nome di tutti i file regolari contenuti nella directory parametro ai quali avete accesso e che sono più piccoli della dimensione data. Si controlli che i parametri passati sulla linea di comando siano due e che il primo sia una directory.
Esempio di soluzione (prima parte: controllo dei parametri):

```
if test $# -ne 2
then
    echo 'usage: listfiles <dirpath> <dimensione>'
    exit 1
fi
if ! test -d $1
then
    echo 'usage: listfiles <dirpath> <dimensione>'
    exit 1
fi
```

Esempio (II)

Esempio di soluzione (seconda parte: esecuzione del compito stabilito nell'esercizio):

```
for i in $1/*
do
    if test -r $i -a -f $i
    then
        size=‘wc -c <$i‘
        if test $size -lt $2
        then
            echo ‘basename $i‘ has size $size bytes
        fi
    fi
done

exit 0
```

Esercizi

- Progettare uno script che prende in input come parametri i nomi di due directory e copia tutti i file della prima nella seconda, trasformando tutte le occorrenze della stringa `SP` in `SU` in ogni file.
- Progettare uno script `drawsquare` che prende in input un parametro intero con valore da 2 a 15 e disegna sullo standard output un quadrato (utilizzando i caratteri +, - e |) come nel seguente esempio:

```
> ./drawsquare 4
+--+
|  |
|  |
+--+
```

- Progettare uno script che prende in input come parametro il nome di una directory e cancella tutti i file con nome `core` dall'albero di directory con radice la directory parametro.

Tabelle Riassuntive della shell `bash` in UNIX/Linux

15 ottobre 2019

1 Principali comandi e parole chiave della shell

Argomento	**Comandi visti a lezione**
Shell e sessione	`bash` (altre shell: `sh`, `csh`, `tcsh`, `ksh`), `logout`, `exit`, `history`, `alias`, `unalias`, `man`, `who`, `date`
File e directory	`pwd`, `cd`, `ls`, `mkdir`, `rmdir`, `cp`, `mv`, `rm`, `ln`, `chmod`, `touch`, `mount`, `df`, `du`, `basename`, `cmp`, `diff`, `find`
Processi e job	`ps`, `top`, `kill`, `jobs`, `fg`, `bg`
Visualizzazione	`echo`, `cat`, `more`, `less`, `tail`, `head`
Filtro	`wc`, `uniq`, `grep`, `fgrep`, `egrep`, `sort`, `tr`, `cut`, `paste`, `sed`
Editor	`nano`, `vi`, `emacs`, `xemacs`, `mc`
Script	`set`, `shift`, `if then else fi`, `while do done`, `until do done`, `for in do done`, `case in esac`, `test`, `read`

La filosofia alla base della shell UNIX/Linux è che i comandi devono essere programmi efficienti e di dimensione contenuta in grado di svolgere compiti semplici. Per l'esecuzione di compiti complessi i comandi vanno combinati tramite:

1. pipeline (metacarattere `|`);
2. script (vedi Sezione 5).

Per usare proficuamente la shell ed evitare di digitare molto testo, esistono i seguenti meccanismi:

1. caratteri jolly (metacaratteri `?` e `*`);
2. command completion (tasto `<Tab>`);
3. command line editing della shell (`Ctrl-A`, `Ctrl-E`, tasti cursore, tasto Backspace ecc.);
4. history e ripetizione comandi (metacarattere `!`).

La prima fonte di informazione sulla shell e sui comandi della shell è il manuale online installato sui vari sistemi UNIX/Linux, accessibile tramite il comando `man`. In particolare la sezione 1 del manuale è interamente dedicata a questo scopo.

2 Metacaratteri della shell

Simbolo	Significato	Esempio d'uso
`>`	Ridirezione dell'output	`ls >temp`
`>>`	Ridirezione dell'output (append)	`ls >>temp`
`<`	Ridirezione dell'input	`wc -l <text`
`<<delim`	ridirezione dell'input da linea di comando (here document)	`wc -l <<delim`
`*`	Wildcard: stringa di 0 o più caratteri, ad eccezione del punto (`.`)	`ls *.c`
`?`	Wildcard: un singolo carattere, ad eccezione del punto (`.`)	`ls ?.c`
`[...]`	Wildcard: un singolo carattere tra quelli elencati	`ls [a-zA-Z].bak`
`{...}`	Wildcard: le stringhe specificate all'interno delle parentesi	`ls {prog,doc}*.txt`
`|`	Pipe	`ls | more`
`;`	Sequenza di comandi	`pwd;ls;cd`
`||`	Esecuzione condizionale. Esegue un comando se il precedente fallisce.	`cc prog.c || echo errore`
`&&`	Esecuzione condizionale. Esegue un comando se il precedente termina con successo.	`cc prog.c && a.out`
`(...)`	Raggruppamento di comandi	`(date;ls;pwd)>out.txt`
`#`	Introduce un commento	`ls # lista di file`
`\`	Fa in modo che la shell non interpreti in modo speciale il carattere che segue.	`ls file.\*`
`!`	Ripetizione di comandi memorizzati nell'history list	`!ls`

3 Metacaratteri delle espressioni regolari

metacarattere	tipo	significato
`^`	B	inizio della linea
`$`	B	fine della linea
`\<`	B	inzio di una parola
`\>`	B	fine di una parola
`.`	B	un singolo carattere (qualsiasi)
`[str]`	B	un qualunque carattere in `str`
`[^str]`	B	un qualunque carattere non in `str`
`[a-z]`	B	un qualunque carattere tra `a` e `z`
`\`	B	inibisce l'interpretazione del metacarattere che segue
`*`	B	zero o più ripetizioni dell'elemento precedente
`+`	E	una o più ripetizioni dell'elemento precedente
`?`	E	zero od una ripetizione dell'elemento precedente
`{j,k}`	E	un numero di ripetizioni compreso tra `j` e `k` dell'elemento precedente
`s|t`	E	l'elemento `s` oppure l'elemento `t`
`(exp)`	E	raggruppamento di `exp` come singolo elemento

Nella tabella precedente B (basic) indica che la sequenza di caratteri è utilizzabile sia in `grep` che in `egrep`, mentre E (extended) indica che la sequenza di caratteri è utilizzabile solo in `egrep` (o in `grep` usando l'opzione `-E`).

4 Variabili di stato automatiche

La shell `bash` mette a disposizione numerose variabili di stato; le principali sono:

Variabile	Contenuto
`$?`	*exit status* dell'ultimo comando eseguito dalla shell
`$$`	`PID` della shell corrente
`$!`	il `PID` dell'ultimo comando eseguito in background
`$-`	le opzioni della shell corrente
`$#`	numero dei parametri forniti allo script sulla linea di comando
`$*`, `$@`	lista di tutti i parametri passati allo script sulla linea di comando

In particolare `$$` viene usata per generare nomi di file temporanei che siano unici fra utenti diversi e sessioni di shell diverse, e.g., `/tmp/tmp$$`.

5 Sintassi degli Script

5.1 Variabili ed assegnamento

Le variabili della shell sono stringhe di caratteri a cui è associato un certo spazio in memoria. Il valore di una variabile per la shell è sempre una stringa di caratteri (a meno di non forzare un'interpretazione diversa: si veda, e.g., la Sezione 5.1.4). Le variabili della shell possono essere utilizzate sia sulla linea di comando che negli script. Non c'è dichiarazione esplicita delle variabili (il primo utilizzo di una variabile la dichiara implicitamente).

5.1.1 Assegnamento di una variabile

La sintassi è la seguente: `variabile=valore` (importante: non lasciare spazi a sinistra ed a destra dell'operatore `=` per non confonderlo con l'operatore di confronto). Esempio:

```
> x=valore
```

5.1.2 Utilizzo di una variabile

Per accedere al valore di una variabile si utilizza il `$`:

```
> echo il valore di x e': $x
il valore di x: valore
```

5.1.3 Il comando `export`

Le variabili sono **locali** alla shell o allo script in cui sono definite. Per rendere globale una variabile (*variabile d'ambiente*) si usa il comando `export`:

```
> export x        # promuove x a variabile di ambiente
```

5.1.4 Aritmetica con le variabili della shell

La shell fornisce anche la possibilità di costruire espressioni numeriche complesse, da utilizzare con il comando di test, tramite la sintassi seguente:
`$[expression]`
Ad esempio:

```
> num1=2
> num1=$[$num1*3+1]
> echo $num1
7
```

Esiste anche un'altra sintassi per lo stesso scopo:

```
> num1=2
> num1=$(($num1*3+1))
> echo $num1
7
```

5.2 Controllo di flusso

Nel seguito verranno riportati brevemente i vari comandi per alterare il normale flusso di esecuzione sequenziale degli script. Essi sono analoghi ai vari costrutti sintattici disponibili nei comuni linguaggi imperativi (C, Java ecc.); tuttavia l'espressione che funge da *guardia* (denotata da `condition_command` nel seguito) può essere un qualunque comando UNIX/Linux che restituisca un exit status numerico (con la convenzione che `0` significhi *vero*, mentre un qualunque valore diverso da zero significhi *falso*).

5.2.1 Comando condizionale (`if`)

La sintassi è la seguente (il ramo `else` è opzionale)

```
if condition_command
then
    true_commands
else
    false_commands
fi
```

Se `condition_command` restituisce come exit status zero, allora vengono eseguiti i `true_commands`, altrimenti vengono eseguiti i `false_commands`.

5.2.2 Iterazione indeterminata (`while`)

La sintassi è la seguente:

```
while condition_command
do
    commands
done
```

L'effetto risultante è che vengono eseguiti i comandi `commands` finché `condition_command` restituisce zero come exit status.

5.2.3 Iterazione indeterminata (`until`)

La sintassi è la seguente:

```
until condition_command
do
    commands
done
```

L'effetto risultante è che vengono eseguiti i comandi `commands` finché `condition_command` restituisce un valore diverso da zero come exit status.

5.2.4 Iterazione determinata (`for`)

La sintassi è la seguente:

```
for var in wordlist
do
    commands
done
```

I comandi `commands` vengono ripetuti un numero di volte pari alla lunghezza di `wordlist`, istanziando la variabile `var` ad un valore di una delle componenti di `wordlist` ad ogni iterazione.

Esempio:

```
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo $i
done
```

Sintassi alternativa (simile a C, C++, Java ecc.):

```
for ((i=1; i<=10; i++))
do
    echo $i
done
```

5.2.5 Selezione (`case`)

La sintassi è la seguente:

```
case string in
expression_1)
    commands_1
    ;;
expression_2)
    commands_2
    ;;
...
*)
    default_commands
    ;;
esac
```

L'effetto risultante è che vengono eseguiti i comandi `commands_1`, `commands_2`,... a seconda del fatto che `string` sia uguale a `expression_1`, `expression_2`,...
I comandi `default_commands` vengono eseguiti soltanto se il valore di `string` non coincide con nessuno fra `expression_1`, `expression_2`,...
I valori `expression_1`, `expression_2`,... possono essere specificati usando le solite regole per l'espansione del percorso (caratteri jolly).

5.3 Comando `test`

Se la condizione che si vuole specificare non è esprimibile tramite l'exit status di un "normale" comando, si può utilizzare l'apposito comando `test`:
`test expression`
che restituisce un exit status pari a `0` se `expression` è vera, pari a `1` altrimenti.
Si possono costruire vari tipi di espressioni:

- espressioni che controllano se un file possiede certi attributi:
  `-e` *f* restituisce vero se *f* esiste;
  `-f` *f* restituisce vero se *f* è un file ordinario;
  `-d` *f* restituisce vero se *f* è una directory;
  `-r` *f* restituisce vero se *f* è leggibile dall'utente;
  `-w` *f* restituisce vero se *f* è scrivibile dall'utente;
  `-x` *f* restituisce vero se *f* è eseguibile dall'utente;
- espressioni su stringhe:
  `-z` *str* restituisce vero se *str* è di lunghezza zero;
  `-n` *str* restituisce vero se *str* non è di lunghezza zero;
  str1 `=` *str2* restituisce vero se *str1* è uguale a *str2*;
  str1 `!=` *str2* restituisce vero se *str1* è diversa da *str2*;
- espressioni su valori numerici:
  num1 `-eq` *num2* restituisce vero se *num1* è uguale a *num2*;
  num1 `-ne` *num2* restituisce vero se *num1* non è uguale a *num2*;
  num1 `-lt` *num2* restituisce vero se *num1* è minore di *num2*;
  num1 `-gt` *num2* restituisce vero se *num1* è maggiore di *num2*;
  num1 `-le` *num2* restituisce vero se *num1* è minore o uguale a *num2*;
  num1 `-ge` *num2* restituisce vero se *num1* è maggiore o uguale a *num2*
- espressioni composte:
  exp1 `-a` *exp2* restituisce vero se sono vere sia *exp1* che *exp2*
  exp1 `-o` *exp2* restituisce vero se è vera *exp1* o *exp2*
  `!` *exp* restituisce vero se non è vera *exp*

Invece di `test expression` si può utilizzare la sintassi alternativa `[ expression ]`, facendo attenzione a lasciare uno spazio dopo `[` e prima di `]`.

6 Editor

6.1 Modalità e comandi principali dell'editor `vi`

6.1.1 Edit mode

La modalità di edit è usata principalmente per muovere il cursore nel punto di interesse all'interno del file di testo che si sta editando (è anche la modalità di

default quando si avvia `vi` e ci si può sempre ritornare, nel caso ci si trovi in un'altra modalità, premendo il tasto `Esc`).

Comando	Effetto
`k`, `j`, `h`, `l` (od i tasti cursore)	muove il cursore su, giù, a sinistra ed a destra
`Ctrl-f`, `Ctrl-b`	muove il cursore avanti/indietro di una pagina
`H`, `M`, `L`	muove il cursore alla prima riga, all'ultima od a quella nel mezzo dello schermo
`w`	muove il cursore all'inizio della parola successiva
`e`	muove il cursore alla fine della parola successiva
`b`	muove il cursore all'inizio della parola precedente
`0`	muove il cursore all'inizio della linea corrente
`^`	muove il cursore nella posizione del primo carattere della linea che non sia un `whitespace`
`$`	muove il cursore alla fine della linea corrente
`/`*string*	cerca nel file la stringa *string*
`?`*string*	cerca "all'indietro" nel file la stringa *string*
`n`	cerca l'occorrenza della stringa successiva (in avanti o all'indietro)
n`r`*c*	rimpiazza *n* caratteri con *c* a partire dalla posizione del cursore
n`x`	cancella *n* caratteri dalla posizione del cursore
n`dw`	cancella *n* parole dalla posizione del cursore
n`db`	cancella *n* parole prima del cursore
n`dd`	cancella *n* linee a partire da quella del cursore
`d$`	cancella tutti i caratteri dalla posizione del cursore fino alla fine della linea
`d0`	cancella tutti i caratteri dalla posizione del cursore fino all'inizio della linea
`J`	unisce la linea corrente alla successiva
`p`	incolla il testo copiato/cancellato a destra del cursore
`P`	incolla il testo copiato/cancellato a sinistra del cursore
`yy`	copia la riga corrente in memoria
n`yy`	copia *n* righe in memoria a partire dalla posizione del cursore
`u`	annulla l'ultimo comando
`.`	ripete l'ultimo comando
`ZZ`	termina l'esecuzione di `vi`, salvando le modifiche

6.1.2 Insert mode

Siccome l'edit mode utilizza un gran numero di tasti alfanumerici, per inserire del testo in un file si rende necessaria un'altra modalità: l'**insert** mode. Per uscire dalla modalità di inserimento basta premere il tasto `Esc` o `Ctrl-[` nei terminali senza tasto `Esc`.

Comando	Effetto
`i`	inserisce del testo alla sinistra del cursore
`a`	inserisce del testo alla destra del cursore
`I`	inserisce del testo all'inizio della linea corrente
`A`	inserisce del testo alla fine della linea corrente
`o`	inserisce una nuova linea sotto la posizione del cursore
`O`	inserisce una nuova linea sopra la posizione del cursore

6.1.3 Command mode

Tutti i comandi del **command** mode iniziano con i due punti (`:`); dopo aver inserito tale carattere il cursore si sposta nell'ultima riga dello schermo dove compaiono i caratteri del comando successivamente digitati. La pressione del tasto invio provoca l'esecuzione del comando.

Comando	Effetto
`:q`	termina `vi` se non vi sono delle modifiche non salvate
`:q!`	termina `vi` perdendo le eventuali modifiche non salvate
`:w`	salva il file originale
`:wq`	salva il file originale e termina `vi` (stesso effetto di `ZZ`)
`:w` *file*	salva il contenuto nel file *file*
`:r` *file*	legge il contenuto del file *file* inserendolo dopo la posizione del cursore
`:e` *file*	edita il file *file*, sostituendo il contenuto corrente
`:f` *file*	cambia il nome del contenuto corrente in *file*
`:f`	stampa il nome e lo stato del testo corrente
`:`*n*	sposta il cursore alla linea *n*
`:/`*str*`/`	sposta il cursore alla prossima linea contenente *str*
`:s/`*str1*`/`*str2*`/`	sostituisce la prima occorrenza di *str1* sulla linea con *str2*
`:set` *option*	definisce un'opzione, e.g. `:set` *number* aggiunge i numeri di linea

6.2 Emacs: il minibuffer ed alcuni comandi utili

Sotto la mode line si trova il **minibuffer**, che è una parte dell'interfaccia di Emacs che si occupa di visualizzare e di permettere all'utente di editare/completare i comandi.

Comando	Effetto
`M-x text-mode`	cambia il major mode in text (modalità testo inglese)
`C-h m`	informazioni sul modo corrente
`C-x C-f` *file*	apre il file *file*
`C-x C-s`	salva il buffer corrente
`C-<Spacebar>`	imposta il mark (inizio di una regione di testo su cui eseguire un comando in seguito)
`M-w`	copia la regione di testo compresa fra il mark e la posizione del cursore
`C-w`	taglia la regione di testo compresa fra il mark e la posizione del cursore
`C-y`	incolla la regione di testo copiata/tagliata in precedenza
`C-x u`	undo
`C-x C-c`	termina Emacs

6.2.1 Comandi essenziali di `mc` (Midnight Commander)

Midnight Commander, che si esegue con `mc`, è un browser per l'esplorazione del filesystem da console, che permette anche di visualizzare e modificare i file.

Pur non utilizzando X, fornisce una interfaccia più user-friendly rispetto ad altri editor e permette di:

- navigare tra file e cartelle utilizzando le frecce e il tasto `Invio`;
- visualizzare un file con `F3`;
- modificare un file con `F4`;
- salvare un file con `F2` (in edit mode);
- uscire dall'editor e da `mc` con `F10`.