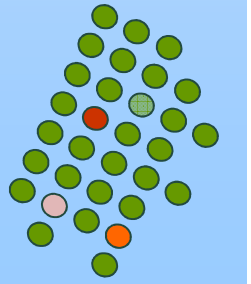
Elementi introduttivi al sistema operativo UNIX (LINUX)

Capitolo 1,2 -- Stevens



Standard Unix
&
Implementazioni

Standard Unix: ANSI

- American National Standards Institute: venditori + utenti
- Membro dell' International Organization for Standardization
- 1989: ANSI C per la standardizzazione del linguaggio C
  - portabilità di programmi C ad una grande varietà di S.O. e non solo per UNIX
  - Definisce non solo la semantica e la sintassi ma anche una libreria standard divisa in 15 aree (individuate dagli header, vedi fig. 2.1)

Standard Unix: IEEE POSIX

- Institute of Electrical and Electronic Engineers propose una famiglia di standard
  - Portable Operating System Interface
  - Lo standard 1003.1 relativo a interfacce di s.o.: definizione di servizi che un s.o. deve fornire per essere POSIX COMPLIANT
    - Definisce una interfaccia non una implementazione
    - Non è fatta distinzione tra system call e funzioni di libreria
    - Non prevede la figura di “superuser”, ma certe operazioni richiedono appropriati privilegi

Standard Unix: XPG3

- X/Open: gruppo di venditori di Computer
- Hanno prodotto 7 volumi di una guida di portabilità
- X/Open Portability Guide, Issue 3 1989
- Il II vol. definisce interfacce per un s.o. Unix-like, a partire da IEEE 1003.1, ma con variazioni (e.g. msg in varie lingue)

Implementazioni: SVR4

- System V Release 4 è stato prodotto dalla AT&T
- SVR4 è conforme a POSIX e XPG3

Implementazioni: 4.3+BSD

- Berkeley Software Distributions (sono distribuite da UCB)
- Conforme allo standard POSIX
- Benchè fosse inizialmente legato a codice sorgente AT&T e quindi alle sue licenze, è stata creata una versione (free) molto interessante per PC (intel based) FreeBSD

Implementazioni: Linux

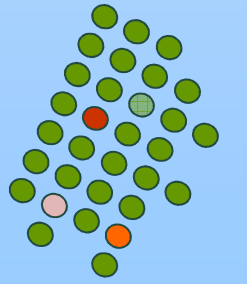
- Il primo kernel sviluppato da Linus Torvalds nel 1991
- Conforme a POSIX, ma include anche la maggior parte di funzioni di SVR4 e 4.3BSD
- Disponibile su Intel, Compaq (ex Digital), Alpha, Sparc, McIntosh e Amiga
- Free

Distribuzioni Linux

- Slackware
- Debian
- RedHat
- SuSe
- Mandrake

Introduzione a
Bash (Bourne Again Shell)

Capitolo 1 -- Rosenblatt



Introduzione

- Shell: Interfaccia con il sistema operativo
- Interfaccia testuale (command line - CLI)
- E’ un programma che interpreta comandi
- Viene eseguito quando facciamo login
- Termina al logout
- Bash e’ standard con Linux
- Altre shell: sh, tcsh, ksh

Introduzione

- L’indipendenza della shell dal sistema operativo Unix ha portato ad una grande varietà si shell
  - sh (include reminiscenze di ALGOL)
  - csh (usa una sintassi simile al C, prende comandi interattivi o programmi)
  - tcsh (come la csh + permette di viaggiare su e giù per la lista dei comandi dati, spelling correction dei comandi)
  - bash

Bash: Introduzione

- Sviluppata, a partire dal 1988, nel progetto GNU
- progetto GNU:
  - sviluppare una versione gratuita di UNIX
  - GNU = Gnu's Not Unix (acronimo ricorsivo)
  - open software, FSF
  - “copyleft”: software sotto “copyleft” e’ distribuito gratis con il codice sorgente e deve essere mantenuto tale
    - non si puo' vendere software preso gratis
- Steve Bourne ha scritto una delle prime shell per UNIX (1979): sh
- Bash: Bourne Again Shell, in tributo a S. Bourne

Bash: introduzione

```
login: rescigno
passwd:
Welcome to the system.
bash> echo "la shell e' $SHELL, versione $BASH_VERSION"
la shell e' /usr/local/bin/bash, versione 2.05.0(1)-release
bash> logout
Good bye.
login:
```

- Digitiamo il comando su una linea
- La shell intrerpreta la linea di comandi digitata
- ... esegue i comandi
- L'output del comando viene visualizzato su video

Bash: introduzione

```
bash> sort –n num_telefoni > num_tel.ordinati
```

Il compito della shell:

1. separare i token del comando:
   - sort, -n, num_telefoni, >, num_tel.ordinati
2. Determinare il significato dei token
   1. sort comando da eseguire
   2. -n e num_telefoni argomenti
   3. > operatore di ridirezione
   4. num_tel.ordinati nome del file per la ridirezione
3. Preparare il sistema in modo tale che l'output vada nel file num_tel.ordinati
4. Cercare ed eseguire il comando sort

Comandi, argomenti e opzioni

```
bash> lp –d lp1 –h file1 file2 file3
```

- linea comandi: parole (stringhe) separate da spazi o TAB
- La prima parola e' il comando
- Il resto sono gli argomenti
- Gli argomenti sono spesso nomi di file, ma non necessariamente:

```
mail rescigno
```

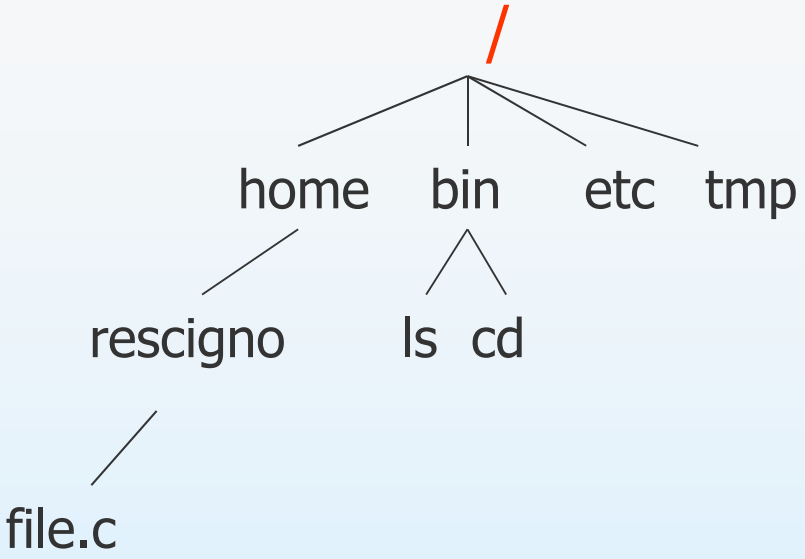
- Una opzione e' un argomento speciale che impartisce istruzioni al comando, cioe' modifica il comportamento di default
- A volte un'opzione ha un proprio argomento

File e directory

- Un file può contenere qualunque tipo di informazione ed esistono file di differenti tipi
  - File regolari = contengono caratteri leggibili
  - File eseguibili = sono invocati come comandi
  - Directory = sono dei contenitori in cui sono presenti altri file od anche altre sottodirectory

File e directory

- Filesytem gerarchico
  - root /
- Nomi di file speciali:
  - dot .
  - dot dot ..



- Pathname: sequenza di zero o più nomi di file separati da /
  - pathname assoluto: pathname che descrive la posizione di un file nel sistema a partire dalla root

    Es: /home/rescigno/file.c
  - pathname relativo: pathname che descrive la posizione di un file nel sistema a partire dalla directory in cui siamo

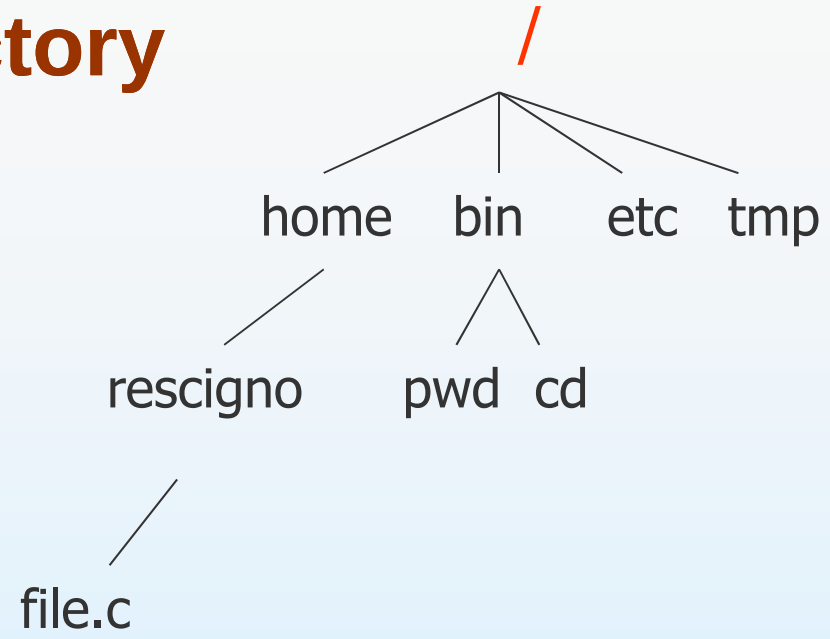
    Es: rescigno/file.c

Directory

- current working directory (cwd)
  - al login, cwd = home directory
- Conoscere la cwd: pwd
- Cambiare directory di lavoro: cd
- Tilde:
  - ~: home directory
  - ~user: home directory dell'utente "user"

```
[/home/user/rescigno]> cd pippo
[/home/user/rescigno/pippo]> cd –
[/home/user/rescigno]> cd /usr/lib
[/usr/lib]> cd ~
[/home/user/rescigno]> cd ~giuper
[/home/user/giuper]> cd
[/home/user/rescigno]> _
```

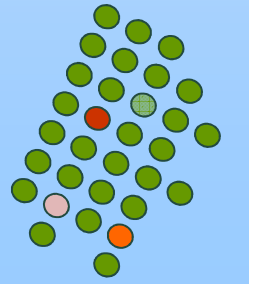

Directory



- Per avere l'elenco dei file presenti nella cwd : ls
  - Se si vuole conoscere l'elenco dei file presenti in una specifica directory, si fa succedere ls dalla pathname della directory

```
[/]> ls
        home  bin  etc  tmp
[/]> ls /bin
        pwd cd
[/]> cd bin
[/bin]> ls
       pwd cd
```

UNIX / LINUX ed il linguaggio C



Tipi di dati primitivi

- Nel file header <sys/types.h> (come anche in altri header) sono definiti alcuni tipi di dati system-dependent chiamati tipi di dati primitivi
  - Sono definiti utilizzando typedef
  - I loro nomi finiscono in genere con _t
- La tabella 2.8 mostra i principali tipi di dati primitivi che useremo
- Esercizio:
  - in Linux dove sono definiti?
    - /usr/include/sys

gcc: il compilatore C sotto Linux

- Supponiamo di aver scritto un programma C: prova.c
- Per compilarlo scriviamo

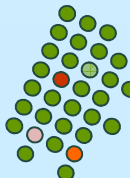
  gcc prova.c
- Questo produrrà nella directory corrente un file eseguibile

  a.out
- Se si vuole dare un nome specifico all'eseguibile si utilizza l'opzione –o

  gcc prova.c –o prova
- Questo produrrà nella directory corrente un file eseguibile

  prova
- Per mandare in esecuzione un file eseguibile

  prova oppure ./prova



gdb: il debugger C sotto Linux

Per illustrare l'utilizzo ed il funzionamento del debugger è comodo utilizzare un programma di esempio, con alcuni errori, e mostrare i vari comandi del debugger tramite i quali si può riuscire a capire l'errore.

Consideriamo uguali.c un programma che, date due stringhe per argomento, controlla se queste sono uguali.

- Perché il debugger possa funzionare, è necessario compilare con l'opzione *–g*:

  gcc –g uguali.c –o uguali

- questa opzione istruisce il compilatore ad inserire all'interno dell'eseguibile una serie di informazioni addizionali necessari al debugger per maneggiare l' eseguibile uguali, che potrà essere "debuggato".

gdb: comandi di base

- lanciamo l'eseguibile con due stringhe uguali, si ha:

```
/> uguali ciao ciao
/> DIVERSI
```

- subito ci accorgiamo che qualcosa non va e lanciamo il debugger con il nome dell'eseguibile

```
/> gdb  uguali
```

- A questo punto si entra nella shell del debugger che viene comandato inserendo direttamente i comandi

```
/> gdb uguali
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 Free Software Foundation, Inc.
...
(gdb)
```

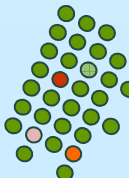

gdb: comandi di base

- Una volta lanciato il gdb mostra il suo prompt: (gdb)
- la (numerosa) lista dei comandi può essere ottenuta tramite il comando help (specificando successivamente un sotto argomento)
- run, seguito dagli argomenti che intendiamo passare; in questo modo però il programma viene eseguito senza interruzioni, non dandoci la possibilità di vedere ciò che accade

```
(gdb) run  ciao ciao
DIVERSI
```

- breakpoint, break seguito dal numero di riga del programma o dal nome di una funzione, inserisce un *breakpoint*, ovvero un particolare punto del programma che, quando incontrato dal gdb, mette in pausa l'esecuzione del programma consentendo di inserire nuovamente comandi dalla command line sulla funzione main. Un breakpoint è un modo di segnalare al debugger di fermare l'esecuzione quando il programma arriva in quel punto

```
(gdb) break  main
Breakpoint 1 at 0x8048643: file uguali.c, line 19.
```

gdb: comandi di base

- Possiamo ora lanciare l'eseguibile.
  - il debugger si blocca arrivato alla funzione main, mostrando i suoi argomenti, e la prossima riga di programma che sarà eseguita.

```
(gdb) run ciao ciao
Starting program: /home/lorenzo/articoli/uguali ciao
ciao

Breakpoint 1, main (argc=3, argv=0xbffff834) at
uguali.c:19
19 if ( uguali( argv[0], argv[1] ) )
```

- A questo punto il programma è fermo, ed il debugger è in attesa di un comando. I principali comandi
  - n, next - per avanzare alla successiva linea di programma senza entrare in sotto-funzioni
  - s, step - per avanzare alla successiva linea di programma entrando nelle sotto-funzioni
  - p, print - per visualizzare o assegnare il contenuto di una variabile
  - l, list - per visualizzare il codice sorgente del programma sotto esame
  - finish - per completare l'esecuzione del programma sino alla fine della funzione corrente
  - c, continue - per continuare l'esecuzione del programma fino al prossimo breakpoint
  - quit - per uscire dal gdb.

gdb: comandi di base

```
(gdb) s
ugualì (s1=0xbffff98b "/home/lorenzo/articoli/ugualì",
s2=0xbffff9a9 "ciao")
at ugualì.c:8
8 l1 = strlen( s1 ) ; l2 = strlen( s2 ) ;
```

- L’errore a questo punto è evidente: s1 ed s2 sono diversi; s1 non corrisponde con quello che volevamo fosse; in effetti gli argomenti veri e propri passati ad un programma iniziano dall'indice 1 dell'array *argv*
- quindi abbiamo capito l'errore e quindi possiamo modificare il programma (magari da un altro terminale): if (ugualì(argv[1], argv[2]))
- Ricompilando e facendo girare il programma vediamo che esso gira correttamente in questo caso
- Rimanendo con un terminale con il debugger aperto, proviamo a lanciare su un altro terminale

```
/> ugualì ciao miao
/> UGUALI
```


gdb: comandi di base

```
/> uguali ciao miao
/> UGUALI
```

- A questo punto pensiamo che l'errore sia nella funzione uguali, quindi togliamo il breakpoint dal main e lo mettiamo sulla funzione uguali e facciamo girare con run

```
(gdb) delete 1
(gdb) break uguali
Breakpoint 2 at 0x80485a6: file uguali.c, line 8.
(gdb) run ciao miao
The program being debugged has been started already.
Start it from the beginning? (y or n) y
...`/home/lorenzo/articoli/uguali' has changed; re-
reading symbols.
Starting program: /home/lorenzo/articoli/uguali ciao miao


Breakpoint 2, uguali (s1=0xbffff9a9 "ciao", s2=0xbffff9ae
"miao") at uguali.c:8
8 l1 = strlen( s1 ) ; l2 = strlen( s2 ) ;
```


gdb: il debugger C sotto Linux (cont.)

- Eseguiamo ora il programma passo dopo passo con n

```
(gdb) n
9 if ( l1 != l2 ) return 0 ;
(gdb) n
11 for ( i = 1 ; i < l1 ; ++i )
(gdb) n
12 if ( s1[i] != s2[i] )
(gdb) n
11 for ( i = 1 ; i < l1 ; ++i )
```

- il test sulla prima lettera ('c' e 'm') sarebbe dovuto fallire ed invece ha avuto successo; possiamo visualizzare il contenuto di s1[i] col comando print

```
(gdb) print s1[i]
s1[i] = 105 'i’
```

- a questo punto ci ricordiamo che il primo carattere in una stringa (vettore di caratteri) ha indice 0, e non 1: quindi dobbiamo far partire il ciclo for da 0 e non da 1.

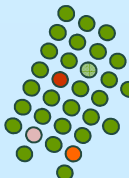
gdb: il debugger C sotto Linux (cont.)

- a questo punto ci ricordiamo che il primo carattere in una stringa (vettore di caratteri) ha indice 0, e non 1: quindi dobbiamo far partire il ciclo for da 0 e non da 1.

```
(gdb) display s2[i]
1: s2[i] = 105 'i'
(gdb) n
12 if ( s1[i] != s2[i] )
1: s2[i] = 97 'a'
```

- A questo punto il nostro programma funziona a dovere e possiamo uscire dal debugger con quit.

```
(gdb) quit
```

gdb: analizzare un file di core

- Quando un programma crash di solito termina con l'indicazione

"Segmentation Fault (core dumped)"

e viene generato il file *core*. Questi non è altro che una fotografia della memoria usata dal programma al momento in cui è stato terminato. Utilizzando gdb possiamo analizzare il file di core per localizzare l'errore, per far cio' si avvia gdb con il nome dell'eseguibile, che in questo esempio chiamiamo crash, e core

```
gdb crash core
```

- Lanciato il gdb appare subito l'indirizzo di memoria dove è avvenuto l'errore

```
#0 0x40054b03 in ?? () from /lib/libc.so.6
```

- per capire quale funzione l'ha causato viene usato il comando bt (backtrace) che stampa lo stack delle chiamate per raggiungere quel punto

```
(gdb) bt
#0 0x40054b03 in ?? () from /lib/libc.so.6
#1 0x80484fc in copia (s=0xbffffc87 "crash") at crash.c:6
#2 0x804852e in main (argc=1, argv=0xbffffba0) at
crash.c:14
```

- In questo modo è facile capire che l'errore è avvenuto al momento in cui è stata eseguita l'istruzione alla linea numero 6 del programma crash.c