Appunti di Linux

14 gennaio 2009

Indice

1 Comandi utili

- *ls* Mostra la lista dei file e delle cartelle nella directory corrente
- *ls -l* Mostra una lista più dettagliata
- *touch* nomefile : crea un nuovo file chiamato nomefile
- *ls* >z Mette l'outuput del comando dentro un nuovo file z
- *ls* <z Prende l'input del comando da un file z
- *cut -f 1 -d ' '* Taglia il primo campo (field -f 1) rispetto al delimitatore ' '

2 I filtri di Unix

Unix mette a disposizione tanti filtri utili (molti saranno discussi dopo) da usare con la tecnica del pipelining. Eccone alcuni:

- *sort* Ordina un qualsiasi flusso in input (sia esso un file o l'output di less)
- *grep,awk* Discussi in seguito.
- *uniq* Filtra le linee ripetute in un file.

3 I Processi in Linux

Ogni processo in Linux nasce con tre canali: stdin, stdout ed stderr. Il primo serve per gestire gli input da tastiera, i secondi due per l'output e gli errori, che sono visualizzati presso il monitor del TTY corrente. Per ogni processo che è in esecuzione in Linux posso inviare dei *signals* quali:

- STOP: Voglio fermare il mio processo. *CTRL + Z*
- TERM: Voglio terminare il mio processo. *CTRL + C*

3.1 Processi in background,foreground o attivi

Bene, abbiamo visto come interrompere o terminare un processo. Ma se vogliamo fare il resume dell' ultimo processo fermato? Basta scrivere *fg* che sta per foreground (in sovraimpressione). Se invece voglio resumare quel processo ma voglio che sia fatto in background allora basta digitare *bg % id-processo*. Se invece voglio lanciare un processo direttamente in background senza passare per il bg basta invocarlo con l'opzione *&* Altro comando utile è *jobs* che fornisce la lista di tutti i job su quel tty.

3.2 La lista dei processi

I processi attivi sul calcolatore si posso mostrare con il comando *ps aux* alias di processes, mostra tutti i processi attivi. Se nel campo STAT c'è la S vuol dire che quel processo è in Sleeping, mentre se c'è la R vuol dire che è in running. *Tips:* Per leggere più comodamente mando l'output a less col comando pipe: *ps -lax | less* dove quel -lax mi da una versione più dettagliata. Altre varianti:

pstree mostra i processi con struttura ad albero.

pstree -p mostra i processi con struttura ad albero compresi i pid di ogni processo.

3.2.1 Uccidere i processi

Si hanno due possibilità:

- *kill* pid uccide il processo con quel pid.
- *kill -9* uccide il processo in modo perentorio.

3.3 La tecnica del pipelining

E' una tecnica molto ultile che permette di attaccare il file stdout di un processo al file stdin di un altro creando curiose catene Esempio:

ls | tr l z Invoca la lista, prende l'output e lo manipola. In che modo? Sostituendo (tr) le lettere l con z.

ls | head -n Mostra i primi n file di una directory. Senza niente mostra i primi 10 file.

ls | *tail -n* Mostra gli ultimi n file di una directory. Senza niente mostra gli ultimi 10 file.

ls | *wc* Conta le frasi,parole e caratteri.

3.4 Un primo approccio pratico: Manipoliamo passwd

Obiettivo: Sapere quante occorrenze della stringa /usr/bin/false ci sono nel file.

Abbiamo già parlato di passwd dicendo che curiosamente non contiene le password di sistema ma una serie di record con campi che servono al sistema per gestire utenti e demoni. Essendo la sua struttura così ben predisposta alla manipolazione andremo ad eseguire operazioni sull' output dello stesso.

- *Fase 1: Modifichiamo il separatore:* andiamo a modificare il separatore dei campi in passwd:
  cat /etc/passwd | tr : "\n" | less
- *Fase 2: Ordinare i campi:* per far ciò usiamo l'ottimo filtro *sort*:
  cat /etc/passwd | tr : "\n" | sort | less
- *Fase 3: Eliminare i doppioni:* per far ciò utilizziamo *uniq* che si applica ad un output già ordinato:
  cat cat /etc/passwd | tr : "\n" | sort | uniq -c | less .
  Con l'opzione -c accanto al nome vedo anche quante volte compare il campo (era ciò che volevamo)
- *Fase 4: Ordinare in ordine numerico anzichè alfabetico e mostrare la classifica:* per far ciò usiamo *sort -nr* che ordina in modo inverso e numerico e *head* che ci mostra i primi 10:
  cat /etc/passwd | tr : "\n" | sort | uniq -c | sort -nr | head

4 I filtri Grep e Awk

Grep: permette di effettuare selezione di righe in output secondo pattern matching (espressioni regolari)

Awk: permette di fare le stesse cose di grep, ma è più potente perchè è un vero e proprio linguaggio di programmazione.

4.1 Grep

4.1.1 Le espressioni regolari

Una espressione regolare è a tutti gli effetti un pattern:
Es.

- *grep 'a'*: riporta tutti gli output contenenti a
- *grep 'ab?c'*: ? si applica all'oggetto prima e specifica che quell' oggetto può esserci o non esserci
- *grep 'ab*c'*: * si applica all'oggetto prima e specifica che quell' oggetto deve esserci almeno una volta o n volte
- *grep -E '(a | b)'*: matcha a o b. Per usare la pipe | e le parentesi ( ) si deve avviare grep con l'opzione -E
- *grep -v espressione_ regolare*: toglie i risultati matchati dall' output
- *grep '[a-z]'*: è equivalente a: a|b|c|....z
- *grep '[0-9]'*: è equivalente a 0|1|...9
- . : Il punto rappresenta un carattere qualsiasi.

Esempi:

grep ' ^[0-9]$'* : Matcha una parola fatta solo di cifre. ^ è il delimitatore di inizio riga, $ è il delimitatore di fine riga.

grep -v '^$' : Toglie tutte le righe vuote.

\: Serve a matchare caratteri speciali.

4.1.2 Esempi pratici su /etc/passwd

Alcuni esempi di matching:
cat /etc/passwd | grep ':/var[^:]:'* : matcha un qualsiasi campo con dentro var. Notare come [^:] significhi qualsiasi carattere tranne i due punti.
cat /etc/passwd | grep ':[^:]$'* : matcha l'ultimo campo.
Uniamo i due risultati:
cat /etc/passwd | grep ':/var[^:]:[^:]*$'*: matcha il penultimo campo contenente var

4.2 Awk

Ha una sintassi del tipo:

awk '[BEGIN{}] /pattern/ {action} [END{}]'

BEGIN e/o AND sono opzionali. All' interno di pattern o action posso usare delle variabili:
$0 : L'intera riga.
$1: Il primo campo.
$N : l'N-esimo campo. I campi sono delimitati di default da spazi e tabulatori, per cambiare il delimitatore di campo:

[...]BEGIN{FS= ":" }[...il resto del programma...]
oppure:
awk -v FS=: [...il resto del programma...]

4.2.1 Awk e le stringhe

L'unico tipo di dato che esiste in awk è il tipo stringa. Anche operazioni come somme o sottrazioni possono essere fatte ma restituiscono stringhe. Ci sono alcune variabili paraticolari in awk:
NF: Number of Fields
NR: Number of Records
OFS: Output Field Separator

4.2.2 Alcuni Esempi

awk '$1 ~/pippo/' : (tilde è il carattere ascii) Matcha $1 con pippo
awk '/pippo/ {s = s+1} END{print s}': Stampa le occorrenze di pippo
awk 'for(i=1;i<=NF;i++) print i': Stampa il numero incrementale dei campi.
awk '{print $NF }': il valore dell' ultimo campo (NF stampa il numero dell' ultimo campo)
awk '{print $(NF-1) }': il valore del penultimo campo.
awk '{if ($7 == "/bin/sh") print $1,$7}': Stampa tutte le occorrenze di $7 associate al primo campo. La virgola introduce un field separator tra i due campi. *awk '{print $2,$1 }'*: Ribalta due campi. Per migliorare l'impaginazione:
awk '{printf "%-20s %10s \n", $2,$1 }': In questo modo allineo anche a sinistra (-) il primo campo.

4.2.3 Sostituire i match con gsub

Per sostituire i match trovati con qualcos'altro a nostra discrezione ci vengono in soccorso due tool interni ad awk:

- *sub*: Permette di sostituire con qualcos'altro il primo match trovato.
- *gsub*: Permette di sostituire con qualcos'altro tutti i match trovati.

USAGE:

gsub (/epressione_regolare/ , "sostituzione")

Se non specifichiamo nulla dopo *sostituzione* stiamo operando su $0, l'intera riga. Se vogliamo specificare un altro campo:
gsub (/epressione_regolare/ , "sostituzione",$nome_campo)
Nota Bene: Se assegno qualcosa a un qualsiasi campo, allora ho come effetto collaterale che devo ricalcolare $0 a partire dai nuovi campi separati da OFS.
Esempio:
cat /etc/passwd | awk -v FS=: '{gsub(/var/, pippo); print}': Questo comando sostituisce tutte le occorrenze di var con pippo e stampa il risultato.

5 Programmare in Shell

5.1 I sei passi

Abbiamo già discusso le potenzialità della shell, qui le approfondiremo. Innanzitutto è fondamentale elencare i **6 passi** fondamentali, eseguiti in sequenza dalla shell per valutare una espressione:

1. brace expansion {}
2. tilde expansion ~
3. variable,arithmetic,parameter expansion
4. command substitution ' ' (eseguo il comando nei backquote e l'output di tale comando viene inserito nel comando che ho lanciato (vedi esempio touch)
5. word splitting
6. pathname expansion

5.1.1 Esempi vari

Esempio 1:
$ p1=ciao
$ p2=mamma
$ echo $p{1,2}
ciao mamma

Esempio 2:
$ echo a{1,2}b{3,4}
a1b3 a1b4 a2b3 a2b4

Esempio 3:
$ echo ~
/Users/alfredo (la tilde restituisce la home di qualcuno, in questo caso la mia)

Esempio 4:
$ touch 'hostname' (crea un file vuoto che si chiama come il nome del computer attuale. Questo perchè dentro ' ' ho messo il comando speciale hostname)
NOTA BENE: Su tastiera mac i backquote si fanno con alt+9.

5.2 Il nostro primo .sh

Prima di creare il nostro primo script è bene tenere sottolineare come esistano due famiglie di eseguibili:
-I binari, esempio un file compilato da C e quindi tradotto in una sequenza di uno e zero compresibili al calcolatore
-Gli script, che sono file interpretati da un interprete quale Python o la JVM.
I nostri script .sh saranno interpretati dall' interprete che decideremo di specificare qui:

1. #! *nome_interprete*

Dove 1 è la riga di codice del nostro script, nel nostro caso la prima. Quindi preliminarmente bisogna segnalare al kernel quale interprete dovrà usare:
es.
1.#! /bin/bash : lo script sarà interpretato dalla bash
1.#!/usr/bin/awk -f : lo script sarà eseguito da awk
e così via...

Creiamo lo script

$ touch programma.sh
$ chmod +x programma.sh

in questo modo abbiamo reso il nostro programma eseguibile.

5.2.1 Variabili speciali

All' interno del nostro script possiamo utilizzare delle variabili speciali:

- $0...$n: Sono variabili legate ai parametri con cui lanciamo il nostro script. $0 è il nome del file, $1 è il primo parametro e così via...
- $* : Mi da tutti i parametri immessi in input, escluso il nome del file, ovvio.
- $?: Fornisce il valore di ritorno del comando eseguito immediatamente prima: Se ritorna 0 vuol dire che tutto è andato a buon fine, 1 si è verificato un errore

5.2.2 Controllo di flusso

If Ha la seguente sintassi:

if [ ] ; **then** *esegui_1* **else** *esegui_2* **fi**

notiamo come [ ] siano un abbreviazione di test (in realtà la seconda parentesi viene ignorata). Il comando **test** è molto potente perchè ci permette di valutare espressioni e ci fornisce il risultato (0 = positivo , 1 = negativo) che possiamo sempre verificare con *echo $?*. Sulla base di test si base la struttura di controllo if.

Case Il case (switch) permette di valutare una espressione e di fornire comportamenti diversi a seconda della stessa.

5.2.3 Iterazione

for Ha la seguente sintassi:

for i **in** a b c d e ; **do** echo $i ; **done**

```
$ for i in a b c d e ; do echo $i ; done
a
b
c
d
e
```

6 Memory Managment

Dopo aver introdotto il concetto di memoria virtuale e aver capito che in linux possiamo trovare molte info utili in /proc, adesso diamo uno sguardo ad alcuni comandi in unix.

6.1 Il comando free

Il comando free permette di visualizzare alcune informazioni utili sulla memoria.

6.2 Il comando vmstat

Il comando vmstat, invocato con l'opzione "1" fornisce delle statistiche sulla memoria virtuale che vengono aggiornate in real time. Una particolare mensione va ai buffer, una zona di memoria dove il sistema operativo conserva le strutture dati. Le altre sigle sono:

- bi = block interrupt
- in = interrupt
- cs = context switch, sinonimo di process switch
- us = user mode
- sy = system mode
- id = idle
- wa = waiting (una sorta di busy waiting, difatto il kernel non sta facendo nulla)
- r = processi ready

6.3 Altri comandi

Altri comandi che possono tornarci utili sono:

memtester ***dimensione da allocare*** : alloca un tot di memoria per testare la ram

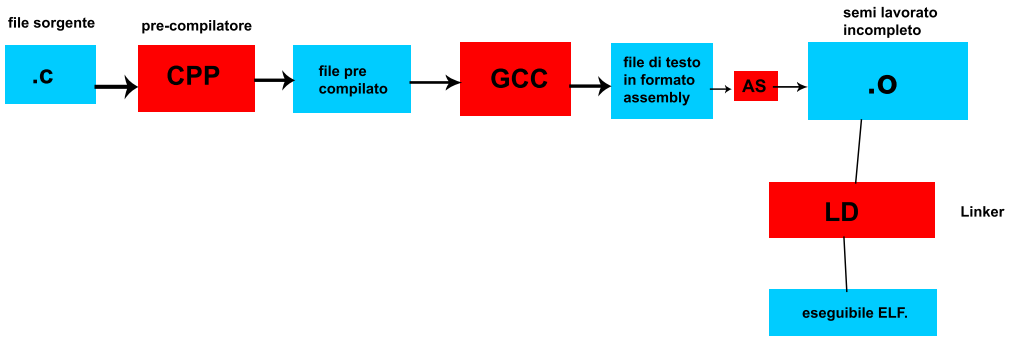
find trova un file all' interno del sistema, se non si specifica nulla semplicemente elenca tutti i file del sistema

/usr/bin/time : Fornisce utili statistiche sul tempo di esecuzione di un comando o di un processo.
Ricordiamo che /dev/null è un particolare valore con il quale si segnala, ad esempio ad una cat, di non stampare l'out, un pò come dire esegui il comando ma non stampare nulla

7 Compilazione e Linking

7.1 Un semplice schema

Un semplice schema di quello che succede nel nostro calcolatore, ogni qualvolta compiliamo un file è il seguente:



Ovvero si parte dal .c, tramite un precompilatore si ottiene un file precompilato, dopodichè tramite gcc :

- Un assemblatore converte il file di testo in assembly generato da gcc in un .o

- Un linker esegue il linking dinamico ottenendo un file in formato elf

Tipicamente i .o di sistema vengono inclusi in appositi raccoglitori con estensione .a, e vengono detti *librerie*.
Il semilavorato non contiene il codice delle procedure invocate nel .c, e tiene una tabella di simboli che associa alcune funzioni utilizzate (o variabili) al programma .c
Il problema delle dipendenze è rappresentabile come un grafo e con questo nome si intende il problema per cui alcune librerie, per funzionare, hanno bisogno, a loro volta, di altre librerie.

7.2 Direttive per il compilatore

- #**include** <nome_libreria.h> oppure "mia_libreria.h": Permette di includere una libreria nel file, in modo da poter sfruttare le funzioni dichiarate al suo interno. Con la include è come mettere il codice dei prototipi dentro il file che include la libreria. Ad esempio, se includo la stdio.h e uso la printf, è come se nel mio file avessi tutto il codice della printf all' interno.Dentro l'include troviamo:
  - Prototipi
  - Definizioni di Simboli
  - Parte di compilazione condizionale
- #**ifdef** e #**ifndef**: Due direttive che fanno parte della compilazione condizionale. In particolare ifdef è molto utile nello sviluppo di codice multipiattaforma:

```
...
#ifdef __LINUX_H__
        //esegui codice linux.
        //Nota che il simbolo __LINUX_H__ e' di pura fantasia.
#else
        #ifdef __WINDOWS_H__
                //esegui codice windows.
                //Nota che il simbolo __WINDOWS_H__ e' di
                //pura fantasia.
        #endif
#endif
...
```

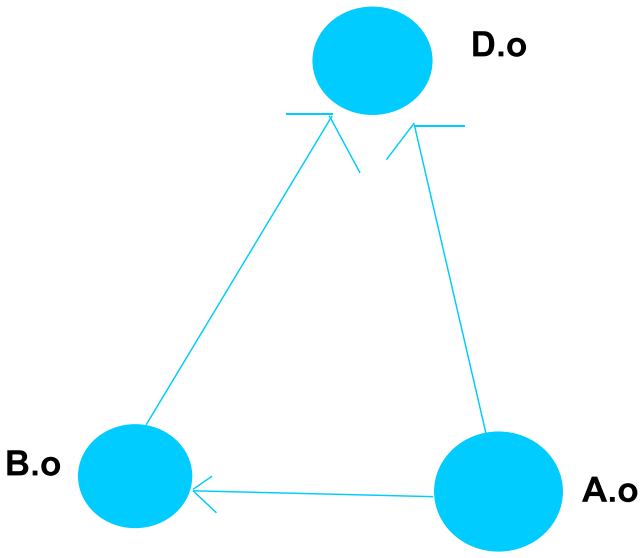
- **#define**: Permette di definire simboli nel precompilatore accoppiati con una stringa:

```
...
#define P 3.14
...
```

  Come effetto ottengo che ogni volta uso P nel mio programma il compilatore lo sostituisce con 3.14.

7.3 Compilazione condizionale

La compilazione condizionale risulta molto utile in caso di /textitmultiple including. Cosa si intende? Abbiamo visto come il problema delle dipendenze sia rappresentabile come un grafo; con questa idea in testa possiamo pensare ad una situazione come quella in figura, dove A e B sono due librerie scritte da due persone diverse: A conosce l'interfaccia che B ha dichiarato per la sua libreria.



Il file A.c conterrà pertanto i seguenti include:

```
...
#include <B.h>
#include <D.h>
...
```

Mentre il file B.c ...

```
...
#include <D.h>
...
```

Guardando quel grafo possiamo notare come non sarebbe necessario includere anche D.h in A.c, poichè tale dichiarazione viene effettuata già da B.c: l'inclusione in A però rappresenta un **atto dovuto** perchè, se un domani il programmatore di B decidesse di cambiare il codice e non usasse più la libreria D.h, il programma smetterebbe di funzionare! Così però abbiamo sollevato un altro problema, allo stato attuale la libreria D.h è inclusa due volte! Per risolvere questo genere di situazioni entra in gioco la compilazione condizionale che funziona come segue:

```
// file D.h
#ifndef __D_H__
        #def __D_H__
                ...
                //codice di D.h
                ...
        #endif
```

E' in questo modo che vanno scritti correttamente gli include!. In questo modo abbiamo ottenuto che, se quel simbolo non è definito allora viene definito e si include il codice, altrimenti nulla di fatto. Ciò si traduce con il fatto che una libreria può essere inclusa quante volte si vuole (anche IMPLICITAMENTE) e per il pre-compilatore conta come se fosse stata inclusa una volta sola!
Su linux la directory /usr/include contiene tutte le inclusioni.

7.4 Linking Dinamico

Il linking dinamico è un processo con cui il link sfrutta delle regioni di memoria condivisa per ridurre al minimo le dimensioni di un eseguibile. Ciò è possibile in virtù del fatto che le librerie dinamiche non vengono incluse nell' eseguibile ELF.

7.5 Strumenti di Sistema

7.5.1 Gcc e le sue opzioni

-c ferma la compilazione al file oggetto

-S ferma la compilazione al codice assembly

-E ferma la compilazione al codice precompilato

-O genera codice ottimizzato

-Wall attiva tutti i warning

gcc hello.c -static -o hello : Non usa il linking dinamico

7.5.2 Objdump, le sue opzioni, e altri programmi

objdump: Strumento che ci permette di eseguire il dump di un bin per debug/esplorazione. Le opzioni di objdump:

-s : fa il dump di tutto.

-T: esegue il dump e mostra la symbol table

-D: disassembla il binario

ldd nome_file : mi dice con quali librerie è stato linkato il mio file

ltrace: Intercetta tutte le chiamate del programma alle librerie dinamiche

8 Makefile

Per i makefile si veda la cartella 09.

9 It's time to debug

9.1 GDB - GNU DeBugger

Il gdb è un potente debugger di binari in formato elf da linea di comando.
Nota: Per poter debuggare un programma bisogna compilarlo con l'opzione -g in modo da includere nel bin i simboli di debug.

9.2 Avviare gdb

Avviare gdb è molto semplice e si fa con il comando da shell:

gdb <file_da_debuggare>

Fatto ciò ci troviamo difronte ad un prompt testuale:

```
GNU gdb 6.3.50-20050815 (Apple version gdb-962) [...]
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU [...]
welcome to change it and/or distribute copies [...]
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type [...]
This GDB was configured as "i386-apple-darwin"...
Reading symbols for shared libraries ... done

(gdb)_
```

A questo punto il programma aspetta da noi comandi.

9.3 Il comando run

Permette di lanciare un programma. Durante questa lezione useremo come studio di caso il seguente programma:

```
1       #include <stdio.h>
2
3       int main(){
4           printf("hello World! \n");
5           return 0;
6       }
```

Per lanciare la sua esecuzione usiamo il comando run.

run prova.c

Nota: Il programma accetta per comodità anche solo la prima lettera del comando, in questo caso r, ma solo se non ci sono ambiguità tra due o più comandi.
In quel caso dobbiamo disambiguare il nome scrivendo più lettere:

```
(gdb) run
Starting program: /Users/Alfredo/Uni/III ANNO/Algoritmi[...]
Reading symbols for shared libraries ++. done
hello World!

Program exited normally.
```

9.4 Il comando help

Il comando help fornisce utili informazioni circa un particolare comando:

```
(gdb) help run
Start debugged program.  You may specify arguments [...]
If 'start-with-shell' is set to 1, args may include [...]
they will be expanded using "sh".  Input and output [...]
">", "<", or ">>" will also be allowed.

With no arguments, uses arguments last specified [...]
To cancel previous arguments and run with [...]
use "set args" without arguments.
```

9.5 Il comando list

Fornisce il listato del programma, ma può essere impiegato come efficiente strumenti di ricerca di funzioni all' interno di un programma. **Nota**: Dare due volte il comando list comporterà un errore da parte di gdb: questo è dovuto al fatto che il programma ha memoria
dei comandi dati e tenta di leggere oltre. Possiamo forzare il programma per leggere nuovamente il programma da capo scrivendo *list 0*

9.6 Il comando next

Permette di avanzare nel programma "entrando" dentro un ciclo o una funzione.

9.7 Il comando print

Permette di stampare (solo una volta) il valore di una variabile:

```
(gdb) print i
```

9.8 Il comando display

Permette di stampare, dopo ogni comandi impartito a gdb, il valore di una variabile:

```
(gdb) display i
```

9.9 Il comando step (into)

Permette di avanzare nel programma "entrando" dentro un ciclo o una funzione.

9.10 Il comando continue

Prosegue con la normale esecuzione del programma senza fermarsi mai, a meno che non incontri un breakpoint.

9.11 I breakpoint

I breakpoint costituiscono gli strumenti più importanti per il debugging, in quanto permettono di fissare un punto all' interno del programma in cui l'esecuzione dello stesso viene congelata, dandoci la possibilità di esplorare il mondo che ci circonda.

9.11.1 Aggiungere un breakpoint

Inseriamo un breakpoint direttamente alla funzione principale:

```
(gdb) break main
Breakpoint 1 at 0x1fd7: file prova.c, line 4.
```

Il debugger ci segnala che ha fatto quanto abbiamo richiesto.

9.11.2 Disabilitare i breakpoint

Ovviamente è possibile disabilitare temporaneamente un breakpoint nel caso non ci servisse più:

```
(gdb) disable 1
```

Con questo comando abbiamo disabilitato il breakpoint appena creato.

9.11.3 Eliminare un breakpoint

Nel caso in cui ci accorgessimo che un breakpoint non sia più utile fino alla fine del debugging è sempre possibile eliminarlo:

```
(gdb) delete 1
```

9.11.4 I breakpoint condizionali

GDB prevede la possibilità di inserire dei breakpoint condizionali, cioè attivabili solo al verificarsi di una determinata condizione:

cond <numero_del_breakpoint> <condizione_C-like>

9.11.5 Ignorare i breakpoint

Un' altra possibilità offerta da gdb è quella di ignorare i breakpoint per n volte, con n fissabile a piacere:

cond <numero_del_breakpoint>
<numero_di_volte_che_verrà_ignorato>

Così scrivere una cosa del genere...

```
(gdb) ignore 3 12
Will ignore next 12 crossings of breakpoint 3.
```

...produrrà come effetto che il breakpoint 3 verrà ignorato per 12 volte prima di diventare attivo.

9.12 Esplorare lo stack

E' possibile esplorare lo stack, andando a guardare il contenuto di ogni singolo frame allocato nello stack.

9.12.1 Il comando bt

Mostra lo stato corrente dello stack in un dato momento dell' esecuzione di un programma:

```
(gdb) bt
#0  main () at prova.c:6
```

9.12.2 Il comando up

Permette (qualora ce ne fossero) di spostarsi nella pila, selezionando di volta in volta un frame allocato sempre più in alto.

9.12.3 Il comando down

Analogo ad up salvo per il fatto che in questo modo si scende verso il fondo della pila (verso il first in).

9.12.4 Il comando info

E' un utile comando che accetta diversi valori. *info frame* fornisce informazioni sul frame corrente:

```
(gdb) info frame
Stack level 0, frame at 0xbffff790:
 eip = 0x1fc4 in main (prova.c:6); saved eip 0x1f82
 source language c.
 Arglist at 0xbffff788, args:
 Locals at 0xbffff788, Previous frame's sp is 0xbffff790
 Saved registers:
  ebx at 0xbffff784, ebp at 0xbffff788, eip at 0xbffff78c
```

Mentre *info stack* fornisce la situazione dello stack:

```
(gdb) info stack
#0  main () at prova.c:6
```

10 Le utility diff e patch

Queste due utility sono molto utili in grandi progetti quando si lavora in team; la prima mostra le differenze tra due file, creando una sorta di delta, la seconda applica il delta al primo file.

10.1 diff

Usage:

$ diff file1.txt file2.txt

10.1.1 Opzioni di diff

-r data una directory genera le differenze ricorsivamente

-N elenca il contenuto dei file nuovi

-u elenca i file in formato unified

10.2 patch

Quello che fa patch si può riassumere così:

$$\text{patch} = \text{x1} + \text{delta}$$

dove delta = x2 - x1
Nota: Per essere indipendenti dalla directory in cui usiamo il comando usare l'opzione -p1.
Usage:

patch [options] [originalfile [patchfile]]

cioè prende il delta (patchfile) e lo applica ad originalfile.

10.3 Un esempio

Create due cartelle:
proj1
—-A.txt
proj2
—-A1.txt

lanciamo i seguenti comandi:
$ diff -ruN proj1/A.txt proj2/A2.txt > mypatch e poi: $ patch -p1 -b proj1/A.txt mypatch

così abbiamo anche salvato il backup di A.txt