Sistemi Operativi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

a.a. 2014/15

Sistemi Operativi
Bruschi Monga Re
Unix power tools
find
Archivi
Software factory
Make
Debugger
Low level programming
diff & patch
Versioning

Lezione XVII: Unix Power tools

find

Sistemi Operativi
Bruschi Monga Re
Unix power tools
find
Archivi
Software factory
Make
Debugger
Low level programming
diff & patch
Versioning

Per selezionare file con determinate caratteristiche si usa `find`

find percorso predicato

Seleziona, nel sottoalbero definito dal percorso, tutti i file per cui il predicato è vero

Spesso usato insieme a a `xargs`

find percorso predicato| xargs comando

funzionalmente equivalente a

comando $(find percorso predicato)

ma evita i problemi di lunghezza della riga di comando perché xargs si preoccupa di "spezzarla" opportunamente.

Due espressioni idiomatiche

Sistemi Operativi
Bruschi Monga Re
Unix power tools
find
Archivi
Software factory
Make
Debugger
Low level programming
diff & patch
Versioning

Spesso si vuole fare un'operazione per ogni file trovato con find. L'espressione piú naturale sarebbe:

```
1 for i in $(find percorso predicato); do
2   comando $i
3 done
```

Questa forma presenta due problemi: può eccedere la misura della linea di comando e non funziona correttamente se i nomi dei file contengono *spazi*

Due espressioni idiomatiche

Un'alternativa è

```
1 find percorso predicato -print0 | xargs -0 -n 1
```

In questo modo (-print0) i file trovati sono separati dal carattere 0 anziché spazi e xargs è capace di adattarsi a questa forma. Un'alternativa piú generale che mostra la potenza del linguaggio di shell che non distingue fra comandi e costrutti di controllo di flusso (sono tutti "comandi" utilizzabili in una pipeline)

```
1 find percorso predicato | while read x; do
2   comando $x
3 done
```

read x legge una stringa e la assegna alla variabile *x*.

Esercizi

1. Trovare il file piú "grosso" in un certo ramo
2. Copiare alcuni file (ad es. il cui nome segue un certo pattern) di un ramo in un altro mantenendo la gerarchia delle directory
3. Calcolare lo spazio occupato dai file di proprietà di un certo utente
4. Scrivere un comando che conta quanti file ci sono in un determinato ramo del filesystem

Archivi

Un archivio *archive* è un file di file, cioè un file che contiene i byte di diversi altri file e i relativi *metadati*. (Cfr. con una directory, che è un file speciale, che sostanzialmente contiene solo l'elenco dei file)

- `ar` L'archiviatore classico, generalmente utilizzato per le librerie (provare ar t /usr/lib/i86/libc.a
- `tar` Tape archive, standard POSIX
  tar cvf archivio.tar lista_files

Gli archivi possono essere compressi con `compress` o, piú comunemente, con `gzip` o `bzip2`
I file `.zip` sono archivi compressi.

Altre utility

Altre utility "standard" di cui è bene conoscere almeno l'esistenza

Prog. (sez. man)	**Descrizione**
`uniq (1)`	report or omit repeated lines
`cut (1)`	remove sections from each line of files
`tr (1)`	translate or delete characters
`dd (1)`	convert and copy a file
`stat (1)`	display file or file system status
`tee (1)`	read from standard input and write to standard output ...
`basename (1)`	strip directory and suffix from filenames
`dirname (1)`	strip non-directory suffix from file name
`sed (1)`	stream editor for filtering and transforming text
`seq (1)`	print a sequence of numbers

Inoltre è molto utile conoscere le espressioni regolari (`man 7 re_format`), usate da `grep`, `sed`, ecc.

Esercizi

1. Creare un archivio `tar.gz` contenente tutti i file la cui dimensione è minore di 50KB
2. Rinominare un certo numero di file: per esempio tutti i file `.png` in `.jpg`
3. Creare un file da 10MB costituito da caratteri casuali (usando `/dev/random`) e verificare se contiene la parola JOS
4. Trovare l'utente che ha il maggior numero di file nel sistema
5. Trovare i 3 utenti che, sommando la dimensione dei loro file, occupano piú spazio nel sistema.

Lezione XVIII: The UNIX software factory

UNIX software factory

- UNIX nasce come sistema *per i programmatori* (l'unica tipologia di utente all'inizio degli anni '70...)
- progettato insieme ad un linguaggio di programmazione (C)
- la 'filosofia di UNIX' (piccoli programmi che fanno molto bene una sola cosa su file) si adatta perfettamente al paradigma di sviluppo edit-compile-debug
- tool all'avanguardia nell'elaborazione di *file di testo* (per lo piú organizzati per "righe") e per la scrittura dei programmi di elaborazione stessi (lex, yacc,...)

Edit/Compile

- Editor: `ed`, `vi`, `emacs` manipolano arbitrariamente i byte di un file, generalmente interpretandoli come caratteri stampabili (testo)
- Compilatore: `cc` (`gcc`)
  1. `cc` sorgente (`.c`) $\rightsquigarrow$ assembly (`.s`)
  2. `as` assembly $\rightsquigarrow$ *oggetto* (`.o`)
  3. (`ar` archivia diversi oggetti in una *libreria* (`.a`)
  4. `ld` *oggetti* e *librerie* $\rightsquigarrow$ eseguibile (`a.out`) (il formato storico è COFF, oggi ELF)

Si noti che a sua volta anche la compilazione vera e propria è fatta da due passi (pre-processore `cpp` e compilazione `cc1`).

Make

Stuart Feldman, 1977 at Bell Labs.
Permette di specificare dipendenze fra processi di generazione.
Dipendenze: se cambia (secondo la data dell'ultima modifica) un prerequisito, allora il processo di generazione deve essere ripetuto.

```
1 helloworld.o: helloworld.c
2         cc -c -o helloworld helloworld.c
3
4 helloworld: helloworld.o
5         cc -o $@ $<
6
7 .PHONY: clean
8 clean:
9         rm helloworld.o helloworld
```


Concetti fondamentali

Breakpoint

Un punto del programma in cui l'esecuzione deve essere bloccata, tipicamente per esaminare lo stato in quell'istante.

Stepping

Eseguire il programma *passo a passo*. La granularità del passo può arrivare fino all'istruzione macchina.

Stato

Lo stato del programma può essere analizzato come:

- forma simbolica: secondo i simboli definiti nel linguaggio di alto livello e conservati come *simboli di debugging*
- memoria virtuale: stream di byte suddiviso in segmenti
  - Text: contiene le istruzioni (spesso read only)
  - Initialized Data Segment: variabili globali inizializzate
  - Uninitialized Data Segment (bss): variabili globali non inizializzate
  - Stack: collezione di *stack frame* per le chiamate di procedura. Cresce verso il basso.
  - Heap: Strutture dati create dinamicamente. Cresce verso l'alto tramite system call `brk` (API `malloc`).

Uso del debugger

- `break ...` (un simbolo o un indirizzo `*0x...`)
- `run ...` (eventualmente con argv)
- `print ...` (`x`)
- `next` (`nexti`)
- `step` (`stepi`)
- `backtrace`

Simboli

La *symbol table* serve al *linker* per associare nomi simbolici e indirizzi prodotti dal compilatore:

- contenuta in tutti gli oggetti, generalmente viene lasciata anche negli eseguibili (ma può essere scartata con `strip`)
- una versione piú ricca viene detta "simboli di debug" (vari formati, p.es. DWARF)
- le tabelle dei simboli possono essere consultate con `nm`

Perché capire i dettagli delle fasi?

Per costruire sistemi operativi a volte serve alterare il flusso tradizionale

```
1 gcc -O -nostdinc -I. -c bootmain.c
2 gcc -nostdinc -I. -c bootasm.S
3 ld -m elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
4 objdump -S bootblock.o > bootblock.asm
5 objcopy -S -O binary -j .text bootblock.o bootblock
```

```
1 $ nm kernel | grep _start
2 8010b50c D _binary_entryother_start
3 8010b4e0 D _binary_initcode_start
4 0010000c T _start
```


Assembly in C

In alcuni casi è comodo mischiare l'assembly al C (meno laborioso di organizzare il collegamento)

```
1 __asm__("nop");
2
3 __asm__("movl %eax, %ebx");
4 __asm__("xorl %ebx, %edx");
5 __asm__("movl $0, _booga");
6
7 __asm__("pushl %eax\n\t"
8       "movl $0, %eax\n\t"
9       "popl %eax");
```

Attenzione! Il compilatore C non "vede" l'effetto delle istruzioni assembly.

Assembly in C (cont.)

Si possono fare anche cose piú complicate, ma la sintassi è poco "amichevole"

```
1 __asm__("cld\n\t"
2       "rep\n\t"
3       "stosl"
4       : /* no output registers */
5       : "c" (count), "a" (fill_value), "D" (dest)
6       : "%ecx", "%edi" );
```

La sintassi è

```
1 __asm__( "statements" : output_registers : input_registers : clobbered_registers);
```

`http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html`

diffutils

Con `cmp` è possibile controllare se due file sono identici.
Per i file di testo organizzato il righe esistono strumenti piú sofisticati:

- `diff` elenca le modifiche necessarie per trasformare un file in un altro (`diff3` si aiuta con un "antenato" comune, fondamentale per facilitare il *merge*)
- `diff` (e in maniera piú evoluta `diff3`) cerca di identificare le righe che *non sono cambiate*: le modifiche sono organizzate per hunk
- `patch` riapplica gli hunk di modifica al file originale (o versioni *leggermente* modificate dei medesimi)

Revision, version, configuration management

Dagli anni '80 sono stati proposti molti strumenti per trattare in modo efficiente:

- le successive revisioni di un file
- le versioni di un prodotto software
- le configurazioni che permettono di ottenere una specifica versione del prodotto

SCCS, RCS, CVS, SVN, `git`...
Si basano tutti sulla conservazione della "storia" dello sviluppo in un *repository*: per lavorare occorre fare *checkout* di un *artifact*, e poi chiedere il *commit* delle modifiche.

Versioning

L'idea può essere incorporata a vari livelli: Emacs può "salvare" automaticamente le versioni precedenti dei file (generalmente una sola `*` , altrimenti `* 1 ...`), oppure addirittura nel *file system*.
Git invece ricrea un suo "file system": blob e tree, ref.

- multi-phase commit: *working directory*, *stage* e *local repository*
- distribuito senza necessariamente server centralizzati: `pull` e `push`
- in un commit è conservato l'insieme delle modifiche (come 'diff') fatte ad un insieme (*change-set*) di file: perciò è associato a un *tree*
- una branch è semplicemente una *reference* mobile a una linea di sviluppo.