



Sistemi
Operativi

Bruschi
Monga Re

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS

Sistemi Operativi¹

Mattia Monga

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

a.a. 2016/17



Sistemi
Operativi

Bruschi
Monga Re

Lezione XVII: The UNIX software factory

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS



Sistemi
Operativi

Bruschi
Monga Re

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS

- 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,...)



- 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-processor `cpp` e compilazione `cc1`).



Sistemi
Operativi

Bruschi
Monga Re

- Scrivere in assembly (nasm) una funzione *somma* che restituisce (in *eax* secondo la convenzione del C) la somma di due interi (passati sullo stack, secondo la convenzione del C)
- Scrivere un programma C che usa la funzione *somma*
- Collegare i due programmi in un unico eseguibile

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS



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.

```
helloworld.o: helloworld.c  
    cc -c -o helloworld helloworld.c
```

```
helloworld: helloworld.o  
    cc -o $@ $<
```

```
.PHONY: clean  
clean:  
    rm helloworld.o helloworld
```



- Scrivere in assembly (nasm) una funzione *somma* che restituisce (in *eax* secondo la convenzione del C) la somma di due interi (passati sullo stack, secondo la convenzione del C)
- Scrivere un programma C che usa la funzione *somma*
- Collegare i due programmi in un unico eseguibile
- **Codificare il procedimento in un Makefile**

Debugger: Concetti fondamentali



Sistemi
Operativi

Bruschi
Monga Re

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.

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS



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`).

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS



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



Sistemi
Operativi

Bruschi
Monga Re

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`

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS



- 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-processor `cpp` e compilazione `cc1`).



Perché capire i dettagli delle fasi?

Per costruire sistemi operativi a volte serve alterare il flusso tradizionale

```
gcc -O -nostdinc -I. -c bootmain.c
gcc -nostdinc -I. -c bootasm.S
ld -m elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootmain.o bootasm.o
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
$ nm kernel | grep _start
8010b50c D _binary_entryother_start
8010b4e0 D _binary_initcode_start
0010000c T _start
```

Sistemi
Operativi

Bruschi
Monga Re

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi

diff & patch
Versioning

JOS



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

```
__asm__("nop");

__asm__("movl %eax, %ebx");
__asm__("xorl %ebx, %edx");
__asm__("movl $0, _booga");

__asm__("pushl %eax\n\t"
        "movl $0, %eax\n\t"
        "popl %eax");
```

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

Sistemi
Operativi

Bruschi
Monga Re

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS



Assembly in C (cont.)

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

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

La sintassi è

```
__asm__( "statements" : output_registers : input_registers :
```

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

Sistemi Operativi

Bruschi
Monga Re

Software
factory
Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS



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



Sistemi
Operativi

Bruschi
Monga Re

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.

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS



L'idea può essere incorporata a vari livelli: Emacs può “salvare” automaticamente le versioni precedenti dei file (generalmente una sola $*~$, altrimenti $*~1~\dots$), 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.



Servono almeno 512MB di ram (-m 512 in Qemu) e persistence-jos.qcow (-hda persistence-jos.qcow in Qemu) in modo da salvare il proprio lavoro.

```
$ cd /home/user/joslab
$ make
$ make qemu-nox
```

```
K> kerninfo
```

```
Special kernel symbols:
```

_start		0010000c	(phys)
entry	f010000c	(virt)	0010000c (phys)
etext	f0101a6d	(virt)	00101a6d (phys)
edata	f0112300	(virt)	00112300 (phys)
end	f0112944	(virt)	00112944 (phys)

```
Kernel executable memory footprint: 75KB
```

```
Per uscire Ctrl-a+x
```



Sistemi
Operativi

Bruschi
Monga Re

Seguiremo

<http://pdos.csail.mit.edu/6.828/2016/labs/lab1/>

(spesso semplificando per motivi di tempo: non è vietato cercare di seguire tutti gli spunti del corso MIT! Tenete conto che gli studenti MIT hanno circa 2 settimane per realizzare gli obiettivi di ogni lab)

Software
factory

Make
Debugger
Low level
programming

Programmare
sistemi
operativi
diff & patch
Versioning

JOS