Sistemi Operativi[1]

Mattia Monga

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

a.a. 2014/15

Lezione XIX: JOS

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`).

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.

Iniziare con JOS

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

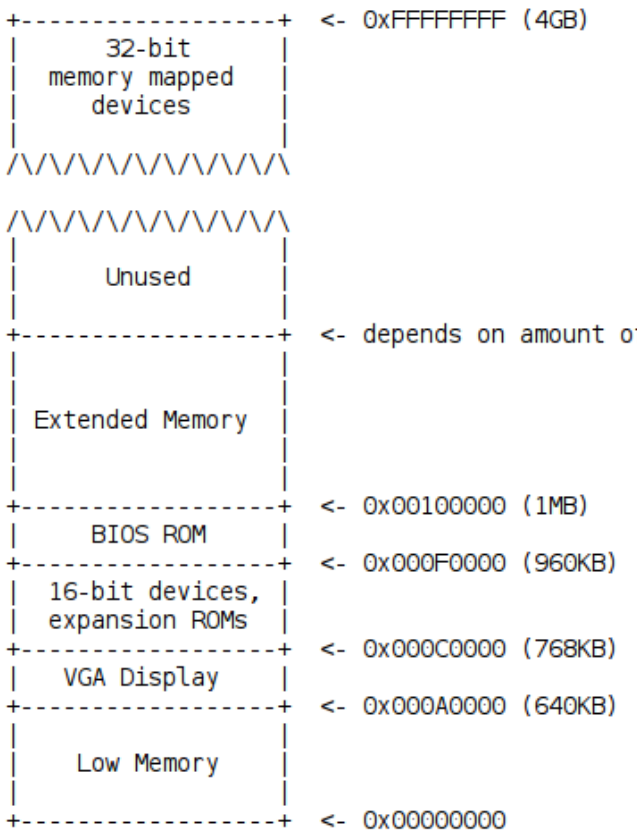
```
1  $ cd /home/jos/joslab
2  $ make
3  $ make qemu-nox
4
5  K> kerninfo
6  Special kernel symbols:
7     _start 0010000c (phys)
8     entry f010000c (virt) 0010000c (phys)
9     etext f0101a6d (virt) 00101a6d (phys)
10    edata f0112300 (virt) 00112300 (phys)
11    end f0112944 (virt) 00112944 (phys)
12 Kernel executable memory footprint: 75KB
```

Per uscire `Ctrl-a+x`

Struttura dell'esercitazione

Seguiremo
http://pdosnew.csail.mit.edu/6.828/2014/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)

Layout della memoria



La mappa della memoria è definita dal costruttore. Generalmente accessibile via firmware o con tecniche di probing (GRUB2 fornisce un comando lsmmap)

Layout della memoria

Start

```
1 [f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
```

L'indirizzo fisico è calcolato secondo il Real-Mode addressing (a 16 bit)