

# Programmazione di Sistema e di Rete

Matteo Aprile  
Professore: Franco Tommasi

## INDICE

### I Libri di testo consigliati

### II Comandi utili

- II-A find: trovare tutti i file eseguibili . . . .
- II-B find: trovare file di intestazione del mac  
come stdio.h . . . . .
- II-C lld: per capire che librerie usa il codice

### III Introduzione - 23/27.09.22

- III-A System call . . . . .
- III-B Programma Make - 28.09.22 . . . . .

### IV Compilazione - 28.09.22

- IV-A direttive di preprocessore . . . . .
- IV-B Librerie . . . . .
- IV-C Creazione librerie . . . . .
- IV-D Aggiornamento librerie . . . . .

### Riferimenti bibliografici

#### I. LIBRI DI TESTO CONSIGLIATI

- Advanced Programming in the Unix Environment, 3th ed, Stevens, Rago
- TCP/IP 1, Stevens (facoltativo)
- Unix Networking Programming the Socket Networking API, Stevens
- The Linux Programing Interface, Kerrisk
- manset
- Gapil Guida alla Programmazione in Linux, Simone Piccardi

#### II. COMANDI UTILI

##### A. find: trovare tutti i file eseguibili

```
1 $ find . -type f -perm -0100
2 ./standards/makeopt.awk
3 ./standards/makeconf.awk
4 ./proc/awkexample
5 ./systype.sh
6 ./advio/fixup.awk
```

##### B. find: trovare file di intestazione del mac come stdio.h

```
1 find /Applications/Xcode.app/ -name stdio.h 2>/dev/
null
```

##### C. lld: per capire che librerie usa il codice

```
1 lld [nomev codice]
```

## III. INTRODUZIONE - 23/27.09.22

### A. System call

- 1 Sono uguali alle funzioni di libreria dal punto di vista sintattico, però cambia il modo di compilarle. Notare che non possono essere usati i nomi delle SC per delle function call.
- 1 Per poi poter "raccontare" tra umani le sequenze di bit che vengono mandate ai processori si usa assembly.
- 1 Sono effettivamente delle chiamate a funzioni ma poi dal codice assembly puoi capire che è una system call dato che ha dei meccanismi specifici.

Alcuni esempi di chiamate e registri:

- **eax** : registro dove metti il **numero della sc**
- **int 0x80**: avvisa il **kernel** che serve chiamare una sc
- **exit()**: chiudere un processo
- **write()**:

```
2 1 mov edx,4 ; lunghezza messaggio
3 2 mov ecx,msg ; puntatore al messaggio
4 3 mov ebx,1 ; file descriptor
5 4 mov eax,4 ; numero della sc
6 5 int 0x80
```

dove nel **file descriptor** indichi a quale file devi mandare l'output. Questo viene usato dato che così non deve cercare il path ogni volta ma lo mantiene aperto riferendosi ad esso tramite il numero.

### B. Programma Make - 28.09.22

Quando viene avviato verifica la presenza di un file chiamato "Makefile", oppure si usa 'make -f'. In questo file ci sono le **regole di cosa fare per automatizzare delle azioni per un numero n di file**. Se, durante la compilazione di massa, **una di queste dà un errore il programma make si interrompe**, per evitare ciò si usa '-i' (ignore).

Andiamo a guardare **cosa contiene Makefile**:

```
1 DIRS = lib intro sockets advio daemons datafiles db
2 environ \
3 fileio filedir ipc1 ipc2 proc pty relation
4 signals standards \
5 stdio termios threadctl threads printer
6 exercises
7 all:
8 for i in $(DIRS); do \
9 (cd $$i && echo "making $$i" && $(MAKE) ) ||
10 exit 1; \
11 done
12 clean:
13 for i in $(DIRS); do \
14 (cd $$i && echo "cleaning $$i" && $(MAKE)
15 clean) || exit 1; \
16 done
```

dove **all e' detto target**, cioè la cosa che si vuole fare, eseguiamo allora un "make all". **Essendo il primo target, sarà anche quello di default.**

Possono essere presenti dei **prerequisiti**, dopo i ":", che possono essere a loro volta dei target.

Obbligatoriamente avremo, dopo i prerequisiti, la **riga delle regole** che **indica cosa può fare il target.**

#### IV. COMPILAZIONE - 28.09.22

##### A. direttive di preprocessore

Sono delle **indicazioni date a gcc prima di iniziare la compilazione.**

Iniziano tutte con '#':

- **#include**: serve ad **includere delle librerie** di sistema (<lib.h>) oppure di librerie fatte da noi e non in directory standard ("lib.h")
- **#define**:
  - permette di **creare delle "macro"**, che vanno a sostituire una stringa con un'altra (es: #define BUFLN), può capitare che debbano essere definite delle macro prima che si compili il programma, in questi casi si usa scrivere es: '-DMACOS'
  - permette di **creare delle "function like macro"** (es: #define ABSOLUTE\_VALUE(x) (((x<0)?-(x):(x)))
- **#ifdef, #ifndef, #endif**: usata per far accadere qualcosa nel caso un macro sia stata definita

```
1 #ifdef VAR
2 print("hello");
3 #endif
```

**Per evitare che piu' file includano lo stesso si usano degli #ifndef in tutto il codice, in modo da evitare doppie definizioni.**

##### B. Librerie

Durante la fase di compilazione creiamo dei file oggetto (.o) per ogni file in cui è scritta la descrizione delle funzioni di libreria (.c)

```
1 gcc -c bill.c
```

Si andrà poi a creare il **prototipo della funzione (.h).**

In fine **tramite il linker si andranno ad unire tutti i file per crearne uno unico** con tutte le definizioni delle funzioni incluse nelle librerie, di sistema e non, importate. Si vanno quindi a **sciogliere tutti i riferimenti incrociati.**

```
1 gcc -o program program.o bill.o
```

Per quanto riguarda le **funzioni di sistema** NON abbiamo il file sorgente ma abbiamo direttamente l'eseguibile. In compenso abbiamo un **file di libreria**, cioè un insieme di file oggetto linkati in un unico file, dove c'è il codice oggetto di tutte le funzioni.

Abbiamo **2 tipi di librerie**:

- **statiche**: è una **collezione di file oggetto** che hanno il codice compilato delle funzioni e che verranno **linkati al momento della compilazione**. Il programma che si crea sarà possibile essere eseguito solo sullo stesso OS.

**Il problema si ha nell'aggiornamento delle librerie al momento della scoperta di un bug.** Una volta corretto servirà ricevere la versione corretta per poter aggiornare il programma.

- **dynamic**: ricordano il concetto di plug-in, quindi **viene invocato a runtime e caricato in memoria** (es: aggiornamenti dei OS). **L'eseguibile non viene toccato la correzione avviene solo nella libreria.**

Il requisito maggiore è che chi si passa il codice debba avere lo stesso OS dell'altro utente. Notare che **non cambia il prototipo** dato che sennò bisognerà ricompilare l'intero programma.

In generale le **librerie statiche sono molto pericolose** infatti alcuni OS le aboliscono **per le questioni di sistema**. Su linux si ha come libreria statica 'lib.c' che è la libreria con le funzioni più usate in c. Per macos è stata abolita.

Per compilare con la versione dinamica non servono opzioni, per la statica si usa:

```
1 gcc -static
```

##### C. Creazione librerie

Per costruire una **libreria statica per MacOS:**

- 1) costruiamo il **file oggetto**:

```
1 gcc -c libprova.c
```

- 2) costruiamo la **libreria** (con ar=archive, c=create se lib.a non esiste):

```
1 ar rcs libprova.a libprova.o
```

- 3) costruire il **codice** che usa la libreria (con -Wall=verbose warning, -g=debugging, -c=create del file):

```
1 gcc -Wall -g -c useprova.c
```

- 4) **linker** per risolvere le chiamate incrociate (con -L.=dove prendere la libreria, -l[nomelib]=usare la libreria):

```
1 gcc -g -o useprova useprova.o -L. -lprova
```

Per capire che librerie usa il codice si usa:

```
1 ottool -L [nomecodice]
```

Per costruire una **libreria statica per Linux:**

- 1) costruiamo il **file oggetto**:

```
1 gcc -fPIC -Wall -g -c libprova.c
```

- 2) costruiamo la **libreria** (con 0.0=versione della libreria):

```
1 gcc -g -shared -Wl,-soname,libprova.so.0 -o
libprova.so.0.0 libprova.o -lc
```

- 3) costruire il **link simbolico per aggiornare le librerie** senza aggiornare gli eseguibili e senza cambiare il nome del programma:

```
1 ln -sf libprova.so.0.0 libprova.so.0
```

- 4) **linker** per risolvere le chiamate:

```
1 ln -sf libprova.so.0 libprova.so
```

Per capire che librerie usa il codice si usa:

```
1 ldd [nomecodice]
```

#### D. Aggiornamento librerie

Su **Linux** il sistema **andra'** a prendere direttamente una **libreria dinamica**, per evitare ciò e far trovare la nostra, basterà **impostare una variabile di ambiente**:

```
LD_LIBRARY_PATH=`pwd` ldd useprova
```

Tipicamente la libreria viene distribuita nelle directory di sistema andandola ad "installare".

Su **MacOS** la libreria dinamica è un **.dylib**:

```
gcc -dynamiclib libprova.c -o libprova.dylib
```

Quindi eseguendo il programma **trovera' la libreria controllando nella directory corrente** e quindi non serve creare la variabile di ambiente come su Linux.

i file di intestazione del mac come stdio.h per cercarla uso:

```
find /Applications/Xcode.app/ -name stdio.h 2>/dev/  
null
```

#### RIFERIMENTI BIBLIOGRAFICI

[1] <https://en.wikibooks.org/wiki/LaTeX/Hyperlinks>