Modulo 4: Programmi eseguibili

Laboratorio di Sistemi Operativi I
Anno Accademico 2005-2006

Francesco Pedullà
(Tecnologie Informatiche)

Massimo Verola
(Informatica)

Copyright © 2005 Francesco Pedullà, Massimo Verola
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license can be found at: http://www.gnu.org/licenses/fdl.html#TOC1

Sommario

- Ciclo di vita di un programma: dall'editing all'esecuzione
- File eseguibili, file oggetto e librerie
- Mappa di memoria del processo
- Il formato ELF e **`binutils`**
- Il compilatore GNU C: **`gcc`**
- Il debugger: **`gdb`**
- Ausilio alla compilazione: **`make`**

© 2005 F. Pedullà, M. Verola

Assemblatori, compilatori, interpreti - I

- Il processore è in grado di eseguire soltanto istruzioni codificate in linguaggio macchina binario, che dipende dal processore (o dalla famiglia di processori)
- Per agevolare la codifica di programmi, sono state realizzate varie tipologie di traduttori da linguaggi ad alto livello (*codice sorgente*) al linguaggio macchina:
  - **Assemblatori** (Intel x86, IBM PowerPC)
  - **Compilatori** (C, FORTRAN, Pascal, C++)
  - **Interpreti** (Basic, Lisp, APL, PHP)
- Recentecemente sono stati introdotti linguaggi che ricadono in una categoria intermedia tra quelli compilati e interpretati, in quanto il compilatore genera un bytecode che viene poi intepretato, come per Java, Perl, Python

© 2005 F. Pedullà, M. Verola

Assemblatori, compilatori, interpreti - II

- Il funzionamento interno dei tre sistemi di traduzione è molto diverso:
  - **Assemblatore**: traduce le istruzioni da un formato testuale simbolico (assembler) nelle corrispondenti in linguaggio macchina (la corrispondenza tra le istruzioni è uno-a-uno) – il programmatore vede l'architettura interna del sistema
  - **Compilatore**: espande singole istruzioni dal liguaggio sorgente in una o più istruzioni in linguaggio macchina – il programmatore non vede l'architettura interna del sistema
  - **Interprete**: traduce passo per passo ogni istruzione dal linguaggio sorgente nelle corrispondenti istruzioni in linguaggio macchina

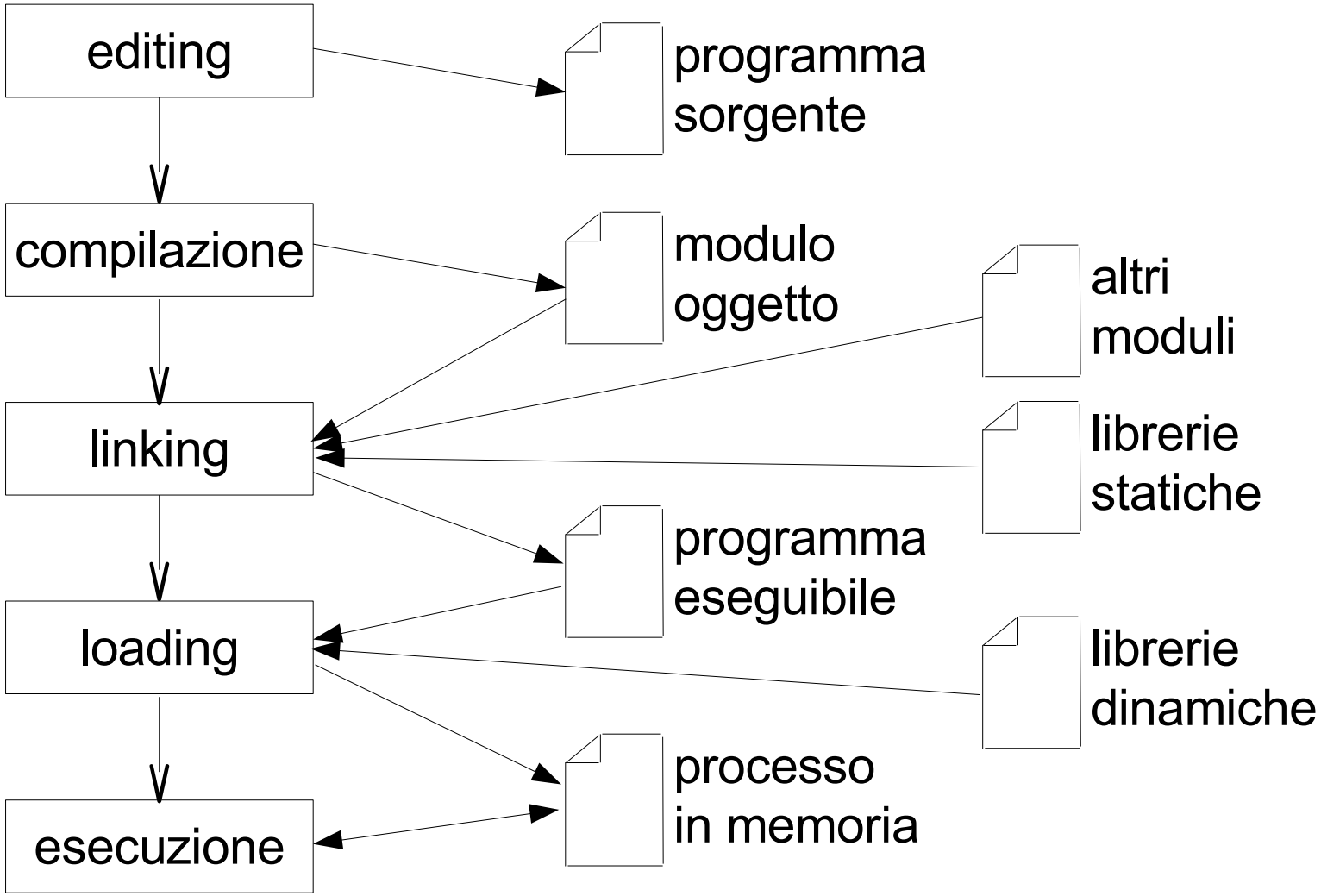
© 2005 F. Pedullà, M. Verola

Assemblatori, compilatori, interpreti - III

- L'operazione di traduzione dal sorgente al binario puo' avvenire in uno o piu' passi (2 o 3)
- I programmi interpretati hanno tempi di esecuzione molto maggiori di quelli compilati
- Per accelerare l'esecuzione dei programmi interpretati sono state introdotte tecniche di caching della traduzione e di compilazione *just-in-time*
- Tipicamente, lo sviluppo di un programma utilizzando un linguaggio interpretato è piu' semplice e rapido

© 2005 F. Pedullà, M. Verola

Fasi di realizzazione di un programma



© 2005 F. Pedullà, M. Verola

File eseguibile

- Un *file eseguibile* (o semplicemente *eseguibile*) è un file in un formato binario che il computer può direttamente eseguire.
- A differenza dei *file sorgente*, non è leggibile da un operatore umano.
- Per trasformare un file sorgente in un eseguibile, si fa ricorso a un compilatore o un assemblatore.
- Un eseguibile può contenere ulteriori informazioni oltre al codice del programma da eseguire, come ad esempio informazioni per il *debugging* e il *profiling*.
- Il codice all'interno di un eseguibile può contenere, oltre a istruzioni macchina, anche chiamate a servizi del sistema operativo (*system calls*): quindi un eseguibile è in genere dipendente sia dal processore che dal sistema operativo.

© 2005 F. Pedullà, M. Verola

File oggetto

- Un file oggetto (o *modulo oggetto*) è un file che contiene il codice generato da un compilatore dopo aver processato un file sorgente.
- Un file oggetto può essere *linkato* con altri file oggetto per generare un file eseguibile o una *libreria* (file contenitore di più oggetti)
- Oltre a codice macchina, un file oggetto contiene istruzioni di rilocazione per il linker, simboli di programma e informazioni di debugging
- Uno specifico file oggetto fa riferimento ad uno dei vari formati codificati (a.out, COFF, ELF, ...)
- I tipi di dati supportati da un file oggetto sono:
  - *BSS (Block Started by Symbol)*: strutture dati globali non inizializzate
  - *text segment*: codice binario del programma (codice macchina)
  - *data segment*: strutture dati globali inizializzate
- Diventano *segmenti* una volta caricati in memoria

© 2005 F. Pedullà, M. Verola

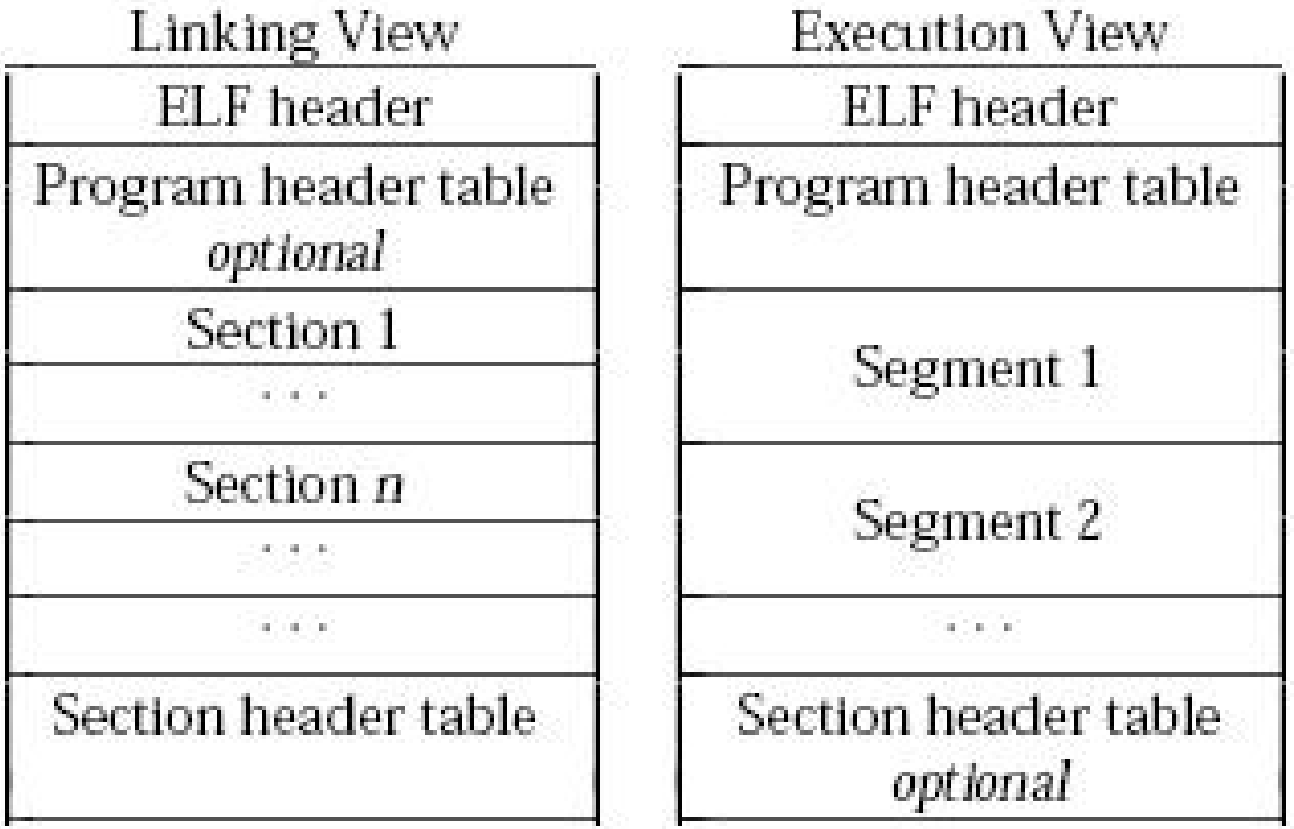
Formato ELF - I

- Executable and Linking Format (ELF): formato binario per i file eseguibili sviluppato e pubblicato da UNIX System Laboratories (USL)
- 3 tipi di file ELF:
  - *eseguibile*: codice e dati pronti per l'esecuzione
  - *rilocabile*: codice e dati pronti per il linking con altri file rilocabili e con le librerie dinamiche (*shared objects*)
  - *libreria dinamica*: codice e dati pronti per il *linker/loader dinamico* **`ld-linux.so`**
- Sono disponibili librerie e programmi di sistema (alcuni verranno citati in seguito nelle *binutils*) per manipolare ed estrarre informazioni sui file ELF

© 2005 F. Pedullà, M. Verola

Formato ELF - II

- Gli *object file* sono generati dall'*assembler* e dal *link editor* e sono rappresentazioni binarie di programmi che dovranno essere eseguiti direttamente sulla CPU
- Vengono utilizzati nella fase di linking e per l'esecuzione del programma
- Sono organizzati all'interno in header e sezioni/segmenti (a seconda se si l'object file e' su disco – linking view – o in esecuzione – execution view)



© 2005 F. Pedullà, M. Verola

Formato ELF - III

- I file ELF contengono diverse sezioni, tra cui:
  - **.bss:** dati non inizializzati che contribuisco all'immagine in memoria del programma
  - **.comment**: informazioni sul controllo di versione
  - **.data:** dati inizializzati che contribuisco all'immagine in memoria del programma
  - **.debug**: informazioni simboliche per debugging
  - **.dynamic**: informazioni per il linking dinamico
  - **.fini**: codice da eseguire dopo il codice utente
  - **.init**: codice da eseguire prima del codice utente
  - **.rel***name*: informazioni per la rilocazione
  - **.strtab**: stringhe associate con la *symbol table*
  - **.symtab:** symbol table
  - **.text:** istruzione eseguibili del programma

© 2005 F. Pedullà, M. Verola

Formato ELF - IV

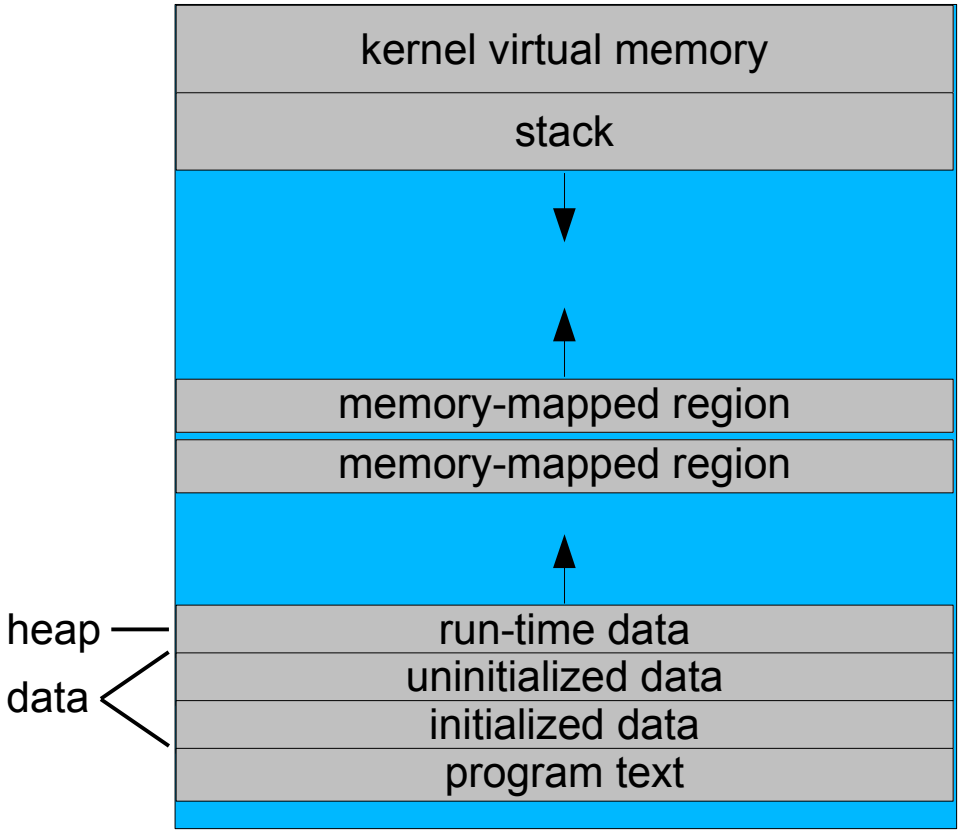
- L' *ELF header* risiede all'inizio del file e descrive l'organizzazione del file
- Il *program header*, presente soltanto per i file eseguibili, descrive come creare una *process image* in memoria
- Le sezioni contengono i dati centrali dell'*object file* (istruzioni, dati, symbol table, informazioni per la rilocazione)
- Esiste una *section header table* che permette di localizzare tutti le sezioni nel file
- Ogni sezione ha un *section header*
- Ogni sezione occupa una regione contigua di byte nel file e non puo' andare in overlap con la sezione successiva
- Ci possono essere degli spazi vuoti nel file (*inactive space*)
- Il formato completo del file e' definito in **`/usr/include/elf.h`**
- Il formato e' descritto in **`man elf`**

© 2005 F. Pedullà, M. Verola

Mappatura di file eseguibili nella memoria

- Il loader di Linux non carica i programmi nella memoria fisica, ma piuttosto esegue una mappatura dell'eseguibile sulla memoria virtuale
- Soltanto quando il programma inizia l'esecuzione e tenta di accedere ad una data pagina di memoria, il kernel rileva un *page fault* e procede all'effettivo caricamento della pagina nella memoria fisica

Configurazione in memoria virtuale dei vari segmenti per eseguibili in formato ELF



© 2005 F. Pedullà, M. Verola

binutils

Le binutils sono una collezione di programmi per la gestione dei file oggetto ed eseguibili.

- **ar**: Create, modify, and extract from archives
- **nm**: List symbols from object files
- **objcopy**: Copy and translate object files
- **objdump**: Display information from object files
- **ranlib**: Generate index to archive contents
- **readelf**: Display the contents of ELF format files.
- **size**: List section sizes and total size
- **strings**: List printable strings from files
- **strip**: Discard symbols
- **c++filt**: Filter to demangle encoded C++ symbols
- **cxxfilt**: MS-DOS name for c++filt
- **addr2line**: Convert addresses to file and line
- **nlmconv**: Converts object code into an NLM
- **windres**: Manipulate Windows resources
- **dlltool**: Create files needed to build and use DLLs

© 2005 F. Pedullà, M. Verola

Librerie statiche

- Contenitori di file oggetto
- Per convenzione il nome comincia con **`lib`**.
- Per convenzione usano l'estensione **`.a`**
- Create con il comando **`ar`**:
  `ar rcs my_library.a file1.o file2.o`
- Gradualmente sostituite dalle librerie dinamiche ma ancora utili ed usate, specialmente per sviluppi applicativi
- Si linkano come un normale file oggetto, tramite due opzioni di **`gcc:`**
  - **`-l`**, che in realta' viene passata a **`ld`**, per cui va specificata dopo i file da compilare
  - **`-L`** che permette di specificare una directory in cui cercare le librerie

© 2005 F. Pedullà, M. Verola

Librerie dinamiche - I

- Contenitori di file oggetto
- Per convenzione usano l'estensione **`.so`**
- I loro oggetti non vengono inseriti all'interno di file eseguibili, ma:
  - gli oggetti vengono caricati in memoria solo quando servono e possono essere condivisi tra piu' applicazioni
  - risulta sufficiente aggiornare la libreria senza toccare gli eseguibili, molto comodo per le librerie di sistema
- Gli oggetti contenuti devono essere rilocabili (*position-independent*)
- Create dal compilatore o dal linker/loader:

```
gcc -fPIC -g -c -Wall a.c
gcc -fPIC -g -c -Wall b.c
gcc -shared -Wl,-soname,libmystuff.so.1 -o
     libmystuff.so.1.0.1 a.o b.o -lc
```


© 2005 F. Pedullà, M. Verola

Librerie dinamiche - II

- In Linux usano diversi nomi:
  - *realname* (comprende versione e release): **`/usr/lib/libreadline.so.3.0`**
  - *soname* (comprende solo la versione, spesso un link simbolico al realname creato da **`ldconfig`**): **`/usr/lib/libreadline.so.3`**
  - *linker name* (usato dal linker, di solito un link simbolico al *soname*, da creare manualmente): **`/usr/lib/libreadline.so`**
- Il loader **`/lib/ld.linux.so*`** si fara' carico di portare in memoria gli oggetti necessari al momento dell'esecuzione, estraendoli dalle librerie che si aspetta di trovare nelle *cache* creata da **`ldconfig`**, a sua volta caricata sulla base delle directory elencate in **`/etc/ld.so.conf`**.

© 2005 F. Pedullà, M. Verola

Utilizzo di alcune binutils - I

- Creare un file **`main.c`** con il sorgente C dell'esempio in basso
- Per generare un file eseguibile compilarlo con
  `gcc main.c -o main`

```
#include "stdio.h"
#include "stdlib.h"
int myGlob=10;
int main(int argc, char **argv)
{
   int myLocal=3;
   printf("Hello\n");
   exit(0);
}
```


© 2005 F. Pedullà, M. Verola

Utilizzo di alcune binutils - II

- Con **size** possiamo vedere le dimensioni dei diversi segmenti:

```
penguin@antarctic:~/binutils> size main.o
   text    data     bss     dec     hex filename
    899     268       4    1171     493 main
```

- Con **strings** possiamo vedere le stringhe memorizzate nel file:

```
penguin@antarctic:~/binutils> strings main
/lib/ld-linux.so.2
...
libc.so.6
...
GLIBC_2.0
PTRh@
Hello
```


© 2005 F. Pedullà, M. Verola

Utilizzo di alcune binutils - III

- Con **nm** possiamo vedere tutti i simboli nel file (nell'esempio ci limitiamo ai simboli undefined e global):

```
penguin@antarctic:~/binutils> nm -u -g main
         U exit@@GLIBC_2.0
         w __gmon_start__
         w _Jv_RegisterClasses
         U __libc_start_main@@GLIBC_2.0
         U puts@@GLIBC_2.0
```

- Significato di alcuni tag che definiscono il tipo di simbolo:
  B (BSS): simbolo nella sezione dati non inizializzati
  D (data): simbolo nella sezione dati inizializzati
  N: il simbolo serve per il debug
  R (readonly): il simbolo sta in una sezione dati readonly
  T (text): il simbolo sta nella sezione text (codice)
  U (*undefined):* il simbolo non e' definito all'interno del file (deve esser cercato in una libreria esterna)
  W (weak): per il simbolo esiste un default di sistema se non puo' essere risolto

© 2005 F. Pedullà, M. Verola

Utilizzo di alcune binutils - IV

Con **readelf** possiamo ottenere informazioni molto dettagliate sul file oggetto:

```
penguin@antarctic:~/binutils> readelf -h main
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 ...
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  ...
```


© 2005 F. Pedullà, M. Verola

Il compilatore gcc

```
#include "stdio.h"
int main(int argc, char **argv)
{
   printf("Hello\n");
}
```

- Per compilare questo programma con **gcc** e generare un eseguibile:

```
gcc hello.c -o hello
```

- Oppure si può procedere in 2 passi, prima si genera il file oggetto e poi si effettua il *linking* per generare l'eseguibile:

```
gcc -c hello.c
gcc hello.o -o hello
```


© 2005 F. Pedullà, M. Verola

Le opzioni di gcc

- Per istruire il compilatore ad inserire informazioni simbolici necessarie al debugger **gdb** con **gcc** e generare un eseguibile:

```
gcc -g hello.c -o hello
```

- Per visualizzare molti “warning” sulle istruzioni sintatticamente corrette, ma sospette:

```
gcc -Wall hello.c -o hello
```

- Per generare codice binario ottimizzato:

```
gcc -O hello.c -o hello
```

- Per linkare la libreria dinamica “libmy.so” ed indicare anche il path della sua directory **/usr/local/myLibs/**:

```
gcc hello.c -o hello -lmy -L/usr/local/myLibs
```

- Per indicare al preprocessore dove cercare gli include file:

```
gcc hello.c -I/usr/local/myInc -o hello
```


© 2005 F. Pedullà, M. Verola

Compilare file multipli

- Per compilare più sorgenti e generare un eseguibile:

```
gcc file1.c file2.c -o myprog
```

- Quando il numero di file sorgente aumenta, è opportuno compilare i file separatamente e poi linkarli. In tal modo, se un sorgente viene modificato, basta ricompilare soltanto quel sorgente e linkare di nuovo l'eseguibile, senza dover ricompilare tutti gli altri sorgenti:

```
gcc -c file1.c
gcc -c file2.c
gcc file1.o file2.o -o myprog
vi file2.c
```

... editing e modifiche di file2.c ...

```
gcc -c file2.c
gcc file1.o file2.o -o myprog
```

Nota: file1.c non viene ricompilato, non essendo stato modificato. Tale tecnica verrà automatizzato con l'utilizzo del programma **`make`** (trattato in seguito),

© 2005 F. Pedullà, M. Verola

Tabella riepilogativa

Input file	*Output file*	*Azione*	*Comando*
Sorgente .c	File oggetto .o	Preprocessing + compiling + assembling	gcc -c
Sorgente .c	Preprocessato .i	Preprocessing	gcc -E oppure cpp
Preprocessato .i	File assembler .s	Compiling	gcc -S
File assembler .s	File oggetto .o	Assembling	gcc -c oppure as
Files oggetto .o	Libreria statica .a	Creazione libreria	ar
Files oggetto .o	Libreria dinamica .so	Creazione libreria	gcc -shared
Files oggetto .o + libreria statica + libreria dinamica	Eseguibile	Linking	gcc -o oppure ld
Eseguibile	Processo	Loading	ld-linux.so

© 2005 F. Pedullà, M. Verola

Il debugger gdb

- Un debugger è un programma che permette all'utente di controllare completamente l'esecuzione di un altro programma e di esaminare il valore delle variabili
- Un debugger viene chiamato in causa generalmente come ausilio per la risoluzione delle seguenti questioni:
  - Su quale istruzione o espressione il programma ha dato errore?
  - Quale linea di programma contiene la chiamata alla funzione in cui si è verificato l'errore e quali erano i valori dei parametri passati?
  - Quali sono i valori delle variabili ad un certo punto dell'esecuzione del programma?
  - Qual è il risultato di una particolare espressione nel programma?
- Il debugger più diffuso in ambiente Linux è **gdb**, il debugger della GNU

© 2005 F. Pedullà, M. Verola

Come utilizzare gdb

- Bisogna prima di tutto informare il compilare che si intende creare una versione dell'eseguibile adatta al debugging. Si utilizza il flag **-g** nel comando **gcc** di compilazione del sorgente:

```
gcc -g trees.c -o trees
```

- Si lancia il debugger indicando come file di input l'eseguibile:

```
gdb trees
```

- A questo punto si può interagire con l'interfaccia a linea di comando di gdb. Esiste una serie di comandi per definire breakpoint, lanciare il programma, visualizzare i valore di variabili, procedere passo passo nell'esecuzione

- Esistono anche una versione dotata di interfaccia grafica: **xgdb**

© 2005 F. Pedullà, M. Verola

I comandi gdb - I

- **`run command-line-arguments`**

Lanciare il programma nel modo usuale, premettendo **`run`**

- **`break place`**

Crea un *breakpoint*; il programma si ferma quando l'esecuzione arriva al breakpoint. E' spesso comodo mettere un breakpoint all'inizio di una funzione da analizzare:

```
(gdb) break myFunc
Breakpoint 2 at 0x2290: file main.c, line 20
```

Il comando break permette di fermare l'esecuzione all'inizio della funzione. Si può anche mettere un breakpoint su di una linea di codice del file sorgente:

```
(gdb) break 20
Breakpoint 2 at 0x2290: file main.c, line 20
```

Lanciando il programma, quando si raggiunge il breakpoint si ha un messaggio del tipo:

```
Breakpoint 1, myFunc (head=0x6110, NumNodes=4)
  at main.c:16
```


© 2005 F. Pedullà, M. Verola

I comandi gdb - II

- **`delete N`**

Cancella il breakpoint numero N. Se si omette N, verranno cancellati tutti i breakpoint. Utilizzare **`info`** per visualizzare informazioni sui breakpoint.

- **`help command`**

Fornisce una breve descrizione di un comando GDB o di un argomento.

- **`step`**

Esegue la linea corrente di programma e si ferma alla prossima istruzione.

- **`next`**

Simile a **`step`**, tuttavia se la linea corrente è una chiamata ad una funzione, esegue l'intero codice della funzione e si ferma alla linea seguente (cioè non "entra" nel codice della funzione, come **`step`**).

- **`list`**

Mostra alcune righe di codice intorno alla linea corrente.

© 2005 F. Pedullà, M. Verola

I comandi gdb - III

- **`finish`**

Continua ad eseguire il comando **`next`** fino alla fine della funzione corrente

- **`continue`**

Continua l'esecuzione regolare del programma fino a prossimo breakpoint

- **`where`**

Visualizza la gerarchia di chiamate delle funzioni che hanno portato il programma all'istruzione corrente.

- **`print E`**

Stampa il valore di E, dove E è un'espressione in C (di solito è soltanto una variabile). Il comando **`display`** è simile, eccetto che ogni volta che si esegue un comando **`next`** o **`step`**, ristampa il valore

- **`quit`**

Esce da **`gdb`**.

© 2005 F. Pedullà, M. Verola

Ausilio alla compilazione: l'utility `make` (I)

- **Il problema:**
  - Programmi di grandi dimensioni non possono essere contenuti in un file singolo, ma devono essere organizzati e suddivisi in vari file.
  - Nel linguaggio C, ad esempio, un programma grande è organizzato su numerosi file:
    - i file *.h contengono le funzioni prototipo ed eventuali costanti
    - i file *.c contengono le definizioni di funzioni
    - i file *.c contengono direttive per il preprocessore di inclusione dei file *.h di pertinenza
    - il file *.c vengono preprocessati e compilati separatamente per generare dei file oggetto
    - viene generato un eseguibile (o una libreria) tramite linking dei file oggetto

© 2005 F. Pedullà, M. Verola

Ausilio alla compilazione: l'utility `make` (II)

- **Lo strumento:**
  - L'utility **`make`** permette di gestire automaticamente la compilazione ed il linking dei file sorgenti di un programma
  - Tiene traccia delle relazioni tra i moduli che compongono il programma
  - Esegue solo i comandi necessari alla ricostruzione del file eseguibile in funzione dei sorgenti che sono stati modificati dopo l'ultima compilazione
  - **`make`** utilizza un file di configurazione denominato ***`makefile`***
  - Il nome di default utilizzato da make per il makefile e' **`GNUmakefile`** o **`makefile`** o **`Makefile`**
  - L'utility **`make`** puo' essere utilizzata non solo per la compilazione ma in generale per qualunque operazione che dipenda dalla disponibilita' di file nuovi o modificati
  - Esiste una versione parallela chiamata **`pmake`**

© 2005 F. Pedullà, M. Verola

Esempio (1): il *main*

```
/* sample.c */
#include <stdio.h>
#include "my_math.h"
int main()
{
   int a, b, c;
   puts("Input three numbers:");
   scanf("%d %d %d", &a, &b, &c);
   printf("The average of %d %d %d is %f.\n",
          a,b,c,average(a,b,c));
   return 0;
}
```


© 2005 F. Pedullà, M. Verola

Esempio (2): i file *.h e *.c

```
/* my_math.h */
#define PI 3.1415926
float average(int x,
   int y, int z);
float sum(int x,
   int y, int z);
```

```
/* my_math.c */
#include "my_math.h"
float average(int x, int y,
   int z)
{
   return sum(x,y,z)/3;
}

float sum(int x,
   int y, int z)
{
   return x+y+z;
}
```


© 2005 F. Pedullà, M. Verola

Come ragiona il `make`: le dipendenze

- **Per generare `my_math.o`**
  - Abbiamo bisogno di **`my_math.c`** e **`my_math.h`**
  - **`gcc -c my_math.c`**
- **Per generare `sample.o`**
  - Abbiamo bisogno di **`sample.c`** e **`my_math.h`**
  - **`gcc -c sample.c`**
- **Per generare l'eseguibile**
  - Abbiamo bisogno di **`my_math.o`** e **`sample.o`**
  - **`gcc -o sample sample.o my_math.o`**

© 2005 F. Pedullà, M. Verola

La struttura del *makefile* (I)

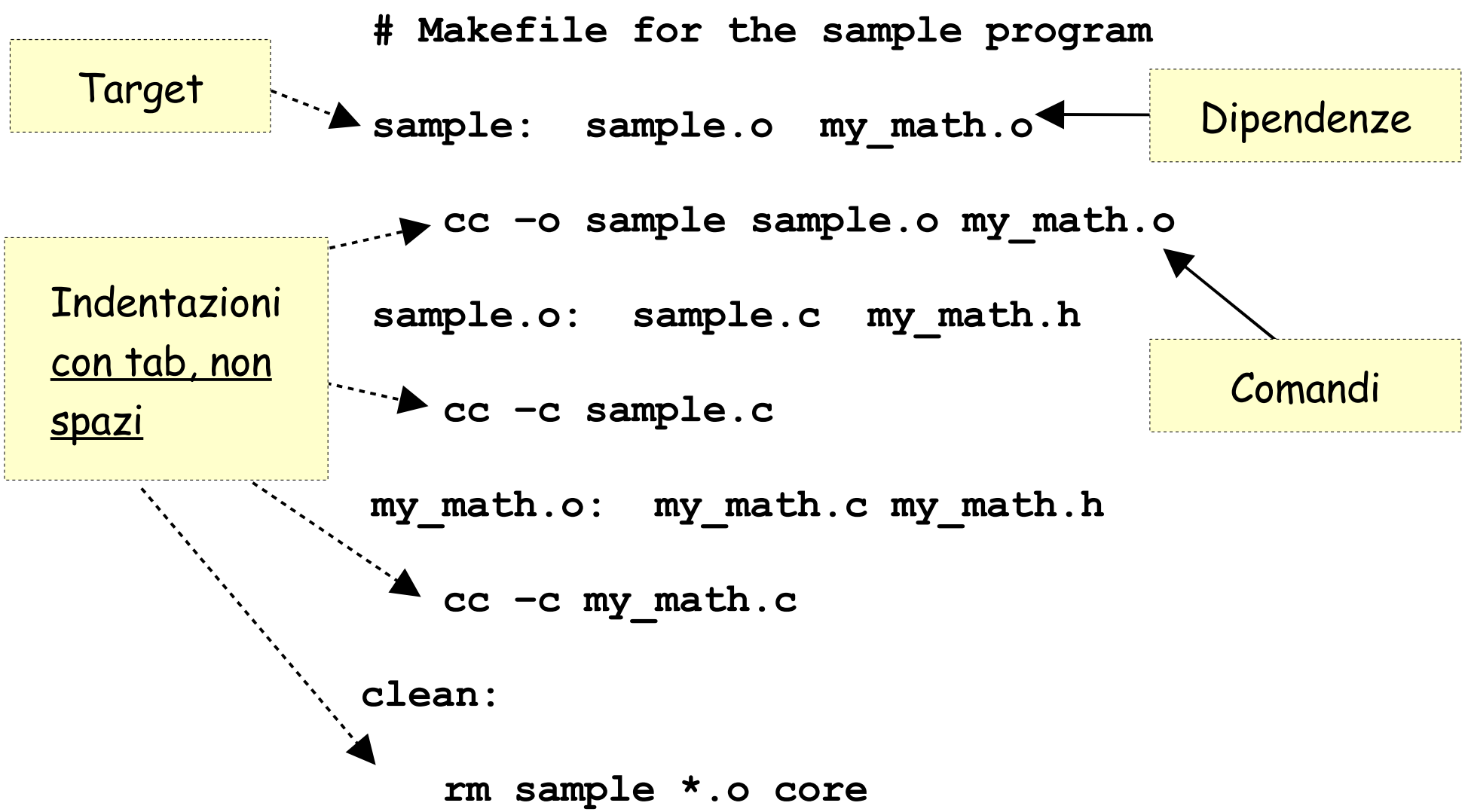
- **Un makefile consiste in un insieme di regole del tipo:**

```
target ... : prerequisiti ...
        regola1
        regola2
        ...
```

- **Target:**
  - nome del file da generare
  - nome simbolico indicante un'azione da svolgere
- **Prerequisiti:**
  - i file utilizzati come input devono essere già esistenti e le altre azioni definite dal makefile devono essere già completate prima di passare ad eseguire le regole
- **Regole**
  - comandi da eseguire per svolgere le varie attività necessarie al raggiungimento o generazione del target
- **Da dove parte il motore logico di analisi del make:**
  - la prima regola incontrata nel makefile è quella di *default*, o *radice*

© 2005 F. Pedullà, M. Verola

La struttura del *makefile* (II)



© 2005 F. Pedullà, M. Verola

Come utilizzare `make`

- Si scrive il **`makefile`** codificando opportunamente target, dipendenze e regole per la compilazione ordinata del programma
- Si lancia l'utility **`make`**
- **`make`** verifica le regole e le dipendenze, identifica la “regola radice” da cui partire e rigenera in modo ordinato, secondo le dipendenze codificate, i file target per i quali è necessario un aggiornamento, cioè quelli con data antecedente ad almeno una delle proprie dipendenze

© 2005 F. Pedullà, M. Verola

Esempio (I)

- **`sample`** è il target da generare, quindi si parte da qui per analizzare tutta la gerarchia delle dipendenze
- **`sample.o`** esiste ma dipende da **`sample.c`** che è più recente, quindi va rigenerato applicando la regola: **`cc -c sample.c`**
- **`my_math.o`** esiste e non deve essere rigenerato in quanto più recente delle sue dipendenze **`my_math.c`** e **`my_math.h`**
- A questo punto **`sample`** risulta più vecchio di **`sample.o`** quindi va rigenerato: **`cc -o sample sample.o my_math.o`**

© 2005 F. Pedullà, M. Verola

Esempio (II)

```
% ls -l
... Sep  9 19:50 sample.c
... Sep  9 19:46 sample
... Sep  9 19:43 sample.o
... Sep  9 19:24 my_math.o
... Sep  9 19:20 my_math.c
... Sep  9 19:05 my_math.h
```

```
# Makefile for the sample program
sample:  sample.o  my_math.o
    cc -o sample sample.o my_math.o
sample.o:  sample.c  my_math.h
    cc -c sample.c
my_math.o:  my_math.c my_math.h
    cc -c my_math.c
clean:
    rm sample *.o core
```


© 2005 F. Pedullà, M. Verola

Ricompilazione e *cleanup*

- E' bene inserire una regola per rimuovere i file che possono essere rigenerati, quelli di backup dell'editor, i *core* file

```
clean:
    rm sample *.o core
```

```
$ make clean
```

- Per ricompilare tutto il progetto:
  - Rimuovere tutti i file generati e lanciare il make:
    - **`make clean ; make`**
  - Oppure è possibile cambiare la data di ultima modifica di un file da cui dipendono tutti (direttamente o indirettamente):
    - **`touch my_math.h ; make`**
    - Nota: il comando **`touch my_math.h`** fa prendere l'ora corrente al file

© 2005 F. Pedullà, M. Verola

Utilizzo delle variabili nei makefile

- **Variabili nei makefile**
  - Utilizzate per semplificare la modifica dei Makefile
  - Utilizzate per ridurre la dimensione dei file semplificando espressioni ripetute
- **Sintassi**
  - Definizione: `name = value`
  - Riferimenti: `$(name)` o `${name}`
- **Esempi**

```
CC      = gcc
HDIR    = include
```


© 2005 F. Pedullà, M. Verola

Variabili predefinite e automatiche

- Alcuni esempi di variabili predefinite:
  - **`CC`** specifica il compilatore C da utilizzare (default: **`cc`** o **`gcc`**)
  - **`CFLAGS`**: opzioni di default da passare al compilatore
- Alcuni esempi di variabili automatiche:
  - **$@** nome del target della regola corrente
  - **$?** lista delle dipendenze all'interno della regola corrente che hanno una data piu' recente del target
  - **$^** lista di tutte le dipendenze all'interno della regola corrente
  - N.B.: si possono passare variabili anche al momento dell'invocazione: **`make 'OBJECTS=file1.o file2.o'`**

© 2005 F. Pedullà, M. Verola

Un esempio di makefile con macro

```
CC      = gcc
HDIR    = include
INCPATH = -I$(HDIR)
DEPH    =  $(HDIR)/queue_types.h \
           $(HDIR)/stack_types.h \
OBJECTS = stack.o queue.o

main: main.o $(OBJECTS)
    $(CC) -o main main.o $(OBJECTS)
main.o: main.c $(DEPH)
    $(CC) $(INCPATH) -c main.c
stack.o: stack.c $(DEPH)
    $(CC) $(INCPATH) -c stack.c
queue.o: queue.c $(DEPH)
    $(CC) $(INCPATH) -c queue.c
clean:
    rm  -f  *.o  main  core
```


© 2005 F. Pedullà, M. Verola

Makefile "avanzati"

- **Le funzionalità di make non si esauriscono qui**
  - Strutture di controllo come statement condizionali e loop
  - Dipendenze automatiche e regole implicite che agiscono come default quando non sono presenti regole esplicite
  - Semplici funzioni di supporto per trasformare testo
  - Variabili automatiche che si riferiscono a vari elementi di un makefile, come target e dipendenze.

© 2005 F. Pedullà, M. Verola

Dipendenze automatiche

- **make** è in grado di inferire alcune dipendenze e regole, basandosi su di un database interno di regole standard

```
OBJECTS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(OBJECTS)
        cc -o edit $(OBJECTS)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h
```

Dipendenza automatica da main.c
Regola implicita:
    gcc -c main.c

© 2005 F. Pedullà, M. Verola

Dipendenze automatiche

- Uno stile alternativo: è possibile "raggruppare" le dipendenze

```
OBJECTS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(OBJECTS)
        cc -o edit $(OBJECTS)

$(OBJECTS) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```


© 2005 F. Pedullà, M. Verola

Regole "statiche"

- Specificano il comportamento per una classe di file

```
.PHONY: all eps png
all: eps png
eps: quota.eps transfer.eps single.eps
png: quota.png transfer.png single.png

%.eps : %.plt seteps.gpt
    gnuplot seteps.gpt $< ; mv output.eps $@

%.png : %.plt seteps.gpt
    gnuplot setpng.gpt $< ; mv output.png $@

clean:
    rm -f *.eps *.png
```

Afferma che sono solo azioni, non file

Identifica il prerequisito

Identifica il target

© 2005 F. Pedullà, M. Verola

Alcune opzioni di make

- **`make -f` *`file`***: utilizza ***`file`*** come makefile
- **`make -n`**: stampa i comandi che verrebbero eseguito, senza pero' eseguirli realmente
- **`make -d`**: stampa informazioni di debug, oltre a quelle standard
- **`make -p`**: stampa il database (regole e valori delle variabili) a seguito del processamente del **`makefile`**; vengono anche stampate tutte le variabili e regole predefinite

© 2005 F. Pedullà, M. Verola

Documentazione

- http://www.gnu.org/software/make/manual/make.html
- *The GNU Binary Utilities* di R. Pesch e J. Osier, Free Software Foundation
- *Debugging with gdb* di R. Stallman, R. Pesch et al, Free Software Foundation
- *The ELF Object File Format: Introduction* di E. Youngdale, Linux Journal
- *ELF: from the Programmer's Perspective* di Hongjiu Lu
- *Program Library HOWTO* di D. Wheeler

© 2005 F. Pedullà, M. Verola