Capitolo 65

Compilazione e formato binario eseguibile

65.1 Compilazione di programmi composti da più file sorgenti

«

Per poter compilare un programma distribuito tra più file sorgenti, all'interno di questi file occorre dichiarare quali simboli (riferiti a variabili e funzioni) devono essere pubblici e come tali accessibili anche dagli altri; inoltre, nei file in cui si fa riferimento a simboli esterni, occorre dichiarare questa dipendenza.

65.1.1 Inclusione di file

«

Prima di affrontare il problema del collegamento di più file oggetto in un file eseguibile singolo, conviene considerare l'inclusione automatica del contenuto di un file. In altri termini, si può ottenere una funzione simile al «copia-incolla», dichiarando in un file che, in un certo punto, va incluso il contenuto di un altro. Per esempio, per incorporare in un certo punto, il contenuto del file '`funzioni.s`', occorre scrivere la direttiva seguente:

```
…
; GNU AS
.include "funzioni.s"
…
```

```
…
; NASM
%include "funzioni.s"
…
```

Naturalmente, il file '`funzioni.s`' contiene qualcosa che si può copiare e incollare, tale e quale, in quel certo punto del sorgente che

vi fa riferimento.

Il file in questione viene cercato nella directory corrente, ma nella riga di comando di GNU AS è possibile aggiungere l'opzione '**-I**' per indicare altri percorsi di ricerca; nello stesso modo, con NASM si può usare l'opzione '**-i**':

```
$ as -I directory ... [Invio]

$ nasm -i directory ... [Invio]
```

65.1.2 Due file sorgenti da collegare assieme

Per dimostrare come si gestiscono più file sorgenti assieme, viene mostrato un esempio molto semplice, composto da due soli file: nel primo si trova la funzione '**_start**'; nel secondo si trova la funzione '**f_rs**' che esegue una ricerca sequenziale all'interno di un array (nella sezione 64.14.1 si vede lo stesso programma, tutto intero, in un file unico). I due listati seguenti sono realizzati per GNU AS:

```
# rs-main.s
#
.section .data
lista: .int 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 # Interi senza
                                               # segno.
a:     .int 0                          # Indice minimo.
z:     .int 9                          # Indice massimo.
#
.section .text
.globl _start
.extern f_rs
#
_start:
    push  z           # f_rs ($lista, $7, a, z) ==> EAX
```

```
    push  a           # Si cerca il valore 7 nell'array
    push  $7          # «lista», tra gli indici «a» e «z».
    push  $lista      # Viene restituito l'indice
    call  f_rs        # dell'elemento trovato, oppure -1 se
    add   $16, %esp   # non è presente.
bp1:
    mov   %eax, %ebx # Restituisce l'indice trovato,
    mov   $1, %eax   # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
```

```
# rs-f.s
#
.section .data
#
.section .text
.globl f_rs
#
# Ricerca sequenziale all'interno di una lista di valori.
# f_rs (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rs:
    enter $4, $0
    pusha
    .equ  rs_i,     -4          # Gli si associa EAX.
    .equ  rs_lista,  8          # Gli si associa ESI.
    .equ  rs_x,     12          # Gli si associa EDX.
    .equ  rs_a,     16
    .equ  rs_z,     20
    #
    mov   rs_lista(%ebp), %esi # ESI contiene l'indirizzo
                               # dell'array.
```

```
    mov   rs_x(%ebp),     %edx # EDX contiene il valore
                               # cercato.
    #
    mov   rs_a(%ebp),     %eax # EAX viene usato come indice
                               # di scansione.
f_rs_loop:
    cmp   rs_z(%ebp),     %eax # Se EAX è maggiore
    ja    f_rs_non_trovato     # dell'indice massimo,
                               # l'elemento cercato non c'è.
    #
    cmp   (%esi,%eax,4),  %edx # Se il valore cercato
    je    f_rs_trovato         # corrisponde a quello
                               # dell'indice corrente,
                               # termina la scansione.
    #
    inc   %eax                 # Incrementa l'indice di
    jmp   f_rs_loop            # scansione e salta
                               # all'inizio del ciclo.
    #
f_rs_non_trovato:
    popa                # Conclude la funzione con EAX = -1.
    mov $-1, %eax       #
    leave               #
    ret                 #
f_rs_trovato:
    mov %eax, rs_i(%ebp)   # Salva EAX nella variabile
                           # locale prevista.
    popa                   # Conclude la funzione con EAX
    mov rs_i(%ebp), %eax   # pari al valore salvato nella
    leave                  # variabile locale.
    ret                    #
```

Nel primo dei due listati, corrispondente al file '`rs-main.s`', si deve osservare la dichiarazione esterna del simbolo '**`f_rs`**', cor-

rispondente al nome della funzione contenuta nel file ‘rs-f.s’.

```
…
.section .text
.globl _start
.extern f_rs
…
```

Dal momento che i dati necessari all’elaborazione vengono passati alla funzione attraverso i parametri della chiamata, a parte ‘**_start**’, non ci sono altre dichiarazioni di simboli pubblici nel file ‘f-main.s’. Nel secondo listato, corrispondente al file ‘rs-f.s’, il simbolo ‘**f_rs**’ viene reso pubblico, per consentire al file ‘rs-main.s’ di farvi riferimento.

```
…
.section .text
.globl f_rs
…
```

Seguono gli stessi due listati, nella versione adatta a NASM:

```
; rs-main.s
;
section .data
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi senza
                                            ; segno.
a:     dd 0                              ; Indice minimo.
z:     dd 9                              ; Indice massimo.
;
section .text
global _start
extern f_rs
```

```
;
_start:
    push  long [z] ; f_rs ($lista, $7, a, z) ==> EAX
    push  long [a] ; Si cerca il valore 7 nell'array
    push  long 7   ; «lista», tra gli indici «a» e «z».
    push  lista    ; Viene restituito l'indice dell'elemento
    call  f_rs     ; trovato, oppure -1 se non è presente.
    add   esp, 16  ;
bp1:
    mov   ebx, eax ; Restituisce l'indice trovato,
    mov   eax, 1   ; ammesso che sia abbastanza piccolo
    int   0x80     ; da poter essere rappresentato come
                   ; valore di uscita.
```

```
; rs-f.s
;
section .data
;
section .text
global f_rs
;
; Ricerca sequenziale all'interno di una lista di valori.
; f_rs (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
f_rs:
    enter 4, 0
    pusha
    rs_i     equ -4              ; Gli si associa EAX.
    rs_lista equ  8              ; Gli si associa ESI.
    rs_x     equ 12              ; Gli si associa EDX.
    rs_a     equ 16
    rs_z     equ 20
```

```
    ;
    mov   esi, [rs_lista+ebp] ; ESI contiene l'indirizzo
                              ; dell'array.
    mov   edx, [rs_x+ebp]     ; EDX contiene il valore
                              ; cercato.
    ;
    mov   eax, [rs_a+ebp]     ; EAX viene usato come indice
                              ; di scansione.
f_rs_loop:
    cmp   eax, [rs_z+ebp]     ; Se EAX è maggiore dell'indice
    ja    f_rs_non_trovato    ; massimo, l'elemento cercato
                              ; non c'è.
    ;
    cmp   edx, [esi+eax*4]    ; Se il valore cercato
    je    f_rs_trovato        ; corrisponde a quello
                              ; dell'indice corrente,
                              ; termina la scansione.
    ;
    inc   eax                 ; Incrementa l'indice di
    jmp   f_rs_loop           ; scansione e salta all'inizio
                              ; del ciclo.
    ;
f_rs_non_trovato:
    popa                ; Conclude la funzione con EAX = -1.
    mov eax, -1         ;
    leave               ;
    ret                 ;
f_rs_trovato:
    mov [rs_i+ebp], eax ; Salva EAX nella variabile
                        ; locale prevista.
    popa                ; Conclude la funzione con EAX pari
    mov eax, [rs_i+ebp] ; al valore salvato nella variabile
    leave               ; locale.
    ret                 ;
```

In questo caso, le direttive salienti sono, rispettivamente:

```
…
section .text
global _start
extern f_rs
…
```

```
…
section .text
global f_rs
…
```

Per compilare il tutto in un solo file eseguibile, occorre procedere secondo i comandi seguenti. Nel caso di GNU AS:

```
$ as --gstabs -o rs-main.o rs-main.s [Invio]

$ as --gstabs -o rs-f.o rs-f.s [Invio]

$ ld -o rs rs-main.o rs-f.o [Invio]
```

Nel caso di NASM:

```
$ nasm -g -f elf -o rs-main.o rs-main.s [Invio]

$ nasm -g -f elf -o rs-f.o rs-f.s [Invio]

$ ld -o rs rs-main.o rs-f.o [Invio]
```

Questo programma restituisce l'indice dell'elemento cercato e trovato nell'array. In questo caso, si tratta del quarto elemento che corrisponde all'indice 3:

```
$ ./rs ; echo $? [Invio]
```

3

65.1.3 Incorporazione di codice in linguaggio C

«

Per collegare assieme sorgenti scritti in linguaggi differenti, si agisce in modo analogo a quanto già mostrato per il solo linguaggio assemblatore. C'è però da considerare che ogni compilatore ha le proprie caratteristiche, sia per ciò che riguarda le convenzioni di chiamata delle funzioni, sia per il modo di nominare i simboli associati alle funzioni stesse. Nel caso di GCC (*GNU compiler collection*), valgono le convenzioni di chiamata comuni e i nomi delle funzioni non vengono modificati.

Qui si mostra un listato, in linguaggio C, da usare in sostituzione del file '`rs-f.s`' descritto nella sezione precedente:

```
/* f_rs (<lista>, <x>, <ele-inf>, <ele-sup>) */

int f_rs (int lista[], int x, int a, int z)
{
    int i;

    /* Scandisce l'array alla ricerca dell'elemento. */

    for (i = a; i <= z; i++)
      {
        if (x == lista[i])
          {
            return i;
          }
      }

    /* La corrispondenza non è stata trovata. */
```

```
    return -1;
}
```

Per compilare questo file e generare un file oggetto, ammesso che il sorgente si chiami ‘`rs-f.c`’, si procede con il comando seguente:

```
$ cc -c -o rs-f.o rs-f.c[Invio]
```

Il collegamento con il file ‘`rs-main.o`’ avviene nel modo già visto:

```
$ ld -o rs rs-main.o rs-f.o[Invio]
```

65.2 Librerie dinamiche e librerie statiche

La compilazione dei programmi, secondo quanto descritto in precedenza, genera sempre file eseguibili «completi», in quanto incorporano tutto il codice necessario al proprio funzionamento. Oltre che suddividere il sorgente in file separati, da riunire assieme in un file eseguibile unico, è possibile costruire una libreria di funzioni, a cui i programmi accedono dopo essere stati avviati, senza incorporarne il codice. Un libreria di questo genere è nota come ***libreria dinamica***, in quanto richiede la creazione di un «collegamento» (*link*) istantaneo, mentre il programma che la richiede è in funzione.

Il concetto di libreria dinamica si contrappone a quello di ***libreria statica***, la quale comporta l’inclusione del proprio codice nel file eseguibile, in fase di compilazione.

65.2.1 Il processo di «collegamento» dinamico

«

Il programma eseguibile che ha bisogno di utilizzare una libreria dinamica, si avvale di un altro programma che a sua volta deve eseguire il «collegamento dinamico» (*dynamic link*). Il nome di questo collegatore dinamico viene definito in fase di compilazione del primo programma e in un sistema GNU/Linux è costituito generalmente dal file '`/lib/ld-linux.so.2`'. A sua volta, il collegatore dinamico cerca il file contenente la libreria richiesta dal programma in un gruppo di directory che solitamente sono '`/lib/`', '`/usr/lib/`' e altre, secondo la configurazione contenuta nel file '`/etc/ld.so.conf`'.

Il file '`/etc/ld.so.conf`' deve essere elaborato attraverso il programma '**`ldconfig`**' che a sua volta produce il file '`/etc/ld.so.cache`', il quale viene interpellato effettivamente da '`/lib/ld-linux.so.2`'. Pertanto, quando si modifica il file '`/etc/ld.so.conf`', occorre ricordarsi di riavviare '**`ldconfig`**'.

Se esiste la variabile di ambiente '**`LD_LIBRARY_PATH`**', i file delle librerie vengono cercati nei percorsi che questa contiene. Per esempio, per utilizzare i file contenuti nella directory corrente, continuando eventualmente in altre directory consuete, basta assegnare il percorso '`.`', seguito dagli altri a cui si è interessati:

```
$ export LD_LIBRARY_PATH=".:/lib:/usr/lib:/usr/local/lib" [Invio]
```

65.2.2 Creazione di una libreria dinamica

Per compilare dei file sorgenti in modo che diventino una libreria dinamica, occorre usare delle opzioni particolari in fase di collegamento (*link*) e nei file sorgenti è necessario pubblicizzare le funzioni in modo particolare. A titolo di esempio si prendono due funzioni, rispettivamente per il calcolo della potenza e del fattoriale (sono già usate nella sezione 64.11 in programmi compilati in modo statico), contenute in due file separati. La coppia di listati è completa e vengono mostrate entrambe le versioni per GNU AS e NASM, evidenziando le direttive significative per ottenere una libreria dinamica.

```
# lib_pwr.s
.section .text
.globl f_pwr
.type f_pwr, @function
#
f_pwr:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %esi  # Base.
    mov   12(%ebp), %edi # Esponente.
    #
    cmp   $0, %esi       # Se la base è pari a 0,
    jz    f_pwr_end_0    # restituisce 0.
    #
    cmp   $0, %edi       # Se l'esponente è pari a 0,
    jz    f_pwr_end_1    # restituisce 1.
    #
    dec   %edi           # Riduce l'esponente di una unità.
    push  %edi           # f_pwr (ESI, EDI) ==> EAX
    push  %esi           #
```

```
    call  f_pwr           #
    add   $8, %esp        #
    mul   %esi            # EDX:EAX = EAX*ESI
    mov   %eax, -4(%ebp)  # Salva il risultato.
    jmp   f_pwr_end_X     # Conclude la funzione.
    #
f_pwr_end_0:
    popa                  # Conclude la funzione con EAX = 0.
    mov $0, %eax          #
    leave                 #
    ret                   #
f_pwr_end_1:
    popa                  # Conclude la funzione con EAX = 1.
    mov $1, %eax          #
    leave                 #
    ret                   #
f_pwr_end_X:
    popa                  # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax    # al valore salvato nella variabile
    leave                 # locale.
    ret                   #
```

```
# lib_fact.s
.section .text
.globl f_fact
.type f_fact, @function
#
f_fact:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %edi   # Valore di cui calcolare il
                          # fattoriale.
    #
```

```
    cmp   $1, %edi        # Il fattoriale di 1 è 1.
    jz    f_fact_end_1    #
    #
    mov   %edi, %esi      # ESI contiene il valore di cui si
    dec   %esi            # vuole il fattoriale, ridotto di
                          # una unità.
    #
    push  %esi            # f_fact (ESI) ==> EAX
    call  f_fact          #
    add   $4, %esp        #
    mul   %edi            # EDX:EAX = EAX*EDI
    mov   %eax, -4(%ebp)  # Salva il risultato.
    jmp   f_fact_end_X    # Conclude la funzione.
    #
f_fact_end_1:
    popa                  # Conclude la funzione con EAX = 1.
    mov $1, %eax          #
    leave                 #
    ret                   #
f_fact_end_X:
    popa                  # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax    # al valore salvato nella variabile
    leave                 # locale.
    ret                   #
```

```
; lib_pwr.s
section .text
global f_pwr:function
;
f_pwr:
    enter 4,0
    pusha
    ;
    mov   esi, [ebp+8]  ; Base.
```

```
    mov   edi, [ebp+12] ; Esponente.
    ;
    cmp   esi, 0        ; Se la base è pari a 0,
    jz    f_pwr_end_0   ; restituisce 0.
    ;
    cmp   edi, 0        ; Se l'esponente è pari a 0,
    jz    f_pwr_end_1   ; restituisce 1.
    ;
    dec   edi           ; Riduce l'esponente di una unità.
    push  edi           ; f_pwr (ESI, EDI) ==> EAX
    push  esi           ;
    call  f_pwr         ;
    add   esp, 8        ;
    mul   esi           ; EDX:EAX = EAX*ESI
    mov   [ebp-4], eax  ; Salva il risultato.
    jmp   f_pwr_end_X   ; Conclude la funzione.
    ;
f_pwr_end_0:
    popa                ; Conclude la funzione con EAX = 0.
    mov eax, 0          ;
    leave               ;
    ret                 ;
f_pwr_end_1:
    popa                ; Conclude la funzione con EAX = 1.
    mov eax, 1          ;
    leave               ;
    ret                 ;
f_pwr_end_X:
    popa                ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]    ; al valore salvato nella variabile
    leave               ; locale.
    ret                 ;
```

```
; lib_fact.s
```

```
section .text
global f_fact:function
;
f_fact:
    enter 4,0
    pusha
    ;
    mov   edi, [ebp+8] ; Valore di cui calcolare il
                       ; fattoriale.
    ;
    cmp   edi, 1       ; Il fattoriale di 1 è 1.
    jz    f_fact_end_1 ;
    ;
    mov   esi, edi     ; ESI contiene il valore di cui si
    dec   esi          ; vuole il fattoriale, ridotto di
                       ; una unità.
    ;
    push  esi          ; f_fact (ESI) ==> EAX
    call  f_fact       ;
    add   esp, 4       ;
    mul   edi          ; EDX:EAX = EAX*EDI
    mov   [ebp-4], eax ; Salva il risultato.
    jmp   f_fact_end_X ; Conclude la funzione.
    ;
f_fact_end_1:
    popa               ; Conclude la funzione con EAX = 1.
    mov eax, 1         ;
    leave              ;
    ret                ;
f_fact_end_X:
    popa               ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]   ; al valore salvato nella variabile
    leave              ; locale.
    ret                ;
```

Come si può osservare, non basta dichiarare come globale il simbolo della funzione: occorre anche specificare il suo ruolo di funzione.

Ammesso che i file si chiamino, rispettivamente, '`lib_pwr.s`' e '`lib_fact.s`', si compilano come di consueto per ottenere i file oggetto relativi:

```
$ as --gstabs -o lib_pwr.o lib_pwr.s[Invio]

$ as --gstabs -o lib_fact.o lib_fact.s[Invio]
```

Ovvero:

```
$ nasm -g -f elf -o lib_pwr.o lib_pwr.s[Invio]

$ nasm -g -f elf -o lib_fact.o lib_fact.s[Invio]
```

Poi, per ottenere la libreria vera e propria, si procede con '**`ld`**' nel modo seguente (a questo punto non fa differenza l'origine dei file oggetto):

```
$ ld -shared -o libmate.so lib_pwr.o lib_fact.o[Invio]
```

Così facendo si ottiene il file '`libmate.so`' che costituisce la libreria voluta (la sigla «so» sta per *Shared object*).

65.2.3 Creare un programma che utilizza una libreria dinamica

«

Seguendo l'esempio della sezione precedente, si può creare un programma che si avvale della funzione '**`f_fact`**', contenuta nella libreria dinamica '`libmate.so`':

```
# op1!
#
.section .data
```

```
op1:    .int    5
#
.section .text
.globl _start
.extern f_fact
#
_start:
    push  op1         # f_fact (op1) ==> EAX
    call  f_fact      #
    add   $4, %esp    #
    #
    mov   %eax, %ebx # Restituisce il valore del fattoriale,
    mov   $1, %eax    # ammesso che sia abbastanza piccolo
    int   $0x80       # da poter essere rappresentato come
                      # valore di uscita.
```

```
; op1!
;
section .data
op1:    dd    5
;
section .text
global _start
extern f_fact
;
_start:
    push  long [op1] ; f_fact (op1) ==> EAX
    call  f_fact     ;
    add   esp, 4     ;
    ;
    mov   ebx, eax   ; Restituisce il valore del fattoriale,
    mov   eax, 1     ; ammesso che sia abbastanza piccolo
    int   0x80       ; da poter essere rappresentato come
                     ; valore di uscita.
```

La compilazione per produrre il file oggetto avviene nel modo consueto:

```
$ as --gstabs -o fact.o fact.s[Invio]
```

Ovvero:

```
$ nasm -g -f elf -o fact.o fact.s[Invio]
```

Poi, la trasformazione in file eseguibile richiede l'uso di opzioni particolari per '**ld**':

```
$ ld -L . ↩
↪    -dynamic-linker /lib/ld-linux.so.2 ↩
↪    -lmate ↩
↪    -o fact ↩
↪    fact.o[Invio]
```

Vanno osservate alcune opzioni:

Opzione	Descrizione
`-L .`	Indica di cercare la libreria nella directory corrente.
`-dynamic-linker /lib/ld-linux.so.2`	Indica di usare, al momento dell'avvio del programma che si sta creando, il «collegatore dinamico» costituito dal file '`/lib/ld-linux.so.2`'.
`-lmate`	Indica di instaurare il collegamento dinamico con la libreria «mate», ovvero il file '`libmate.so`' (che secondo l'opzione '**`-L .`**' si trova nella directory corrente).

Dato il modo in cui viene usata l'opzione '**-l**', si comprende che i file delle librerie devono avere sempre un nome che inizia per '`lib`…'.

Dall'ultimo comando mostrato si ottiene il file eseguibile '**fact**' nella directory corrente, il quale ha bisogno della libreria '`libmate.so`'. Se si vuole avviare questo programma, è necessario che il file della libreria si trovi in uno dei percorsi previsti. In questo caso si trova provvisoriamente nella directory corrente e si può utilizzare la variabile di ambiente '**LD_LIBRARY_PATH**' per istruire di conseguenza il collegatore dinamico:

```
$ LD_LIBRARY_PATH="." ./fact ; echo $?[Invio]

  120
```

Con '**ldd**' si può verificare la dipendenza del programma dalle librerie, ma anche in questo caso va utilizzata la variabile di ambiente '**LD_LIBRARY_PATH**':

```
$ LD_LIBRARY_PATH="." ldd fact[Invio]

        linux-gate.so.1 =>  (0xffffe000)
        libmate.so => ./libmate.so (0xb7f46000)
```

65.2.4 Creare un file che utilizza una libreria dinamica standard

Così come è possibile utilizzare le proprie librerie dinamiche, si possono sfruttare benissimo quelle scritte da altri autori. Per poter utilizzare le funzioni comuni del linguaggio C, ci si può avvalere della libreria omonima, '**c**', ovvero '`libc.so`', che di norma si trova nella directory '`/lib/`'. A titolo di esempio viene mostrato un programma che emette un messaggio attraverso lo standard output:

```
# hello.s
#
.section .data
msg:    .ascii  "Ciao mondo!\n\0"
#
.section .text
.globl _start
.extern printf
.extern exit
#
_start:
    push  $msg          # printf (msg)
    call  printf        #
    add   $4, %esp      #
    #
    push  $0            # exit (0)
    call  exit          #
```

```
; hello.s
;
section .data
msg:    db      "Ciao mondo!", 0x0A, 0x00
;
section .text
global _start
extern printf
extern exit
;
_start:
    push  long msg      ; printf (msg)
    call  printf        ;
    add   esp, 4        ;
    ;
    push  long 0        ; exit (0)
```

```
    call  exit          ;
```

Il programma utilizza due funzioni, '**printf**' e '**exit**', la prima per visualizzare un messaggio e la seconda per concluderne il funzionamento. La funzione '**printf**' richiede come primo argomento (in questo caso anche l'unico) l'indirizzo iniziale di una stringa terminata da un byte completamente a zero: nel sorgente per GNU AS il codice di interruzione di riga e lo zero vengono inseriti con le sequenze '**\n\0**', mentre in quello per NASM i codici relativi sono messi direttamente in forma numerica.

Il sorgente si compila come di consueto per ottenere il file oggetto. Successivamente, il collegamento avviene con il comando seguente:

```
$ ld -dynamic-linker /lib/ld-linux.so.2 ↩
↪    -lc ↩
↪    -o hello ↩
↪    hello.o[Invio]
```

Rispetto al caso descritto nella sezione precedente, si può osservare che manca l'opzione '**-L**', in quanto la libreria va cercata nei percorsi standard previsti; inoltre, conformemente all'esempio già visto, per indicare la libreria è stato usato solo il nome '**c**', da cui l'opzione '**-lc**'.

Dal momento che la libreria si trova nei percorsi standard, per avviare il programma non servono accorgimenti particolari:

```
$ ./hello[Invio]

  Ciao mondo!
```

Con '**ldd**' si può verificare la dipendenza del programma dalle librerie:

```
$ ldd hello[Invio]

        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/tls/libc.so.6 (0xb7e71000)
        /lib/ld-linux.so.2 (0xb7fb3000)
```

65.2.5 Librerie statiche

«

È utile sapere come sono organizzate le «librerie statiche» in un sistema GNU. Di per sé sono semplicemente file oggetto, compilati in modo da rendere pubblici i simboli delle funzioni a cui si può essere interessati esternamente, ma raccolti in un archivio che costituisce la libreria.

```
$ ar -cvq libmate.a lib_pwr.o lib_fact.o[Invio]
```

Il comando appena mostrato crea la libreria «mate» nel file ‘`libmate.a`’, composta dai file oggetto ‘`lib_pwr.o`’ e ‘`lib_fact.o`’. Il file della libreria è un semplice archivio «ar», che non prevede la compressione.

Il modo più semplice per collegare un programma che utilizza una libreria statica di questo genere è quello di indicare il file della libreria come se fosse un file oggetto:

```
$ ld -o fact fact.o libmate.a[Invio]
```

65.3 Dal sorgente all'immagine in memoria

«

Ciò che succede a partire da un file sorgente fino al programma in esecuzione in memoria è definito da un procedimento molto complesso, anche se il compilatore e il sistema operativo consentono di ignorarlo quasi completamente. Qui si mostra un esempio banale,

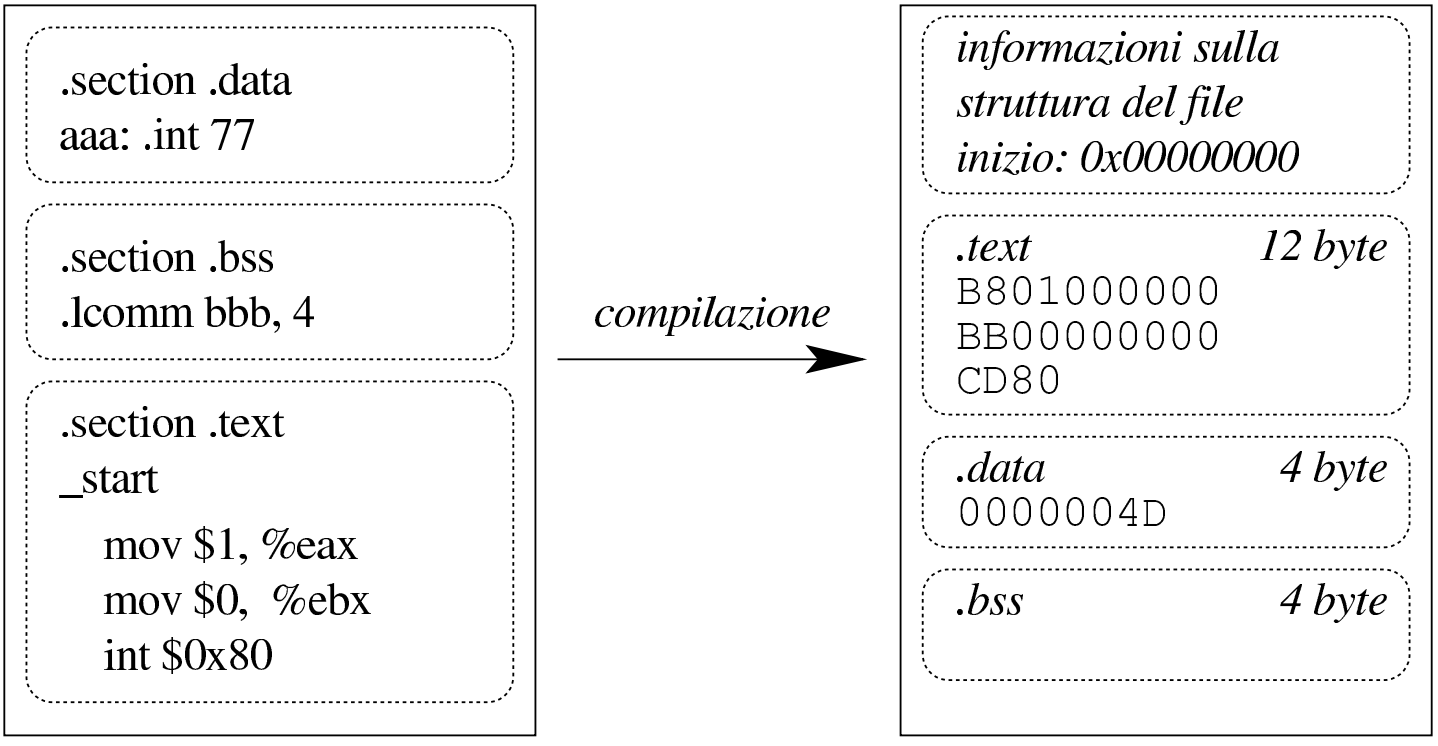
con tutti i passaggi fino ad arrivare all'immagine in memoria, senza però entrare nella questione dell'uso di librerie dinamiche.

La dimostrazione che appare si basa implicitamente sul formato ELF, sia per i file oggetto rilocabili, sia per i file eseguibili, senza entrare per ora nel dettaglio del formato stesso.

65.3.1 File oggetto

Di norma, la compilazione di un sorgente produce un file oggetto ***rilocabile***, ma non eseguibile. Questo file oggetto contiene il codice ottenuto dall'interpretazione del sorgente, diviso in sezioni (come descritto nel sorgente stesso) che possono essere ricomposte, successivamente, con una cerca libertà.

Figura 65.26. Dalle sezioni del file sorgente a quelle del file oggetto rilocabile.



A titolo di esempio si può prendere il file seguente (che riproduce quanto si vede nella figura), compilandolo nel modo consueto (qui

si mostra solo l'uso di GNU AS, per semplicità). Si suppone che il file sorgente si chiami '`prg.s`':

```
.section .data
aaa:   .int 77
.section .bss
.lcomm bbb, 4
.section .text
.globl _start
_start:
    mov   $1, %eax
    mov   $0, %ebx
    int   $0x80
```

```
$ as -o prg.o prg.s[Invio]
```

Con Objdump si può analizzare il contenuto del file oggetto generato:

```
$ objdump -x prg.o[Invio]
```

```
prg.o:     file format elf32-i386
prg.o
architecture: i386, flags 0x00000010:
HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  00000000  00000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  00000000  00000000  00000044  2**2
                  ALLOC
SYMBOL TABLE:
```

```
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l       .data  00000000 aaa
00000000 l     O .bss   00000004 bbb
00000000 g       .text  00000000 _start
```

Anche solo intuitivamente, si comprende che il file oggetto riproduce le tre sezioni del sorgente, assegnando loro degli attributi. Per esempio, la sezione '**.text**' deve essere caricata in memoria, usata in sola lettura e può essere eseguita; in modo analogo, la sezione '**.data**' deve essere caricata in memoria in lettura-scrittura (l'informazione è implicita, in quanto non appare l'attributo '**READONLY**'), ma non può essere eseguita; la sezione '**.bss**' viene allocata soltanto e non prevede limitazioni particolari, a parte il fatto di non poter essere eseguibile.

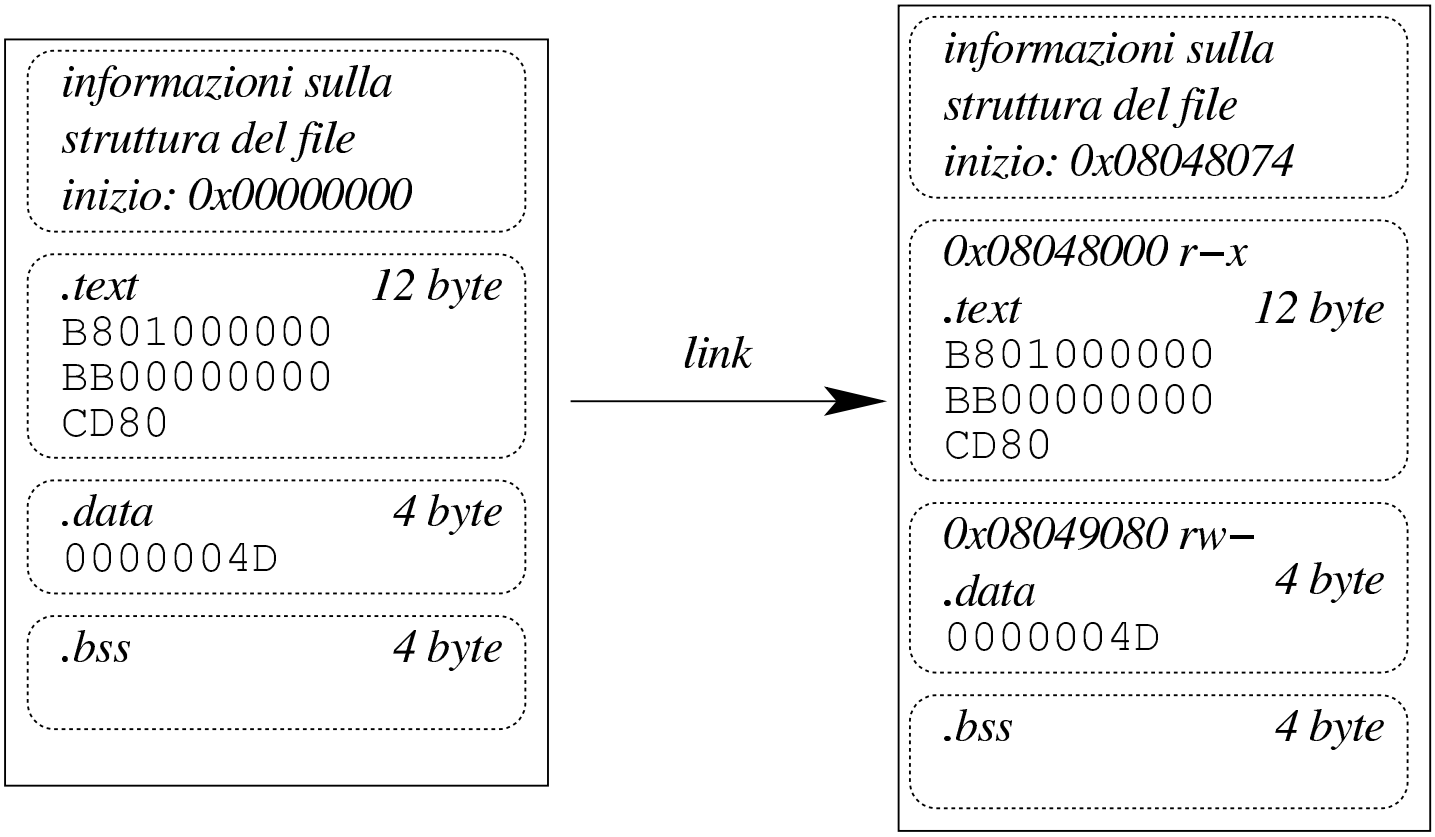
Per il momento, l'indirizzo iniziale di riferimento è 00000000_{16}.

Un file oggetto di questo tipo non può essere eseguito perché non contiene le informazioni necessarie al caricamento in memoria.

65.3.2 File eseguibile

Per ottenere un file eseguibile, i file oggetto che servono vengono raccolti da un collegatore (*link editor*) che riordina i vari componenti e produce un file con le informazioni necessarie al caricamento in memoria.

Figura 65.29. Dalle sezioni del file oggetto rilocabile ai segmenti del file oggetto eseguibile.



Continuando nell'ipotesi della sezione precedente, si passa a generare un file eseguibile a partire dal file oggetto precedente:

```
$ ld -o prg prg.o[Invio]
```

Con Objdump si può analizzare il contenuto del file eseguibile generato:

```
$ objdump -x prg[Invio]

prg:     file format elf32-i386
prg
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
```

```
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000080 memsz 0x00000080 flags r-x
    LOAD off    0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  08048074  08048074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  08049080  08049080  00000080  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  08049084  08049084  00000084  2**2
                  ALLOC
SYMBOL TABLE:
08048074 l    d  .text  00000000 .text
08049080 l    d  .data  00000000 .data
08049084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049080 l       .data  00000000 aaa
08049084 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
08049084 g       *ABS*  00000000 __bss_start
08049084 g       *ABS*  00000000 _edata
08049088 g       *ABS*  00000000 _end
```

Il file eseguibile è organizzato in ***segmenti*** che possono riferiti, ognuno, a una o più sezioni; ma il file può contenere anche sezioni che non sono riconducibili a un segmento. Il segmento, a differenza della sezione pura e semplice, deve descrivere in che modo il contenuto deve essere caricato in memoria e quali caratteristiche deve avere durante il funzionamento.

Dal rapporto generato da Objdump, precisamente nel riepilogo intitolato '**`Program Header`**', si può vedere cosa deve essere caricato in memoria e in quale posizione (gli indirizzi si riferiscono alla me-

moria virtuale). Si notano solo due segmenti, riferiti rispettivamente alla sezione '**.text**', contenente il codice da eseguire, e alla sezione '**.data**'.

Il segmento che riguarda la sezione '**.text**' deve essere caricato in memoria a partire dall'indirizzo 08048000_{16}, con permessi di lettura ed esecuzione; il segmento che si riferisce alla sezione '**.data**' deve essere caricato in memoria a partire dall'indirizzo 08049080_{16}, con permessi di lettura e scrittura. Non esiste un segmento per la sezione '**.bss**' in quanto non contiene dati e si sa comunque che deve essere allocata a partire dall'indirizzo 08049084_{16} (i permessi di lettura e scrittura sono impliciti in questo caso).

Quello che appare indicato come indirizzo iniziale è la posizione in cui si trova la prima istruzione da eseguire, pertanto è la posizione a cui deve passare il controllo il sistema di caricamento, dopo che è stata prodotta l'immagine del processo elaborativo in memoria. Questo indirizzo è interno al primo segmento, il quale è lungo 128_{10}byte (80_{16}byte), pertanto, tra l'indirizzo iniziale e quello ci devono essere delle informazioni amministrative, mentre nello spazio rimanente (esattamente 12 byte) ci sono le istruzioni vere e proprie.

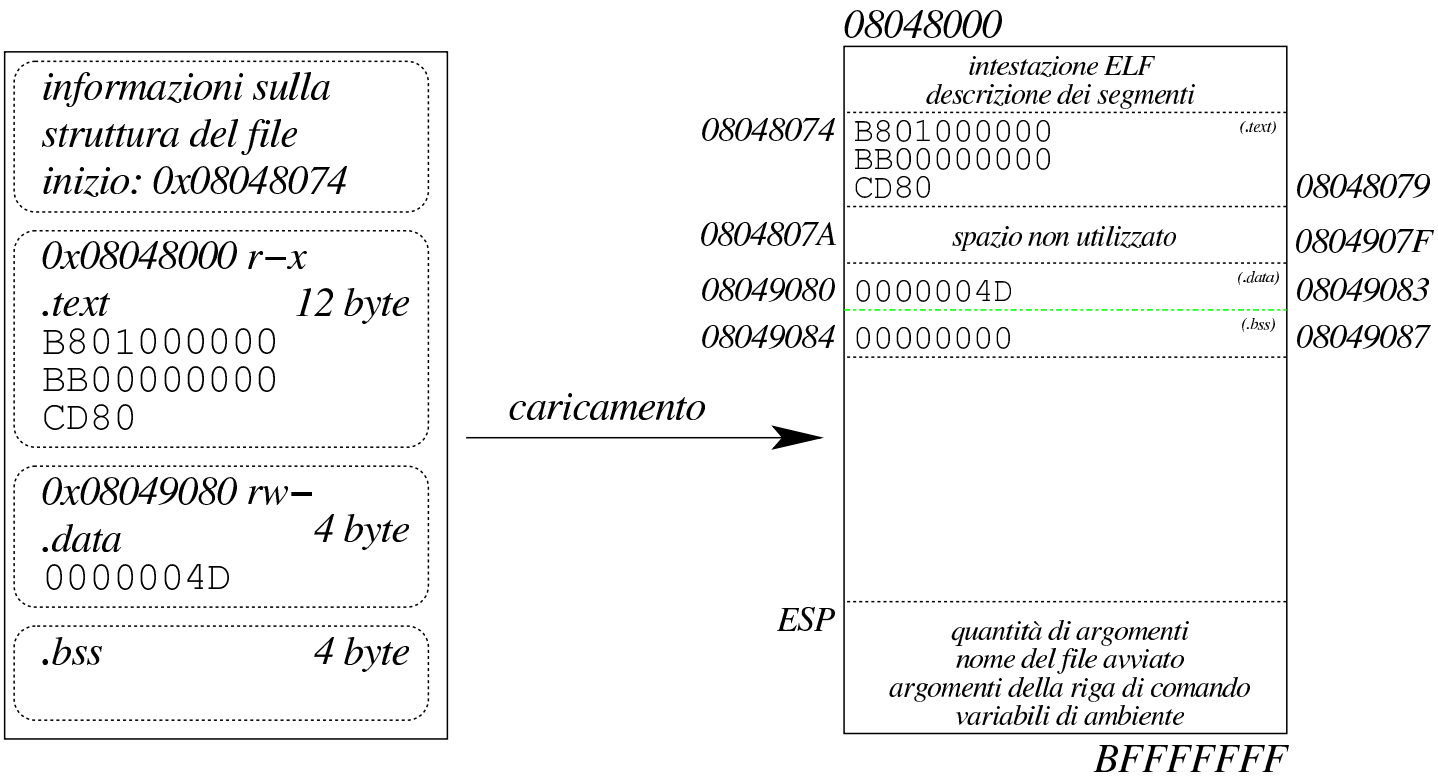
I 116 byte iniziali del primo segmento, di questo esempio, contengono precisamente l'intestazione ELF e la descrizione dei due segmenti del programma.

65.3.3 Immagine del processo nella memoria virtuale

Il sistema operativo legge il file eseguibile ed estrapola i segmenti, collocandoli in memoria, allocando anche lo spazio non inizializzato (privo pertanto di un segmento nel file eseguibile). Oltre a questo impila sul fondo le variabili di ambiente, gli argomenti della chiamata, il nome del file avviato effettivamente,... Per ultimo, su questa pila, mette la quantità di argomenti ricevuti nella riga di comando e lì posiziona il registro ***ESP***; successivamente, l'incremento di questo registro implica la crescita della pila dei dati.

In un sistema GNU/Linux i processi elaborativi utilizzano un'area della memoria virtuale che va da 08048000_{16} a $BFFFFFFF_{16}$, come se ognuno di questi disponesse della stessa dotazione di memoria e fosse sempre tutta propria. È il sistema operativo che crea questa astrazione e alloca o libera la memoria quando serve. Si osservi che lo spazio non allocato non può essere utilizzato e se il programma vi volesse fare riferimento (senza seguire la procedura prevista per l'allocazione) si otterrebbe un ***errore di segmentazione*** (*segmentation fault*).

Figura 65.31. Dal file oggetto eseguibile all'immagine del processo nella memoria virtuale.



Continuando nell'ipotesi delle sezioni precedenti, si può eseguire il programma sotto il controllo di GDB:

```
$ gdb prg [Invio]
```

Per fissare uno stop occorre indicare un indirizzo che punti almeno all'inizio della seconda istruzione (se si pretende di puntare alla prima istruzione, GDB poi non si ferma). Sapendo che la prima istruzione è all'indirizzo 08048074_{16} e che occupa cinque byte, si può usare l'indirizzo 08048079_{16} per indicare l'inizio della seconda:

```
(gdb) break *0x08048079 [Invio]
```

```
(gdb) run [Invio]
```

Si vuole verificare che i dati siano dove previsto:

```
(gdb) print (int)*0x08049080[Invio]

  $1 = 77

(gdb) print (int)*0x08049084[Invio]

  $2 = 0
```

Il secondo indirizzo fa riferimento a una memoria non inizializzata che viene posta inizialmente a zero; pertanto il risultato coincide con le previsioni. Si può verificare la presenza dell'intestazione ELF e della descrizione dei segmenti:

```
(gdb) print /x (char[116])*0x08048000[Invio]

  $3 = {0x7f, 0x45, 0x4c, 0x46, 0x1, 0x1, 0x1, 0x0, 0x0, 0x0,
  0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x2, 0x0, 0x3, 0x0, 0x1, 0x0,
  0x0, 0x0, 0x74, 0x80, 0x4, 0x8, 0x34, 0x0, 0x0, 0x0, 0xb0,
  0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x34, 0x0, 0x20, 0x0,
  0x2, 0x0, 0x28, 0x0, 0x7, 0x0, 0x4, 0x0, 0x1, 0x0, 0x0, 0x0,
  0x0, 0x0, 0x0, 0x0, 0x0, 0x80, 0x4, 0x8, 0x0, 0x80, 0x4,
  0x8, 0x80, 0x0, 0x0, 0x0, 0x80, 0x0, 0x0, 0x0, 0x5, 0x0,
  0x0, 0x0, 0x0, 0x10, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x80,
  0x0, 0x0, 0x0, 0x80, 0x90, 0x4, 0x8, 0x80, 0x90, 0x4, 0x8,
  0x4, 0x0, 0x0, 0x0, 0x8, 0x0, 0x0, 0x0, 0x6, 0x0, 0x0, 0x0,
  0x0, 0x10, 0x0, 0x0}

(gdb) print (char[4])*0x08048000[Invio]

  $4 = "\177ELF"
```

Se si tenta di raggiungere un'area di memoria non allocata, si ottiene un errore:

```
(gdb) print /x (int)*0x0804A000[Invio]
```

```
Cannot access memory at address 0x804a000
```

Il registro ***ESP*** si trova effettivamente in una zona abbastanza profonda della memoria virtuale:

```
(gdb) info registers[Invio]

…
esp            0xbf87a540       0xbf87a540
…
(gdb) quit[Invio]
```

65.3.4 Allineamento dei segmenti in memoria

Riprendendo il rapporto generato da Objdump, va osservato che i segmenti da caricare in memoria sono «allineati»:

```
Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000080 memsz 0x00000080 flags r-x
    LOAD off    0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-
```

È indicato che l'allineamento è da blocchi di 4096 byte (1000_{16}byte) ovvero 2^{12} byte. Ciò comporta un allontanamento significativo del secondo segmento dal primo (da 08048080_{16} che sarebbe il primo byte libero dopo il primo segmento, si salta a 08049080_{16}). Questo distacco lo produce il collegatore, o *link editor* (GNU LD), evidentemente per qualche motivo importante: in un sistema GNU/Linux la memoria virtuale è organizzata in pagine da 4 Kibyte (4096 byte) e non sarebbe possibile distinguere i permessi di accesso se i segmenti occupassero la stessa pagina.

È il caso di osservare che, nell'esempio mostrato, il distacco appare solo negli indirizzi che i segmenti devono prendere in memoria, perché nel file eseguibile, invece, sono collocati uno di seguito all'altro:

```
Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000080 memsz 0x00000080 flags r-x
    LOAD off    0x00000080 vaddr 0x08049080 paddr 0x08049080 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-
```

65.3.5 Script per il collegamento

GNU LD,[1] ovvero il programma che si usa per collegare i file oggetto rilocabili, consente di definire la struttura del file eseguibile da generare, con un certo grado di dettaglio, attraverso quello che viene definito uno script (precisamente *link script* o *linker script*).

Esiste una configurazione predefinita di come deve essere realizzata la struttura del file eseguibile e la si può consultare con l'opzione '**`--verbose`**':

```
$ ld --verbose [Invio]
```

```
…
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR("/usr/i486-linux-gnu/lib32"); SEARCH_DIR("/usr/local/lib32");…
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = 0x08048000); . = 0x08048000 + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  .hash           : { *(.hash) }
    …
    …
    …
```

```
    .debug_weaknames 0 : { *(.debug_weaknames) }
    .debug_funcnames 0 : { *(.debug_funcnames) }
    .debug_typenames 0 : { *(.debug_typenames) }
    .debug_varnames  0 : { *(.debug_varnames) }
    /DISCARD/ : { *(.note.GNU-stack) }
  }
```

Con l'opzione '**-T**' è possibile rimpiazzare completamente la configurazione predefinita, indicando lo script da caricare al suo posto. A titolo di esempio viene mostrato uno script molto semplificato che può essere usato con il programma apparso nelle sezioni precedenti, producendo un risultato simile:

```
ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        *(.bss)
        *(COMMON)
    }
}
```

La direttiva che appare nella prima riga, dichiara il simbolo in corrispondenza del quale associare il punto di inizio; infatti, secondo le convenzioni comuni, il simbolo '**_start**' è quello che in un sorgente in linguaggio assemblatore segnala l'inizio del programma:

```
ENTRY (_start)
```

Successivamente appare un blocco, all'interno del quale si dichiara la configurazione delle sezioni del programma. La prima direttiva di questo blocco definisce l'indirizzo iniziale di riferimento, ottenuto sommando a 08048000_{16} la dimensione dell'intestazione (ov-

vero l'intestazione ELF vera e propria, assieme alla descrizione dei segmenti):

```
    . = 0x08048000 + SIZEOF_HEADERS;
```

Di seguito appare la descrizione delle tre sezioni tipiche: '**.text**', '**.data**' e '**.bss**'. La prima è la più semplice, in quanto si limita a dichiarare di collocare la sezione '**.text**' a partire dalla posizione corrente (quella raggiunta in quel punto), rappresentata da un punto singolo, '.', purché ci siano effettivamente sezioni con quel nome da collocare:

```
    .text . : { *(.text) }
```

In questo tipo di direttiva, il punto che rappresenta la posizione corrente è facoltativo (nel senso che può essere omesso); ciò che appare tra parentesi graffe è il contenuto che la nuova sezione '**.text**' deve avere e in questo caso rappresenta la somma delle sezioni '**.text**' dei file oggetto rilocabili. In particolare, l'asterisco iniziale serve a precisare che in mancanza di tali sezioni nei file oggetto rilocabili, non si deve creare la sezione corrispondente nel file eseguibile.

La sezione '**.data**' viene dichiarata in modo simile, con la differenza che, al posto del punto, viene indicato di spostare in avanti l'indirizzo in modo che sia un multiplo di 1000_{16}, ovvero di 4096_{10}. In questo modo si vuole ottenere che la sezione '**.data**' sia distanziata nel file eseguibile e che così distante sia anche il segmento caricato in memoria. In pratica, l'espressione '**ALIGN (0x1000)**' si traduce nel calcolo di un indirizzo adeguato all'allineamento che si intende ottenere, di 4096 byte:

```
    .data ALIGN (0x1000) : { *(.data) }
```

L'ultima sezione, '**.bss**', è un po' più articolata, in quanto prevede l'inclusione delle sezioni con lo stesso nome provenienti dai file oggetto rilocabili (se ce ne sono), con l'aggiunta di tutto ciò che costituisce dati non inizializzati, rappresentato dall'espressione '***(COMMON)**'.

Riprendendo il programma di esempio già visto nelle sezioni precedenti, ammesso che lo script appena descritto sia contenuto nel file '`config.ld`', il file oggetto '`prg.o`' potrebbe essere elaborato nel modo seguente:

```
$ ld -T config.ld -o prg prg.o[Invio]
```

Ecco cosa si può vedere con Objdump:

```
$ objdump -x prg[Invio]
```

```
prg:     file format elf32-i386
prg
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000080 memsz 0x00000080 flags r-x
    LOAD off    0x00001000 vaddr 0x08049000 paddr 0x08049000 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  08048074  08048074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  08049000  08049000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  08049004  08049004  00001004  2**2
                  ALLOC
SYMBOL TABLE:
08048074 l    d  .text  00000000 .text
```

```
08049000 l    d  .data  00000000 .data
08049004 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049000 l       .data  00000000 aaa
08049004 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
```

Come si vede, questa volta il segmento riferito alla sezione '**.data**' parte esattamente da 08049000_{16}, ma così vale anche per la posizione della stessa sezione '**.data**' nel file eseguibile. In pratica, ciò comporta che il file eseguibile sia un po' più grande rispetto a prima, mentre l'utilizzo della memoria non cambia in modo sostanziale.

Sempre a titolo di esempio, si può provare a vedere cosa succede se si evita di allineare la sezione '**.data**':

```
ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data . : { *(.data) }
    .bss . : {
        *(.bss)
        *(COMMON)
    }
}
```

In questo modo, dato che il contenuto della sezione '**.text**' è molto breve, succede che il contenuto di tutte le sezioni finisce nello stesso segmento, il quale, di conseguenza, deve avere tutti i permessi necessari:

```
$ ld -T config.ld -o prg prg.o[Invio]
```

$ **objdump -x prg** [*Invio*]

```
prg:     file format elf32-i386
prg
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000084 memsz 0x00000088 flags rwx

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000000c  08048074  08048074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  08048080  08048080  00000080  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  08048084  08048084  00000084  2**2
                  ALLOC
SYMBOL TABLE:
08048074 l    d  .text  00000000 .text
08048080 l    d  .data  00000000 .data
08048084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08048080 l       .data  00000000 aaa
08048084 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
```

Naturalmente, anche il file eseguibile torna a essere di dimensioni più piccole.

Sia chiaro che gli esempi di script apparsi qui non possono essere validi in generale, ma servono solo per comprendere a grandi linee il meccanismo. Per utilizzare seriamente questo strumento occorre prima uno studio approfondito del manuale di GNU LD.

65.3.6 Osservazioni sui simboli

I file oggetto, rilocabili o eseguibili, contengono un elenco di simboli, che Objdump raccoglie in una tabella, denominata '**SYMBOL TABLE**'. Vale la pena di confrontare tale tabella nelle varie situazioni descritte qui, come riepilogato nelle figure successive.

Figura 65.49. La tabella dei simboli nel file oggetto rilocabile prodotto dalla compilazione del file sorgente.

```
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l       .data  00000000 aaa
00000000 l     O .bss   00000004 bbb
00000000 g       .text  00000000 _start
```

Figura 65.50. La tabella dei simboli nel file oggetto eseguibile prodotto da GNU LD secondo la configurazione predefinita.

```
08048074 l    d  .text  00000000 .text
08049080 l    d  .data  00000000 .data
08049084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049080 l       .data  00000000 aaa
08049084 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
08049084 g       *ABS*  00000000 __bss_start
08049084 g       *ABS*  00000000 _edata
08049088 g       *ABS*  00000000 _end
```

Figura 65.51. La tabella dei simboli nel file oggetto eseguibile prodotto da GNU LD secondo la configurazione predisposta nella sezione precedente.

```
08048074 l    d  .text  00000000 .text
08048080 l    d  .data  00000000 .data
08048084 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08048080 l       .data  00000000 aaa
08048084 l     O .bss   00000004 bbb
08048074 g       .text  00000000 _start
```

Si può osservare che, dopo la compilazione che produce un file oggetto rilocabile, appaiono gli stessi simboli previsti nel sorgente, con l'aggiunta di nomi corrispondenti a quelli delle sezioni. Nella trasformazione standard in file eseguibile, si vede la comparsa di altri simboli, in particolare: '**`.shstrtab`**', '**`.symtab`**',

‘**.strtab**’. Questi rappresentano la collocazione nel file di informazioni amministrative, relative al formato ELF. Inoltre, nel caso specifico dell’eseguibile generato secondo la configurazione predefinita di GNU LD, si vede la comparsa di simboli aggiuntivi che evidentemente dipendono dall’organizzazione della configurazione stessa.

Per comprendere come si possano inserire dei simboli addizionali attraverso lo script per GNU LD, si può riprendere l’esempio già visto nella sezione precedente, ritoccando leggermente la definizione della sezione ‘**.bss**’:

```
ENTRY (_start)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

I simboli che si vogliono aggiungere sono ‘**_sbss**’ e ‘**_ebss**’, con lo scopo di individuare l’inizio e la fine della nuova sezione ‘**.bss**’.

Figura 65.53. La tabella dei simboli dopo l'introduzione forzata di '**_sbss**' e '**_ebss**'.

```
SYMBOL TABLE:
08048074 l    d  .text  00000000 .text
08049000 l    d  .data  00000000 .data
08049004 l    d  .bss   00000000 .bss
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
08049000 l       .data  00000000 aaa
08049004 l     O .bss   00000004 bbb
08049004 g       .bss   00000000 _sbss
08049008 g       .bss   00000000 _ebss
08048074 g       .text  00000000 _start
```

Eventualmente si può sperimentare cosa cambia nel contenuto dei file oggetto (rilocabili o eseguibili) quando si compila un sorgente con l'opzione '**--gstabs**' di GNU AS o con l'opzione '**-g**' di NASM.

65.3.7 Formati dei file oggetto

«

I file oggetto rilocabili e i file eseguibili possono essere realizzati secondo diversi formati, ma dipende dal sistema operativo qual è la scelta che si deve operare. Negli esempi mostrati, partendo dal presupposto di utilizzare un sistema GNU/Linux, si fa riferimento al formato ELF, in quanto è quello che deve essere usato e gli strumenti comuni sono già configurati per generare file conformi a tale standard.

Il formato del file che si deve produrre condiziona anche i tipi di sezioni che si possono dichiarare nel sorgente in linguaggio assemblatore. Il formato ELF dà molta libertà, comunque prevede una serie numerosa di sezioni con funzioni specifiche, in particolare

‘`.rodata`’ che comporta la creazione di un segmento di memoria con dati inizializzati, ma in sola lettura.

65.4 Formato ELF

Il formato ELF è il contenitore di un programma che non si trova necessariamente nello stato di poter essere eseguito. Il formato ELF si distingue per la presenza di un’intestazione che si trova obbligatoriamente all’inizio del file; quindi, il contenuto del file è affiancato da una serie di tabelle che lo descrivono in base a vari criteri.

65.4.1 Sezioni e segmenti

Per semplificare la descrizione di un formato ELF, lo si può immaginare composto da sezioni, il cui scopo è quello di descrivere tutto ciò che compone il programma, e da segmenti, con i quali si descrive in che modo il programma deve essere rappresentato in memoria ed eseguito. L’informazione relativa alle sezioni è indispensabile quando deve intervenire un «collegatore» (*linker*); l’informazione data dai segmenti riguarda l’avvio del programma.

Se si vuole abbandonare questo tipo di rappresentazione astratta, il formato ELF lo si può vedere come un involucro del codice eseguibile e dei dati inizializzati, contenente un’intestazione di riconoscimento (che si trova obbligatoriamente all’inizio del file) e da una serie di tabelle, più o meno concatenate tra di loro, alcune delle quali possono essere facoltative, in base al contesto per il quale il file oggetto è predisposto.

Tabella 65.54. Componenti principali che descrivono un formato ELF.

Tabella	Descrizione
ELF header	È l'intestazione del file e deve trovarsi necessariamente all'inizio dello stesso. Contiene poi i riferimenti alla tabella dei segmenti (*program header table*) e a quella delle sezioni (*section header table*).
program header table	È la tabella dei segmenti da caricare in memoria, con le informazioni necessarie a procedere in tal senso. La presenza di questa tabella è obbligatoria in un file oggetto eseguibile.
section header table	È la tabella delle sezioni.
string table	È la tabella delle stringhe, a cui fanno riferimento le altre tabelle quando devono indicare una stringa di qualunque tipo.
symbol table	È la tabella dei simboli associati a varie parti del contenuto. La tabella dei simboli, per indicare i nomi dei simboli, deve fare riferimento alla tabella delle stringhe.

65.4.2 Intestazione ELF

«

L'intestazione ELF è il componente più importante del formato, in quanto la sua presenta è obbligatoria. L'intestazione consente di identificare un file ELF come tale e di raggiungere le tabelle delle sezioni e dei segmenti, da cui poi si arriva al contenuto rimanente del file.

Tabella 65.55. Intestazione ELF secondo l'architettura x86, in particolare con le informazioni necessarie a produrre un file eseguibile.

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione

`e_ident[0]`
`e_ident[1]`
`e_ident[2]`
`e_ident[3]`
	8 bit
	8 bit
	8 bit
	8 bit
		8 bit
		8 bit
		8 bit
		8 bit
			Impronta di identificazione del formato: deve corrispondere a $7F_{16}$, '**`E`**', '**`L`**', '**`F`**'.

`e_ident[4]`	8 bit	8 bit	Definisce la classe del file: 01_{16} rappresenta un file oggetto a 32 bit; 02_{16} rappresenta invece un file a 64 bit.
`e_ident[5]`	8 bit	8 bit	Definisce la codifica dei dati: 01_{16} rappresenta un formato LSB, ovvero *little endian*; 02_{16} formato MSB, ovvero *big endian*. Sia 01_{16}, sia 02_{16}, si riferiscono a una rappresentazione numerica dei valori negativi attraverso il complemento a due.
`e_ident[6]`	8 bit	8 bit	Definisce la versione dell'intestazione (inizialmente esiste solo la versione 01_{16}).

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
`e_ident[7]`	8 bit	8 bit	Questi byte definiscono informazioni di importanza minore e di solito vengono lasciati a 00_{16}.
`e_ident[8]`	8 bit	8 bit	
`e_ident[9]`	8 bit	8 bit	
`e_ident[10]`	8 bit	8 bit	
`e_ident[11]`	8 bit	8 bit	
`e_ident[12]`	8 bit	8 bit	
`e_ident[13]`	8 bit	8 bit	
`e_ident[14]`			
`e_ident[15]`	8 bit	8 bit	Dichiara la dimensione in byte della sequenza di identificazione. Il valore obbligato per questo byte è 10_{16}, ovvero 16_{10}.
`e_type`	16 bit	16 bit	Definisce il tipo di file oggetto. Un file oggetto rilocabile ha il codice 01_{16}; un file oggetto eseguibile ha il codice 02_{16}.
`e_machine`	16 bit	16 bit	Definisce il tipo di architettura. Il codice 03_{16} si riferisce al tipo Intel.
`e_version`	32 bit	32 bit	Definisce la versione del file oggetto (inizialmente esiste solo la versione 00000001_{16}).

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
`e_entry`	32 bit	64 bit	Contiene l'indirizzo a cui occorre passare il controllo per l'esecuzione del programma.
`e_phoff`	32 bit	64 bit	*program header table offset* Contiene lo scostamento, rispetto all'inizio del file, necessario per raggiungere il primo byte della tabella che descrive i segmenti da caricare in memoria. Tale tabella è nota come *program header table* ed è obbligatoria la sua presenza in un file oggetto eseguibile.
`e_shoff`	32 bit	64 bit	*section header table offset* Contiene lo scostamento, rispetto all'inizio del file, necessario per raggiungere il primo byte della tabella che descrive le sezioni. Tale tabella è nota come *section header table*.
`e_flags`	32 bit	32 bit	Contiene degli indicatori specifici per il tipo di microprocessore. Nel caso dell'architettura x86-32 può contenere semplicemente valori a zero.
`e_ehsize`	16 bit	16 bit	*ELF header size* Contiene la dimensione dell'intestazione ELF.

Nome mnemonico	Dimensione x86-32	Dimensione x86-64	Descrizione
`e_phentsize`	16 bit	16 bit	*program header entry size* Definisce la dimensione di una voce descrittiva di un segmento, nella tabella dei segmenti. Tutte le voci di tale tabella hanno la stessa dimensione.
`e_phnum`	16 bit	16 bit	*program header number* Definisce la quantità di voci contenute nella tabella di descrizione dei segmenti.
`e_shentsize`	16 bit	16 bit	*section header entry size* Definisce la dimensione di una voce descrittiva di una sezione, nella tabella delle sezioni. Tutte le voci di tale tabella hanno la stessa dimensione.
`e_shnum`	16 bit	16 bit	*section header number* Definisce la quantità di voci contenute nella tabella di descrizione delle sezioni.
`e_shstrndx`	16 bit	16 bit	*section header string index* Definisce l'indice, all'interno della tabella delle sezioni, che identifica la voce che fa riferimento alla tabella delle stringhe.

65.4.3 Descrizione dei segmenti

La descrizione dei segmenti, necessaria per mettere in esecuzione un programma, è contenuta nella tabella *program header*, composta da un array di voci, di dimensione uniforme, ognuna delle quali descrive un segmento. Si raggiunge la prima voce di questo array con lo scostamento indicato nell'intestazione ELF ('**e_phoff**'), quindi, sapendo la dimensione di ogni voce ('**e_phentsize**') e la quantità di queste ('**e_phnum**'), è possibile scandire anche le altre.

Tabella 65.56. Descrizione di una voce nella tabella dei segmenti, secondo l'architettura x86-32.

Nome mnemonico	Dimensione	Descrizione
`p_type`	32 bit	Definisce il tipo di operazione da compiere. La situazione più semplice è costituita da codice eseguibile e dati da caricare in memoria: 01_{16}.
`p_offset`	32 bit	Definisce lo scostamento, dall'inizio del file, necessario a raggiungere il primo byte del segmento.
`p_vaddr`	32 bit	Definisce l'indirizzo assoluto, nell'ambito della memoria virtuale, dove il primo byte del segmento deve trovarsi in memoria, una volta caricato.
`p_paddr`	32 bit	Equivale al campo '**p_vaddr**', ma si riferisce alla «memoria fisica». In un sistema GNU/Linux questo valore è sempre uguale a '**p_vaddr**'.

Nome mnemonico	Dimensione	Descrizione
`p_filesz`	32 bit	Definisce la dimensione del segmento nel file e in casi particolari può essere pari a zero.
`p_memsz`	32 bit	Definisce la dimensione del segmento rappresentato in memoria e in casi particolari può essere pari a zero.
`p_flags`	32 bit	Definisce degli indicatori che descrivono i permessi del segmento: 1 = esecuzione; 2 = scrittura; 4 = lettura. Per avere permessi multipli si sommano i permessi elementari. Generalmente, il segmento di una porzione di codice dispone di permessi di accesso in lettura e in esecuzione, mentre quello di un'area di dati, consente normalmente la lettura e la scrittura.
`p_align`	32 bit	Definisce l'allineamento in memoria, a blocchi del valore indicato, il quale a sua volta deve essere una potenza di due.

Tabella 65.57. Descrizione di una voce nella tabella dei segmenti, secondo l'architettura x86-64.

Nome mnemonico	Dimensione	Descrizione
`p_type`	32 bit	Definisce il tipo di operazione da compiere. La situazione più semplice è costituita da codice eseguibile e dati da caricare in memoria: 01_{16}.
`p_flags`	32 bit	Definisce degli indicatori che descrivono i permessi del segmento: 1 = esecuzione; 2 = scrittura; 4 = lettura. Per avere permessi multipli si sommano i permessi elementari. Generalmente, il segmento di una porzione di codice dispone di permessi di accesso in lettura e in esecuzione, mentre quello di un'area di dati, consente normalmente la lettura e la scrittura.
`p_offset`	64 bit	Definisce lo scostamento, dall'inizio del file, necessario a raggiungere il primo byte del segmento.
`p_vaddr`	64 bit	Definisce l'indirizzo assoluto, nell'ambito della memoria virtuale, dove il primo byte del segmento deve trovarsi in memoria, una volta caricato.
`p_paddr`	64 bit	Equivale al campo '**`p_vaddr`**', ma si riferisce alla «memoria fisica». In un sistema GNU/Linux questo valore è sempre uguale a '**`p_vaddr`**'.

Nome mnemonico	Dimensione	Descrizione
`p_filesz`	64 bit	Definisce la dimensione del segmento nel file e in casi particolari può essere pari a zero.
`p_memsz`	64 bit	Definisce la dimensione del segmento rappresentato in memoria e in casi particolari può essere pari a zero.
`p_align`	64 bit	Definisce l'allineamento in memoria, a blocchi del valore indicato, il quale a sua volta deve essere una potenza di due.

65.4.4 Definizione manuale di un formato ELF

«

Un programma eseguibile in formato ELF deve contenere l'intestazione ELF e la descrizione dei segmenti; le sezioni possono anche mancare del tutto. Viene mostrato un esempio di programma banale (non fa altro che restituire il valore 77) in cui tutto, anche ciò che costituisce il formato ELF, viene definito nel sorgente. Il programma in sé trae spunto dal documento *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux* di Brian Raiter, come annotato alla fine del capitolo. Si osservi che nel sorgente le sezioni non vengono indicate affatto.

```
.code32
.globl _start
#
file_begin:
#
# ELF header.
#
```

```
elf_header_begin:
    .byte  0x7F           # e_ident
    .byte  'E', 'L', 'F'  #
    .byte  1              #
    .byte  1              #
    .byte  1              #
    .byte  0, 0, 0, 0     #
    .byte  0, 0, 0, 0     #
    .byte  16             #
    #
    .short 2      # e_type     2 = executable file
    .short 3      # e_machine  3 = 386
    .int   1      # e_version  1 = current version
    .int   _start # e_entry        start address
    .int   (program_header_begin - file_begin)
                  # e_phoff        program header offset
    .int   0      # e_shoff    0 = no section header table
    .int   0      # e_flags        no flags
    .short (elf_header_end - elf_header_begin)
                  # e_ehsize       ELF header size
    .short (program_header_end - program_header_begin)
                  # e_phentsize    program header entry size
    .short 1      # e_phnum        program header entries
    .short 0      # e_shentsize 0 = no section header table
    .short 0      # e_shnum        section header entries
    .short 0      # e_shstrndx  0 = undefined
elf_header_end:
#
# Program header table, with just one entry.
#
program_header_begin:
    .int   1          # p_type  1 = segment to be loaded
    .int   0          # p_offset    segment's offset
    .int   0x08048000 # p_vaddr     segment's virtual
```

```
                        #               address
    .int    0x08048000 # p_paddr      segment's physical
                        #               address
    .int    (file_end - file_begin)
                        # p_filesz     file image size
    .int    (file_end - file_begin)
                        # p_memsz      memory image size
    .int    5           # p_flags 5 = read + execute
    .int    0x1000      # p_align      segment's memory
                        #               alignment
program_header_end:
#
# Program code.
#
_start:
    mov $77, %ebx
    mov $1, %eax
    int $0x80
#
file_end:
```

Il sorgente scritto nel formato adatto a GNU AS va compilato così:

```
$ as -o elf_test.o elf_test.s[Invio]
```

```
$ ld --oformat binary -o elf_test elf_test.o[Invio]
```

Come si vede, GNU LD viene usato in modo da produrre un formato binario, puro e semplice, ovvero un file privo di formato, dato che è già tutto incluso nella descrizione del programma stesso.

Si mostra anche il sorgente adatto a NASM, dove va annotata anche l'origine, ovvero l'indirizzo in cui tutto deve essere caricato in memoria:

```
    bits 32
    org  0x08048000
;
file_begin:
;
; ELF header.
;
elf_header_begin:
    db 0x7F             ; e_ident
    db 'E', 'L', 'F' ;
    db 1                ;
    db 1                ;
    db 1                ;
    db 0, 0, 0, 0       ;
    db 0, 0, 0, 0       ;
    db 16               ;
    ;
    dw 2            ; e_type      2 = executable file
    dw 3            ; e_machine   3 = 386
    dd 1            ; e_version   1 = current version
    dd _start       ; e_entry         start address
    dd (program_header_begin - file_begin)
                    ; e_phoff         program header offset
    dd 0            ; e_shoff     0 = no section header table
    dd 0            ; e_flags         no flags
    dw (elf_header_end - elf_header_begin)
                    ; e_ehsize        ELF header size
    dw (program_header_end - program_header_begin)
                    ; e_phentsize     program header entry
                    ;                 size
    dw 1            ; e_phnum         program header entries
    dw 0            ; e_shentsize 0 = no section header table
    dw 0            ; e_shnum         section header entries
    dw 0            ; e_shstrndx  0 = undefined
```

```
elf_header_end:
;
; Program header table, with just one entry.
;
program_header_begin:
    dd 1            ; p_type     1 = segment to be loaded
    dd 0            ; p_offset       segment's offset
    dd 0x08048000   ; p_vaddr        segment's virtual
                    ;                address
    dd 0x08048000   ; p_paddr        segment's physical
                    ;                address
    dd (file_end - file_begin)
                    ; p_filesz       file image size
    dd (file_end - file_begin)
                    ; p_memsz        memory image size
    dd 5            ; p_flags    5 = 1 (execute) + 4 (read)
    dd 0x1000       ; p_align        segment's memory
                    ;                 alignment
program_header_end:
;
; Program code.
;
_start:
    mov ebx, 77
    mov eax, 1
    int 0x80
;
file_end:
```

In questo caso, la compilazione non richiede altro che NASM, il quale produce direttamente il formato binario voluto:

```
$ nasm -f bin -o elf_test elf_test.s[Invio]
```

```
$ chmod +x elf_test [Invio]
```

I valori che rappresentano scostamenti e dimensioni del codice, sono calcolati attraverso il compilatore, facendo riferimento ai simboli rappresentati dalle etichette che delimitano le varie porzioni del sorgente. Ecco come si presenta il programma eseguibile dal punto di vista di Objdump:

```
$ objdump -x elf_test [Invio]

  ELF_test:     file format elf32-i386
  ELF_test
  architecture: i386, flags 0x00000102:
  EXEC_P, D_PAGED
  start address 0x08048054

  Program Header:
      LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
           filesz 0x00000060 memsz 0x00000060 flags r-x

  Sections:
  Idx Name          Size      VMA       LMA       File off  Algn
  SYMBOL TABLE:
  no symbols
```

65.4.5 Esempio più complesso

Viene mostrato un esempio più complesso, composto sempre da una sola voce nella tabella dei segmenti; in particolare viene definita una variabile inizializzata, incorporata nel segmento del codice, che nel sorgente appare in fondo. Viene mostrata solo la versione per GNU AS, trattandosi del programma per il calcolo del fattoriale, già descritto in precedenza.

```
.code32
.globl _start
#
```

```
file_begin:
#
# ELF header.
#
elf_header_begin:
    .byte  0x7F           # e_ident
    .byte  'E', 'L', 'F' #
    .byte  1              #
    .byte  1              #
    .byte  1              #
    .byte  0, 0, 0, 0     #
    .byte  0, 0, 0, 0     #
    .byte  16             #
    #
    .short 2        # e_type      2 = executable file
    .short 3        # e_machine   3 = 386
    .int   1        # e_version   1 = current version
    .int   _start   # e_entry         start address
    .int   (program_header_begin - file_begin)
                    # e_phoff         program header offset
    .int   0        # e_shoff     0 = no section header table
    .int   0        # e_flags         no flags
    .short (elf_header_end - elf_header_begin)
                    # e_ehsize        ELF header size
    .short (program_header_end - program_header_begin)
                    # e_phentsize     program header entry
                    #                 size
    .short 1        # e_phnum         program header entries
    .short 0        # e_shentsize 0 = no section header table
    .short 0        # e_shnum         section header entries
    .short 0        # e_shstrndx  0 = undefined
elf_header_end:
#
# Program header table, with just one entry.
```

```
#
program_header_begin:
    .int 1          # p_type  1 = segment to be loaded
    .int 0          # p_offset    segment's offset
    .int 0x08048000 # p_vaddr     segment's virtual address
    .int 0x08048000 # p_paddr     segment's physical address
    .int (file_end - file_begin)
                    # p_filesz    file image size
    .int (file_end - file_begin)
                    # p_memsz     memory image size
    .int 5          # p_flags 5 = read + execute
    .int 0x1000     # p_align     segment's memory alignment
program_header_end:
#
# Program code.
#
_start:
    mov  op1, %esi  # ESI contiene il valore di cui si vuole
                    # calcolare il fattoriale.
    push %esi       # f_fact (ESI) ==> EAX
    call f_fact     #
    add  $4, %esp   #
    mov  %eax, %ebx # Restituisce il valore del fattoriale,
    mov  $1, %eax   # ammesso che sia abbastanza piccolo
    int  $0x80      # da poter essere rappresentato come
                    # valore di uscita.
#
# Fattoriale di un numero senza segno.
# f_fatt (a) ==> EAX
# EAX = a!
#
f_fact:
    enter $4, $0
    pusha
```

```
    #
    mov  8(%ebp), %edi # Valore di cui calcolare il
                       # fattoriale.
    #
    cmp  $1, %edi      # Il fattoriale di 1 è 1.
    jz   f_fact_end_1  #
    #
    mov  %edi, %esi    # ESI contiene il valore di cui si
    dec  %esi          # vuole il fattoriale, ridotto di
                       # una unità.
    #
    push %esi          # f_fact (ESI) ==> EAX
    call f_fact        #
    add  $4, %esp      #
    mul  %edi          # EDX:EAX = EAX*EDI
    mov  %eax, -4(%ebp)# Salva il risultato.
    jmp  f_fact_end_X  # Conclude la funzione.
    #
f_fact_end_1:
    popa               # Conclude la funzione con EAX = 1.
    mov $1, %eax       #
    leave              #
    ret                #
f_fact_end_X:
    popa               # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax # al valore salvato nella variabile
    leave              # locale.
    ret                #
#
# Initialized data.
#
op1:    .int    5
#
file_end:
```

65.5 Programmi completamente autonomi

A volte esiste la necessità di realizzare un programma funzionante in modo autonomo, ovvero *stand alone*, senza il sostegno del sistema operativo. Un lavoro di questo tipo richiede lo studio delle stesse problematiche che riguardano inizialmente la costruzione di un nuovo sistema operativo, ma di norma è meglio fermarsi alla produzione di un programma singolo.[2]

C'è da osservare che l'avvio di un programma «autonomo» in un elaboratore x86 può essere di una complessità mostruosa, a causa di problematiche ereditate dall'architettura originale del microprocessore 8088/8086. La prima difficoltà che si incontra a tale proposito sta nel far sì che il microprocessore si metta a lavorare in «modalità protetta», ovvero in una condizione che consenta di utilizzare in modo ragionevole la memoria centrale. Nel tempo, questa e altre questioni sono diventate di competenza dei programmi che si occupano di avviare un sistema operativo, come è il caso di GRUB 1 e di SYSLINUX, che così predispongono un contesto più confortevole al programma o al kernel da avviare successivamente.

Qui si mostrano esempi che utilizzano anche codice in linguaggio C, il quale viene descritto a partire dal capitolo 66.

65.5.1 Le specifiche «multiboot»

Le specifiche *multiboot* sono definite dal documento *Multiboot specification*, disponibile presso *http://www.gnu.org/software/grub/manual/multiboot/* . Si tratta della definizione di un'interfaccia tra sistema di avvio e sistema operativo (o programma autonomo), inizialmente per un'architettura x86. Il documento citato contiene sia

le specifiche, sia un esempio completo di programma che interagisce con il sistema di avvio secondo le specifiche stesse. Qui si riassumono i concetti principali.

65.5.1.1 Formato del file che deve essere avviato

Il file-immagine che contiene il programma da avviare (programma che potrebbe essere il kernel di un sistema operativo), deve contenere un'intestazione particolare, definita *multiboot header*, costituita in pratica da un'impronta di riconoscimento e da una serie di dati. Attraverso questa intestazione, il sistema di avvio è almeno in grado di riconoscere il file-immagine come qualcosa che deve essere avviato effettivamente e di recepirne le caratteristiche.

Questa intestazione deve trovarsi nella parte iniziale del file-immagine da caricare ed eseguire, ma non è necessario che sia esattamente all'inizio dello stesso, essendo sufficiente che sia contenuta completamente entro i primi 8 Kibyte.

Figura 65.62. La prima parte obbligatoria dell'intestazione.

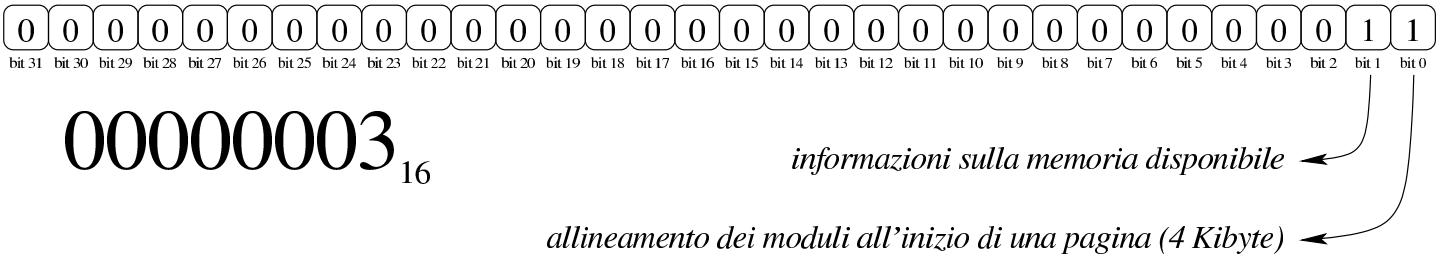
Il primo campo da 32 bit, definito ***magic***, contiene un'impronta di riconoscimento, costituita precisamente dal numero $1BADB002_{16}$. Questa serve al sistema di avvio a individuare la presenza e l'ini-

zio di una tale intestazione. Il secondo campo da 32 bit, definito ***flags***, contiene degli indicatori con i quali si richiede un certo comportamento al sistema di avvio. Il terzo campo da 32 bit, definito ***checksum***, contiene un numero calcolato in modo tale che la somma tra i numeri contenuti nei tre campi da 32 bit porti a ottenere zero, senza considerare i riporti.

I nomi indicati sono quelli definiti dallo standard e, come si vede, il campo ***checksum*** si ottiene calcolando $-(\textbf{\textit{magic}} + \textbf{\textit{flags}})$, dove si deve intendere che i calcoli avvengono con valori interi senza segno e si ignorano i riporti.

Se il file-immagine da avviare è informato ELF, le informazioni che il sistema di avvio necessita per piazzarlo correttamente in memoria e per passare il controllo allo stesso, sono già disponibili e non c'è la necessità di occuparsi di altri campi facoltativi che possono seguire i tre già descritti. Stante questa semplificazione, per quanto riguarda il campo ***flags*** sono importanti i primi due bit, mentre gli altri vanno lasciati a zero.

Figura 65.63. Il campo ***flags*** e il suo utilizzo fondamentale.

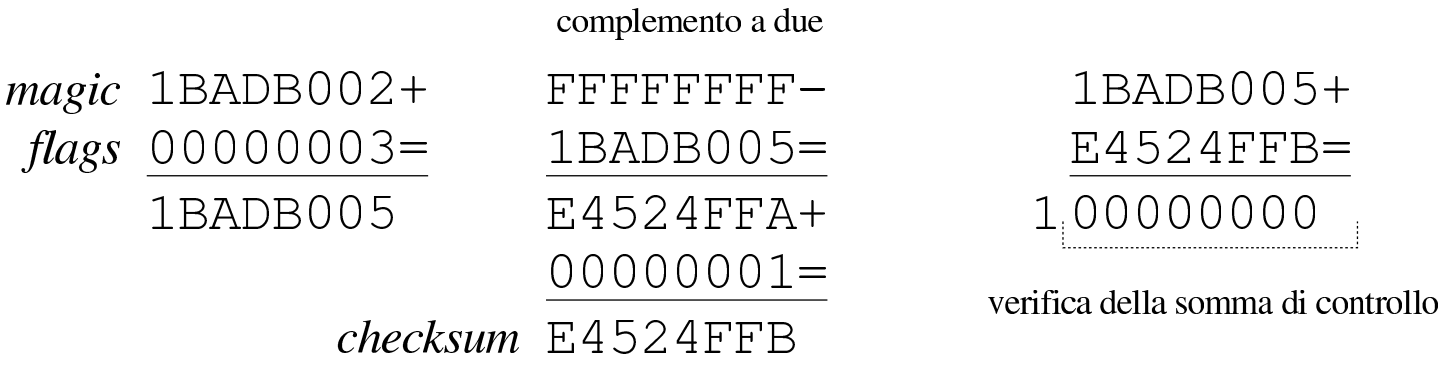


Il bit meno significativo del campo ***flags***, se impostato a uno, serve a richiedere il caricamento in memoria dei moduli eventuali (assieme al file-immagine principale) in modo che risultino allineati all'inizio di una «pagina» (ovvero all'inizio di un blocco da 4 Kibyte). Alcuni

sistemi operativi hanno la necessità di trovare i moduli allineati in questo modo e in generale l'attivazione di tale bit non può creare danno.

Il secondo bit del campo ***flags*** serve a richiedere al sistema di avvio di passare le informazioni disponibili sulla memoria. Queste informazioni vengono rese disponibili a partire da un'area a cui punta inizialmente il registro ***EBX***. In generale si tratta di un'informazione utile (che al massimo può essere ignorata), pertanto conviene attivare anche questo bit.

Figura 65.64. Calcolo del campo ***checksum***.



65.5.1.2 Situazione dopo l'avvio del file-immagine

Quando, dopo il trasferimento in memoria del programma, il sistema di avvio passa il controllo allo stesso, la situazione che questo programma si trova è sostanzialmente quella seguente, dove però sono stati omessi molti dettagli importanti:

- il microprocessore è in modalità protetta;
- il registro ***EAX*** contiene il numero $2BADB002_{16}$ (si osservi che la prima cifra è cambiata, rispetto all'impronta che deve avere il file-immagine da avviare);

- il registro ***EBX*** deve contenere l'indirizzo fisico, a 32 bit, di una serie di campi contenenti informazioni passate dal sistema di avvio (*multiboot information structure*).

Di norma, la prima cosa che fa il programma che è stato avviato in questo modo è di azzerare il registro ***EFLAGS*** e di predisporre uno spazio per la pila dei dati, posizionando il registro ***ESP*** di conseguenza.

65.5.1.3 Informazioni passate dal sistema di avvio al programma

Il sistema di avvio conforme alle specifiche *multiboot* offre una serie di informazioni, collocate in una struttura che parte dall'indirizzo indicato nel registro ***EBX***. Questa struttura ha un certo grado di complessità, in quanto può fare riferimento ad altre strutture. Qui viene descritta brevemente solo una prima porzione di questa struttura; per l'approfondimento occorre consultare le specifiche, pubblicate presso *http://www.gnu.org/software/grub/manual/multiboot/* .

Figura 65.65. Inizio della struttura informativa offerta da un sistema di avvio aderente alle specifiche *multiboot*.

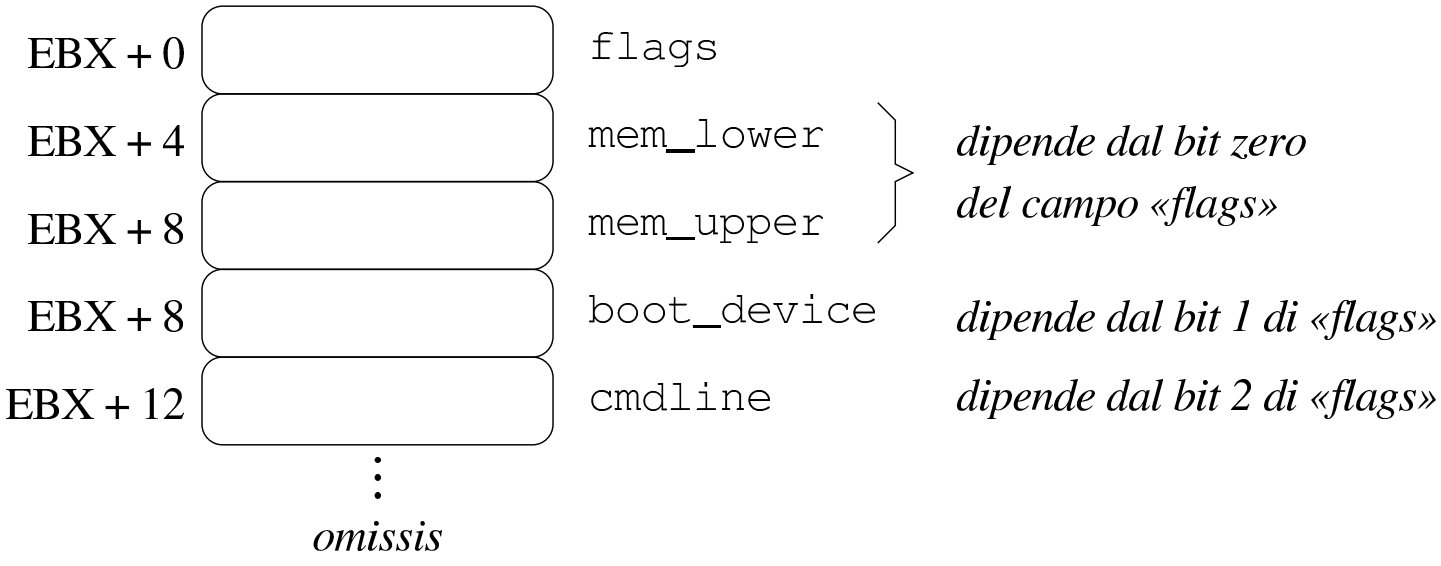


Tabella 65.66. Descrizione dei primi campi della struttura informativa fornita dal sistema di avvio *multiboot*.

Nome mnemonico del campo	bit del campo '**flags**' da cui dipende	Descrizione
`flags`		Il primo campo definisce una serie di indicatori, con i quali si dichiara se una certa informazione, successiva, viene fornita ed è valida.

Nome mnemonico del campo	bit del campo '**flags**' da cui dipende	Descrizione

`mem_lower`
 	
`mem_upper`
	0	Se è attivo il bit meno significativo del campo '**flags**', i campi '**mem_lower**' e '**mem_upper**' contengono la dimensione della memoria bassa (da zero a un massimo di 640 Kibyte) e della memoria alta (quella che si trova a partire da un mebibyte). La dimensione è da intendersi in kibibyte (simbolo Kibyte) e, per quanto riguarda la memoria alta, viene indicata solo la dimensione continua fino al primo «buco».

`boot_device`	1	Se è attivo il secondo bit, partendo dal lato meno significativo, il campo '**boot_device**' dà informazioni sull'unità di avvio. L'informazione è divisa in quattro byte, come descritto nelle specifiche *multiboot*.
`cmdline`	2	Se è attivo il terzo bit, partendo dal lato meno significativo, il campo '**cmdline**' contiene l'indirizzo iniziale di una stringa che riproduce la riga di comando passata al kernel.

Come si può intuire leggendo la tabella che descrive i primi cinque

campi, il significato dei bit del campo '**flags**' viene attribuito, mano a mano che l'aggiornamento delle specifiche prevede l'espansione della struttura informativa. Per esempio, un campo '**flags**' con il valore 100_2 sta a significare che esistono i campi fino a '**cmdline**' e il contenuto di quelli precedenti non è valido, ma i campi successivi, non esistono affatto. La comprensione di questo concetto dovrebbe rendere un po' più semplice la lettura delle specifiche.

65.5.2 Esempio di programma da avviare secondo le specifiche «multiboot»

«

I listati seguenti mostrano il contenuto dei file necessari a produrre un programma da avviare secondo le specifiche *multiboot*. Per la precisione, il programma non fa alcunché e serve solo come base di partenza per lo sviluppo di qualcosa di più complesso, con l'ausilio del linguaggio C. I file mostrati hanno, nell'ordine di apparizione, i nomi: '`loader.s`', '`kernel.c`', '`linker.ld`' e '`Makefile`'.

Listato 65.67. File '`loader.s`' usato per la prima parte del codice, contenente l'intestazione *multiboot* e la preparazione dell'ambiente minimo di funzionamento, compresa la collocazione della pila dei dati. Il programma chiama la funzione '**_kernel**' presente nel file '`kernel.c`', passando come parametri il codice di riconoscimento del sistema di avvio e il puntatore alle altre informazioni che questo può fornire. Al ritorno dalla chiamata della funzione, il programma tenta di arrestare il microprocessore, ma se non ci riesce si mette in un ciclo senza fine che produce apparentemente lo stesso risultato.

```
.globl  _loader
.extern _kernel
#
```

```
# Dimensione della pila interna al kernel. Qui vengono
# previsti 16384 elementi (0x4000) da 32 bit, pari a 65536
# byte.
#
.equ STACK_SIZE,  0x4000
#
# Si inizia subito con il codice che si mescola con i dati.
#
_loader:
    jmp boot    # Salta all'inizio del codice.
    .align 4    # Fa in modo di riempire lo spazio mancante
                # al completamento di un blocco di 4 byte.
#
# Intestazione «multiboot», poco dopo l'inizio del
# file-immagine.
#
multiboot_header:
    .int 0x1BADB002                     # magic
    .int 0x00000003                     # flags
    .int -(0x1BADB002 + 0x00000003)     # checksum
#
# Inizia il codice di avvio.
#
boot:
    #
    # Regola ESP alla base della pila.
    #
    movl $(stack_max + STACK_SIZE), %esp
    #
    # Azzera gli indicatori (e per questo usa la pila appena
    # sistemata).
    #
    pushl $0
    popf
```

```
    #
    # Chiama il kernel scritto in C, passandogli le
    # informazioni ottenute dal sistema di avvio.
    #
    # void _kernel (unsigned int magic,
    #               void *multiboot_info)
    #
    pushl %ebx   # Puntatore alla struttura contenente le
                 # informazioni passate dal sistema di
                 # avvio.
    pushl %eax   # Codice di riconoscimento del sistema di
                 # avvio.
    #
    call _kernel # Chiama la funzione _kernel()
    #
halt:
    hlt          # Se il kernel termina, ferma il
                 # microprocessore.
    jmp halt     # Se non si è fermato, crea un ciclo
                 # senza fine.
#
# Alla fine del programma, viene collocato lo spazio per la
# pila dei dati, senza inizializzarlo. Per scrupolo si
# allinea ai 4 byte (32 bit).
#
.align 4
.comm stack_max, STACK_SIZE
#
```

Listato 65.68. File ‘`kernel.c`’ che potrebbe contenere idealmente il kernel di un piccolo sistema operativo. In questo caso il programma non fa alcunché e ignora anche la presenza di parametri nella chiamata.

```
void _kernel(void)
{
    ;
}
```

Listato 65.69. File ‘`linker.ld`’, da usare come script per GNU LD. Si può osservare che la sezione ‘**`.data`**’ viene distanziata da ‘**`.text`**’ e ‘**`.rodata`**’, in quanto si deve collocare in una pagina di memoria differente, per poter limitare i permessi di accesso in scrittura ai soli dati variabili.

```
ENTRY (_loader)
SECTIONS {
    . = 0x00100000;
    .text : { *(.text) }
    .rodata : { *(.rodata) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

Listato 65.70. File ‘Makefile’ da usare per la compilazione.

```
all: loader kernel link
#
clean:
        rm *.o
        rm kernel
#
loader:
        as -o loader.o loader.s
#
kernel:
        gcc -Wall -Werror -o kernel.o -c kernel.c \
            -nostdlib -nostartfiles -nodefaultlibs
#
link:
        ld --script=linker.ld -o kernel loader.o kernel.o
```

Per avviare il programma che si ottiene, si può usare GRUB 1, utilizzando le direttive seguenti nel suo file di configurazione:

```
…
title  mio kernel
kernel (fd0)/kernel
…
```

In questo caso si suppone di utilizzare un dischetto per l’avvio e che il file da avviare sia ‘kernel’, contenuto proprio nella radice.

65.5.3 Visualizzazione di messaggi

«

Quando si scrive un programma autonomo, come descritto sinteticamente nella sezione precedente, occorre considerare che il linguaggio C non può essere usato sfruttando le librerie consuete, pertan-

to occorre produrre tutto internamente, anche le funzioni per la visualizzazione dei messaggi. I listati seguenti vanno a sostituire il file ‘`kernel.c`’ della sezione precedente, allo scopo di visualizzare qualcosa sullo schermo (il contenuto dei primi campi della struttura informativa creata dal sistema di avvio), attraverso delle funzioni elementari definite internamente.

Per visualizzare un messaggio sullo schermo di un elaboratore x86 è necessario scrivere in una porzione di memoria che parte dall'indirizzo $B8000_{16}$ (a partire da 736 Kibyte), utilizzando coppie di byte, dove il primo byte è un codice che descrive i colori da usare per il carattere e il suo sfondo, mentre il secondo contiene il carattere da visualizzare. Nel programma il codice in questione è 07_{16} che mostra un carattere bianco su sfondo nero. Ciò che si deve osservare è che il programma tratta la coppia di byte come un numero a 16 bit, nel quale il carattere e il suo codice di visualizzazione sembrano invertiti, a causa del fatto che l'architettura è di tipo *little endian*.

Listato 65.72. File ‘`kernel.c`’ che include automaticamente i file ‘`display.c`’ e ‘`multiboot.c`’.

```
#include "display.c"
#include "multiboot.c"
//
//
//
void
_kernel (unsigned long magic, type_multiboot_info *info)
{
  clear_screen ();
  //
  print_string ("Salve!\n\0");
  //
```

```
  if (magic == 0x2BADB002)
    {
      print_string ("Sono stato avviato attraverso un \0");
      print_string ("sistema di avvio aderente alle \0");
      print_string ("specifiche \n\0");
      print_string ("\"multiboot\". Ecco solo alcune \0");
      print_string ("informazioni:\n\0");
      multiboot_information (info);
    }
  else
    {
      print_string ("Sono stato avviato attraverso un \0");
      print_string ("sistema di avvio che non e` \0");
      print_string ("conforme alle specifiche\n\0");
      print_string ("\"multiboot\".\n\0");
    }
}
```

Listato 65.73. File ‘display.c’, contenente le funzioni necessarie a visualizzare stringhe e numeri in forma di stringa.

```
static unsigned short *Screen = (unsigned short *) 0xB8000;
//
static const unsigned int Rows = 25, Columns = 80;
static       unsigned int Row = 0, Column = 0;
static       unsigned char Attrib = 0x07;
//
//
//
static unsigned short
screen_cell (unsigned char c, unsigned char attrib)
{
  //
  // Assembla i due caratteri in un numero a 16 bit.
  //
```

```
  return (short) c | (((short) attrib) * 0x100);
}
//
//
//
static void
clear_screen (void)
{
  unsigned int i;
  //
  for (i = 0; i < (Rows * Columns) ; i++)
    {
      //
      // Scrive uno spazio nella posizione.
      //
      *(Screen + i) = screen_cell (0x20, Attrib);
    }
}
//
//
//
static void
new_line (void)
{
  int i, j;
  //
  Column = 0;
  Row++;
  //
  if (Row >= Rows)
    {
      //
      // Copia il testo in su.
      //
```

```
      for (i = 0; i < (Rows - 1) * Columns ; i++)
        {
          j = i + Columns;
          //
          // Trascrive la cella della riga successiva.
          //
          *(Screen + i) = *(Screen + j);
        }
      //
      // Mette l'indice di riga nell'ultima posizione.
      //
      Row = Rows - 1;
      //
      // Pulisce la riga alla base dello schermo.
      //
      for (i = ((Rows - 1) * Columns) ;
           i < (Rows * Columns) ;
           i++)
        {
          //
          // Cancella la cella dello schermo.
          //
          *(Screen + i) = screen_cell (0x20, Attrib);
        }
    }
}
//
//
//
static void
print_char (unsigned char c)
{
  //
  // Put the character.
```

```
  //
  if (c == '\n' || c == '\r')
    {
      new_line ();
    }
  else
    {
      *(Screen + (Row * Columns + Column))
       = screen_cell (c, Attrib);
      //
      // Move cursor.
      //
      Column++;
      if (Column >= Columns)
        {
          new_line ();
        }
    }
}
//
//
//
static void
print_string (char *string)
{
  unsigned int i;
  //
  for (i = 0; i < 100000 ; i++)
    {
      if (string[i] != 0)
        {
          print_char (string[i]);
        }
      else
```

```
        {
          break;
        }
    }
}
//
//
//
static void
reverse_string (char *string)
{
  unsigned int i, j;
  unsigned char c;
  //
  // Scandisce la stringa alla ricerca del valore a zero.
  //
  for (i = 0; string[i] != 0; i++)
    {
      ;
    }
  //
  // L'indice "i" punta alla cella a zero.
  // Viene rimesso l'indice "i" in modo da puntare
  // all'ultimo carattere.
  //
  i--;
  //
  // Si inverte l'ordine delle cifre.
  //
  for (j = 0; j < i; j++, i--)
    {
      c = string[i];
      string[i] = string[j];
      string[j] = c;
```

```
    }
  //
}
//
//
//
static void
num_to_string (unsigned long num, unsigned int base,
               char *string)
{
  unsigned int i;
  unsigned char remainder;
  //
  if (num == 0)
    {
      string[0] = '0';
      string[1] = 0;
      return;
    }
  //
  for (i = 0; num != 0; i++)
    {
      remainder = num % base;
      num = num / base;
      //
      if (remainder <= 9)
        {
          string[i] = '0' + remainder;
        }
      else
        {
          string[i] = 'A' + remainder - 10;
        }
    }
```

```
  //
  // Aggiunge la terminazione, tenendo conto che l'indice
  // "i" è già posizionato dopo l'ultima cifra inserita.
  //
  string[i] = 0;
  //
  reverse_string (string);
}
//
//
//
static void
print_num (unsigned long num, char base)
{
  char string[100];
  //
  if (base == 'x')
    {
      num_to_string (num, 16, string);
      print_string ("0x\0");
      print_string (string);
    }
  else if (base == 'o')
    {
      num_to_string (num, 8, string);
      print_string ("0o\0");
      print_string (string);
    }
  else if (base == 'b')
    {
      num_to_string (num, 2, string);
      print_string ("0b\0");
      print_string (string);
    }
```

```
  else
    {
      num_to_string (num, 10, string);
      print_string (string);
    }
}
```

Listato 65.74. File ‘multiboot.c’, contenente la definizione parziale della struttura delle informazioni *multiboot* e la funzione necessaria a visualizzarne il contenuto.

```
//
// The multiboot information.
//
typedef struct multiboot_info
{
  unsigned long flags;
  unsigned long mem_lower;
  unsigned long mem_upper;
  unsigned long boot_device;
            char *cmdline;
} type_multiboot_info;
//
//
//
static void
multiboot_information (type_multiboot_info *info)
{
  print_string ("flags:       \0");
  print_num (info->flags, 'b');
  print_string ("\n\0");
  //
  if ((info->flags & 1) > 0)
    {
      print_string ("mem_lower:   \0");
```

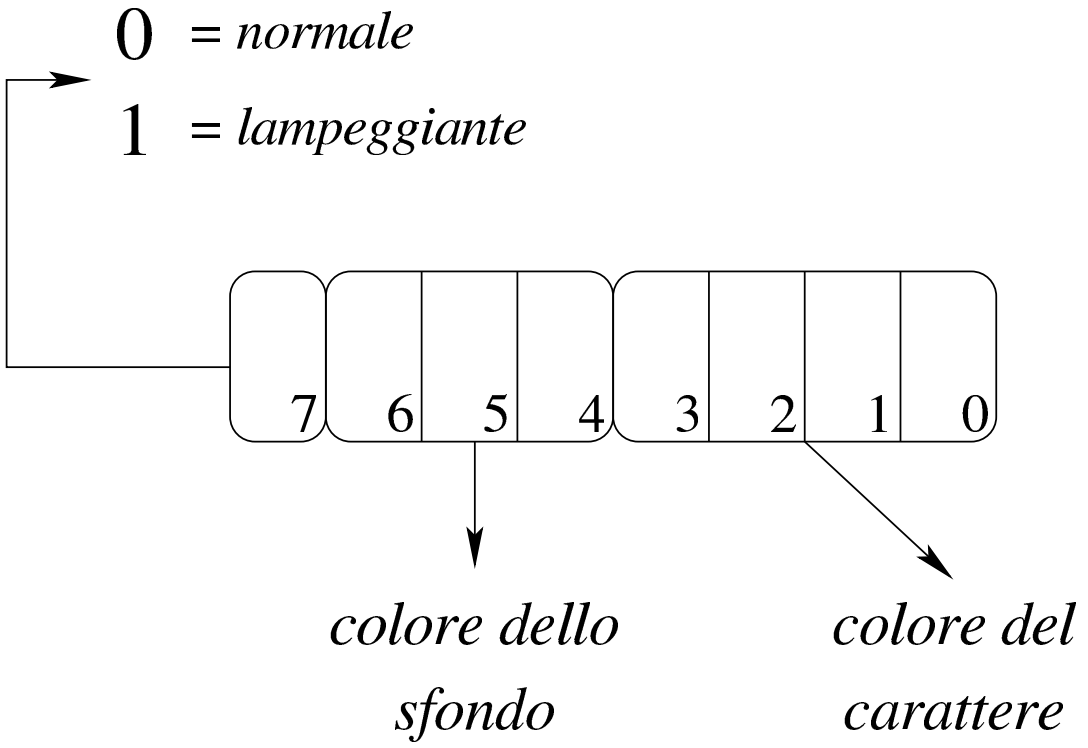
```
      print_num (info->mem_lower, 'x');
      print_string (" \0");
      print_num (info->mem_lower, 'd');
      print_string (" Kibyte\0");
      print_string ("\n\0");
      //
      print_string ("mem_upper:   \0");
      print_num (info->mem_upper, 'x');
      print_string (" \0");
      print_num (info->mem_upper, 'd');
      print_string (" Kibyte\0");
      print_string ("\n\0");
    }
  if ((info->flags & 2) > 0)
    {
      print_string ("boot_device: \0");
      print_num (info->boot_device, 'x');
      print_string ("\n\0");
    }
  if ((info->flags & 4) > 0)
    {
      print_string ("cmdline:     \0");
      print_string (info->cmdline);
      print_string ("\n\0");
    }
}
```

Utilizzando questo programma si potrebbe visualizzare una schermata simile a quella seguente:

```
Salve!
Sono stato avviato attraverso un sistema di avvio aderente
alle specifiche "multiboot". Ecco solo alcune informazioni:
flags:       0b11111100111
mem_lower:   0x27F 639 Kibyte
mem_upper:   0x7C00 31744 Kibyte
boot_device: 0xFFFFFF
cmdline:     (fd0)/kernel
```

65.5.4 Colori dello schermo

Nella sezione precedente si accenna al fatto che a partire dall'indirizzo di memoria $B8000_{16}$, ciò che si scrive serve a ottenere una rappresentazione sullo schermo. Ogni carattere utilizza due byte, in quanto uno dei due contiene il carattere vero e proprio e l'altro l'attributo che ne definisce il colore. Il byte del colore va usato suddividendo i bit nel modo seguente:



Come si vede, se è attivo il bit più significativo si ottiene un carattere lampeggiante, quindi i tre bit successivi descrivono lo sfondo e i quattro bit meno significativi descrivono invece il colore del carattere (in primo piano). Pertanto, i colori dello sfondo sono in quantità minore rispetto a quelli utilizzabili per il primo piano.

Tabella 65.77. Colore associato al primo piano o allo sfondo.

Codice	Sfondo	Primo piano
0_{16}	nero	nero
1_{16}	blu	blu
2_{16}	verde	verde
3_{16}	ciano (azzurro)	ciano (azzurro)
4_{16}	rosso	rosso
5_{16}	magenta (violetto)	magenta (violetto)
6_{16}	marrone	marrone
7_{16}	bianco	bianco
8_{16}	nero con lampeggio	grigio scuro
9_{16}	blu con lampeggio	blu chiaro
A_{16}	verde con lampeggio	verde chiaro
B_{16}	ciano con lampeggio	ciano chiaro
C_{16}	rosso con lampeggio	rosa
D_{16}	magenta con lampeggio	magenta chiaro
E_{16}	marrone con lampeggio	giallo
F_{16}	bianco con lampeggio	bianco luminoso

Per esempio, un colore indicato come 28_{16} genera un testo di colore grigio scuro su sfondo verde, mentre $A0_{16}$ genera un testo lampeggiante nero su sfondo verde.

65.6 Compilazione C dal basso in alto

Il valore del linguaggio C sta nel consentire una programmazione molto vicina a livello del linguaggio macchina, in modo relativamente indipendente dall'architettura. Ma ciò si può comprendere solo se si conosce il contesto operativo del linguaggio assemblatore, in modo particolare per quanto riguarda la gestione della memoria e tanto più per il modo in cui si utilizza la pila dei dati.

Per la compilazione dei programmi di esempio si fa riferimento a GCC[3] (*GNU compiler collection*) e precisamente al programma frontale '**gcc**'.

65.6.1 Compilazione di un programma che non fa uso di librerie

Un programma in linguaggio C che non faccia uso di librerie di alcun tipo, deve seguire alcune regole che riguardano i programmi scritti in linguaggio assemblatore. Il listato seguente contiene la procedura per il calcolo del fattoriale, partendo da un valore già presente in memoria (si calcola precisamente il fattoriale di 5), ma il risultato non viene visualizzato in alcun modo, dal momento che questa sarebbe un'operazione che richiede proprio l'uso di librerie apposite:

```
int x = 5;
int i = 0;
void _start (void)
{
    i = (x - 1);
    while (i > 0)
      {
        x = x * i;
        i--;
```

```
        }
}
```

Se si conoscono i rudimenti del linguaggio C, si può osservare che, al posto della funzione ***main()***, appare invece ***_start()***, come si fa in un programma scritto in linguaggio assemblatore.

Supponendo che il file che contiene quanto mostrato si chiami '`fact.c`', la compilazione potrebbe iniziare dalla trasformazione in linguaggio assemblatore:

```
$ gcc -Wall -Werror -S -o fact.s fact.c ↩
↪     -nostdlib -nostartfiles -nodefaultlibs[Invio]
```

Si otterrebbe il file '`fact.s`', in linguaggio assemblatore, che, a seconda della versione di GCC, potrebbe essere molto simile al listato seguente:

```
        .file   "fact.c"
.globl x
        .data
        .align 4
        .type   x, @object
        .size   x, 4
x:
        .long   5
.globl i
        .bss
        .align 4
        .type   i, @object
        .size   i, 4
i:
        .zero   4
        .text
.globl _start
```

```
        .type   _start, @function
_start:
        pushl   %ebp
        movl    %esp, %ebp
        movl    x, %eax
        decl    %eax
        movl    %eax, i
        jmp     .L2
.L3:
        movl    x, %edx
        movl    i, %eax
        imull   %edx, %eax
        movl    %eax, x
        movl    i, %eax
        decl    %eax
        movl    %eax, i
.L2:
        movl    i, %eax
        testl   %eax, %eax
        jg      .L3
        popl    %ebp
        ret
        .size   _start, .-_start
        .ident  "GCC: (GNU) 4.1.2 20061115 ↩
↪(prerelease) (Debian 4.1.1-21)"
        .section        .note.GNU-stack,"",@progbits
```

Tale file in linguaggio assemblatore può essere compilato con GNU AS e GNU LD nel modo consueto:

```
$ as -o fact.o fact.s[Invio]
```

```
$ ld -o fact fact.o[Invio]
```

Si può ispezionare il programma ottenuto con Objdump:

$ **objdump -x fact** [*Invio*]

```
fact:     file format elf32-i386
fact
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048094

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x000000cd memsz 0x000000cd flags r-x
    LOAD off    0x000000d0 vaddr 0x080490d0 paddr 0x080490d0 align 2**12
         filesz 0x00000004 memsz 0x00000008 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000039  08048094  08048094  00000094  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  080490d0  080490d0  000000d0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  080490d4  080490d4  000000d4  2**2
                  ALLOC
  3 .comment      0000003a  00000000  00000000  000000d4  2**0
                  CONTENTS, READONLY
SYMBOL TABLE:
08048094 l    d  .text  00000000 .text
080490d0 l    d  .data  00000000 .data
080490d4 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  *ABS*  00000000 .shstrtab
00000000 l    d  *ABS*  00000000 .symtab
00000000 l    d  *ABS*  00000000 .strtab
00000000 l    df *ABS*  00000000 fact.c
080490d0 g     O .data  00000004 x
080490d4 g     O .bss   00000004 i
08048094 g     F .text  00000039 _start
080490d4 g       *ABS*  00000000 __bss_start
080490d4 g       *ABS*  00000000 _edata
080490d8 g       *ABS*  00000000 _end
```

È possibile fare in modo che GCC interpelli automaticamente GNU AS, in modo da generare un file oggetto senza mostrare la creazione del file in linguaggio assemblatore (la trasformazione in linguaggio assemblatore avviene ugualmente, in un file temporaneo che poi viene cancellato in modo automatico). Pertanto, la compilazione si ridurrebbe ai due comandi seguenti:

```
$ gcc -Wall -Werror -c -o fact.o fact.c↩
↪     -nostdlib -nostartfiles -nodefaultlibs[Invio]

$ ld -o fact fact.o[Invio]
```

È il caso di osservare che il programma eseguibile ottenuto dal sorgente mostrato, produce un errore, dal momento che manca la chiamata della funzione del sistema operativo che ne conclude l'attività.

65.6.2 Uso di GDB e di DDD

Per poter sfruttare programmi come GDB, allo scopo di analizzare il funzionamento del programma, è necessario inserire delle informazioni aggiuntive durante la fase di trasformazione nel formato del linguaggio assemblatore. In pratica, si tratta di utilizzare l'opzione '**-gstabs**', o altre simili, nella riga di comando di GCC. Riprendendo l'esempio della sezione precedente, la compilazione verrebbe eseguita con il comando seguente:

```
$ gcc -Wall -Werror -gstabs -S -o fact.s fact.c ↩
↪     -nostdlib -nostartfiles -nodefaultlibs[Invio]
```

In questo caso, nel file in linguaggio assemblatore si troverebbero delle informazioni in più:

```
        .file   "fact.c"
        .stabs  "fact.c",100,0,2,.Ltext0
        .text
.Ltext0:
        .stabs  "gcc2_compiled.",60,0,0,0
        .stabs  "int:t(0,1)=r(0,1);-2147483648;2147483647;",128,0,0,0
        .stabs  "char:t(0,2)=r(0,2);0;127;",128,0,0,0
        .stabs  "long int:t(0,3)=r(0,3);-2147483648;2147483647;",128,0,0,0
        .stabs  "unsigned int:t(0,4)=r(0,4);0;4294967295;",128,0,0,0
        .stabs  "long unsigned int:t(0,5)=r(0,5);0;4294967295;",128,0,0,0
        .stabs  "long long int:t(0,6)=r(0,6);-0;4294967295;",128,0,0,0
        .stabs  "long long unsigned int:t(0,7)=r(0,7);0;-1;",128,0,0,0
        .stabs  "short int:t(0,8)=r(0,8);-32768;32767;",128,0,0,0
        .stabs  "short unsigned int:t(0,9)=r(0,9);0;65535;",128,0,0,0
        .stabs  "signed char:t(0,10)=r(0,10);-128;127;",128,0,0,0
        .stabs  "unsigned char:t(0,11)=r(0,11);0;255;",128,0,0,0
        .stabs  "float:t(0,12)=r(0,1);4;0;",128,0,0,0
        .stabs  "double:t(0,13)=r(0,1);8;0;",128,0,0,0
        .stabs  "long double:t(0,14)=r(0,1);12;0;",128,0,0,0
        .stabs  "void:t(0,15)=(0,15)",128,0,0,0
.globl x
        .data
        .align 4
        .type   x, @object
        .size   x, 4
x:
        .long   5
.globl i
        .bss
        .align 4
        .type   i, @object
        .size   i, 4
i:
        .zero   4
        .text
        .stabs  "_start:F(0,15)",36,0,0,_start
.globl _start
        .type   _start, @function
_start:
```

```
        .stabn  68,0,4,.LM0-_start
.LM0:
        pushl   %ebp
        movl    %esp, %ebp
        .stabn  68,0,5,.LM1-_start
.LM1:
        movl    x, %eax
        decl    %eax
        movl    %eax, i
        .stabn  68,0,6,.LM2-_start
.LM2:
        jmp     .L2
.L3:
        .stabn  68,0,8,.LM3-_start
.LM3:
        movl    x, %edx
        movl    i, %eax
        imull   %edx, %eax
        movl    %eax, x
        .stabn  68,0,9,.LM4-_start
.LM4:
        movl    i, %eax
        decl    %eax
        movl    %eax, i
.L2:
        .stabn  68,0,6,.LM5-_start
.LM5:
        movl    i, %eax
        testl   %eax, %eax
        jg      .L3
        .stabn  68,0,11,.LM6-_start
.LM6:
        popl    %ebp
        ret
        .size   _start, .-_start
.Lscope0:
        .stabs  "x:G(0,1)",32,0,0,0
        .stabs  "i:G(0,1)",32,0,0,0
        .stabs  "",100,0,0,.Letext0
.Letext0:
        .ident  "GCC: (GNU) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)"
```

```
        .section        .note.GNU-stack,"",@progbits
```

Per la compilazione successiva non ci sono cambiamenti; va quindi osservato che non è più compito di GNU AS l'inserimento di tali informazioni:

```
$ as -o fact.o fact.s[Invio]

$ ld -o fact fact.o[Invio]
```

Per utilizzare GDB o DDD si procede come nel caso di un programma che parte direttamente da un sorgente in linguaggio assemblatore:

```
$ gdb fact[Invio]

(gdb) break _start[Invio]

 Breakpoint 1 at 0x8048097: file fact.c, line 5.


(gdb) run[Invio]

 Breakpoint 1, _start () at fact.c:5
 5           i = (x - 1);


(gdb) stepi[Invio]

 0x0804809c      5           i = (x - 1);


(gdb) stepi[Invio]

 0x0804809d in _start () at fact.c:5
 5           i = (x - 1);


(gdb) stepi[Invio]
```

```
6           while (i > 0)
```

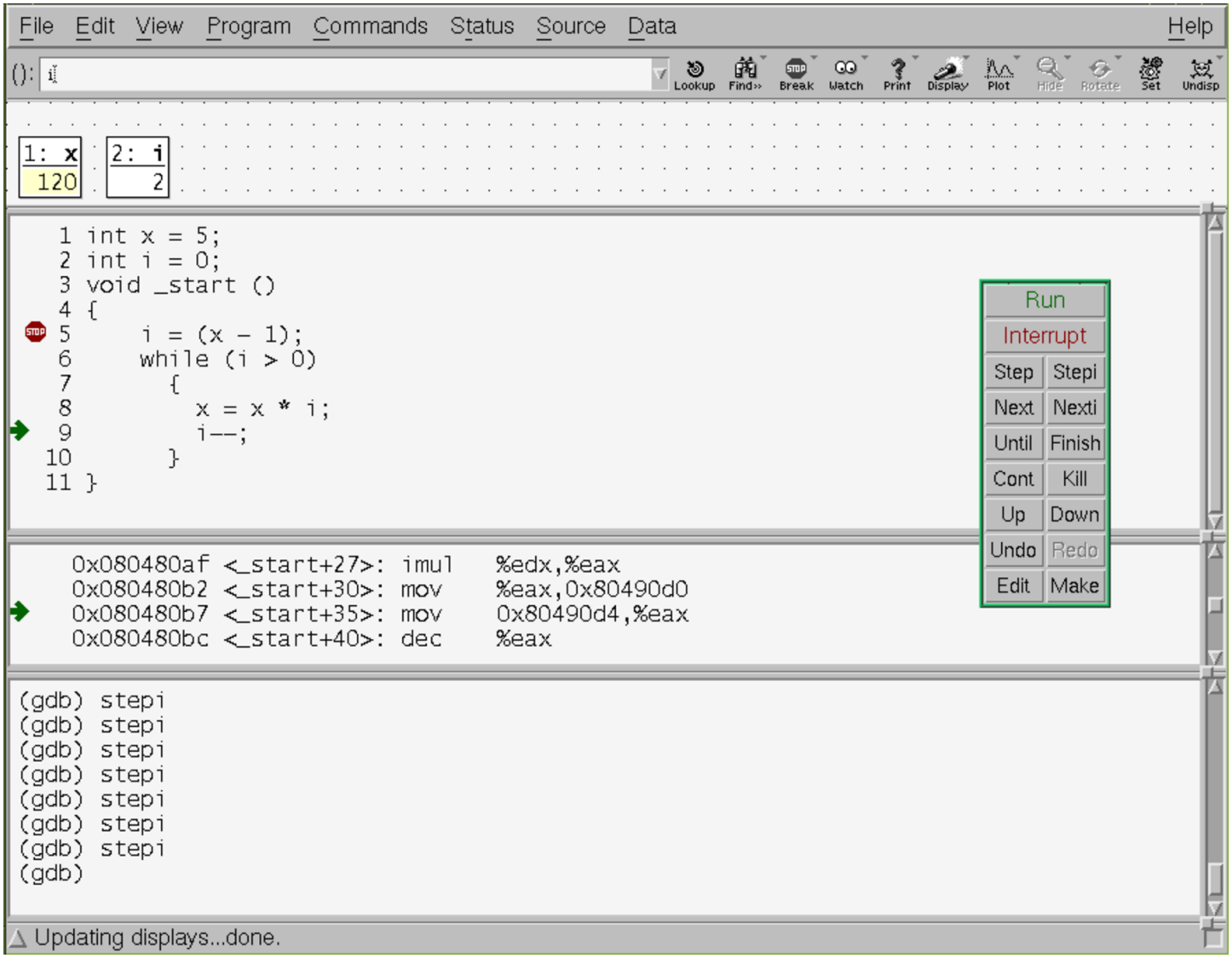
Come si può osservare, occorrono più comandi di avanzamento per passare alla riga successiva del codice originale, perché in realtà si fa riferimento alle istruzioni in linguaggio macchina.

```
(gdb) print i[Invio]

  $1 = 4

(gdb) print x[Invio]

  $2 = 5

(gdb) quit[Invio]
```

Naturalmente, se si può utilizzare DDD, tutto diventa più semplice:

```
$ ddd fact[Invio]
```

Figura 65.89. DDD che mette in evidenza lo stato di due variabili (si attiva la loro visualizzazione facendo un clic sul pulsante a icona denominato DISPLAY) durante il funzionamento, passo passo, del programma.



65.6.3 Da «_start» a «main»

Per fare in modo che un programma in linguaggio C inizi dalla funzione ***main()***, così come si prevede sia, si può istruire il collegatore (*linker*), attraverso uno script apposito che, in un sistema GNU/Linux, potrebbe essere come quello seguente:

```
ENTRY (main)
SECTIONS {
    . = 0x08048000 + SIZEOF_HEADERS;
    .text . : { *(.text) }
    .data ALIGN (0x1000) : { *(.data) }
    .bss . : {
        _sbss = .;
        *(.bss)
        *(COMMON)
        _ebss = .;
    }
}
```

Il nuovo sorgente C:

```
int x = 5;
int i = 0;
int main ()
{
    i = (x - 1);
    while (i > 0)
      {
        x = x * i;
        i--;
      }
    return x;
}
```

Per la compilazione, i passaggi sarebbero quelli seguenti, supponendo che lo script per GNU LD sia contenuto nel file '`config.ld`':

```
$ gcc -Wall -Werror -S -o fact.s fact.c ↩
↪    -nostdlib -nostartfiles -nodefaultlibs[Invio]
```

```
$ as -o fact.o fact.s[Invio]

$ ld -T config.ld -o fact fact.o[Invio]
```

Tuttavia, rimane ancora il problema della conclusione del programma che non avviene in modo grazioso. Se si osserva la nuova versione del programma, la funzione (che ora si chiama ***main()***) restituisce un valore intero, corrispondente al risultato del calcolo eseguito, solo che non è stato chiarito in che modo quel valore debba essere acquisito dal sistema operativo. Si può quindi procedere in un modo diverso, creando un piccolo programma in linguaggio assemblatore, da associare a quello in linguaggio C:

```
.section .text
.globl _start
.extern main
_start:
    call  main
    mov   %eax, %ebx
    mov   $1, %eax
    int   $0x80
```

Supponendo che questo file si chiami '**start.s**', la compilazione complessiva potrebbe essere svolta nel modo seguente:

```
$ gcc -Wall -Werror -gstabs -S -o fact.s fact.c ↩
↪    -nostdlib -nostartfiles -nodefaultlibs[Invio]

$ as -o fact.o fact.s[Invio]

$ as --gstabs -o start.o start.s[Invio]

$ ld -o fact start.o fact.o[Invio]
```

Come si vede sono state aggiunte le opzioni '**-gstabs**' e

‘**--gstabs**’, dove appropriato; inoltre non serve più lo script per GNU LD. Se si avvia il programma, questo si arresta correttamente restituendo il fattoriale di 5:

```
$ ./fact ; echo $?[Invio]

  120
```

65.6.4 Compilazione naturale di un programma in linguaggio C

Quando non si utilizzano le opzioni ‘**-nostdlibs**’, ‘**-nostartfiles**’ e ‘**-nodefaultlibs**’, la compilazione attraverso GCC avviene in modo più intuitivo, con l’inclusione automatica di tutto quello che è necessario per far sì che il programma parta dalla propria funzione ***main()***; inoltre, se non si specifica il nome che si vuole produrre, si ottiene direttamente un file eseguibile con il nome ‘`a.out`’, secondo la tradizione.

In condizioni normali vengono inclusi nella compilazione alcuni file-oggetto che hanno un nome corrispondente al modello ‘`crt*.o`’ e la libreria Libc. All’interno di uno di quei file-oggetto si trova la funzione ***_start()***, dalla quale si arriva poi alla chiamata di ***main()*** in modo analogo a quanto mostrato nella sezione precedente, ma questi file potrebbero coinvolgere anche la libreria Libc.

L’opzione ‘**-nostartfiles**’ serve a impedire che vengano incorporati automaticamente i file che contengono la funzione ***_start()*** e tutto ciò che altrimenti si prevede di far fare al programma prima di entrare nella funzione ***main()***. L’opzione ‘**-nodefaultlibs**’ serve a impedire l’inclusione automatica della libreria Libc. L’op-

zione '**-nostdlibs**' richiede entrambe le cose ed è stata usata negli esempi in modo ridondante.

Ecco la classica compilazione che produce direttamente il file eseguibile con il nome '`a.out`':

```
$ gcc -Wall -Werror -gstabs fact.c[Invio]
```

Per buona abitudine è bene usare sempre l'opzione '**-Wall**' e possibilmente anche '**-Werror**'; inoltre, l'uso di '**-gstabs**' diventa essenziale per potersi avvalere di programmi come '**GDB**'.

Si può verificare che questo basta per arrivare al risultato voluto:

```
$ ./a.out ; echo $?[Invio]

  120
```

Se poi si vogliono usare comandi tradizionali, da '**gcc**' occorre passare a '**cc**', ma in un sistema GNU si tratta normalmente di un collegamento simbolico a '**gcc**' stesso.

Tabella 65.95. Riepilogo delle opzioni utilizzate con '**gcc**' nel corso del capitolo.

Opzione	Descrizione
`-S`	Genera un file in linguaggio assemblatore (prevale sull'opzione '**-c**').
`-o` ***nome_file***	Dichiara il nome del file che si vuole ottenere.

Opzione	Descrizione
`-c`	Fa sì che la compilazione salti la fase di collegamento (*link*). In condizioni normali serve a generare solo i file-oggetto. Se si usa questa opzione, ma non si specifica l'opzione '**`-o`**', il file-oggetto ha un nome con la stessa radice del file sorgente e l'estensione '`.o`'.
`-Wall`	Richiede di mostrare tutti i messaggi che avvertono dell'uso imperfetto del linguaggio (*warning*).
`-Werror`	Fa sì che tutte le segnalazioni di avvertimento siano trattate come errori e portino al fallimento della compilazione.
`-gstabs`	Inserisce delle annotazioni, con le quali i programmi come GDB possono abbinare il sorgente originale all'esecuzione controllata del programma.

65.7 Compilazione C dall'alto in basso

Tradizionalmente, la compilazione di un programma scritto in linguaggio C avviene utilizzando il comando '**`cc`**' come nell'esempio seguente, sapendo che se non si usa l'opzione '**`-o`**' si ottiene il file '`a.out`':

```
$ cc mio.c[Invio]
```

Tuttavia, l'elaborazione del file in linguaggio C richiede diversi passaggi, prima di arrivare al file eseguibile finale; passaggi che è bene tenere in considerazione.

In un sistema GNU il compilatore standard è GCC (*GNU compiler collection*) che si usa sia per il C, sia per altri linguaggi. Nel caso del linguaggio C, il programma frontale è precisamente '**gcc**', al quale corrisponde comunque il collegamento '**cc**'.

Qui non si esaurisce il problema e si accenna soltanto alle situazioni più comuni. Per un approfondimento si vedano i documenti citati nella bibliografia che conclude il capitolo.

65.7.1 Le fasi della compilazione

«

La compilazione di un programma scritto in linguaggio C prevede diverse fasi: precompilazione, trasformazione in linguaggio assemblatore, trasformazione in file-oggetto, collegamento (*link*) di uno o più file-oggetto in un file eseguibile. Per conservare i file intermedi della compilazione si può usare l'opzione '**-save-temps**' di '**gcc**', come nell'esempio seguente:

```
$ gcc -save-temps mio.c[Invio]
```

In questo caso si ottengono i file '`mio.i`', '`mio.s`' e '`mio.o`', contenenti rispettivamente il risultato elaborato dal precompilatore, la trasformazione in linguaggio assemblatore e il file-oggetto finale. Se poi il programma contenuto nel file sorgente è completo, si ottiene anche il file '`a.out`' che costituisce il programma eseguibile.

Eventualmente, alcune opzioni di '**gcc**' consentono di fermare l'elaborazione a uno stadio prestabilito: '**-E**' serve a ottenere solo l'elaborazione da parte del precompilatore; '**-S**' serve a ottenere il sorgente in linguaggio assemblatore; '**-c**' serve a compilare, ma senza eseguire il collegamento finale (pertanto si ottiene il file-oggetto rilocabile).

65.7.2 Precompilatore

Ogni compilatore C «standard» prevede che il file sorgente venga elaborato, prima della compilazione vera e propria, attraverso un precompilatore, il quale elabora il sorgente e genera un altro sorgente ottenuto dall'interpretazione delle istruzioni di «precompilazione». Queste istruzioni di precompilazione costituiscono un linguaggio indipendente dal C vero e proprio. Il precompilatore di GCC è '**cpp**' e di norma viene chiamato automaticamente da '**gcc**' stesso, come già accennato nella sezione precedente.

```
#include <stdio.h>
int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

Nell'esempio mostrato, l'istruzione '**#include <stdio.h>**' riguarda il precompilatore e richiede l'inclusione del file '`stdio.h`' in quella posizione (il file si deve trovare all'interno di una directory prestabilita). Con l'opzione '**-E**' di '**gcc**' (oppure anche con '**-save-temps**') si può vedere il risultato della precompilazione:

```
$ gcc -E -o mio.i mio.c[Invio]
```

Il file '`mio.i`' che si genera dall'elaborazione ha un aspetto simile al pezzo che si vede nel listato successivo:

```
# 1 "mio_file.c"
# 1 "<built-in>"
# 1 "<command line>"
...
...
```

```
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
…
…
extern void funlockfile (FILE *__stream)
            __attribute__ ((__nothrow__));
# 834 "/usr/include/stdio.h" 3 4

# 2 "mio_file.c" 2
int main (void)
{
    printf ("Ciao a tutti!\n");
    return 0;
}
```

65.7.3 Compilazione dei file intermedi

«

Di norma, ogni compilatore tradizionale del linguaggio C si prende cura di tutte le fasi della compilazione, chiamando a sua volta i programmi necessari. Pertanto, con lo stesso programma frontale è possibile avviare manualmente la compilazione da fasi successive. Per esempio:

1. `$` **`cc mio_file.i`**[*Invio*]
2. `$` **`cc mio_file.s`**[*Invio*]
3. `$` **`cc mio_file.o`**[*Invio*]

Negli esempi si mostra l'uso del comando '**`cc`**', ma '**`gcc`**' è perfettamente conforme a questa convenzione tradizionale. Come si può

intuire, dall'estensione del nome del file il programma frontale determina quali azioni deve intraprendere: nel primo caso avvia la compilazione saltando solo la fase iniziale dell'analisi del precompilatore; nel secondo caso avvia l'assemblatore (e quindi continua con il collegatore); nell'ultimo caso avvia soltanto il collegatore (*linker*).

Naturalmente è possibile mescolare file differenti assieme, se la somma di questi deve portare a un solo file-eseguibile finale. Per esempio, si può compilare un programma composto dai file '`uno.c`', '`due.i`', '`tre.s`' e '`quattro.o`', dove ognuno viene elaborato in base alle proprie esigenze e alla fine il tutto viene collegato assieme:

```
$ cc uno.c due.i tre.s quattro.o [Invio]
```

65.7.4 L'uso di librerie

In generale, la compilazione di un programma scritto secondo il linguaggio C implica automaticamente l'utilizzo della libreria Libc e il collegamento (*link*) con dei file-oggetto predefiniti, che contengono il codice necessario a preparare il programma prima di passare all'esecuzione della funzione ***main()***.

Con '**gcc**', per escludere l'utilizzo di qualunque libreria predefinita vanno usate le opzioni '**-nostartfiles**' e '**-nodefaultlibs**'; eventualmente l'opzione '**-nostdlibs**' dovrebbe valere per entrambe queste opzioni e può essere usata assieme a loro, benché sia ridondante.

Quando si fa uso di funzioni che non sono state dichiarate nel proprio programma, si tratta sempre di qualcosa che è contenuto in una libre-

ria, di solito quella predefinita (Libc), ma per usarle correttamente è indispensabile che sia inserita all'inizio del file la dichiarazione del loro prototipo. Per questo, a seconda delle funzioni che si utilizzano, si includono i file che contengono i prototipi necessari; nel caso della funzione ***printf()*** si include comunemente il file '`stdio.h`'.

Se si utilizza una funzione che appartiene a una libreria prevista nella compilazione, della quale però non si dichiara il prototipo, si può anche ottenere una compilazione «corretta», ma non è detto che, durante il funzionamento del programma, il passaggio degli argomenti attraverso i parametri della funzione avvenga in modo altrettanto corretto. In pratica, è molto probabile che la chiamata di tali funzioni produca risultati errati.

65.7.5 Librerie statiche e librerie dinamiche

«

Le librerie statiche sono file-oggetto raccolti in archivi generati con il programma '**`ar`**', dove i nomi dei file di tali archivi hanno estensione '`.a`'. L'uso di queste librerie implica l'incorporazione del codice utilizzato nel programma finale.

Per compilare un programma che utilizza delle librerie statiche è sufficiente indicare i nomi dei file che le contengono, assieme agli altri file del programma:

```
$ gcc mio.c /usr/lib/libncurses.a [Invio]
```

In alternativa, secondo la modalità normale, quando i file di tali librerie si trovano nelle directory previste, si può usare l'opzione '**`-l`**', a cui si attacca il nome della libreria, ottenuto dal nome del file to-

gliendo l'estensione e il prefisso '`lib`'. Pertanto, l'esempio appena mostrato andrebbe trasformato così:

```
$ gcc -static mio.c -lncurses[Invio]
```

Le librerie dinamiche sono realizzate in modo differente rispetto a quelle statiche e sono contenute normalmente in file con estensione '`.so`'. La compilazione con l'uso di librerie dinamiche avviene in modo analogo a quanto visto per quelle statiche:

```
$ gcc mio.c /usr/lib/libncurses.so[Invio]
```

Oppure:

```
$ gcc -dynamic mio.c -lncurses[Invio]
```

Come si può intuire dagli esempi mostrati, se una stessa libreria è fornita sia in versione statica, sia in versione dinamica, le opzioni '**`-static`**' e '**`-dynamic`**' servono a precisare che tipo di compilazione si vuole. Se però si omette di specificarlo, in generale vengono utilizzate le librerie dinamiche.

L'opzione '**`-l`**' implica una ricerca dei file delle librerie all'interno di directory prestabilite, ma può succedere che sia necessario esplicitarlo nella riga di comando. In tal caso si può usare l'opzione '**`-L`**':

```
$ gcc mio.c -L/opt/mia/lib -lmia[Invio]
```

Nell'esempio appena mostrato, la compilazione richiede l'uso della libreria '**`mia`**' ('`libmia.so`' o '`libmia.a`') che va cercata prima nella directory '`/opt/mia/lib/`'.

Dal momento che l'uso delle librerie si affianca all'inclusione dei file che ne contengono il prototipo, conviene ricordare anche l'op-

zione '**-I**', con la quale si richiede di cercare i file da includere a cominciare dalla directory specificata:

```
$ gcc mio.c -I/opt/mia/include -L/opt/mia/lib -lmia[Invio]
```

In questo nuovo esempio, si specifica anche che i file da includere vanno cercati a cominciare dalla directory '`/opt/mia/include/`'.

Naturalmente, il problema dei percorsi di ricerca per i file da includere riguarda solo quelli che nel sorgente si indicano tra parentesi angolari, come in questo esempio:

```
#include <stdio.h>
```

Diversamente, se il nome fosse messo tra apici doppi, il file verrebbe cercato nel percorso indicato esplicitamente nel sorgente stesso.

A ogni modo, quando la compilazione manifesta dei problemi che non sembrano dovuti a errori sintattici, conviene usare l'opzione '**-v**', con la quale si vede esattamente cosa tenta di fare il programma frontale e dove si interrompe la compilazione. Ciò può essere molto utile per capire, per esempio, quando il problema deriva da file mancanti (librerie o altro).

Per il procedimento necessario alla produzione di una libreria, statica o dinamica, si veda la sezione 65.2.

65.7.6 L'ordine dei file e delle librerie nella compilazione

La compilazione corretta richiede che i file e le librerie siano indicati nella riga di comando secondo un ordine logico: prima il file che contiene la funzione ***main()***, poi i file o le librerie contenenti le funzioni chiamate dal primo file, poi i file o le librerie contenenti le funzioni chiamate dai predecessori e così di seguito. Per esempio, se il file '`uno.c`' contiene la funzione ***main()*** e a sua volta chiama la funzione ***due()*** contenuta nel file '`due.s`', la riga di comando per la compilazione deve avere l'aspetto seguente:

```
$ gcc uno.c due.s …
```

Se poi la funzione ***due()*** si avvale della funzione ***tre()***, contenuta nella libreria '`libtre.a`', la riga di comando si sviluppa così:

```
$ gcc uno.c due.s -ltre …
```

Naturalmente, anche la funzione ***tre()*** potrebbe avvalersi di una funzione contenuta in una seconda libreria. Per esempio potrebbe usare la funzione ***quattro()*** della libreria '`libquattro.so`':

```
$ gcc uno.c due.s -ltre -lquattro …
```

Questa è una regola generale da considerare in fase di collegamento (*link*). Si osservi che GNU LD (ovvero il programma usato automaticamente da '**`gcc`**' per questo scopo) non richiede necessariamente tale accorgimento, ma ugualmente è meglio curarsi di rispettare il principio.

65.7.7 Prevenzione e ricerca degli errori

«

Il linguaggio C può essere usato «bene» o «male», così come ogni altro linguaggio. Nel caso particolare del C, certi modi leciti di scrivere un programma possono essere facilmente motivo di errori banali, evitabili se si chiede al compilatore di segnalare anche le piccole mancanze. In pratica, con '**`gcc`**' è bene usare sempre l'opzione '**`-Wall`**' per ottenere la segnalazione di una serie numerosa di avvertimenti; eventualmente a questa opzione si può aggiungere '**`-Werror`**', con la quale si trasformano gli avvertimenti in errori, così da evitare che in loro presenza la compilazione vada a buon fine.

Per analizzare il funzionamento del programma con GDB o altri analizzatori simili, conviene aggiungere l'opzione '**`-gstabs`**', oppure un'altra opzione che inizi per '**`-g`**...', in base alle caratteristiche del programma usato per l'analisi.

Infine, disponendo di un sistema GNU, o di un altro sistema compatibile con il modello di Unix, è bene abilitare lo scarico dell'immagine dei processi elaborativi in un file (*core dump*). Così facendo, quando durante il funzionamento un programma tenta di eseguire un'azione che il sistema impedisce, questo programma viene fermato e scaricato in un file '`core`' che può essere analizzato successivamente con GDB. A titolo di esempio viene mostrato un sorgente che produce un errore del genere:

```
int main (void)
{
    int a;
    a = 1 / 0;
    return a;
}
```

Se si compila il programma con l'accortezza di aggiungere l'opzione '**-Wall**' si viene avvisati del problema, ma in questo caso si preferisce ignorarlo:

```
$ gcc -Wall -gstabs errore.c[Invio]

  errore.c: In function ‘main’:
  errore.c:4: warning: division by zero
```

Prima di proseguire, ci si assicura che lo scarico dell'immagine del processo elaborativo sia abilitata:[4]

```
$ ulimit -c unlimited[Invio]
```

Si avvia il programma difettoso:

```
$ ./a.out[Invio]

  /bin/sh: line 1: 12134 Floating point exception↩
 ↪(core dumped) ./a.out
```

Il messaggio della shell avvisa di avere «scaricato la memoria», ovvero di avere creato il file '`core`'. Con GDB si può procedere alla ricerca di cosa è stato a causare l'errore:

```
$ gdb a.out core[Invio]

  …
  Core was generated by `./a.out’.
  Program terminated with signal 8, Arithmetic exception.
  #0  0x08048344 in main () at errore.c:4
  4           a = 1 / 0;
```

65.7.8 Problemi con l'ottimizzazione

«

Il compilatore '**gcc**' consente di utilizzare diverse opzioni per ottenere un risultato più o meno ottimizzato. L'ottimizzazione richiede una potenza elaborativa maggiore, al crescere del livello di ottimizzazione richiesto. In situazioni particolari, può succedere che la compilazione non vada a buon fine a causa di questo problema, interrompendosi con segnalazioni più o meno oscure, riferite alla scarsità di risorse. In particolare potrebbe essere rilevato un uso eccessivo della memoria virtuale, per arrivare fino allo scarico della memoria (*core dump*).

È evidente che in queste situazioni diventa necessario diminuire il livello di ottimizzazione richiesto, modificando opportunamente le opzioni relative. L'opzione in questione è '**-O*n***', come descritto nella tabella 65.102. In generale, l'assenza di tale opzione implica la compilazione normale senza ottimizzazione, mentre l'uso dell'opzione '**-O0**' può essere utile alla fine della serie di opzioni, per garantire l'azzeramento delle richieste di ottimizzazione precedenti.

Tabella 65.102. Opzioni di ottimizzazione per '**gcc**'.

Opzione	Descrizione
`-O` `-O1`	Ottimizzazione minima.
`-O2`	Ottimizzazione media.
`-O3`	Ottimizzazione massima.
`-O0`	Annullamento delle richieste precedenti di ottimizzazione.

Alle volte, compilando un programma, può succedere che a causa del livello eccessivo di ottimizzazione prestabilito, non si riesca a produrre alcun risultato. In questi casi, può essere utile ritoccare lo script di Make, dopo l'uso del comando '**`configure`**'; per la precisione si deve ricercare un'opzione che inizia per '**`-O`**'. Purtroppo, il problema sta nel fatto che spesso si tratta di più di uno script, in base all'articolazione dei file che compongono il sorgente.

Ammesso che si tratti dei file '`Makefile`', si potrebbe usare il comando seguente per attuare la ricerca:

```
$ find . -name Makefile ↩
↪        -exec echo \{\} \; ↩
↪        -exec grep \\-O \{\} \; [Invio]
```

Il risultato potrebbe essere simile a quello che si vede qui di seguito:

```
./doc/Makefile
./backend/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./frontend/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./include/Makefile
./japi/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./lib/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./sanei/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./tools/Makefile
CFLAGS = -g -O2 -W -Wall -DSCSIBUFFERSIZE=131072
./Makefile
```

In questo caso, si può osservare che i file '`./doc/Makefile`', '`./include/Makefile`' e '`Makefile`', non contengono tale stringa.

Tabella 65.104. Riepilogo delle altre opzioni utilizzate con '**gcc**' nel corso del capitolo.

Opzione	Descrizione
`-E`	Elabora il file solo con il precompilatore.
`-S`	Genera un file in linguaggio assemblatore (prevale sull'opzione '**-c**').
`-c`	Fa sì che la compilazione salti la fase di collegamento (*link*). In condizioni normali serve a generare solo i file-oggetto. Se si usa questa opzione, ma non si specifica l'opzione '**-o**', il file-oggetto ha un nome con la stessa radice del file sorgente e l'estensione '`.o`'.
`-o` ***nome_file***	Dichiara il nome del file che si vuole ottenere.
`-static` `-dynamic`	Richiede espressamente di compilare utilizzando le librerie statiche o dinamiche.
`-l`***libreria***	Indica il nome di una libreria da utilizzare. Il nome del file che la contiene può essere '`lib`***libreria*****`.a`**' o '`lib`***libreria*****`.so`**', a seconda che si tratti di una libreria statica o dinamica.
`-L`***percorso***	Indica un percorso in cui ricercare i file delle librerie, che prende la precedenza sugli altri già considerati.
`-I`***percorso***	Indica un percorso in cui ricercare i file da includere, che prende la precedenza sugli altri già considerati.

Opzione	Descrizione
`-Wall`	Richiede di mostrare tutti i messaggi che avvertono dell'uso imperfetto del linguaggio (*warning*).
`-Werror`	Fa sì che tutte le segnalazioni di avvertimento siano trattate come errori e portino al fallimento della compilazione.
`-gstabs`	Inserisce delle annotazioni, con le quali i programmi come GDB possono abbinare il sorgente originale all'esecuzione controllata del programma.

65.8 Compilazione guidata con Make

La compilazione di un programma, in qualunque linguaggio sia scritto, può essere un'operazione molto laboriosa, soprattutto se si tratta di aggregare un sorgente suddiviso in più parti, o peggio, se si tratta di un progetto costituito da più programmi. Per semplificare la procedura si potrebbe predisporre uno script che esegue sequenzialmente tutte le operazioni necessarie, ma la tradizione richiede di utilizzare il programma Make.

Uno dei vantaggi più appariscenti nell'uso di Make sta nella possibilità di evitare che vengano rielaborati i file che non sono stati modificati, abbreviando quindi il tempo di compilazione necessario quando si procede a una serie di modifiche limitate.

Make viene usato normalmente assieme a uno script, denominato comunemente '`Makefile`',[5] scritto in un modo che dovrebbe risultare molto semplice da interpretare; tuttavia, è comunque possibile fare il contrario, specialmente con le versioni più evolute di tale pro-

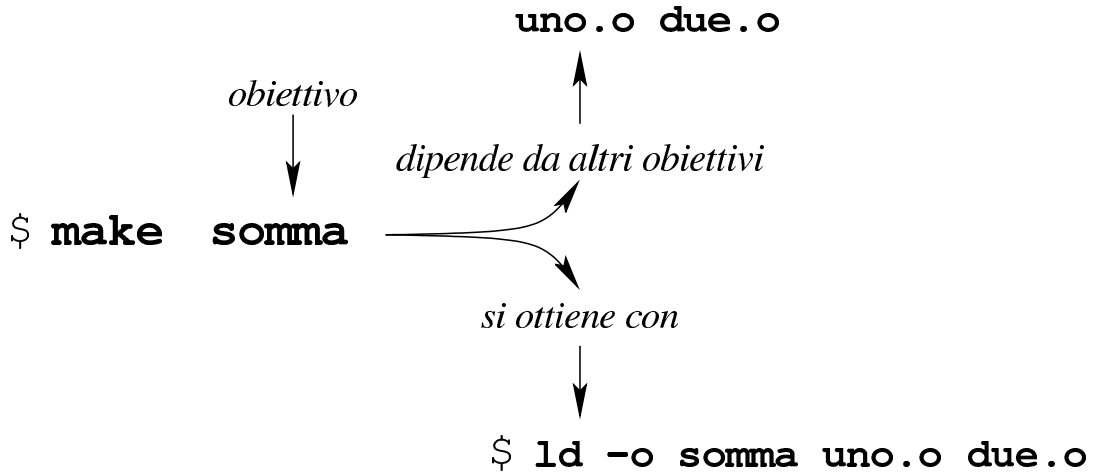
gramma. Evidentemente, Make è utile quando lo si utilizza con moderazione, ovvero con uno script semplice e lineare, altrimenti uno script di shell è sicuramente più appropriato al caso.

Gli esempi che qui mostrano script di Make non contengono commenti, pertanto è bene chiarire subito che le righe bianche o vuote vengono ignorate, così come si ignora il testo che appare alla destra del simbolo '**#**'.

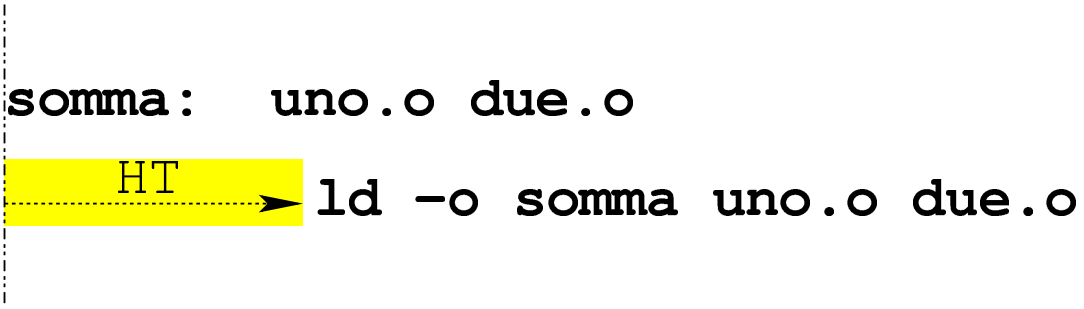
65.8.1 Obiettivo, dipendenze e comandi

Make viene usato per realizzare un ***obiettivo*** attraverso uno o più comandi da impartire alla shell (precisamente '`/bin/sh`'), dopo che sono stati soddisfatti altri eventuali obiettivi da cui quello finale dipende. In linea di principio, **l'obiettivo è rappresentato dal nome di un file che deve essere generato**.

Per esempio, volendo produrre il programma '**somma**' che si ottiene dalla compilazione dei file '`uno.c`' e '`due.c`', l'obiettivo «somma» che si ottiene con il comando '**ld -o somma uno.o due.o**', dipende dagli obiettivi «uno.o» e «due.o», in quanto i file '`uno.o`' e '`due.o`' devono essere presenti per poter eseguire il collegamento con il programma '**ld**'.



Nello script di Make, l'obiettivo di esempio mostrato si descrive come si vede nella figura successiva, dove il tratteggio verticale a sinistra rappresenta l'inizio della prima colonna. Ciò che descrive un obiettivo è, nel suo complesso, una ***regola***:



Si deve tenere a mente che la riga che definisce l'obiettivo e le dipendenze deve iniziare dalla prima colonna, mentre le righe contenenti dei comandi devono trovarsi rientrate con un carattere di tabulazione orizzontale (<*HT*>); al contrario, degli spazi veri e propri come rientro non sono ammissibili.

L'esempio introdotto è incompleto, perché non esplicita in che modo ottenere gli obiettivi '**`uno.o`**' e '**`due.o`**'. Ecco come potrebbe essere composto lo script completo delle regole che descrivono tutte le dipendenze:

```
somma: uno.o due.o
        ld -o somma uno.o due.o

uno.o: uno.c mate.h
        cc -c -o uno.o uno.c

due.o: due.c
        cc -c -o due.o due.c
```

Per comprendere l'esempio va chiarito che per ottenere il file '`uno.o`' è necessario il file '`uno.c`' che a sua volta include il file '`mate.`

h’.

Il vantaggio di usare Make sta nel fatto che questo tiene conto della data di modifica dei file, nel momento in cui valuta le dipendenze. Nel caso dell’esempio, per eseguire il collegamento (*link*) dei file oggetto nel file eseguibile ‘**somma**’, è necessario disporre di tali file oggetto, ma se il file eseguibile esiste e se questi file oggetto esistono e hanno una data di modifica antecedente a quella del file eseguibile, allora sarebbe da intendere che tale operazione non sia necessaria. Tuttavia, i file ‘uno.o’ e ‘due.o’ sono indicati come obiettivi da ottenere attraverso altri file: nel caso di ‘uno.o’ è stabilito che dipende dai file ‘uno.c’ e ‘mate.h’; nel caso di ‘due.o’ è stabilito che dipende solo dal file ‘due.c’ (si osservi che per i file ‘uno.c’, ‘mate.h’ e ‘due.c’ non sono state dichiarate altre dipendenze). A questo punto è logico attendersi che anche la data dei file di partenza conti. In pratica, le date di modifica di ‘uno.c’ e ‘mate.h’ devono essere antecedenti a qualla di ‘uno.o’ e così deve essere antecedente anche quella di ‘due.c’ rispetto a quella di ‘due.o’. Se a un certo punto si modifica il file ‘mate.h’ (e quindi la data di modifica viene aggiornata dal sistema operativo), la dipendenza che riguarda il file ‘uno.o’ richiede la ripetizione dei comandi relativi; quindi viene ricompilato il file ‘uno.o’ e viene eseguito nuovamente il collegamento che genera il file eseguibile ‘**somma**’.

Figura 65.108. Modello sintattico di una regola, per la definizione di un obiettivo in uno script di Make.

obiettivo … : [***dipendenza*** …]
<HT>***comando***
…

Per ottenere lo stesso risultato pratico dell'esempio mostrato, si può modificare il modo in cui si indica la dipendenza dovuta al file 'mate.h':

```
…
uno.o: uno.c
        cc -c -o uno.o uno.c

uno.c: mate.h
        touch uno.c mate.h
…
```

Ciò che appare nel pezzo mostrato indica che il file 'uno.o' dipende da 'uno.c' soltanto, ma il file 'uno.c' dipende dal file 'mate.h'. Se il file 'mate.h' si trova ad avere una data più recente di 'uno.c', le date vengono rese uguali e viene rifatta la compilazione.

65.8.2 Obiettivi fittizi

In generale, un obiettivo di Make viene raggiunto con la creazione o l'aggiornamento di un file che ha lo stesso nome dell'obiettivo, attraverso dei comandi stabiliti. In pratica, l'obiettivo è quel file da generare o aggiornare. Tuttavia, spesso si definiscono obiettivi che non implicano la creazione di un file con tale nome; pertanto servono per essere eseguiti sempre, assicurando che le dipendenze eventuali siano rispettate.

```
all: somma moltiplicazione

somma: …
        …
moltiplicazione: …
        …
…
```

L'esempio mostra una situazione tipica in cui si utilizza un obiettivo fittizio, in questo caso denominato '**all**'. Questo obiettivo ha il solo scopo di richiamare automaticamente gli obiettivi '**somma**' e '**moltiplicazione**' (ma nell'esempio, questi ulteriori obiettivi non vengono descritti). C'è da osservare però una cosa importante: **se per qualunque ragione dovesse esistere un file con lo stesso nome dell'obiettivo, avente una data di modifica successiva a quella dei file degli obiettivi da cui dipende, l'operazione non verrebbe eseguita**, salve naturalmente altre ipotesi riferite alle dipendenze degli obiettivi precedenti.[6]

Per ovviare all'inconveniente dovuto alla possibilità che esista un file con lo stesso nome di un obiettivo fittizio, non correlato a tale file, si può usare uno strattagemma consolidato:

```
clean: FORCE
        rm *.o core

FORCE:
```

In questo caso, l'obiettivo '**FORCE**' (usato comunemente per questo scopo), non ha dipendenze, non ha comandi, inoltre si dà per certo che non possa esistere un file con lo stesso nome; pertanto l'obiettivo risulta sempre da raggiungere. L'obiettivo '**clean**' che ha evidentemente lo scopo di eliminare alcuni file non più necessari, dipendendo dall'obiettivo '**FORCE**', viene eseguito in ogni caso, anche se esistesse un file '`clean`', perché la dipendenza non è mai soddisfatta.[7]

65.8.3 Scelta dell'obiettivo

Make è costituito generalmente dal programma eseguibile '**make**' e si usa solitamente secondo la sintassi seguente:

```
make [opzioni] [obiettivi]
```

Per esempio, il comando seguente richiede a Make di «raggiungere» l'obiettivo '**somma**':

```
$ make somma [Invio]
```

Se però non si specifica l'obiettivo, questo viene determinato in modo predefinito:

```
$ make [Invio]
```

Ammesso che nella directory corrente sia presente lo script di Make (per convenzione deve trattarsi del file '`Makefile`'), l'obiettivo viene cercato al suo interno e se non è stato definito si intende il primo che appare nel file.[8]

È comunque possibile utilizzare Make anche senza script, ma in tal caso l'indicazione dell'obiettivo nella riga di comando è obbligatoria. L'utilizzo di Make senza uno script dipende da quelle che sono definite ***regole implicite***. In pratica, quando si richiede un obiettivo non previsto espressamente, Make cerca di fare la cosa più logica, partendo dal presupposto che il contesto sia relativo alla compilazione di un programma. Si osservi l'esempio seguente:

```
$ make prova [Invio]
```

Se non è stato definito l'obiettivo '**prova**', Make considera il contenuto della directory corrente e cerca qualcosa che sia ragionevol-

mente trasformabile nel file ‘`prova`’. Per esempio, se trova il file ‘`prova.c`’ esegue automaticamente il comando ‘**`cc -o prova prova.c`**’. Questa proprietà di Make consente di omettere la descrizione delle regole degli obiettivi «ovvi». Questo sistema di regole implicite serve anche per semplificare il lavoro di stesura di uno script di Make, quando si descrive un obiettivo finale e non si stabiliscono le regole per ottenere le dipendenze:

```
somma: uno.o due.o
        ld -o somma uno.o due.o

uno.c: mate.h
        touch mate.h uno.c
```

Questo esempio richiama quanto già mostrato in precedenza: dato che la costruzione dei file ‘`uno.o`’ e ‘`due.o`’ richiede dipendenze prevedibili, non è necessario descriverne le regole.

65.8.4 Interpretazione dei comandi che portano a un obiettivo

«

In condizioni normali, i comandi che devono essere eseguiti per il raggiungimento di un certo obiettivo, vengono passati alla shell ‘`/bin/sh`’, indipendentemente dalla shell utilizzata dall’utente che avvia il programma ‘**`make`**’.

I comandi troppo lunghi possono essere spezzati e ripresi nella riga successiva, se alla fine della riga interrotta appare il simbolo ‘`\`’, esattamente come sarebbe in uno script per una shell Bourne. C’è però da osservare che, in questo caso, il comando passato alla shell comprende letteralmente sia ‘`\`’, sia il codice di interruzione di riga successivo, ma questo fatto, di norma, non ha conseguenze nel

risultato.

Sul problema dell'interruzione e proseguimento delle righe dei comandi occorre soffermarsi su un fatto: nella riga che viene ripresa, il carattere di tabulazione iniziale viene omesso automaticamente, nel momento in cui viene chiesto alla shell di eseguire il comando. L'esempio seguente rappresenta il contenuto di uno script che dovrebbe chiarire il meccanismo. Per ora si sorvoli sulla presenza della chiocciola all'inizio dei comandi:

```
esempio:
        @echo "supercalifragilisti\
        chespiralidoso"
        @echo "supercalifragilisti \
        chespiralidoso"
```

Ecco cosa succede:

```
$ make esempio[Invio]

  supercalifragilistichespiralidoso
  supercalifragilisti chespiralidoso
```

Si può osservare che nel primo caso la parola è rimasta unita, mentre nel secondo è separata perché uno spazio è stato inserito prima della segnalazione dell'interruzione.

I comandi di una regola sono eseguiti uno alla volta, ma Make tiene conto del risultato. Se il comando eseguito restituisce zero, ovvero se risulta eseguito correttamente, allora Make avvia il successivo, altrimenti interrompe l'operazione segnalando il fallimento dell'obiettivo e di quelli che da lui dipendono.[9] Pertanto, se i comandi possono restituire un errore anche se ciò non pregiudica il raggiungimento dell'obiettivo previsto, occorre provvedere in qualche modo.

Per esempio così:

```
obiettivo: …
        …
        mkdir ciao ; true
        …
```

In questo caso, la regola che descrive l'obiettivo contiene un comando che serve a garantire la presenza di una certa directory. Il comando in questione potrebbe fallire se la directory esiste già, senza per questo pregiudicare il resto del procedimento, così si unisce al comando '**`true`**' che complessivamente fa sì che l'esito sia sempre «corretto». È comunque possibile usare un prefisso che informa Make di ignorare gli errori; si tratta del segno '**`-`**', pertanto l'esempio appena apparso può essere modificato così:

```
obiettivo: …
        …
        -mkdir ciao
        …
```

In generale, prima di avviare ogni comando, Make lo visualizza, in modo da far capire ciò che accade all'utente. In alcune situazioni, però, ciò può essere spiacevole, pertanto è possibile utilizzare il prefisso '**`@`**' che evita tale comportamento:

```
mio: …
        @echo "sto per eseguire la compilazione, bla bla..."
        cc -o mio mio.c
```

Come si vede nell'esempio, si vuole fare in modo che il comando '**`echo`**' non sia «descritto», dato che già serve a mostrare qualcosa.

Tabella 65.118. Alcuni prefissi da usare nelle righe che contengono comandi.

Prefisso	Significato
`-`	fa in modo che gli errori vengano ignorati;
`+`	fa in modo che il comando venga eseguito sempre;
`@`	fa in modo che il testo del comando non venga mostrato.

65.8.5 Variabili o «macro»

All'interno di uno script di Make è possibile definire delle variabili, altrimenti note come «macro». Le variabili si dichiarano attraverso direttive espresse nella forma seguente:

```
nome = stringa
```

In particolare, la stringa non deve essere delimitata e l'ordine della dichiarazione delle variabili non viene tenuto in considerazione, come dimostrato poco più avanti. L'espansione di una variabile si indica attraverso due modi possibili:

```
$(nome)
```

Oppure:

```
${nome}
```

Si osservi l'esempio seguente, in particolare a proposito del fatto che l'ordine di dichiarazione delle variabili non è significativo:

```
bindir = $(exec_prefix)/bin
prefix = /usr/local
sbindir = $(exec_prefix)/sbin
exec_prefix = $(prefix)

all:
        @echo "prefix = $(prefix)"
        @echo "exec_prefix = $(exec_prefix)"
        @echo "bindir = $(bindir)"
        @echo "sbindir = $(sbindir)"
```

Ammesso che questo sia lo script di Make contenuto nella directory corrente:

```
$ make [Invio]

  prefix = /usr/local
  exec_prefix = /usr/local
  bindir = /usr/local/bin
  sbindir = /usr/local/sbin
```

Il fatto che l'ordine nella dichiarazione nelle variabili non conti, implica che l'assegnamento a una variabile del proprio stesso contenuto produca un circolo vizioso. In pratica, una cosa come la dichiarazione seguente **non è ammissibile**:

```
opzioni = -c
opzioni = -gstabs $(opzioni)
```

Invece di agire così, per aggiungere qualcosa a una variabile occorre una direttiva differente:

nome `+=` ***stringa***

Si osservi l'esempio seguente e ciò che succede provando a usare '**`make`**':

```
opzioni = -c
opzioni += -gstabs

all:
        @echo "opzioni = $(opzioni)"
```

`$` **`make`** [*Invio*]

```
  opzioni = -c -gstabs
```

Come si può intendere, le variabili di Make che appaiono all'interno dei comandi, vengono espanse prima dell'esecuzione dei comandi stessi; di conseguenza, se si vuole usare il dollaro ('**`$`**') in modo che la shell lo recepisca, occorre raddoppiarlo:

```
obiettivo: …
        …
        NUM=3 ; echo $$NUM
        …
```

GNU Make recepisce le variabili di ambiente e le assimila tra le proprie variabili, ma se nel proprio script vengono ridefinite, ciò prevale sul valore ottenuto dall'esterno.

Make prevede delle variabili predefinite, il cui scopo principale è controllare il funzionamento delle regole implicite, ma che spesso

vengono usate per coerenza anche nei comandi di obiettivi dichiarati esplicitamente. La tabella 65.127 ne elenca alcune e l'esempio successivo, riprendendone un altro già apparso, mostra in che modo potrebbero essere usate:

```
somma: uno.o due.o
        $(LD) $(LDFLAGS) -o somma uno.o due.o

uno.o: uno.c mate.h
        $(CC) -c $(CFLAGS) -o uno.o uno.c

due.o: due.c
        $(CC) -c $(CFLAGS) -o due.o due.c
```

Trattandosi di variabili conosciute, se utilizzate correttamente si facilita la lettura dello script, consentendo di precisare, se ce ne fosse bisogno, il nome del compilatore e le opzioni da dare:

```
LD = ld
CC = gcc
CFLAGS = -gstabs

somma: uno.o due.o
        $(LD) $(LDFLAGS) -o somma uno.o due.o

uno.o: uno.c mate.h
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -o uno.o uno.c

due.o: due.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) -o due.o due.c
```

Tabella 65.127. Elenco di alcune variabili predefinite di Make.

Nome	Contenuto usuale	Annotazioni
`MAKE`	`make`	Il nome del programma stesso. Di solito viene usata questa informazione per l'avvio di altri Make in sottodirectory con un proprio script.
`SHELL`	`/bin/sh`	La shell che deve eseguire i comandi: è bene evitare di cambiare il valore di questa variabile, ovvero, se dichiarata, è bene confermarlo.
`AR` `ARFLAGS`	`ar` `rw`	Il programma di archiviazione usato per creare le librerie statiche e le sue opzioni consuete.
`LD` `LDFLAGS`	`ld`	Il programma usato per collegare i file-oggetto in un file eseguibile e le sue opzioni consuete. Di norma non sono previste opzioni particolari.
`AS` `ASFLAGS`	`as`	Il programma usato per compilare un file in linguaggio assemblatore e le sue opzioni consuete. Di norma non sono previste opzioni particolari.
`CPP` `CPPFLAGS`	`cpp`	Il precompilatore e le sue opzioni consuete. Di norma non sono previste opzioni particolari.

Nome	Contenuto usuale	Annotazioni

`CC`
 	
`CFLAGS`
	`cc`	Il programma usato per compilare un file in linguaggio C e le sue opzioni consuete. Di norma non sono previste opzioni particolari.

`PC`
 	
`PFLAGS`
	`pc`	Il programma usato per compilare un file in linguaggio Pascal e le sue opzioni consuete. Di norma non sono previste opzioni particolari.

65.8.6 Utilizzo oculato delle variabili

«

Dal momento che le variabili possono essere espanse in ogni posizione di uno script di Make, le definizioni ripetitive possono essere semplificate. Nell'esempio successivo si dichiara la variabile '**`obj`**', contenente l'elenco dei file-oggetto coinvolti nella produzione di un certo file eseguibile:

```
obj = aaa.o bbb.o ccc.o ddd.o \
      eee.o fff.o ggg.o
…
prog: $(obj)
    ld -o prog $(obj)
…
```

Generalmente, i nomi delle variabili sono scritti utilizzando solo lettere maiuscole, ma non c'è un obbligo in tal senso. Di solito, l'utilizzo di lettere maiuscole per le variabili vuole indicare la possibilità di modificarne il contenuto per qualunque adattamento possa essere necessario; per questo, se invece si utilizzano nomi di variabili con

lettere minuscole (come nell'esempio mostrato), di solito lo si fa per quelle cose che è bene non modificare.

Come già accennato, GNU Make eredita le variabili di ambiente come proprie variabili-macro, anche se poi queste possono essere ridefinite nello script. In ogni caso, il modo «normale» di assegnare un valore a una variabile, nel momento dell'avvio del programma '**make**', è quello di usare la riga di comando, per esempio, così:

```
$ make "CFLAGS = -O" "LDFLAGS = -s" obiettivo [Invio]
```

Si supponga di avere uno script come quello seguente e si osservi cosa succede con il comando appena mostrato:

```
CFLAGS = -gstabs
LDFLAGS = -S

all:
        @echo "CFLAGS:  $(CFLAGS)"
        @echo "LDFLAGS: $(LDFLAGS)"
```

```
$ make "CFLAGS = -O" "LDFLAGS = -s" obiettivo [Invio]

  CFLAGS:  -O
  LDFLAGS: -s
```

Pertanto, questo modo di passare il valore alle variabili di Make prevale sulla dichiarazione interna di uno script.

65.8.7 Espansione e continuazione al di fuori dei comandi

Il testo di uno script di Make, quando non costituisce un comando da passare alla shell e non si tratta nemmeno di un commento, può essere espanso, sia a causa dell'uso di variabili, sia per la presenza di

caratteri che si espandono in nomi di file, come si fa comunemente per le shell POSIX (quindi si possono usare l'asterisco, il punto interrogativo e le parentesi quadre, con lo stesso significato che hanno per una shell POSIX e si possono anche proteggere i simboli, contro l'espansione, facendoli precedere da una barra obliqua inversa: '`\`'). Inoltre, è possibile continuare il testo su più righe, usando il simbolo '`\`' alla fine della riga che deve continuare. L'esempio che appare sotto serve a mostrare l'effetto dell'espansione, ma non è un modello da seguire, perché in pratica si creerebbero delle complicazioni:

```
prog: *.o
        $(LD) $(LDFLAGS) -o prog *.o
```

In questo caso, la realizzazione del file '`prog`' dipende da tutti i file-oggetto presenti nella directory corrente. L'esempio non è utile in generale, perché se tali file sono assenti viene meno la realizzazione dell'obiettivo. È comunque interessante osservare che l'espansione di '`*.o`' nell'elenco delle dipendenze avviene per opera di Make, mentre ciò che appare nel comando viene espanso dalla shell.

65.8.8 Variabili automatiche

«

Alcune variabili non possono essere dichiarate e nemmeno modificate nel loro contenuto. Si tratta delle ***variabili automatiche***, composte da un solo carattere. Per esempio, la variabile '`@`' rappresenta l'obiettivo attuale, ma per espandere il suo contenuto è sufficiente scrivere '`$@`', senza bisogno di parentesi.

Per espandere una variabile si possono sempre evitare le parentesi se il nome di questa è composto da un solo carattere; tuttavia, si preferisce rinunciare alle parentesi solo quando si tratta precisamente di variabili automatiche.

Tabella 65.132. Alcune variabili automatiche.

Variabile automatica	Significato
`*`	Il nome dell'obiettivo attuale, ma senza suffisso; se l'obiettivo non ha suffisso, la variabile risulta vuota.
`@`	L'obiettivo attuale, completo.
`<`	La voce che costituisce la prima dipendenza (quella più a sinistra).
`?`	L'elenco delle dipendenze associate a file che sono più recenti di quello che rappresenta l'obiettivo.
`^`	L'elenco di tutte le dipendenze previste.
`$`	Si espande semplicemente nel simbolo '**`$`**' e, come per tutte le variabili automatiche, va usato aggiungendo un altro dollaro: '**`$$`**'. In pratica, nei comandi, le variabili di ambiente vanno annotate raddoppiando il simbolo dollaro; per esempio: '**`$$HOME`**'.

Per comprendere meglio il significato della descrizione fatta nella tabella precedente, si consideri di disporre dei file seguenti: '`uno.c`', '`due.c`', '`somma.o`'. Inoltre, si suppone che solo '`due.c`' abbia una data di modifica successiva a quella di '`somma.o`'. A tale proposito, si consideri lo script seguente:

```
somma.o: uno.c due.c
        @echo \$$\* = $*
        @echo \$$@ = $@
        @echo \$$\< = $<
        @echo \$$\? = $?
        @echo \$$\^ = $^
```

Se viene avviato, si può leggere lo stato delle variabili automatiche:

```
$ make [Invio]

  $* = somma
  $@ = somma.o
  $< = uno.c
  $? = due.c
  $^ = uno.c due.c
```

Come si può vedere, in questo caso la variabile automatica '**?**' consentirebbe di individuare le dipendenze per le quali si richiede una nuova compilazione.

È importante notare che le variabili automatiche possono essere usate solo all'interno di comandi, perché il loro contenuto si definisce dopo la dichiarazione dell'obiettivo e delle sue dipendenze.

65.8.9 Regole implicite

«

Le regole implicite sono quelle che descrivono degli obiettivi predefiniti, nel modo più logico possibile. Come già accennato altrove, queste regole definiscono i comandi attraverso delle variabili che è possibile controllare.

Tabella 65.135. Comandi comuni di regole implicite.

Comando	Condizione di utilizzo
`$(CC) -c $(CPPFLAGS) $(CFLAGS)` ↩ ↪***nome*`.c`	Se si richiede la realizzazione dell'obiettivo denominato ***nome.o*** ed esiste il file '***nome*`.c`'.
`$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)` ↩ ↪***nome*`.cc` `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)` ↩ ↪***nome*`.cpp`	Se si richiede la realizzazione dell'obiettivo denominato ***nome.o*** ed esiste il file '***nome*`.cc`' o '***nome*`.cpp`'.
`$(AS) $(ASFLAGS)` ***nome*`.s`	Se si richiede la realizzazione dell'obiettivo denominato ***nome.o*** ed esiste il file '***nome*`.s`'. Se il file '***nome*`.s`' è assente ma al suo posto esiste '***nome*`.S`', allora il primo viene generato dal comando successivo.
`$(CPP) $(CPPFLAGS)` ***nome*`.S`	Se si richiede la realizzazione dell'obiettivo denominato ***nome.s*** ed esiste il file '***nome*`.S`'.
`$(CC) $(LDFLAGS)` ***nome*`.o` ↩ ↪`$(LOADLIBES) $(LDLIBS)`	Se si richiede la realizzazione dell'obiettivo denominato ***nome*** ed esiste il file '***nome*`.o`'.

Naturalmente, le regole implicite si concatenano tra di loro. Per esempio, si suppone di disporre del file '`prova.c`' e di volerlo compilare utilizzando Make nel modo seguente:

```
$ make prova [Invio]
```

Si sta facendo riferimento all'obiettivo '**prova**' che si intende non sia stato dichiarato nello script di Make. Pertanto, per realizzare questo obiettivo, Make deve cercare una regola implicita appropriata e in questo caso è quella che serve a collegare un file oggetto 'prova.o':

```
prova: prova.o
        $(CC) $(LDFLAGS) prova.o $(LOADLIBES) $(LDLIBS)
```

Questa regola implicita, evidentemente, dipende da un'altra regola che descrive in che modo viene ottenuto il file 'prova.o'. Dal momento che Make trova il file 'prova.c', la regola è questa:

```
prova.o: prova.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) prova.c
```

Di conseguenza viene eseguita la compilazione.

65.8.10 Uno script per ogni sottodirectory

«

Di solito si predispone uno script di Make per ogni sottodirectory che contenga qualcosa da costruire; poi, in una o in alcune directory si colloca uno script realizzato in modo da avviare lo stesso programma '**make**' nelle sottodirectory inferiori.

A titolo di esempio, si suppone di avere un progetto suddiviso in tre sottodirectory: 'mele/', 'arance/' e 'limoni/'. All'interno di ogni sottodirectory c'è un file 'Makefile'. Nella directory che contiene queste sottodirectory c'è un file 'Makefile' con una regola per avviare sequenzialmente gli altri file equivalenti delle sottodirectory:

```
sub = mele arance limoni

all:
        for d in $(sub) ; do cd $$d ; $(MAKE) ; cd .. ; done
```

Viene usato un ciclo per la scansione delle sottodirectory che la shell interpreta così:

```
for d in mele arance limoni
do
    cd $d
    make
    cd ..
done
```

Ogni volta che si usa Make in questo modo, si dovrebbe vedere un avvertimento come quello seguente:

```
make[1]: Entering directory ‘/home/tizio/mele’
…
make[1]: Leaving directory ‘/home/tizio/mele’
make[1]: Entering directory ‘/home/tizio/arance’
…
make[1]: Leaving directory ‘/home/tizio/arance’
make[1]: Entering directory ‘/home/tizio/limoni’
…
make[1]: Leaving directory ‘/home/tizio/limoni’
```

Per lasciare a Make il controllo del ciclo di avvii nelle sottodirectory, si può usare un meccanismo differente, come quello che si vede nel listato successivo:

```
sub = mele arance limoni

all: $(sub)

$(sub): FORCE
        cd $@ && $(MAKE)

FORCE:
```

Si può vedere che l'obiettivo '**all**' (evidentemente un obiettivo fittizio), dipende dai nomi delle sottodirectory. Successivamente è dichiarata una regola con obiettivo multiplo, ovvero una regola che vale indifferentemente per i tre obiettivi di ogni sottodirectory (la variabile automatica '**$@**' si espande nel nome dell'obiettivo preso in considerazione effettivamente). Tale regola dipende però da un altro obiettivo, senza dipendenze e senza comandi, per il quale si è certi che non possa esistere un file con lo stesso nome.

Come mostrato in questi esempi, invece di scrivere il nome del programma eseguibile '**make**', è stata usata la variabile '**MAKE**', la quale riproduce il nome del comando usato per avviare l'interpretazione dello script. Per esempio, se per qualunque motivo il programma '**make**' fosse nominato in maniera differente o fosse usato al di fuori dei percorsi di ricerca per gli eseguibili, con la variabile '**MAKE**' si garantisce sempre di trovare lo stesso programma che risulta già in funzione.

65.8.11 Una regola per più obiettivi

«

La sezione precedente introduce il concetto di obiettivo multiplo, che però può risultare difficile da intendere con l'esempio mostrato, dal momento che si utilizza una variabile per esprimere l'elenco di

obiettivi. Lo stesso esempio può essere tradotto così, senza l'uso di variabili:

```
all: mele arance limoni

mele arance limoni: FORCE
        cd $@ && $(MAKE)

FORCE:
```

Senza usare una regola del genere, occorrerebbe suddividere la stessa in tre (una per ogni singolo obiettivo):

```
…
mele: FORCE
        cd mele && $(MAKE)

arance: FORCE
        cd arance && $(MAKE)

limoni: FORCE
        cd limoni && $(MAKE)
…
```

Questo dovrebbe chiarire anche l'utilità della variabile automatica '**$@**', per individuare l'obiettivo preso effettivamente in considerazione in un dato momento.

65.8.12 Regole fittizie tipiche

Generalmente si organizza uno script di Make in modo da avere alcuni obiettivi fittizi, con cui eseguire le operazioni più comuni in modo complessivo. Tra questi, l'obiettivo più importante in assoluto è quello che deve essere eseguito in modo predefinito (ovvero il

primo) e generalmente viene chiamato '**all**'. Tale obiettivo serve di norma per indicare soltanto delle dipendenze da soddisfare.

Tabella 65.144. Obiettivi fittizi comuni.

Obiettivo	Significato comune
`all`	Le azioni da compiere quando non si indica alcun obiettivo in modo esplicito.
`clean`	I comandi da eseguire per cancellare i file oggetto, i binari già compilati ed eventualmente altri file temporanei.
`install`	I comandi necessari a installare i programmi dopo la compilazione.

Stando alla tabella appena mostrata, si può ricordare che le fasi tipiche di un'installazione di un programma distribuito in forma sorgente sono appunto quelle seguenti:

1. `#` **`make`**[*Invio*]

   Richiama automaticamente l'obiettivo '**all**', coincidente con i comandi necessari per la compilazione del programma.

2. `#` **`make install`**[*Invio*]

   Provvede a installare gli eseguibili compilati nella loro destinazione prevista.

Supponendo di avere realizzato un programma, denominato '`mio_prog.c`', il cui eseguibile debba essere installato nella directory '`/usr/local/bin/`', si potrebbe utilizzare uno script composto come l'esempio seguente, dove l'obiettivo '**all**' richiama la dipendenza dal programma '**mio_prog**' che viene soddisfatta in modo implicito:

```
all: mio_prog

clean:
        rm -f core *.o mio_prog

install:
        cp mio_prog /usr/local/bin
```

65.8.13 Variabili per l'installazione

In uno script di Make realizzato per la compilazione di un programma (composto eventualmente da uno o più file eseguibili) e per la sua installazione successiva, sono presenti normalmente alcune variabili, più o meno standardizzate, per descrivere la collocazione finale dei file. Alcune di queste variabili sono elencate nella tabella successiva.

Tabella 65.146. Alcune variabili usate comunemente per definire l'installazione.

Variabile	Utilizzo
`prefix`	La variabile '**`prefix`**' viene usata normalmente come punto di partenza da cui traggono origine altre variabili più specifiche. Di solito, il valore dato a questa variabile per la distribuzione di un pacchetto sorgente è '`/usr/local/`', dove poi chi compila e installa deve attribuire un valore che per sé possa essere più appropriato.
`exec_prefix`	La variabile '**`exec_prefix`**' rappresenta il punto di riferimento iniziale per l'installazione dei file eseguibili e di solito corrisponde al contenuto di '**`prefix`**'.

Variabile	Utilizzo
`bindir`	Rappresenta la directory in cui vanno installati i file eseguibili a disposizione di tutti gli utenti e corrisponde normalmente alla directory '`bin/`' successiva al contenuto di '**`exec_prefix`**'. Pertanto, se '**`prefix`**' e '**`exec_prefix`**' indicano '`/usr/local/`', '**`bindir`**' indica normalmente '`/usr/local/bin/`'.
`sbindir`	Rappresenta la directory in cui vanno installati i file eseguibili utili per l'amministrazione e corrisponde normalmente alla directory '`sbin/`' successiva al contenuto di '**`exec_prefix`**'.

Uno script che usa queste variabili potrebbe essere realizzato così:

```
prefix = /usr/local
exec_prefix = $(prefix)
bindir=$(exec_prefix)/bin

all: mio_prog

clean:
        rm -f core *.o mio_prog

install:
        cp mio_prog $(bindir)
```

65.8.14 Definizione della shell

«

In linea di principio, la shell usata per eseguire i comandi contenuti nelle regole di uno script di Make è '`/bin/sh`'. Tuttavia, il nome e il percorso esatto possono essere controllati attraverso la variabile '**`SHELL`**'. In generale può essere conveniente aggiungere nello script la dichiarazione seguente, in modo da non avere sorprese:

```
SHELL = /bin/sh
```

Per esempio, utilizzando GNU Make occorre considerare che le variabili di ambiente sono ereditate e la presenza eventuale di una variabile '**SHELL**' potrebbe creare problemi: è per questo che la dichiarazione suggerita può essere conveniente.

Evidentemente, nello stesso modo descritto è possibile cambiare la shell che interpreta i comandi, ma ciò è sicuramente sconsigliabile, nell'ottica della creazione di script «standard».

65.8.15 Installazione dei programmi

È il caso di osservare che, normalmente, l'installazione dei programmi, ovvero la loro copia nella destinazione finale, dopo la compilazione, si esegue con il programma '**install**' (eventualmente si veda la sezione 20.11.7). Il motivo di questa scelta sta normalmente nella facilità con cui, assieme alla copia, si definiscono i permessi e la proprietà del file nella destinazione. Lo script seguente riprende gli ultimi esempi e concetti già visti:

```
SHELL = /bin/sh
prefix = /usr/local
exec_prefix = $(prefix)
bindir=$(exec_prefix)/bin

all: mio_prog

clean:
        rm -f core *.o mio_prog

install:
```

```
install -o root -g bin -m 755 mio_prog $(bindir)
```

65.9 Riferimenti

«

- *YoLinux Tutorial - Static, Shared Dynamic and Loadable Linux Libraries*, http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
- Steve Chamberlain, Ian Lance Taylor, *Using LD, the GNU linker*, http://www.zemris.fer.hr/~leonardo/oszur/tehnicki.dokumenti/gnu-linker.pdf
- *a.out (file format)*, http://en.wikipedia.org/wiki/A.out_(file_format)
- *COFF*, http://en.wikipedia.org/wiki/COFF
- *Portable Executable*, http://en.wikipedia.org/wiki/Portable_Executable
- *DWARF*, http://en.wikipedia.org/wiki/DWARF
- *Mach-O*, http://en.wikipedia.org/wiki/Mach-O
- *Executable and linkable format* http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- *format of executable and linking format (ELF) files*, pagina di manuale *elf(5)*
- Brian Raiter, *A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux*, http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html

- *Multiboot specification*, http://www.gnu.org/software/grub/manual/multiboot/
- osdev.org wiki, *Tutorial: bare bones*, http://wiki.osdev.org/Tutorial:Bare_bones
- Gergor Brunmar: *The booting process*, *The world of Protected mode*, *Mixing Assembly and C-code*, http://www.osdever.net/tutorials/pdf/gb_booting.pdf , http://www.osdever.net/tutorials/pdf/gb_pmode.pdf , http://www.osdever.net/tutorials/pdf/gb_asm_and_c.pdf
- mr. xsism, *Xosdev*, http://www.osdever.net/tutorials/pdf/ch01.pdf , http://www.osdever.net/tutorials/pdf/ch02.pdf
- Tim Robinson, *Writing a Kernel in C*, http://www.osdever.net/tutorials/pdf/ckernel.pdf
- Joachim Nock, K.J., *Making a Simple C kernel with Basic printf and clearscreen Functions*, http://www.osdever.net/tutorials/view/writing-a-simple-c-kernel
- OSDevWiki, *Category: Tutorials*, http://wiki.osdev.org/Category:Tutorials
- OSDevWiki, *Category: Babystep*, http://wiki.osdev.org/Category:Babystep
- Daniel Rowell Faulkner, *Hello World Boot Loader*, http://www.osdever.net/tutorials/view/hello-world-boot-loader
- OSRC, *the Operating System resource center*, http://www.nondot.org/sabre/os/articles
- SigOps, *How to Write an Operating System*, http://www.acm.uiuc.edu/sigops/roll_your_own/ , http://www.acm.uiuc.edu/

sigops/roll_your_own/hardware/kb.html , *http://www.acm.uiuc.edu/sigops/roll_your_own/hardware/text.html*

- *GNU Compiler Collection*, *http://en.wikipedia.org/wiki/GNU_Compiler_Collection*
- *Using the GNU compiler collection (GCC)*, *http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/* , *http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc.pdf*
- Brian Gough, *An Introduction to GCC*, *http://www.network-theory.co.uk/docs/gccintro/*
- Richard M. Stallman, Roland McGrath and Paul D. Smith, *GNU Make: A Program for Directing Recompilation*, ISBN 1-882114-83-3, *http://www.gnu.org/software/make/manual/*

[1] **GNU Binutils** GNU GPL

[2] Sono pochi i sistemi operativi affermati, mentre esistono una miriade di progetti più o meno abbandonati; quindi: prima di pensare di scrivere il proprio sistema converrebbe dare un'occhiata a quanti lavori del genere sono stati catalogati.

[3] **GCC** GNU GPL

[4] Di norma, il comando '**`ulimit`**' è gestito internamente dalla shell; in questo esempio si fa riferimento a una shell POSIX.

[5] Lo script di Make potrebbe essere nominato anche in altri modi, per esempio senza l'iniziale maiuscola (quindi solo '`makefile`') ma in generale conviene attenersi al suggerimento di usare il nome '`Makefile`' che, in un elenco ordinato per nome dei file di una directory, ha il vantaggio di apparire prima di altri a causa dell'iniziale maiuscola.

[6] Nel caso di GNU Make esiste una direttiva apposita per dichiarare quali obiettivi sono fittizi (*phoney*).

[7] GNU Make offre un meccanismo più accurato per impedire che un obiettivo fittizio sia bloccato da un file, ma il metodo mostrato è valido in generale.

[8] Nel caso di GNU Make, nella ricerca del primo obiettivo si escludono i nomi che iniziano con un punto, dal momento che a quelli viene dato un significato particolare.

[9] Di solito il fallimento di un obiettivo può comportare la cancellazione contestuale del file corrispondente all'obiettivo mancato, anche se si tratta di una versione precedente.