

# EXECUTABLE AND LINKABLE FORMAT

## Introduzione

Il formato ELF (formato eseguibile e collegabile) fu sviluppato e pubblicato dagli UNIX System Laboratories (USL) come parte della Application Binary Interface (ABI: interfaccia tra sistema operativo e le applicazioni a livello di linguaggio macchina). Il comitato Tool Interface Standards ha scelto lo standard ELF come formato di file oggetto portabile che opera su architettura Intel a 32 bit per una varietà di sistemi operativi.

Lo standard ELF ha l'obiettivo di facilitare lo sviluppo del software fornendo agli sviluppatori una serie di definizioni di interfacce che si estendono a vari ambienti operativi. Questo dovrebbe ridurre il numero di implementazioni diverse di interfacce, riducendo di conseguenza la necessità di ricodificare e ricompilare.

## Questo documento

Questo documento è destinato agli sviluppatori che creano file oggetto o file eseguibili per vari sistemi operativi a 32 bit. E' diviso in tre parti:

Parte 1 – “File oggetto” descrive il formato ELF per i tre tipi principali di file oggetto.

Parte 2 - “Caricamento e collegamento dinamico dei programmi” descrive le informazioni del file oggetto e le azioni del sistema per la creazione del processo.

Parte 3 - “libreria C” elenca i simboli contenuti in `libsys`, nello standard ANSI C, le routine e i simboli dei dati globali richiesti dalle routine della `libc`.

Nota: i riferimenti all'architettura x86 sono stati cambiati in quelli per l'architettura Intel.

## Nota del traduttore

Alcuni termini richiedono una definizione preliminare a causa di possibili ambiguità.

**File oggetto** o codice oggetto: è la traduzione del codice sorgente in linguaggio macchina operata dal compilatore.

**File oggetto condiviso** (shared object file nel documento): sinonimo di libreria. La libreria può essere a collegamento statico oppure a collegamento dinamico; in quest'ultimo caso nel mondo Unix si parla di librerie condivise. Nel testo si userà sia file oggetto condiviso che libreria.

**Linking** o collegamento (non tradotto nel testo, poiché è di uso corrente anche in italiano): integrazione dei moduli a cui un programma fa riferimento (sottoprogrammi o librerie) per creare una singola unità eseguibile. Il linker (nel testo inglese link editor) è il programma che esegue il collegamento a compile-time. Il dynamic linker o loader esegue il collegamento a run-time. A volte i due termini vengono considerati sinonimi. Per evitare confusione tra il processo a *compile-time* e quello a *run-time*, in questa traduzione si usa *linker* per il primo e *loader* per il secondo.

**Binding** (reso nel testo con collegamento): è il processo tramite cui viene effettuato il collegamento fra una entità di un software e il suo corrispondente valore. Si distinguono un binding statico (early binding) a *compile-time* e uno dinamico (late binding) a *run-time*.

**Relocation** (rilocazione): meccanismo che mette in corrispondenza gli indirizzi “logici” o “virtuali” di un processo, con quelli “fisici” delle locazioni dove sono effettivamente disponibili le informazioni. Il loader provvede a eseguire questa sostituzione di indirizzi.

Traduzione a cura di Andrea Cassani

[www.parolamia.eu](http://www.parolamia.eu)

sono graditi commenti, osservazioni, correzioni: [and.cassani@gmail.com](mailto:and.cassani@gmail.com)

# PARTE 1

## FILE OGGETTO

### Introduzione

La parte 1 descrive il formato di file oggetto chiamato ELF (Executable and Linking Format). Ci sono tre tipi principali di file oggetto:

- file rilocabile che contiene codice e dati disponibili per il linking con altri file oggetto per creare un eseguibile o un file di libreria
- file eseguibile che contiene un programma pronto per l'esecuzione; il file specifica come la funzione `exec(BA_OS)` crea un processo.
- File oggetto condiviso idoneo ad essere collegato in due modi. Nel primo, il linker [`ld(SD_CMD)`] può processarlo con altri file rilocabili o librerie per creare un altro file oggetto. Nel secondo, il loader lo combina con un eseguibile e altre librerie per creare un processo.

I file oggetto, creati dall'assembler e dal linker, sono rappresentazioni binarie di un programma destinate ad essere eseguite direttamente dal processore. Non sono compresi in questa definizione i programmi che richiedono un interprete, come gli script di shell.

Dopo l'introduzione, la Parte 1 si concentra sul formato del file e su quel che riguarda la costruzione del programma. La Parte 2 descrive le varie parti del file oggetto, concentrandosi sulle informazioni necessarie per eseguire il programma.

### Formato del file

I file oggetto prendono parte al linking e all'esecuzione del programma. Per comodità ed efficienza, il formato del file oggetto fornisce prospettive diverse del proprio contenuto, a seconda dei bisogni di queste due attività. La figura mostra la struttura del file oggetto.

prospettiva “collegamento”

ELF header
Program header table optional
Section 1
...
Section n
...
...
Section header table

prospettiva “esecuzione”

ELF header
Program header table
Segment 1

Segment 2
...
Section header table optional

L'*header* (o *intestazione*) *ELF* è collocato all'inizio del file e contiene una “mappa” che ne descrive la struttura. Le *sezioni* contengono le informazioni per la prospettiva del linking: istruzioni, dati, tabella dei simboli, informazioni per la rilocazione, etc. La descrizione delle sezioni speciali si trova al termine della Parte 1. Nella Parte 2 vengono descritti i *segmenti* e la prospettiva dell'esecuzione del programma.

La *program header table* (tabella della intestazione del programma - da ora in poi PHT), se presente, istruisce il sistema su come creare una immagine del processo. I file impiegati per costruire tale immagine, (eseguire il programma), devono avere una PHT; i file rilocabili no. La *section header table* (tabella della intestazione di sezione - da ora in poi SHT) contiene la descrizione delle *sezioni* del file. Ogni sezione ha una voce nella SHT; ogni voce contiene informazioni come il nome della sezione, le dimensioni, etc. I file usati nella fase di linking devono avere una SHT; gli altri file oggetto possono non averla.

Nota: anche se la figura mostra la PHT subito dopo l'header ELF e la SHT dopo le varie sezioni, i file possono non rispecchiare questo schema. Inoltre le sezioni e i segmenti non hanno un ordine definito. Solo l'header ELF ha una posizione fissa nel file.

## Rappresentazione dei dati

Il formato di file oggetto supporta numerosi processori con architetture a 8 e 32 bit. E' comunque estensibile ad architetture superiori o inferiori. Il file oggetto rappresenta una parte dei dati di controllo in un formato indipendente dall'hardware, identificabile e interpretabile in modo univoco. Altri dati usano la codifica adatta per il singolo processore, indipendentemente dalla macchina su cui il file viene creato.

Tipi di dati a 32 bit

Nome	Dimensione	Allineamento	Scopo
Elf32_Addr	4	4	unsigned program address
Elf32_Half	2	2	unsigned medium integer
Elf32_Off	4	4	unsigned file offset
Elf32_Sword	4	4	signed large integer
Elf32_Word	4	4	unsigned large integer
unsigned char	1	1	unsigned small integer

Tutte le strutture dati definite nel formato di file oggetto seguono le dimensioni e l'allineamento “naturali” per le rispettive tipologie. Se necessario, le strutture prevedono riempimenti per assicurare l'allineamento a 4 byte per gli oggetti a 32 bit. I dati hanno anche allineamenti adatti a partire dall'inizio del file. Ad esempio, una struttura che contiene un membro di tipo `Elf32_Addr` sarà allineata ai 4 byte entro il file.

Per ragioni di portabilità, ELF non usa campi di bit.

## ELF header

Alcune strutture di controllo possono aumentare di dimensioni, poiché l'header ELF contiene le loro dimensioni attuali. Se il formato cambia, un programma può imbattersi in strutture di controllo più grandi o più piccole dello standard. I programmi possono ignorare le informazioni “eccedenti”. L'utilizzo delle informazioni “mancanti” dipende dal contesto e verrà specificato all'atto della definizione delle estensioni.

## ELF header

```
#define EI_NIDENT (16)

typedef struct {
    unsigned char    e_ident[EI_NIDENT];    /* Magic number and other info */
    Elf32_Half      e_type;                 /* Object file type */
    Elf32_Half      e_machine;             /* Architecture */
    Elf32_Word      e_version;             /* Object file version */
    Elf32_Addr      e_entry;               /* Entry point virtual address */
    Elf32_Off       e_phoff;               /* Program header table file */
    Elf32_Off       e_shoff;               /* Section header table file offset */
    Elf32_Word      e_flags;               /* Processor-specific flags */
    Elf32_Half      e_ehsize;              /* ELF header size in bytes */
    Elf32_Half      e_phentsize;           /* Program header table entry size */
    Elf32_Half      e_phnum;               /* Program header table entry count */
    Elf32_Half      e_shentsize;           /* Section header table entry size */
    Elf32_Half      e_shnum;               /* Section header table entry count */
    Elf32_Half      e_shstrndx;            /* Section header string table index */
} Elf32_Ehdr;
```

**e\_ident** il byte iniziale identifica il file come file oggetto e fornisce un dato indipendente dal processore con cui decodificare e interpretare il contenuto. La descrizione completa è fornita nel paragrafo “Identificazione ELF”.

**e\_type** identifica il tipo di file:

nome	valore	significato
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_NUM	5	Number of defined types
ET_LOOS	0xfe00	OS-specific range start
ET_HIOS	0xfeff	OS-specific range end
ET_LOPROC	0xff00	Processor-specific range start
ET_HIPROC	0xffff	Processor-specific range end

Anche se il contenuto del file core non è specificato, il tipo ET\_CORE è riservato ad esso. I valori da ET\_LOPROC a ET\_HIPROC inclusi sono riservati a dati specifici per processore. Altri valori sono riservati e verranno assegnati in base alle necessità di nuovi tipi di file.

**e\_machine** specifica l'architettura

nome	valore	significato
EM_NONE	0	No machine
EM_M32	1	AT&T WE 32100

EM_SPARC	2	SUN SPARC
EM_386	4	Intel 80386
EM_68K	5	Motorola m68k family
EM_88K	6	Motorola m88k family
EM_860	7	Intel 80860
EM_MIPS	8	MIPS R3000 big-endian
EM_S370	9	IBM System/370
EM_MIPS_RS3_LE	10	MIPS R3000 little-endian

Altri valori sono riservati e verranno assegnati a nuove architetture. I nomi ELF specifici per processore usano il nome della macchina per distinguersi. Ad esempio, i flags riportati sotto usano il prefisso `EF_`: un flag chiamato WIDGET per la macchina `EM_XYZ` sarà definito `EF_XYZ_WIDGET`.

`e_version` identifica la versione

nome	valore	significato
EV_NONE	0	Invalid ELF version
EV_CURRENT	1	Current version
EV_NUM	2	

Il valore 1 indica il formato originale; future estensioni creeranno nuove versioni con numeri superiori. Il valore di `EV_CURRENT`, anche se impostato a 1, cambierà se necessario per riflettere il numero di versione corrente.

`e_entry` indica l'indirizzo virtuale al quale il sistema trasferisce il controllo, avviando così il processo. Se il file non ha un indirizzo iniziale, il campo vale zero.

`e_phoff` è l'offset della PHT in byte. Se manca, il valore è zero.

`e_shoff` è l'offset della SHT in byte. Se manca, il valore è zero.

`e_flags` contiene flags specifici per processore associati al file. I nomi sono nella forma `EF_macchina_flag`. Vedi "informazioni sul processore" per le definizioni.

`e_ehsize` è la dimensione dell'header ELF in byte.

`e_phentsize` è la dimensione in bytes di una voce della PHT; tutte le voci hanno la stessa dimensione.

`e_phnum` è il numero di voci nella PHT. Perciò il prodotto di `e_phentsize` e `e_phnum` dà la dimensione in byte della tabella. Se un file non ha la PHT, `e_phnum` vale zero.

`e_shentsize` è la dimensione di un header di sezione in bytes. E' una voce della SHT; tutte le voci hanno la stessa dimensione.

`e_shnum` è il numero delle voci nella SHT. Di conseguenza il prodotto di `e_shentsize` e `e_shnum` dà la dimensione della SHT in bytes. Se un file non ha SHT, `e_shnum` vale zero.

`e_shstrndx` è l'indice della voce nella SHT associata alla tabella delle stringhe dei nomi delle sezioni.

Se la tabella non è presente, il valore è SHN\_UNDEF. Vedi “Sezioni” e “Tabella delle stringhe” per maggiori informazioni.

## Identificazione ELF

Come ricordato prima, ELF fornisce una struttura che supporta una molteplicità di processori, codifiche di dati e hardware. Per questo scopo, i bytes iniziali del file definiscono come interpretare il file, indipendentemente dal processore su cui viene fatta l'analisi e dal resto del contenuto del file.

I primi byte dell'header ELF corrispondono al campo `e_ident`

Indici di identificazione `e_ident[ ]`

nome	valore	significato
EI_MAG0	0	File identification byte 0 index
EI_MAG1	1	File identification byte 1 index
EI_MAG2	2	File identification byte 2 index
EI_MAG3	3	File identification byte 3 index
EI_CLASS	4	File class byte index
EI_DATA	5	Data encoding byte index
EI_VERSION	6	File version byte index
EI_PAD	9	Byte index of padding bytes
EI_NIDENT	16	Sizeof <code>e_ident[ ]</code>

Questi indici permettono di accedere ai bytes che contengono i seguenti valori.

Da EI\_MAG0 a EI\_MAG3: I primi 4 byte contengono il “magic number”, che identifica il file come ELF.

nome	valore	significato
ELFMAG0	0x7f	Magic number byte 0
ELFMAG1	'E'	Magic number byte 1
ELFMAG2	'L'	Magic number byte 2
ELFMAG3	'F'	Magic number byte 3
ELFMAG	"\177ELF"	

EI\_CLASS Il byte successivo, `e_ident[EI_CLASS]` identifica la classe del file

nome	valore	significato
ELFCLASSNONE	0	Invalid class
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

Il formato è definito per essere portabile tra macchine di varie dimensioni, senza che sia imposta la dimensione maggiore su quella minore. La classe ELFCLASS32 supporta macchine con file e spazio di

indirizzi virtuali fino a 4 gigabyte; usa i tipi prima definiti.

La classe ELFCLASS64 è riservata per le architetture a 64 bit. La sua presenza qui dimostra come il file oggetto possa cambiare, ma il formato a 64 bit non è ulteriormente specificato. Se necessario verranno definite altre classi con diversi tipi base e diverse dimensioni per i dati.

**EI\_DATA** il byte `e_ident[EI_DATA]` definisce la codifica specifica del processore. Sono definite le codifiche seguenti.

nome	valore	significato
ELFDATANONE	0	Invalid data encoding
ELFDATA2LSB	1	2's complement, little endian
ELFDATA2MSB	2	2's complement, big endian

Maggiori informazioni su queste codifiche sono fornite in seguito. Altri valori sono riservati e saranno assegnati a nuove codifiche se necessario.

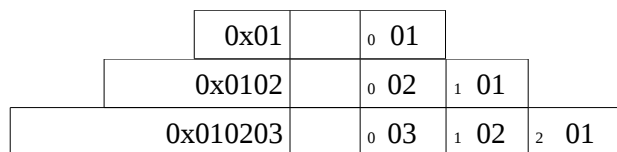
**EI\_VERSION** il byte `e_ident[EI_VERSION]` fornisce il numero di versione. Corrisponde a `EV_CURRENT`, come spiegato sopra per `e_version`.

**EI\_PAD** segna l'inizio dei byte non usati in `e_ident`. Sono riservati e posti a zero; i programmi che leggono il file li devono ignorare. Il valore di `EI_PAD` cambierà in futuro se i byte non usati assumeranno un significato.

La codifica dei dati definisce l'interpretazione degli oggetti di base del file. Come descritto sopra, la classe ELFCLASS32 usa oggetti che occupano 1, 2 o 4 bytes. In questa codifica, gli oggetti sono rappresentati come indicato sotto.

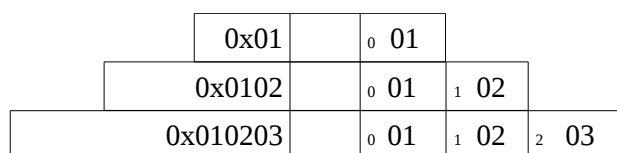
La codifica ELFDATA2LSB indica che i valori sono in complemento a 2, con il byte meno significativo che occupa l'indirizzo inferiore. (Little endian - ndt)

codifica ELFDATA2LSB



La codifica ELFDATA2MSB indica che i valori sono in complemento a 2, con il byte più significativo che occupa l'indirizzo inferiore (big endian - ndt).

codifica ELFDATA2MSB



Informazioni sull'hardware

Per l'identificazione del file in `e_ident`, l'architettura Intel a 32 bit richiede i valori seguenti:

<code>e_ident[EI_CLASS]</code>	ELFCLASS32
<code>e_ident[EI_DATA]</code>	ELFDATA2LSB

L'identificazione del processore sta nel campo `e_machine` dell'header ELF e deve avere il valore EM\_386. Il campo `e_flags` contiene flags associati al file. Per l'architettura Intel a 32 bit non sono definiti flags: perciò il campo contiene zero.

## Sezioni

La SHT permette di individuare tutte le sezioni del file. E' un array di strutture `Elf32_Shdr` descritte di seguito. Un indice della SHT è un indice dell'array. Il campo `e_shoff` dell'header ELF dà l'offset in bytes della SHT a partire dall'inizio del file; `e_shnum` indica il numero delle voci della SHT; `e_shentsize` dà le dimensioni in bytes di ogni voce.

Alcuni indici della SHT sono riservati; un file oggetto non ha sezioni per questi indici speciali.

### Indici delle sezioni speciali

nome	valore	significato
SHN_UNDEF	0	Undefined section
SHN_LORESERVE	0xff00	Start of reserved indices
SHN_LOPROC	0xff00	Start of processor-specific
SHN_HIPROC	0xff1f	End of processor-specific
SHN_ABS	0xfff1	Associated symbol is absolute
SHN_COMMON	0xfff2	Associated symbol is common
SHN_HIRESERVE	0xffff	End of reserved indices

`SHN_UNDEF` indica un riferimento di sezione indefinito, mancante, irrilevante o comunque privo di significato. Per esempio, un simbolo relativo a una sezione con numero `SHN_UNDEF` è un simbolo indefinito.

Nota: Anche se l'indice 0 è riservato come valore indefinito, la SHT contiene una voce con indice 0. Quindi, se il campo `e_shnum` dell'header ELF indica 6 voci per la SHT, esse hanno indice da 0 a 5. I contenuti della voce iniziale sono specificati più avanti in questa sezione.

`SHN_LORESERVE` specifica il limite inferiore nell'intervallo degli indici riservati

da `SHN_LOPROC` a `SHN_HIPROC` i valori in questo intervallo sono riservati a dati specifici per processore

`SHN_ABS` indica valori assoluti per i riferimenti corrispondenti. Per esempio i simboli definiti in relazione alla sezione `SHN_ABS` hanno valori assoluti e non sono soggetti a rilocazione

`SHN_COMMON` i simboli definiti in relazione a questa sezione sono simboli comuni, come FORTRAN COMMON o variabili esterne C non allocate.

`SHN_HIRESERVE` specifica il limite superiore nell'intervallo degli indici riservati. Il sistema riserva indici tra `SHN_LORESERVE` e `SHN_HIRESERVE`, compresi; questi valori non

referenziano la SHT. Cioè la SHT *non* contiene voci per gli indici riservati.

Le sezioni contengono tutte le informazioni presenti in un file oggetto, eccetto l'header ELF, la PHT e la SHT. In più, le sezioni soddisfano varie condizioni.

- Ogni sezione in un file ha esattamente un header di sezione che la descrive. Possono esistere header di sezione a cui non corrisponde una sezione.
- Ogni sezione occupa una sequenza contigua di bytes in un file.
- Le sezioni non possono sovrapporsi. Nessun byte in un file si trova in più di una sezione.
- Un file oggetto può avere spazio non occupato. I vari header e le sezioni possono non coprire ogni byte in un file. Il contenuto degli spazi non attivi è indefinito.

Un header di sezione ha la seguente struttura.

```
typedef struct {
    Elf32_Word  sh_name;           /* Section name (string tbl index) */
    Elf32_Word  sh_type;          /* Section type */
    Elf32_Word  sh_flags;         /* Section flags */
    Elf32_Addr  sh_addr;          /* Section virtual addr at execution */
    Elf32_Off   sh_offset;        /* Section file offset */
    Elf32_Word  sh_size;          /* Section size in bytes */
    Elf32_Word  sh_link;          /* Link to another section */
    Elf32_Word  sh_info;          /* Additional section information */
    Elf32_Word  sh_addralign;     /* Section alignment */
    Elf32_Word  sh_entsize;       /* Entry size if section holds table */
} Elf32_Shdr;
```

sh_name	contiene il nome della sezione. Il valore è un indice nella tabella delle stringhe delle sezioni (vedi più avanti “Tabella delle stringhe”) che dà la posizione di una stringa con terminatore \0.
sh_type	definisce il significato e i contenuti della sezione. I tipi e la loro descrizione sono elencati i seguito.
sh_flags	le sezioni hanno un flag di 1 bit che descrive vari attributi. I valori sono elencati in seguito.
sh_addr	se la sezione compare nell'immagine di memoria del processo, questo campo fornisce l'indirizzo del suo primo byte. Altrimenti il campo contiene 0.
sh_offset	contiene l'offset in bytes, rispetto all'inizio del file, del primo byte della sezione. Un tipo di sezione, SHT_NOBITS descritto poi, non occupa spazio nel file, e il suo campo sh_offset individua la posizione concettuale nel file.
sh_size	contiene la dimensione in bytes. A meno che il tipo non sia SHT_NOBITS, la sezione occupa sh_size byte nel file. Una sezione di tipo SHT_NOBITS può avere dimensione diversa da zero, ma non occupa spazio nel file.
sh_link	contiene un link all'indice della SHT, il cui significato dipende dal tipo di sezione. Una tabella più avanti elenca i valori.
sh_info	contiene informazioni ulteriori, la cui interpretazione dipende dal tipo di sezione. Una tabella descrive tali valori.

`sh_addralign` alcune sezioni hanno vincoli di allineamento. Per esempio, se una sezione contiene una `doubleword`, il sistema deve assicurare l'allineamento a `doubleword` per l'intera sezione. Quindi il valore di `sh_addr` modulo `sh_addralign` deve essere uguale a zero (cioè il resto della divisione intera tra `sh_addr` e `sh_addralign` deve essere zero - ndt). Sono ammessi solo valori pari a 0 o a una potenza intera di 2. I valori 0 e 1 indicano mancanza di vincoli di allineamento.

`sh_entsize` alcune sezioni contengono una tabella composta da voci di dimensioni fisse, come la tabella dei simboli. Per queste, il campo indica le dimensioni in byte di ogni voce. E' pari a zero se la sezione non contiene tabelle con voci di dimensioni fisse.

#### Tipi di sezione, `sh_type`

nome	valore	significato
SHT_NULL	0	Section header table entry unused
SHT_PROGBITS	1	Program data
SHT_SYMTAB	2	Symbol table
SHT_STRTAB	3	String table
SHT_RELA	4	Relocation entries with addends
SHT_HASH	5	Symbol hash table
SHT_DYNAMIC	6	Dynamic linking information
SHT_NOTE	7	Notes
SHT_NOBITS	8	Program space with no data (bss)
SHT_REL	9	Relocation entries, no addends
SHT_SHLIB	10	Dynamic linker symbol table
SHT_DYNSYM	11	Reserved
SHT_LOPROC	0x70000000	Start of processor-specific
SHT_HIPROC	0x7fffffff	End of processor-specific
SHT_LOUSER	0x80000000	Start of application-specific
SHT_HIUSER	0x8fffffff	End of application-specific

`SHT_NULL` contrassegna l'header di sezione come inattivo; esso non ha una sezione associata. Gli altri campi dell'header hanno valori indefiniti.

`SHT_PROGBITS` la sezione contiene informazioni definite dal programma, il cui formato e significato sono determinati solo dal programma stesso.

`SHT_SYMTAB` e `SHT_DYNSYM` queste sezioni contengono una tabella dei simboli. Attualmente un file oggetto può avere solo una sezione di ogni tipo, ma in futuro tale restrizione potrà essere eliminata. Tipicamente `SHT_SYMTAB` contiene i simboli per il linker, sebbene possa essere impiegata anche dal loader. In quanto tabella di simboli completa, può contenerne molti non necessari per il loader. Di conseguenza, un file può contenere anche una sezione `SHT_DYNSYM`, che contiene un insieme minimo di simboli per il linking dinamico, per risparmiare spazio. Vedi oltre "Tabella dei simboli" per i dettagli.

`SHT_STRTAB` la sezione contiene una tabella delle stringhe. Un file può averne più di una. Vedi oltre "Tabella delle stringhe" per i dettagli.

- SHT\_RELA** la sezione contiene le voci di rilocazione con addendi espliciti, come quelli del tipo `Elf32_Rela` per i file a 32 bit. Un file può averne più di una sezione di rilocazione. Vedi oltre “Rilocazione” per i dettagli.
- SHT\_HASH** contiene una tabella di hash dei simboli. Tutti gli oggetti coinvolti nel linking dinamico devono contenerne una. Attualmente può esistere solo una per file, ma tale limite potrà essere eliminato in futuro. Vedi “Tabella degli hash” nella Parte 2 per i dettagli.
- SHT\_DYNAMIC** la sezione contiene informazioni per il linking dinamico. Attualmente può esistere solo una tabella per file, ma il limite potrà essere eliminato in futuro. Vedi “Sezione dinamica” nella Parte 2 per i dettagli.
- SHT\_NOTE** contiene informazioni che identificano il file in qualche modo. Vedi “Sezione note” nella Parte 2 per i dettagli.
- SHT\_NOBITS** questo tipo non occupa spazio nel file; per il resto è simile a `SHT_PROGBITS`. Anche se non contiene bytes, il campo `sh_offset` contiene l'offset concettuale.
- SHT\_REL** la sezione contiene voci di rilocazione senza addendi espliciti, come quelli di tipo `Elf32_Rel` per i file a 32 bit. Un file oggetto può avere più di una sezione di rilocazione. Vedi oltre “Rilocazione” per i dettagli.
- SHT\_SHLIB** questo tipo è riservato ma ha significato non definito. I programmi che contengono sezioni di questo tipo non sono conformi alla ABI.

Da `SHT_LOPROC` a `SHT_HIPROC` questo intervallo, estremi compresi, è riservato per valori specifici per processore.

- SHT\_LOUSER** indica il limite inferiore dell'intervallo di indici riservati per i programmi applicativi.
- SHT\_HIUSER** indica il limite superiore dell'intervallo di indici riservati per i programmi applicativi. I tipi di sezione tra `SHT_LOUSER` e `SHT_HIUSER` possono essere usati per le applicazioni senza entrare in conflitto con tipi di sezioni definite attualmente o in futuro.

Gli altri valori di tipo di sezione sono riservati. Come ricordato prima, l'header di indice 0 (`SHN_UNDEF`) esiste, anche se contrassegna riferimenti di sezione indefiniti. Questa voce ha il seguente contenuto.

Nome	Valore	Note
<code>sh_name</code>	0	no name
<code>sh_type</code>	<code>SHT_NULL</code>	inactive
<code>sh_flags</code>	0	no flags
<code>sh_addr</code>	0	no address
<code>sh_offset</code>	0	no file offset
<code>sh_size</code>	0	no size
<code>sh_link</code>	<code>SHN_UNDEF</code>	no link information
<code>sh_info</code>	0	no auxiliary information
<code>sh_addralign</code>	0	no alignment
<code>sh_entsize</code>	0	no entries

Il campo `sh_flags` dell'header di sezione contiene un flag di 1 bit che descrive particolari attributi della sezione. I valori definiti sono i seguenti; altri valori sono riservati.

nome	valore	significato
SHF_WRITE	0x1	Writable
SHF_ALLOC	0x2	Occupies memory during execution
SHF_EXECINSTR	0x4	Executable
SHF_MASKPROC	0xf0000000	Processor-specific

Se in `sh_flags` il bit di flag è settato, l'attributo della sezione è "on". Altrimenti, l'attributo è "off" o non applicato. Attributi non definiti sono posti a zero.

SHF\_WRITE      la sezione contiene dati scrivibili durante l'esecuzione del processo

SHF\_ALLOC      la sezione occupa memoria durante l'esecuzione del processo. Alcune sezioni di controllo non risiedono nell'immagine di memoria di un file oggetto; per esse il flag è off.

SHF\_EXECINSTR      la sezione contiene codice macchina eseguibile.

SHF\_MASKPROC      tutti i bit sono riservati per valori processore-specifici.

Due campi dell'header di sezione, `sh_link` e `sh_info`, contengono informazioni speciali, dipendenti dal tipo di sezione.

sh_type	sh_link	sh_info
SHT_DYNAMIC	L'indice dell'header di sezione della tabella delle stringhe usato come voce nella sezione	0
SHT_HASH	L'indice dell'header di sezione della tabella dei simboli a cui si applica la tabella degli hash	0
SHT_REL SHT_RELA	L'indice dell'header di sezione della tabella dei simboli associata	L'indice dell'header di sezione della sezione a cui si applica la rilocalizzazione
SHT_SYMTAB SHT_DYNSYM	L'indice dell'header di sezione della tabella delle stringhe associata	Il maggiore degli indici della tabella dei simboli dell'ultimo simbolo locale (binding STB_LOCAL)
Altro	SHN_UNDEF	0

## Sezioni speciali

Diverse sezioni contengono programma e informazioni di controllo. Quelle elencate di seguito vengono usate dal sistema e hanno tipi e attributi indicati.

Nome	Tipo	Attributi
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	see below
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	see below
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	see below
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.plt	SHT_PROGBITS	see below
.relname	SHT_REL	see below
.relocate	SHT_RELA	see below
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_STRTAB	see below
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

**.bss** questa sezione contiene dati non inizializzati che fanno parte dell'immagine di memoria del programma. Per definizione, il sistema inizializza i dati a zero quando inizia l'esecuzione del programma. Non occupa spazio nel file, come indicato dal tipo SHT\_NOBITS.

**.comment** contiene informazioni di controllo della versione.

**.data e .data1** contengono dati inizializzati che fanno parte della immagine di memoria del programma.

**.debug** contiene informazioni di debugging. Il contenuto non è specificato.

**.dynamic** contiene informazioni per il linking dinamico. Gli attributi di questa sezione comprendono il bit SHF\_ALLOC. Dipende dal processore se il bit SHF\_WRITE è settato o no. Vedi Parte 2 per maggiori informazioni.

**.dynstr** contiene le stringhe necessarie al linking dinamico, di solito le stringhe che rappresentano i nomi associati alle voci della tabella dei simboli. Vedi Parte 2 per maggiori informazioni.

<code>.dysym</code>	contiene la tabella dei simboli per il linking dinamico, come indicato in “Tabella dei simboli”. Vedi Parte 2 per maggiori informazioni.
<code>.fini</code>	contiene le istruzioni macchina che contribuiscono al codice di terminazione del processo. Ciò significa che quando un programma termina normalmente, il sistema cerca di eseguire il codice in questa sezione.
<code>.got</code>	contiene la Global Offset Table (da ora in poi GOT). Vedi “Sezioni speciali” nella Parte 1 e “Global offset table” nella Parte 2 per maggiori informazioni.
<code>.hash</code>	contiene la tabella degli hash dei simboli. Vedi “Tabella degli hash” nella Parte 2 per maggiori informazioni.
<code>.init</code>	contiene le istruzioni macchina che contribuiscono al codice di inizializzazione del processo. Quando il programma inizia la sua esecuzione, il sistema cerca di eseguire il codice in questa sezione prima di saltare al punto d’inizio (la funzione main in C).
<code>.interp</code>	contiene il percorso nel filesystem dell’interprete del programma. Se il file ha un segmento caricabile in memoria che contiene questa sezione, i suoi attributi includeranno SHF_ALLOC; altrimenti, il bit sarà “off”. Vedi Parte 2 per maggiori informazioni.
<code>.line</code>	contiene le informazioni sui numeri di linea per il debugging, che descrivono la corrispondenza fra codice sorgente e codice macchina. Il contenuto non è specificato.
<code>.note</code>	contiene informazioni nel formato descritto in “Sezione note” della Parte 2.
<code>.plt</code>	contiene la Procedure Linkage Table (da ora in poi PLT) Vedi “Sezioni Speciali” nella Parte 1° e “Procedure linkage table” nella Parte 2° per maggiori informazioni.
<code>.relname</code> e <code>.relaname</code>	contengono informazioni di rilocazione, come descritto in “Rilocazione”. Se il file ha un segmento caricabile in memoria che comprende la rilocazione, gli attributi delle sezioni includono il bit SHF_ALLOC; altrimenti il bit è “off”. Per convenzione, <i>name</i> è fornito dalla sezione a cui si applica la rilocazione. Perciò una sezione di rilocazione per <code>.text</code> avrà normalmente il nome di <code>.rel.text</code> o <code>.rela.text</code> .
<code>.rodata</code> e <code>.rodata1</code>	contengono dati di sola lettura che normalmente fanno parte di una segmento non scrivibile nell’immagine del processo. Vedi “Header di programma” nella Parte 2 per maggiori informazioni.
<code>.shstrtab</code>	contiene i nomi delle sezioni.
<code>.strtab</code>	contiene stringhe, di solito quelle che rappresentano i nomi associati alle voci della tabella dei simboli. Se il file ha un segmento caricabile in memoria che comprende la tabella delle stringhe dei simboli, gli attributi comprenderanno il bit SHF_ALLOC; altrimenti il bit sarà “off”.
<code>.symtab</code>	contiene una tabella dei simboli, come descritto in “Tabella dei simboli”. Se il file ha un segmento caricabile in memoria che comprende la tabella dei simboli, gli attributi della sezione comprenderanno il bit SHF_ALLOC; altrimenti il bit sarà “off”.
<code>.text</code>	questa sezione contiene il “testo”, cioè il codice eseguibile del programma.

Le sezioni il cui nome è preceduto dal punto (.) sono riservate al sistema, anche se le applicazioni possono usarle se il loro significato è adatto. Possono usare i nomi senza il punto per evitare conflitti con le sezioni di

sistema. Lo standard del formato permette di definire sezioni non presenti nell'elenco. Un file può avere più di una sezione con lo stesso nome.

I nomi riservati ad una specifica architettura sono formati premettendo una abbreviazione dell'architettura stessa al nome della sezione. Il nome dovrebbe essere scelto tra i nomi di architettura usati per il campo `e_machine`. Per esempio `.foo.psect` è la sezione `psect` per l'architettura `foo`. Le estensioni esistenti sono identificate dal loro nome storico

Estensioni preesistenti

<code>.sdata</code>	<code>.tdesc</code>
<code>.sbss</code>	<code>.lit4</code>
<code>.lit8</code>	<code>.reginfo</code>
<code>.gptab</code>	<code>.liblist</code>
<code>.conflict</code>	

## Tabella delle stringhe

Una tabella delle stringhe contiene sequenze di caratteri con terminatore `\0`, comunemente chiamate stringhe. Il file le usa per rappresentare simboli e nomi di sezioni. Si fa riferimento ad una stringa puntando al suo indice nella tabella delle stringhe. Il primo byte, di indice zero, per definizione contiene un carattere nullo. Così pure l'ultimo byte deve contenere un carattere nullo per assicurare che tutte le stringhe abbiano terminatore `\0`. La stringa di indice zero indica o un nome non esistente o nullo, a seconda del contesto. E' ammessa una sezione della tabella delle stringhe vuota; il suo campo `sh_size` deve contenere zero. Indici diversi da zero non sarebbero validi per questa tabella.

Il campo `sh_name` di un header di sezione contiene un indice nella sezione della tabella delle stringhe, come indicato dal campo `e_shstrndx` dell'header ELF. La figura seguente mostra una tabella delle stringhe con 25 bytes e le stringhe associate a vari indici.

indice	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	<code>\0</code>	n	a	m	e	.	<code>\0</code>	V	a	r
10	i	a	b	l	e	<code>\0</code>	a	b	l	e
20	<code>\0</code>	<code>\0</code>	x	x	<code>\0</code>					

Indice	Stringa
0	<i>none</i>
1	name.
7	Variable
11	able
16	able
24	<i>null string</i>

Come mostrato nell'esempio, un indice può riferirsi a qualunque byte nella sezione. Una stringa può essere presente più di una volta; possono esistere riferimenti a sottostringhe e una singola stringa può essere referenziata più volte. Sono ammesse anche stringhe non referenziate.

## Tabella dei simboli

La tabella dei simboli di un file oggetto contiene informazioni necessarie per localizzare e rilocare definizioni e riferimenti simbolici del programma. Un indice della tabella dei simboli è un elemento di questo array. L'indice 0 identifica sia la prima voce della tabella, sia il simbolo indefinito. I contenuti della voce iniziale sono elencati più avanti in questa sezione

nome: STN\_UNDEF                      valore: 0

Una voce della tabella dei simboli ha il seguente formato.

```
typedef struct {
    Elf32_Word    st_name;           /* Symbol name (string tbl index) */
    Elf32_Addr    st_value;         /* Symbol value */
    Elf32_Word    st_size;         /* Symbol size */
    unsigned char st_info;         /* Symbol type and binding */
    unsigned char st_other;        /* Symbol visibility */
    Elf32_Section st_shndx;        /* Section index */
} Elf32_Sym;
```

**st\_name**                      contiene un indice nella tabella delle stringhe dei simboli, che contiene la rappresentazione a caratteri dei nomi dei simboli. Se il valore è diverso da zero, rappresenta un indice che fornisce il nome del simbolo. Nel caso contrario, la voce della tabella non ha nome.

Nota: i simboli extern del C hanno lo stesso nome nel C e nella tabella dei simboli del file.

**st\_value**                      questo campo fornisce il valore del simbolo associato. A seconda del contesto, può essere un valore assoluto, un indirizzo o altro; i dettagli sono elencati in seguito.

**st\_size**                      molti simboli hanno una dimensione. Per esempio, quella di una variabile è il numero di bytes. Contiene 0 se il simbolo non ha dimensione o ne ha una sconosciuta.

**st\_info**                      specifica il tipo di simbolo e gli attributi di collegamento (binding, vedi nota iniziale). Un elenco dei valori possibili è riportato più avanti. Il codice che segue mostra come manipolare i valori

<code>#define ELF32_ST_BIND(val)</code>	<code>(((unsigned char) (val)) &gt;&gt; 4)</code>
<code>#define ELF32_ST_TYPE(val)</code>	<code>((val) &amp; 0xf)</code>
<code>#define ELF32_ST_INFO(bind, type)</code>	<code>(((bind) &lt;&lt; 4) + ((type) &amp; 0xf))</code>

**st\_other**                      di regola ha valore 0 e non ha significato definito.

**st\_shndx**                      ogni voce della tabella dei simboli è “definita” in relazione ad una qualche sezione; questo campo contiene l'indice della tabella degli header di sezione relativa. Come indicato nella figura seguente, alcuni indice di sezione hanno significati speciali

Il collegamento (binding) di un simbolo definisce la visibilità e il comportamento del linking.

nome	valore	significato
STB_LOCAL	0	Local symbol
STB_GLOBAL	1	Global symbol

STB_WEAK	2	Weak symbol
STB_LOPROC	13	Start of processor-specific
STB_HIPROC	15	End of processor-specific

STB\_LOCAL i simboli locali non sono visibili al di fuori del file oggetto che li definisce. Simboli locali con lo stesso nome possono esistere in più file senza interferire l'uno con l'altro.

STB\_GLOBAL i simboli globali sono visibili a tutti i file oggetto collegati. La definizione di un simbolo globale in un file soddisfa il riferimento non definito in un altro file allo stesso simbolo.

STB\_WEAK questi simboli assomigliano a quelli globali, ma la loro definizione ha precedenza minore.

da STB\_LOPROC a STB\_HIPROC i valori in questo intervallo sono riservati ad elementi processore-specifici

Simboli globali e “weak” differiscono per due caratteristiche principali.

- quando il linker combina diversi file oggetto rilocabili, non ammette definizioni multiple di simboli STB\_GLOBAL con lo stesso nome. D'altra parte, se esiste un simbolo globale già definito, la presenza di un simbolo “weak” con lo stesso nome non provoca un errore. Il linker considera la definizione globale e ignora la “weak”. Allo stesso modo, se esiste un simbolo comune (cioè un simbolo il cui `st_shndx` vale SHN\_COMMON), la presenza di un simbolo “weak” con lo stesso nome non provoca un errore. Il linker considera la definizione del simbolo comune e ignora quello “weak”.
- Quando il linker cerca librerie in archivio, estrae i componenti che contengono definizioni di oggetti o simboli globali indefiniti. La definizione del componente può riferirsi a un simbolo globale o “weak”. Il linker *non* estrae i componenti di archivio per risolvere simboli “weak” non definiti. Simboli “weak” non risolti hanno valore zero.

In ogni tabella dei simboli, tutti quelli con collegamento STB\_LOCAL precedono quelli globali o “weak”. Come descritto in “Sezioni”, il campo `sh_info` di un header di sezione contiene l'indice della tabella dei simboli per il primo simbolo non locale.

Il tipo di simbolo fornisce una classificazione generale dei valori associati.

nome	valore	significato
STT_NOTYPE	0	Symbol type is unspecified
STT_OBJECT	1	Symbol is a data object
STT_FUNC	2	Symbol is a code object
STT_SECTION	3	Symbol associated with a section
STT_FILE	4	Symbol's name is file name
STT_LOPROC	13	Start of processor-specific
STT_HIPROC	15	End of processor-specific

STT\_NOTYPE tipo non specificato.

STT\_OBJECT simbolo associato a un oggetto dato, come una variabile, un array, etc.

STT\_FUNC simbolo associato ad una funzione o ad altro codice eseguibile.

**STT\_SECTION** simbolo associato ad una sezione. La voce della tabella dei simboli di questo tipo esiste in primo luogo per la rilocazione e normalmente ha collegamento **STB\_LOCAL**.

**STT\_FILE** per convenzione, il nome del simbolo riporta quello del file sorgente associato al file oggetto. Un simbolo file ha collegamento **STB\_LOCAL**, il suo indice di sezione è **SHN\_ABS** e precede gli altri simboli **STB\_LOCAL** se presenti.

da **STT\_LOPROC** a **STT\_HIPROC** i valori di questo intervallo, estremi compresi, sono riservati ad elementi processore-specifici.

I simboli funzione (con tipo **STT\_FUNC**) nelle librerie condivise hanno un significato speciale. Quando un altro file oggetto referencia una funzione in una libreria condivisa, il linker crea automaticamente una voce nella **PLT** per il simbolo referenziato. I simboli di oggetti condivisi con tipo diverso da **STT\_FUNC** non vengono invece referenziati automaticamente attraverso la **PLT**.

Se il valore di un simbolo fa riferimento a una locazione specifica entro una sezione, il suo campo indice di sezione, **st\_shndx**, contiene un indice nella **SHT**. Se la sezione viene spostata durante la rilocazione, il valore del simbolo cambia di conseguenza, e i riferimenti al simbolo continuano a puntare alla stessa locazione nel programma.

Alcuni valori di indici di sezioni speciali hanno altri significati.

**SHN\_ABS** il simbolo ha un valore assoluto che non cambia per effetto della rilocazione

**SHN\_COMMON** il simbolo identifica un blocco comune non ancora allocato. Il valore del simbolo dà dei vincoli di allineamento, come un elemento **sh\_addralign**. Quindi il linker allocherà la memoria per il simbolo a un indirizzo che sarà multiplo di **st\_value**. La dimensione del simbolo indica quanti bytes sono richiesti.

**SHN\_UNDEF** indica che il simbolo è indefinito. Quando il linker combina un file oggetto con un altro che definisce il simbolo indicato, il riferimento al simbolo del primo file sarà linkato alla definizione attuale.

Come riportato sopra, la voce della tabella dei simboli per l'indice 0 (**SHN\_UNDEF**) è riservata; il contenuto è il seguente.

Nome	Valore	Note
<b>st_name</b>	0	No name
<b>st_value</b>	0	Zero value
<b>st_size</b>	0	No size
<b>st_info</b>	0	No type, local binding
<b>st_other</b>	0	
<b>st_shndx</b>	<b>SHN_UNDEF</b>	No section

## Valore dei simboli

Le voci della tabella dei simboli per file oggetto differenti danno interpretazioni diverse al campo **st\_value**.

- in file rilocabili, **st\_value** contiene i vincoli di allineamento per un simbolo il cui indice è

## SHN\_COMMON

- in file rilocabili, `st_value` contiene un offset di sezione per un simbolo definito, cioè rappresenta l'offset dall'inizio della sezione identificata da `st_shndx`.
- negli eseguibili e nelle librerie condivise, `st_value` contiene un indirizzo virtuale. Per rendere i simboli di questi file più utili per il loader, l'offset di sezione (punto di vista del file) dà origine a un indirizzo virtuale (punto di vista della memoria) per il quale il numero di sezione è irrilevante.

Nonostante i valori della tabella dei simboli abbiano significato simile per file oggetto diversi, i dati consentono un accesso efficiente da parte dei programmi appropriati.

## Rilocazione

La rilocazione è la connessione dei riferimenti simbolici con le definizioni simboliche. Ad esempio, quando un programma chiama una funzione, l'istruzione `call` deve trasferire il controllo all'indirizzo di destinazione appropriato per l'esecuzione. In altre parole, i file rilocabili devono avere informazioni che descrivono come modificare il contenuto delle sezioni, facendo in modo che i file eseguibili e le librerie abbiano le istruzioni adatte per creare l'immagine del processo. Le voci di rilocazione sono queste istruzioni.

Voci di rilocazione.

```
typedef struct {
    Elf32_Addr  r_offset;          /* Address */
    Elf32_Word  r_info;           /* Relocation type and symbol index */
} Elf32_Rel;
```

```
typedef struct {
    Elf32_Addr  r_offset;          /* Address */
    Elf32_Word  r_info;           /* Relocation type and symbol index */
    Elf32_Sword r_addend;         /* Addend */
} Elf32_Rela;
```

`r_offset` fornisce la locazione a cui applicare la rilocazione. Per un file rilocabile, il valore è l'offset in bytes relativo all'inizio della sezione della unità di memorizzazione interessata alla rilocazione. Per un eseguibile o una libreria condivisa, il valore è l'indirizzo virtuale dell'unità di memorizzazione interessata dalla rilocazione.

`r_info` fornisce sia l'indice della tabella dei simboli rispetto alla quale deve essere effettuata la rilocazione, sia il tipo di rilocazione da applicare. Per esempio, la voce di rilocazione di una chiamata di funzione dovrebbe contenere l'indice nella tabella dei simboli della funzione da chiamare. Se l'indice è `STN_UNDEF`, ovvero il valore indefinito, la rilocazione usa 0 come "valore del simbolo". I tipi di rilocazione sono processore-specifici. Quando il testo parla del tipo di rilocazione o dell'indice della tabella dei simboli di una voce di rilocazione, si riferisce al risultato dell'applicazione di `ELF32_R_TYPE` o `ELF32_R_SYM` rispettivamente, al campo `r_info`.

Come manipolare le informazioni nel campo `r_info`.

<code>#define ELF32_R_SYM(val)</code>	<code>((val) &gt;&gt; 8)</code>
<code>#define ELF32_R_TYPE(val)</code>	<code>((val) &amp; 0xff)</code>
<code>#define ELF32_R_INFO(sym, type)</code>	<code>((sym) &lt;&lt; 8) + ((type) &amp; 0xff)</code>

`r_addend` contiene un addendo costante usato per calcolare il valore da memorizzare nel campo

rilocabile.

Come mostrato in precedenza, solo le voci `Elf32_Rel` contengono un addendo esplicito. Il tipo `Elf32_Rel` inserisce un addendo implicito nella locazione da modificare. A seconda dell'architettura del processore, può essere conveniente o necessaria una forma piuttosto che l'altra. Di conseguenza, l'implementazione per una macchina specifica può usare esclusivamente una forma o impiegarle entrambe, a seconda del contesto.

Una sezione di rilocazione referencia due altre sezioni: una tabella dei simboli e una sezione da modificare. I membri dell'header di sezione `sh_info` e `sh_link`, descritti in precedenza nel paragrafo “Sezioni”, specificano queste relazioni. Le voci di rilocazione di differenti tipi di file interpretano in maniera diversa il campo `r_offset`.

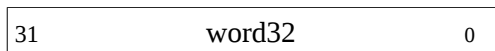
- nei file rilocabili, `r_offset` contiene un offset di sezione. Perciò la sezione di rilocazione stessa descrive come modificare un'altra sezione del file; gli offset di rilocazione indicano una unità di memorizzazione entro la seconda sezione.
- nei file eseguibili e nelle librerie condivise, `r_offset` contiene un indirizzo virtuale. Per rendere queste voci di rilocazione utili per il loader, l'offset di sezione (prospettiva del file) dà origine a un indirizzo virtuale (prospettiva della memoria).

Anche se l'interpretazione di `r_offset` cambia per tipi differenti di file per consentire un accesso efficiente da parte del programma interessato, il significato del tipo di rilocazione rimane lo stesso.

## Tipi di rilocazione

Le voci di rilocazione descrivono come alterare le istruzioni e i campi dati seguenti (il numero di bit compare nell'angolo in basso del riquadro).

Campi di rilocazione.



`word32` definisce un campo di 32 bit (4 bytes) con allineamento arbitrario. I valori sono indicati nell'ordine dei bytes dell'architettura Intel 32-bit.

0x01020304



Il procedimento descritto di seguito ipotizza la trasformazione di un file rilocabile in un eseguibile o una libreria condivisa. Concettualmente, il linker unisce uno o più file rilocabili per produrre un output. Per prima cosa decide come combinare e allocare i file originari, poi aggiorna il valore dei simboli e infine realizza la rilocazione. La rilocazione applicata agli eseguibili o alle librerie è simile e porta allo stesso risultato. La descrizione seguente usa questa notazione:

- A è l'addendo usato per calcolare il valore del campo rilocabile
- B è l'indirizzo base della memoria in cui è stato caricato un oggetto condiviso durante l'esecuzione. Generalmente una libreria condivisa è costruita con indirizzo virtuale di base pari a 0, ma l'indirizzo a runtime sarà diverso.
- G è l'offset relativo alla GOT nel quale si trova l'indirizzo del simbolo della voce di rilocazione durante l'esecuzione. Vedi “Global offset table” nella Parte 2 per maggiori dettagli.

- GOT è l'indirizzo della GOT. Vedi "Global offset table" nella Parte 2 per maggiori dettagli.
- L indica la posizione (offset di sezione o indirizzo) della voce della PLT relativa ad un simbolo. Una voce della PLT ridirige una chiamata di funzione alla opportuna destinazione. Il linker costruisce la PLT iniziale, mentre il loader ne modifica le voci durante l'esecuzione. Vedi "Procedure linkage table" nella Parte 2 per maggiori dettagli.
- P indica la posizione (offset di sezione o indirizzo) della unità di memorizzazione che viene rilocata (calcolata usando `r_offset`).
- S indica il valore del simbolo il cui indice si trova nella voce di rilocazione.

Il valore di `r_offset` di una voce di rilocazione contiene l'offset o l'indirizzo virtuale del primo byte della unità di memorizzazione interessata. Il tipo di rilocazione specifica quale bit cambiare e come calcolarne il valore. System V usa solo rilocazioni del tipo `Elf32_Rel`, e il campo da rilocare contiene l'addendo. In ogni caso addendo e risultato calcolato usano lo stesso ordine dei byte.

#### Tipi di rilocazione

nome	valore	campo	calcolo
R_386_NONE	0	0	None
R_386_32	1	word32	S + A
R_386_PC32	2	word32	S + A - P
R_386_GOT32	3	word32	G + A - P
R_386_PLT32	4	word32	L + A - P
R_386_COPY	5	none	none
R_386_GLOB_DAT	6	word32	S
R_386_JMP_SLOT	7	word32	S
R_386_RELATIVE	8	word32	B + A
R_386_GOTOFF	9	word32	S + A - GOT
R_386_GOTPC	10	word32	GOT + A - P

Alcuni tipi hanno anche un significato oltre al semplice metodo di calcolo.

- R\_386\_GOT32 calcola la distanza tra la base della GOT e la voce del simbolo nella GOT. Inoltre fornisce istruzioni al linker per costruire una GOT.
- R\_386\_PLT32 calcola l'indirizzo della voce del simbolo nella PLT e inoltre fornisce istruzioni al linker per costruire una PLT.
- R\_386\_COPY il linker crea questo tipo di rilocazione per il linking dinamico. Il suo campo offset punta a una locazione in un segmento scrivibile. L'indice della tabella dei simboli identifica un simbolo che deve esistere sia nel file oggetto corrente sia in un oggetto condiviso. Durante l'esecuzione il loader copia i dati associati al simbolo dell'oggetto condiviso nella locazione indicata dall'offset.
- R\_386\_GLOB\_DAT questo tipo di rilocazione è usato per assegnare a una voce della GOT l'indirizzo del simbolo specificato. Questo tipo speciale permette di determinare la corrispondenza tra simboli e voci della GOT.

- R\_386\_JMP\_SLOT      il linker crea questo tipo per il linking dinamico. Il suo campo offset fornisce la locazione di una voce della PLT. Il loader modifica la voce della PLT per trasferire il controllo all'indirizzo del simbolo in questione (vedi "Procedure linkage table" nella Parte 2).
- R\_386\_RELATIVE      il linker crea questo tipo per il linking dinamico. Il suo campo offset indica una locazione entro un oggetto condiviso che contiene un valore che rappresenta un indirizzo relativo. Il loader calcola il corrispondente indirizzo virtuale aggiungendo all'indirizzo relativo l'indirizzo virtuale a cui è stato caricato l'oggetto condiviso. Le voci di rilocazione di questo tipo devono specificare 0 per l'indice della tabella dei simboli.
- R\_386\_GOTOFF      calcola la differenza tra il valore di un simbolo e l'indirizzo della GOT. Inoltre fornisce istruzioni al linker per la costruzione della GOT.
- R\_386\_GOTPC      assomiglia al tipo R\_386\_PC32, ma usa l'indirizzo della GOT nel calcolo. Il simbolo normalmente referenziato è `_GLOBAL_OFFSET_TABLE_`, che fornisce ulteriori istruzioni al linker per la costruzione della GOT.

## PARTE 2

# CARICAMENTO E LINKING DINAMICO DEI PROGRAMMI

### Introduzione

La Parte 2 descrive le istruzioni dei file oggetto e le azioni del sistema che creano i programmi in esecuzione. Alcune informazioni si applicano a tutti i sistemi, altre sono specifiche di ogni processore.

Gli eseguibili e i file di libreria rappresentano staticamente il programma. Per eseguirli, il sistema usa i file per creare una rappresentazione dinamica detta immagine del processo. Essa ha segmenti che contengono codice, dati, stack e così via. I paragrafi di questa Parte trattano i seguenti argomenti.

- *Header del programma.* Questa sezione integra la Parte 1 descrivendo le strutture del file oggetto che riguardano direttamente l'esecuzione del programma. La struttura principale, la PHT, localizza le immagini dei segmenti entro il file e contiene altre informazioni necessarie a creare l'immagine in memoria del programma.
- *Caricamento del programma.* Dato un file oggetto, il sistema deve caricarlo in memoria per eseguire il programma.
- *Linking dinamico.* Dopo il caricamento, il sistema deve completare l'immagine del processo risolvendo i riferimenti simbolici tra i file oggetto che formano il processo.

Nota: esistono convenzioni per i nomi per le costanti ELF che contengono valori legati al processore. Nomi come DT\_, PT\_, per estensioni specifiche per ogni processore, ne incorporano il nome: ad es. DT\_M32\_SPECIAL. Le estensioni preesistenti che non usano tale convenzione vengono però supportate.

Estensioni preesistenti: DT\_JMP\_REL

### Header del programma

L'header del programma (d'ora in poi PH - ndt) di un eseguibile o di una libreria condivisa è un array di strutture, ognuna delle quali descrive un segmento o altra informazione di cui il sistema ha bisogno per preparare il programma per l'esecuzione. Un segmento contiene una o più sezioni come descritto in "Contenuto dei Segmenti". PH ha significato solo per eseguibili e librerie condivise. I valori `e_phentsize` e `e_phnum` dell'header ELF definiscono le dimensioni di PH.

```
typedef struct {
    Elf32_Word  p_type;           /* Segment type */
    Elf32_Off   p_offset;        /* Segment file offset */
    Elf32_Addr  p_vaddr;         /* Segment virtual address */
    Elf32_Addr  p_paddr;        /* Segment physical address */
    Elf32_Word  p_filesz;        /* Segment size in file */
    Elf32_Word  p_memsz;         /* Segment size in memory */
    Elf32_Word  p_flags;         /* Segment flags */
    Elf32_Word  p_align;         /* Segment alignment */
} Elf32_Phdr;
```

`p_type` specifica il tipo di segmento che l'elemento dell'array descrive, o come interpretare le informazioni dell'elemento stesso. I tipi possibili e il loro significato sono elencati in seguito.

<code>p_offset</code>	contiene l'offset dall'inizio del file del primo byte del segmento
<code>p_vaddr</code>	contiene l'indirizzo virtuale del primo byte del segmento in memoria
<code>p_paddr</code>	nei sistemi per i quali ha un significato, questo elemento contiene l'indirizzo fisico del segmento. Poiché System V ignora l'indirizzamento fisico per i programmi applicativi, questo elemento ha un contenuto non specificato per eseguibili e librerie condivise.
<code>p_filesz</code>	contiene il numero di bytes del segmento nell'immagine del file; può essere zero.
<code>p_memsz</code>	contiene il numero di bytes del segmento nell'immagine di memoria; può essere zero.
<code>p_flags</code>	contiene flags importanti per il segmento. I possibili valori sono riportati più avanti.
<code>p_align</code>	come descritto più oltre in “Caricamento del programma”, i segmenti di un processo caricabile devono avere valori congruenti per <code>v_addr</code> e <code>p_offset</code> modulo la dimensione della pagina. Questo elemento fornisce il valore di allineamento del segmento in memoria e nel file. Valori 0 e 1 indicano nessun allineamento richiesto. Altrimenti <code>p_align</code> deve essere una potenza di 2 intera e positiva, e <code>p_vaddr</code> deve essere uguale a <code>p_offset</code> modulo <code>p_align</code> .

Alcune voci descrivono i segmenti del processo; altre forniscono informazioni supplementari e non contribuiscono all'immagine del processo. Le voci relative ai segmenti possono apparire in qualsiasi ordine, eccetto i casi definiti in seguito. Seguono i valori definiti per il tipo; altri valori sono riservati per usi futuri.

#### Tipi di segmento, `p_type`

nome	valore	significato
PT_NULL	0	Program header table entry unused
PT_LOAD	1	Loadable program segment
PT_DYNAMIC	2	Dynamic linking information
PT_INTERP	3	Program interpreter
PT_NOTE	4	Auxiliary information
PT_SHLIB	5	Reserved
PT_PHDR	6	Entry for header table itself
PT_LOPROC	0x70000000	Start of processor-specific
PT_HIPROC	0x7fffffff	End of processor-specific

`PT_NULL` elemento dell'array non utilizzato; gli altri membri sono indefiniti. Permette di avere nella PHT voci da ignorare.

`PT_LOAD` specifica un segmento caricabile, descritto da `p_filesz` e `p_memsz`. I bytes del file sono mappati all'inizio del segmento di memoria. Se la dimensione di quest'ultimo (`p_memsz`) è maggiore di quella del file (`p_filesz`), i bytes “eccedenti” hanno per definizione valore 0 e sono posti di seguito all'area inizializzata del segmento. La dimensione del file non può essere maggiore di quella della memoria. Le voci relative ai segmenti caricabili nella PHT compaiono in ordine crescente, ordinati per valore di `p_vaddr`.

`PT_DYNAMIC` fornisce informazioni per il linking dinamico. Vedi oltre “Sezione dinamica” per

maggiori informazioni.

PT_INTERP	contiene la locazione e la dimensione del percorso nel filesystem (con carattere di terminazione \0) dell'eseguibile da invocare come interprete. Questo elemento è significativo solo per gli eseguibili (anche se può comparire nelle librerie condivise); non può essere presente più di una volta. Se presente, deve precedere ogni voce di segmento caricabile. Vedi oltre “Interprete del programma” per ulteriori informazioni.
PT_NOTE	contiene la locazione e la dimensione di informazioni ausiliarie. Vedi oltre “Sezione note” per ulteriori informazioni.
PT_SHLIB	riservato, con significato non specificato. Programmi che contengono questo elemento non sono conformi alla ABI.
PT_PHDR	se presente, contiene locazione e dimensione della PHT, sia nel file che nell'immagine di memoria. Questo tipo di segmento non può essere presente più di una volta in un file. Inoltre può essere presente solo se la PHT fa parte dell'immagine di memoria. Se presente, deve precedere ogni voce di segmento caricabile. Vedi oltre “Interprete del programma” per ulteriori informazioni.

da PT\_LOPROC a PT\_HIPROC i valori in questo intervallo, inclusi gli estremi, sono riservati a valori processore-specifici.

Note: Se non espressamente richiesti, tutti i tipi di segmento sono opzionali. Perciò la PHT di un file può contenere solo gli elementi rilevanti per i suoi contenuti.

## Indirizzo base

Eseguibili e librerie condivise hanno un indirizzo base, l'indirizzo virtuale inferiore associato alla immagine di memoria del file. Un impiego dell'indirizzo base è la rilocazione dell'immagine di memoria durante il linking dinamico.

L'indirizzo base è calcolato durante l'esecuzione in base a tre valori: l'indirizzo di caricamento in memoria, la dimensione massima della pagina, l'indirizzo virtuale inferiore di un segmento caricabile del programma. Come descritto in “Caricamento del programma”, gli indirizzi virtuali nel PH possono non rappresentare gli indirizzi virtuali dell'immagine del programma in memoria. Per calcolare l'indirizzo base, si determina l'indirizzo di memoria associato al valore inferiore di `p_vaddr` per un segmento PT\_LOAD. Poi si ottiene l'indirizzo base troncando l'indirizzo di memoria al multiplo più prossimo della dimensione massima della pagina. A seconda del tipo di file, l'indirizzo di memoria può coincidere o meno con `p_vaddr`.

Come descritto in “Sezioni” nella Parte 1, la sezione `.bss` è di tipo SHT\_NOBITS. Anche se non occupa spazio nel file, contribuisce all'immagine di memoria. Normalmente questi dati non inizializzati si trovano alla fine del segmento, rendendo perciò `p_memsz` di maggiore di `p_filesz`.

## Sezione note

Talvolta uno sviluppatore deve identificare un file oggetto con informazioni speciali che altri programmi possano verificare per conformità o compatibilità. Le sezioni del tipo SHT\_NOTE e gli elementi del PH di tipo PT\_NOTE possono essere impiegate a questo scopo. Il campo note nelle sezioni e negli elementi del PH contiene un numero qualsiasi di voci, ognuna delle quali è un array di word di 4 bytes nel formato richiesto dal processore. Le etichette qui riportate per spiegare l'organizzazione delle note non sono parte dello standard.

Namesz
Descsz

Type
Name
Desc

**namesz e name** i primi `namesz` bytes in `name` contengono la rappresentazione a caratteri con terminatore `\0` del proprietario o creatore della voce. Non ci sono metodi formalizzati per evitare conflitti di nomi. Per convenzione, i produttori usano il proprio nome, ad es. “XYZ Computer Company” come identificatore. Se non è presente un nome, `namesz` contiene 0. Ci sono spaziatori, se necessario, per assicurare l'allineamento a 4 byte. Gli spaziatori non vengono inclusi in `namesz`.

**descsz e desc** i primi `descsz` bytes in `desc` contengono il descrittore di nota. L'ABI non pone vincoli al contenuto del descrittore. Se non è presente, `descsz` vale 0. Ci sono spaziatori, se necessario, per assicurare l'allineamento a 4 byte; non vengono inclusi in `descsz`.

**type** fornisce l'interpretazione del descrittore. Ogni fonte controlla i propri tipi; possono esistere interpretazioni multiple di un singolo tipo. Perciò un programma deve riconoscere sia il nome che il tipo per interpretare un descrittore. Come regola, i tipi non devono essere negativi. L'ABI non definisce il significato dei descrittori.

Come esempio, il segmento note riportato contiene due voci

	+0	+1	+2	+3	
namesz	7				No descriptor
descsz	0				
type	1				
name	X	Y	Z		
	C	o	\0	pad	
namesz	7				
descsz	8				
type	3				
name	X	Y	Z		
	C	o	\0	pad	
desc	word 0				
	word 1				

Nota: il sistema riserva note senza nome (`namesz == 0`) e con nome di lunghezza zero (`name[0] == '\0'`) ma attualmente non definisce tipi. Tutti gli altri nomi devono avere almeno un carattere non nullo.

Nota: Le note sono opzionali. La presenza di note non incide sulla conformità alla ABI, e neppure sulla esecuzione del programma.

### Caricamento del programma

Quando il sistema crea o amplia l'immagine del processo, copia un segmento del file in un segmento di memoria virtuale. Quando – e se – il sistema legge fisicamente il file, dipende dal comportamento del

programma, dal carico del sistema, ecc.

Un processo non richiede una pagina fisica fino a quando non riferenzia la pagina logica durante l'esecuzione, e i processi di solito lasciano molte pagine non referenziate. Perciò ritardare la lettura del supporto fisico spesso permette di evitarla, aumentando l'efficienza del sistema. Per ottenere in pratica questo risultato, occorre che gli eseguibili e le librerie condivise abbiano immagini di segmento con offset rispetto al file e indirizzi virtuali congruenti, modulo la dimensione di pagina.

Indirizzi virtuali e offset relativi al file per i segmenti dell'architettura System V sono congruenti modulo 4 KB (0x1000) o potenze di 2 superiori. Poiché 4KB è la dimensione massima di pagina, il file risulta adatto alla paginazione indipendentemente dalla dimensione della pagina fisica.

File eseguibile.

File Offset	File	Virtual Address
0	ELF header	
PHT		
	Other information	
0x100	Text segment	0x8048100
	...	
	0x2be00 bytes	0x8073eff
0x2bf00	Data segment	0x8074f00
	...	
	0x4e00 bytes	0x8079cff
0x30d00	Other information	
	...	

program header segments

campo	codice	dati
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x1000	0x1000

Anche se gli offset e i gli indirizzi virtuali nell'esempio sono congruenti modulo 4 KB per codice e dati, almeno quattro pagine contengono codice o dati misti (a seconda della dimensione di pagina e del blocco dati del filesystem)

- la prima pagina di codice contiene l'header ELF, la PHT e altre informazioni
- l'ultima pagina di codice contiene una copia della parte iniziale dei dati
- la prima pagina di dati ha una copia della parte terminale del codice
- l'ultima pagina di dati può contenere informazioni non rilevanti per il processo in esecuzione.

Logicamente il sistema attribuisce i permessi della memoria come se ogni segmento fosse completo e separato; gli indirizzi del segmento sono stabiliti in modo da assicurare che ogni pagina logica nello spazio

degli indirizzi abbia un singolo insieme di permessi. Nell'esempio precedente la regione del file che contiene la parte terminale del codice e l'inizio dei dati viene mappata due volte: a un indirizzo virtuale per il codice e a un indirizzo differente per i dati.

La parte finale del segmento dati richiede una gestione speciale per i dati non inizializzati, che per il sistema hanno valore iniziale zero. Se l'ultima pagina dati include informazioni non presenti nella pagina logica di memoria, i dati estranei devono essere inizializzati a zero, non i contenuti sconosciuti dell'eseguibile. Le "impurità" nelle altre tre pagine non sono parte logica dell'immagine di processo: non è specificato se il sistema debba eliminarli. Segue la rappresentazione dell'immagine in memoria, con pagine di 4 KB (0x1000).

Process image segments

Virtual address	Contents	Segment
0x8048000	Header padding 0x100 bytes	Text
0x8048100	Text segment ... 0x2be00 bytes	
0x8073f00	Data padding 0x100 bytes	
0x8074000	Text padding 0xf00 bytes	Data
0x8074f00	Data segment ... 0x4e00 bytes	
0x8079d00	Uninitialized data 0x1024 zero bytes	
0x807ad24	Page padding 0x2dc zero bytes	

Tra eseguibili e librerie condivise, un aspetto del caricamento dei segmenti è diverso. I segmenti degli eseguibili tipicamente contengono codice assoluto. Perché il processo venga eseguito correttamente, i segmenti devono trovarsi all'indirizzo virtuale usato per costruire il file eseguibile. Quindi il sistema usa senza modifiche il valore di `p_vaddr` come indirizzo virtuale.

D'altra parte, i segmenti delle librerie condivise contengono codice indipendente dalla posizione. Questo fa sì che l'indirizzo virtuale di un segmento cambi da un processo ad un altro, senza compromettere l'andamento dell'esecuzione. Anche se il sistema sceglie indirizzi virtuali per i processi individuali, mantiene le *posizioni relative* dei segmenti. Poiché il codice indipendente dalla posizione usa l'indirizzamento relativo tra segmenti, la differenza tra indirizzi virtuali in memoria deve coincidere con la differenza tra indirizzi virtuali nel file. La tabella seguente mostra una possibile assegnazione di indirizzi virtuali di librerie per diversi processi, illustrando il posizionamento relativo costante. Mostra anche il calcolo dell'indirizzo base.

Origine	Codice	Dati	Indirizzo base
File	0x200	0x2a400	0x0
Processo 1	0x80000200	0x8002a400	0x80000000
Processo 2	0x80081200	0x800ab400	0x80081000

Origine	Codice	Dati	Indirizzo base
Processo 3	0x900c0200	0x900ea400	0x900c0000
Processo 4	0x900c6200	0x900f0400	0x900c6000

## Linking dinamico

### Interprete del programma

Un programma eseguibile può avere un elemento PT\_INTERP nel PH. Durante exec(BA\_OS), il sistema estrae il percorso dell'interprete dal segmento PT\_INTERP e crea l'immagine iniziale del processo a partire dai segmenti dell'interprete. Invece di usare l'immagine del segmento del file eseguibile originale, il sistema compone una immagine in memoria per l'interprete. E' poi responsabilità dell'interprete ricevere il controllo dal sistema e fornire un ambiente per il programma applicativo.

L'interprete riceve il controllo in due modi. Nel primo, può ricevere un descrittore per leggere il file eseguibile, partendo dall'inizio. Può usare il descrittore per leggere e/o mappare i segmenti dell'eseguibile in memoria. Nel secondo, il sistema può caricare l'eseguibile in memoria, invece di passare un descrittore all'interprete. Con la possibile eccezione del descrittore, lo stato iniziale del processo dell'interprete coincide con quello che l'eseguibile dovrebbe ricevere. L'interprete stesso non può richiedere un secondo interprete. Un interprete può essere o un file di libreria o un eseguibile.

- Una libreria condivisa (è il caso normale) ha un caricamento indipendente dalla posizione, con indirizzi che possono variare da processo a processo; il sistema crea i suoi segmenti nell'area del segmento dinamico usata da mmap (KE\_OS) e dai servizi relativi. Di conseguenza, un interprete da libreria condivisa non entra in conflitto con gli indirizzi del segmento originale dell'eseguibile.
- Un eseguibile è caricato a un indirizzo fisso; il sistema crea i suoi segmenti usando l'indirizzo virtuale presente nella PHT. Di conseguenza, gli indirizzi di un interprete eseguibile possono collidere con quelli del primo eseguibile; l'interprete è responsabile della risoluzione dei conflitti.

### Loader

Durante la costruzione di un file che usa il linking dinamico, i linker aggiunge al PH un elemento di tipo PT\_INTERP, istruendo il sistema ad invocare il loader come interprete del programma.

Nota: La posizione del loader è specifica per ogni processore.

Exec(BA\_OS) e il loader cooperano per creare l'immagine del processo, il che comporta le seguenti azioni:

- aggiungere all'immagine del processo i segmenti di memoria dell'eseguibile
- aggiungere all'immagine del processo i segmenti di memoria dell'oggetto condiviso
- effettuare la rilocazione dell'eseguibile e degli oggetti condivisi
- chiudere il descrittore di file usato per leggere l'eseguibile, se ricevuto
- trasferire il controllo al programma, facendo sembrare che lo stesso abbia ricevuto il controllo direttamente da exec(BA\_OS).

Il linker crea anche vari dati che aiutano il loader per gli eseguibili e le librerie condivise. Come mostrato in "Header di Programma", essi risiedono in segmenti caricabili che li rendono disponibili durante l'esecuzione. (ricordare che il contenuto esatto dei segmenti è specifico per processore. Vedi la documentazione del processore per informazioni più complete).

- una sezione `.dynamic` con tipo SHT\_DYNAMIC contiene vari dati. Le strutture all'inizio della sezione contengono gli indirizzi di altre informazioni per il linking dinamico
- la sezione `.hash` con tipo SHT\_HASH contiene la tabella degli hash dei simboli
- le sezioni `.got` e `.plt` con tipo SHT\_PROGBITS contengono due tabelle diverse: la GOT e la PLT. La sezione seguente spiega come il loader usa e cambia le tabelle per creare l'immagine di

memoria.

Poiché ogni programma conforme all' ABI importa le funzioni fondamentali da una libreria condivisa, il loader prende parte ad ogni esecuzione di programma.

Come spiegato in “Caricamento del programma”, le librerie possono occupare in memoria indirizzi virtuali differenti da quelli contenuti nella PHT del file. Il loader ricolloca l'immagine di memoria aggiornando gli indirizzi assoluti prima che l'applicazione prenda il controllo. Anche se i valori assoluti sarebbero corretti se la libreria venisse caricata all'indirizzo specificato nella PHT, questo normalmente non avviene.

Se l'ambiente del processo contiene la variabile LD\_BIND\_NOW con un valore non nullo, il loader elabora tutte le rilocalizzazioni prima di trasferire il controllo al programma. Per esempio, tutte le variabili d'ambiente riportate specificano questo comportamento:

- LD\_BIND\_NOW = 1
- LD\_BIND\_NOW = on
- LD\_BIND\_NOW = off

Nel caso contrario, LD\_BIND\_NOW non è definita o ha valore nullo. Il loader può eseguire le procedure di linking in modo “lazy”, evitando la risoluzione dei simboli e il sovraccarico della rilocalizzazione per funzioni non ancora chiamate. Vedi “Procedure linkage table” per maggiori informazioni.

## Sezione dinamica

Se un file è coinvolto in un linking dinamico, la sua PHT avrà un elemento di tipo PT\_DYNAMIC.

Questo “segmento” contiene la sezione .dynamic. Un simbolo speciale, \_DYNAMIC, identifica la sezione, che contiene un array delle seguenti strutture.

```
typedef struct {
    Elf32_Sword d_tag;           /* Dynamic entry type */
    union {
        Elf32_Word d_val;       /* Integer value */
        Elf32_Addr d_ptr;       /* Address value */
    } d_un;
} Elf32_Dyn;

extern ELF32_Dyn_DYNAMIC[ ];
```

Per ogni oggetto con questo tipo, d\_tag controlla l'interpretazione di d\_un.

d\_val è un valore intero con vari significati

d\_ptr rappresenta indirizzi virtuali del programma. Come ricordato prima, gli indirizzi virtuali di un file possono non coincidere con gli indirizzi virtuali in memoria durante l'esecuzione. Quando vengono interpretati gli indirizzi contenuti nella struttura dinamica, il loader calcola gli indirizzi attuali sulla base del file originale e dell'indirizzo base della memoria. Per coerenza, i file *non* contengono voci di rilocalizzazione per “correggere” gli indirizzi nella struttura dinamica.

La tabella seguente riassume i tag necessari per eseguibili e librerie condivise. Se un tag è definito “mandatory” (obbligatorio - ndt), deve obbligatoriamente essere presente. “Optional” significa che il tag può essere presente ma non è necessario.

nome	valore	d_un	eseguibile	librerie
DT_NULL	0	ignored	mandatory	mandatory
DT_NEEDED	1	d_val	optional	optional
DT_PLTRELSZ	2	d_val	optional	optional
DT_PLTGOT	3	d_ptr	optional	optional
DT_HASH	4	d_ptr	mandatory	mandatory
DT_STRTAB	5	d_ptr	mandatory	mandatory
DT_SYMTAB	6	d_ptr	mandatory	mandatory
DT_RELA	7	d_ptr	mandatory	optional
DT_RELASZ	8	d_val	mandatory	optional
DT_RELAENT	9	d_val	mandatory	optional
DT_STRSZ	10	d_val	mandatory	mandatory
DT_SYMENT	11	d_val	mandatory	mandatory
DT_INIT	12	d_ptr	optional	optional
DT_FINI	13	d_ptr	optional	optional
DT_SONAME	14	d_val	ignored	optional
DT_RPATH	15	d_val	optional	ignored
DT_SYMBOLIC	16	ignored	ignored	optional
DT_REL	17	d_ptr	mandatory	optional
DT_RELSZ	18	d_val	mandatory	optional
DT_RELENT	19	d_val	mandatory	optional
DT_PLTREL	20	d_val	optional	optional
DT_DEBUG	21	d_ptr	optional	ignored
DT_TEXTREL	22	ignored	optional	optional
DT_JMPREL	23	d_ptr	optional	optional
DT_LOPROC	0x70000000	unspecified	unspecified	unspecified
DT_HIPROC	0x7fffffff	unspecified	unspecified	unspecified

DT\_NULL      identifica la fine dell'array `_DYNAMIC`

DT\_NEEDED    contiene l'offset relativo alla tabella delle stringhe di una stringa con terminatore '\0' che rappresenta il nome di una libreria necessaria. L'offset è un indice della tabella memorizzata in `DT_STRTAB`. Vedi “Dipendenze di oggetti condivisi” per maggiori informazioni. L'array dinamico può contenere più di una voce di questo tipo. Il loro ordine relativo ha significato, mentre non ne ha la loro relazione con le voci di altri tipi.

DT\_PLTRELSZ    contiene la dimensione totale in bytes delle voci di rilocazione associate alla PLT. Se è presente una `DT_JMPREL`, una `DT_PLTRELSZ` deve accompagnarla.

DT\_PLTGOT     contiene un indirizzo associato alla PLT e/o alla GOT. Vedi questo capitolo nella documentazione del processore per i dettagli.

DT_HASH	contiene l'indirizzo della tabella degli hash, descritta in “Tabella degli hash”.La tabella degli hash si riferisce alla tabella dei simboli indicata dall'elemento DT_SYMTAB.
DT_SYMTAB	contiene l'indirizzo della tabella dei simboli, descritta nella Parte 1, con voci Elf32_Sym per i file a 32 bit.
DT_RELA	contiene l'indirizzo di una tabella di rilocazione, descritta nella Parte 1. Le voci hanno addendi espliciti, come Elf32_Rel per i file a 32 bit. Un file oggetto può avere diverse sezioni di rilocazione. Quando costruisce la tavola di rilocazione per un eseguibile o un file di libreria, il linker concatena queste sezioni per formare una sola tabella. Anche se le sezioni rimangono indipendenti nel file oggetto, il loader vede una sola tabella. Quando il loader crea l'immagine di processo per un eseguibile o aggiunge una libreria all'immagine, legge la tabella di rilocazione e compie le azioni connesse. Se questo elemento è presente, la struttura dinamica deve avere anche gli elementi DT_RELASZ e DT_RELAENT. Quando la rilocazione è obbligatoria, devono essere presenti DT_RELA o DT_REL (sono possibili ma non necessari entrambi).
DT_RELASZ	contiene la dimensione totale in bytes della tabella di rilocazione DT_RELA.
DT_RELAENT	contiene la dimensione in bytes di una voce di rilocazione DT_RELA
DT_STRSZ	contiene la dimensione in bytes della tabella delle stringhe
DT_SYMENT	contiene la dimensione in bytes di una voce della tabella dei simboli.
DT_INIT	contiene l'indirizzo della funzione di inizializzazione, discussa nel paragrafo “Funzioni di inizializzazione e terminazione”
DT_FINI	contiene l'indirizzo della funzione di terminazione, discussa nel paragrafo “Funzioni di inizializzazione e terminazione”
DT_SONAME	contiene l'offset nella tabella delle stringhe di una stringa con terminatore '\0' che rappresenta il nome dell'oggetto condiviso. L'offset è un indice nella tabella memorizzata nella voce DT_STRTAB. Vedi “Dipendenze degli oggetti condivisi” per ulteriori informazioni
DT_RPATH	contiene l'offset della tabella delle stringhe relativo ad una stringa con terminatore '\0' che rappresenta il percorso di ricerca delle librerie, discusso in “Dipendenze degli oggetti condivisi”. L'offset è un indice nella tabella riferita dalla voce di DT_STRTAB.
DT_SYMBOLIC	la sua presenza in una libreria condivisa altera l'algoritmo di risoluzione dei simboli del loader per i riferimenti entro le librerie. Invece di iniziare la ricerca di un simbolo nel file eseguibile, il loader inizia dall'oggetto condiviso stesso. Se non lo trova, il loader cerca nel file eseguibile e in altri oggetti condivisi come al solito.
DT_REL	simile a DT_RELA, tranne che la tabella ha addendi impliciti, come Elf32_Rel per i file a 32 bit. Se questo elemento è presente, la struttura dinamica deve avere anche DT_RELSZ e DT_RELENT.
DT_RELSZ	contiene la dimensione totale in byte della tabella di rilocazione DT_REL
DT_RELENT	contiene la dimensione in byte della voce di rilocazione di DT_REL
DT_PLTREL	specifica il tipo di voce di rilocazione a cui si riferisce la PLT. Il campo d_val contiene DT_REL o DT_RELA a seconda del caso. Tutte le rilocazioni in una PLT devono usare lo stesso tipo di rilocazione.

DT_DEBUG	usato per il debugging. Il suo contenuto non è specificato nell' ABI; i programmi che usano questa voce non sono conformi ad ABI.
DT_TEXTREL	l'assenza di questo campo significa che nessuna voce di rilocazione dovrebbe provocare una modifica per un segmento non scrivibile, come indicato dai permessi del segmento nella PHT. Se è presente, una o più voci di rilocazione possono richiedere modifiche a un segmento non scrivibile, e il loader può procedere di conseguenza.
DT_JMPREL	se presente, il membro <code>d_ptr</code> di questo campo contiene l'indirizzo delle voci di rilocazione associate unicamente alla PLT. Queste voci di rilocazione vengono ignorate dal loader durante l'inizializzazione del processo se è abilitato il binding "lazy". Se questa voce è presente, le voci correlate di tipo DT_PLTRELSZ e DT_PLTREL devono essere presenti anch'esse.

Da DT\_LOPROC a DT\_HIPROC i valori nell'intervallo, estremi inclusi, sono riservati per dati processore-specifici.

Eccetto che per l'elemento DT\_NULL, alla fine dell'array, e l'ordine relativo degli elementi DT\_NEEDED, le voci possono comparire in qualsiasi ordine. I tag che non appaiono nella tabella, sono riservati.

## Dipendenze di oggetti condivisi

Quando il linker elabora una libreria a linking statico, ne estrae i componenti e li copia nel file oggetto di output. Queste funzioni sono disponibili durante l'esecuzione senza coinvolgere il loader. Anche le librerie condivise forniscono funzioni, e il loader deve collegare il file condiviso appropriato all'immagine del processo. Perciò eseguibili e librerie condivise descrivono le proprie specifiche dipendenze.

Quando il loader crea i segmenti di memoria per un file oggetto, le dipendenze (registrate nel campo DT\_NEEDED della struttura dinamica) indicano quale oggetto condiviso è necessario per fornire le routine al programma. Connettendo ripetutamente gli oggetti condivisi referenziati con le loro dipendenze, il loader costruisce un'immagine completa del processo. Nel risolvere i riferimenti simbolici, il loader esamina la tabella dei simboli con una ricerca per ampiezza (breadth-first search). Ciò significa che dapprima cerca nella tabella dei simboli dell'eseguibile stesso, poi nelle tabelle dei simboli delle voci DT\_NEEDED (in ordine), poi nelle voci DT\_NEEDED di secondo livello, e così via. I file condivisi devono essere leggibili per il processo; non sono richiesti altri permessi.

Nota: anche se un oggetto condiviso è referenziato più volte nella lista delle dipendenze, il loader lo collega solo una volta al processo.

I nomi nella lista delle dipendenze sono copie o delle stringhe di DT\_SONAME o dei percorsi nel filesystem degli oggetti condivisi usati per costruire il file oggetto. Ad esempio, se il linker usa un oggetto condiviso con una voce DT\_SONAME per lib1 e un altro col percorso `/usr/lib/lib2`, l'eseguibile conterrà lib1 e `/usr/lib/lib2` nella lista delle dipendenze.

Se l'oggetto condiviso ha uno o più slash (/) nel proprio nome, come `/usr/lib/lib2` oppure `dir/file`, il loader usa tali stringhe come percorsi di ricerca. Se il nome non ha slash, come lib1, esistono tre strade per la ricerca, seguite secondo questo ordine di precedenza:

- primo, il tag dell'array dinamico DT\_RPATH può fornire una stringa contenente una lista di directory, separate dal simbolo due punti (:). Ad esempio, la stringa `/home/dir/lib:/home/dir2/lib:` suggerisce al loader di cercare prima in `/home/dir/lib`, poi in `/home/dir2/lib` e infine nella directory corrente .
- secondo, una variabile d'ambiente del processo chiamata LD\_LIBRARY\_PATH può contenere un elenco di directory, seguite o meno dal simbolo punto e virgola (;) e da un secondo elenco. I seguenti valori sono equivalenti a quelli del primo esempio:
  - LD\_LIBRARY\_PATH = `/home/dir/lib:/home/dir2/lib:`

- LD\_LIBRARY\_PATH = /home/dir/lib;/home/dir2/lib:
- LD\_LIBRARY\_PATH = /home/dir/lib:/home/dir2/lib;;

La ricerca in queste directory avviene dopo quella in DT\_PATH. Anche se alcuni programmi, come il linker, trattano in modo diverso gli elenchi prima e dopo il punto e virgola, il loader non lo fa. Comunque il loader accetta la notazione col punto e virgola, con il significato riportato prima.

- infine, se la ricerca nei due gruppi di directory ricordati prima fallisce, il loader cerca in /usr/lib.

Nota: per motivi di sicurezza, il loader ignora le variabili di ricerca ambientali (come LD\_LIBRARY\_PATH) per programmi setuid e setgid. Naturalmente cerca in DT\_PATH e /usr/lib.

## Global offset table

Il codice indipendente dalla posizione (PIC) normalmente non può contenere indirizzi assoluti. La GOT contiene indirizzi assoluti per i dati privati, rendendoli così disponibili senza compromettere l'indipendenza dalla posizione e la condivisibilità del codice. Un programma riferisce la sua GOT usando indirizzi indipendenti dalla posizione, estraendo poi valori assoluti, ridirigendo così i riferimenti indipendenti dalla posizione a locazioni assolute di memoria.

Inizialmente la GOT contiene le informazioni richieste dalle proprie voci di rilocazione (vedi “Rilocazione” nella Parte 1). Dopo che il sistema ha creato i segmenti di memoria per un file oggetto, il loader elabora le voci di rilocazione, alcune delle quali sono del tipo R\_386\_GLOB\_DAT e si riferiscono alla GOT. Il loader identifica il valore dei simboli associati, ne calcola gli indirizzi assoluti e inserisce nelle voci delle tabelle di memoria appropriate il valore opportuno. Anche se gli indirizzi assoluti sono sconosciuti quando il linker costruisce il file oggetto, il loader conosce gli indirizzi di ogni segmento di memoria e può perciò calcolare gli indirizzi assoluti dei simboli in essi contenuti.

Se un programma richiede accesso diretto all'indirizzo assoluto di un simbolo, questo avrà una voce nella GOT. Poiché gli eseguibili e le librerie condivise hanno GOT separate, l'indirizzo di un simbolo può apparire in diverse tabelle. Il loader elabora tutte le rilocazioni nella GOT prima di dare il controllo al codice del processo, assicurando così la disponibilità degli indirizzi assoluti durante l'esecuzione.

La voce di indice zero della tabella è riservata all'indirizzo della struttura dinamica referenziata dal simbolo \_DYNAMIC. Questo permette ad un programma, come il loader, di trovare la propria struttura dinamica prima di aver elaborato le voci di rilocazione. Questo è importante soprattutto per il loader, il quale deve inizializzare sé stesso senza riferirsi ad altri programmi per rilocare la propria immagine di memoria. Nell'architettura Intel a 32 bit anche le voci 1 e 2 nella GOT sono riservate. Il paragrafo “Procedure linkage table” le descrive.

Il sistema può scegliere indirizzi in diversi segmenti di memoria per lo stesso oggetto condiviso in programmi diversi; può scegliere anche diversi indirizzi di librerie per diverse esecuzioni dello stesso programma. Però i segmenti di memoria non cambiano indirizzi una volta formata l'immagine in memoria. Per tutta la durata della vita del processo i suoi segmenti di memoria si trovano a indirizzi virtuali fissi.

Il formato ed il significato della GOT sono specifici per ogni processore. Per l'architettura Intel a 32 bit il simbolo \_GLOBAL\_OFFSET\_TABLE\_ può essere usato per accedere alla tabella.

```
extern Elf32_Addr    _GLOBAL_OFFSET_TABLE_[];
```

Il simbolo \_GLOBAL\_OFFSET\_TABLE\_ può trovarsi nel mezzo della sezione .got, consentendo così riferimenti positivi o negativi degli indirizzi entro l'array stesso.

## Procedure linkage table

Mentre la GOT dirige gli indirizzi indipendenti dalla posizione verso locazioni di memoria assolute, la PLT dirige le chiamate di funzioni indipendenti dalla posizione in locazioni assolute. Il linker non può risolvere i trasferimenti di esecuzione (come le chiamate di funzione) da un eseguibile o da una libreria ad un'altra. Di conseguenza il linker fa in modo di assegnare il controllo del trasferimento del programma alle voci

della PLT. Nell'architettura System V le PLT risiedono nel codice condiviso, ma esse usano indirizzi nella GOT privata. Il loader determina gli indirizzi assoluti delle destinazioni e modifica di conseguenza l'immagine in memoria della GOT. Il loader perciò può ridirigere le voci senza compromettere l'indipendenza dalla posizione e la condivisibilità del codice. Gli eseguibili e i file di libreria hanno PLT separate.

PLT assoluta:

```
.PLT0: pushl   got_plus_4
        jmp     *got_plus_8
        nop
        nop
        nop
        nop
.PLT1: jmp     *name1_in_GOT
        pushl  $offset @PC
.PLT2: jmp     *name2_in_GOT
        push  $offset
        jmp   .PLT0@PC
...
```

PLT indipendente dalla posizione

```
.PLT0: pushl   4(%ebx)
        jmp     *8(%ebx)
        nop
        nop
        nop
        nop
.PLT1: jmp     *name1@GOT(%ebx)
        pushl  $offset
        jmp   .PLT0@PC
.PLT2: jmp     *name2@GOT(%ebx)
        pushl  $offset
        jmp   .PLT0@PC
...
```

nota: come mostrato in figura, le istruzioni della PLT usano indirizzamenti degli operandi diversi per codice assoluto o indipendente dalla posizione. Però le interfacce con il loader sono le stesse.

Seguendo i passi sotto elencati, il loader e il programma cooperano per risolvere i riferimenti simbolici tramite la PLT e la GOT.

1 – quando crea per la prima volta l'immagine in memoria del programma, il loader assegna valori speciali alla seconda e terza voce della GOT. I passi sotto riportati lo illustrano.

2 - se la PLT è indipendente dalla posizione, l'indirizzo della GOT deve trovarsi in %ebx. Ogni oggetto condiviso nell'immagine del processo ha la propria PLT e trasferisce il controllo ad una voce della PLT stessa solo dall'interno dello stesso file oggetto. Di conseguenza, la funzione chiamante è responsabile del settaggio del registro base della GOT prima della chiamata alla PLT.

3 – come esempio, supponiamo che il programma chiami name1, che trasferisce il controllo all'etichetta .PLT1.

4 – la prima istruzione salta all'indirizzo nella GOT per name1. Inizialmente la GOT contiene l'indirizzo della successiva istruzione pushl, non il vero indirizzo di name1.

5 – Di conseguenza, il programma inserisce nello stack un offset di rilocazione (*offset*). Si tratta di un offset a 32 bit non negativo nella tabella delle rilocazioni. La voce di rilocazione designata è del tipo `R_386_JMP_SLOT`, e il suo offset indica la voce della GOT usata nella precedente istruzione `jmp`. La voce di rilocazione contiene anche un indice della tabella dei simboli, indicando così al loader quale simbolo viene referenziato, `name1` in questo caso.

6 – Dopo aver inserito l'offset di rilocazione, il programma salta a `.PLT0`, la prima voce nella PLT. L'istruzione `pushl` inserisce nello stack il valore della seconda voce della GOT (`got_plus_4` o `4(%ebx)`), fornendo così al loader una word di informazione di identificazione. Il programma poi salta all'indirizzo della terza voce della GOT (`got_plus_8` o `8(%ebx)`), che trasferisce il controllo al loader.

7 – Quando il loader riceve il controllo, estrae il valore dallo stack, considera la voce di rilocazione indicata, trova il valore del simbolo, inserisce l'indirizzo reale di `name1` nella GOT e trasferisce il controllo alla destinazione.

8 – La successiva esecuzione della voce nella PLT trasferisce direttamente il controllo a `name1`, senza invocare un'altra volta il loader. Quindi l'istruzione `jmp in .PLT1` salta a `name1`, invece di ritornare all'istruzione `pushl`.

La variabile d'ambiente `LD_BIND_NOW` può cambiare il comportamento del loader. Se il valore è non nullo, il loader valuta le voci della PLT prima di trasferire il controllo al programma. Perciò processa le voci di rilocazione del tipo `R_386_JMP_SLOT` durante il processo di inizializzazione. In caso contrario, il loader valuta la PLT in modalità “lazy”, rinviando la risoluzione dei simboli fino alla prima esecuzione di una voce della tabella.

Nota: La modalità “lazy” migliora la performance dell'applicazione poiché i simboli non usati non aumentano il carico sul loader. Tuttavia due situazioni possono rendere tale modalità non ottimale per alcune applicazioni. Ad esempio, il riferimento iniziale a una funzione condivisa impegna più tempo delle chiamate successive, perché il loader intercetta la chiamata per risolvere il simbolo. Alcune applicazioni non possono permettere questa imprevedibilità. Oppure, se avviene un errore, il loader non può risolvere il simbolo e termina il programma. In modalità “lazy” questo può accadere in un istante qualunque. Ancora una volta, alcune applicazioni non possono accettare questa imprevedibilità. Uscendo dalla modalità “lazy”, l'errore deve avvenire durante l'inizializzazione del processo, prima che l'applicazione riceva il controllo.

## Tabella degli hash

Una tabella hash di oggetti `Elf32_Word` supporta l'accesso alla tabella dei simboli. L'esempio riportato serve alla spiegazione ma non fa parte dello standard.

<code>nbucket</code>
<code>nchain</code>
<code>bucket[0]</code>
....
<code>bucket[nbucket -1]</code>
<code>chain[0]</code>
....
<code>chain[nchain -1]</code>

L'array `bucket` contiene `nbucket` elementi e l'array `chain` ne contiene `nchain`; gli indici iniziano da 0. Entrambi contengono indici della tabella dei simboli. Le voci della tabella `chain` corrispondono a quelle della tabella dei simboli. Il numero di elementi della tabella dei simboli è pari a `nchain`; in questo modo gli indici della tabella dei simboli identificano anche le voci della tabella `chain`. Una funzione di hash, riportata di seguito, accetta il nome di un simbolo e restituisce un valore che può essere usato per calcolare un indice `bucket`. Di conseguenza, se la funzione di hash restituisce un valore `x` per un nome, `bucket[x%nbucket]` dà un

indice y sia per la tabella dei simboli che per la tabella chain. Se la voce della tabella dei simboli non è quella desiderata, chain[y] fornisce la voce successiva della tabella dei simboli che ha lo stesso valore hash. Si possono seguire i link di chain finché la voce della tabella dei simboli selezionata non contiene il nome richiesto oppure l'elemento di chain contiene il valore STN\_UNDEF.

Funzione di hash:

```
unsigned long elf_hash (const unsigned char *name) {
    unsigned long h = 0, g;
    while (*name) {
        h = (h << 4) + *name++;
        if(g = h & 0xf0000000) {
            h ^= g >> 24;
        }
        h &= tilde g;
    }
    return h;
}
```

### **Inizializzazione e terminazione delle funzioni**

Dopo che il loader ha costruito l'immagine del processo e ha eseguito le rilocazioni, ogni oggetto condiviso ha l'opportunità di eseguire un codice di inizializzazione. Queste funzioni di inizializzazione sono chiamate senza un ordine specifico, ma tutte le inizializzazioni degli oggetti condivisi sono effettuate prima che il file eseguibile ottenga in controllo.

Similmente, gli oggetti condivisi hanno funzioni di terminazione eseguite con il meccanismo atexit(BA\_OS) dopo che il processo di base ha iniziato la propria sequenza di terminazione. Anche in questo caso l'ordine di chiamata non è definito.

Gli oggetti condivisi designano le proprie funzioni di inizializzazione e terminazione attraverso le voci DT\_INIT e DT\_FINI nella struttura dinamica, descritta in precedenza nel paragrafo "Sezione dinamica". Di solito il codice per queste funzioni si trova nelle sezioni .ini e .fini, ricordate nel paragrafo "Sezioni" della Parte 1.

Nota: anche se la terminazione atexit(BA\_OS) viene eseguita normalmente, non è garantito che lo sia in caso di morte del processo. In particolare non viene eseguita se il processo chiama \_exit o se termina a causa dell'invio di un segnale che non può essere ignorato o catturato.

## **PARTE 3 LA LIBRERIA C**

**Questa parte non è stata ancora tradotta**

Traduzione a cura di Andrea Cassani

[www.parolamia.eu](http://www.parolamia.eu)

sono graditi commenti, osservazioni, correzioni: [and.cassani@gmail.com](mailto:and.cassani@gmail.com)