Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

a.a. 2012 /2013

Corso di Linux Avanzato
Prof. Marco Cesati
Ing. Emiliano Betti

Implementazione di un meccanismo di binding immediato
indipendente dalle variabili d'ambiente

Studenti:
Claudio Pupparo
Damiano Rossato

INDICE

1. Introduzione

Un qualsiasi programma odierno mediamente complicato fà utilizzo di librerie esterne. Generalmente si preferisce non compilare staticamente tali librerie nell'eseguibile, per questioni di:

- consumo maggiore di risorse nel caso di librerie condivise fra più processi
- efficienza dimensionale, in quanto gli eseguibili risultanti hanno dimensioni maggiori
- flessibilità, ossia per non dover ricompilare tutto l'eseguibile nel caso in cui venisse fornita una nuova versione di una delle librerie

Al linking statico si preferisce quindi il linking dinamico, questo compito è svolto da un componente chiamato **Linker Dinamico**.
Il comportamento di default del linker dinamico è il seguente: durante l'esecuzione, nel momento in cui viene incontrato un simbolo appartenente ad una libreria esterna linkata dinamicamente, questo viene risolto tramite un'apposita procedura; i successivi riferimenti al simbolo non avranno bisogno di ripetere tale procedura, in quanto l'indirizzo dello stesso è già disponibile. Tale meccanismo è definito **Lazy Binding**. Tutti i simboli esterni non utilizzati in fase di esecuzione, rimarranno irrisolti.
In particolare il processo di lazy binding descrive la rilocazione a tempo di esecuzione dell'indirizzo di una funzione di libreria. Questa procedura migliora il tempo di inizializzazione per le applicazioni che fanno uso di una lunga lista di librerie. D'altra parte uno degli svantaggi riguarda l'impossibiiltà di predire il tempo di esecuzione di una certa operazione, in quanto dipenderà dalla presenza oppure assenza in memoria, di un simbolo già risolto.

E' possibile creare una procedura alternativa, che permetta la risoluzione di tutti i simboli esterni utilizzati nel codice prima del loro effettivo utilizzo in fase d'esecuzione, attraverso l'uso della variabile di ambiente ***LD_BIND_NOW***. Quest'ultima se definita, comporta la risoluzione da parte del linker dinamico di tutti i simboli all'avvio del programma, invece di procedere alla risoluzione nel punto del loro primo riferimento.

1.1 Obiettivo

Questo progetto trova una sua applicazione nell'ambito dei **sistemi real time**. Infatti in questi tipi di sistemi il requisito più importante non è rappresentato tanto dalla minimizzazione dei tempi di risposta, quanto dalla **predicibilità del sistema**.
In relazione al linking dinamico, si può notare come tale requisito vada in qualche modo a scontrarsi con la procedura di Lazy Binding; infatti il tempo d'esecuzione nella risoluzione di un simbolo varierà a seconda se il simbolo sia stato già incontrato e quindi risolto, o se invece si tratta del primo utilizzo dello stesso.
La risoluzione immediata di tutti i simboli esterni, garantisce un tempo d'esecuzione uniforme ad

ogni utilizzo degli stessi, garantendo quindi la predicibilità del sistema.

Il progetto svolto consiste quindi in una modifica del linker dinamico, volta a creare un meccanismo di “binding immediato” che sia slegato dalla necessità di specificare la variabile d’ambiente *LD_BIND_NOW*; in particolare, la presenza di una specifica sezione nell’eseguibile indicherà, al linker dinamico, la necessità di eseguire tale procedura di linking.

2. Struttura di un file ELF

Il formato ELF (Execution and Linking Format) è uno standard comune per i file eseguibili, file oggetto e per le librerie dinamiche. Esso fornisce visioni parallele dei contenuti del file stesso, in modo da rispecchiare le necessità nelle diverse fasi di utilizzo dello stesso. La differenza principale riguarda la suddivisione delle informazioni contenute all'interno del file.
Mentre un file oggetto è organizzato in **sezioni**, contenenti informazioni di diversa natura, nel file eseguibile le informazioni sono suddivise in **segmenti**. All'interno dei segmenti ritroviamo varie sezioni, ciascuna con dati omogenei, derivanti dalle sezioni dei file oggetto e delle librerie utilizzate per la creazione del file eseguibile stesso.
In Fig.1 è riportata la struttura generica di un file secondo il formato ELF.

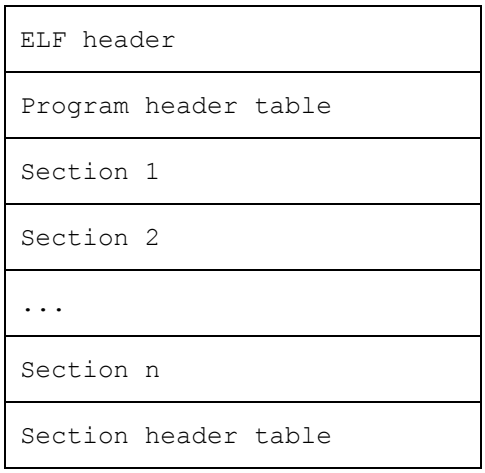

Fig.1 - Struttura file ELF

Possiamo notare la presenza di quattro blocchi principali: l'ELF Header, la Program Header Table (la tabella dei descrittori di segmento), le Sezioni ed infine la Section Header Table (la tabella dei descrittori di sezione). Di seguito verranno approfonditi l'ELF Header e la Section Header Table.
Relativamente alle sezioni non ci sono strutture particolari a cui fare riferimento in quanto quella parte del file che li contiene viene trattata come una mera sequenza di byte; la Program Header Table non verrà approfondita in quanto non è stata utilizzata durante la fase di implementazione.

2.1 ELF Header

L'**ELF header** risiede all'inizio del file e racchiude una "mappa" che descrive l'organizzazione interna del file stesso. Viene implementato dalla seguente struttura:

```
#define EINIDENT 16

typedef  struct {
      unsigned char     eident[EINIDENT];
      Elf32_Half  e_type;
      Elf32_Half  e_machine;
      Elf32_Word  e_version;
      Elf32_Addr  e_entry;
      Elf32_Off   e_phoff;
      Elf32_Off   e_shoff;
      Elf32_Word  e_flags;
      Elf32_Half  e_ehsize;
      Elf32_Half  e_phentsize;
      Elf32_Half  e_phnum;
      Elf32_Half  e_shentsize;
      Elf32_Half  e_shnum;
      Elf32_Half  e_shstrndx;
} Elf3_Ehdr;
```

Le spiegazioni dei campi più significativi per il progetto sono riportate di seguito:

- **e_shoff** contiene l'offset in numero di byte della section header table a partire dall'inizio del file
- **e_shnum** indica quante voci sono contenute nella section header table
- **e_shentsize** esprime la dimensione in byte di un section header (ogni voce ha la stessa dimensione)
- **e_shstrndx** contiene l'indice nella section header table della voce associata con la **Section Name String Table**. Se il file non contiene nessuna section name string table, questo campo contiene il valore *SHN_UNDEF*.

Eseguendo il comando **readelf -h nome_eseguibile** è possibile visualizzare il contenuto dell'intestazone ELF di un file eseguibile, in un formato comprensibile all'utente. In Fig.2 viene mostrato un esempio

```
Intestazione ELF:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Classe:                            ELF64
  Dati:                              complemento a 2, little endian
  Versione:                          1 (current)
  SO/ABI:                            UNIX - System V
  Versione ABI:                      0
  Tipo:                              EXEC (file eseguibile)
  Macchina:                          Advanced Micro Devices X86-64
  Versione:                          0x1
```

```
Indirizzo punto d'ingresso:         0x4004c0
Inizio intestazioni di programma    64 (byte nel file)
Inizio intestazioni di sezione:     4480 (byte nel file)
Flag:                               0x0
Dimensione di questa intestazione: 64 (byte)
Dimens. intestazioni di programma:  56 (byte)
Numero intestazioni di programma:  9
Dimens. intestazioni di sezione:   64 (byte)
Numero di intestazioni di sezione: 29
Indice della tabella di stringhe delle intestazioni di sezione: 26
```

Fig.2 - Output readelf -h nome_eseguibile

2.2 Section Header Table

La **Section Header Table** è un array di strutture Elf32_Shdr (in un'architettura a 32 bit). Ogni voce quindi, è un **section header**, cioè una struttura che contiene informazioni riguardanti una delle sezioni contenute nel file oggetto. Questa tabella permette di localizzare tutte le sezioni all'interno del file. Di seguito viene mostrata la struttura di un section header:

```
typedef struct {
      Elf32_Word  sh_name;
      Elf32_Word  sh_type;
      Elf32_Word  sh_flags;
      Elf32_Addr  sh_addr;
      Elf32_Off   sh_offset;
      Elf32_Word  sh_size;
      Elf32_Word  sh_link;
      Elf32_Word  sh_info;
      Elf32_Word  sh_addralign;
      Elf32_Word  sh_entsize;
} Elf32_Shdr;
```

I campi contraddistinti da una sottolineatura, sono quelli che sono stati utilizzati in fase di implementazione e sono spiegati di seguito:

- **sh_name** questo campo specifica il nome della sezione. In particolare è un indice all'interno della **section header string table**, che restituisce la posizione di una stringa zero terminata.
- **sh_size** contiene la dimensione della sezione in byte. A meno che le sezione non sia del tipo *SHT_NOBITS*, la sezione occupa *sh_size* byte nel file. Una sezione di tipo *SHT_NOBITS* potrebbe avere una dimensione diversa da zero, ma comunque non occuperebbe spazio all'interno del file

- **sh_offset** indica il numero dei byte dello spiazzamento dall'inizio del file al primo byte nella sezione.

Il comando **readelf -S nome_eseguibile** mostra invece le informazioni contenute nelle intestazioni di sezione del file ELF. In Fig.3 è riportato un esempio:

```
Ci sono 29 intestazioni di sezione, iniziando dall'offset 0x1180:

Intestazioni di sezione:
  [N°] Nome              Tipo             Indirizzo          Offset
       Dimensione        DimEnt           Flag   Link  Info  Allin
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000         0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A      0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A      0     0     4
  [ 3] .hash             HASH             0000000000400278  00000278
       000000000000002c  0000000000000004   A      4     0     8
  [ 4] .dynsym           DYNSYM           00000000004002a8  000002a8
       0000000000000090  0000000000000018   A      5     1     8
  [ 5] .dynstr           STRTAB           0000000000400338  00000338
       000000000000004b  0000000000000000   A      0     0     1
  [ 6] .gnu.version      VERSYM           0000000000400384  00000384
       000000000000000c  0000000000000002   A      4     0     2
  [ 7] .gnu.version_r    VERNEED          0000000000400390  00000390
       0000000000000020  0000000000000000   A      5     1     8
  [ 8] .rela.dyn         RELA             00000000004003b0  000003b0
       0000000000000018  0000000000000018   A      4     0     8
  [ 9] .rela.plt         RELA             00000000004003c8  000003c8
       0000000000000078  0000000000000018   A      4     11    8
  [10] .init             PROGBITS         0000000000400440  00000440
       000000000000001a  0000000000000000  AX      0     0     4
  [11] .plt              PROGBITS         0000000000400460  00000460
       0000000000000060  0000000000000010  AX      0     0     16
  [12] .text             PROGBITS         00000000004004c0  000004c0
       00000000000001d4  0000000000000000  AX      0     0     16
  [13] .fini             PROGBITS         0000000000400694  00000694
       0000000000000009  0000000000000000  AX      0     0     4
  [14] .rodata           PROGBITS         00000000004006a0  000006a0
       000000000000000e  0000000000000000   A      0     0     4
  [15] .eh_frame_hdr     PROGBITS         00000000004006b0  000006b0
       0000000000000034  0000000000000000   A      0     0     4
  [16] .eh_frame         PROGBITS         00000000004006e8  000006e8
       00000000000000d4  0000000000000000   A      0     0     8
  [17] .init_array       INIT_ARRAY       0000000000600e10  00000e10
       0000000000000008  0000000000000000  WA      0     0     8
  [18] .fini_array       FINI_ARRAY       0000000000600e18  00000e18
       0000000000000008  0000000000000000  WA      0     0     8
  [19] .jcr              PROGBITS         0000000000600e20  00000e20
       0000000000000008  0000000000000000  WA      0     0     8
```

```
  [20] .dynamic          DYNAMIC            0000000000600e28  00000e28
       00000000000001d0  0000000000000010  WA       5     0     8
  [21] .got              PROGBITS           0000000000600ff8  00000ff8
       0000000000000008  0000000000000008  WA       0     0     8
  [22] .got.plt          PROGBITS           0000000000601000  00001000
       0000000000000040  0000000000000008  WA       0     0     8
  [23] .data             PROGBITS           0000000000601040  00001040
       0000000000000010  0000000000000000  WA       0     0     8
  [24] .bss              NOBITS             0000000000601050  00001050
       0000000000000008  0000000000000000  WA       0     0     4
  [25] .comment          PROGBITS           0000000000000000  00001050
       000000000000003d  0000000000000001  MS       0     0     1
  [26] .shstrtab         STRTAB             0000000000000000  0000108d
       00000000000000f1  0000000000000000           0     0     1
  [27] .symtab           SYMTAB             0000000000000000  000018c0
       0000000000000630  0000000000000018          28    44     8
  [28] .strtab           STRTAB             0000000000000000  00001ef0
       000000000000025e  0000000000000000           0     0     1

Legenda dei flag:
  W (scrittura), A (allocazione), X (esecuzione), M (unione), S (stringhe),
l (grande)
  I (informazioni), L (ordine link), G (gruppo), T (TLS), E (esclusione), x
(sconosciuto)
  O (richiesta elaborazione aggiuntiva SO) o (specifico del SO), p
(specifico del processore)
```

Fig.3 - Output readelf -S nome_eseguibile

2.3 Section Name String Table

La **Section Name String Table** è una particolare sezione contenente l'elenco dei nomi di tutte le sezioni del file ELF, sotto forma di stringhe zero terminate. In Fig.4 è mostrata la struttura di una generica String Table

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

Fig. 4 - String Table

Il campo *sh_name* di ogni sezione è utilizzato come indice all'interno della tabella per identificare una particolare entry, come in Fig. 5

Index	String
0	*none*
1	name.
7	Variable
11	able
16	able
24	*null string*

Fig.5 - Indici di una String Table

Tramite il comando **readelf -h nome_eseguibile** è possibile ottenere l'indice della Section Name String Table (*Indice della tabella di stringhe delle intestazioni di sezione* in Fig.2). A questo punto è possibile visionare la tabella tramite il comando **readelf -p index**, di cui un esempio di output è mostrato in Fig.6:

```
Dump delle stringhe della sezione ".shstrtab":
  [     1]  .symtab
  [     9]  .strtab
  [    11]  .shstrtab
  [    1b]  .interp
  [    23]  .note.ABI-tag
  [    31]  .hash
  [    37]  .dynsym
  [    3f]  .dynstr
  [    47]  .gnu.version
  [    54]  .gnu.version_r
  [    63]  .rela.dyn
  [    6d]  .rela.plt
  [    77]  .init
  [    7d]  .text
  [    83]  .fini
  [    89]  .rodata
  [    91]  .eh_frame_hdr
  [    9f]  .eh_frame
  [    a9]  .init_array
  [    b5]  .fini_array
  [    c1]  .jcr
  [    c6]  .dynamic
  [    cf]  .got
  [    d4]  .got.plt
  [    dd]  .data
  [    e3]  .bss
  [    e8]  .comment
  [    f1]  .bind_now
```

Fig.6 - readelf -p index

3. Implementazione

Per realizzare quanto richiesto è stato modificato il linker dinamico **ld.so**. Questo programma è contenuto nella libreria **glibc** (in particolare si è lavorato con la versione 2.18), e il codice sorgente si trova nel file *glibc-2.18/elf/rtld.c.*
In particolare l'algoritmo ricerca nel file eseguibile una specifica sezione di nome "**.bind_now**"; se presente viene disabilitato il lazy binding, ed i simboli esterni vengono risolti all'avvio del programma, anzichè quando vengono referenziati. Il codice aggiunto è contenuto interamente nella nuova funzione ***find_now_section***. Tale funzione viene richiamata subito dopo ***process_envvars*** che controlla quali variabili d'ambiente legate al linking sono state definite nel sistema.
L'effetto della soluzione realizzata è lo stesso che si avrebbe abilitando la variabile d'ambiente *LD_BIND_NOW*, come già precedentemente descritto. Lo pseucodice dell'algoritmo è mostrato in Fig.7

```
BIND_NOW_SECTION = ".bind_now"
elf_header = map_file_to_memory()
section_header_table = elf_header + elf_header.e_shoff
section_name_str_tbl_header = section_header_table[elf_header.e_shstrndx]
IF exists(section_name_str_tbl_header) THEN
      section_name_str_tbl=elf_header+section_name_str_tbl_header.sh_offset
      IF section_name_str_tbl.sh_size > 0 THEN
            k = 0
            WHILE k < elf_header.e_shnum
                  section_header = section_header_table[k]
                  section_name_index = section_header.sh_name
                  section_name = section_name_str_table[section_name_index]
                  IF section_name == BIND_NOW_SECTION THEN
                        dl_lazy = 0
                        BREAK
                  ENDIF
                  k++
            ENDWHILE
      ENDIF
ENDIF
```

Fig.7 - Pseudocodice algoritmo implementato

Il primo passo consiste nell'effettuare il mapping del file eseguibile in memoria tramite la funzione ***mmap***.

```
void *mmap(void *addr, size_t len, int prot, int flags,
                int fildes, off_t off);
```

I parametri del mapping sono:

- ***addr***: Indirizzo in cui creare l'area di memoria. Valore zero, in quanto si lascia scegliere al kernel
- ***len***: lunghezza del mapping. Valore pari alla dimensione del file, in quanto si vuole mappare il file in memoria nella sua interezza. La dimensione del file è ottenuta tramite la funzione ***stat***, che restituisce una struttura con gli attributi del file.
- ***prot***: livello di protezione. Valore pari a ***PROT_READ***, ossia l'area di memoria è accedibile in sola lettura.
- ***flags***: ***MAP_PRIVATE***, in quanto le modifiche all'area di memoria non devono essere propagate sul file.
- ***fildes***: il file descriptor è ottenuto tramite la funzione ***open*** eseguita sul file ***/proc/self/exe***, contenente il percorso dell'eseguibile corrente.
- ***off***: offset all'interno del file a partire dal quale eseguire il mapping. Poichè si vuole mappare il file intero, tale campo è posto a zero.

L'indirizzo restituito da *mmap*, ossia l'inizio dell'area di memoria con il contenuto del file, corrisponde alla posizione del primo byte dell'ELF Header, in quanto come già detto tale struttura si trova sempre all'inizio di un file ELF.
Come già accennato, le principali strutture di cui si è fatto uso all'interno dell'algoritmo sono:

- Section Header Table: tabella con i descrittori di tutte le sezioni
- Section Name String Table: sezione contenente i nomi di tutte le sezioni del file ELF

La Section Header Table è ottenuta sommando all'indirizzo dell'ELF Header, l'offset *e_shoff* (un campo dell'ELF Header).

Per trovare l'indirizzo della Section Name String Table è necessario effettuare una serie di passi:

- Verificare che la tabella sia esistente (ossia che il campo *e_shstrndx* dell'ELF Header sia diverso da *SHN_UNDEF*)
- Accedere alla entry *e_shstrndx* all'interno della Section Header Table ottenendo la posizione del descrittore della Section Name String Table.
- Verificare che la dimensione della sezione sia maggiore di zero (controllando il campo *sh_size*)
- Sommare all'indirizzo dell'ELF Header l'offset *sh_offset* (un campo del descrittore della tabella) ottenendo così la posizione della tabella

Una volta ottenuta la posizione delle due strutture sopracitate, è possibile procedere con la ricerca della nuova sezione.

In particolare viene eseguito un ciclo fra tutti i descrittori di sezione (le voci della Section Header Table); per ogni voce si prende il relativo campo *sh_name*, un indice all'interno della Section Name String Table che punta all'entry contenente il nome della sezione in esame; viene quindi confrontata la stringa "*.bind_now*" con il nome della sezione correntemente scansionata. Se si trova una corrispondenza, e quindi la sezione *.bind_now* è presente nel file ELF, viene **disabilitato il lazy_binding** ponendo a zero la variable **dl_lazy**, acceduta tramite la macro ***GLRO(dl_lazy)***.

Una volta modificato il linker dinamico è necessario "istruire" l'eseguibile ad utilizzare la versione modificata. Il formato ELF prevede una sezione speciale ***.interp*** contenente il path all'interprete (il linker dinamico), visionabile tramite il comando:

```
objdump -sj .interp /path/executable
```

Generalmente tale path corrisponde a:

- */lib/ld-linux.so.2* in un'architettura a 32 bit
- */lib64/ld-linux-x86-64.so.2* in un'architettura a 64 bit

Per modificare la sezione è stato utilizzato il programma **patchelf** lanciando il seguente comando:

```
patchelf --set-interpreter /path/linker /path/executable
```

Prima di lanciare l'eseguibile è necessario aggiungere la nuova sezione *.bind_now* mediante il comando **objcopy**:

```
objcopy --add-section .section_name=/path/file/name /path/executable
```

Il contenuto della nuova sezione viene preso dal file */path/file/name*. Poichè in questo caso l'importante è che la sezione sia presente a prescindere dal contenuto, è possibile specificare un file vuoto. In Fig.8 è evidenziata la sezione *.bind_now* appena aggiunta ad un eseguibile.

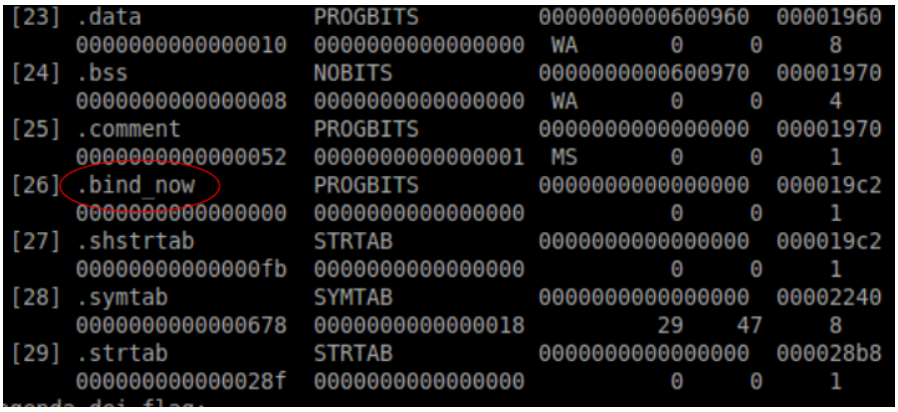

Fig.8 - Sezione *.bind_now*

3.1 Debug

Durante l'implementazione dell'algoritmo, è stato necessario effettuare il debug del codice. Lo strumento utilizzato è il **GNU Project Debugger** (**gdb**), un tool da riga di comando.
Nel file **config.make** della libreria glibc (*glibc-2.18/build/config.make*) possiamo trovare l'elenco delle opzioni di compilazione. Di base è compilata con ottimizzazioni di secondo livello (opzione -O2)

```
...
# Build tools.
CC = gcc
CXX = g++
BUILD_CC =
CFLAGS = -g -O2
...
```

Fig.9 - Ottimizzazioni glibc

il che significa che le istruzioni possono essere riordinate dal compilatore per utilizzare in maniera più efficiente i registri. Chiaramente le ottimizzazioni rendono più complicata la fase di debug, in quanto il flusso di esecuzione non segue l'ordine esatto delle istruzioni presenti nel codice così come sono state scritte.
Non essendo possibile disattivare le ottimizzazioni dal file di configurazione (la libreria interrompe il processo di compilazione con un messaggio di errore) si è scelto di utilizzare la parola chiave **volatile** per ogni nuova variabile definita nella funzione. Tale keyword previene appunto l'applicazione di ottimizzazioni sulla variabile.
Una volta appurata la correttezza del codice, ogni occorrenza della keyword è stata rimossa dalla funzione realizzata.

3.2 Test

Sono stati eseguiti dei test, per verificare la correttezza della soluzione realizzata.
La variabile d'ambiente **LD_DEBUG** fornisce un ottimo strumento di debug per ottenere informazioni sul processo di linking dinamico. Lanciando un qualsiasi comando con **LD_DEBUG=help** si ottiene la seguente lista di opzioni, tra cui **statistics** è risultata quella più utile.

```
libs        display library search paths
reloc       display relocation processing
files       display progress for input file
symbols     display symbol table processing
bindings    display information about symbol binding
```

```
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit
```

Fig.10 - LD_DEBUG

Senza avere eseguito il patching di un eseguibile con la versione modificata del linker dinamico, si ottengono i risultati in Fig.11 e Fig.12:

Lazy Binding

Comando: `LD_DEBUG=statistics /path/executable`

Output:

```
     21165:                   number of relocations: 84
     21165:        number of relocations from cache: 5
     21165:          number of relative relocations: 1264
<Program Output>
     21165:
     21165:  runtime linker statistics:
     21165:             final number of relocations: 89
     21165:  final number of relocations from cache: 5
```

Fig.11 - Statistiche Lazy Binding

Bind Now

Comando: `LD_BIND_NOW=1 LD_DEBUG=statistics /path/executable`

Output:

```
     21166:                   number of relocations: 96
     21166:        number of relocations from cache: 5
     21166:          number of relative relocations: 1264
<Program Output>
     21166:
     21166:  runtime linker statistics:
     21166:             final number of relocations: 96
     21166:  final number of relocations from cache: 5
```

Fig. 12 - Statistiche Bind Now

Da notare come il numero finale di rilocazioni sia differente, in quanto nel primo test effettuando il lazy binding i simboli non utilizzati rimangono irrisolti, a differenza del secondo caso in cui tutti i simboli vengono risolto all'avvio dell'eseguibile.

In Fig.13 è riportato uno screenshot dell'esecuzione di un programma in cui è stata utilizzata la versione modificata del linker dinamico

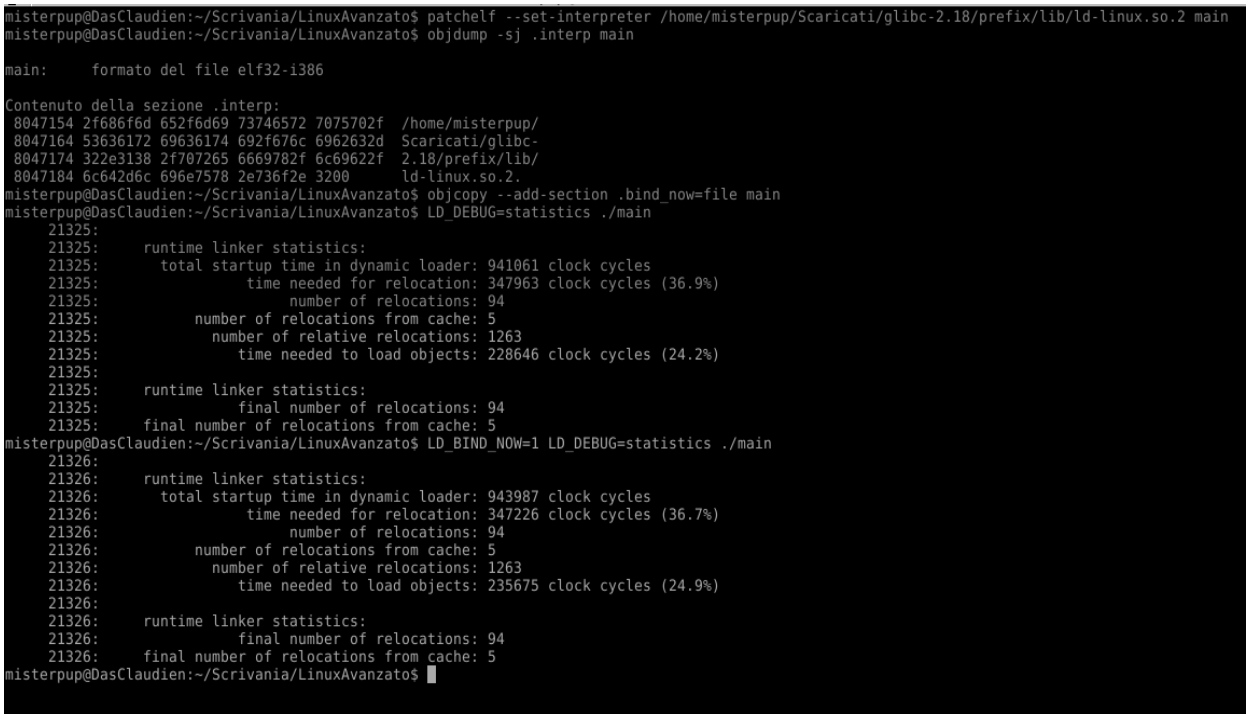

Fig.13 - Esecuzione con versione modificata del linker dinamico

Si può notare come a seguito dell'aggiunta della sezione *.bind_now* nell'eseguibile, il numero di rilocazioni finali sia lo stesso nelle due esecuzioni, indipendetemente dall'utilizzo o meno della variabile d'ambiente *LD_BIND_NOW*.

Conclusioni

Come da requisito è stato implementato un meccanismo di binding immediato, che permette la risoluzione di tutti i simboli esterni all'avvio di un file eseguibile, prima dell'esecuzione della prima istruzione. Questa modalità di binding, prescinde dalla definizione della variabile di ambiente LD_BIND_NOW. L'algoritmo sviluppato determina il suo comportamento in base alla presenza o meno di un'apposita sezione all'interno dell'eseguibile. I test hanno dimostrato che per i programmi che rispettano quest'ultimo requisito, la risoluzione dei simboli avviene sempre all'avvio del programma stesso.

Bibliografia

Cesati M., Bovet D. P., *Linkaggio di Programmi*, Marzo 2011
Executable and Linkable Format (ELF) - Tool Interface Standards (TIS)
GDB: The GNU Projects Debugger, disponibile all'indirizzo: http://www.sourceware.org/gdb/
The GNU C Library, disponibile all'indirizzo: http://www.gnu.org/software/libc/
PatchELF, disponibile all'indirizzo: https://nixos.org/patchelf.html