O objektnim datotekama, linkerima i loaderima
©2004 Željko Vrba

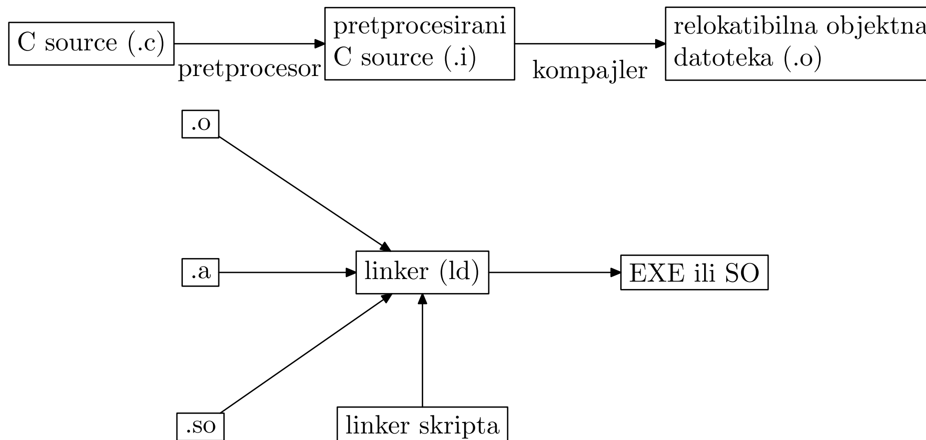
Tutorijali i knjige o programiranju se često koncentriraju samo na programski jezik, no često zanemaruju *interakciju programa i okoline*, tj. operacijskog sustava. Koraci kod kompajliranja te generiranja finalnog izvršnog programa su šturo, ako i uopće, dokumentirani.

Ovaj članak objašnjava takve "nedokumentirane" stvari. Prvi dio članka opisuje faze kompajliranja i primjenjiv je na bilo koju implementaciju C i C++ jezika. Pri opisu najnižih mehanizama članak daje primjere na UNIX sustavima koji koriste ELF object file format.[1]

1 Kompajliranje 1
1.1 Pretprocesor 2
1.2 Kompajler 2
1.3 Linker 5
2 Debug informacije 10
3 Učitavanje programa 10
3.1 Statičko i mehanizam dinamičkog linkanja 12
3.2 Parametri koji utječu na dinamičko linkanje 14
3.3 Implicitno i eksplicitno dinamičko linkanje 14
3.4 Razlike između ELF i COFF 15
4 Literatura 16

1 Kompajliranje

Koraci pri kompajliranju jedne source datoteke te linkanje objektnih datoteka u izvršni program prikazani su na **slici 1**.



Slika 1 Kompajliranje

[1] U nastavku će se pisati samo ELF.

1.1 Pretprocesor

C pretprocesor obrađuje *pretprocesorske direktive*: `#include, #if, #ifdef, #ifndef, #endif, #define`, aritmetičke izraze u direktivama itd. Rezultat pretprocesiranja je C source sa potpuno izvedenom tekstualnom supstitucijom.

Na primjer, rezultat pretprocesiranja sljedećeg jednostavnog koda

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

#define SQR(x) ((x)*(x))

int main(int argc, char **argv)
{
  printf("%d\n", SQR(atoi(argv[1])));
}
```

je

```
/*
  ovdje pretprocesor umetne KOMPLETAN sadrzaj sistemskih
  includeova, takoder rekurzivno procesirajuci pretprocesorske
  direktive u njima.
*/
int main(int argc, char **argv)
{
  printf("%d\n", ((atoi(argv[1]))*(atoi(argv[1]))));
}
```

Ovdje se odmah vidi i *nedostatak* `#define` makroa u odnosu na funkcije: funkcija `atoi` se poziva *dvaput* i kad bi imala "side-efekte", oni bi bili dvaput izvršeni što često nije poželjno. Zato je konvencija da se makroi pišu svim velikim slovima kako bi se mogli razlikovati od funkcija (gdje se svaki argument izračunava *točno jednom nespecificiranim redoslijedom.*).

Napomena: iako počinje znakom `#`, `#pragma` je direktiva *kompajleru*, a ne pretprocesoru.

Kod nekih kompajlera je pretprocesiranje integrirano u sam kompajler, a kod nekih je to poseban program (npr. kod gcc-a). Na gcc-u je moguće vidjeti pretprocesirani program (onako kako ga vidi kompajler) izvođenjem naredbe `gcc -E prog.c`

1.2 Kompajler

Kompajler prevodi pretprocesirani source u **relokatibilni objektni kod**. Na UNIX-u rezultat kompajliranja ima tradicionalno ekstenziju `.o`, a na Win32 `.obj`. Svaki operacijski sustav ima svoj standardni format objektnog koda. UNIX koristi a.out, COFF ili ELF, dok se na Windowsima koristi COFF uz Microsoftove ekstenzije. ELF je najfleksibilniji format, pogotovo kad se koriste shareani libraryji (DLL u Win32 terminologiji, a u nastavku SO, od shared object).

Objektni kod se sastoji od više **section**-a. Section se najlakše može opisati kao skup podataka koji čine jednu logičku cjelinu i unutar objektne datoteke zauzimaju uzastopni niz bajtova. Svaki section ima svoje **ime** i skup dodatnih **atributa** koji su upute linkeru što učiniti s podacima i kako generirati finalnu izvršnu datoteku.

Za primjer će se koristiti sljedeći kod:[2]

```
#include <stdio.h>

int a, b = 16;
static char *msg = "Hello, world!";
static const int x = 13;
static int c;

static void print(const char *m)
{
  static int a = 5, b;
  printf("print: %d %d %s\n", a++, b, m);
}

void init(void)
{
  print(msg);
  printf("Init gotov, a=%d, b=%d, x=%d, c=%d\n");
}

int main(int argc, char **argv)
{
  init();
  print(argv[0]);
}
```

Na UNIX-u se naredbom `objdump` (dio paketa GNU binutils) može pogledati sadržaj objektnog fajla (u nastavku teksta $ označava shell prompt). Kompajliranjem prethodnog koda dobije se objektna datoteka koja se može analizirati:

```
Script started on Sun 22 Aug 2004 11:08:46 AM CEST
$ gcc -c -g p1.c
$ objdump -h p1.o

p1.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000007a  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         0000000c  00000000  00000000  000000b0  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, DATA
  2 .bss          00000008  00000000  00000000  000000bc  2**2
                  ALLOC
  3 .debug_abbrev 000001b9  00000000  00000000  000000bc  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_info   00001242  00000000  00000000  00000275  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  5 .debug_line   000000eb  00000000  00000000  000014b7  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  6 .rodata       00000064  00000000  00000000  000015c0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

[2] Dakako, bezveze je (na engleskom se to zove “contrived example”), ali ilustrira bitne koncepte objektnih datoteka.

```
  7 .debug_frame  00000068  00000000  00000000  00001624  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_pubnames 00000030  00000000  00000000  0000168c  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_aranges 00000020  00000000  00000000  000016bc  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
 10 .debug_str    00000025  00000000  00000000  000016dc  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .note.GNU-stack 00000000  00000000  00000000  00001701  2**0
                  CONTENTS, READONLY
 12 .comment      00000012  00000000  00000000  00001701  2**0
                  CONTENTS, READONLY
$ readelf -S p1.o
There are 24 section headers, starting at offset 0x17dc:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00007a 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 001d70 000048 08     22   1  4
  [ 3] .data             PROGBITS        00000000 0000b0 00000c 00  WA  0   0  4
  [ 4] .rel.data         REL             00000000 001db8 000008 08     22   3  4
  [ 5] .bss              NOBITS          00000000 0000bc 000008 00  WA  0   0  4
  [ 6] .debug_abbrev     PROGBITS        00000000 0000bc 0001b9 00      0   0  1
  [ 7] .debug_info       PROGBITS        00000000 000275 001242 00      0   0  1
  [ 8] .rel.debug_info   REL             00000000 001dc0 0000e0 08     22   7  4
  [ 9] .debug_line       PROGBITS        00000000 0014b7 0000eb 00      0   0  1
  [10] .rel.debug_line   REL             00000000 001ea0 000008 08     22   9  4
  [11] .rodata           PROGBITS        00000000 0015c0 000064 00   A  0   0 32
  [12] .debug_frame      PROGBITS        00000000 001624 000068 00      0   0  4
  [13] .rel.debug_frame  REL             00000000 001ea8 000030 08     22   c  4
  [14] .debug_pubnames   PROGBITS        00000000 00168c 000030 00      0   0  1
  [15] .rel.debug_pubnam REL             00000000 001ed8 000008 08     22   e  4
  [16] .debug_aranges    PROGBITS        00000000 0016bc 000020 00      0   0  1
  [17] .rel.debug_arange REL             00000000 001ee0 000010 08     22  10  4
  [18] .debug_str        PROGBITS        00000000 0016dc 000025 00      0   0  1
  [19] .note.GNU-stack   PROGBITS        00000000 001701 000000 00      0   0  1
  [20] .comment          PROGBITS        00000000 001701 000012 00      0   0  1
  [21] .shstrtab         STRTAB          00000000 001713 0000c6 00      0   0  1
  [22] .symtab           SYMTAB          00000000 001b9c 0001a0 10     23  15  4
  [23] .strtab           STRTAB          00000000 001d3c 000031 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
zvrba@zax:~/work/coredump/linkers/code$
Script done on Sun 22 Aug 2004 11:09:15 AM CEST
```

`objdump` izostavlja neke sectione jer je pisan kao univerzalni program koji može raditi sa mnogim objektnim formatima. Za razliku od njega, `readelf` interpretira isključivo ELF objektne datoteke te daje potpunije informacije.

Objektni kod sadrži, između ostaloga, sljedeće podatke:

Izvršni kod koji je generirao kompajler iz sourcea i koji procesor može izvršavati (ali tek nakon linkanja). Obično se nalazi u `.text` sectionu.

Podaci koji mogu biti **inicijalizirani**, **neinicijalizirani** i **common**. Inicijalizirani podaci idu u `.data` section, a neinicijalizirani i common podaci su rezervirani `.bss` sectionom. Read-only podaci (npr. literal stringovi i `const` varijable) idu u `.rodata` section.

Simboli. Svako simboličko ime u programu (ime varijable ili funkcije) mora biti povezano sa nekom **adresom**. Simboli su spremnljeni u `.symtab`, `.shstrtab` i `.strtab` sectionima. Zadnja dva sectiona sadrže isključivo tablicu *imena* simbola, a dodatni atributi simbola upisani su u `.symtab` sectionu koji se na ime referencira preko indeksa u tablici imena.

Izuzetak su automatske varijable (lokalne varijable funkcije čiji je storage `auto`[3]): njihove simbole u adrese "resolva" kompajler prilikom kompajliranja. One su redovito adresirane relativno pomoću stack pointera.[4]

Relokacije. Informacije o relokacijama (koje nisu uvijek vezane za simbole; relokacije su na nivou pojedinačnih procesorskih instrukcija i debug informacija) služe kao uputa u finalnoj fazi linkanja kako relativnu adresu pretvoriti u apsolutnu da bi se program mogao izvršavati. Relokacije su u `.rel*` sectionima.

Naime, prilikom kompajliranja, za sve varijable unutar jednog sourcea se alocira prostor, ali počevši od adrese 0. Linker za sve te varijable alocira novi prostor te uz pomoć relokacija "popravlja" pojedinačne instrukcije koda pretvarajući pogrešne adrese u ispravne.

Debug informacije. U `.debug*` sectione se spremaju debug informacije za source debugging programa. Ovi sectioni su specifični za UNIX sa DWARF2 ili DWARF3 debugging formatom.

Imena sectiona koja su ovdje navedena specifična su za Linux/ELF, ali principi su primjenjivi i na druge UNIX operacijske sustave, a dijelom i na Win32.

1.3 Linker

Linker povezuje relokatibilne objektne datoteke (`.o`), statičke (`.a`) i dinamičke libraryje (SO, `.so`) u izvršnu datoteku ili SO (**slika 1**). Proces linkanja kontrolira **linker skripta**. Detaljno o linker skriptama može se naći u npr. `info ld` na linuxu. Na linuxu se linker skripte tradicionalno nalaze u `ldscripts` direktoriju. Konkretna lokacija ovisi o distribuciji, npr. na Slackwareu 10 se nalaze u `/usr/lib/ldscripts`.

Jedan od ključnih pojmova pri linkanju je **simbol**: to je ime povezano sa adresom. Tablica simbola se može pogledati na sljedeći način:

```
Script started on Sun 22 Aug 2004 11:10:23 AM CEST
$ readelf -s p1.o

Symbol table '.symtab' contains 26 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS p1.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    5
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    9
     8: 00000000     0 SECTION LOCAL  DEFAULT   11
     9: 00000004     4 OBJECT  LOCAL  DEFAULT    3 msg
    10: 00000010     4 OBJECT  LOCAL  DEFAULT   11 x
    11: 00000008     4 OBJECT  LOCAL  DEFAULT    3 a.0
```

[3] Ovo je C ključna riječ koja se više nizašto ne koristi – sve lokalne varijable su po defaultu `auto` ako se ne deklariraju kao `static` ili `extern`.

[4] Iako, u nekim implementacijama, stack *uopće ne mora postojati*.

```
    12: 00000000     4 OBJECT  LOCAL  DEFAULT    5 b.1
    13: 00000000    42 FUNC    LOCAL  DEFAULT    1 print
    14: 00000004     4 OBJECT  LOCAL  DEFAULT    5 c
    15: 00000000     0 SECTION LOCAL  DEFAULT   12
    16: 00000000     0 SECTION LOCAL  DEFAULT   14
    17: 00000000     0 SECTION LOCAL  DEFAULT   16
    18: 00000000     0 SECTION LOCAL  DEFAULT   18
    19: 00000000     0 SECTION LOCAL  DEFAULT   19
    20: 00000000     0 SECTION LOCAL  DEFAULT   20
    21: 00000000     4 OBJECT  GLOBAL DEFAULT    3 b
    22: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    23: 0000002a    41 FUNC    GLOBAL DEFAULT    1 init
    24: 00000053    39 FUNC    GLOBAL DEFAULT    1 main
    25: 00000004     4 OBJECT  GLOBAL DEFAULT  COM a
zvrba@zax:~/work/coredump/linkers/code$
Script done on Sun 22 Aug 2004 11:10:27 AM CEST
```

Značenje atributa:

Value je "vrijednost" simbola. Interpretacija ovog atributa ovisi o kontekstu:

- Kod relokatibilnih datoteka: ovo je potreban alignment common simbola, odn. offset od početka sectiona (u toj datoteci) čiji je indeks naveden.
- Kod izvršnih datoteka i SO-ova ovo polje određuje virtualnu adresu simbola.

Size je veličina objekta s kojom je simbol povezan ili 0 ako ne zauzima prostor ili je veličina nepoznata. Npr. `main` funkcija je dugačka 39 bajtova, a varijabla `b` zauzima 4 bajta.

Type je tip objekta s kojim je simbol povezan. Najvažniji tipovi su `OBJECT` (podatak) ili `FUNCTION` (kod).

Bind može biti `LOCAL`, `GLOBAL` ili `WEAK`.

- **LOCAL** simboli su vidljivi samo u svojoj datoteci. To se postiže u C-u sa `static` deklaracijom funkcije ili varijable. Dvije različite source datoteke mogu definirati lokalni simbol istog imena, ali će pristupati različitim objektima.
- **GLOBAL** simboli su vidljivi u svim relokatibilnim datotekama koje se linkaju. Globalni simbol u jednoj datoteci će resolvati nedefiniranu referencu u nekoj drugoj datoteci. Ako se isti globalni simbol definira na dva mjesta (u dvije različite datoteke), linker *prijavljuje grešku* o višestruko definiranom simbolu.
- **WEAK** simbol je kao globalni simbol, ali ima niži prioritet: ako postoji "non-weak" definicija istog simbola, uzet će se ta definicija umjesto weak definicije.

Vis odn. visibility je za korištenje naprednih mogućnosti i definiran je ABI-jem – značenje se može naći u literaturi.

Ndx je index sectiona jer je svaki simbol definiran u odnosu na neki section. Tako je npr. simbol `init` definiran u odnosu na `.text` section koji je pod indeksom 1. Postoji nekoliko specijalnih "indeksa": `UND`, `COM` i `ABS`.

- **ABS** je apsolutni simbol čija se vrijednost ne mijenja prilikom relokacije.
- **UND** je simbol koji nije definiran u trenutnoj datoteci. Ta referenca se resolva prilikom linkanja globalnim ili weak simbolima koji su definirani u nekoj drugoj datoteci.
- **COM** je tzv. "common" simbol – za njega još nije alociran nikakav prostor. Npr. `int a;` deklarira globalni simbol `a` i takva deklaracija se može naći u koliko god datoteka.

Međutim, `int a = 7;` je **definicija** simbola `a` i može se pojaviti u samo *jednoj* datoteci. U ostalim datotekama se može pojaviti samo `int a;` ili `extern int a;`
Razlika je što `extern` stvara undefined simbol i ako ni u jednoj datoteci taj simbol nije deklariran niti kao common, linker *prijavljuje grešku* o nedefiniranom simbolu.

Na svim sustavima (uključujući one koji ne koriste ELF) se simboli i atributi u object fajlu mogu pogledati `nm` utilityjem.

Drugi ključan pojam je **relokacija**, a on se može razumjeti tek u vezi sa generiranjem strojnog koda za izvršavanje. Naime, kod se u relokatibilnim datotekama (obično) generira kao da će se izvršavati od adrese 0, a od te adrese se alociraju i adrese za sve podatke.

Sljedeći listing daje disasemblirani kod za naš primjer, zajedno sa pripadajućim relokacijama i simbolima:

```
Script started on Sun 22 Aug 2004 11:11:00 AM CEST
$ objdump -dr p1.o

p1.o:     file format elf32-i386

Disassembly of section .text:

00000000 <print>:
   0:  55                     push   %ebp
   1:  89 e5                  mov    %esp,%ebp
   3:  83 ec 08               sub    $0x8,%esp
   6:  ff 75 08               pushl  0x8(%ebp)
   9:  ff 35 00 00 00 00      pushl  0x0
                      b: R_386_32     .bss
   f:  a1 08 00 00 00         mov    0x8,%eax
                      10: R_386_32    .data
  14:  50                     push   %eax
  15:  ff 05 08 00 00 00      incl   0x8
                      17: R_386_32    .data
  1b:  68 14 00 00 00         push   $0x14
                      1c: R_386_32    .rodata
  20:  e8 fc ff ff ff         call   21 <print+0x21>
                      21: R_386_PC32  printf
  25:  83 c4 10               add    $0x10,%esp
  28:  c9                     leave
  29:  c3                     ret

0000002a <init>:
  2a:  55                     push   %ebp
  2b:  89 e5                  mov    %esp,%ebp
  2d:  83 ec 08               sub    $0x8,%esp
  30:  83 ec 0c               sub    $0xc,%esp
  33:  ff 35 04 00 00 00      pushl  0x4
                      35: R_386_32    .data
  39:  e8 c2 ff ff ff         call   0 <print>
  3e:  83 c4 10               add    $0x10,%esp
  41:  83 ec 0c               sub    $0xc,%esp
  44:  68 40 00 00 00         push   $0x40
                      45: R_386_32    .rodata
  49:  e8 fc ff ff ff         call   4a <init+0x20>
                      4a: R_386_PC32  printf
  4e:  83 c4 10               add    $0x10,%esp
  51:  c9                     leave
  52:  c3                     ret
```

```
00000053 <main>:
  53:  55                      push   %ebp
  54:  89 e5                   mov    %esp,%ebp
  56:  83 ec 08                sub    $0x8,%esp
  59:  83 e4 f0                and    $0xfffffff0,%esp
  5c:  b8 00 00 00 00          mov    $0x0,%eax
  61:  29 c4                   sub    %eax,%esp
  63:  e8 fc ff ff ff          call   64 <main+0x11>
                       64: R_386_PC32  init
  68:  83 ec 0c                sub    $0xc,%esp
  6b:  8b 45 0c                mov    0xc(%ebp),%eax
  6e:  ff 30                   pushl  (%eax)
  70:  e8 8b ff ff ff          call   0 <print>
  75:  83 c4 10                add    $0x10,%esp
  78:  c9                      leave
  79:  c3                      ret
$ readelf -r p1.o

Relocation section '.rel.text' at offset 0x1d70 contains 9 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000b  00000401 R_386_32          00000000   .bss
00000010  00000301 R_386_32          00000000   .data
00000017  00000301 R_386_32          00000000   .data
0000001c  00000801 R_386_32          00000000   .rodata
00000021  00001602 R_386_PC32        00000000   printf
00000035  00000301 R_386_32          00000000   .data
00000045  00000801 R_386_32          00000000   .rodata
0000004a  00001602 R_386_PC32        00000000   printf
00000064  00001702 R_386_PC32        0000002a   init

Relocation section '.rel.data' at offset 0x1db8 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000004  00000801 R_386_32          00000000   .rodata

Relocation section '.rel.debug_info' at offset 0x1dc0 contains 28 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000006  00000501 R_386_32          00000000   .debug_abbrev
0000000c  00000701 R_386_32          00000000   .debug_line
00000010  00000201 R_386_32          00000000   .text
00000014  00000201 R_386_32          00000000   .text
00000060  00001201 R_386_32          00000000   .debug_str
000002af  00001201 R_386_32          00000000   .debug_str
000004ae  00001201 R_386_32          00000000   .debug_str
000007a8  00001201 R_386_32          00000000   .debug_str
000007e0  00001201 R_386_32          00000000   .debug_str
00000ab2  00001201 R_386_32          00000000   .debug_str
00000b67  00001201 R_386_32          00000000   .debug_str
00000e02  00001201 R_386_32          00000000   .debug_str
00000e86  00001201 R_386_32          00000000   .debug_str
00000ed2  00001201 R_386_32          00000000   .debug_str
00001065  00001201 R_386_32          00000000   .debug_str
00001146  00000201 R_386_32          00000000   .text
0000114a  00000201 R_386_32          00000000   .text
00001167  00000301 R_386_32          00000000   .data
00001176  00000401 R_386_32          00000000   .bss
00001185  00000201 R_386_32          00000000   .text
00001189  00000201 R_386_32          00000000   .text
000011a1  00000201 R_386_32          00000000   .text
000011a5  00000201 R_386_32          00000000   .text
000011f9  00001901 R_386_32          00000004   a
00001209  00001501 R_386_32          00000000   b
```

```
0000121a  00000301 R_386_32          00000000   .data
00001229  00000801 R_386_32          00000000   .rodata
0000123d  00000401 R_386_32          00000000   .bss

Relocation section '.rel.debug_line' at offset 0x1ea0 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
000000d6  00000201 R_386_32          00000000   .text

Relocation section '.rel.debug_frame' at offset 0x1ea8 contains 6 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000018  00000f01 R_386_32          00000000   .debug_frame
0000001c  00000201 R_386_32          00000000   .text
00000034  00000f01 R_386_32          00000000   .debug_frame
00000038  00000201 R_386_32          00000000   .text
00000050  00000f01 R_386_32          00000000   .debug_frame
00000054  00000201 R_386_32          00000000   .text

Relocation section '.rel.debug_pubnames' at offset 0x1ed8 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000006  00000601 R_386_32          00000000   .debug_info

Relocation section '.rel.debug_aranges' at offset 0x1ee0 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000006  00000601 R_386_32          00000000   .debug_info
00000010  00000201 R_386_32          00000000   .text
zvrba@zax:~/work/coredump/linkers/code$
Script done on Sun 22 Aug 2004 11:11:17 AM CEST
```

Ovdje je disasemblirani kod isprepleten sa pripadajućim relokacijama. Kasnije su relokacije izlistane još posebno za sebe. Ovdje se neće ulaziti u detalje, dovoljno je reći da relokacije služe kako bi se jednostavno mogle popraviti adrese u instrukcijama. Kada se relokatibilna datoteka linka sa drugima, njeni podaci i kod se više ne mogu nalaziti na pretpostavljenim mjestima u trenutku generiranja koda. Da bi se kod mogao ispravno izvršavati, potrebno je popraviti adrese. Vidi se da i debug informacije imaju svoje relokacije. To je potrebno za source-level debugging (kada je potrebno iz adrese strojne instrukcije ili varijable saznati lokaciju u source kodu).

Osim relokatibilnih datoteka, linker dodaje i libraryje. Statički library je samo kolekcija relokatibilnih datoteka u jednoj datoteci (to je `ar` arhiva i sadržaj se može pogledati `ar` utilityjem). Ako se u libraryju nađe neka relokatibilna datoteka koja resolva neki simbol, onda se *cijela* da datoteka ulinka, neovisno o tome koliko ima funkcija u njoj. Npr. ako je u datoteci definirano 10 globalnih funkcija, a program koristi samo jednu, ulinkat će se kod od svih 10. Zbog toga je poželjno nezavisne funkcije staviti u posebne datoteke.

Osim statičkih libraryja postoje i dinamički libraryji čiji je mehanizam opisan kasnije. U fazi linkanja linker samo provjerava da su u navedenim dinamičkim libraryjima prisutni simboli koje program treba i u izvršnu datoteku ugrađuje referencu na dinamički library. Definicije se stvarno dohvaćaju u trenutku izvršavanja programa.

Gotovo svi UNIX linkeri (uključujući GNU `ld` koji se koristi na Linuxu) su **jednoprolazni**: kada vidi jedan library, resolva trenutno postojeće nedefinirane simbole, a ostali simboli koji bi se kasnije mogli resolvati iz tog libraryja se *zaboravljaju i ostaju nedefinirani.* Zbog toga je bitan i *redoslijed* navođenja libraryja prilikom linkanja: *svi* libraryji koji koriste funkcije nekog libraryja X moraju se navesti *prije* libraryja X. Ako postoji *kružna* zavisnost između libraryja (npr. library A koristi neku funkciju libraryja B i obratno), tada se jedan od tih libraryja treba navesti *dvaput* prilikom linkanja.

2 Debug informacije

Osim DWARF2/3 formata debug informacija, drugi (stariji) rašireni format je STABS koji se sprema u `.stabs` section. Ostali proizvođači kompajlera imaju vlastite proprietary formate debug informacija i način kako ih ugrađuju u objektnu i izvršnu datoteku. Više o DWARF formatu i libraryju za čitanje i pisanje tog formata može se naći u navedenoj literaturi.

Debug informacije su dovoljno detaljne da se mogu iskoristiti i za druge stvari osim za debugiranje programa. Na primjer, napravio sam program koji iz debug informacija crta inheritance klasa u C++ programu - puno je jednostavnije parsirati debug informacije nego C++ kod.

Druga mogućnost je *introspection* ili *reflection* što omogućava dinamičko (po imenu) instanciranje klasa i pozivanje funkcija.

3 Učitavanje programa

Osim sekcija postoje i tzv. **segmenti**, odnosno u ELF terminologiji "**program headers**". Sekcije određuju fizički layout ELF datoteke i imaju značenje *prije linkanja*, a segmenti su bitni u fazi *izvršavanja* (dakle, ne postoje u običnim object datotekama već isključivo izvršnim programima i shareanim libaryjima). Segmenti određuju kako se program učitava u memoriju i jedan segment se može sastojati od više sectiona. Razmještajem u sekcije i segmente prilikom finalnog linkanja u izvršni program ili SO se upravlja pomoću linker skripte.

Primjer segmenata i sectiona (`a.out` je defaultno ime izvršne datoteke koju generira kompajler, ako se ne zada drukčije):

```
Script started on Sat 14 Aug 2004 07:01:16 PM CEST
$ readelf -l a.out

Elf file type is EXEC (Executable file)
Entry point 0x80482c0
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00544 0x00544 R E 0x1000
  LOAD           0x000544 0x08049544 0x08049544 0x00110 0x00120 RW  0x1000
  DYNAMIC        0x000560 0x08049560 0x08049560 0x000c8 0x000c8 RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
  STACK          0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version
          .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata
   03     .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
   04     .dynamic
   05     .note.ABI-tag
   06
$ readelf -S a.out
There are 33 section headers, starting at offset 0x1b30:
```

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048114 000114 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048148 000148 00002c 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          08048174 000174 000060 10   A  5   1  4
  [ 5] .dynstr           STRTAB          080481d4 0001d4 000060 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          08048234 000234 00000c 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         08048240 000240 000020 00   A  5   1  4
  [ 8] .rel.dyn          REL             08048260 000260 000008 08   A  4   0  4
  [ 9] .rel.plt          REL             08048268 000268 000010 08   A  4   b  4
  [10] .init             PROGBITS        08048278 000278 000017 00  AX  0   0  4
  [11] .plt              PROGBITS        08048290 000290 000030 04  AX  0   0  4
  [12] .text             PROGBITS        080482c0 0002c0 0001e0 00  AX  0   0 16
  [13] .fini             PROGBITS        080484a0 0004a0 00001b 00  AX  0   0  4
  [14] .rodata           PROGBITS        080484c0 0004c0 000084 00   A  0   0 32
  [15] .data             PROGBITS        08049544 000544 000018 00  WA  0   0  4
  [16] .eh_frame         PROGBITS        0804955c 00055c 000004 00   A  0   0  4
  [17] .dynamic          DYNAMIC         08049560 000560 0000c8 08  WA  5   0  4
  [18] .ctors            PROGBITS        08049628 000628 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        08049630 000630 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049638 000638 000004 00  WA  0   0  4
  [21] .got              PROGBITS        0804963c 00063c 000018 04  WA  0   0  4
  [22] .bss              NOBITS          08049654 000654 000010 00  WA  0   0  4
  [23] .comment          PROGBITS        00000000 000654 00007e 00      0   0  1
  [24] .debug_aranges    PROGBITS        00000000 0006d8 000058 00      0   0  8
  [25] .debug_pubnames   PROGBITS        00000000 000730 000025 00      0   0  1
  [26] .debug_info       PROGBITS        00000000 000755 00096e 00      0   0  1
  [27] .debug_abbrev     PROGBITS        00000000 0010c3 000124 00      0   0  1
  [28] .debug_line       PROGBITS        00000000 0011e7 0001ca 00      0   0  1
  [29] .debug_str        PROGBITS        00000000 0013b1 00065f 01  MS  0   0  1
  [30] .shstrtab         STRTAB          00000000 001a10 00011e 00      0   0  1
  [31] .symtab           SYMTAB          00000000 002058 000730 10     32  58  4
  [32] .strtab           STRTAB          00000000 002788 00034c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
$
Script done on Sat 14 Aug 2004 07:01:23 PM CEST
```

Ovdje se vidi da se jedan segment sastoji od više sectiona i ima neke flagove. Značenje atributa segmenata i sectiona:

Offset je offset u izvršnoj datoteci gdje počinje segment.

Type je tip segmenta ili sectiona. Tipovi imaju sljedeća značenja:

- **PHDR** je tzv. "Program header". To je upravo tablica segmenata.
- **INTERP, DYNAMIC** su prisutni samo kod SO-ova i dinamički linkanih izvršnih datoteka. O njima će biti riječi kod opisivanja postupka dinamičkog linkanja.
- **LOAD** segmenti čine tzv. "program image" – ono što u stvari jesu instrukcije i podaci programa. Obično su uvijek dva: jedan sadrži kod i read-only podatke, drugi sadrži inicijalizirane podatke.
- **NOTE** služi za dodatne obavijesti. UNIX ABI ovo koristi za informacije o potrebnoj runtime okolini (npr. verzija C libraryja, verzija kernela i slično).
- **STACK** može poslužiti za alociranje stacka unaprijed. Međutim, na Linuxu kernel automatski dodjeljuje stack procesu te to ovdje nije potrebno.

VirtAddr, PhysAddr su adrese u virtualnoj, odn. fizičkoj memoriji kamo se učitava segment. Na modernim operacijskim sustavima fizička adresa nema značaja i program se učitava na virtualnu adresu.

FileSiz, MemSiz Veličina segmenta u datoteci, odn. u memoriji. Ove dvije veličine se mogu *razlikovati*.

Jedan slučaj kada se to sigurno dešava su neinicijalizirani podaci: varijable bez inicijalizirane početne vrijednosti ne zauzimaju nikakav prostor u izvršnoj datoteci, a kernel pri učitavanju programa treba dodijeliti prostor koliko ukupno zauzimaju inicijalizirani i neinicijalizirani podaci.

Flg (flagovi) su **dozvole** za segment: R znači dozvola čitanja podataka, W je dozvola pisanja podataka, a E je dozvola izvršavanja koda.[5]

Sectioni također imaju svoje flagove. Najvažnije je spomenuti A (alloc) flag: ako je on prisutan u sectionu, to znači da će taj section na neki način pridonijeti finalnom image-u programa u memoriji nakon učitavanja. Npr. `.comment` section nije označen kao A te nije dodijeljen nijednom segmentu.

Align Adresa na koju se segment učitava mora biti djeljiva s tom vrijednošću.

Entry point je (virtualna) adresa prve instrukcije programa.

Prvi koraci kod pokretanja programa su sljedeći:

1. Kernel **mapira** izvršnu datoteku u memoriju i to svaki LOAD segment na virtualnu adresu specificiranu u headeru. Mapiranje znači da pristup memoriji od te adrese zapravo pristupa sadržaju datoteke.

   Iz headera se vidi da je prvi LOAD segment na samom početku datoteke (offset 0), a drugi je neposredno iza njega. Takva organizacija izvršne datoteke je zgodna jer jako pojednostavljuje proces učitavanja programa: datoteka se mapira od početka u ukupnoj duljini LOAD segmenata.

   Ostali podaci bitni za dinamičko linkanje, debug podaci i sl. što nije bitno za samo izvršavanje programa nalaze se nakon LOAD segmenata i *ne mapiraju* se u memoriju. Samim time se štedi memorija (u velikim C++ programima debug podaci mogu zauzimati i po 500MB, a samog koda i podataka ima manje od 50MB).

2. Ako je program dinamički linkan, kernel mapira u memoriju dinamički linker koji je specificiran u INTERP segmentu (i .interp sectionu). Kontrola se prenosi na entry point dinamičkog linkera koji relocira program (o tome kasnije).

3. Kontrola se prenosi na entry point programa, bilo iz kernela (statički linkani program) ili iz dinamičkog linkera (dinamički linkani program).

3.1 Statičko i mehanizam dinamičkog linkanja

Izvršni program može biti **statički** i **dinamički** linkan.

[5] Na Intel x86 arhitekturi R automatski podrazumijeva i E – ne mogu se postavljati nezavisno. To je izvor mnogih buffer overflow sigurnosnih bugova koji omogućavaju izvršavanje koda sa stacka. Druge arhitekture procesora (npr. SPARC) te AMD-ova x86-64 razlikuju R i E dozvole.

Statički linkan programa ne ovisi o vanjskim libraryjima – on je u potpunosti samodovoljan i može ga se pokrenuti na bilo kojem računalu s odgovarajućom verzijom operacijskog sustava.

Dinamički linkan program ovisi o SO-ovima koji su u njega ugrađeni u trenutku linkanja, na primjer:

```
$ readelf -d a.out

Dynamic segment at offset 0x560 contains 20 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x8048278
 0x0000000d (FINI)                       0x80484a0
 0x00000004 (HASH)                       0x8048148
 0x00000005 (STRTAB)                     0x80481d4
 0x00000006 (SYMTAB)                     0x8048174
 0x0000000a (STRSZ)                      96 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804963c
 0x00000002 (PLTRELSZ)                   16 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x8048268
 0x00000011 (REL)                        0x8048260
 0x00000012 (RELSZ)                      8 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x8048240
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x8048234
 0x00000000 (NULL)                       0x0
```

Svi SO-ovi koji su označeni kao NEEDED moraju u trenutku izvođenja biti prisutni na sistemu gdje se program izvršava. Prednosti dinamičkog linkanja ima nekoliko:

- Manje zauzeće diska. Gotovo svaki program koristi neke složene C library funkcije poput `sprintf`. Kada ne bi bilo dinamičkog linkanja, svaki program na sustavu bi imao u sebi kopiju te rutine. Ovako svi programi dijele isti kod iz SO-a.

- Lakše održavanje: ako se u nekom sistemskom libraryju (poput C libraryja, `libc.so.6`) nađu bugovi, samo se na jednom mjestu zamijeni SO koji svi programi koriste. Statički linkane programe bi trebalo ponovno linkati na novu verziju libraryja.

- Manje zauzeće memorije. Dobar operacijski sustav omogućava da *različiti* procesi dijele *fizički istu* memoriju. Svaki proces dobije svoju kopiju podataka tek kad pokuša pisati po dijeljenoj memoriji (tzv. **copy on write**).

  ELF omogućava jednostavno učitavanje SO-a *na bilo koju adresu u memoriji.* Posljedica toga je da i *u memoriji* postoji *samo jedna kopija* svakog shareanog libraryja. Kad se pokrene neki novi program koji koristi SO koji se nalazi već u memoriji, ne učitava se nova kopija SO-a već se koristi postojeća.

  Ovaj princip dijeljena je, naravno, primjenjiv samo na read-only podatke i kod (za koji se podrazumijeva da je read-only). Kad neki program mijenja globalne podatke SO-a, tada dobije vlastitu kopiju tih podataka.

3.2 Parametri koji utječu na dinamičko linkanje

Na mehanizam dinamičkog linkanja može se utjecati na nekoliko načina, uglavnom pomoću environment varijabli.

Prilikom pokretanja programa, linker traži potrebne libraryje po određenom algoritmu, detalji kojega se mogu pogledati u `man ld.so`). Path u kojem se traže libraryji određuju sljedeći parametri:

- Direktoriji navedeni u datoteci `/etc/ld.so.conf`. Nakon svake izmjene (dodavanje ili brisanje direktorija), potrebno je pokrenuti program `ldconfig`.
- Environment varijabla `LD_LIBRARY_PATH`.
- Path ugrađen u sam izvršni program. Taj path se pokazuje kao `RPATH` ili `RUNPATH` u dinamičkom segmentu programa. Može se ugraditi u izvršni program pomoću `-rpath` opcije linkeru.

Environment varijabla `LD_PRELOAD` je lista dodatnih libraryja (odvojenih razmacima) koji se učitavaju prije svih ostalih libraryja. Neki razlozi mogu biti side-efekti inicijalizacije/finalizacije prilikom učitavanja i oslobađanja libraryja (`_init` i `_fini` funkcije te GCC konstruktor i i destruktor funkcije) ili za "overridanje" funkcija u drugim SO-ovima: ako je neki simbol definiran u `LD_PRELOAD` libraryju, koristit će se taj simbol umjesto onoga koji bi program inače koristio. Ovo ima utjecaja i na eksplicitno dinamičko linkanje: do svih definicija jednog simbola se može doći `RTLD_NEXT` handleom u pozivu funkcije `dlsym()`. Tako se mogu jednostavno "wrapati" funkcije u postojećim SO-ovima. `LD_PRELOAD` mehanizam nema efekta ako izvršni program nije dinamički linkan.

Osim izvršnih programa, i SO može biti dinamički linkan na druge SO-ove koji se automatski učitavaju prilikom učitavanja 'glavnog' SO-a.

U biti, kod ELF-a i nema neke bitne razlike između izvršnih programa i SO-ova. Najbitnija razlika je da se kod za SO mora kompajlirati kao PIC (position-independent code) sa `-fPIC` opcijom GCC kompajleru. Posljedica toga je da se SO može učitati na bilo koju adresu u memoriji, za razliku od izvršnog programa koji se učitava na fiksnu adresu.

Ako se environment varijabla `LD_BIND_NOW` postavi na 1, onda dinamički linker odmah po učitavanju programa resolva sve dinamičke funkcije umjesto u trenutku poziva (više o tom mehanizmu kasnije).

3.3 Implicitno i eksplicitno dinamičko linkanje

Mehanizam dinamičkog linkanja koji je do sada opisan mogao bi se nazvati implicitnim. Osim njega postoji mogućnost eksplicitnog učitavanja SO-a i pristupu funkcijama i varijablama unutar njega. Na ELF platformama, a i propisan POSIX standardom, raširen je `dlopen()` interface nastao na Solarisu, a preuzeli su ga Linux i *BSD. Druge varijante UNIX-a (HP-UX, AIX,) imaju vlastiti, nekompatibilan s ovim, interface.

Primjer korištenja `dlopen()` i pripadajućih funkcija dan je u sljedećem programu:

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

int main(int argc, char **argv) {
  void *handle;
  double (*cosine)(double);
  char *error;
```

```
  handle = dlopen ("libm.so", RTLD_LAZY);
  if (!handle) {
    fprintf (stderr, "%s\n", dlerror());
    exit(1);
  }

  dlerror();    /* Clear any existing error */
  *(void **) (&cosine) = dlsym(handle, "cos");
  if ((error = dlerror()) != NULL)  {
    fprintf (stderr, "%s\n", error);
    exit(1);
  }

  printf ("%f\n", (*cosine)(2.0));
  dlclose(handle);
  return 0;
}
```

Kao što je rečeno, sa ELF-om je mala razlika između izvršnog programa i SO-a. Tako je moguće i da SO poziva funkcije *glavnog programa*. Preduvjet za to je da se izvršni program linka sa `-rdynamic` opcijom GCC kompajleru, odn. `-E` ili `--export-dynamic` opcijom `ld` linkeru.

Mehanizam *implicitnog* dinamičkog linkanja je drukčiji. Tome služe `.plt` (PLT: program linkage table) i `.got` (GOT: global offset table) sectioni. Kao što se može vidjeti iz podataka o segmentima, `.plt` se nalazi u read-only segmentu, a `.got` je u read-write segmentu.

PLT sadrži tablicu skokova koja se koristi kad se pozivaju funkcije iz SO-a: poziv funkcije skače na kod unutar PLT-a. Na x86 arhitekturi, svaki entry unutar PLT je instruckija *indirektnog skoka* preko lokacije unutar GOT. Nakon učitavanja programa, ti entryji GOT tablice ne pokazuju na pravu funkciju nego na kod *unutar dinamičkog linkera*. *Prvi put* kad se pozove neka funkcija, dinamički linker nađe pravu adresu funkcije i popravi GOT entry. GOT se također koristi i za adresiranje svih globalnih podataka unutar SO-a.

Nakon popravljanja entryja, sam dinamički linker poziva funkciju. Svaki idući put program direktno zove funkciju. Ovakav način učitavanja se zove **lazy binding** i ubrzava učitavanje programa. Može se isključiti postavljanjem environment varijable `LD_BIND_NOW` na 1 i tada dinamički linker resolva sve adrese dinamičkih funkcija prije nego što preda kontrolu programu.

3.4 Razlike između ELF i COFF

Nedostatak Win32/COFF-a je da svaki DLL ima *fiksnu load adresu*. Ako se DLL učita na drugu adresu (npr. proces alocira puno memorije pa zauzme adrese gdje se trebao učitati DLL), rade se relokacije. Efektivno, proces dobije *svoju vlastitu kopiju* DLL-a i ne može koristiti postojeće kopije u memoriji.

Razlog za to je što Win32/COFF nema podršku za PIC. Tako se u DLL ugrađuje i tablica relokacija. Ako OS loader ne može učitati DLL na traženu adresu, sve adrese u kodu i podacima se relociraju prema tablici relokacija. Copy-on-write mehanizam radi na nivou procesorskih pageova (na x86 arhitekturi je jedan page 4kB) te promjena makar jednog bajta unutar pagea uzrokuje stvaranje privatne kopije tog pagea za proces koji ga mijenja. Da bi neki page DLL-a ostao dijeljen između dva procesa, makar se u drugom nije uspio učitati na traženu adresu, u tom pageu *ne smije biti relokacija*. Međutim, tada ispada da bi se trebalo pojaviti 4kB koda u komadu koji ne zove nikakve druge rutine unutar DLL-a i ne referencira globalne ili static podatke – takav kod je vrlo rijedak.

Kod ELF-a SO-a postoji sličan problem, međutim, umjesto relociranja cijelog koda, relociraju se samo PLT i GOT tablice jer kompajler prilikom generiranja koda (zbog `-fPIC` opcije) indirektno preko njih adresira funkcije i globalne podatke unutar SO-a. Tako svaki proces dobije vlastitu kopiju *samo tih tablica* (koje su redovito vrlo male) i promijenjenih podataka unutar SO-a.

4 Literatura

Ovaj članak daje samo površan uvid u principe funkcioniranja linkera i loadera. Puno detaljniji opis, uključujući i primjere u assembleru, može se naći u navedenoj literaturi.

1. Linkovi na detaljnu dokumentaciju vezanu uz ELF i DWARF formate (uključujući specifikacije) te ABI (application binary interface) dokumentacija.
   `http://www.linuxbase.org/spec/refspecs`

2. libdwarf - library za čitanje i pisanje DWARF2 debug formata. Uz source dolazi i detaljna specifikacija samog formata.
   `http://reality.sgi.com/davea/`

3. Puno korisnih programa i libraryja, uključujući “Reflection Package for C++”. Nažalost, ovaj library radi pod UNIX-om samo za STABS debugging format, a pod Win32 radi isključivo uz označavanje klasa makroima (ne zna čitati Microsoftov debug format).
   `http://www.garret.ru/~knizhnik/`