HY255 Εργαστήριο Λογισμικού – L21

Linking, Loading

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2025
Άγγελος Μπίλας

Linking, Loading, Libraries

- Linking: Η διαδικασία όπου από πολλά .o αρχεία και αρχεία βιβλιοθήκης, παράγουμε ένα εκτελέσιμο αρχείο a.out
- Loading: Η διαδικασία όπου το εκτελέσιμο αρχείο a.out φορτώνεται στη μνήμη και αρχίζει η εκτέλεσή του
- Αρχείο βιβλιοθήκης
  - Είναι ένα "archive" που περιέχει πολλά .o αρχεία για να έχουμε λιγότερα αρχεία
  - Άρα τα αρχεία βιβλιοθήκης επί της ουσίας δεν διαφέρουν από τα .o
  - Στη C οι βιβλιοθήκες ονομάζονται libNAME.a, π.χ. libc.a για την standard βιβλιοθήκη της C, libm.a για την βιβλιοθήκη που έχει μαθηματικές συναρτήσεις, libsock.a, libpthread.a, και πολλές άλλες
  - Για να βάλουμε πολλά .o αρχεία σε μια βιβλιοθήκη μπορούμε να χρησιμοποιήσουμε το πρόγραμμα `ar` (archive)

```
gcc -ansi -pedantic -Wall -c max.c
ar -r libmax.a max.o
gcc -ansi -pedantic -Wall -c main.c
gcc -static main.o libmax.a [ή gcc -static main.o -L. -lmax]
```

  - Οι εντολές αυτές είναι για τα συστήματα του Τμήματος (π.χ. milo)
- Το linking των .o αρχείων σε ένα εκτελέσιμο γίνεται από ένα πρόγραμμα που ονομάζεται link editor ή για συντομία linker και παραδοσιακά είναι το εκτελέσιμο `ld`
  - Ο gcc στην παραπάνω ακολουθία εντολών, στην τελευταία εντολή, απλά καλεί τον linker με τα κατάλληλα paths και παραμέτρους – Το πρόγραμμα gcc είναι ένας "driver" που καλεί άλλα προγράμματα (τον preprocessor cpp, τον compiler cc1, τον linker ld)

Static vs. Dynamic Linking

- Static linking: Το linking όπου όλα τα .o αρχεία του προγράμματος και των βιβλιοθηκών τοποθετούνται στο εκτελέσιμο a.out
  - Στο στατικό linking η διαδικασία του loading είναι απλή αφού το εκτελέσιμο a.out περιέχει όλα όσα χρειάζεται (δεδομένα, κώδικα) για να τρέξει το πρόγραμμα
  - Οι στατικές βιβλιοθήκες ονομάζονται libNAME.a (όπως είπαμε)
  - Το 1ο αρνητικό είναι ότι το a.out τείνει να είναι μεγάλο γιατί περιέχει και όλες τις βιβλιοθήκες – αυτό σήμερα για τα πιο πολλά συστήματα είναι μικρό κακό – γενικά ο χώρος στους δίσκους είναι σήμερα "πολύς"
  - Το 2ο και μεγαλύτερο αρνητικό στοιχείο είναι ότι κάθε φορά που βάζουμε το a.out στη μνήμη δημιουργούμε υποχρεωτικά ένα αντίγραφο των βιβλιοθηκών (για κάθε a.out) και καταναλώνουμε μνήμη που είναι πολύτιμη (σχετικά λίγη και ακριβή)
- Dynamic linking: Στο δυναμικό linking τα .o αρχεία του προγράμματος (ενώνονται) σε ένα a.out ΑΛΛΑ ΌΧΙ οι βιβλιοθήκες (που τις ονομάζουμε και shared objects – από όπου προέρχεται και το επίθεμα .so). Το εκτελέσιμο περιέχει μόνο πληροφορίες για το ποιες βιβλιοθήκες θα χρειαστεί, αλλά όχι τον κώδικα των βιβλιοθηκών
  - Ονομάζεται και "linking με shared objects" γιατί ο σκοπός του dynamic linking είναι τα διάφορα modules να είναι shared (.o αρχεία, βιβλιοθήκες)
  - Όταν το εκτελέσιμο a.out φορτώνεται στη μνήμη χρειάζεται να γίνει link με τις βιβλιοθήκες (shared objects)
  - Κάθε βιβλιοθήκη τοποθετείται μόνο μια φορά στη μνήμη για ΌΛΑ τα προγράμματα
  - Αν η βιβλιοθήκη που χρειάζεται ένα πρόγραμμα υπάρχει, τότε γίνεται το linking με αυτή
  - Αν δεν υπάρχει, τότε θα φορτωθεί στη μνήμη, για όλα τα προγράμματα που θα την χρειαστούν στη συνέχεια
  - Στη συνέχεια ο loader επαναλαμβάνει μέρος της διαδικασία του linking ανάμεσα στο a.out και στις δυναμικές βιβλιοθήκες
  - Κατόπιν το a.out μπορεί να εκτελεστεί
  - Οι δυναμικές βιβλιοθήκες ονομάζονται libNAME.so (.so από το shared object)
  - Οι δυναμικές βιβλιοθήκες παράγονται από ένα σύνολο .o αρχεία με τον link editor (ld) με την παράμετρο –shared

```
gcc -ansi -pedantic -Wall -c max.c
ld -shared -o libmax.so max.o
gcc -ansi -pedantic -Wall -c main.c
gcc -shared ./libmax.so main.o [ή gcc -shared main.o -L. -lmax]
```

  Οι εντολές αυτές είναι για τα συστήματα του Τμήματος (π.χ. milo)
- Σήμερα χρησιμοποιείται περισσότερο το δυναμικό linking και είναι και το default στις πιο πολλές περιπτώσεις
  - Αλλά υπάρχουν αρκετοί λόγοι για τους οποίους χρειαζόμαστε και το στατικό linking, το οποίο εξακολουθεί να χρησιμοποιείται
  - Π.χ με το στατικό linking δεν εξαρτόμαστε από το αν υπάρχουν οι βιβλιοθήκες που χρειάζεται το πρόγραμμά μας στο σύστημα που θα τρέξει τελικά

Linking: Link Editor

```
main.c:

extern int x;
int y=1;
extern int f(int);
int main(void){
      x = f(x);
      return x;
}
```

```
f.c:

int x=2;
extern int y;
int f(int a){
  return a+x+y;
}
```

main.o

	offset	contents
0: Data section	+0	y: 1
4: Code section	+0	main:leal **x?**, %ebx
	+4	pushl (%ebx)
	+8	call **f?**
	+12	popl %ecx
	+16	movl %eax, (%ebx)
	+20	ret

EST

index	name	def/used	otype	mem loc	segm
0	x	U	OBJ	-	-
1	y	D	OBJ	+0	data
2	f	U	FUN	-	-
3	main	D	FUN	+0	code

RT

index	loc	EST loc	rtype
0	+0	0 (x)	rel
1	+8	2 (f)	abs

f.o

	offset	contents
0: Data section	+0	x: 2
4: Code section	+0	f: popl %eax
	+4	addl **x?**, %eax
	+8	addl **y?**, %eax
	+12	ret

EST

index	name	def/used	otype	mem loc	segm
0	x	D	OBJ	+0	data
1	y	U	OBJ	-	-
2	f	D	FUN	+0	code

RT

index	loc	EST loc	rtype
0	+4	0 (x)	abs
1	+8	2 (y)	abs

- EST = External Symbol Table: A symbol table (SymTab of assignment 3) that contains all global and external symbols in main.c
- RT = Relocation Table: Contains one entry for each location in the code of main.c that uses a global/external symbol
- otype = Object type: The type of object. OBJ = var, array, etc. FUN = code in a function
- mem loc/segm = memory location/segment: Location of object in memory in the corresponding segment
- EST loc = EST location: Location of referred object in EST
- rtype = Reference type: Type of reference in the code - can be either an absolute (abs) or a relative (rel) address
- **The link editor performs the following three steps:**
  - **1. Merge:** copy files .o files in a single a.out file and assign new addresses to each section
  - **2. Resolve (in EST):** Use the new ESTs in a.out to create a single consolidated EST and resolve the external symbols and references and adjust the addresses in the merged RT
  - **3. Relocate (using RT):** Use the RT in a.out and replace in the code all references to external symbols with the correct addresses (absolute or relative)

1. Merge

- Copy each .o in a.out and put data, code, ESTs, RTs together
- Adjust each section offset in a.out to be the current location in a.out

a.out	offset	contents
0: Data section DS1 (main.o)	+0	y: 1
4: Data section DS2 (f.o)	+0	x: 2
8: Code section CS1 (main.o)	+0 +4 +8 +12 +16 +20	main:leal x, %ebx pushl (%ebx) call f popl %ecx movl %eax, (%ebx) ret

32: Code section CS2 (f.o)	+0
	+4
	+8
	+12
		f: popl %eax
		addl x, %eax
		addl y, %eax
		ret

EST (main.o)	**index**	**name | def/used | otype | loc**
	0	x | U | OBJ | -
	1	y | D | OBJ | +0 (DS1)
	2	f | U | FUN | -
	3	main | D | FUN | +0 (CS1)
EST (f.o)	**index**	**name | def/used | otype | loc**
	0	x | D | OBJ | +0 (DS2)
	1	y | U | OBJ | -
	2	f | D | FUN | +0 (CS2)
RT (main.o)	**index**	**loc | ESD loc | rtype**
	0	+0 | 0 (x) | rel
	1	+8 | 2 (f) | abs
RT (f.o)	**index**	**loc | ESD loc | rtype**
	0	+4 | 0 (x) | abs
	1	+8 | 2 (y) | abs

2. Resolve

- Merge ESTs in a single consolidated EST and keep one (resolved copy) for each symbol
- Merge RTs in a single RT and adjust the location of each symbol in the consolidated EST

a.out	offset	contents
0: Data Section DS1 (main.o)	+0	y: 1
4: Data Section DS2 (f.o)	+0	x: 2
8: Code section CS1 (main.o)	+0 +4 +8 +12 +16 +20	main:leal x, %ebx pushl (%ebx) call f popl %ecx movl %eax, (%ebx) ret

32: Code section CS2 (f.o)	+0
	+4
	+8
	+12
		f: popl %eax
		addl x, %eax
		addl y, %eax
		ret

Consolidated EST

index	name	def/used	otype	loc
~~0~~	~~x~~	~~U~~	~~OBJ~~	~~-~~
0 1	y	D	OBJ	+0 (CS1)
~~2~~	~~f~~	~~U~~	~~FUN~~	~~-~~
1 ~~3~~	main	D	FUN	+0 (DS1)

EST

index	name	def/used	otype	loc
2 ~~0~~	x	D	OBJ	+0 (DS2)
~~1~~	~~y~~	~~U~~	~~OBJ~~	~~-~~
3 ~~2~~	f	D	FUN	+0 (CS2)

RT (main.o)

index	loc	EST loc	type	segm
0	+0	2 ~~0~~ (x)	rel	CS1
1	+8	3 ~~2~~ (f)	abs	CS1

RT (f.o)

index	loc	EST loc	type	
2 ~~0~~	+4	2 ~~0~~ (x)	abs	CS2
3 ~~1~~	+8	0 ~~2~~ (y)	abs	CS2

2. Resolve (final)

- If any symbols in EST remain as U then there is an error and the program cannot run
  - E.g. there is a declaration without any definition in the source program
- If any symbol in EST has two “D” entries, then there is an error and the program cannot run
  - E.g. there are two definitions for the same symbol in the source program

a.out	offset	contents
0: Data Section DS1 (main.o)	+0	y: 1
4: Data Section DS2 (f.o)	+0	x: 2
8: Code section CS1 (main.o)	+0 +4 +8 +12 +16 +20	main:leal x, %ebx pushl (%ebx) call f popl %ecx movl %eax, (%ebx) ret

32: Code section CS2 (f.o)	+0
	+4
	+8
	+12
		f: popl %eax
		addl x, %eax
		addl y, %eax
		ret

Consolidated EST

index	name	def/used	otype	loc
0	y	D	OBJ	+0 (DS1)
1	main	D	FUN	+0 (CS1)
2	x	D	OBJ	+0 (DS2)
3	f	D	FUN	+0 (CS2)

RT

index	loc	EST loc	type	segm
0	+0	2 (x)	rel	CS1
1	+8	3 (f)	abs	CS1
2	+4	2 (x)	abs	CS2
3	+8	0 (y)	abs	CS2

3. Relocate

- Assume a.out will be placed in memory at starting address P=0
- Adjust each section address by adding +P to its offset S
- Traverse RT and modify references (in program machine code) with correct memory address for program P, section S, symbol V, so P+S+V

a.out	offset	contents
P+0: Data Section DS1	+0	y: 1
+4: Data Section DS2	+0	x: 2
+8: Code section CS1	+0	main:leal ~~x~~ (PC-8=P)+4+0, %ebx
	+4	pushl (%ebx)
	+8	call ~~f~~ P+32+0
	+12	popl %ecx
	+16	movl %eax, (%ebx)
	+20	ret
+32: Code section CS2	+0	f: popl %eax
	+4	addl ~~x~~ P+4+0, %eax
	+8	addl ~~y~~ P+0+0, %eax
	+12	ret

Consolidated EST

index	name	def/used	otype	loc
0	y	D	OBJ	DS1+0
1	main	D	FUN	CS1+0
2	x	D	OBJ	DS2+0
3	f	D	FUN	CS2+0

RT

index	loc	EST loc	type	segm
0	+0	2 (x)	rel	CS1
1	+8	3 (f)	abs	CS1
2	+4	2 (x)	abs	CS2
3	+8	0 (y)	abs	CS2

3. Relocate (clean)

- Retain RT for adjusting symbol locations when loading the program at actual address P, before execution

a.out	offset	contents
P+0: Data Section DS1	+0	y: 1
+4: Data Section DS2	+0	x: 2
+8: Code section CS1	+0 +4 +8 +12 +16 +20	main:lea PC-8+4+0, %ebx pushl (%ebx) call +32+0 popl %ecx movl %eax, (%ebx) ret

+32: Code section CS2	+0
	+4
	+8
	+12
		f: popl %eax
		addl +4+0, %eax
		addl +0+0, %eax
		ret

Consolidated EST	**index** 0	**name | def/used | otype | loc** main | D | FUN | +0 (CS1)
RT	**index** 0 1 2 3	**loc | ESD loc | type | segm** +0 | 2 | rel | CS1 +8 | 3 | abs | CS1 +4 | 2 | abs | CS2 +8 | 0 | abs | CS2

Load and Execute

- Load a.out in memory at some address P
- Traverse RT and for each entry add P to the corresponding code location, instruction field
- Drop RT
- Jump to main at address: P+CS1(8)+offset(0)=P+8+0

a.out	offset	contents
P+0: Data Section DS1	+0	y: 1
+4: Data Section DS2	+0	x: 2
+8: Code section CS1	+0	main:leal PC-4, %ebx
	+4	pushl (%ebx)
	+8	call P+32
	+12	popl %ecx
	+16	movl %eax, (%ebx)
	+20	ret
+32: Code section CS2	+0	f: popl %eax
	+4	addl P+4, %eax
	+8	addl P+0, %eax
	+12	ret
Consolidated EST/ SymTab	**index**	**name | def/used | otype | loc**
	0	main | D | FUN | CS1+0
RT	**index**	**loc | | type | segm**
	0	+0 | | rel | CS1
	1	+8 | | abs | CS1
	2	+4 | | abs | CS2
	3	+8 | | abs | CS2

File format for .o, a.out, .a, .so files

- Executable and Linkable Format (ELF)

ELF Header	Ο τύπος του αρχείου, π.χ. .o, εκτελέσιμο, lib, machine type (x86), byte ordering, κλπ.
Program Header Table	Αρχική διεύθυνση και μέγεθος για κάθε segment στη μνήμη
.text	κώδικας μηχανής
.data	αρχικοποιημένες μεταβλητές
.bss	μη αρχικοποιημένες μεταβλητές
.symtab (EST)	πληροφορίες για κάθε σύμβολο του προγράμματος για σκοπούς του linker
.rel.text (RT)	πληροφορίες για κάθε διεύθυνση κώδικα που πρέπει να αλλάξει μετά το relocation
.rel.data (RT)	πληροφορίες για κάθε διεύθυνση μεταβλητής που πρέπει να αλλάξει μετά το relocation
.debug	πληροφορίες για κάθε σύμβολο του προγράμματος για σκοπούς του debugger

- Υπάρχουν συναρτήσεις βιβλιοθήκης για να επεξεργάζεται κανείς τα περιεχόμενα ενός ELF αρχείου

Λάθη κατά το linking

- (1) Undefined symbols

```
f1.c:                    f2.c:

extern int x;            extern int x;

int main(void){          int f(void) {;}
  f();
}
```

- (2) Multiple definitions

```
f1.c:                    f2.c:

int x;                   int x;

int main(void){          int f(void) {;}
  f();
}
```

- Οι κανόνες για το τι είναι σωστό/λάθος είναι λίγο πιο λεπτομερείς:
  - Weak symbols: uninitialized global variables
  - Strong symbols: initialized global variables
- Rules
  - Two strong with same name → error
  - Two weak with same name → use any
  - One strong and one weak → use strong symbol
- Δεδομένου ότι αυτοί οι κανόνες δημιουργούν ασάφεια για το πως θα γίνει τελικά link το πρόγραμμα, χρησιμοποιούμε πάντα το “static” για να κρύβουμε τις μεταβλητές που δεν πρέπει να φαίνονται και ο linker να έχει μόνο μια επιλογή

nm utility

- Το utility nm (name) μας δείχνει τα περιεχόμενα του **symbol table** σε ELF αρχεία (.o, a.out, libs)
  - Η εντολή 'man nm' μας δείχνει το manual page με τις πληροφορίες για το nm

```
$ gcc -Wall -pedantic -ansi -c main.c
$ nm -AS -td main.o
Offset     Segment    Name       Size(d)    Type       Bind
0000       BSS        _c         40         OBJT       GLOBAL
0000       DATA       _i         4          OBJT       GLOBAL
0004       DATA       _l         4          OBJT       LOCAL
0000       TEXT       _main      116        FUNC       GLOBAL
           UNDEF      _malloc               FUNC       GLOBAL
0040       BSS        _w         8          OBJT       LOCAL
```

- Αντίστοιχη πληροφορία μας δίνει και το utility objdump
  - Η εντολή 'man objdump' μας δείχνει το manual page με τις πληροφορίες για το objdump

```
$ gcc -Wall -pedantic -ansi -c main.c
$ objdump -t nm.o
nm.o:     file format elf32-i386
SYMBOL TABLE:
00000000 l     O .bss           00000028 c
00000028 l     O .bss           00000008 w
00000000 g     O .data          00000004 i
00000004 g     O .data          00000004 l
00000000 g     F .text          0000006f main
00000000         *UND*          00000000 malloc
```

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

main.c:
static char c[40];
static double w;

int i=13;
long l=2001;

int main(void){
  int j=3, *jp;
  jp = malloc(sizeof(j));
  c[5] = j;
  w = 2.0 * l;
  return 0;
}
```

Dynamic Linking

- Μέχρι τώρα, στο static linking όλος ο κώδικας περιέχεται στο εκτελέσιμο a.out
  - Το a.out μπορεί να τοποθετηθεί οπουδήποτε στη μνήμη (διεύθυνση P στα προηγούμενα)
  - Ο loader θα κάνει ένα πέρασμα στον RT όταν αντιγράφει το πρόγραμμα στη μνήμη και θα προσθέσει το P στα κατάλληλα σημεία
- Όπως είπαμε το dynamic linking επιδιώκει να έχει ένα μόνο αντίγραφο κάθε βιβλιοθήκης στη μνήμη
- Τι χρειάζεται για να πετύχουμε το dynamic linking? Δύο πράγματα:
  - (1) Η κάθε βιβλιοθήκη (ας πούμε κάθε code segment) να μπορεί να μπει σε μια θέση μνήμης που είναι άγνωστη όταν παράγεται το a.out
  - (2) Η κάθε βιβλιοθήκη πρέπει να περιέχει κώδικα που δουλεύει χωρίς να εξαρτάται από την θέση της βιβλιοθήκης στη μνήμη
- Το (1) είναι σχετικά εύκολο και λύνεται με μικρές αλλαγές στο static linking
  - Η διεύθυνση του κάθε code segment θα γίνει γνωστή κατά το loading στη μνήμη, όχι κατά το linking
  - Τα merge/relocate στάδια πρέπει να γίνουν από τον loader και όχι από τον linker
  - Αλλά περιλαμβάνουν τα ίδια βήματα/πράξεις όπως στο static linking
- Το (2) όμως είναι πιο δύσκολο
  - Αφορά όχι μόνο σε βιβλιοθήκες, αλλά σε οποιοδήποτε κομμάτι κώδικα καλείται από διαφορετικά προγράμματα. Για αυτό μιλάμε για shared objects (SO) – όχι μόνο βιβλιοθήκες.
  - Έστω ένα SO που περιέχει μια συνάρτηση f
  - Κατά ανάγκη, σε δύο διαφορετικά προγράμματα A και B, το SO, και επομένως και η f, θα βρίσκονται σε διαφορετικές διευθύνσεις. Γιατί;
    - Μια διεύθυνση που είναι ελεύθερη στο A μπορεί να είναι δεσμευμένη από άλλη βιβλιοθήκη/SO στο B
    - Ο linker/loader θα βάλει κάθε SO (code segment) όπου βρει ελεύθερες διευθύνσεις
- Αυτό δημιουργεί το εξής πρόβλημα: Πως όμως μπορεί να παραχθεί ο κώδικας που περιέχει η f, αν στα προγράμματα A, B διαφέρει η θέση της f στη μνήμη;
  - Π.χ. αν ο κώδικας της f περιέχει μια απόλυτη διεύθυνση, δεν μπορούμε να παράγουμε μια φορά τον κώδικα της f και να ικανοποιήσουμε και το πρόγραμμα A και το πρόγραμμα B.
- Αυτό είναι το πρόβλημα της δημιουργίας position independent code (PIC)
  - PIC = κώδικας που εκτελείται σωστά όπου και αν τοποθετηθεί στη μνήμη, χωρίς να χρειάζεται relocation
  - Τα shared objects πρέπει να περιέχουν μόνο PIC - διαφορετικά δεν θα εκτελούνται σωστά

Position Independent Code (PIC)

- Consider two lib functions f,g that appear in different addresses in two programs A and B
- Also consider that f calls g
- How can function f call g so that it works in both cases? E.g. the following will not work, because address g must have different values in A and B but the code of f is the same for A and B

```
f:      pushl "some arg for g"
        call g
```

  - This will be fine if we use relative addressing for g, but relative addressing is limited to a small part of the address space and absolute addresses are necessary.
- (read the rest of this slide only for your own interest, Bryant Ch. 7.12)
- Basis for PIC: Add an AT "Address Table" (AT) for each SO that contains one entry for each external symbol in the SO (In the GNU toolchain, AT is called GOT – Global Offset Table)
  - AT is accessed with relative addressing from SO code. When an SO is linked to an a.out, the SO is placed anywhere in memory, its AT is placed in the data segment of the SO, and the data segment follows the SO. Therefore, the relative address of AT, rel(AT), is always the same for all code in the SO, no matter where the SO is placed.
  - AT contains the absolute address of g in memory at a fixed offset, offset(g). There is one AT for each SO, so offset(g) is always the same (for all programs that use the SO).
  - When the SO is linked dynamically, the linker/loader writes in the AT the address of each (external) symbol
  - For all programs that use the SO (and f), the location rel(AT)+offset(g) is the same, but its contents will be different
  - To generate code for f we need to loads g's address to a register and then call g. In f above, conceptually we have

```
call g →    movl MEM[rel(AT)+offset(g)], %ebx
            call *%ebx
```

  - With PIC, the SO is compiled only once, there is no need for relocation, but AT needs to be filled during loading
  - To avoid filling in thousands of function addresses in AT that may not be used, there is a somewhat more complex way to access AT using a second table called PLT – procedure linkage table, which implements a process called lazy binding
- The more recent versions of gcc generate PIC for all functions/programs, unless specified otherwise
  - This allows placing all code and data (not only the stack) in arbitrary memory locations -> basis for a technique known as Address Space Layout Randomization (ASLR), which is used to reduce security vulnerabilities

Symbol Relocation during Loading

- With dynamic linking, the relocation process happens at load time:
- EST/SymTab is maintained in a.out until load time
- We place a.out and all shared objects in memory and determine their starting addresses
- We update symbol addresses in EST (similar to relocation during static linking)
- We scan RT in a.out and for each entry in RT we modify the code of a.out
- We fill in the AT with the addresses of the external symbols used by the shared objects (as part of PIC)
- We jump to function main

Objdump (και readelf) utility

- Μας δείχνει πληροφορίες για <u>όλα</u> τα περιεχόμενα ενός elf αρχείου, όχι μόνο τον Symbol Table
- Section header information
  - `gcc -static -Wall -pedantic -ansi main.c`
  - `objdump -h a.out`

```
a.out:     file format elf32-i386
Sections:
Idx Name      Size      VMA       LMA       File off  Algn
  2 .rel.plt  00000070  08048178  08048178  00000178  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .text     00062ef1  08049090  08049090  00001090  2**4 CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .data.rel.ro  00001914  080d96c0  080d96c0  000906c0  2**5 CONTENTS, ALLOC, LOAD, DATA
 18 .data     00000f40  080db060  080db060  00092060  2**5 CONTENTS, ALLOC, LOAD, DATA
 22 .bss      00000d3c  080dc340  080dc340  00093338  2**5 ALLOC
```

- Debugger information
  - `gcc -static -Wall -pedantic -ansi -g main.c`
  - `objdump -g a.out`
- Relocation information
  - `gcc -Wall -pedantic -ansi main.c`
  - `objdump -R a.out`

```
a.out:     file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0000400c R_386_JUMP_SLOT   malloc@GLIBC_2.0
00004010 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
```

- Όλα τα περιεχόμενα του text (ή οποιουδήποτε άλλου) segment
  - `gcc -static -Wall -pedantic -ansi main.c`
  - `objdump -sj .text a.out`

```
a.out:     file format elf32-i386
Contents of section .text:
8049090 83ec0cff 742420e8 34430100 83c410e8  ....t$ .4C......
80490a0 05000000 e8000000 005753e8 f0080000  .........WS.....
[...]
```

- Disassemble ένα εκτελέσιμο/object αρχείο
  - `gcc -static -Wall -pedantic -ansi main.c`
  - `objdump -Sj .text a.out`

```
a.out:     file format elf32-i386
Disassembly of section .text:
08049090 <__assert_fail_base.cold.0>:
 8049090:       83 ec 0c            sub    $0xc,%esp
 8049093:       ff 74 24 20         pushl  0x20(%esp)
 8049097:       e8 34 43 01 00      call   805d3d0 <__free>
 804909c:       83 c4 10            add    $0x10,%esp
 804909f:       e8 05 00 00 00      call   80490a9 <abort>
```

- Το utility readelf τυπώνει πληροφορίες για headers και symbols που περιέχονται σε elf αρχεία αλλά σε μεγαλύτερη λεπτομέρεια
  - `readelf -a a.out`

Reading

- (static linking) Bryant Ch. 7.{2,3,4,5,6,7,8,9}
- (dynamic linking) Bryant Ch. 7.{10,11,12,13}
- (tools) Bryant Ch. 7.{14}
- If interested, read this seminal paper that discusses many of the related concepts in a clear manner for a specific system (IBM System/360):
  - *Leon Presser and John R. White. 1972. Linkers and Loaders. ACM Comput. Surv. 4, 3 (Sept. 1972), 149–167. DOI:https://doi.org/10.1145/356603.356605*
- If still interested in more details check out Chapter I, Executable and Linkable Format (ELF) in the Linux referenced specs