

6. Instruction Set Architecture – Translation software: assembler, linker, loader, etc.

EECS 370 – Introduction to Computer Organization – Winter 2015

Robert Dick, Andrew Lukefahr, and Satish Narayanasamy

**EECS Department
University of Michigan in Ann Arbor, USA**

© Dick-Lukefahr-Narayanasamy, 2015

The material in this presentation cannot be
copied in any form without our written permission

Instruction Set Architecture (ISA) Design Lectures

- ❑ Lecture 2: Storage types and addressing modes
- ❑ Lecture 3 : LC-2K and ARM architecture
- ❑ Lecture 4 : Converting C to assembly – basic blocks
- ❑ Lecture 5 : Converting C to assembly – functions
- ❑ **Lecture 6 : Translation software; libraries, memory layout**

What happens when you call gcc?

1. C preprocessor

- ☐ Handles macros, #define, #ifdef, #if
- ☐ `gcc -E foo.c > foo.i` (foo.i contains preprocessed source code)

2. Compiler

- ☐ `gcc -S foo.c` (foo.s contains textual assembly)

3. Assembler

- ☐ `as foo.s -o foo.o` or `gcc -c foo.s`

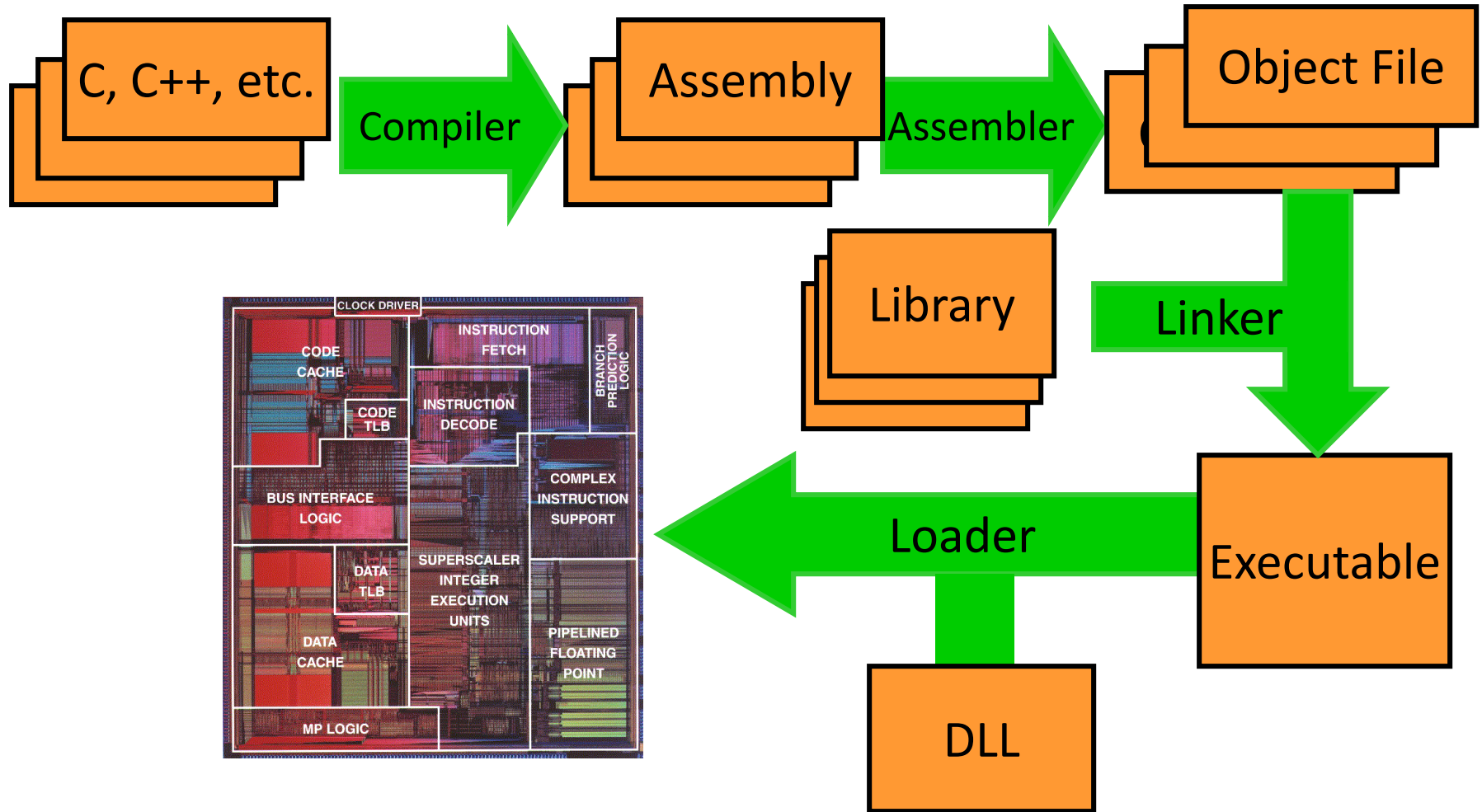
4. Linker

- ☐ `ld foo.o bar.o _bunch_of_other_stuff -o a.out`

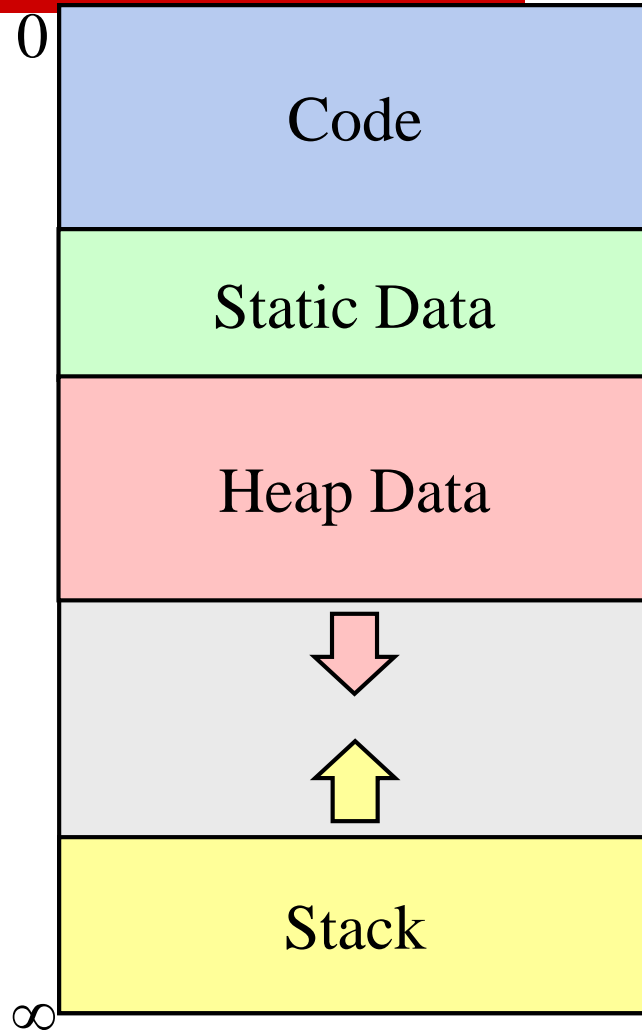
You can run `gcc -v` to see all the commands that it is running

- ☐ Note gcc does not call ld, it calls collect2, which is a wrapper that calls ld

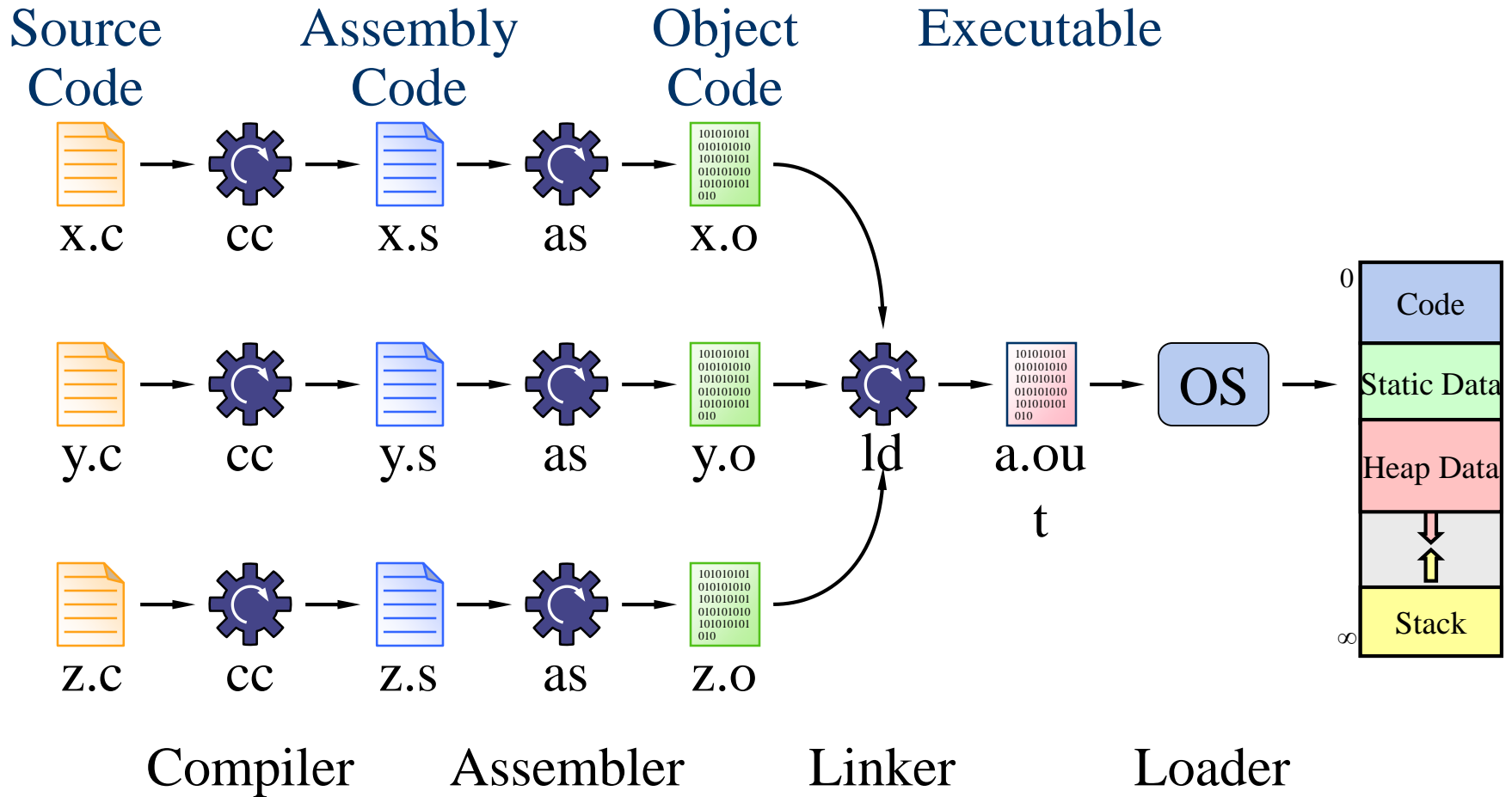
Source Code to Execution



Memory Layout for Process



Source to Process Translation





Example

main.c

```
extern float sin();
extern printf(), scanf();

main() {
    double x, result;
    printf("Type number: ");
    scanf("%f", &x);
    result = sin(x);
    printf("Sine is %f\n",
           result);
}
```

math.c

```
double sin(double x) {
    ...
}
```

stdio.c

```
FILE* stdin, stdout;

int printf(const char* format,
           ...) {
    ...
    fputc(c, stdout);
    ...
}

int scanf(const char* format,
           ...) {
    ...
    c = fgetc(stdin);
    ...
}
```



Object File #1

main.c

```
extern float sin();
extern printf(), scanf();

main() {
    double x, result;
    printf("Type number: ");
    scanf("%f", &x);
    result = sin(x);
    printf("Sine is %f\n",
           result);
}
```

*“Store the final location of sin
at offset 60 in the text section”*



Note: Header section is left
Out for simplicity

main.o

0	main:	text section
30	call printf	
52	call scanf	
60	call sin	
86	call printf	
0	_s1: "Type number: "	data section
14	_s2: "%f"	
17	_s3: "Sine is %f\n"	
	main T[0]	symbols
	_s1 D[0]	
	_s2 D[14]	
	_s3 D[17]	
	sin undef	
	print undef	
	scanf undef	
	printf T[30]	relocation
	printf T[86]	
	scanf T[52]	
	sin T[60]	
	_s1 T[24]	
	_s2 T[54]	
	_s3 T[80]	

Object File #2

stdio.c

```
FILE* stdin, stdout;

int printf(const char* format,
    ...) {
    ...
    fputc(c, stdout);
    ...
}

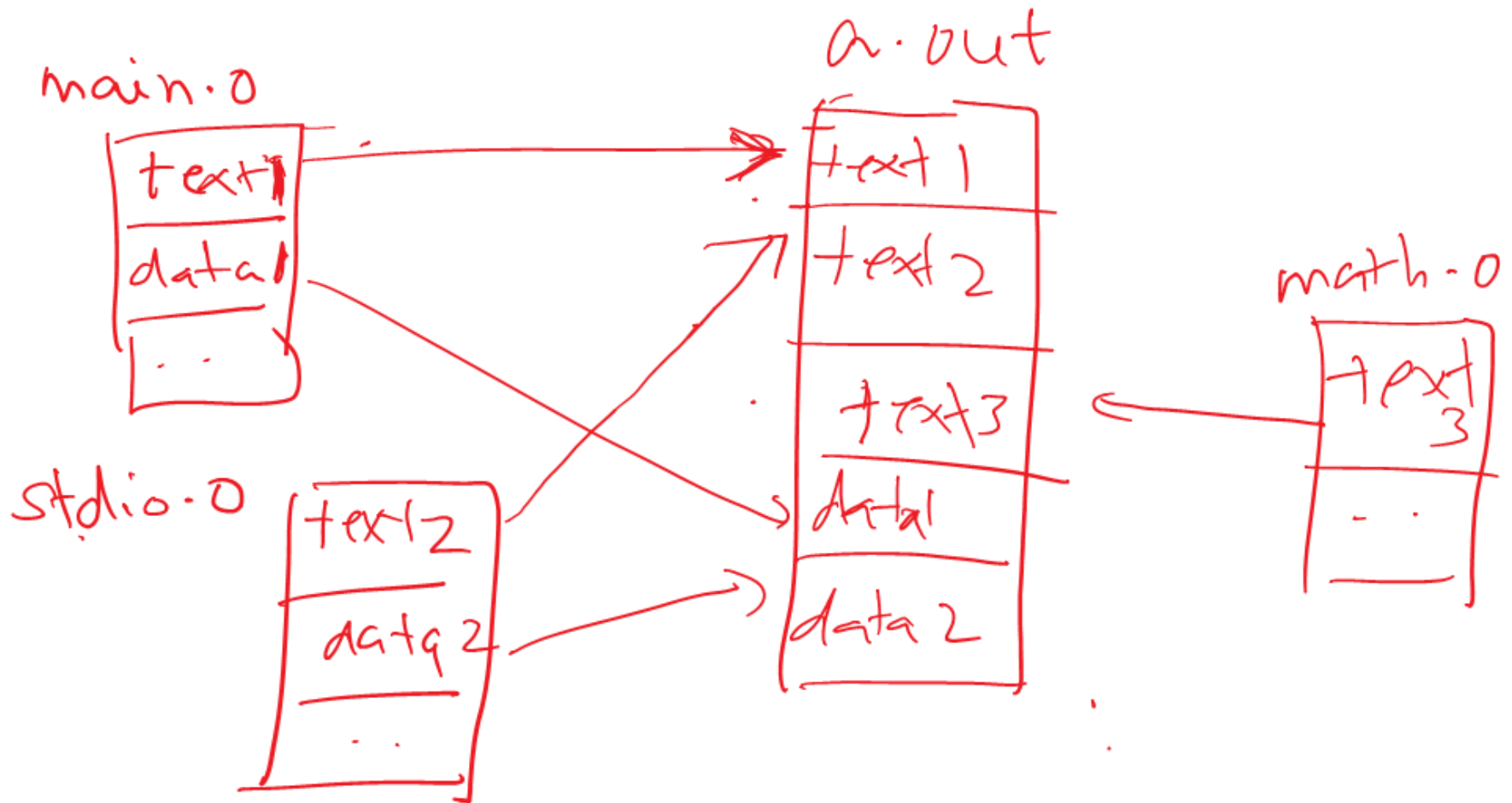
int scanf(const char* format,
    ...) {
    ...
    c = fgetc(stdin);
    ...
}
```

stdio.o

		text section
...		
44	printf:	
...		
118	load stdout	
...		
232	scanf:	
...		
306	load stdin	
...		
		data section
0	stdin:	
8	stdout:	
		symbols
printf	T[44]	
scanf	T[232]	
stdin	D[0]	
stdout	D[8]	
		relocation
stdout	T[118]	
stdin	T[306]	

*Note: Header section is left
Out for simplicity*

Relocation during linking



After linking

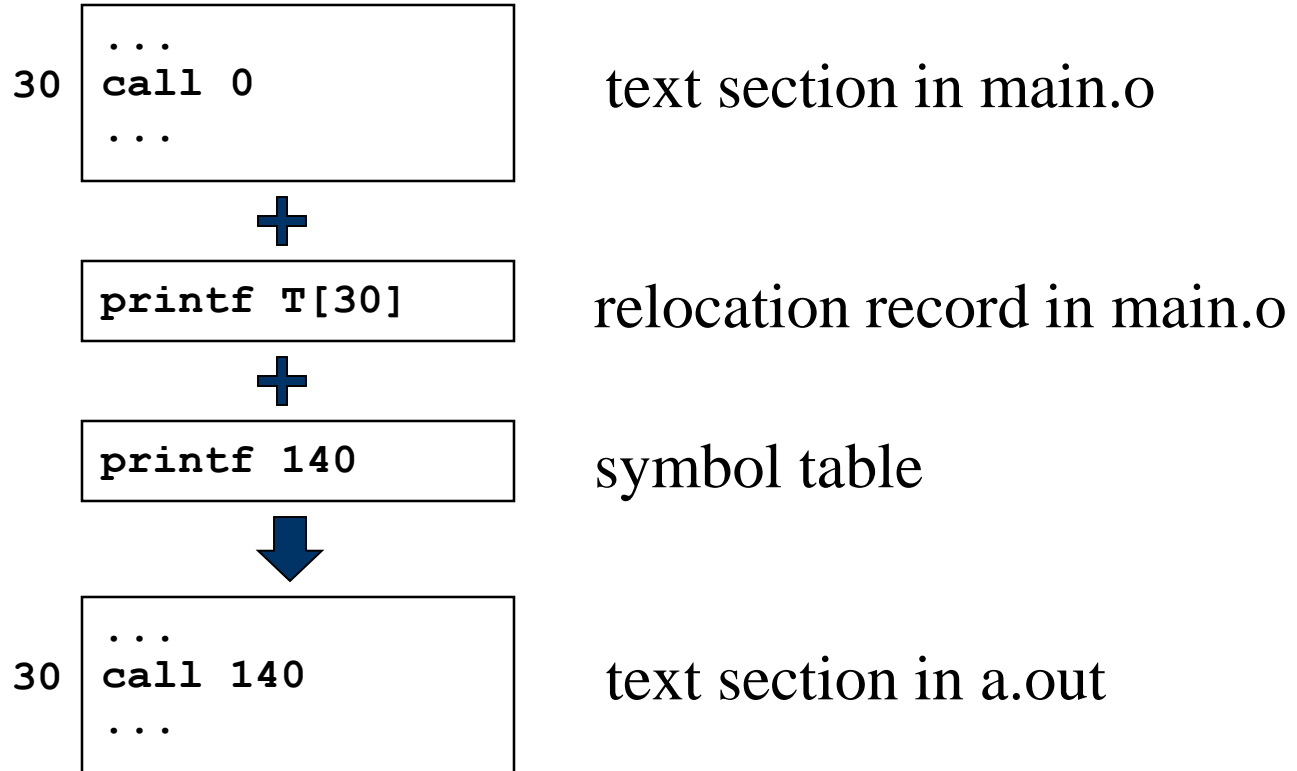
Memory map:

0	main.o text
96	stdio.o text
508	math.o text
720	main.o data
760	stdio.o data
836	

Symbol table:

Name	File	Sec	Offset	Addr
main:	main	T	0	0
_s1:	main	D	0	720
_s2:	main	D	14	734
_s3:	main	D	17	737
printf:	stdio	T	44	140
scanf:	stdio	T	232	328
stdin:	stdio	D	0	760
stdout:	stdio	D	8	768
sin:	math	T	0	508

Relocation



Class Problem 1

In the object file for file1 and file2, which symbols are in the symbol table?

file1.c

```
extern int bar(int);  
extern char c[];  
int a;  
int foo (int x) {  
    int b;  
    a = c[3] + 1;  
    bar(x);  
    b = 27;  
}
```

file2.c

```
extern double d[];  
char c[100];  
int bar (int y) {  
    char *e[100];  
    d[3] = (double)y;  
    c[20] = *e[7];  
}
```

Class Problem 2

file1.c

```
1 extern void  
2 bar(int);  
3 extern char c[];  
4 int a;  
5 int foo (int x) {  
6     int b;  
7     a = c[3] + 1;  
8     bar(x);  
9     b = 27;  
}
```

file2.c

```
extern double d[];  
char c[100];  
void bar (int y) {  
    char *e[100];  
    d[3] = (double)y;  
    c[20] = *e[7];  
}
```

- A) What if file2.c contains extern int j, but no reference to j?
- B) What if variable 'e' is static?
- C) What if the externs in file1.c are deleted?
- D) Which source lines require relocation during linking?

Class Problem 3

Draw out the contents of the object files for file1 and file2, including instruction and data addresses, along with the contents of the symbol tables and relocation tables

file1.s

```
int a;
extern int e;
extern int bar();
void foo(int b)
{
    int d[20];
    ldr a;

L1: ldr d;
    ldr e;
    cmp a, e
    beq L1;
    bl bar
}
```

file2.s

```
int e;
extern int a;
void bar()
{
    static int f;
    int g;
    ldr f;
    ldr a;
    add g, g, #1
    bl printf
}
```


Answer to Problem 3

Header	Name	file1.s	
	Text size	0x14	
	Data size	0x04	
Text	Address	Instruction	
	0	lw a, 0 (\$gp)	
	4	lw d ...(\$sp)	
	8	lw e ...(\$gp)	
	12	beq a e -12	
	16	jal bar	
Data	0	a	
Symbol table	Label	Address	
	a	0	
	e	-	
	foo	0	
	bar	-	
Reloc table	Addr	Instruction type	Dependency
	0	lw	a
	8	lw	e
	16	jal	bar

Header	Name	file2.s
	Text size	0x10
	Data size	0x08

Text	Address	Instruction
	0	lw f, 4(\$gp)
	4	lw a X(\$gp)
	8	lw g ...(\$sp)
	12	add g g 1
	16	sw g ...(\$sp)
	20	jal printf

Data	0	e
	4	f

Symbol table	Label	Address
	e	0
	f	4
	a	-
	bar	0
	printf	-

Reloc table	Addr	Instruction type	Dependency
	0	lw	f
	4	lw	a
	20	jal	printf

Class Problem 4

Show the contents of the resulting executable file when file1.o and file2.o are linked.

file1.s

```
int a;
void foo(int b)
{
    int d[20];
    ldr a;

L1: ldr d;
    ldr e;
    cmp a, e
    beq L1;
    bl bar
}
```

file2.s

```
int e;
void bar()
{
    static int f;
    int g;
    ldr f;
    ldr a;
    add g, g, #1
    bl printf
}
```

Answer to Problem 4

Header	Text size	0x24
	Data size	0x0C
Text	Address	Instruction
	0x0040 0000	lw \$a0, 0x0000 (\$gp)
	0004	lw d ...(\$sp)
	0008	lw \$a1 0x0004 (\$gp)
	000C	beq \$a0 \$a1 -12
	0010	jal 0x400014
	0014	lw \$a2, 0x0008 (\$gp)
	0018	lw \$a3 0x0000 (\$gp)
	001C	lw g ...(\$sp)
	0020	add g g 1
	0024	sw g ...(\$sp)
	0028	jal xxxx
Data	0x1000 0000	a
	0x1000 0004	e
	0x1000 0008	f

\$gp = 0x1000 0000

Following slides describe the following in detail:

- Linux (ELF) object file format**
- Linking process**

Linux (ELF) object file format

Object files contain more than just machine code instructions!

Header: (of an object file) contains sizes of other parts

Text: machine code

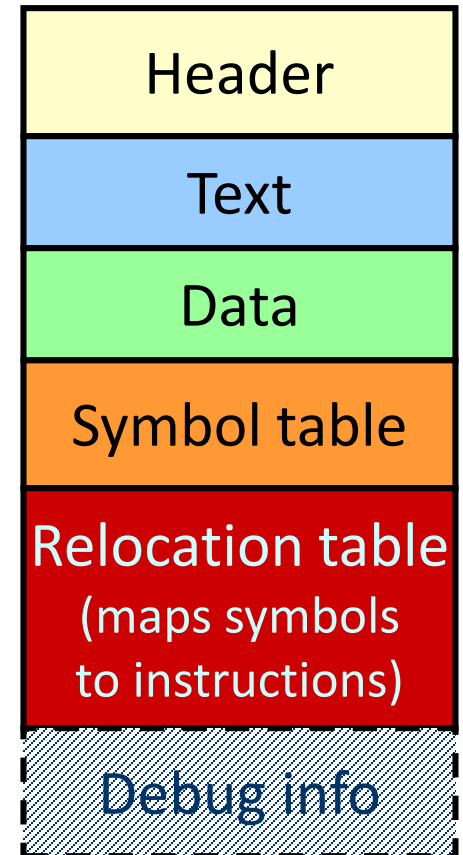
Data: global and static data

Symbol table: symbols and values

Relocation table: references to addresses that may change

Debug info: mapping of object back to source (only exists when debugging options are turned on)

Object code format

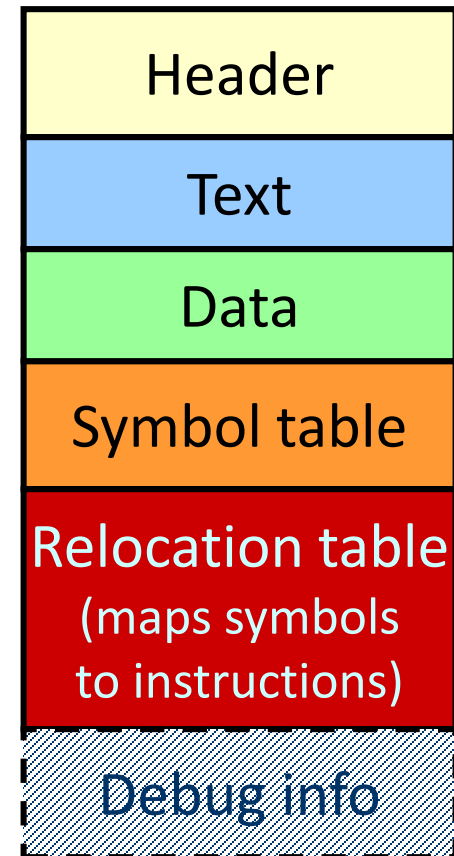


Linux (ELF) object file format (2)

Object code format

Header

- size of other pieces in file
- size of text segment
- size of static data segment
- size of uninitialized data segment
- size of symbol table
- size of relocation table



Linux (ELF) object file format (3)

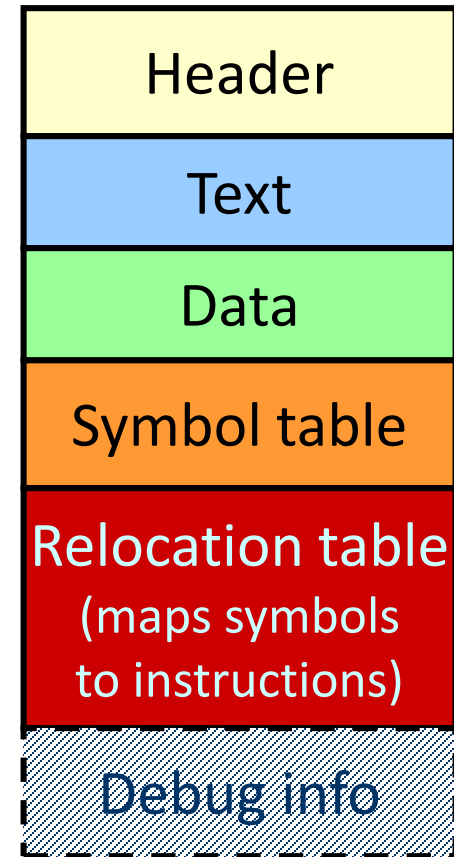
Text segment

- machine code

By default this segment is assumed to be read-only and that is enforced by the OS



Object code format



Linux (ELF) object file format (4)

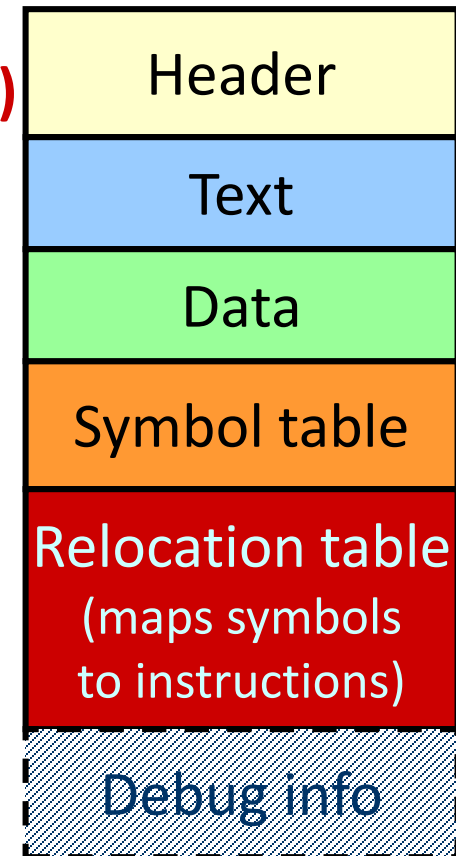
Data segment (Initialized static segment)

- values of initialized globals
- values of initialized static locals



Doesn't contain uninitialized data.
Just keep track of how much memory is
needed for uninitialized data

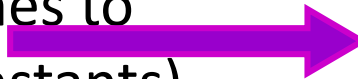
Object code format



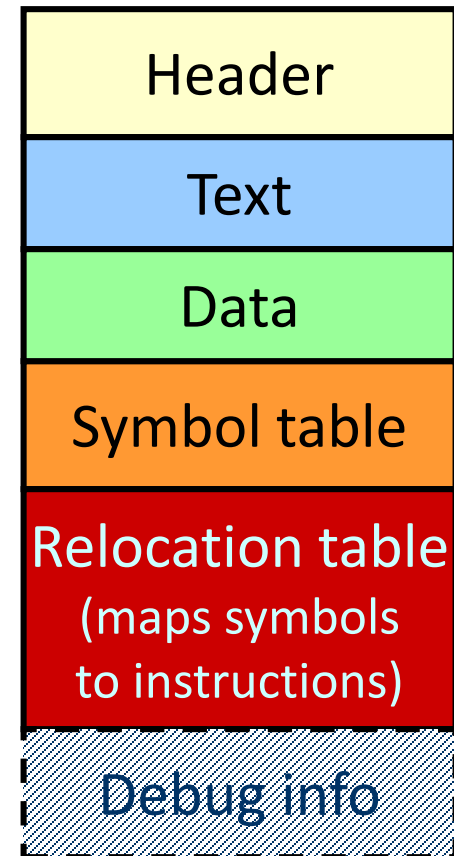
Linux (ELF) object file format (5)

Symbol table:

- It is used by the linker to bind public entities within this object file (function calls and globals)
- Maps string symbol names to values (addresses or constants)
- Associates addresses with global labels. Also lists unresolved labels



Object code format



Linux (ELF) object file format (6)

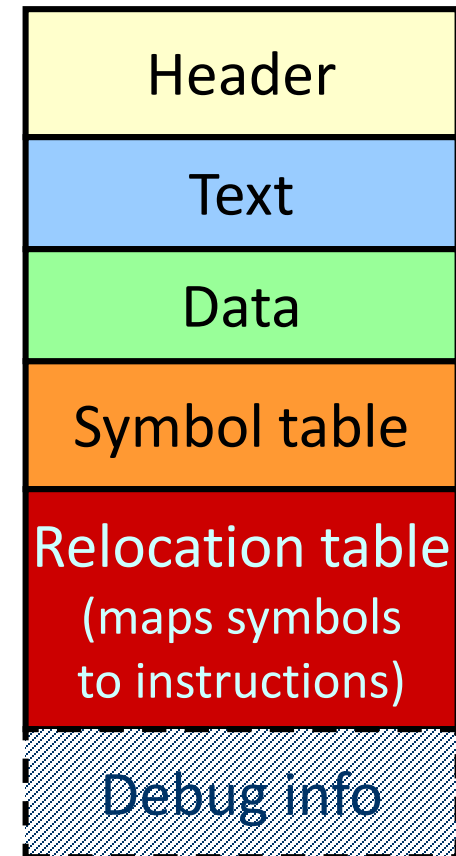
Relocation table :

identifies instructions and data words that rely on absolute addresses. These references must change if portions of program are moved in memory

Used by linker to update symbol uses (e.g., branch target addresses)



Object code format



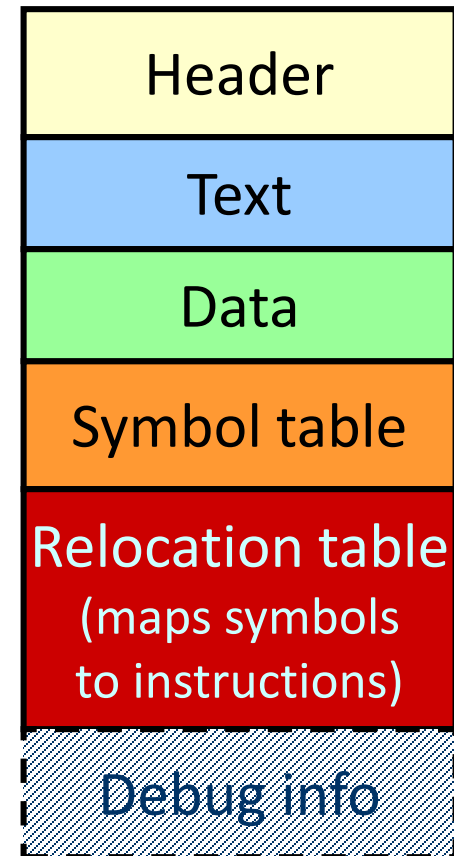
Linux (ELF) object file format (7)

Debug info (optional) :

Contains info on where variables are in stack frames and in the global space, types of those variables, source code line numbers, etc. Debuggers use this information to access debugging info at runtime



Object code format



Linker

- ❑ Stitches independently created object files into a single executable file (i.e., a.out)
 - Step 1: Take text segment from each .o file and put them together.
 - Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- ❑ What about libraries?
 - Libraries are just special object files.
 - You create new libraries by making lots of object files (for the components of the library) and combining them (see ar and ranlib on Unix machines).
- ❑ Step 3: Resolve cross-file references to labels
 - Make sure there are no undefined labels

Linker - continued

- ❑ Determine the memory locations the code and data of each file will occupy
 - Each function could be assembled on its own
 - Thus the relative placement of code/data is not known up to this point
 - Must relocate absolute references to reflect placement by the linker
 - PC-Relative Addressing (beq, bne): never relocate
 - Absolute Address (mov r15, X): always relocate
 - External Reference (usually bl): always relocate
 - Data Reference (often movw/movt): always relocate

- ❑ Executable file contains no relocation info or symbol table
these just used by assembler/linker

Linker - continued

- ❑ Linker assumes first word of first text segment is at fixed address
- ❑ Linker knows:
 - Length of each text and data segment
 - Ordering of text and data segments
- ❑ Linker calculates:
 - Absolute address of each label to be jumped to (internal or external) and each piece of data being referenced
- ❑ To resolve references:
 - Search for reference (data or label) in all symbol tables
 - If not found, search library files (for example, for printf)
 - Once absolute address is determined, fill in the machine code appropriately

Trends in software systems

- ❑ Programmers are expensive
- ❑ Applications are more sophisticated
 - 3D graphics, streaming video, etc
- ❑ Application programmers rely more on library code to make high quality apps while reducing development time
 - This means that more of the executable is library code
 - Why not keep those shared library routines in memory and link an object file at load time? (DLLs)
 - Executable files are smaller (not very important)
 - Updating library routines is easy (e.g., voice activated menus)
- ❑ Porting code to a variety of platforms is costly/time-intensive
 - Utilize virtual instruction sets (e.g., Java bytecode, C# CLR) and VMs
 - “Write once, run everywhere”

Things to remember

- ❑ Compiler converts a single source code file into a single assembly language file
- ❑ Assembler removes pseudos, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each `.s` file into a `.o` file
- ❑ Assembler does 2 passes to resolve addresses, handling internal forward references
- ❑ Linker combines several `.o` files and resolves absolute addresses
- ❑ Linker enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- ❑ Loader loads executable into memory and begins execution

Next Topic

- ❑ We've been looking at the instruction set architecture
 - How to tell the processor what to do

- ❑ Next topic is the real **hardware!!!**
 - Basic hardware building blocks
 - Architecture of a simple processor
 - Read Chapter 3.1 – 3.3



Slides not used in class

Compiler

- ❑ C, C++, etc... → assembly code
- ❑ 2 major parts
 - Frontend
 - Parsing C syntax
 - Source-level analysis
 - Backend
 - Machine independent assembly → optimizations
 - Binding variables/instructions to processor resources
- ❑ Relevant courses
 - EECS 483 (Compiler Construction)
 - EECS 583 (Advanced Compilers)

Assembler

- ❑ Converts assembly (.s file) to machine code or object file (.o file)
- ❑ Generally a simple translation
 - Most assembly instructions map to 1-1 to machine code instructions
 - Pseudo-instructions map to 1 or more machine code instructions
 - Easy to write assembly program with them
 - e.g. ARM pseudo-instr.: “nop” --- assembler translates it to “mov r0, r0”
 - Assembler directives tell the assembler to do something
 - Allocate space for a variable/array/constants
 - Associate a symbol with a value in the symbol table
 - #define variables/labels/registers to convenient names

Assembly Issues

- ❑ Reference to a label in another file
 - Data or jump address
 - Only global labels can be referenced from another file

- ❑ Assume start at fixed address (say 0) for each procedure
 - But what about multiple procedures?

- ❑ Machine code produced by assembler is **NOT** executable!

Assembly → Object file - example

Snippet of C

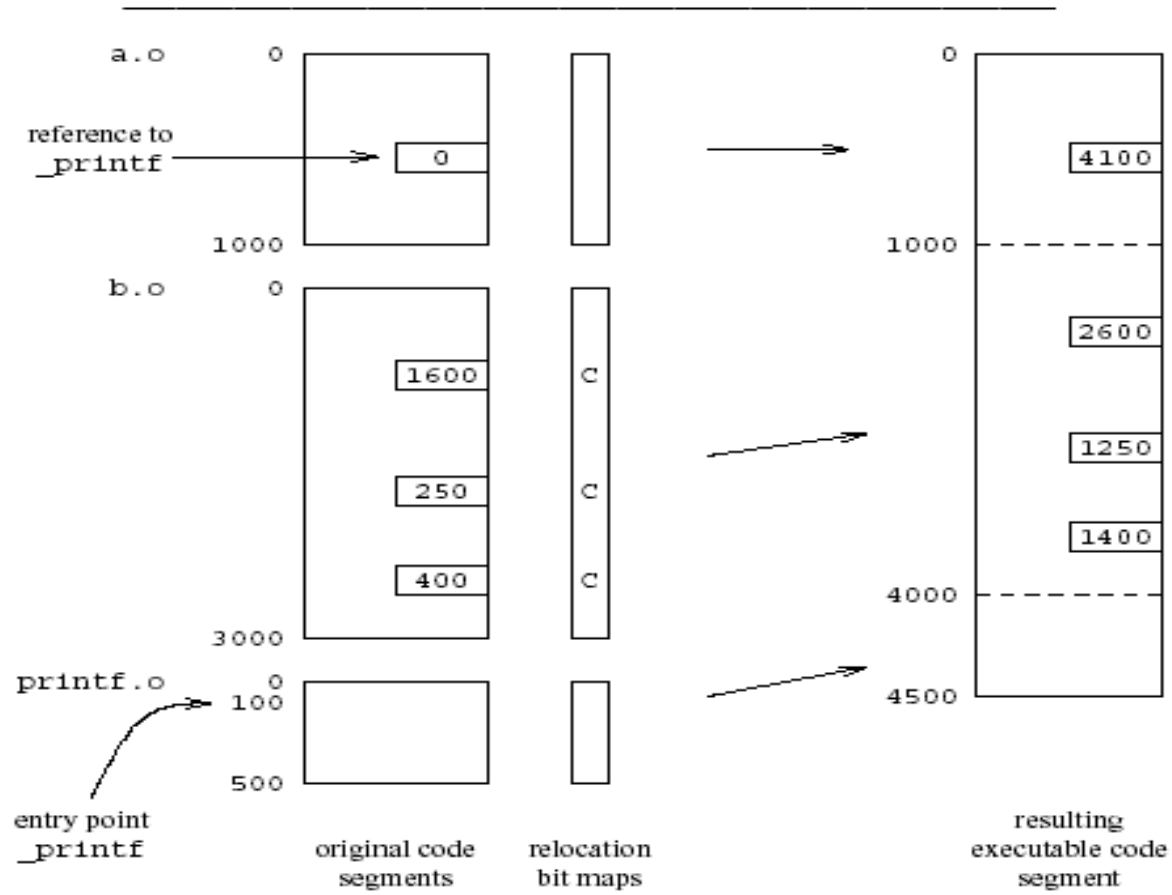
```
int X = 3;
main() {
    ... = X;
    B();
    ...
}
```

Snippet of assembly code

```
movw r1, #.lower16:X
movt r1, #.upper16:X
ldr r0, [r1,#0]
bl B
```

Header	Name	foo	
	Text size	0x100	
	Data size	0x20	
Text	Address	Instruction	
	0	movw r1, #:lower16:0	
	4	movt r1, #:upper16:0	
	8	ldr r0, [r1, #0]	
	12	bl B	
Data	0	X	3
	...		
Symbol table	Label	Address	
	X	0	
	B	-	
	main	0	
Reloc table	Addr	Instruction type	Dependency
	0	movw_lower16	X
	4	movt_upper16	X
	12	bl	B

Example



Example Executable File

Header	Text size	0x200
	Data size	0x40
Text	Address	Instruction
	0x0040 0000	movw r1, #0000
	0x0040 0004	movt r1, #1000
	0x0040 0008	ldr r1, [r0, #0]
	0x0040 000c	bl 0x400100
	...	
	0x0040 0100	sub r13, r13, #20
Data	0x0040 0104	bl 0x400200
Data	0x1000 0000	X
	...	