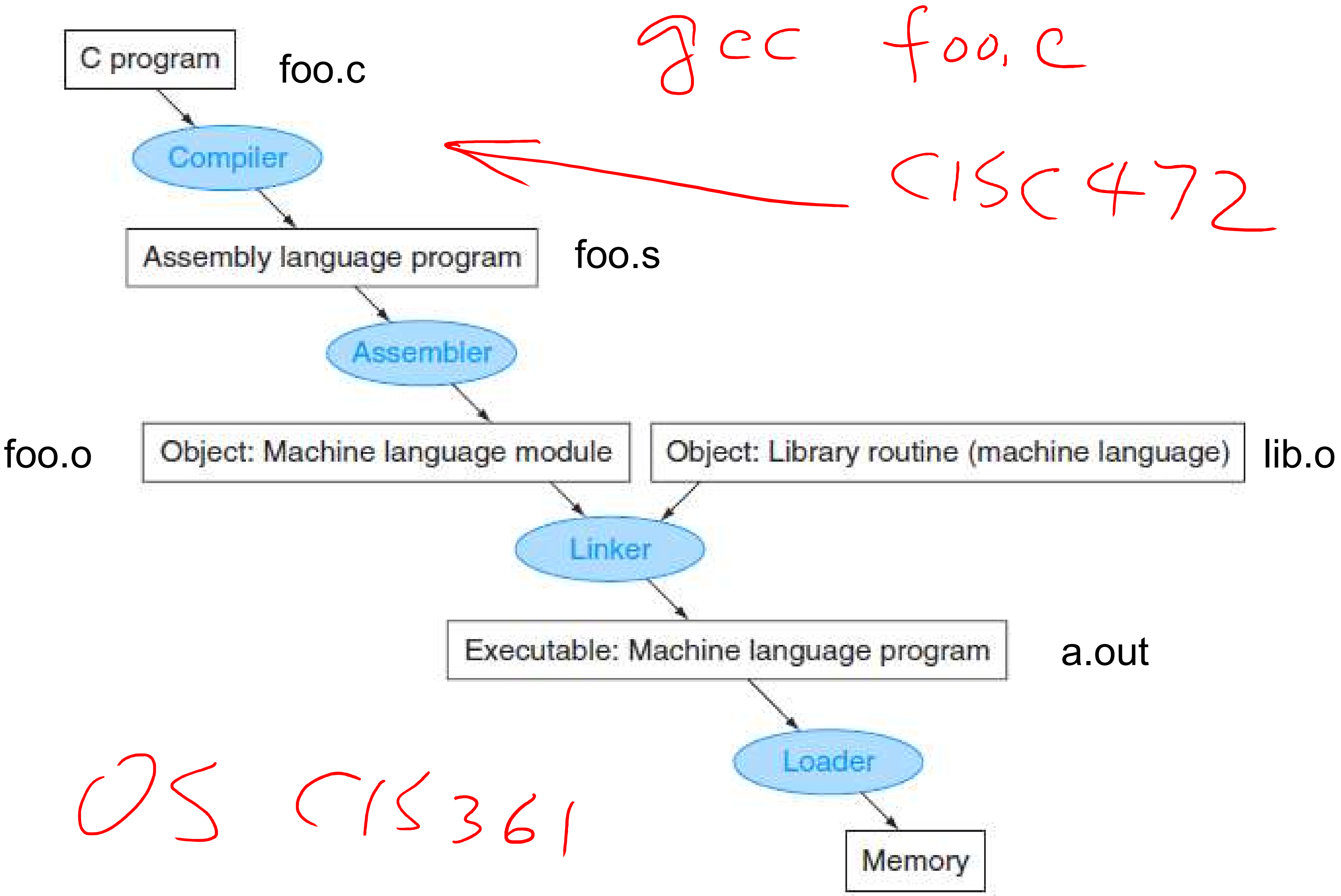
CISC260 Machine Organization and Assembly Language

Assembler, Linker & Loader



C program

foo.c

Compiler

gcc foo.c

cisc472

Assembly language program

foo.s

Assembler

foo.o

Object: Machine language module

Object: Library routine (machine language)

lib.o

Linker

Executable: Machine language program

a.out

Loader

OS CIS361

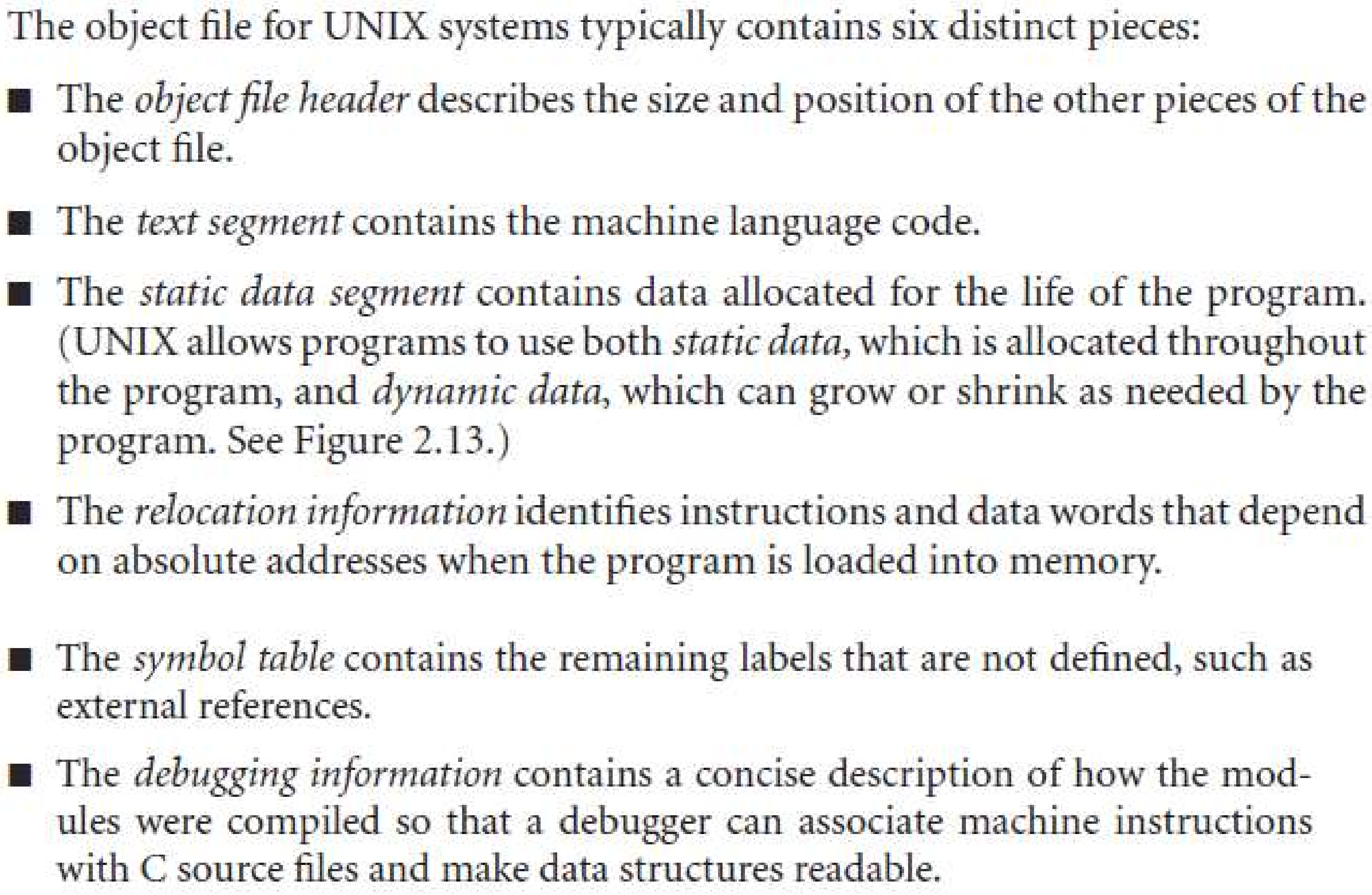
Memory

ADD r0, r1, r2

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, 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.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.

BLT Else

Assembler: assembly code ➔ machine code



The object file for UNIX systems typically contains six distinct pieces:

- The *object file header* describes the size and position of the other pieces of the object file.
- The *text segment* contains the machine language code.
- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use both *static data*, which is allocated throughout the program, and *dynamic data*, which can grow or shrink as needed by the program. See Figure 2.13.)
- The *relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The *symbol table* contains the remaining labels that are not defined, such as external references.
- The *debugging information* contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

Pseudoinstructions => machine instructions

e.g.,

```
LDR r0, =0x12345678     =>     LDR r0, [pc, #8]
                               …..
                               …..
                               …..
                               0x12345678
```

2 passes to resolve addresses

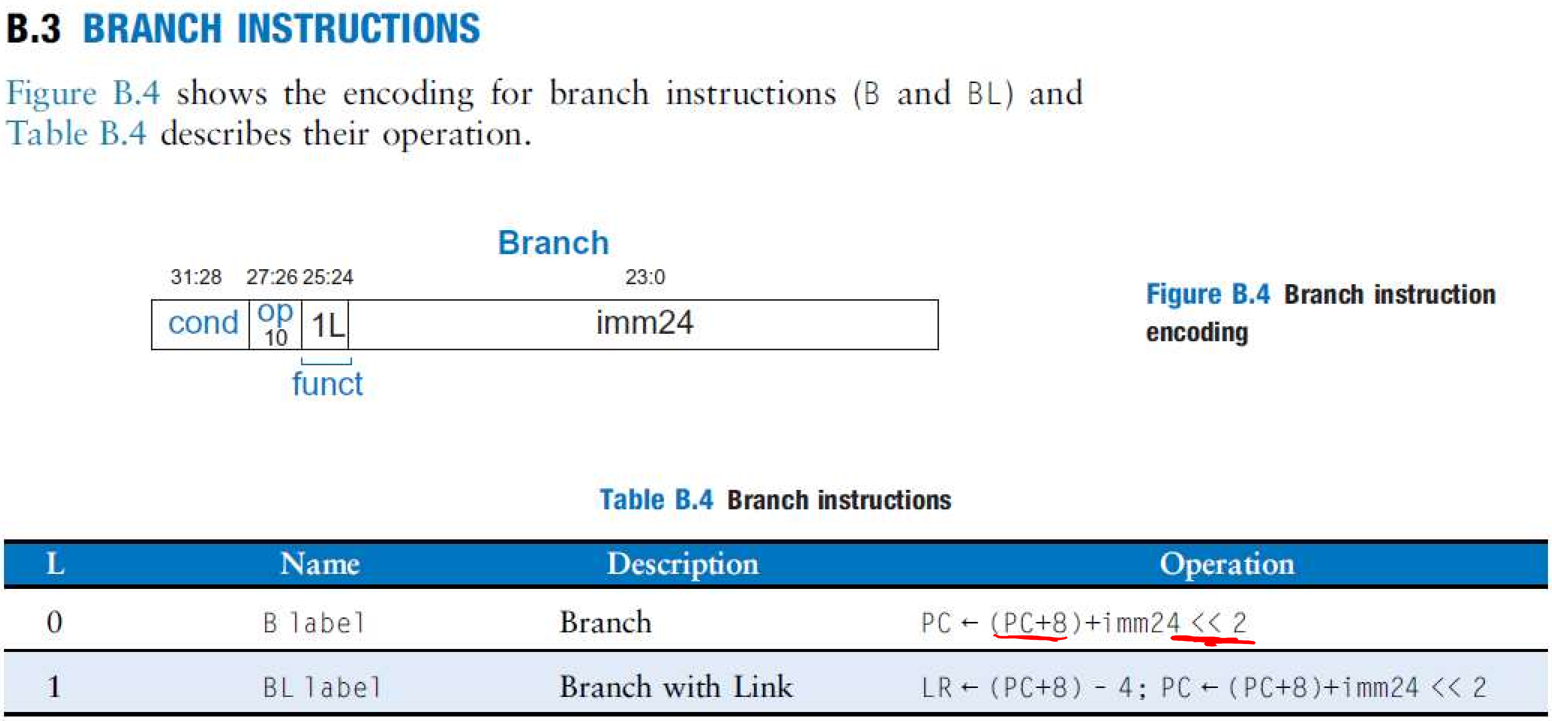
e.g.,

```
        CMP     r0, r1
        BLT     Else
        ….
        ….
        ….
        ….
        ….
Else:   ADD     r0, r0, #1
```

Pass1: assign address to labels

```
0x0000 1000     CMP     r0, r1
0x0000 1004     BLT     Else
0x0000 1008     ….
0x0000 100C     ….
0x0000 1010     ….
0x0000 1014     ….
0x0000 1018     ….
0x0000 101C     ADD     r0, r0, #1
```

symbol	**address**
Else	0x0000101C



B.3 BRANCH INSTRUCTIONS

Figure B.4 shows the encoding for branch instructions (B and BL) and Table B.4 describes their operation.

Branch

31:28	27:26	25:24	23:0
cond	op 10	1L	imm24
		funct	

Figure B.4 Branch instruction encoding

Table B.4 Branch instructions

L	Name	Description	Operation
0	B label	Branch	PC ← (PC+8)+imm24 << 2
1	BL label	Branch with Link	LR ← (PC+8) - 4; PC ← (PC+8)+imm24 << 2

PC-relative addressing: Imm24 = # of words that the branch label is away from PC+8

2 passes to resolve addresses

Pass1: assign address to labels

address:	assembly code	
0x0000 1000	CMP	r0, r1
0x0000 1004	BLT	Else
0x0000 1008	….	
0x0000 100C	….	
0x0000 1010	….	
0x0000 1014	….	
0x0000 1018	….	
0x0000 101C	ADD	r0, r0, #1

Pass2: translate to machine code

address:	machine code
0x0000 1000	E1500001
0x0000 1004	BA000004
0x0000 1008	….
0x0000 100C	….
0x0000 1010	….
0x0000 1014	….
0x0000 1018	….
0x0000 101C	E2800001

symbol	**address**
Else	0x0000101C

signed_immed_24

= [target address – (pc+8)] / 4

= [0x0000101C – (0x00001004 + 8)]/4

= [0x0000101C – 0x0000100C] / 4

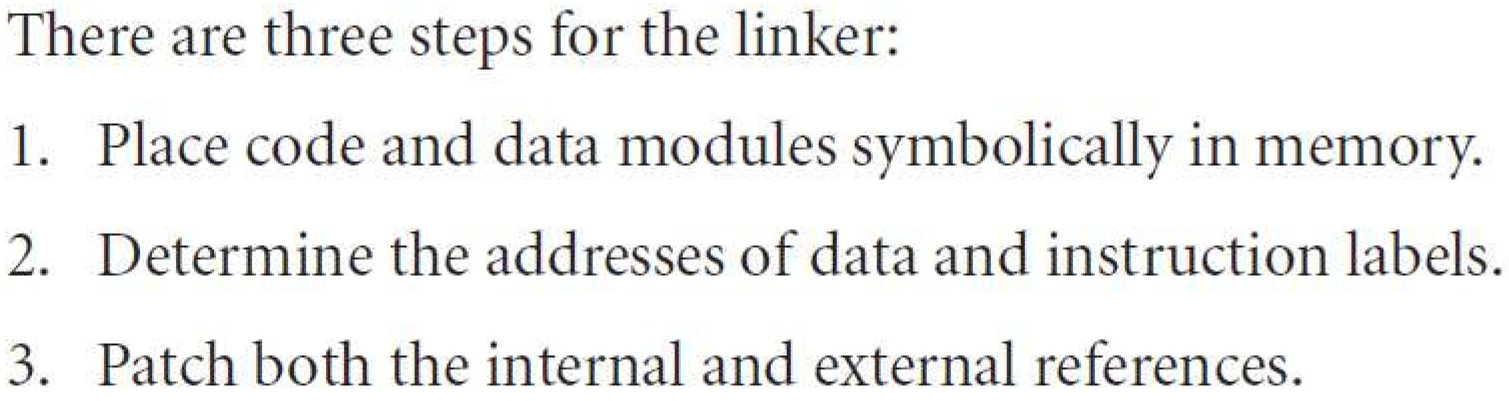
= 0x00000010 / 4

= 0x00000004

Relocation: address change during link

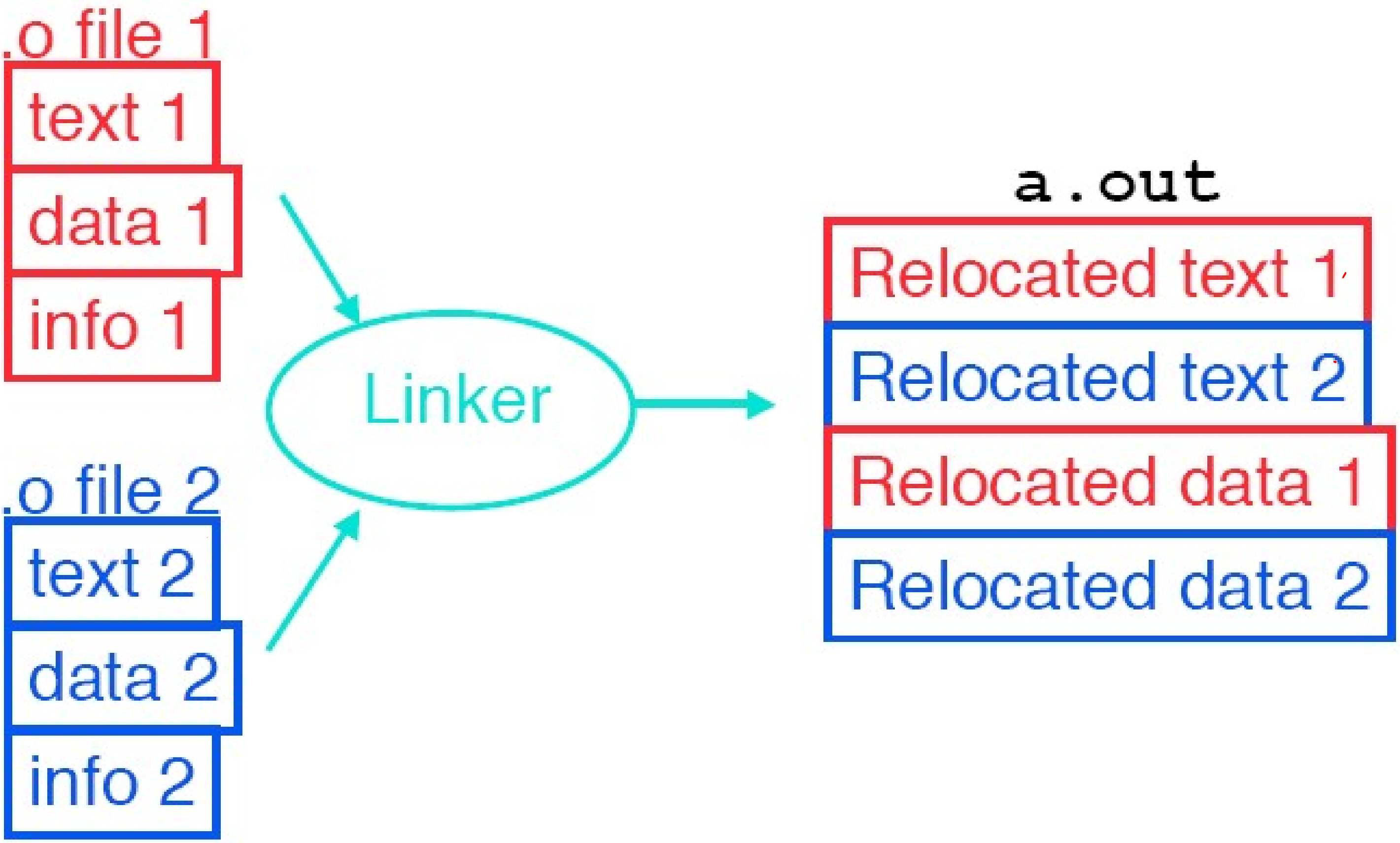
```
.text
…
LDR      r0, =myData                   @ data will be loaded in a different
LDR      r1, [r0, #4]                  @ memory segment
…
…

BL       subroutine                    @ subroutine may be in a separate file
...
…



.data
myData: .word 10, 20
```

Linker is a systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file. **Executable.**



There are three steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.



.o file 1

text 1

data 1

info 1

.o file 2

text 2

data 2

info 2

Linker

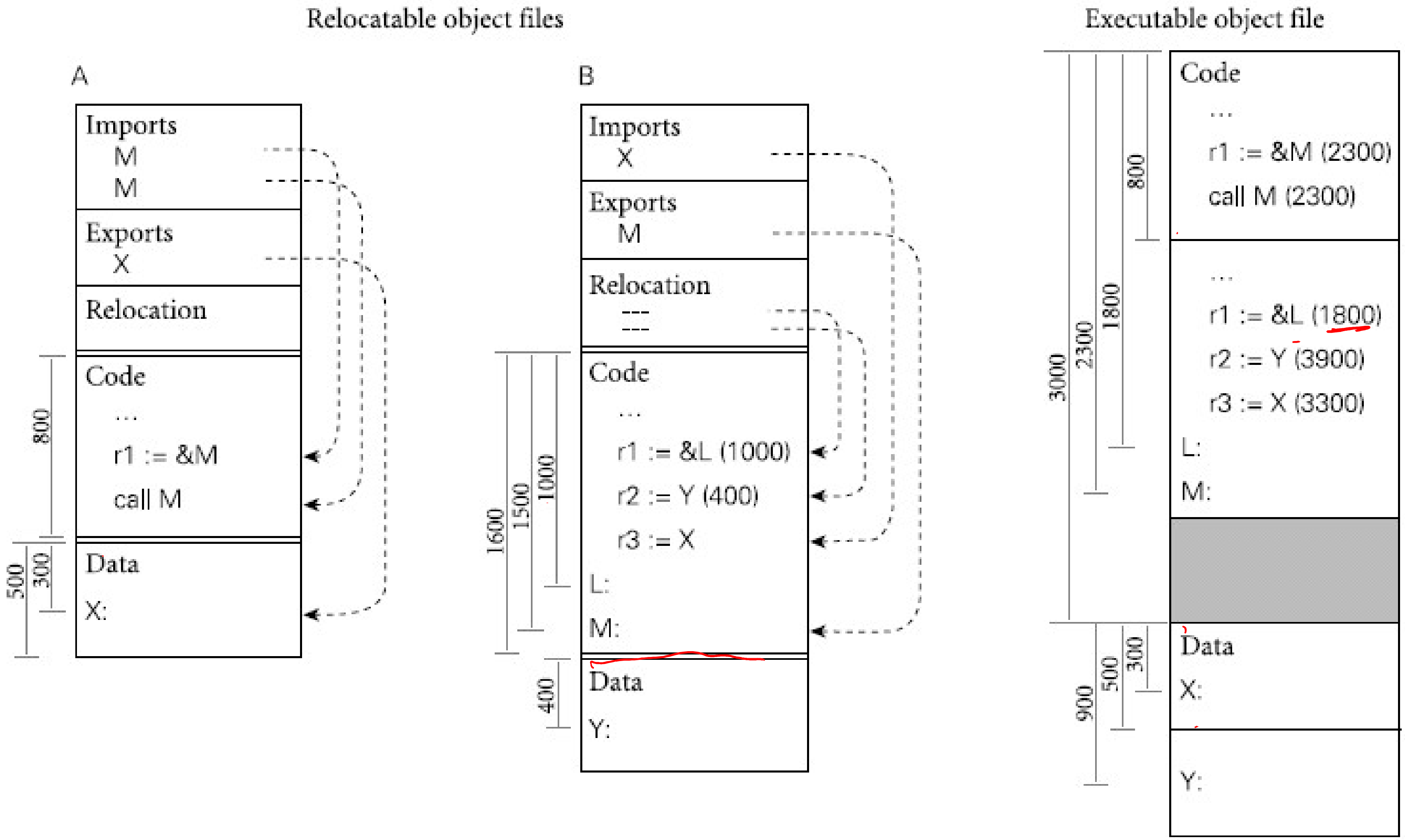
a.out

Relocated text 1

Relocated text 2

Relocated data 1

Relocated data 2



Relocatable object files

A

Imports
M
M

Exports
X

Relocation

Code
...
r1 := &M
call M

Data
X:

B

Imports
X

Exports
M

Relocation

Code
...
r1 := &L (1000)
r2 := Y (400)
r3 := X

L:
M:

Data
Y:

Executable object file

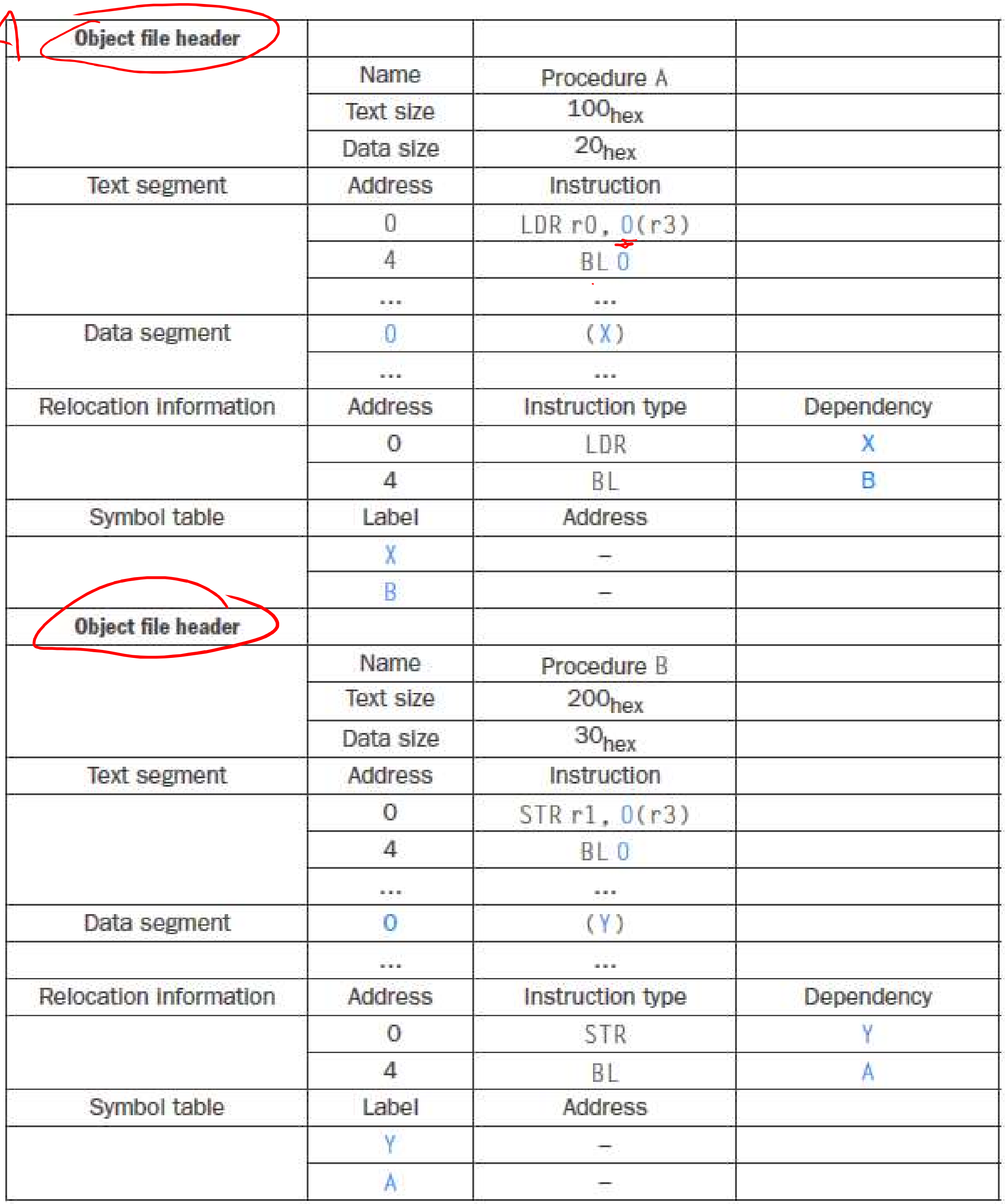
Code
...
r1 := &M (2300)
call M (2300)

...
r1 := &L (1800)
r2 := Y (3900)
r3 := X (3300)

L:
M:

Data
X:

Y:



A

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`LDR r0, 0(r3)`	
	4	`BL 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	LDR	X
	4	BL	B
Symbol table	Label	Address	
	X	–	
	B	–	

B

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`STR r1, 0(r3)`	
	4	`BL 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	STR	Y
	4	BL	A
Symbol table	Label	Address	
	Y	–	
	A	–	

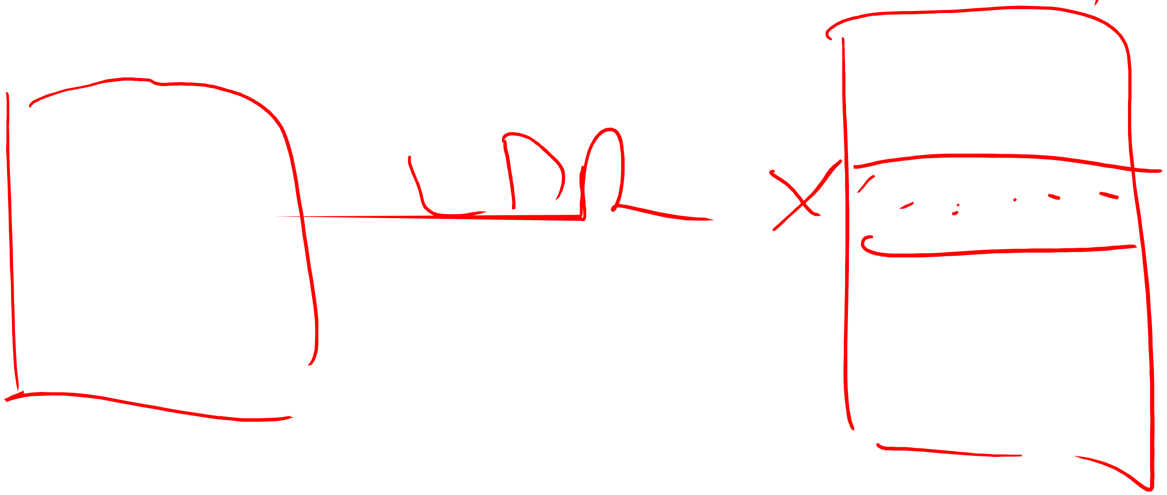
LDR r0, =X

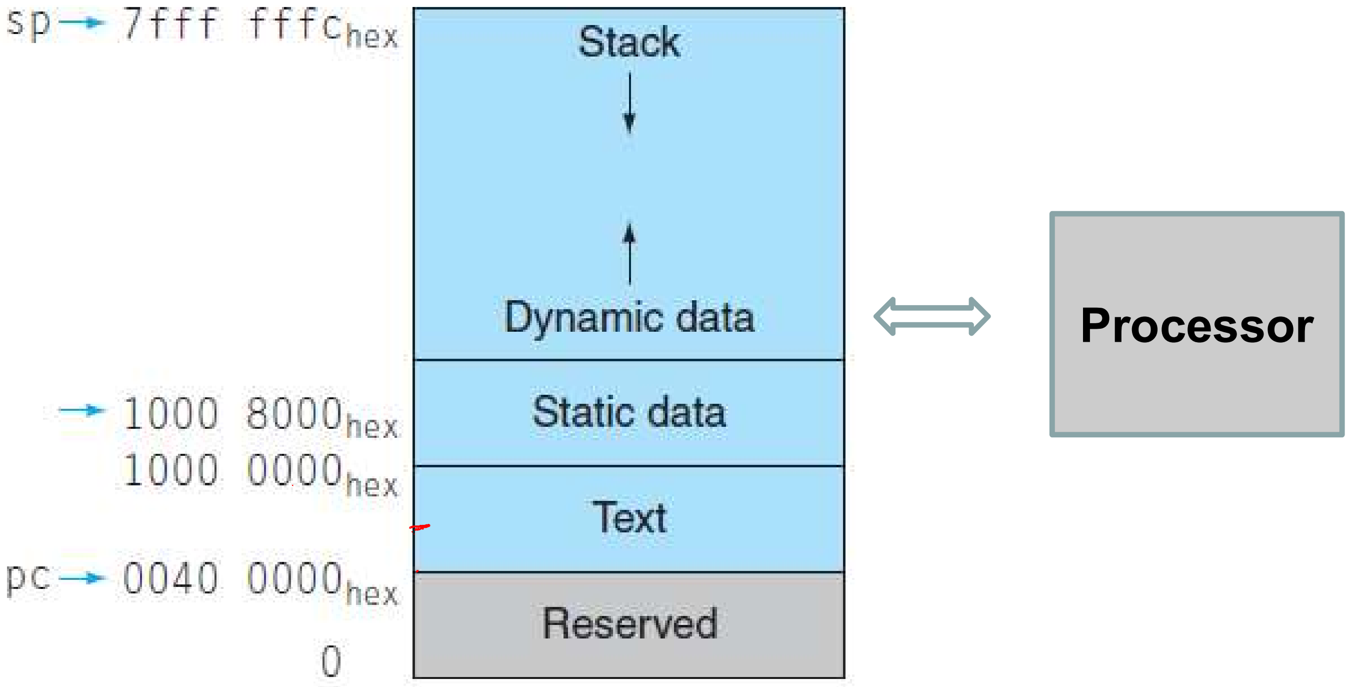
LDR r0, [r0]

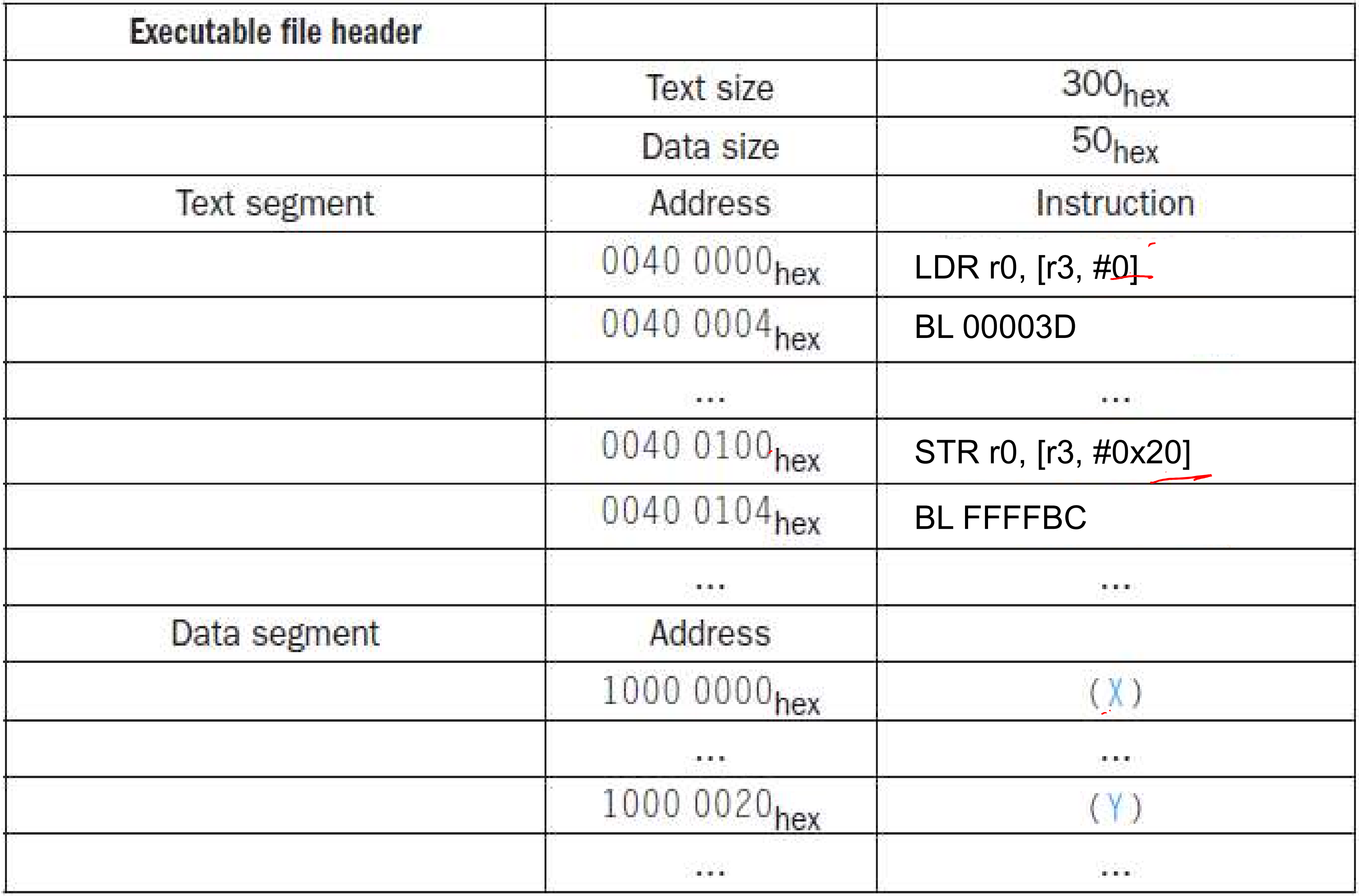
.asciz

s	exponent	fraction
0	11111111	0000

$(-1)^s\ 1.\text{fraction} \times 2^{\boxed{\ }}$ nen $\times 2^{\boxed{\ }}$




sp → 7fff fffc$_{hex}$
Stack
Dynamic data
→ 1000 8000$_{hex}$
Static data
1000 0000$_{hex}$
Text
pc → 0040 0000$_{hex}$
Reserved
0
Processor



Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	LDR r0, [r3, #0]
	$0040\ 0004_{hex}$	BL 00003D
	...	...
	$0040\ 0100_{hex}$	STR r0, [r3, #0x20]
	$0040\ 0104_{hex}$	BL FFFFBC
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

The lower 24 bit in a BL instruction, signed_immed_24, is calculated as:
signed_immed_24 = [target_address – (pc+8)] / 4.

The first BL is the procedure A's call to procedure B.
So, target_address = 0x0040 0100, and pc is the address of the BL instruction, which is 0x0040 0004. Therefore, signed_immed_24
= [0x0040 0100 – (0x0040 0004 +8)] / 4
= 0x0000 00F4 / 4
= 0x 00 003D

The second BL is the procedure B's call to procedure A.
So, target_address = 0x0040 0000, and pc is the address of the BL instruction, which is 0x0040 0104. Therefore, signed_immed_24
= [0x40 0000 – (0x40 0104 +8)] / 4
= 0xFF FEF4 / 4
= 0xFF FFBD

Resolving References (1/2)

- Linker *assumes* first word of first text segment is at address 0x00000000.

  (More on this later when we study "virtual memory")

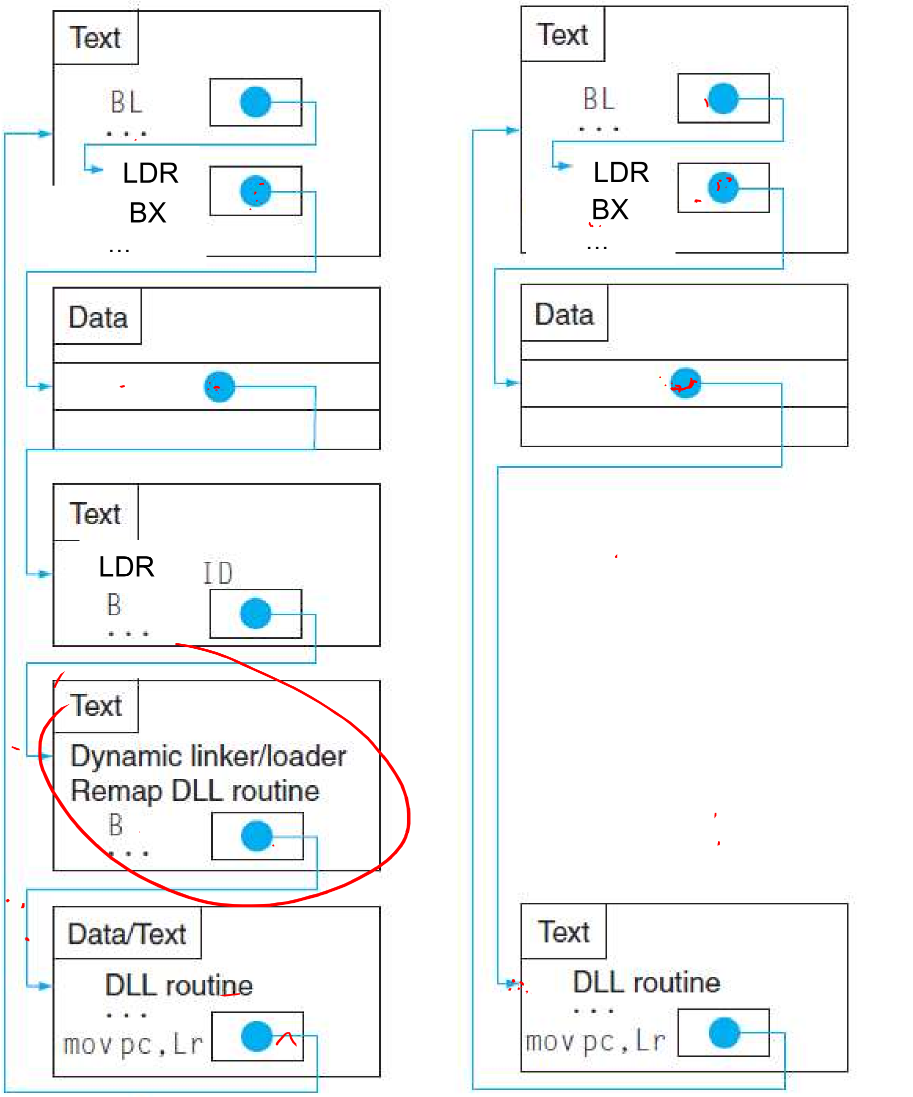
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

Resolving References (2/2)

- To resolve references:
    - search for reference (data or label) in all “user” symbol tables
    - if not found, search library files (for example, for `printf`)
    - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Static vs Dynamically linked libraries

- What we've described is the traditional way: "statically-linked" approach
  - The library is now part of the executable, so if the library updates, we don't get the fix (have to recompile if we have source)
  - It includes the entire library even if not all of it will be used.
  - Executable is self-contained.
- An alternative is dynamically linked libraries (DLL), common on Windows & UNIX platforms



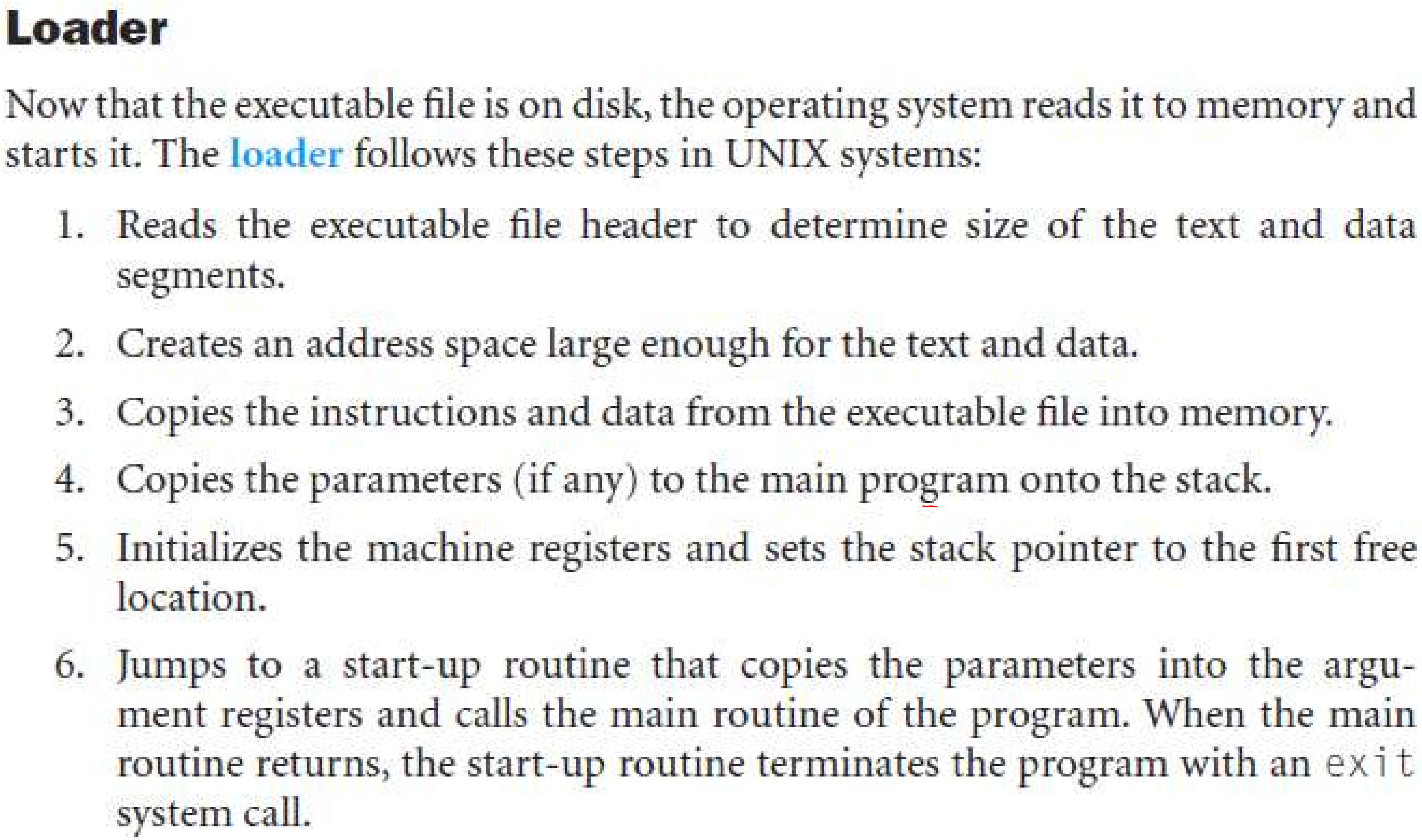
a. First call to DLL routine

b. Subsequent calls to DLL routine

Dynamically linked libraries

This does add quite a bit of complexity to the compiler, linker, and operating system. However, provides many benefits:

- **Space/time savings**
    - **Storing a program requires less disk space**
    - **Sending a program requires less time**
    - **Executing two programs requires less memory (if they share a library)**
- **Upgrades**
    - **By replacing one file (libXYZ.so), you upgrade every program that uses library "XYZ”**



Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. The loader follows these steps in UNIX systems:

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory.
4. Copies the parameters (if any) to the main program onto the stack.
5. Initializes the machine registers and sets the stack pointer to the first free location.
6. Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an `exit` system call.

BL uses PC-relative addressing. Suppose that BL procB is at 0x0040 0000 and procB is at address 0x0040 0108, then what is the value for the signed_immed_24 in the branch instruction?

A. 0x000100 /4

B. 0x000108

[✓ C.] 0x000040

D. 0x000042

00400000 BL . - - \

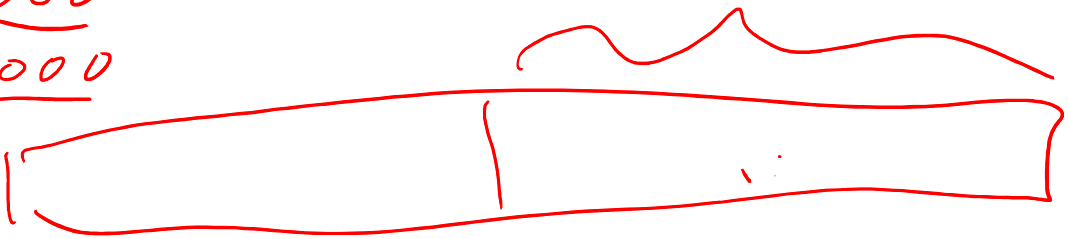
PC+8

00400108

000 1 0000 0000

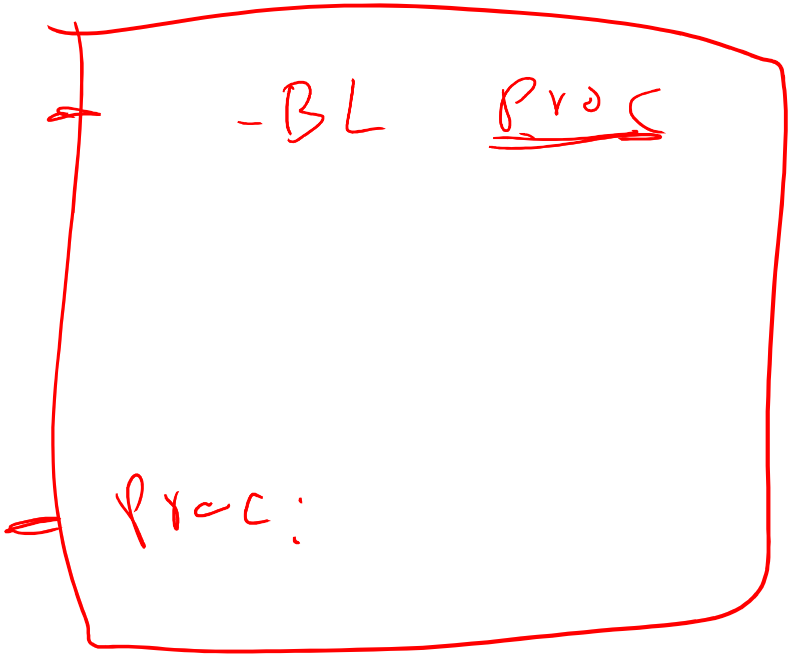
0000 0100 0000

4 0



BL uses PC-relative addressing. Will instruction “BL proc” be affected by relocation during linking?

A. Always
B. ~~Never~~
[✓ C. Depends]



Will instruction “BLT Label1” be affected by relocation during linking?

A. Always
[✓ B. Never]
C. Depends