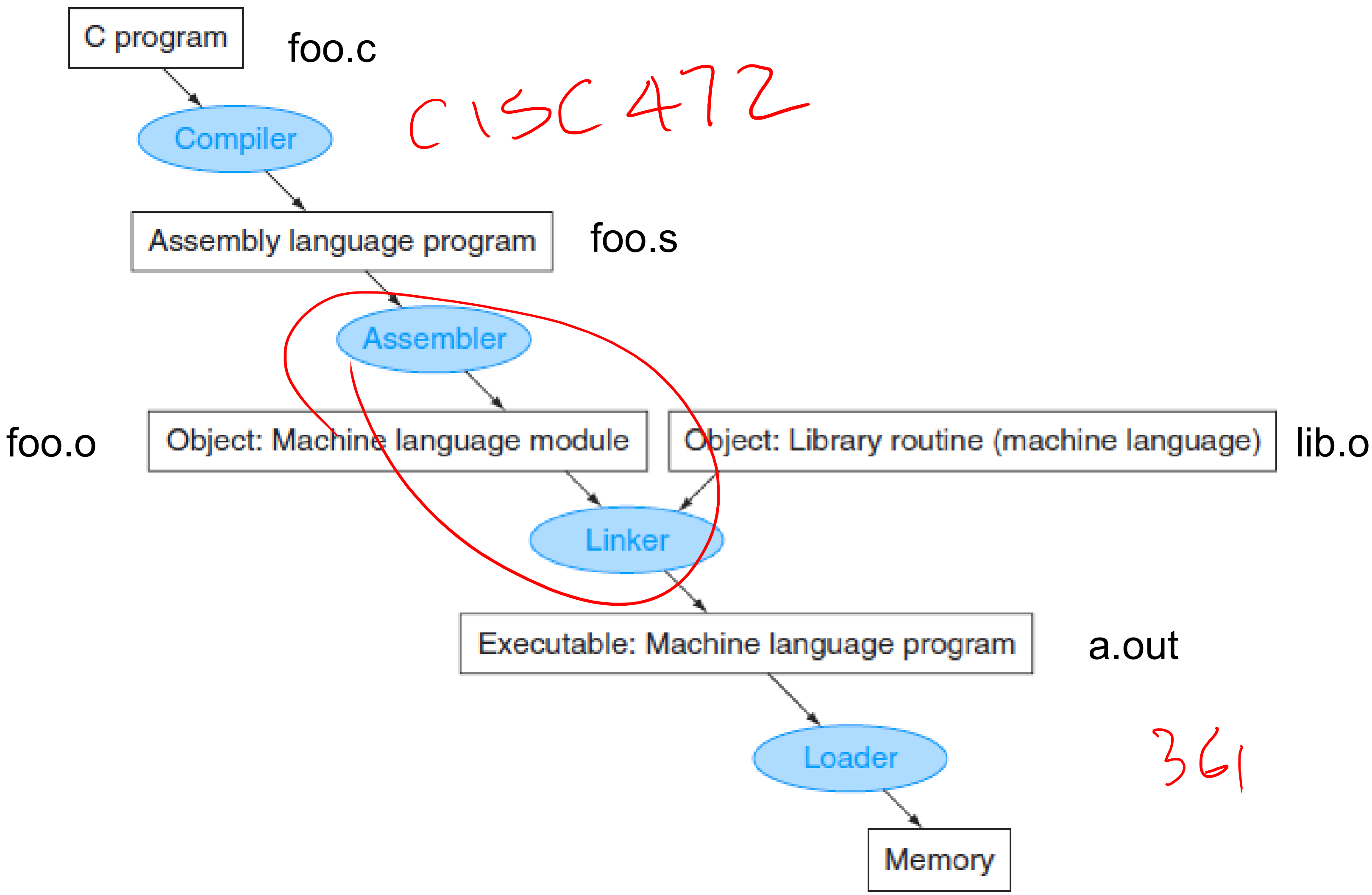
CISC260 Machine Organization and Assembly Language

Assembler, Linker & Loader

OS


C program
foo.c
Compiler
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
361
Memory

- 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.

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]

pc+8

......

......

......

0x12345678

cisc372

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

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

symbol	**address**
Else	0x0000101C

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

Pc+8

signed_immed_24
= [target address – (pc+8)] / 4
= [0x0000101C – (0x00001004 + 8)]/4
= [0x0000101C – 0x0000100C] / 4
= 0x00000010 / 4
= 0x00000004

B, BL

31–28	27	26	25	24	23–0
cond	1	0	1	L	signed_immed_24

B (Branch) and BL (Branch and Link) cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

BL also stores a return address in the link register, R14 (also known as LR).

Syntax

```
B{L}{<cond>}  <target_address>
```

where:

L
: Causes the L bit (bit 24) in the instruction to be set to 1. The resulting instruction stores a return address in the link register (R14). If L is omitted, the L bit is 0 and the instruction simply branches without storing a return address.

<cond>
: Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.

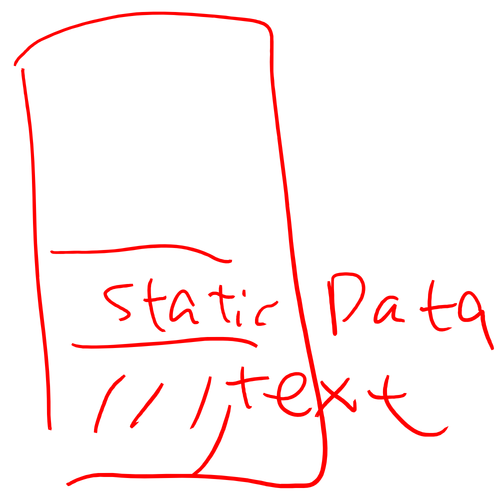
<target_address>
: Specifies the address to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 30 bits.
2. Shifting the result left two bits to form a 32-bit value.
3. Adding this to the contents of the PC, which contains the address of the branch instruction plus 8 bytes.

The instruction can therefore specify a branch of approximately ±32MB (see *Usage* on page A4-11 for precise range).

Relocation: address change during link

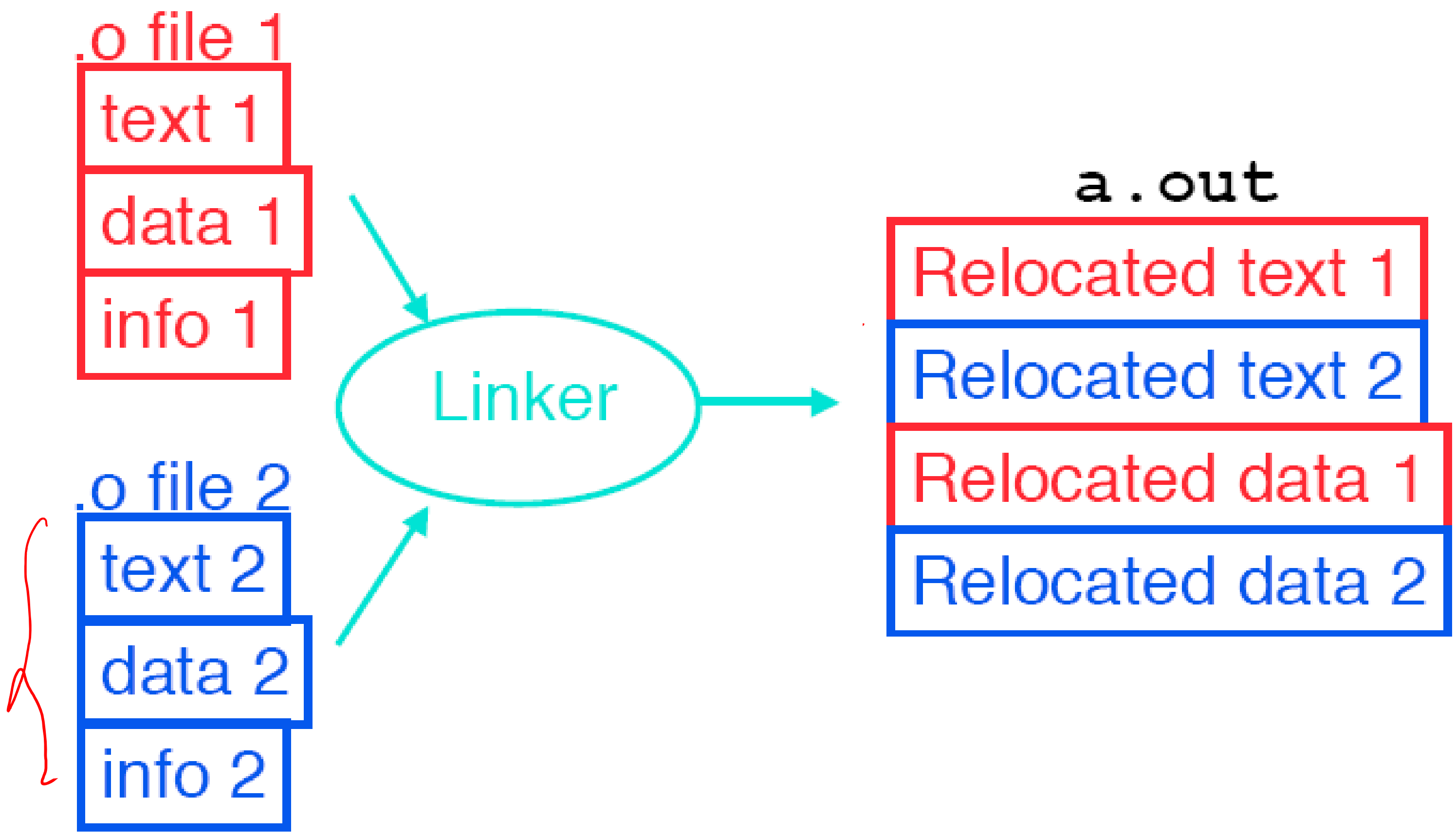
```
.text
…
LDR      r0, =myData                  @ data will be loaded in different
LDR      r1, [r0, #4]                 @ memory segment
…
…

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


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

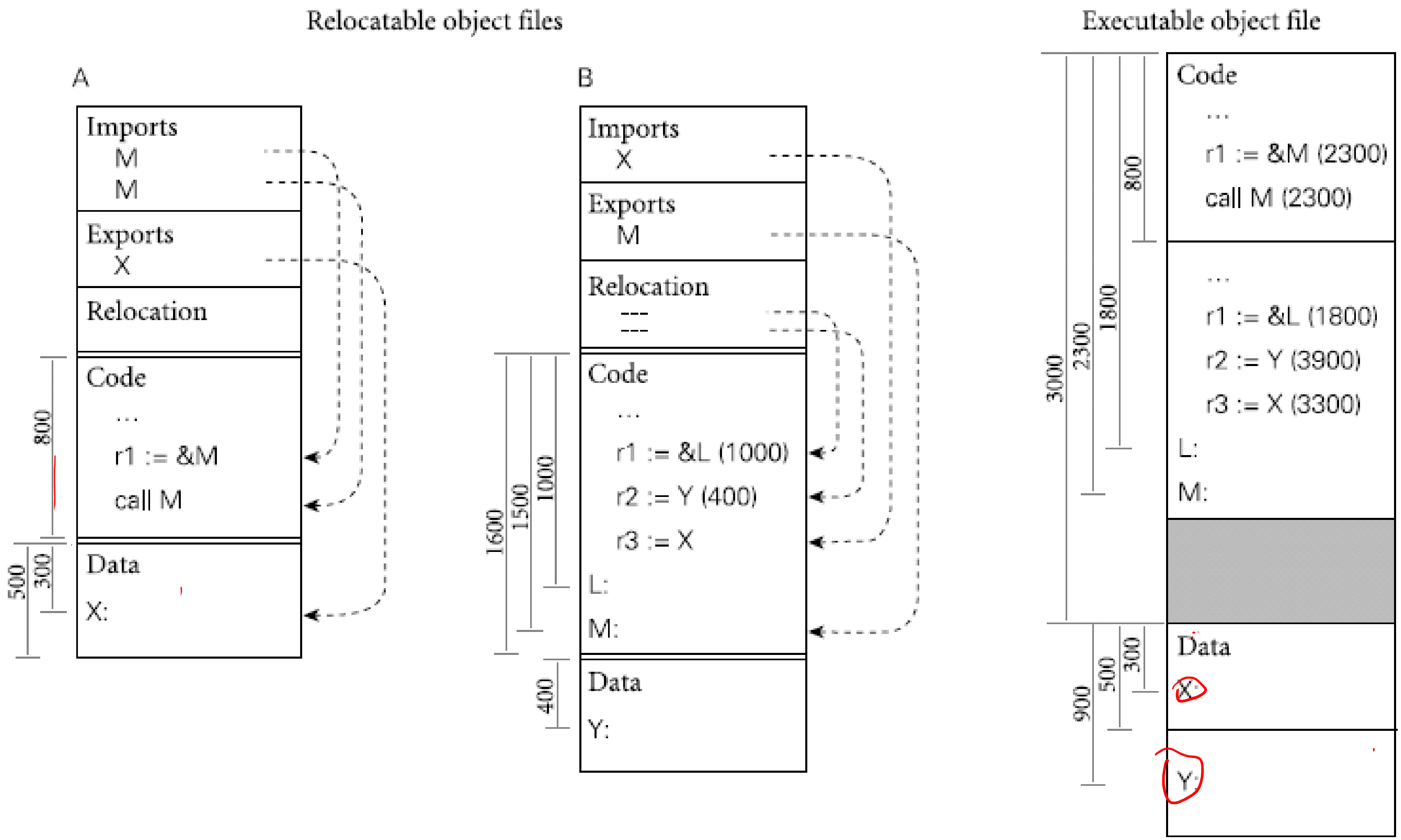


Linker is a systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file. **Executable.**

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
Executable object file
A
Imports
M
M
Exports
X
Relocation
Code
...
r1 := &M
call M
Data
X:
800
500
300
B
Imports
X
Exports
M
Relocation

Code
...
r1 := &L (1000)
r2 := Y (400)
r3 := X
L:
M:
Data
Y:
1600
1500
1000
400
Code
...
r1 := &M (2300)
call M (2300)
...
r1 := &L (1800)
r2 := Y (3900)
r3 := X (3300)
L:
M:
Data
X:
Y:
3000
2300
1800
800
900
500
300

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	–	
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	–	

Note: The following is for the 32-bit ARM7, see Chapter 02_COD 4e ARM

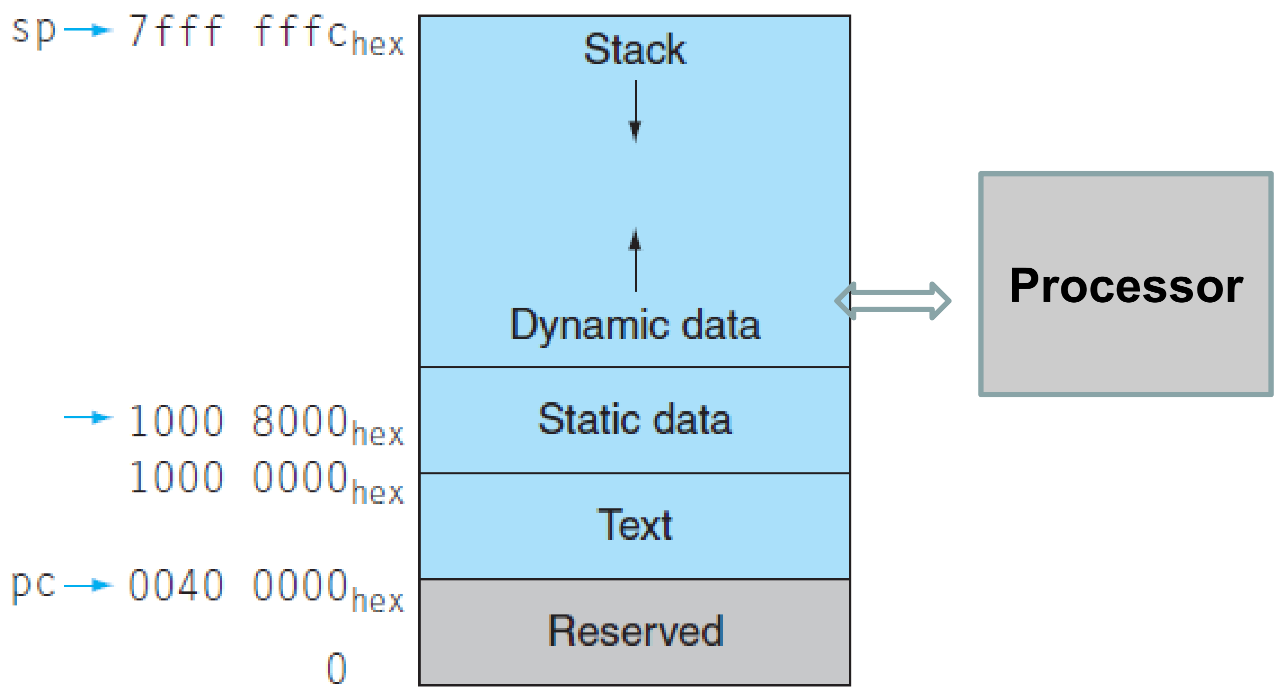


FIGURE 2.13 Typical ARM memory allocation for program and data. These addresses are only a software convention, and not part of the ARM architecture. The stack pointer is initialized to `7fff fffc`$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at `0040 0000`$_{hex}$. The static data starts at `1000 0000`$_{hex}$. Dynamic data, allocated by `malloc` in C and by `new` in Java, is next. It grows up toward the stack in an area called the heap.

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

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.