CSE 31
Computer Organization

Lecture 14 – Program Process (1)

Announcement

- No new Lab assignment this week
  - Work on Project #1 instead
- HW #4 (from zyBooks)
  - Due Monday (10/22)
- Project #1
  - Due Monday (10/22)
    - Don’t start late, you won’t have time!
- Reading assignment
  - Chapter 1.6, 6.1 – 6.7 of zyBooks (Reading Assignment #5)
    - Make sure to do the Participation Activities
    - Due Monday (10/29)

Program Process - Overview

- Interpretation vs Translation
- Translating C Programs
  - Compiler
  - Assembler
  - Linker
  - Loader (next time)
- An Example (next time)

Language Execution Continuum

- An Interpreter is a program that executes other programs.
- Language translation gives us another option.
- In general, we interpret a high level language when efficiency is not critical and translate to a lower level language to improve performance

Java bytecode

Matlab

Scheme Java C++ C Assembly machine language

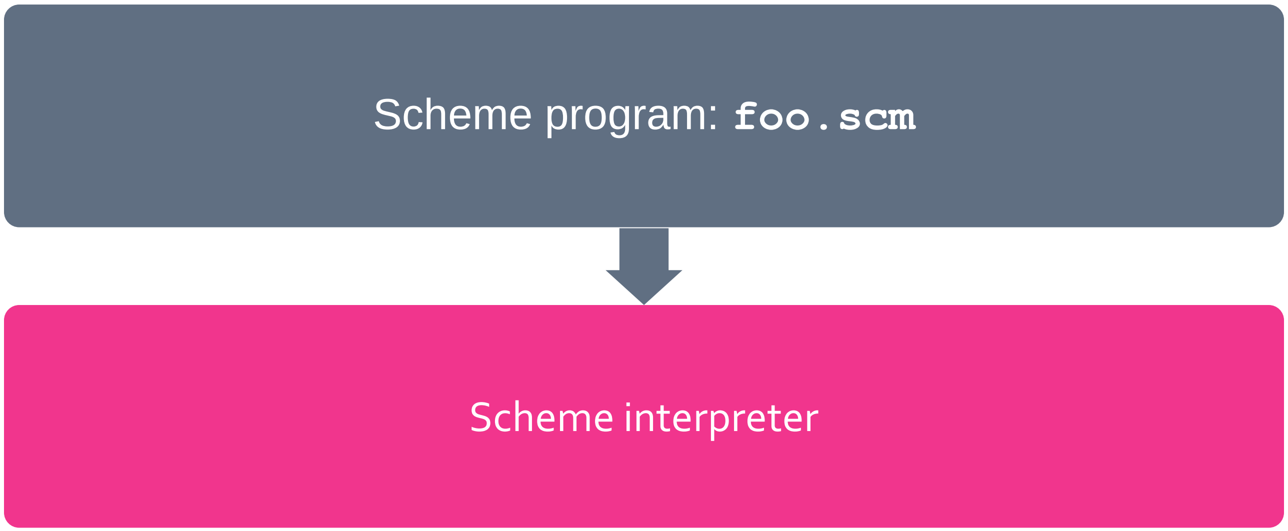
Easy to program Difficult to program

Inefficient to interpret Efficient to interpret

Interpretation vs Translation

- How do we run a program written in a source language?
  - Interpreter: Directly executes a program in the source language
  - Translator: Converts a program from the source language to an equivalent program in another language
- For example, consider a Scheme program **`foo.scm`**

Interpretation

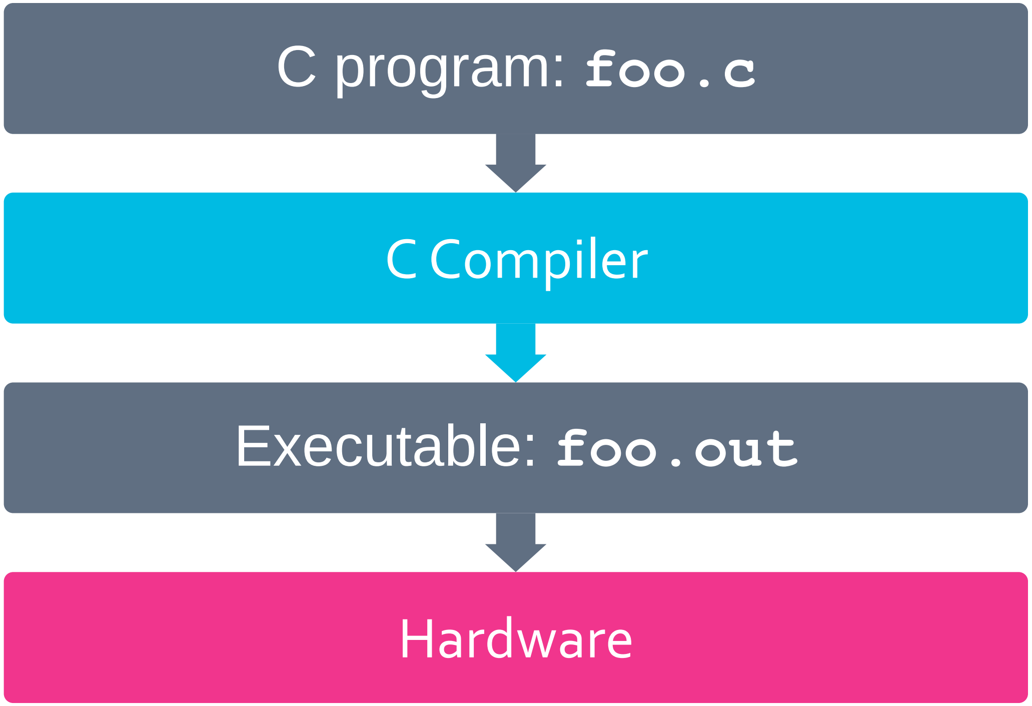


- Scheme Interpreter is just a program that reads a scheme program and performs the functions of that scheme program.

The process happens in run-time

Translation

- C Compiler is a translator from C to machine language.
- The processor is a hardware interpreter of machine language.



Interpretation

- Any good reason to interpret machine language in software?
  - MARS – useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
    - Similar issue with switching to x86.
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary
    - Through emulation

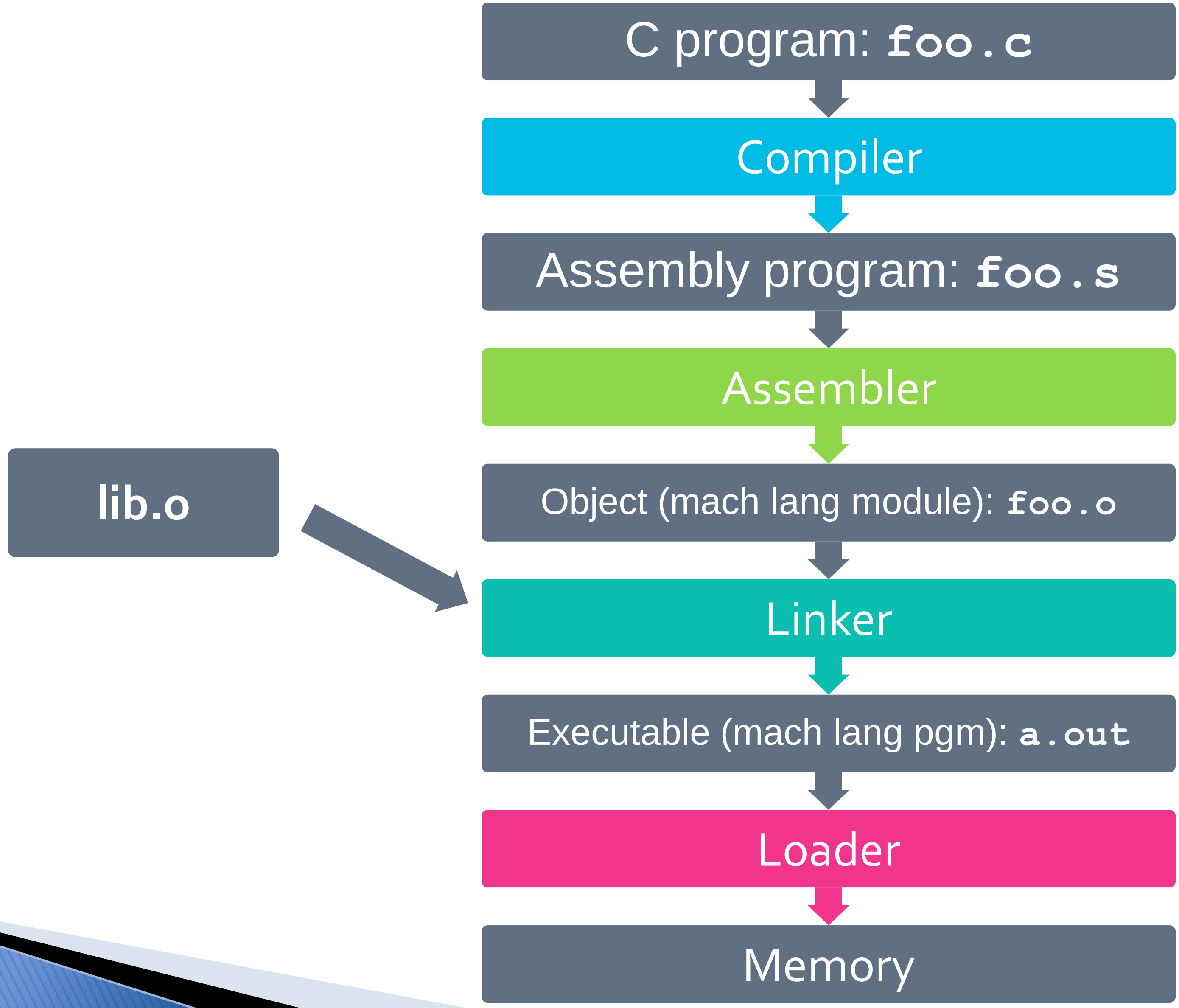
Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., MARS)
    - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?), code smaller (2x?)
- Interpreter provides instruction set independence: run on any machine

Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore higher performance:
  - Important for many applications, particularly operating systems.
- Translation/compilation helps “hide” the program “source” from the users:
  - One model for creating value in the marketplace (eg. Microsoft keeps all their source code secret)
  - Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers.

Steps to Starting a Program (translation)

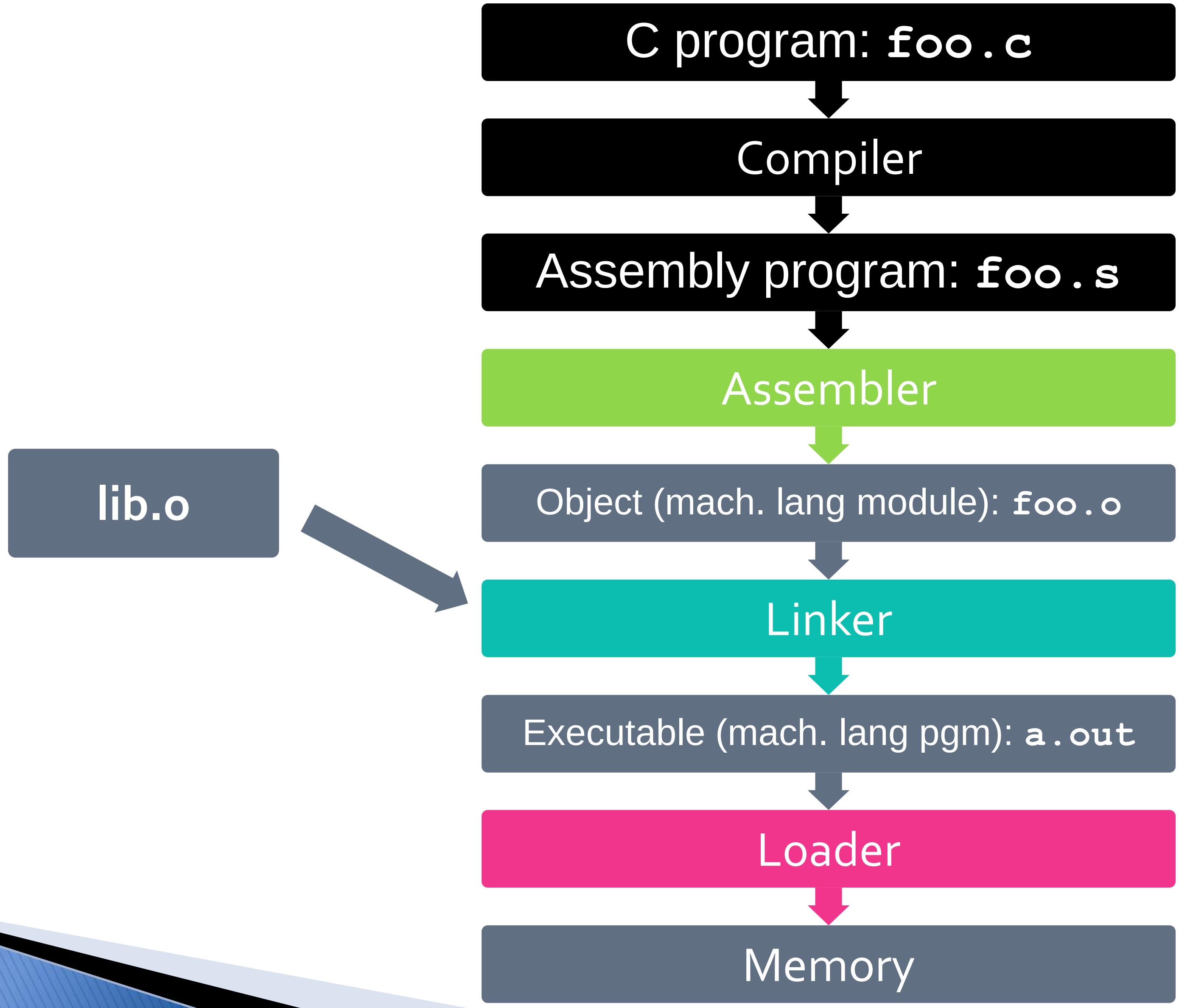


Compiler

- Input: High-Level Language Code
  (e.g., C, Java such as `foo.c`)
- Output: Assembly Language Code
  (e.g., `foo.s` for MIPS)
- Note: Output *may* contain pseudo-instructions
  - Pseudo-instructions: instructions that assembler understands but not in machine (previous lectures)
  - For example:

    `mov $s1,$s2` → **`or $s1,$s2,$zero`**

Where Are We Now?



Assembler

- Input: Assembly Language Code
  (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables
  (e.g., `foo.o` for MIPS)
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File

Assembler Directives

- Give directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text segment (machine code, program memory)
  - `.data`: Subsequent items put in user data segment (binary rep of data in source file, data memory)
  - `.globl sym`: declares `sym` global and can be referenced from other files
  - `.asciiz str`: Store the string **`str`** in memory and null-terminate it
  - `.word w1…wn`: Store the *n* 32-bit quantities in successive memory words

Pseudo-instruction Replacement

- Assembler treats convenient variations of machine language instructions as if real instructions

MAL:	TAL:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mult $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on.
  - All necessary info is within the instruction already.
- What about Branches?
  - PC-Relative
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch.
  - So these can be handled.

Producing Machine Language (2/3)

- “Forward Reference” problem
  - Branch instructions can refer to labels that are “forward” in the program:

```
    or   $v0, $0,  $0
L1: slt  $t0, $0,  $a1
    beq  $t0, $0,  L2
    addi $a1, $a1, -1
    j    L1
L2: add  $t1, $a0, $a1
```

How does it know where in the future steps?

  - Solved by taking 2 passes over the program.
    - First pass remembers position of labels
    - Second pass uses label positions to generate code

Producing Machine Language (3/3)

- What about jumps (`j` and `jal`)?
  - Jumps require absolute address.
  - So, forward or not, still can't generate machine instruction without knowing the position of instructions in memory.
- What about references to data?
  - `la` gets broken up into `lui` and `ori`
  - These will require the full 32-bit address of the data.
- These can't be determined yet, so we create two tables…

Symbol Table

- List of "items" in this file that may be used by other files.
- What are they?
  - Labels: function calling
  - Data: anything in the `.data` section; variables which may be accessed across files

Relocation Table

- List of "items" this file needs the address later.
- What are they?
  - Any label jumped to: **`j`** or **`jal`**
    - Internal
    - external (including lib files)
  - Any piece of data
    - such as the **`la`** instruction

Object File Format

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation information: identifies lines of code that **need** to be “handled”
- symbol table: list of this file’s labels and data that can be referenced
- debugging information
- A standard format is ELF (except MS) http://www.skyfree.org/linux/references/ELF_Format.pdf

Quiz

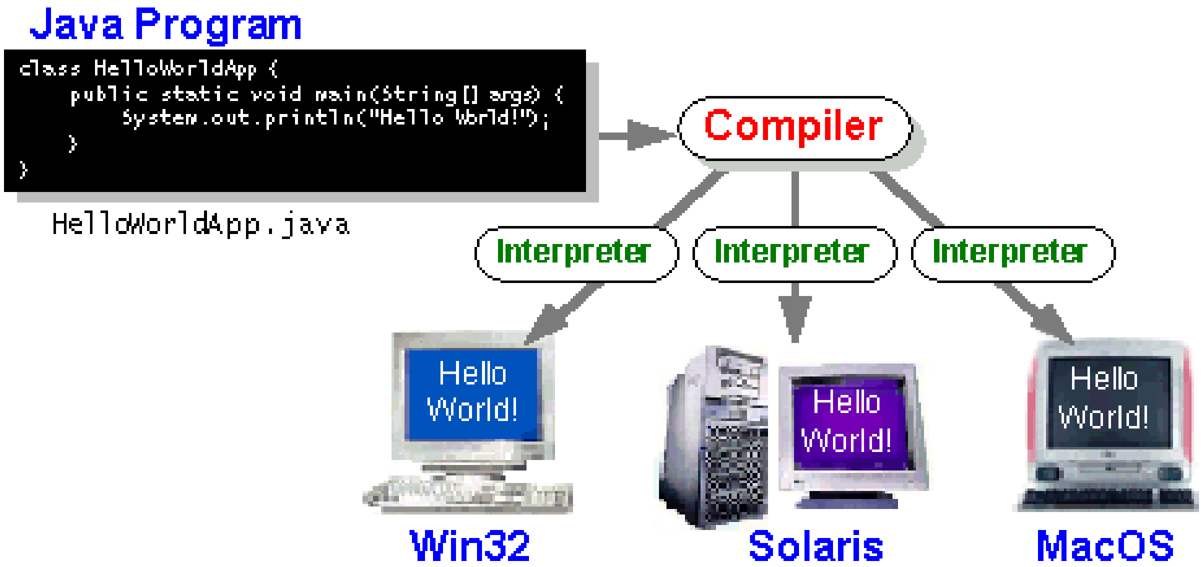
1) Assembler will ignore the instruction **`Loop:nop`** because it does nothing.

2) Java designers used a translater AND interpreter (rather than just a translater) mainly because of (at least 1 of): ease of writing, better error msgs, smaller object code.

	12
a)	FF
b)	FT
c)	TF
d)	TT

Quiz

1) Assembler keeps track of all labels in symbol table…F!



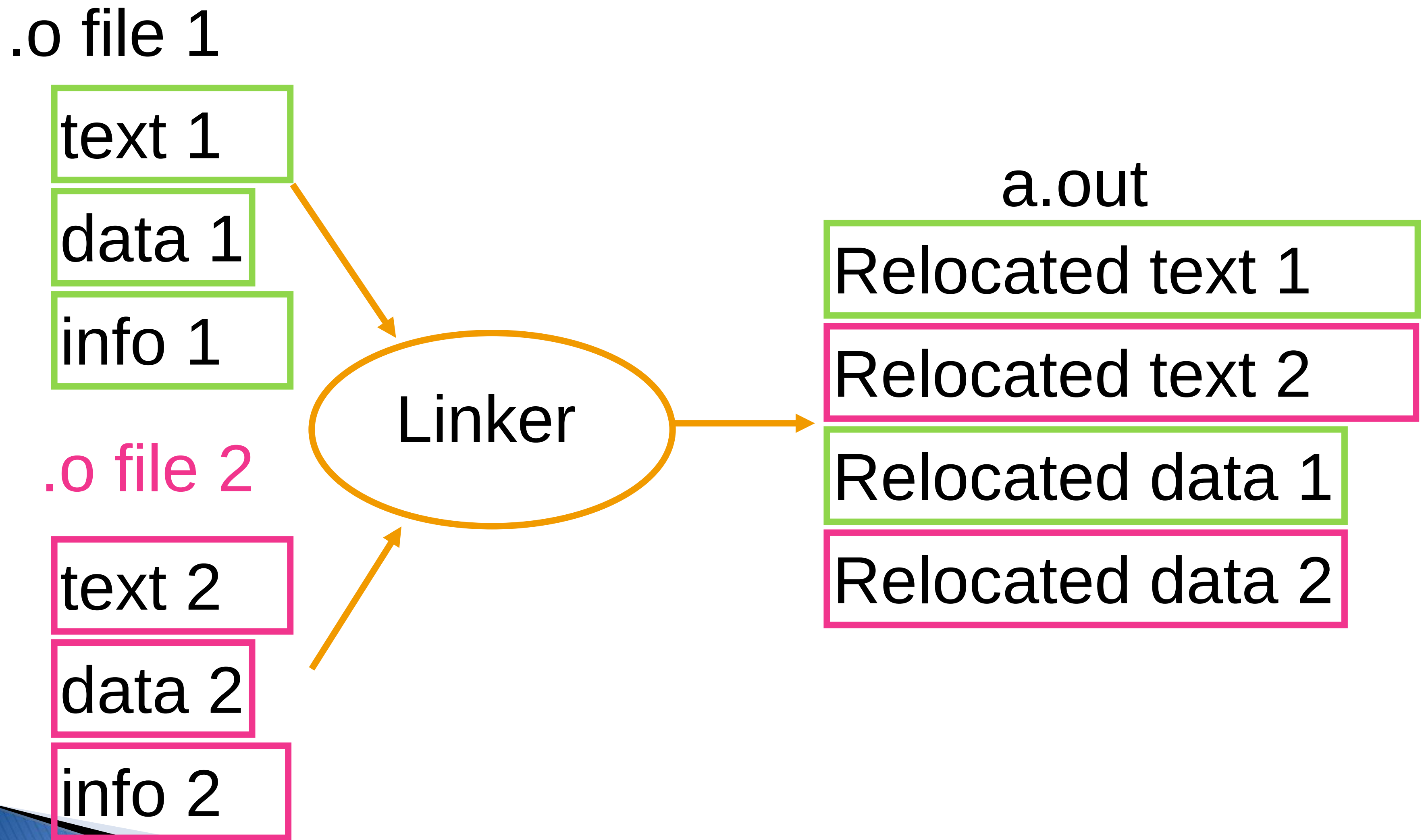
2) Java designers used both mainly because of code portability…F!

	12
a)	**FF**
b)	FT
c)	TF
d)	TT

Linker (1/3)

- Input: Object Code files, information tables (e.g., `foo.o,libc.o` for MIPS)
- Output: Executable Code (e.g., `a.out` for MIPS)
- Combines several object (`.o`) files into a single executable ("linking")
- Enable Separate Compilation of files
  - Changes to one file do not require recompilation of whole program
    - Windows NT source was > 40 M lines of code!
  - Old name "Link Editor" from editing the "links" in jump and link instructions

Linker (2/3)


.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

Linker (3/3)

- 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.
- Step 3: Resolve References
  - Go through Relocation Table; handle each entry
  - That is, fill in all absolute addresses

Four Types of Addresses

- PC-Relative Addressing (`beq`, `bne`)
  - never relocate
- Absolute Address (`j`, `jal`)
  - always relocate
- External Reference (usually `jal`)
  - always relocate
- Data Reference (often `lui` and `ori/load address`)
  - always relocate

Absolute Addresses in MIPS

- Which instructions need relocation editing?
  - J-format: jump, jump and link (jal)

J/JAL	XXXXX

  - Loads and stores to variables in static area, relative to global pointer

LW/SW	\$gp	\$x	address

  - What about conditional branches?

beq/bne	\$rs	\$rt	address

    - PC-relative addressing preserved even if code moves

Resolving References (1/2)

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

- 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)