5118007-02 Computer Architecture

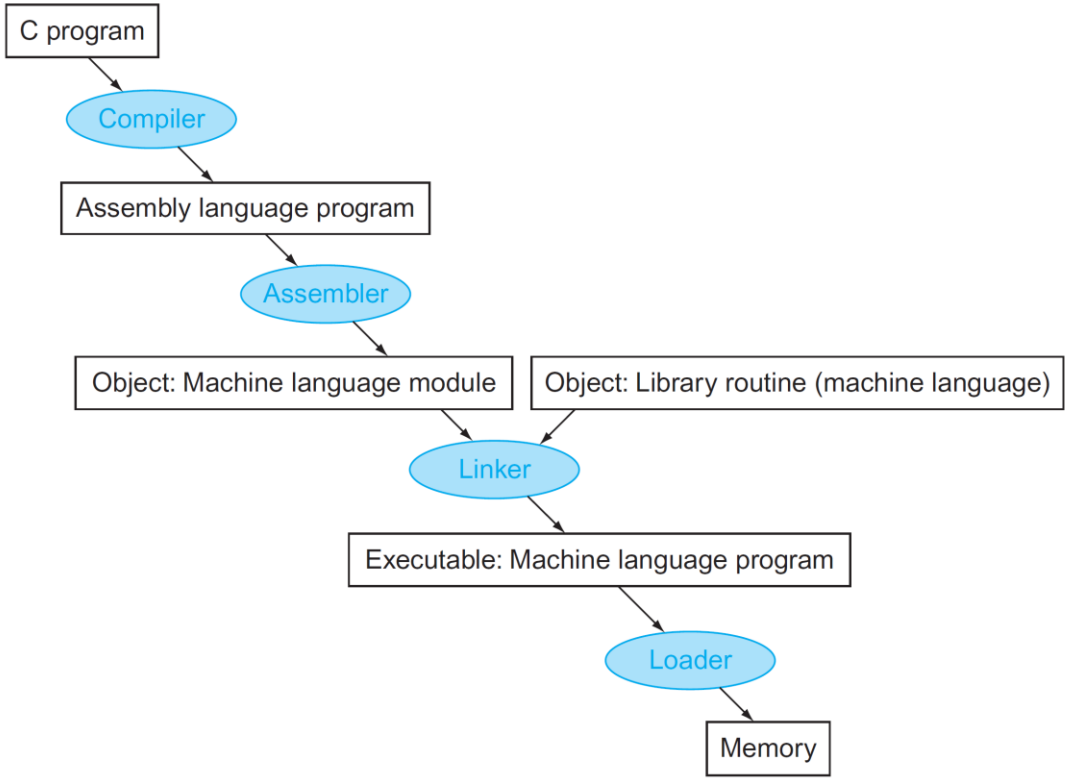
Ch. 2 Instructions: Language of the Computer

2 Apr 2024

Shin Hong

Translating and Starting Program

- Translation hierarchy for C programs



Compiler

- Transforms the C programs into an assembly language program
- In 1970s, most of operating systems were written in assembly language since there were not enough hardware resource and software technologies for optimizing compiler

Assembler

- Turn assembly program into an object file
  - an object is a combination of machine instructions, data and information needed to place instructions in memory
    - object header, text segment, static data segment, relation information, symbol table, debug information
  - the assembler determines the address of all labels
    - the assembler keep track of labels in a symbol table
- Translate pseudoinstructions
  - assembler provides more instructions than the architecture supports
  - Ex.

```
move $t0, $t1
add $t0, $zero, $t1
```

Linker

- It is desirable to compile and assemble each procedure independently
- A linker (link editor) takes independently assembled machine language programs and integrate them into an executable program
  1. Place code and data modules symbolically in memory
  2. Determine the address of data and instruction labels
  3. Patch (or resolve) both the internal and external references

- An executable file does not have any unresolved references, yet it is possible to have partially linked files (i.e., dynamically linked library)

Example

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	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	`sw $a1, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	`lw $a0,` 8000_{hex}`($gp)`
	$0040\ 0004_{hex}$	`jal 40 0100`$_{hex}$
	…	…
	$0040\ 0100_{hex}$	`sw $a1,` 8020_{hex}`($gp)`
	$0040\ 0104_{hex}$	`jal 40 0000`$_{hex}$
	…	…
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	…	…
	$1000\ 0020_{hex}$	(Y)
	…	…

`$gp: 1000 8000`$_{hex}$

- `lw` takes 16-bit signed constant

Loader

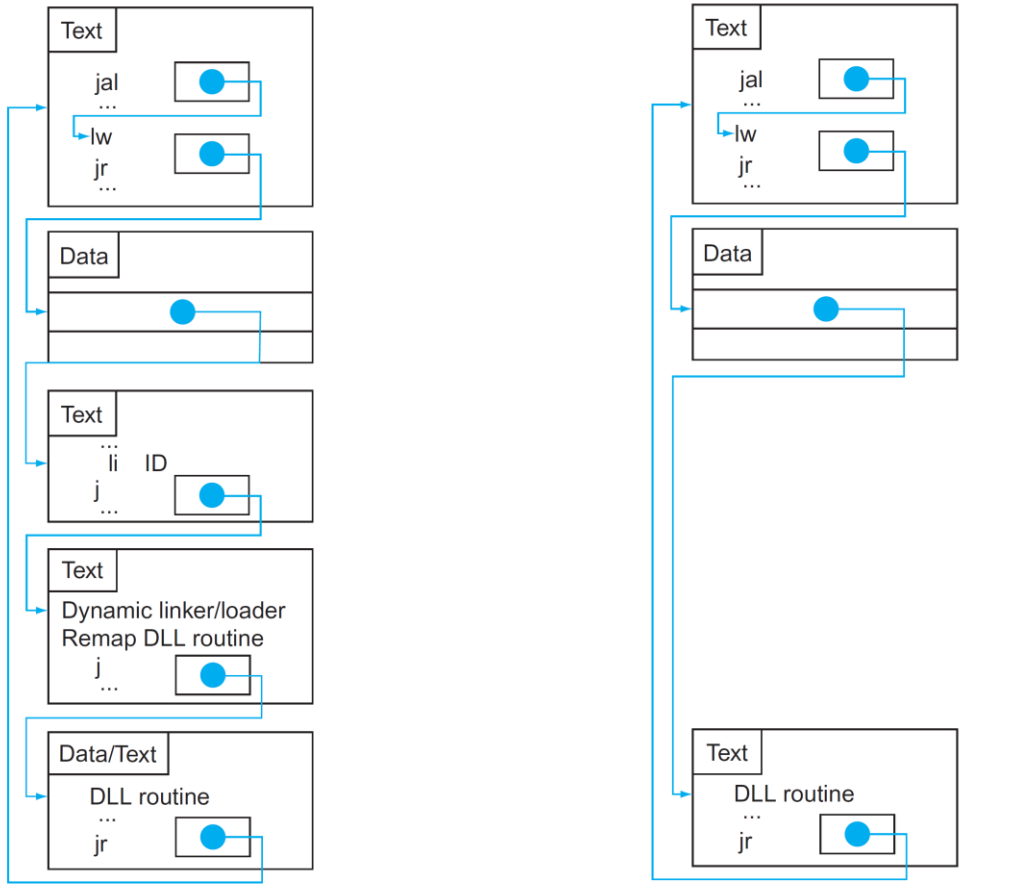
- The OS reads an executable file to memory
- Steps
  1. Determine the size of the text and data segments
  2. Create an address space for the text and data segments
  3. Copies the text and data segments from an executable file to the memory
  4. Initialize machine registers and sets the stack pointers
  5. Jump to a start-up routine that copies the parameters into the argument registers and calls the main routine

Dynamically Linked Libraries (1/2)

- Limitations of static linking
  - memory inefficient due to code duplication
  - difficult to update when new version is released

- Dynamically linked libraries (DLLs)
  - library routines are linked to dummy functions, not actual target library routine
    - a dummy function holds a reference by which it indirectly invokes a target routine once the proper address is given
  - the dynamic linker finds (or loads) the target library from the system runtime and update the external references in the dummy functions
    - lazy linkage

Dynamically Linked Libraries (1/2)



(a) First call to DLL routine

(b) Subsequent calls to DLL routine