

Compilation

0368-3133 (Semester A, 2013/14)

Lecture 13: Assembler, Linker & Loader

Noam Rinetzky

Slides credit: Eli Bendersky, Mooly Sagiv & Sanjeev Setia

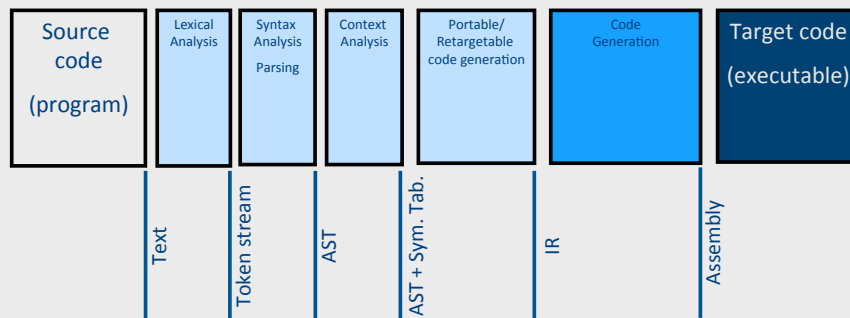
What is a compiler?

“A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).”

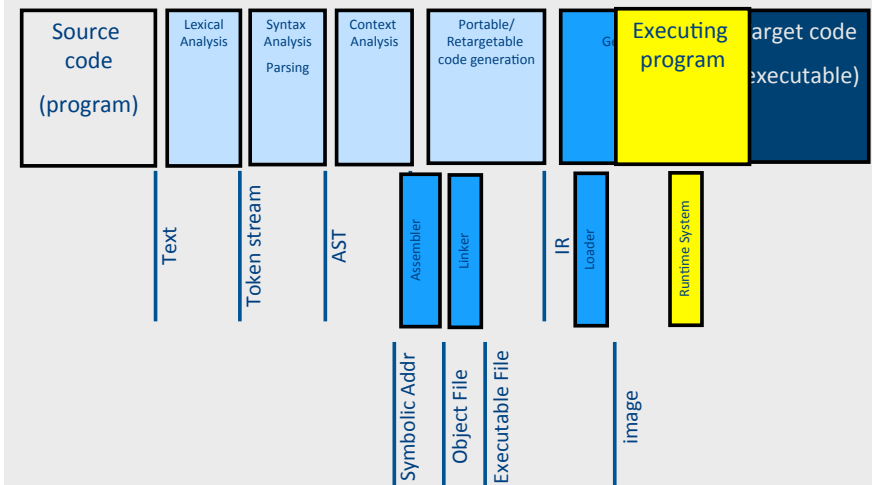
The most common reason for wanting to transform source code is to create an executable program.”

--Wikipedia

Stages of compilation



Compilation → Execution



Program Runtime State

Registers

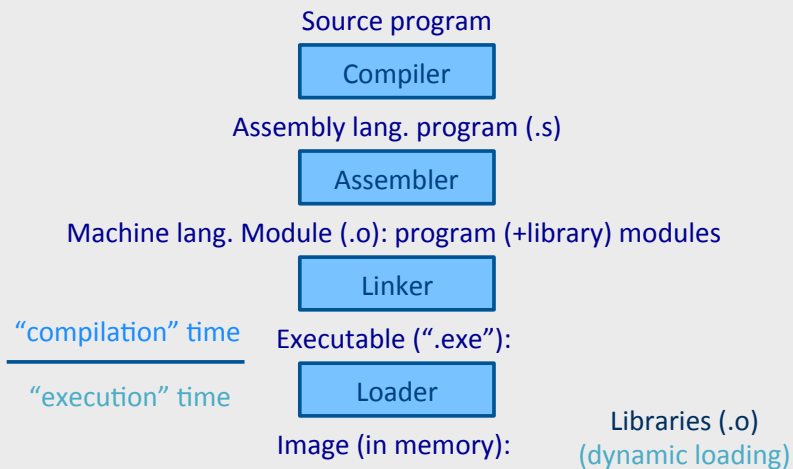
0x11000 foo, extern_foo printf	Code
0x22000 G, extern_G	Static Data
0x33000 x	Stack
0x88000	Heap
0x99000	

Challenges

- goto L2 → JMP 0x110FF
- G:=3 → MOV 0x2200F, 0..011
- foo() → CALL 0x130FF
- extern_G := 1 → MOV 0x2400F, 0..01
- extern_foo() → CALL 0x140FF
- printf() → CALL 0x150FF
- x:=2 → MOV FP+32, 0...010
- goto L2 → JMP [PC +] 0x000FF

0x11000 foo, extern_foo printf	Code
0x22000 G, extern_G	Static Data
0x33000 x	Stack
0x88000	Heap
0x99000	

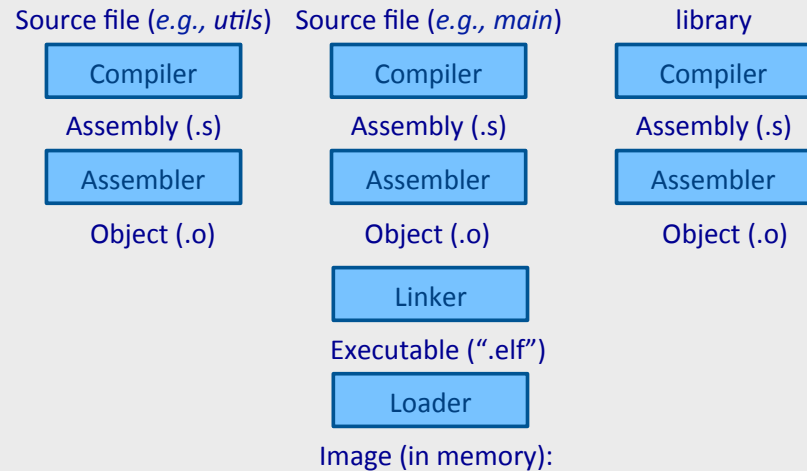
Assembly → Image



Outline

- Assembly
- Linker / Link editor
- Loader
- Static linking
- Dynamic linking

Assembly → Image



Assembler

- Converts (symbolic) assembler to binary (object) code
 - Object files contain a combination of machine instructions, data, and information needed to place instructions properly in memory
 - Yet another (simple) compiler
 - One-to-one translation
- Converts constants to machine repr. (3 → 0...011)
- Resolve internal references
- Records info for code & data relocation

Object File Format

Header	Text Segment	Data Segment	Relocation Information	Symbol Table	Debugging Information
--------	--------------	--------------	------------------------	--------------	-----------------------

- Header: Admin info + “file map”
- Text seg.: machine instruction
- Data seg.: (Initialized) data in machine format
- Relocation info: instructions and data that depend on absolute addresses
- Symbol table: “exported” references + unresolved references

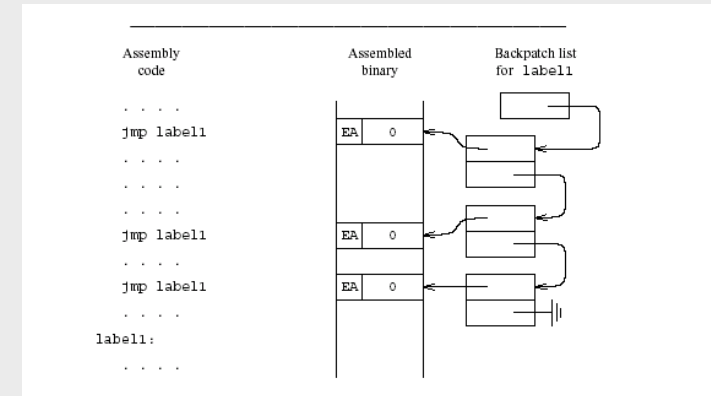
Handling Internal Addresses

```
.data
...
    .align 8
var1:
    .long 666
...
.code
...
    addl var1,%eax
...
    jmp label1
...
label1:
...
...
```

Resolving Internal Addresses

- Two scans of the code
 - Construct a table label → address
 - Replace labels with values
- One scan of the code (**Backpatching**)
 - Simultaneously construct the table and resolve symbolic addresses
 - Maintains list of unresolved labels
 - Useful beyond assemblers

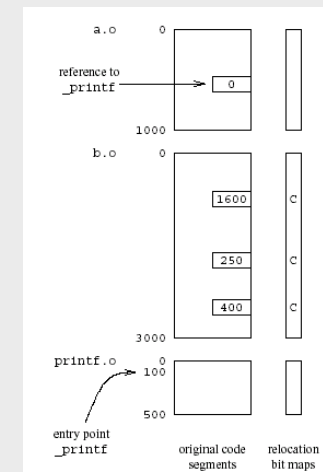
Backpatching



Handling External Addresses

- Record symbol table in “external” table
 - Exported (defined) symbols
 - G, foo()
 - Imported (required) symbols
 - Extern_G, extern_bar(), printf()
- Relocation bits
 - Mark instructions that depend on absolute (fixed) addresses
 - Instructions using globals,

Example



External references resolved by the Linker using the relocation info.

Example of External Symbol Table

External symbol	Type	Address
_options	entry point	50 data
__main	entry point	100 code
_printf	reference	500 code
_atoi	reference	600 code
_printf	reference	650 code
_exit	reference	700 code
_msg_list	entry point	300 data
_Out_Of_Memory	entry point	800 code
_fprintf	reference	900 code
_exit	reference	950 code
_file_list	reference	4 data

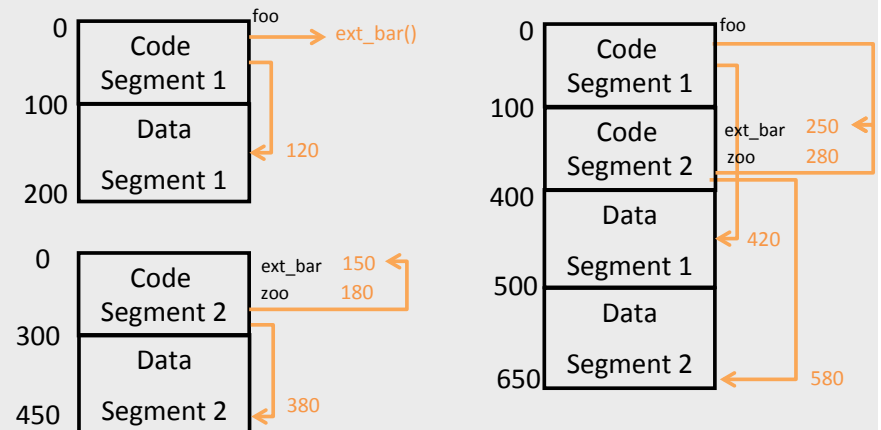
Assembler Summary

- Converts symbolic machine code to binary
 - `addl %edx, %ecx` \Rightarrow `000 0001 11 010 001` = `01 D1` (Hex)
- Format conversions
 - `3` \rightarrow `0x0..011` or `0x000000110...0`
- Resolves internal addresses
- Some assemblers support overloading
 - Different opcodes based on types

Linker

- Merges object files to an executable
 - Enables separate compilation
- Combine memory layouts of object modules
 - Links program calls to library routines
 - `printf()`, `malloc()`
 - Relocates instructions by adjusting absolute references
 - Resolves references among files

Linker



Relocation information

- Information needed to change addresses
 - Data
 - Code
- Positions in the code which contains addresses
 - Data
 - Code
- Two implementations
 - Bitmap
 - Linked-lists

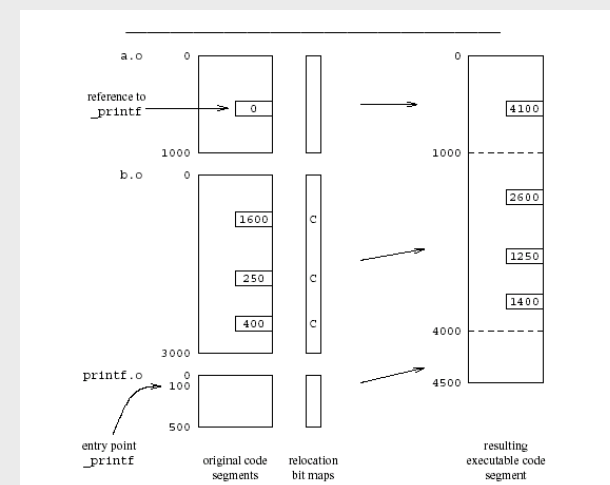
External References

- The code may include references to external names (identifiers)
 - Library calls
 - External data
- Stored in external symbol table

Example of External Symbol Table

External symbol	Type	Address
<code>_options</code>	entry point	50 data
<code>_main</code>	entry point	100 code
<code>_printf</code>	reference	500 code
<code>_atoi</code>	reference	600 code
<code>_printf</code>	reference	650 code
<code>_exit</code>	reference	700 code
<code>_msg_list</code>	entry point	300 data
<code>_Out_Of_Memory</code>	entry point	800 code
<code>_fprintf</code>	reference	900 code
<code>_exit</code>	reference	950 code
<code>_file_list</code>	reference	4 data

Example



Linker (Summary)

- Merge several executables
 - Resolve external references
 - Relocate addresses
- User mode
- Provided by the operating system
 - But can be specific for the compiler
 - More secure code
 - Better error diagnosis

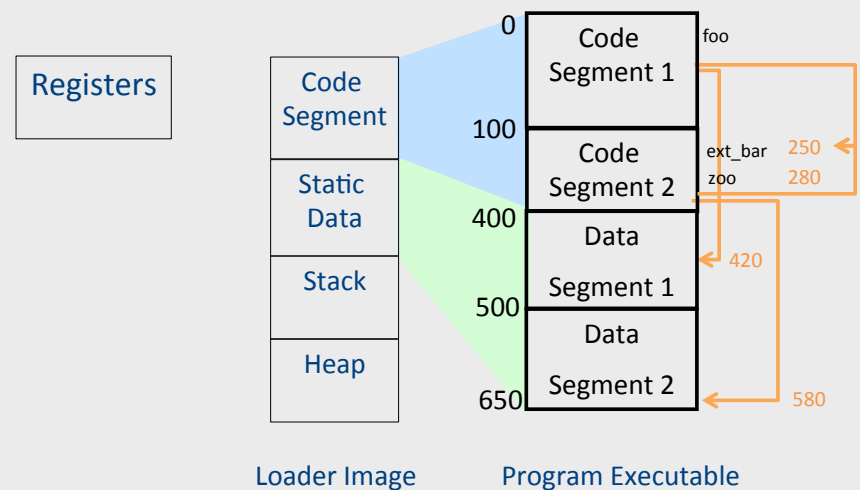
Linker Design Issues

- Merges
 - Code segments
 - Data segments
 - Relocation bit maps
 - External symbol tables
- Retain information about static length
- Real life complications
 - Aggregate initializations
 - Object file formats
 - Large library
 - Efficient search procedures

Loader

- Brings an executable file from disk into memory and starts it running
 - Read executable file's header to determine the size of text and data segments
 - Create a new address space for the program
 - Copies instructions and data into memory
 - Copies arguments passed to the program on the stack
- Initializes the machine registers including the stack ptr
- Jumps to a startup routine that copies the program's arguments from the stack to registers and calls the program's main routine

Program Loading



Loader (Summary)

- Initializes the runtime state
- Part of the operating system
 - Privileged mode
- Does not depend on the programming language
- “Invisible activation record”

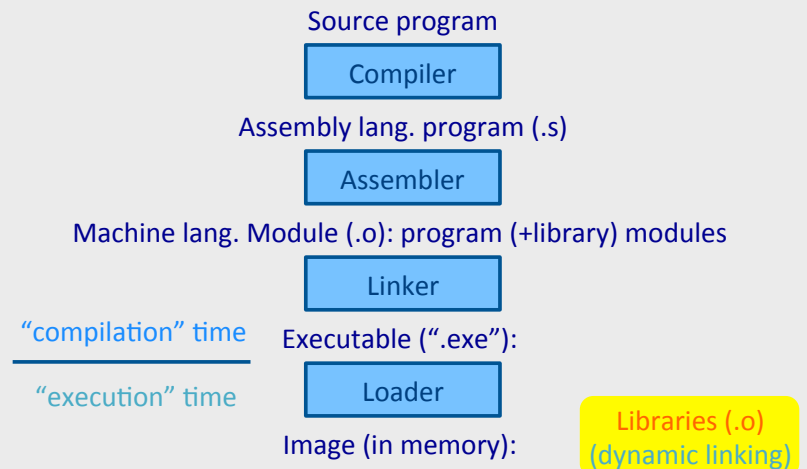
Static Linking (Recap)

- Assembler generates binary code
 - Unresolved addresses
 - Relocatable addresses
- Linker generates executable code
- Loader generates runtime states (images)

Dynamic Linking

- Why dynamic linking?
 - Shared libraries
 - Save space
 - Consistency
 - Dynamic loading
 - Load on demand

What's the challenge?



Position-Independent Code (PIC)

- Code which does not need to be changed regardless of the address in which it is loaded
 - Enable loading the same object file at different addresses
 - Thus, shared libraries and dynamic loading
- “Good” instructions for PIC: use relative addresses
 - relative jumps
 - reference to activation records
- “Bad” instructions for : use fixed addresses
 - Accessing global and static data
 - Procedure calls
 - Where are the library procedures located?

How?

“All problems in computer science can be solved by another level of indirection”

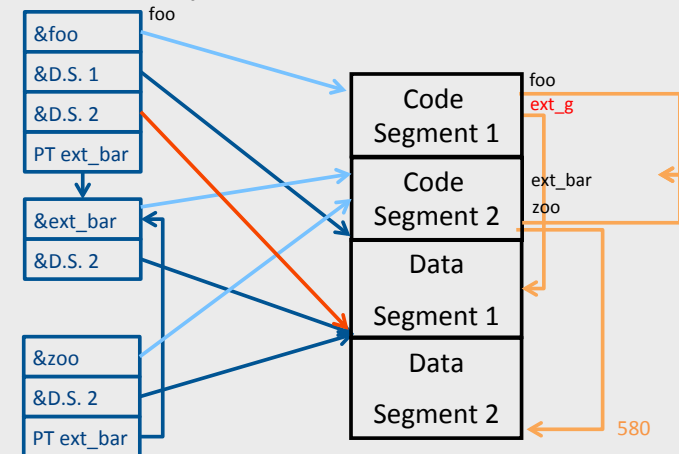
Butler Lampson

PIC: The Main Idea

- Keep the global data in a table
- Refer to all data relative to the designated register

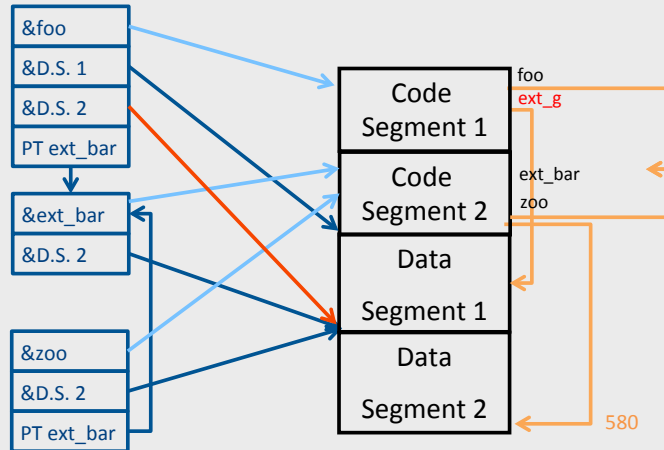
Per-Routine Pointer Table

- Record for every routine in a table



Per-Routine Pointer Table

- Record for every routine in a table



Per-Routine Pointer Table

- Record for every routine in a table
- Record used as a address to procedure

Caller:	Callee:
1. Load Pointer table address into RP	1. RP points to pointer table
2. Load Code address from 0(RP) into RC	2. Table has addresses of pointer table for subprocedures
3. Call via RC	

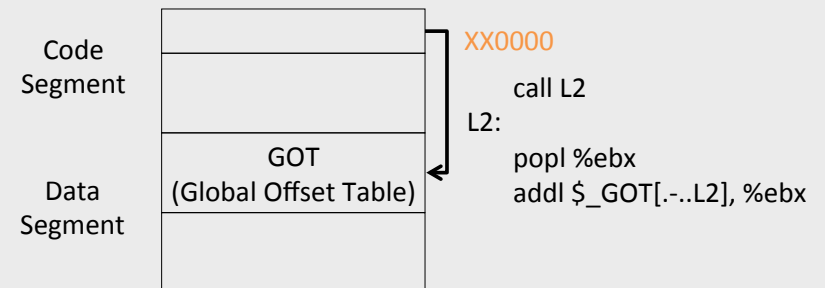


PIC: The Main Idea

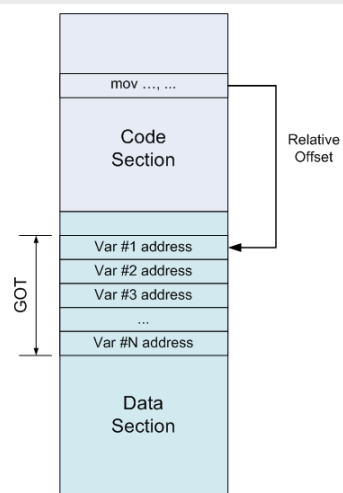
- Keep the global data in a table
- Refer to all data relative to the designated register
- Efficiency: use a register to point to the beginning of the table
 - Troublesome in CISC machines

ELF-Position Independent Code

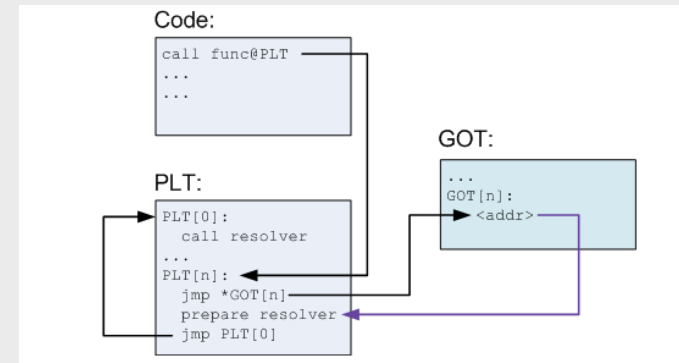
- Executable and Linkable code Format
 - Introduced in Unix System V
- Observation
 - Executable consists of code followed by data
 - The offset of the data from the beginning of the code is known at compile-time



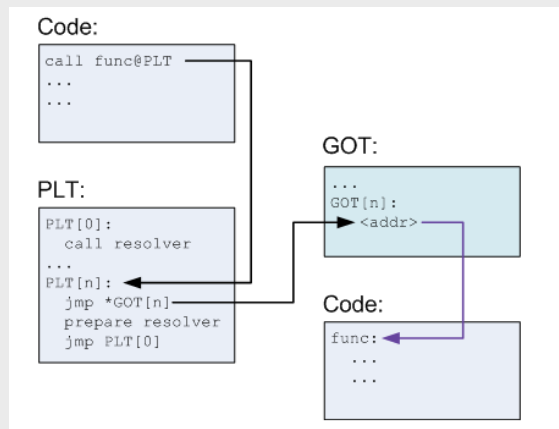
ELF: Accessing global data



ELF: Calling Procedures (before 1st call)



ELF: Calling Procedures (after 1st call)



PIC benefits and costs

- Enable loading w/o relocation
- Share memory locations among processes

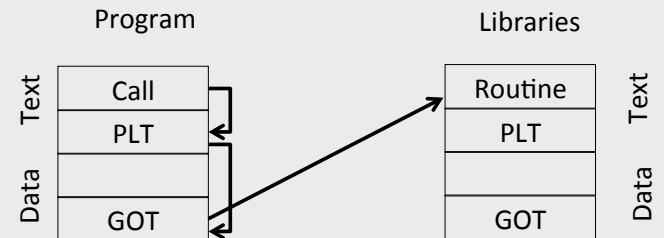
- Data segment may need to be reloaded
- GOT can be large
- More runtime overhead
- More space overhead

Shared Libraries

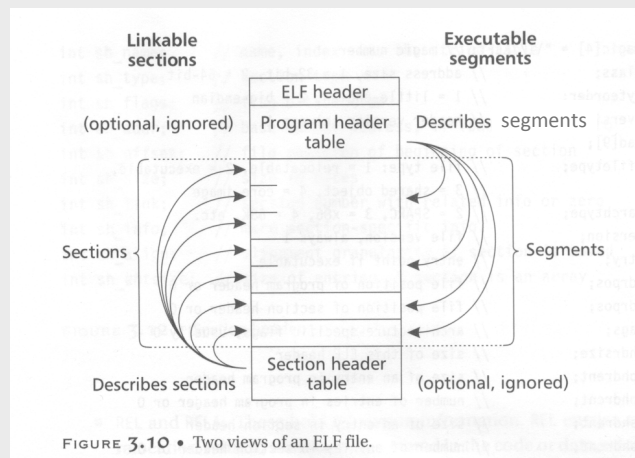
- Heavily used libraries
- Significant code space
 - 5-10 Mega for print
 - Significant disk space
 - Significant memory space
- Can be saved by sharing the same code
- Enforce consistency
- But introduces some overhead

- Can be implemented either with static or dynamic loading

Content of ELF file



ELF Structure



Consistency

- How to guarantee that the code/library used the “right” library version

Loading Dynamically Linked Programs

- Start the dynamic linker
- Find the libraries
- Initialization
 - Resolve symbols
 - GOT
 - Typically small
 - Library specific initialization
- Lazy procedure linkage

Microsoft Dynamic Libraries (DLL)

- Similar to ELF
- Somewhat simpler
- Require compiler support to address dynamic libraries
- Programs and DLL are Portable Executable (PE)
- Each application has its own address
- Supports lazy bindings

Dynamic Linking Approaches

- Unix/ELF uses a single name space and MS/PE uses several name spaces
- ELF executable lists the names of symbols and libraries it needs
- PE file lists the libraries to import from other libraries
- ELF is more flexible
- PE is more efficient

Costs of dynamic loading

- Load time relocation of libraries
- Load time resolution of libraries and executable
- Overhead from PIC prolog
- Overhead from indirect addressing
- Reserved registers

Summary

- Code generation yields code which is still far from executable
 - Delegate to existing assembler
- Assembler translates symbolic instructions into binary and creates relocation bits
- Linker creates executable from several files produced by the assembly
- Loader creates an image from executable