Assembler/Linker/Loader

Mooly Sagiv

html://www.cs.tau.ac.il/~msagiv/courses/wcc11-12.html

Chapter 4.3

J. Levine: Linkers & Loaders

http://linker.iecc.com/

Outline

- Where does it fit into the compiler
- Functionality
- “Backward” description
- Assembler design issues
- Linker design issues
- Advanced Issues
  - Position-Independent Code (PIC)
  - Shared Libraries
  - Dynamic Library Loading

A More Realistic Compiler

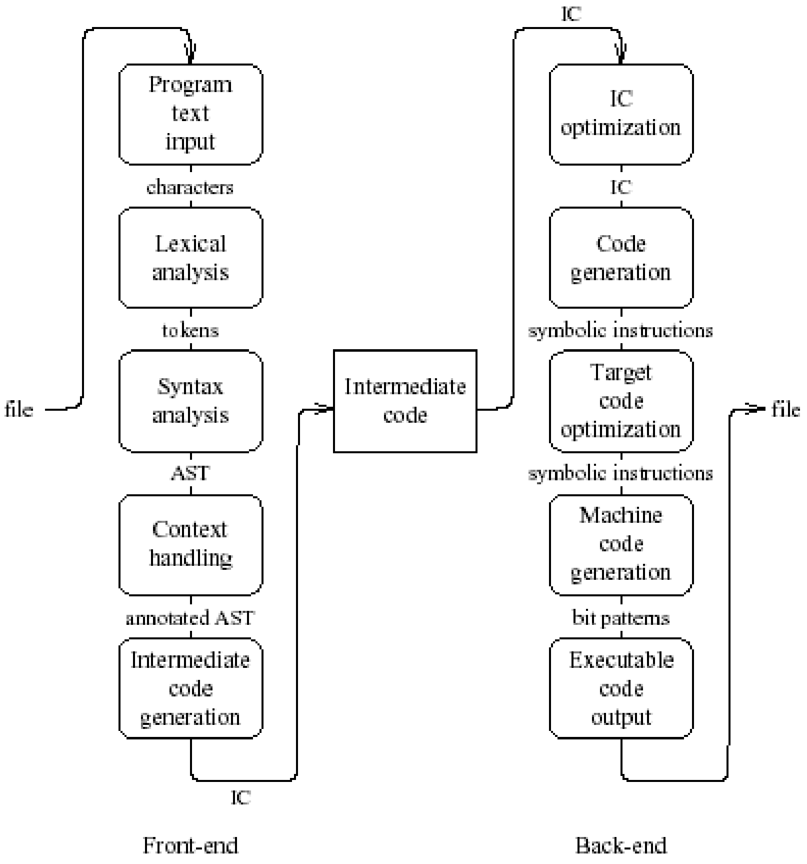
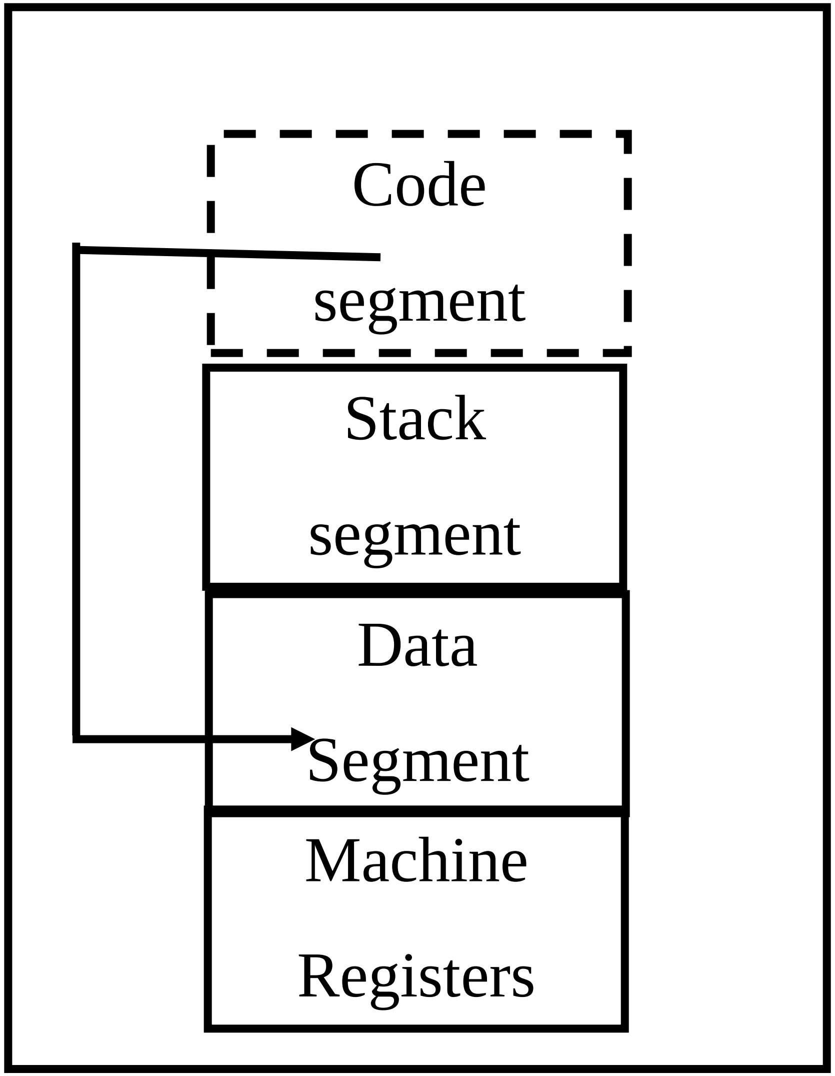


Figure 1.21 Structure of a compiler.

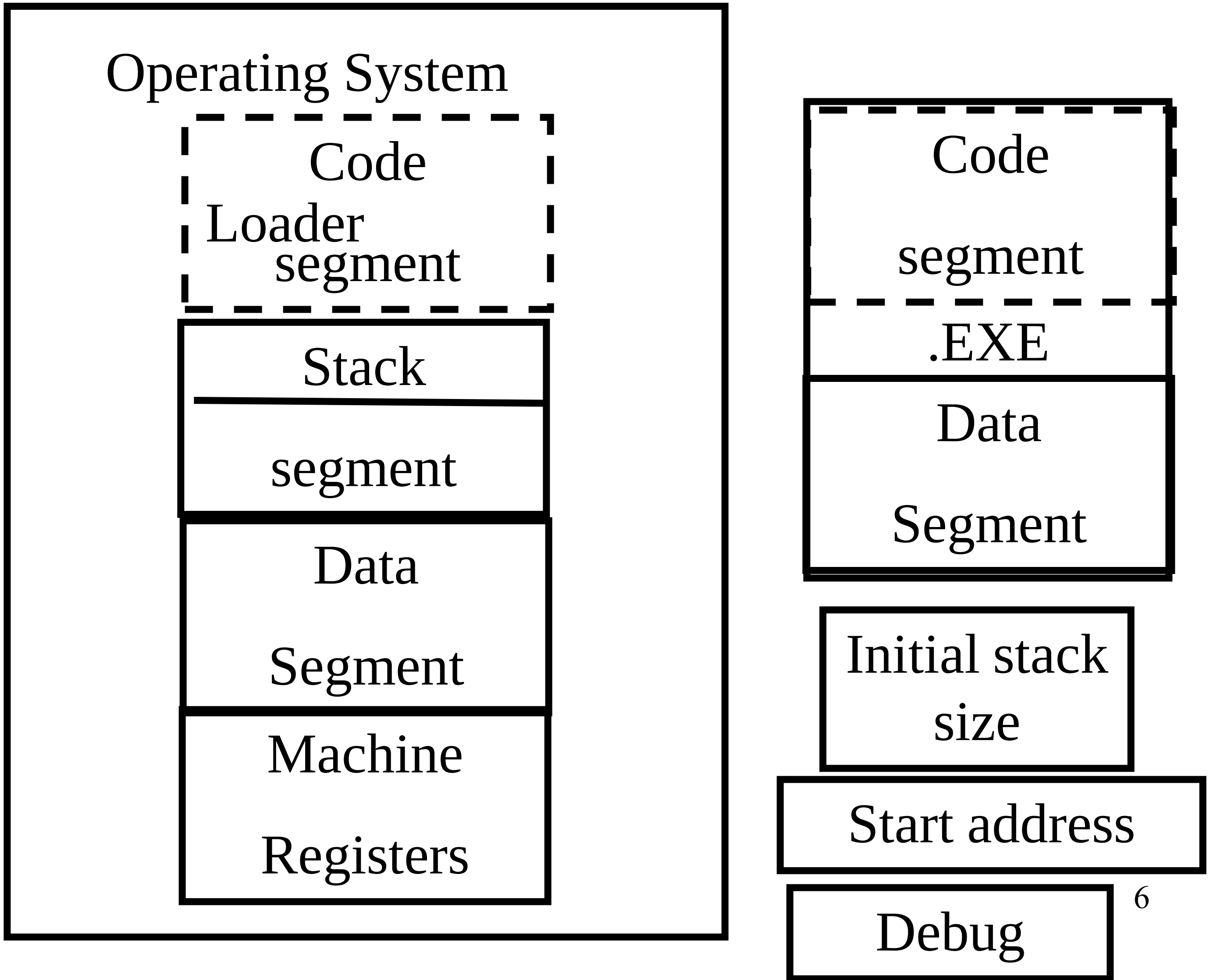
Assembler

- Generate executable code from assembly
- Yet another compiler
- One-to one translation
- Resolve external references
- Relocate code
- How does it fit together?
- Is it really part of the compiler?

Program Runtime State



Program Run



Operating System

Code
Loader
segment

Stack

segment

Data

Segment

Machine

Registers

Code

segment

.EXE

Data

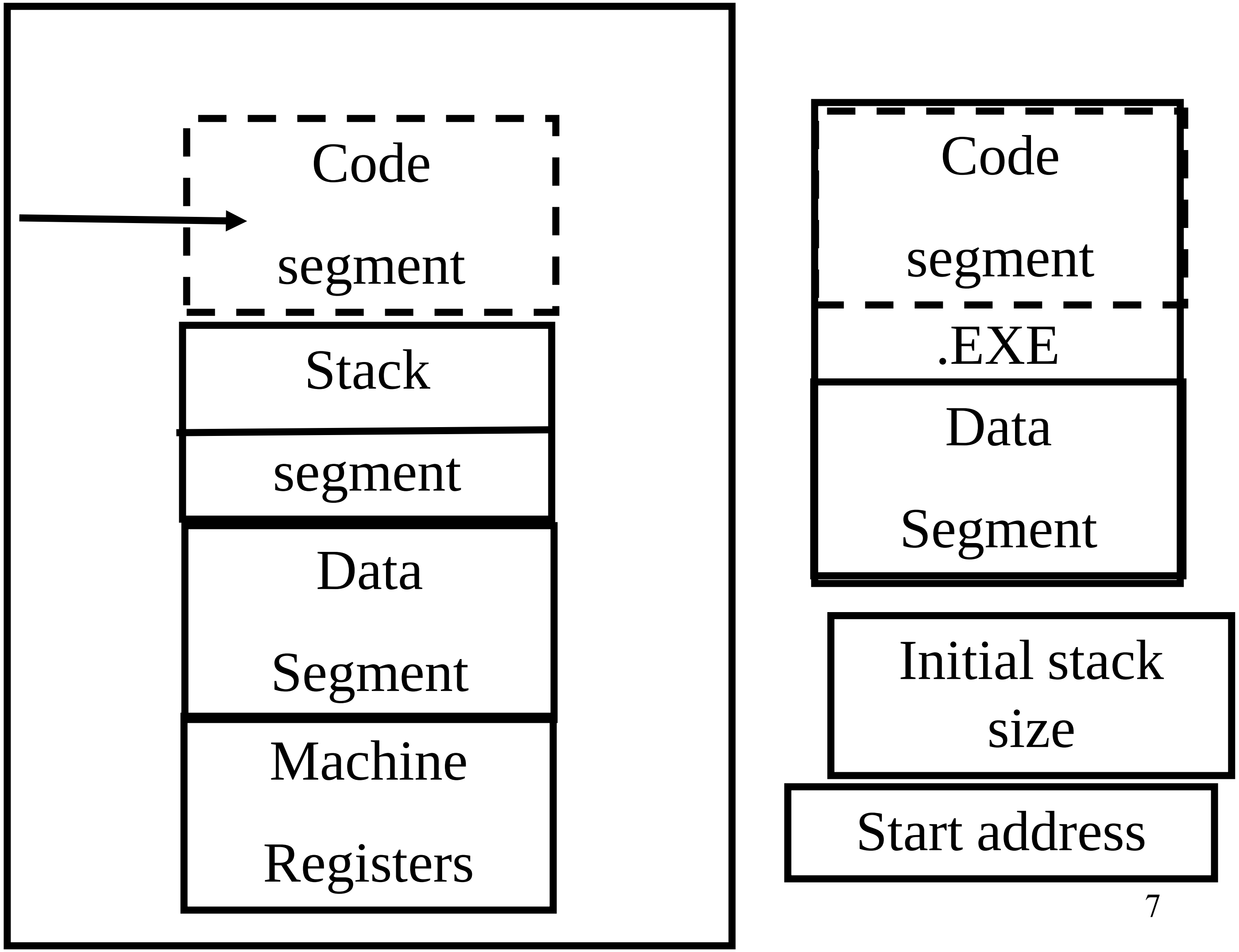
Segment

Initial stack
size

Start address

Debug

Program Run



Code segment

Stack segment

Data Segment

Machine Registers

Code segment

.EXE

Data Segment

Initial stack size

Start address

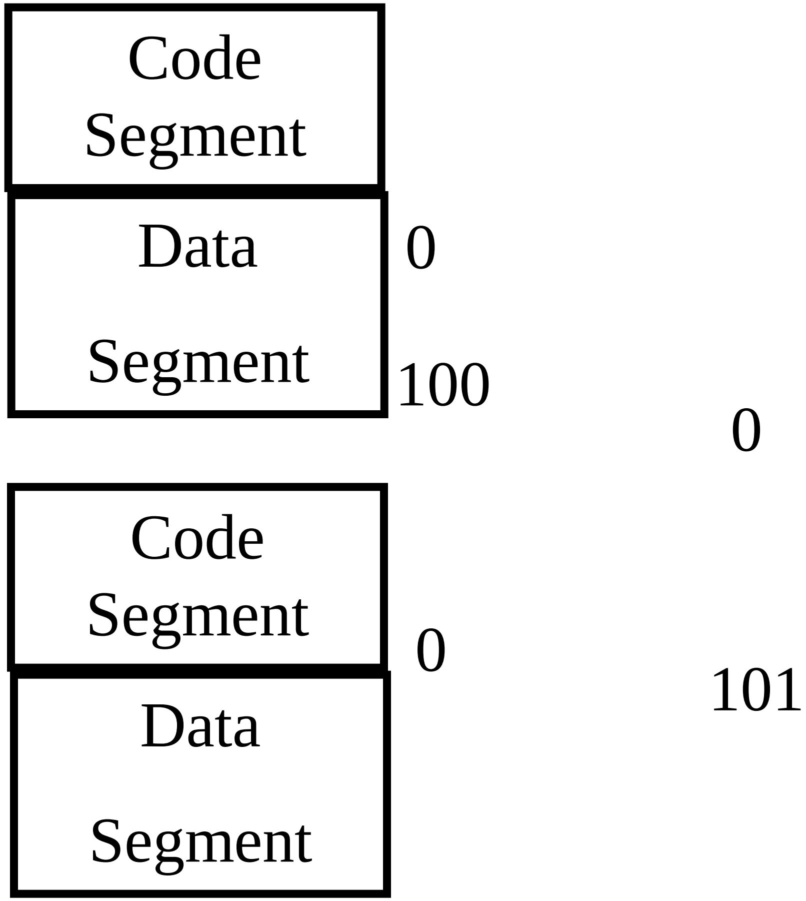
Loader (Summary)

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

Linker

External Symbol Table

Relocation
Bits



Code
Segment

Data
Segment

0

100

0

Code
Segment

Data
Segment

0

101

Linker

- 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

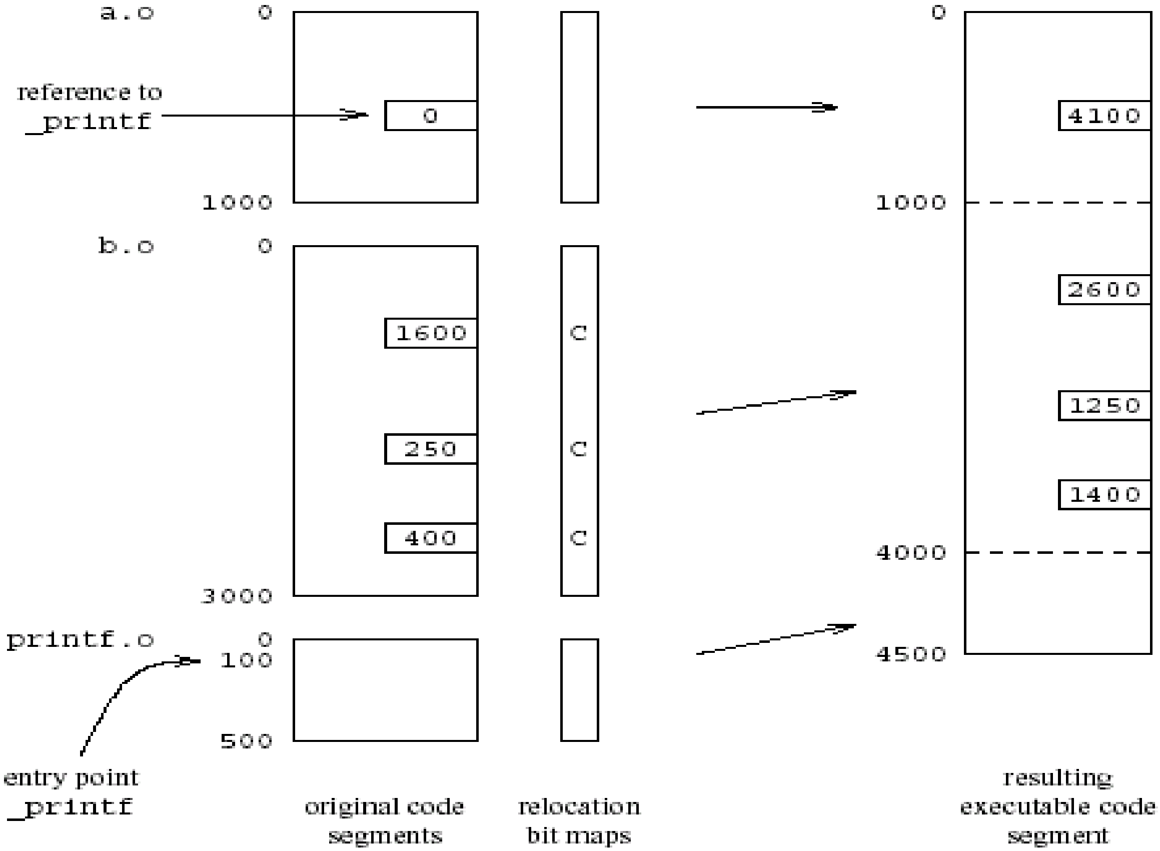
Relocation information

- How to change internal addresses
- 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


a.o
0
reference to
_printf
0
1000
b.o
0
1600
250
400
3000
printf.o
0
100
500
entry point
_printf
c
c
c
0
4100
1000
2600
1250
1400
4000
4500
original code
segments
relocation
bit maps
resulting
executable code
segment

Recap

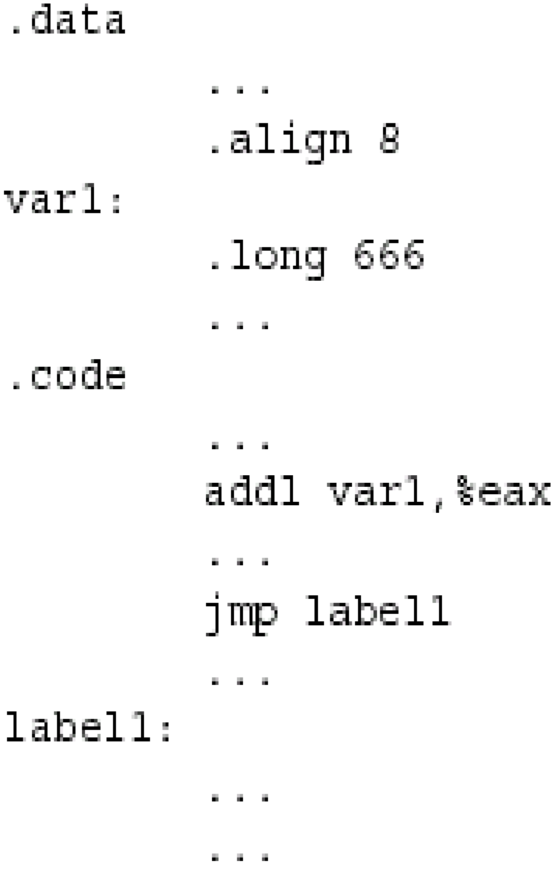
- Assembler generates binary code
  - Unresolved addresses
  - Relocatable addresses
- Linker generates executable code
- Loader generates runtime states (images)

Assembler Design Issues

- Converts symbolic machine code to binary
- One to one conversion

  addl %edx, %ecx ⇒ 000 0001 11 010 001 = 01 D1 (Hex)
- Some assemblers support overloading
  - Different opcodes based on types
- Format conversions
- Handling internal addresses

Handling Internal Addresses

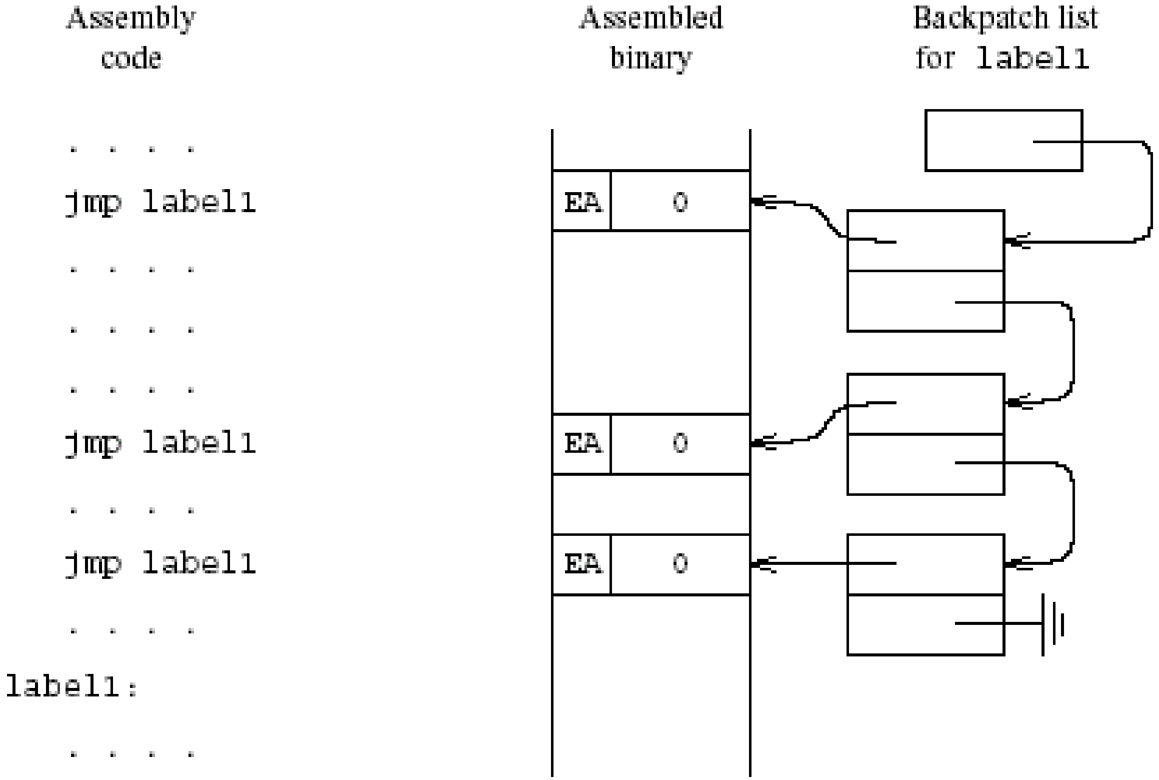


```
.data
        ...
        .align 8
var1:
        .long 666
        ...
.code
        ...
        addl var1,%eax
        ...
        jmp label1
        ...
label1:
        ...
        ...
```

Resolving Internal Addresses

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

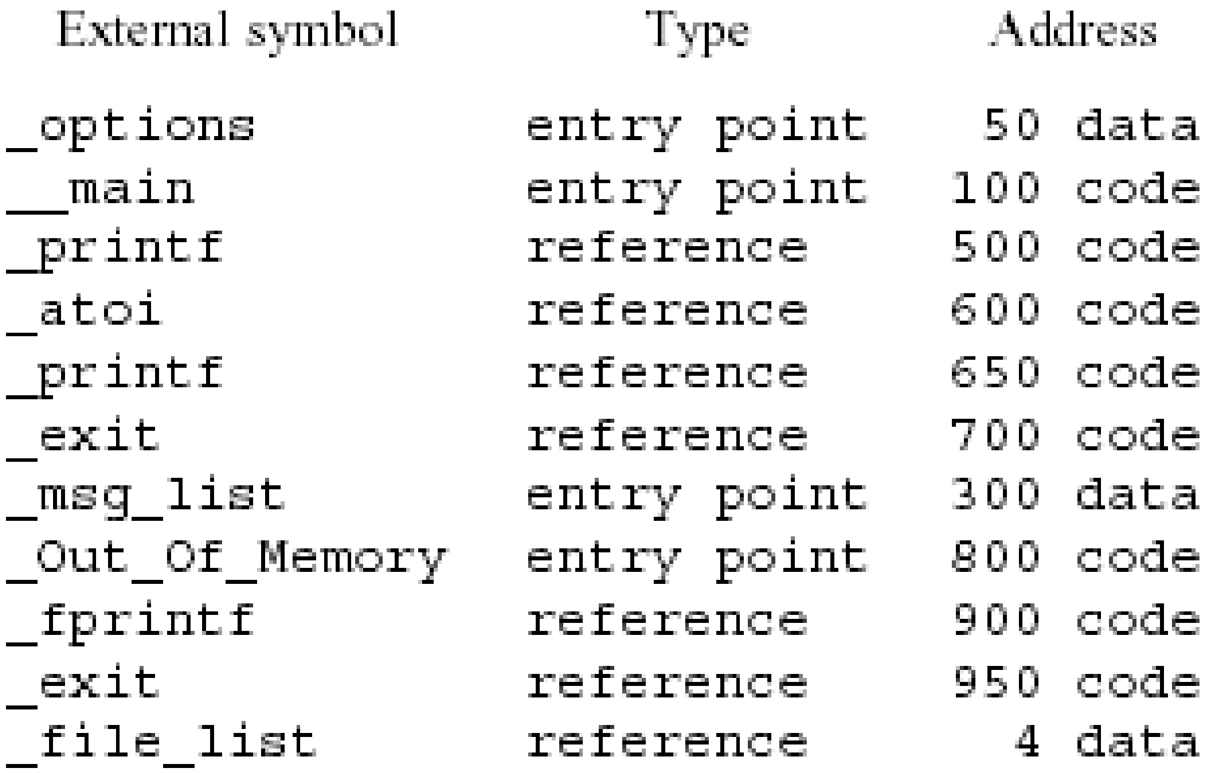
Backpatching



Handling External Addresses

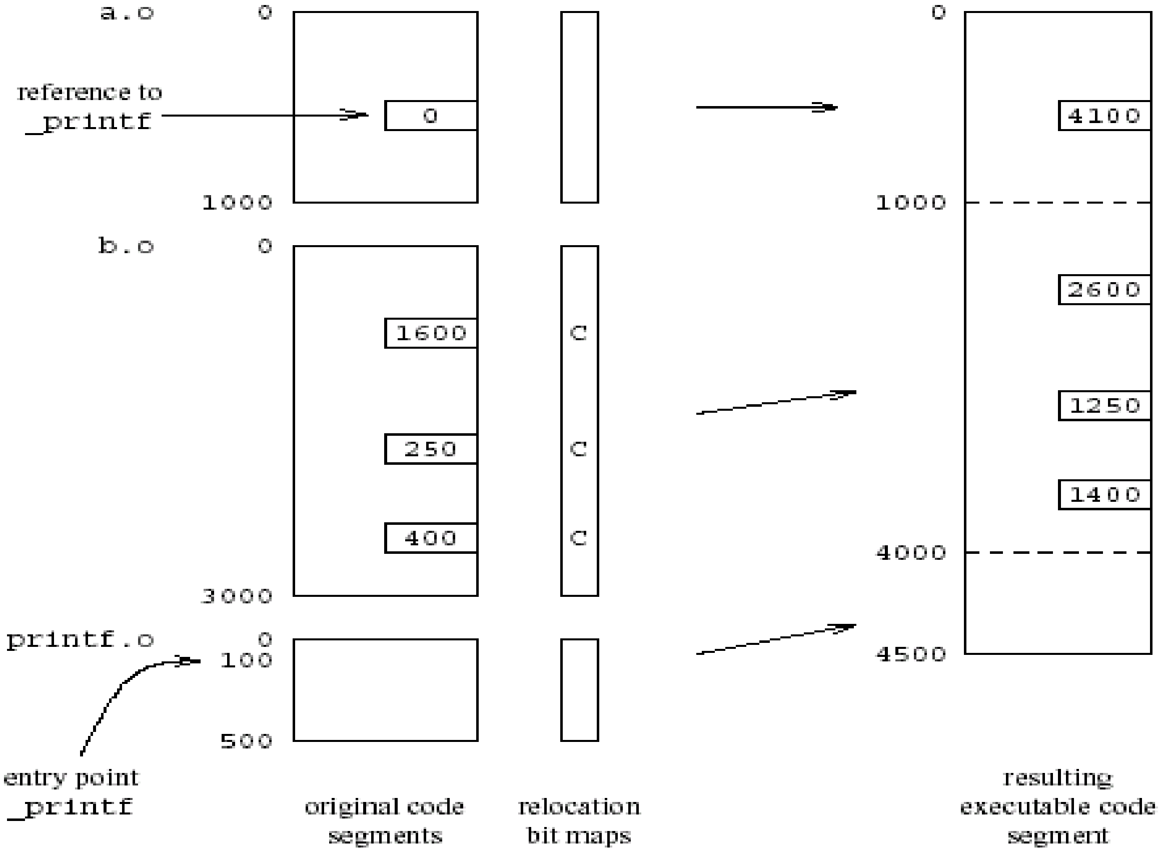
- Record symbol table in external table
- Produce binary version together with the code and relocation bits
- Output of the assembly
  - Code segment
  - Data segment
  - Relocation bits
  - External table

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

Example


a.o
0
reference to
_printf
0
1000
b.o
0
1600
250
400
3000
printf.o
0
100
500
entry point
_printf
C
C
C
original code
segments
relocation
bit maps
0
4100
1000
2600
1250
1400
4000
4500
resulting
executable code
segment

Linker Design Issues

- Append
  - 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

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 program at different addresses
  - Shared libraries
  - Dynamic loading
- Good examples
  - relative jumps
  - reference to activation records
- Bad examples
  - Fixed addresses
    - Global and static data

PIC: The Main Idea

- Keep the data in a table
- Use register to point to the beginning of the table
- Refer to all data relative to the designated register
- But how to set the register?

Per-Routine Pointer Table

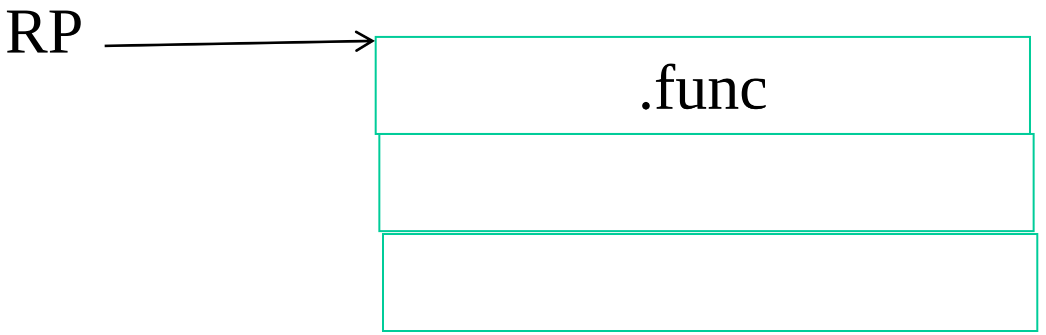
- Store the pointer to the routine in the table

Caller:

1. Load Pointer table address into RP
2. Load Code address from 0(RP) into RC
3. Call via RC

Callee:

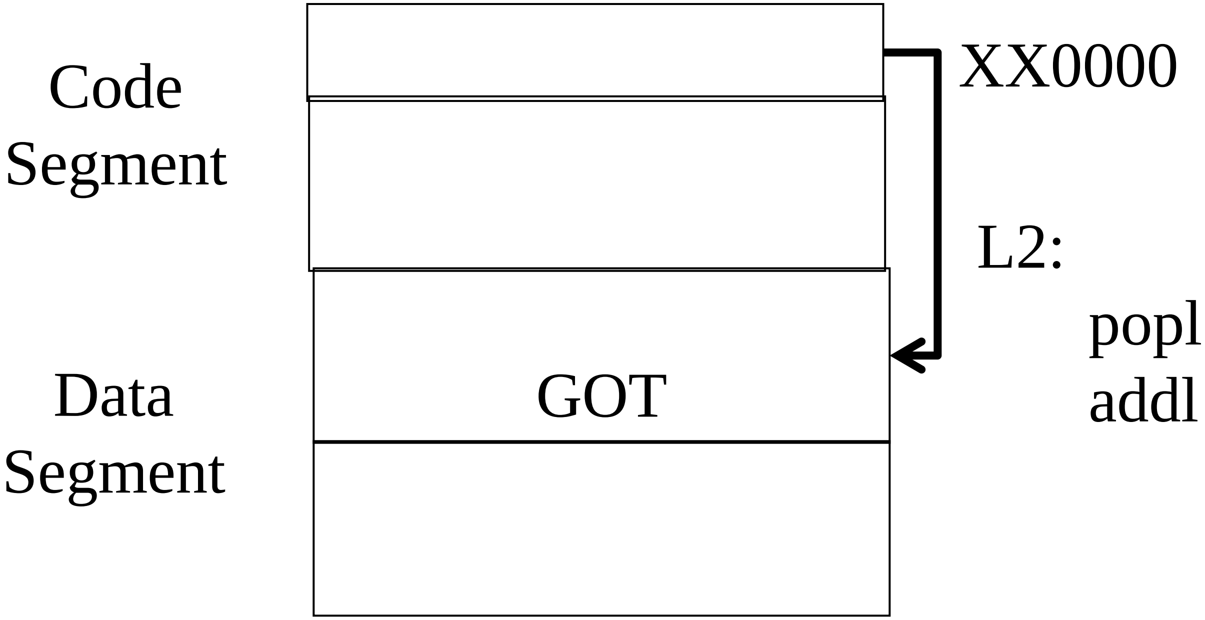
1. RP points to pointer table
2. Table has addresses of pointer table for subprocedures



Other data

ELF-Position Independent Code

- 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



Code Segment

Data Segment

GOT

XX0000

```
                 call L2
L2:
     popl %ebx
     addl $_GOT[.-..L2], %ebx
```

PIC costs and benefits

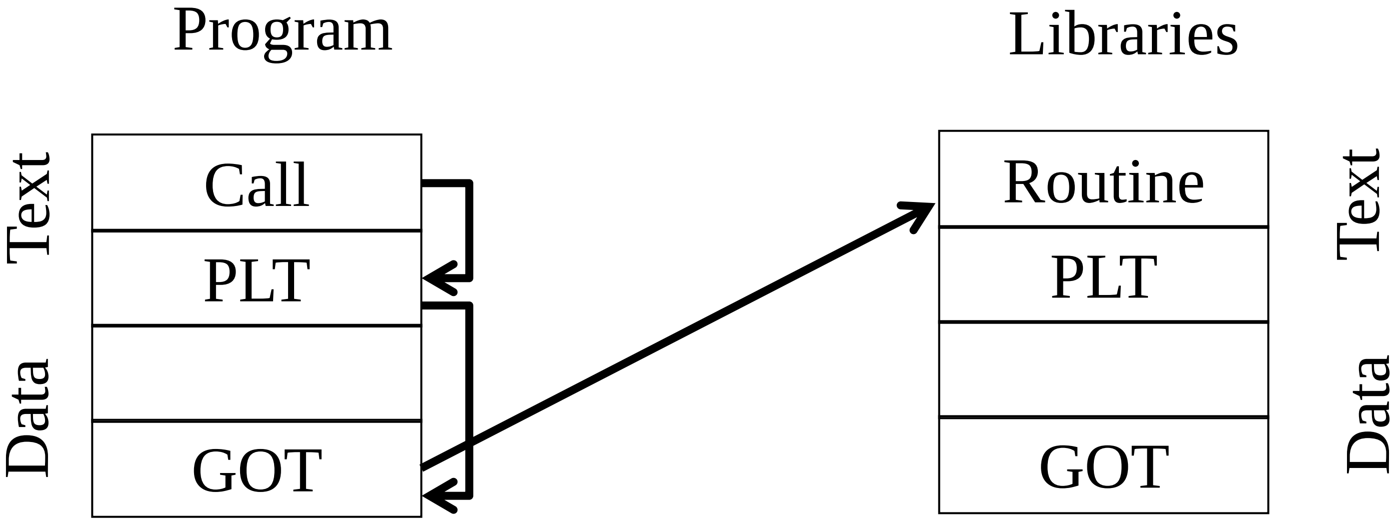
- Enable loading w/o relocation
- Share memory locations among processes

- 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

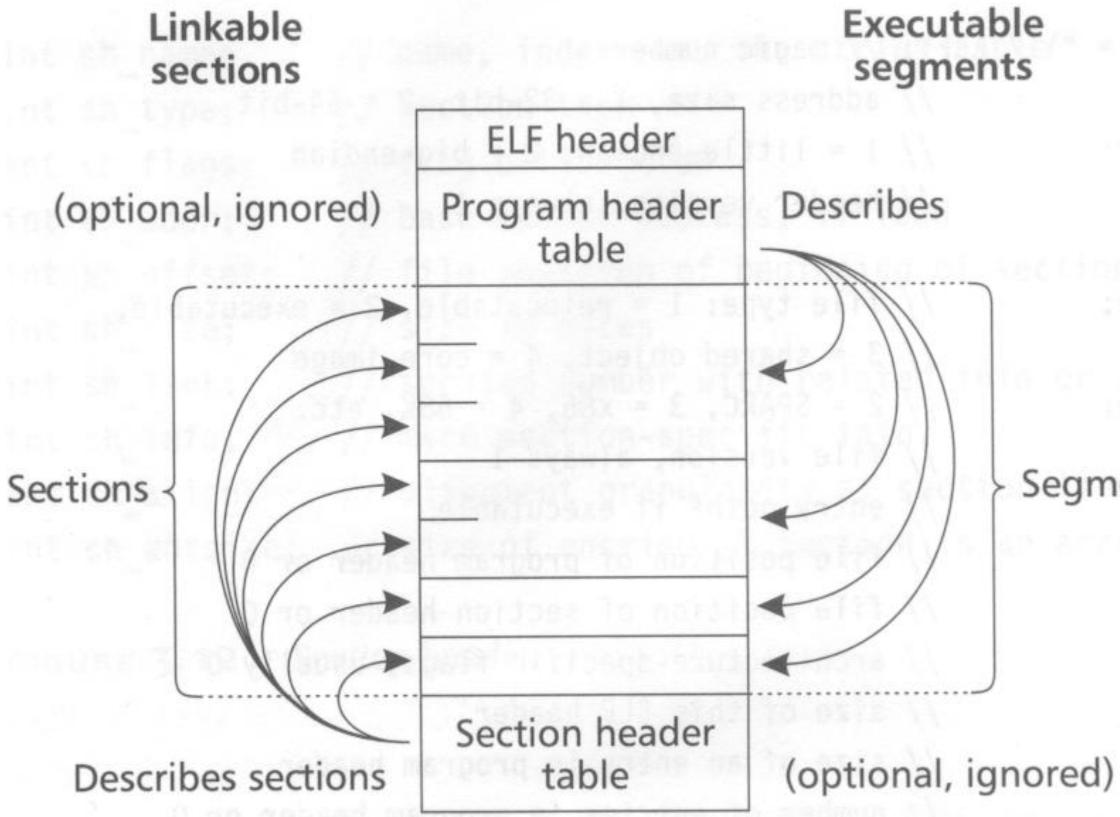


FIGURE 3.10 • Two views of an ELF file.

Consistency

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

Loading Dynamically Linked Programs

- Start the dynamic linker
- Finding 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 it own address
- Supports lazy bindings

Dynamic Linking Approaches

- Unix/ELF uses a single name space 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