GNU Linkers
(Topic 4-2)

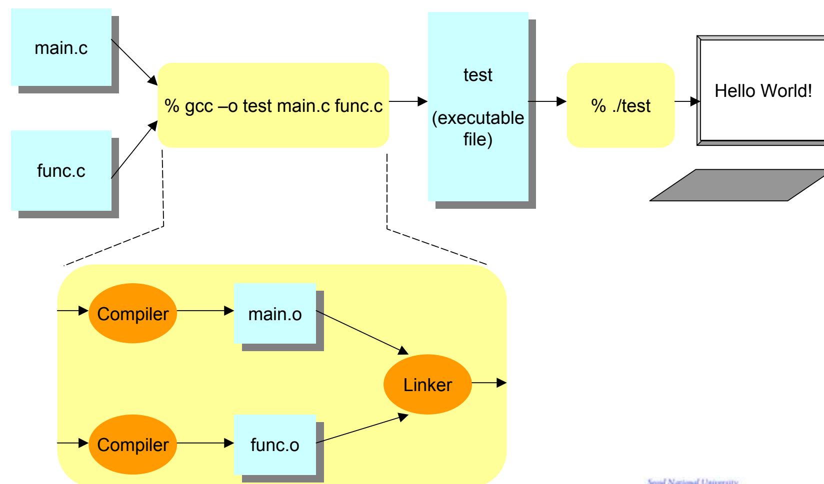
홍 성 수

서울대학교 공과대학 전기 공학부
Real-Time Operating Systems Laboratory

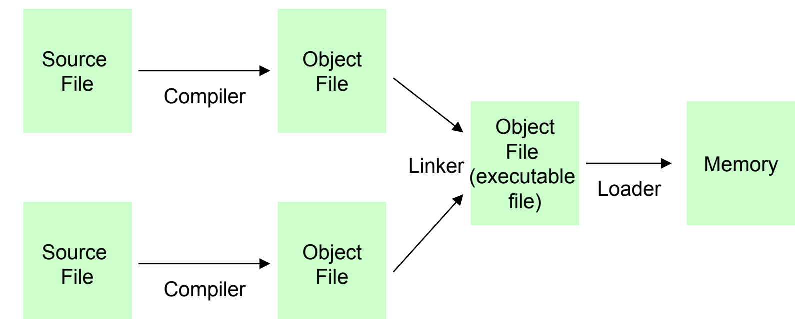
Overview

- Program development process
  - Source file, object file, executable file
  - Compilers, linkers, loaders
- Actual situations
  - Symbol table
  - Relocation table
  - Example

Program Development Process (1)



Program Development Process (2)



Contents in Source, Object Files

- Source file
  - Written in human readable form (ex. C, assembler, …)
  - Ex)

```
int a = 0;
printf("hello world!\n");
mov %eax, $0x0
```

- Object file
  - Binary values to be loaded into memory (instructions, data)
  - Memory addresses to load these binary values
  - Symbol table
    - (function name: address of function)
    - (variable name: address of variable)
  - Relocation table

Compiler, Linker and Loader (1)

- Compiler
  - Convert each source file into object file
- Linker
  - Combine all object files into one object file (or executable file)
  - Specify memory address to load instructions and data
- Loader
  - Read the executable file
  - Extract addresses from the file to load instructions and data
  - Load instructions and data into those positions
  - OS implements the loader function

Compiler, Linker and Loader (2)

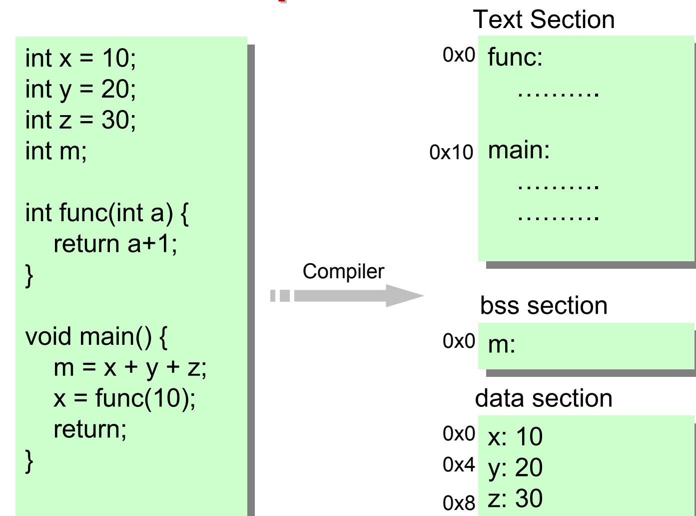
Sections (1)

- What is a section (segment)?
  - Contiguous memory area containing entities with the same property
  - Each section is considered as having private memory
  - Location of symbol is expressed as [section name: offset in section]
- Each object file consists of sections
  - text: contains code to be executed (ex. printf("hello");)
  - data: contains initialized global variables (ex. int a = 0;)
  - bss: contains un-initialized global variables (ex. int b;)
  - other sections (symbol table, relocation table, …)
  - Only text and data sections are loaded into memory

Sections (2)

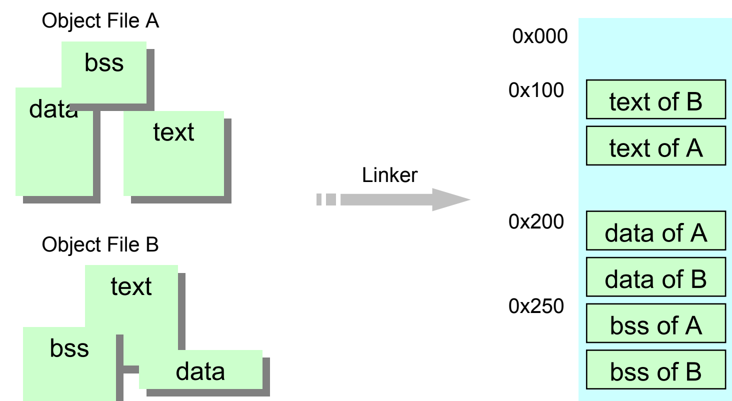
- What is BSS?
  - Acronym for "Block Started by Symbol"
  - Was a pseudo-op in FAP (Fortran Assembly Program), an assembler for the IBM 704-709-7090-7094 machines
    - It defined its label and set aside space for a given number of words.
    - There was another pseudo-op, BES, "Block Ended by Symbol" that did the same except that the label was defined by the last assigned word + 1. (On these machines Fortran arrays were stored backwards in storage and were 1-origin.)
  - The usage is reasonably appropriate, because just as with standard Unix loaders, the space assigned didn't have to be punched literally into the object deck but was represented by a count somewhere.

Example of Sections



Linkers (1)

- Lay out sections from object files in total order



Linkers (2)

- Who specifies this layout?
  - Linker script (or linker command) does.
  - Linker script specifies which section should be loaded in what location
  - Linker reads linker script file when starting, if it is specified
  - If not specified, default linker script file is used instead automatically (usual case)

Loaders

- Read executable file (linked object file)
- Determine addresses to load binary values
  - Specified in executable file
  - Ex) text sections: 0x800 ~ 0x1000
  - Ex) data sections: 0x10100 ~ 0x10200
- Load binary values from executable file into specified memory regions (instructions and initialized data)
- Set PC (Program Counter) to the first address of text section

Problems in Compiling and Linking

- Computer cannot recognize symbolic names
  - symbolic name → memory address
  - using symbol table
- Compiler does not know addresses of sections to be loaded
  - Symbol table alone cannot solve the problem (cross reference)
  - Record which instruction references symbols of other section
  - When linked together, these instructions are fixed to reference actual memory address
  - using relocation table

Symbol Table

- When calling functions or using global variables
  - In source file, symbolic name is used to reference function or variable

```
printf ("hello")
call my_func
mov %eax, my_var
```

  - In object file, computer cannot recognize symbolic name
  - It only understands memory address to call or reference
  - So compiler converts these symbolic names to memory addresses (using symbol table)

```
call 0x103312
mov %eax, 0xF038D
```


Example of Symbol Table

- Symbol Table (symbol name: location)

Symbol Name	Section	Offset
func	text	0x0
main	text	0x10
m	bss	0x0
x	data	0x0
y	data	0x4
z	data	0x8

Example of Symbol Table

- Finding actual memory address of symbol
  - If linker lays out sections like this (according to linker script)
    - Actual memory addresses can be calculated.
    - All symbol references should be replaced by actual addresses

Section	Section
text	0x1000
data	0x3000
bss	0x3500

Symbol Name	Address of symbol
func	0x1000 + 0x00 = 0x1000
main	0x1000 + 0x10 = 0x1010
m	0x3500 + 0x00 = 0x3500
x	0x3000 + 0x00 = 0x3000
y	0x3000 + 0x04 = 0x3004
z	0x3000 + 0x08 = 0x3008

section address
+
offset in section

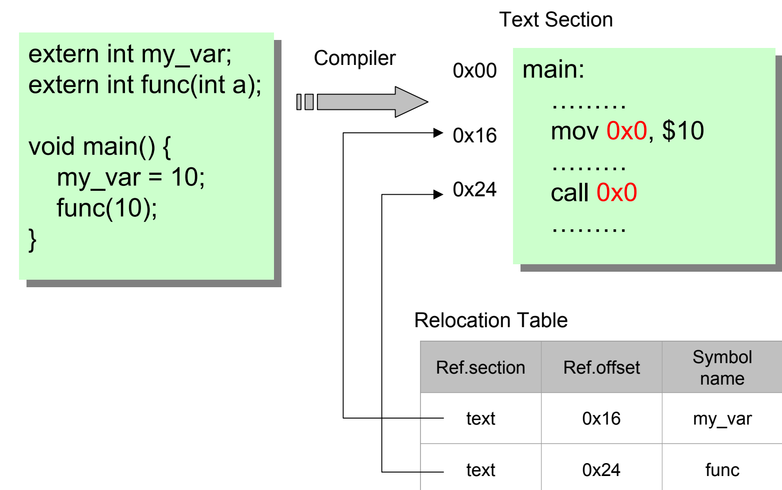
Relocation Table (1)

- Not all symbolic names can be converted to memory addresses by compiler
- Symbols in the same section
  - Can be referenced via PC relative addressing
  - Don't need to know memory addresses in this section
  - This type of relocation can be done by compiler
  - Ex) call local_func → call PC+0x1B

Relocation Table (2)

- Symbols in other sections
  - Cross reference: referencing symbols in other sections
  - Can't be referenced in PC relative addressing mode (why?)
  - Compiler simply assumes addresses of unresolved symbols as zero
  - Instead, compiler provides linker with relocation table to work things out
  - Ex) call printf → call 0x0
- Relocation Table
  - Locations of Instructions that have cross references
  - Symbolic names of those references

Example of Relocation Table



Ref.section	Ref.offset	Symbol name
text	0x16	my_var
text	0x24	func

Relocation and Example

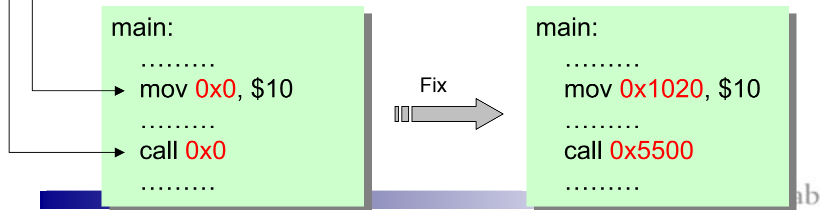
- Linker knows addresses of all symbols
- Fixes addresses of cross referenced symbols (that were zero) to actual addresses

Relocation Table

Ref.section	Ref.offset	Symbol name
text	0x16	my_var
text	0x24	func

Final Symbol Table

Symbol Name	Memory address
my_var	0x1020
func	0x5500



Overall Example

```
main.c:

extern int my_var;
extern int func(int a);

int local_func(int a) {
    return a-10;
}

void main() {
    int result;
    result = local_func(0);
    result = func(10);
    my_var = my_bar+10;
    return;
}
```

```
lib.c:

int my_var = 100;

int func(int a) {
    return a+10;
}
```


Compiling main.c

Symbol Table

Symbol name	section	offset
local_func	text	0x50
main	text	0x0

Relocation Table

Ref.section	Ref.offset	symbol name
text	0x20	func
text	0x30	my_bar
text	0x38	my_bar

Text Section

```
0x00  main:
          ............
0x04      call PC+0x4C
          ............
0x20      call 0x0
          ............
0x30      mov %eax, 0x0
          add %eax, $10
0x38      mov 0x0, %eax
          ............

0x50  local_func:
          ............
```


Compiling lib.c

Symbol Table

Symbol name	section	offset
my_var	data	0x0
func	text	0x0

Relocation Table

<NONE>

Data Section

```
0x00  my_var:
          100
```

Text Section

```
0x00  func:
          .........
          .........
```

Linking main.o and lib.o

- Step 1: Calculate addresses of all symbols

Layout of sections

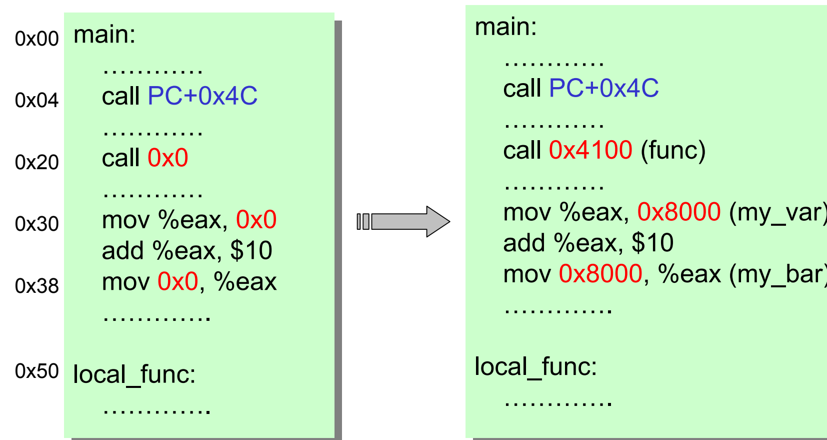
Section	Load Address
main.o (text)	0x4000
lib.o (text)	0x4100
lib.o (data)	0x8000

Final Symbol Table

Symbol name	memory address
main	0x4000 + 0x00 = **0x4000**
local_func	0x4000 + 0x50 = **0x4050**
func	0x4100 + 0x00 = **0x4100**
my_var	0x8000 + 0x00 = **0x8000**

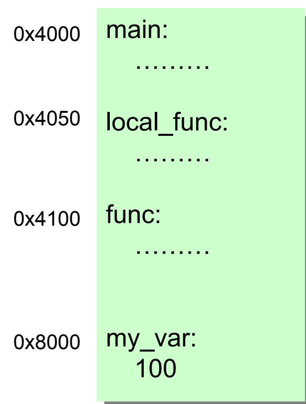
Linking main.o and lib.o

- Step 2: Fix instructions marked in relocation table

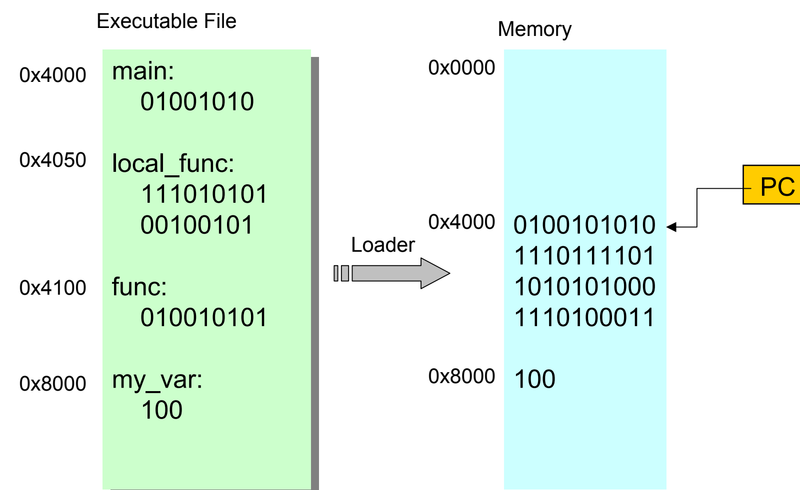


Linking main.o and lib.o

- Step 3: Generate executable file
  - Some additional information
    - Start address: 0x4000



Loading



Project Description

Project Overview

- Goals
  - Understand compiling and linking process
  - Get familiar with GNU development tools
  - Get it done in UNIX system
- C source codes will be provided
- Project 1
  - Compile and link using GNU tools
  - Dump and analyze object and executable files
- Project 2
  - Compile and link by your own hand
  - Compare the result with previous one

Project 1: Description (1)

- Three C source files will be provided
  - (available at homepage)
- Compile to generate object files
  - % gcc –c main.c
  - % gcc –c mylib1.c
  - % gcc –c mylib2.c
  - main.o, mylib1.o, mylib2.o are generated.
- Link object files to generate executable file
  - % gcc –o test main.o mylib1.o mylib2.o or
  - % ld –o test main.o mylib1.o mylib2.o
  - Executable file ‘test’ is generated.

Project 1: Description (2)

- Disassemble each object file and executable file
  - % objdump –D main.o > main.dump
  - % objdump –D mylib1.o > mylib1.dump
  - …
  - % objdump –D test > test.dump
  - In *.dump file, assembler code exists
  - Keep this files.

Project 1: Description (3)

- Disassembled (*.dump) file will look like this


```
Disassembly of section .text:

00000000 <func>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 45 08 0a             addl   $0xa,0x8(%ebp)
   a:   8d 45 0c                lea    0xc(%ebp),%eax
   d:   83 28 14                subl   $0x14,(%eax)
  10:   e8 fc ff ff ff          call   11 <func+0x11>
  15:   8b 45 0c                mov    0xc(%ebp),%eax
  18:   8b 55 08                mov    0x8(%ebp),%edx
  1b:   01 c2                   add    %eax,%edx
  1d:   89 55 fc                mov    %edx,0xfffffffc(%ebp)
  20:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
  23:   c9                      leave
  24:   c3                      ret
Disassembly of section .data:
```


Project 1: Description (4)

- Get section, symbol, and relocation table information for all object files and executable file
  - Section info
    - % objdump –h xxx.o > xxx.section
  - Symbol table
    - % objdump –t xxx.o > xxx.symbol
  - Relocation table
    - % objdump –r xxx.o > xxx.reloc
  - All at once (section, symbol, relocation table)
    - % objdump –x xxx.o > xxx.info

Project 1: Description (5)

- Section Information Example


```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000025  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000005c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000005c  2**2
                  ALLOC
  3 .note         00000014  00000000  00000000  0000005c  2**0
                  CONTENTS, READONLY
  4 .comment      0000002e  00000000  00000000  00000070  2**0
                  CONTENTS, READONLY
```


Project 1: Description (6)

- Symbol Table Example


```
SYMBOL TABLE:
00000000 l    df *ABS*     00000000 test.c
00000000 l    d  .text     00000000
00000000 l    d  .data     00000000
00000000 l    d  .bss      00000000
00000000 l       .text     00000000 gcc2_compiled.
00000000 l    d  .note     00000000
00000000 l    d  .comment  00000000
00000000 g     O .data     00000004 var_b
00000000 g     F .text     00000014 func
00000014 g     F .text     00000014 func2
00000004       O *COM*     00000004 var_a
```

Process 1: Description (7)

- Relocation Table Example


```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_386_32          ext_a
0000000d R_386_PC32        ext_func

RELOCATION RECORDS FOR [.data]:
OFFSET   TYPE              VALUE
00000000 R_386_32          ext_b
```


Project 1: Description (8)

- Print out sections, symbol table, and relocation table for each object file and the executable file.
  - The executable file will not have a relocation table.
  - Sections and a symbol table in the executable file may contain many sections or symbols. You may print only sections and symbols appeared in object files.
- Mark at disassembled file (*.dump)
  - Symbols and their coverage (size)
  - Locations that contain a reference to an unresolved symbol (in object files)
  - Locations that contain a relocated address value by linker (in executable file)

Project 1: Output

- Project 1
  - Disassembled file (*.dump) for each object file and executable file
  - Mark in each file
    - Symbols and relocation addresses

Project 2: Description (1)

- Same as Project 1
  - Same C source file
  - Compile source files (but does not link them into an executable file)
  - Generate disassembled files for each
- In the previous project, default section layout was used
- In current project, ***custom section layout*** is provided (see the web page)

Project 2: Description (2)

- You should do what linkers did the previous project
  - Printout for executable file you generated (***in your head***)
    - Section table
    - Symbol table
    - Tables need not be the same figure as objdump generated table (ex. symbol table only need symbol name, section, offset)
    - Only text, data, bss (COM) sections should be considered (comment, … sections are ignored)
    - Other compiler generated symbols (gcc2_compiled…) should be ignored.
  - In each disassembled object file (*.dump), modify unresolved address to actual address and mark that position.

Project 2: Output

- Project 2
  - Symbol tables for the executable file
  - Relocation tables for the executable file
  - Disassembled file for each object file with unresolved addresses being replaced by actual addresses