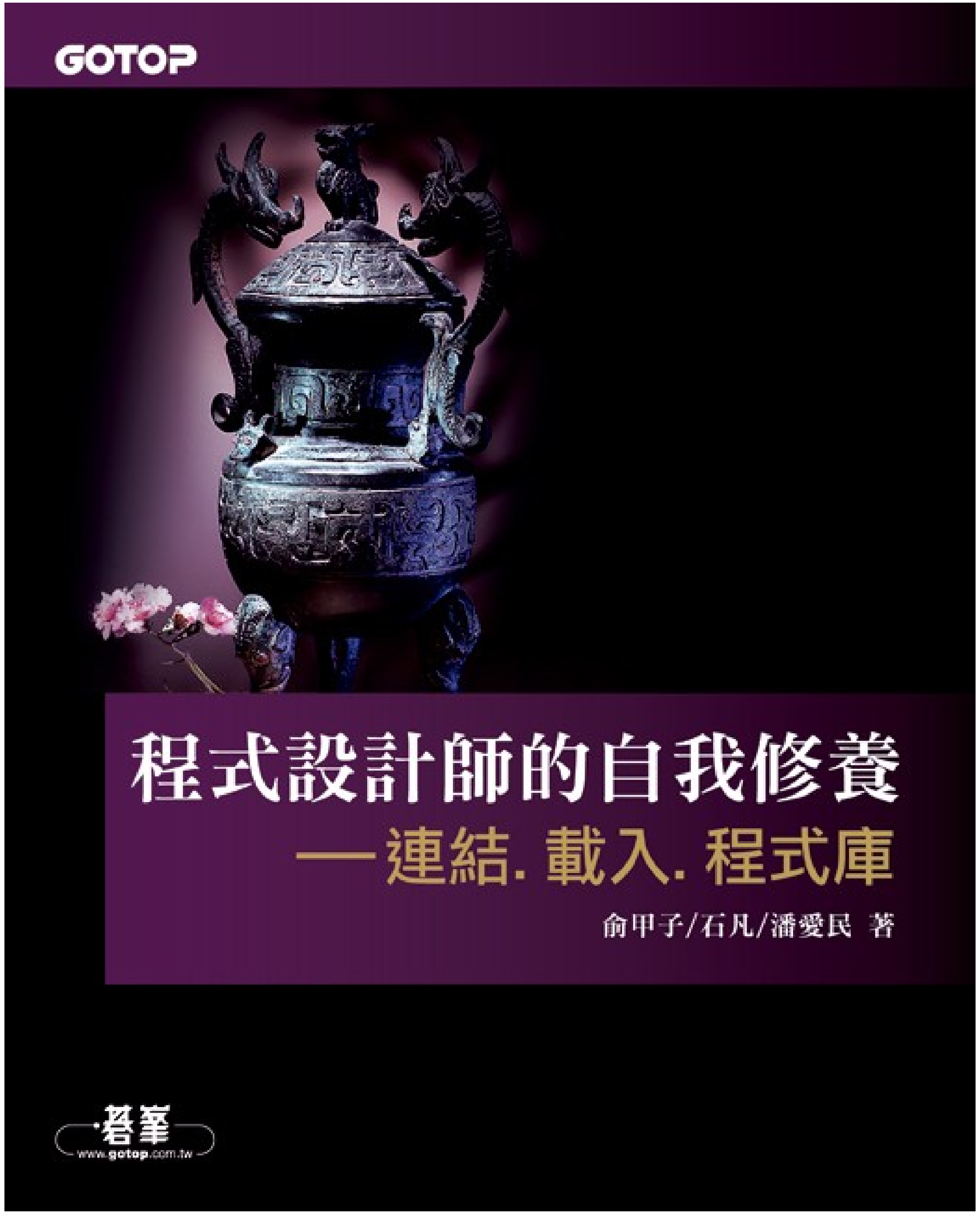
Something behind “Hello World”

Jeff Liaw (廖健富), Jim Huang (黃敬群)

National Cheng Kung University, Taiwan / Apr 14

Outline

- Computer Architecture Review
- Static Linking
  - Compilation & Linking
  - Object File Format
  - Static Linking
- Loading & Dynamic Linking
  - Executable File Loading & Process
  - Dynamic Linking
- Memory
- System Call



GOTOP

程式設計師的自我修養

—連結. 載入. 程式庫

俞甲子/石凡/潘愛民 著

碁峰

www.gotop.com.tw

Hello World!

```
~$ vim hello.c
~$ gcc hello.c
~$ ./a.out
Hello World!
```

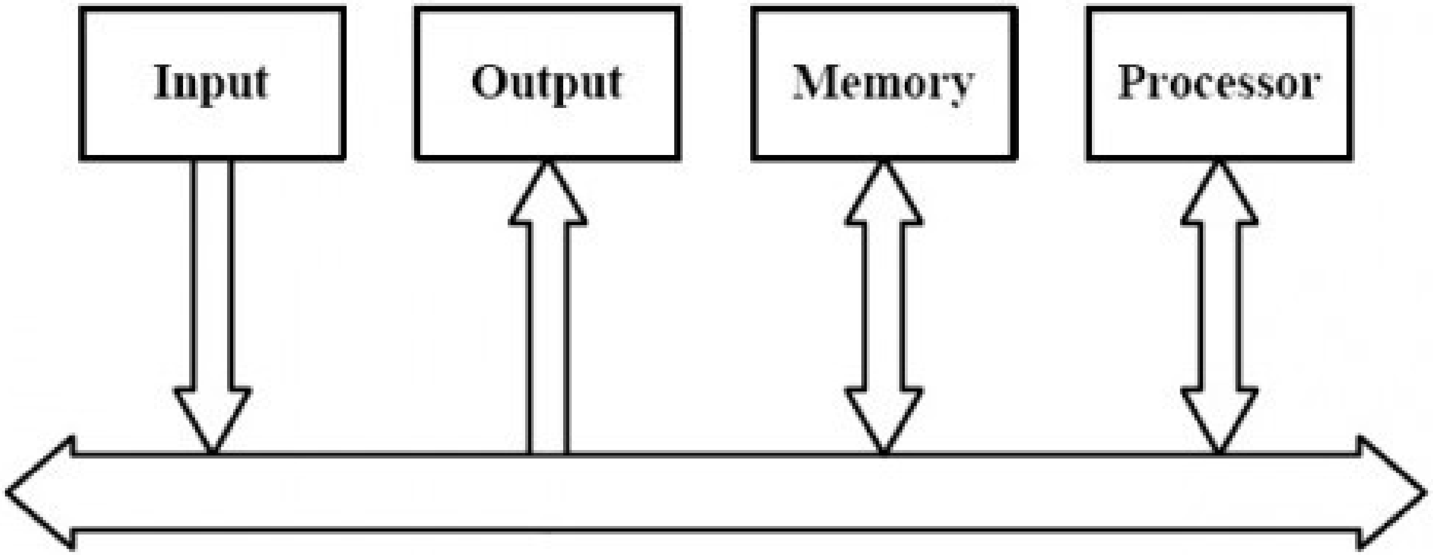
Filename: hello.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf(“Hello World!\n”);

    return 0;
}
```

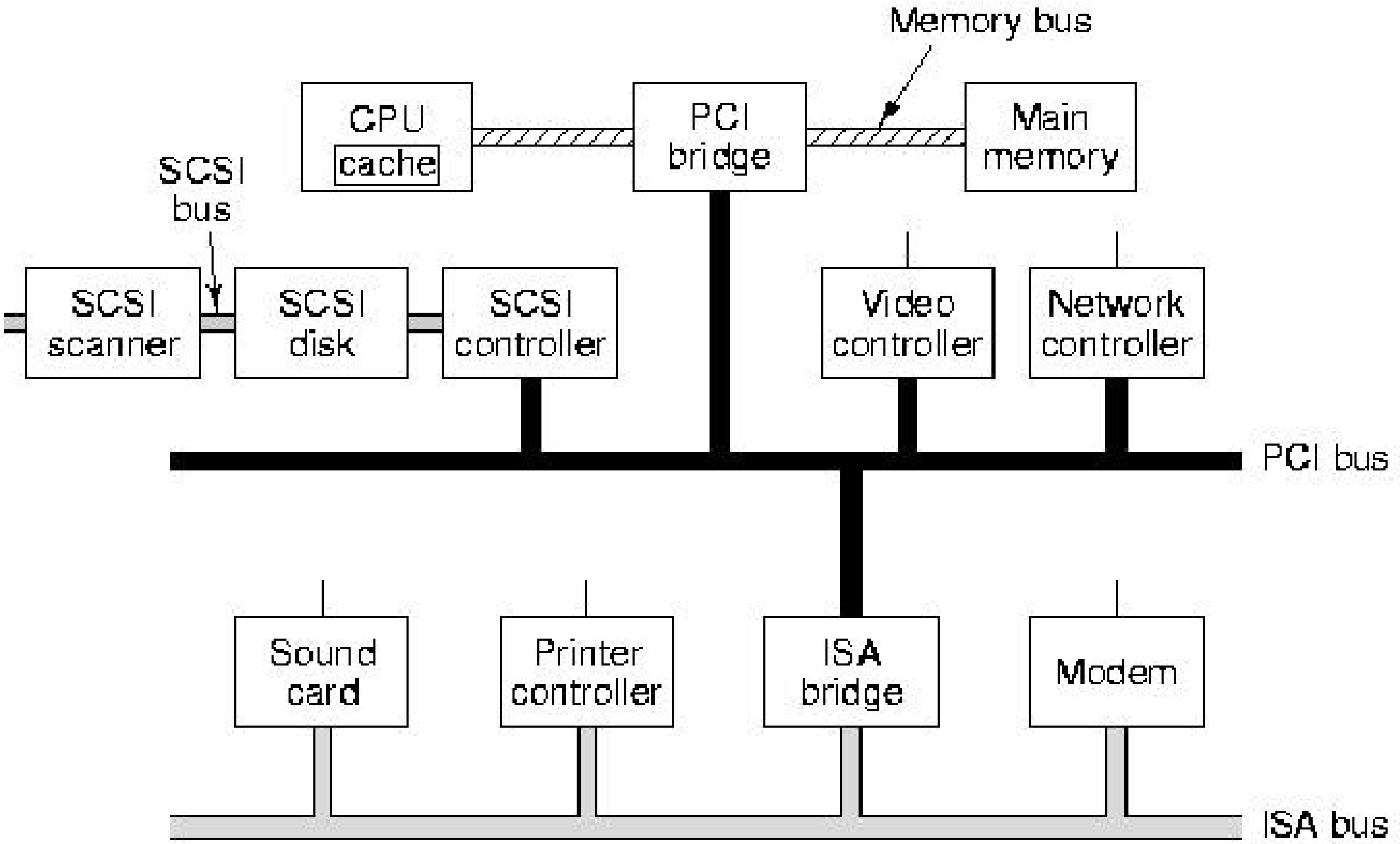
- Why we need to compile the program
- What is in an executable file
- What is the meaning of “#include<stdio.h>”
- Difference between
  - Compiler(Microsoft C/C++ compiler, GCC)
  - Hardware architecture(ARM, x86)
- How to execute a program
  - What does OS do
  - Before main function
  - Memory layout
  - If we don’t have OS

Computer Architecture Review

Computer Architecture



Computer Architecture

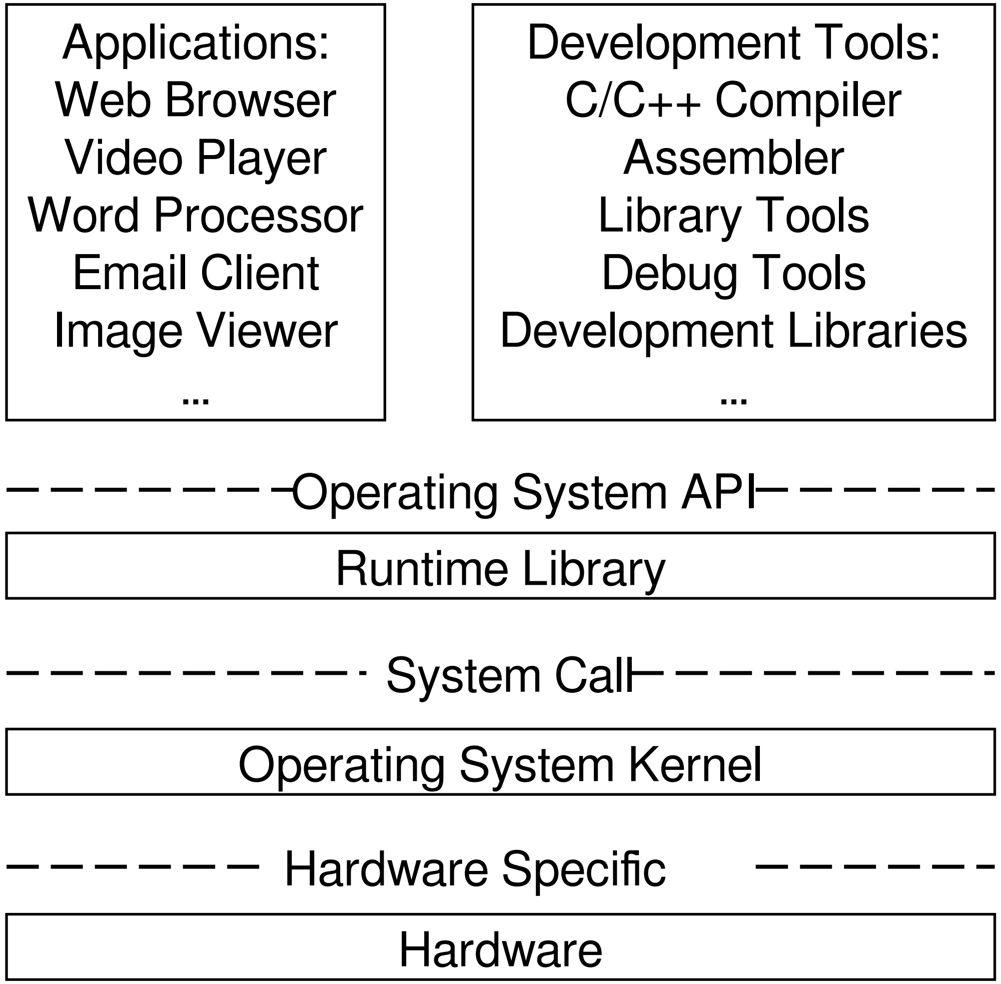


SMP & Multi-core Processor

- Symmetrical Multi-Processing
  - CPU number↑ → Speed ↑?
  - A program can not be divided multiple independent subprogram
- Server application
- Multi-core Processor
  - Share caches with other processor

Software Architecture

- Any problem in computer science can be solved by another layer of indirection
- API: Application Programming Interface
- System call interface
- Hardware specification



Applications:
Web Browser
Video Player
Word Processor
Email Client
Image Viewer
...

Development Tools:
C/C++ Compiler
Assembler
Library Tools
Debug Tools
Development Libraries
...

Operating System API

Runtime Library

System Call

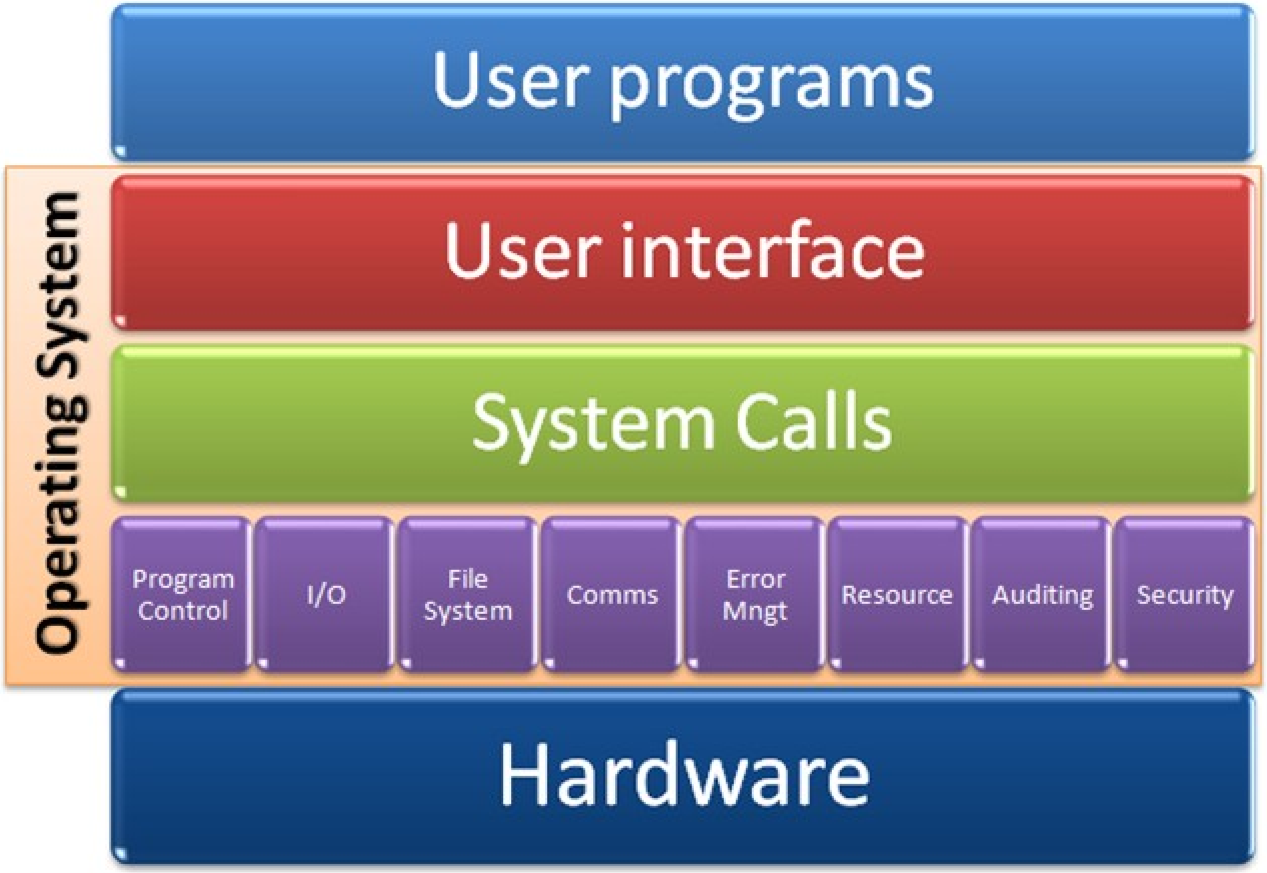
Operating System Kernel

Hardware Specific

Hardware

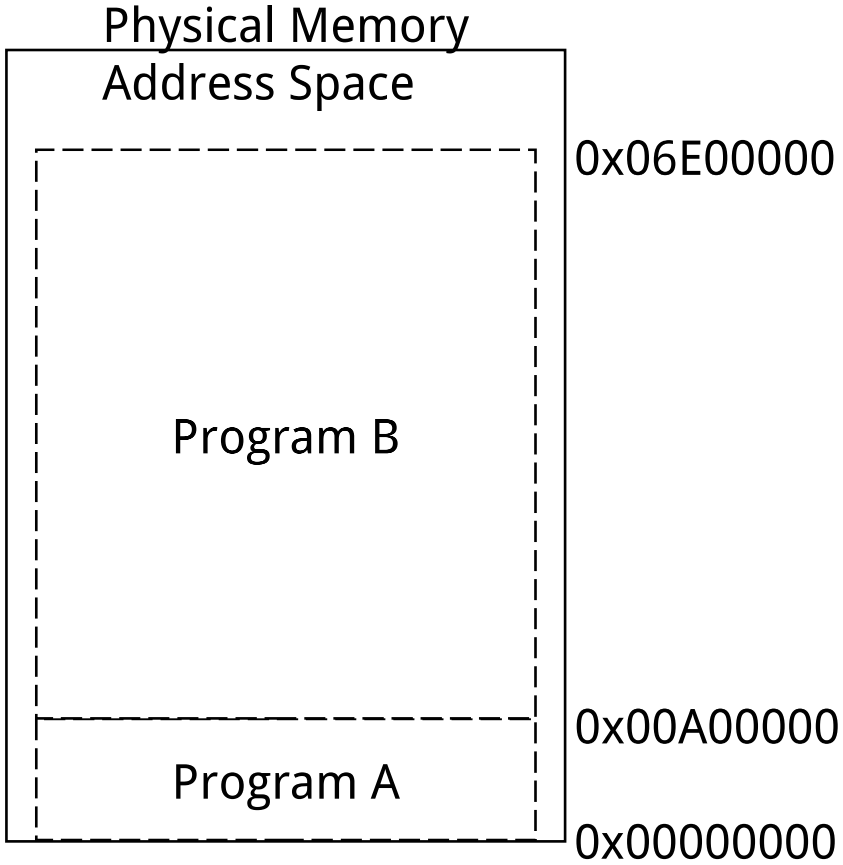
Operating System

- Abstract interface
- Hardware resource
  - CPU
    - Multiprogramming
    - Time-Sharing System
    - Multi-tasking
      - Process
      - Preemptive
  - Memory
  - I/O devices
    - Device Driver



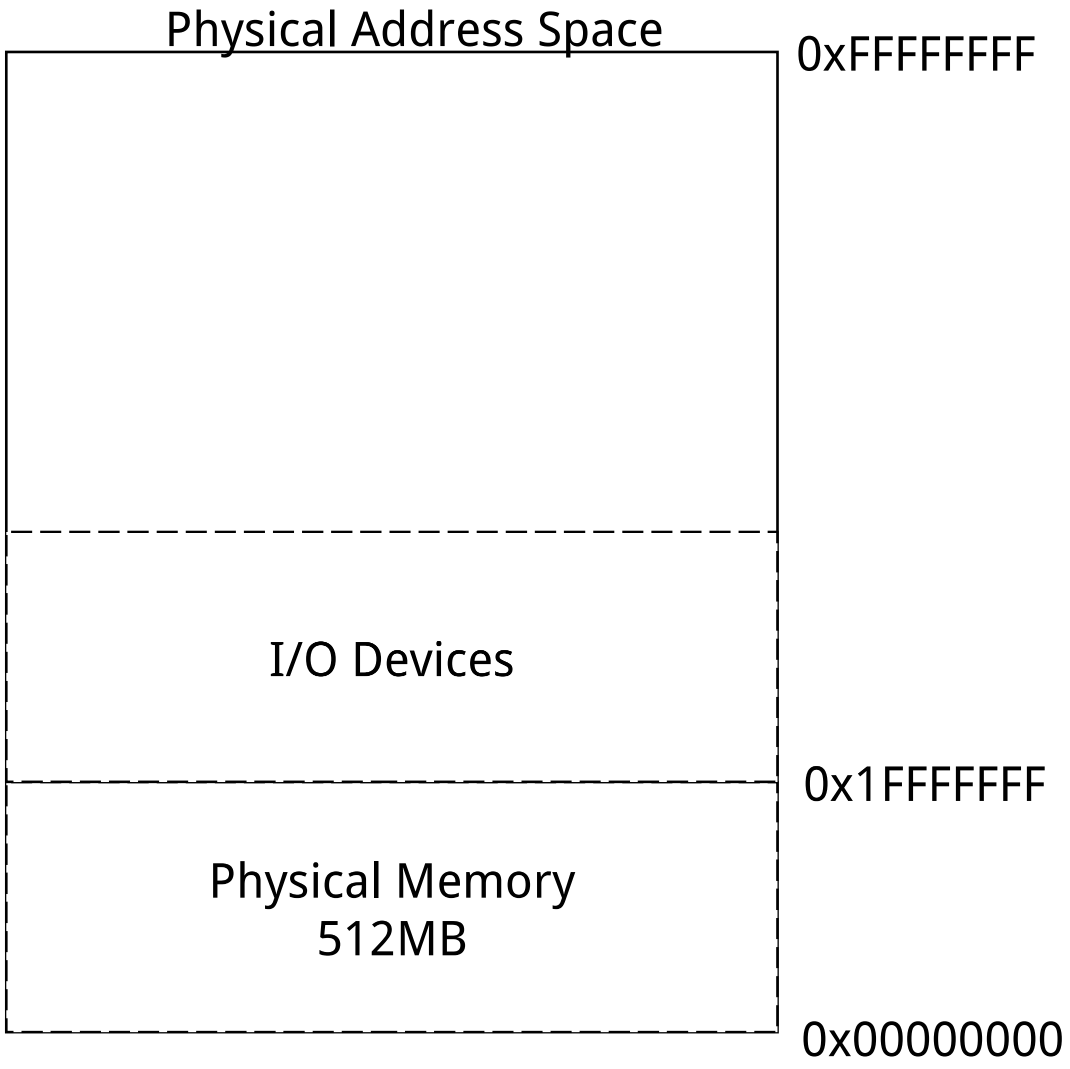
Memory

- How to allocate limited physical memory to lots of programs?
  - Assume we have 128MB physical memory
  - Program A needs 10MB
  - Program B needs 100MB
  - Program C needs 20MB
- Solution 1
  - A gets 0~10MB, B gets 10~110MB
  - No address space isolation
  - Inefficiency
  - Undetermined program address



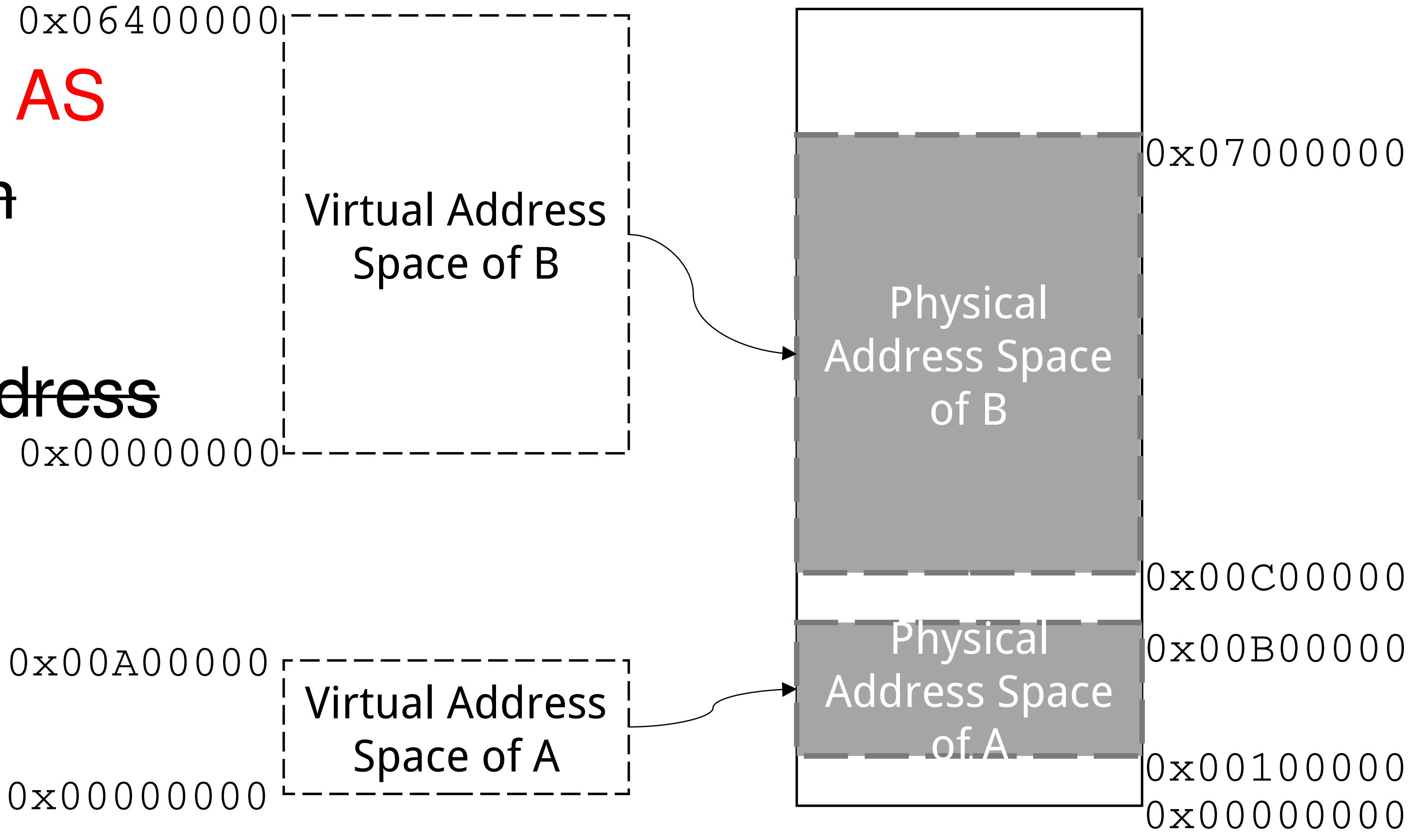
Address Space Isolation

- Own the whole computer
  - CPU, Memory
- Address Space(AS)
  - Array - depends on address length
    - 32bit system →
    - 0x0000000 ~ 0xFFFFFFFF
  - Virtual Address Space
    - Imagination
    - Process use their own virtual address space
  - Physical Address Space



Segmentation

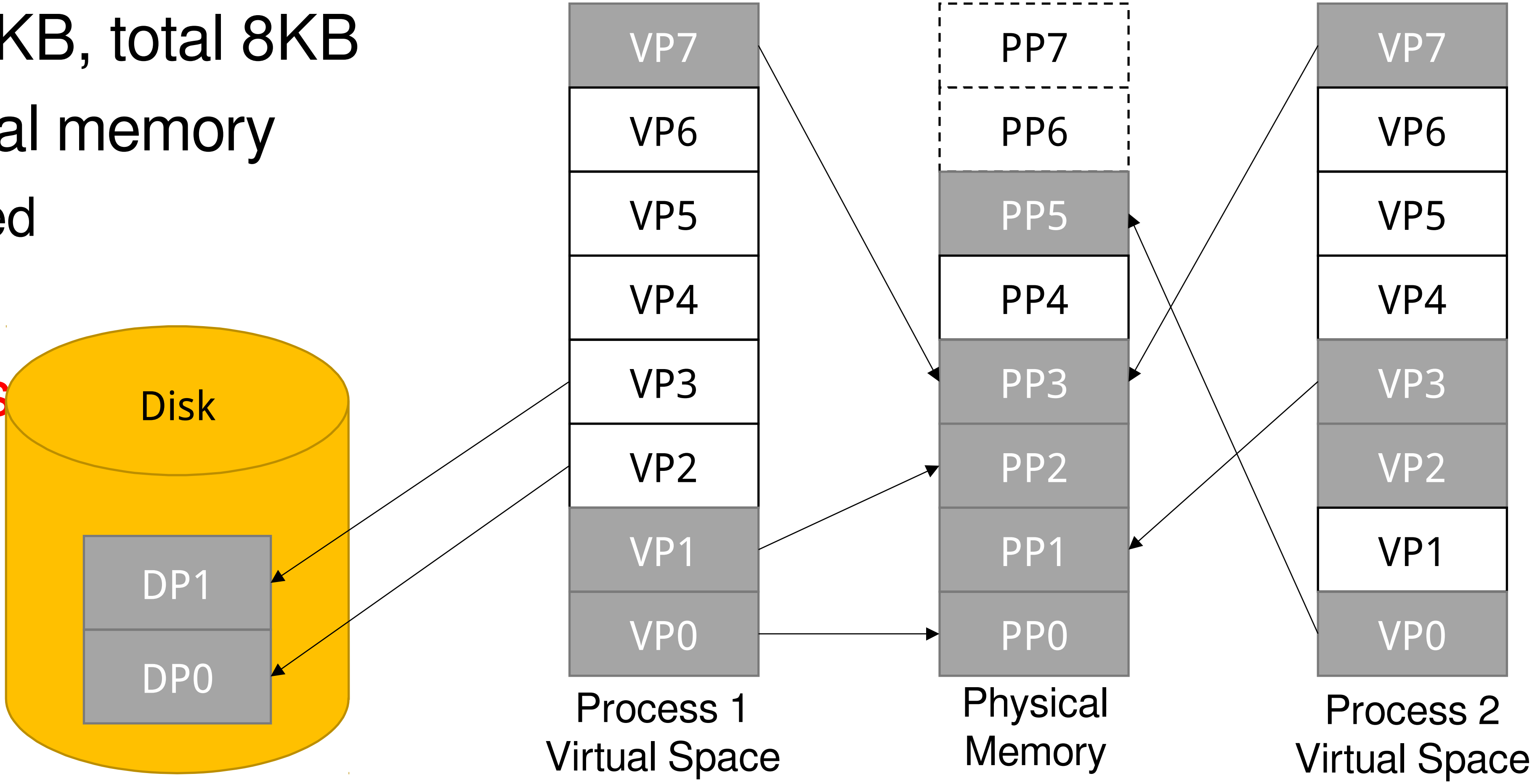
- Virtual AS map to Physical AS
- ~~No address space isolation~~
- Inefficiency
- ~~Undetermined program address~~



0x06400000
Virtual Address Space of B
0x00000000
0x00A00000
Virtual Address Space of A
0x00000000
Physical Address Space of B
0x07000000
0x00C00000
Physical Address Space of A
0x00B00000
0x00100000
0x00000000

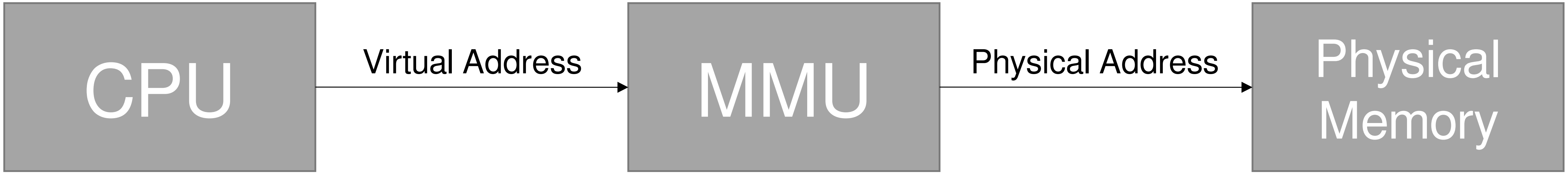
Paging

- Frequently use a small part(locality)
- 8 pages, each 1 KB, total 8KB
- Only 6KB physical memory
  - PP6, PP7 unused
- Page Fault
- Access attributes
  - Read
  - Write
  - Execute



MMU

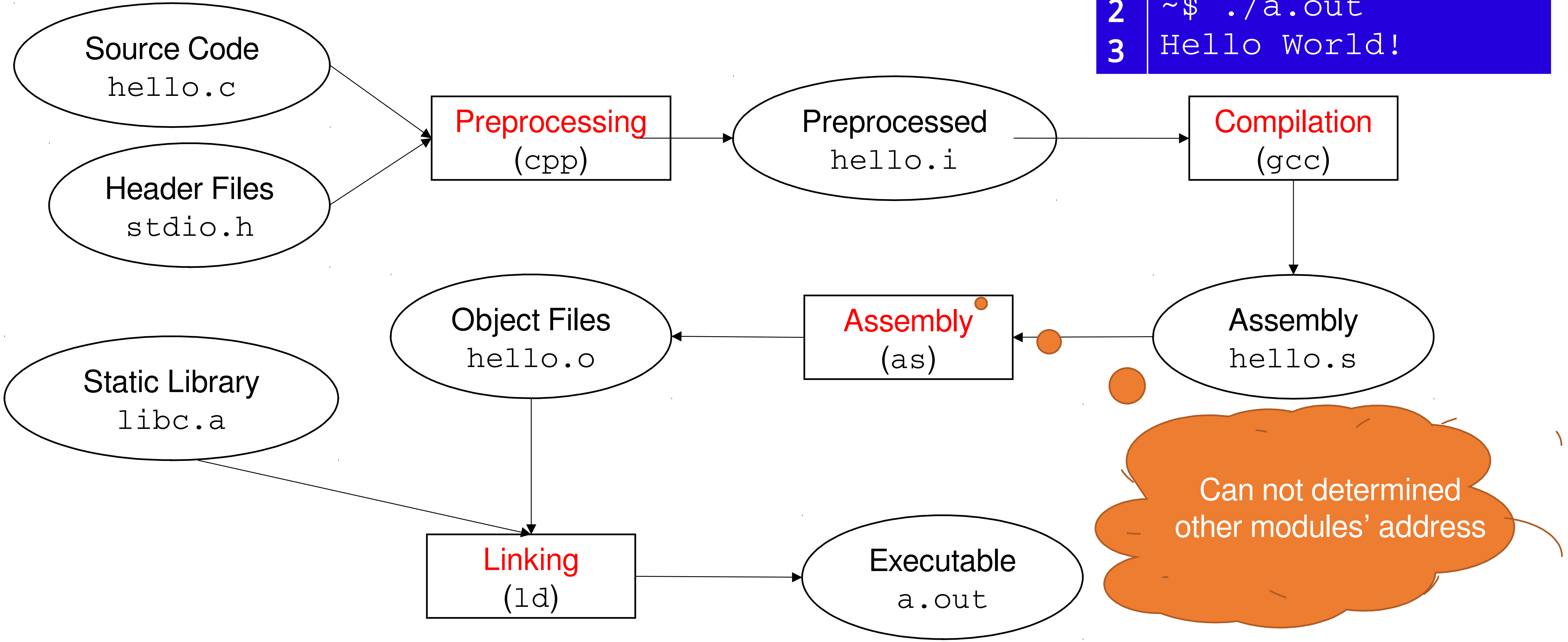
- Memory Management Unit
- Usually place on CPU board



Compilation & Linking

Hello World!

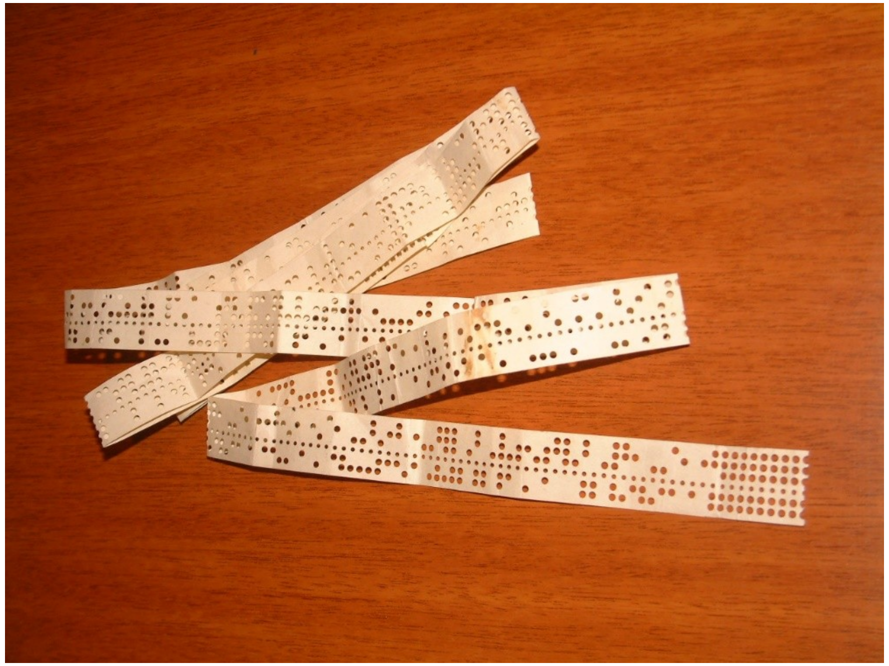
```
~$ vim hello.c
~$ gcc hello.c
~$ ./a.out
Hello World!
```



Can not determined other modules' address

Relocation

- Punched tape
- An architecture with
  - instruction → 1 byte(8 bits)
  - jump → 0001 + jump address
  - Manually modify address → impractical
- Define Symbols(variables, functions)
  - define label “foo” at line 4
  - jump to label “foo”
  - Automatically modify symbol value

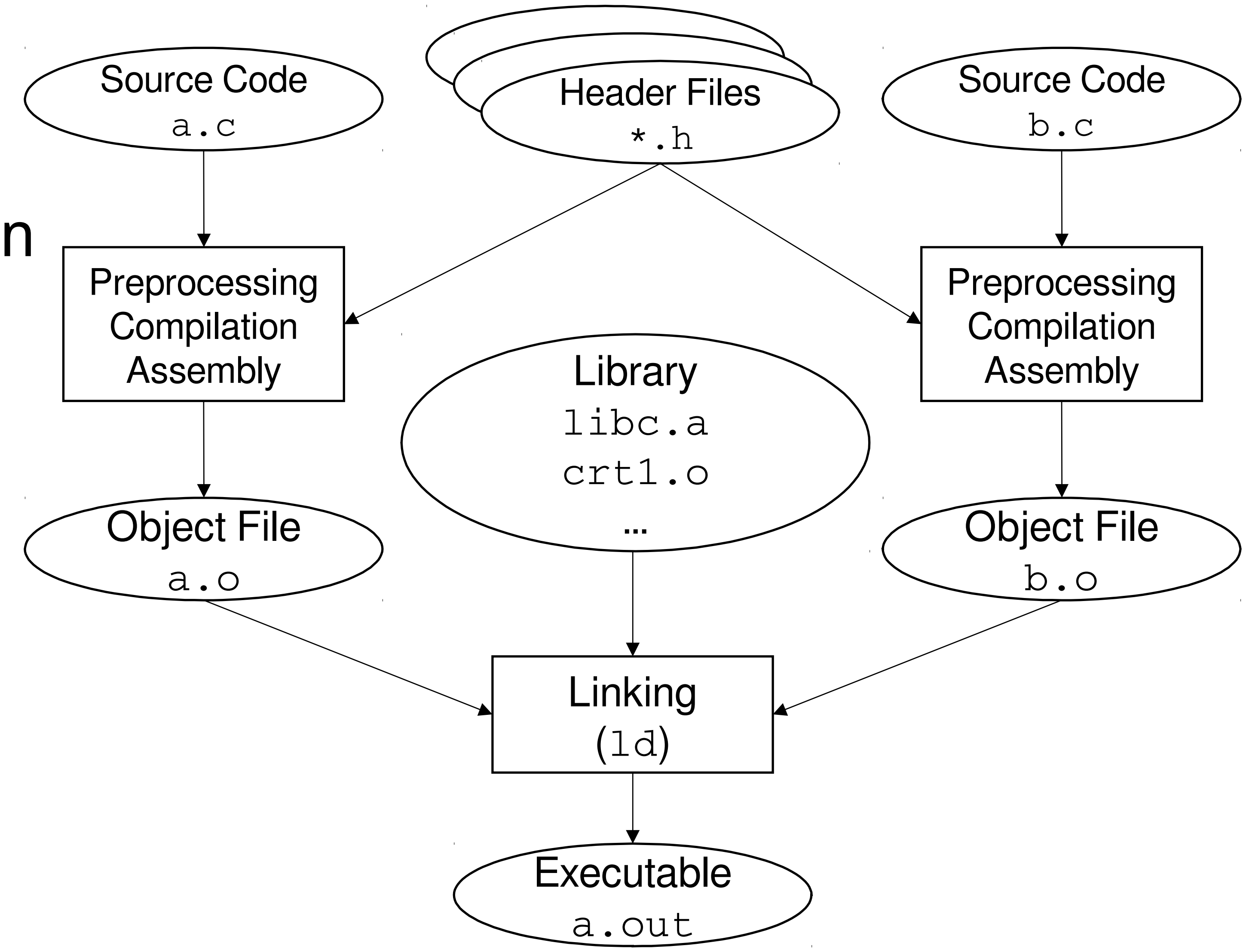


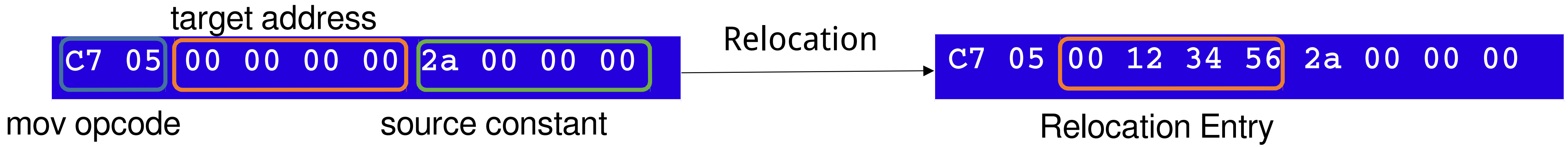
0	0001 0100
1	…
2	…
3	…
4	1000 0111
5	…

Linking

- Address and Storage Allocation
- Symbol Resolution
- Relocation

```
/* a.c */
int var;

/* b.c */
extern int var;
var = 42;

/* b.s */
movl $0x2a, var
```





```
C7 05 00 00 00 00 2a 00 00 00
```

```
C7 05 00 12 34 56 2a 00 00 00
```

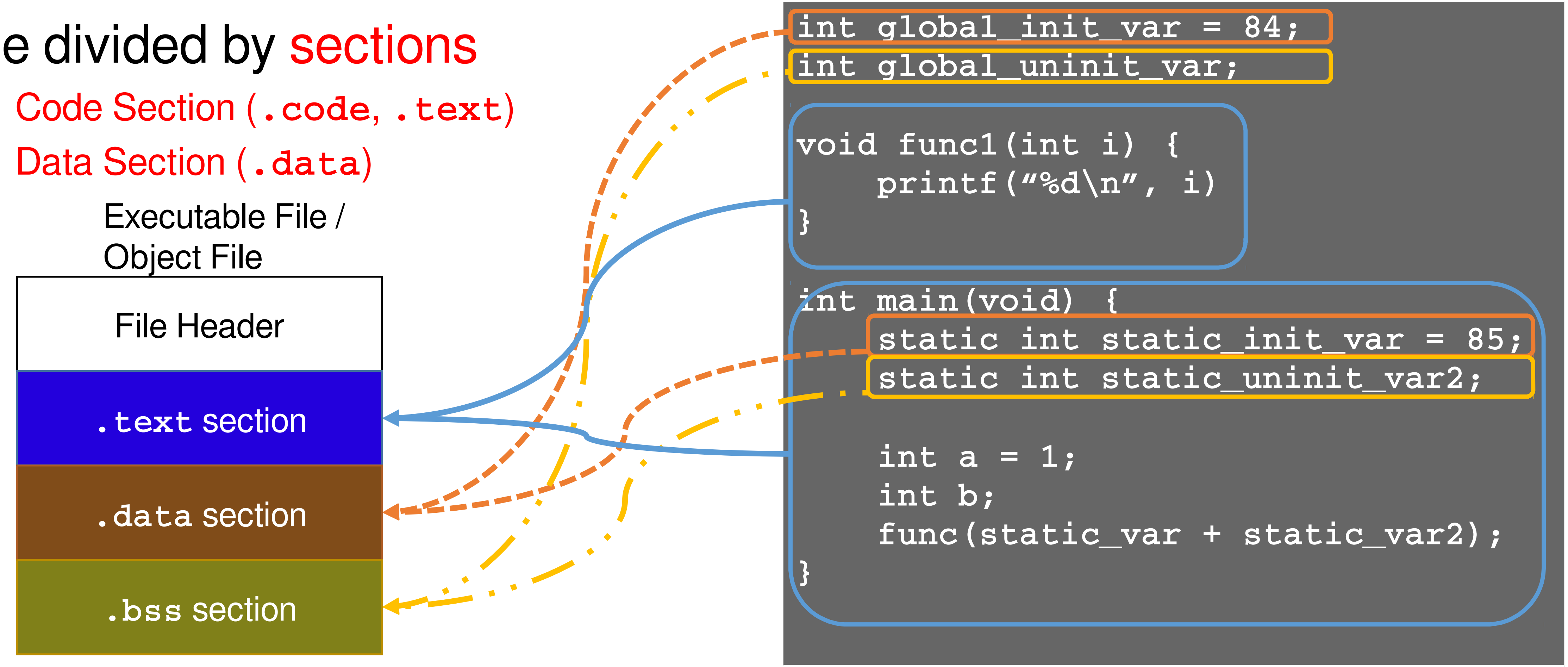
Object File Format

File Format

- Executable file format
  - Derived from COFF(Common Object File Format)
    - Windows : PE (Portable Executable)
    - Linux: ELF (Executable Linkable Format)
  - Dynamic Linking Library (DLL)
    - Windows (.dll); Linux (.so)
  - Static Linking Library
    - Windows (.lib); Linux (.a)
- Intermediate file between compilation and linking → Object file
  - Windows (.obj); Linux (.o)
  - Like executable file format

File Content

- Machine code, data, symbol table, string table
- File divided by sections
  - Code Section (`.code`, `.text`)
  - Data Section (`.data`)

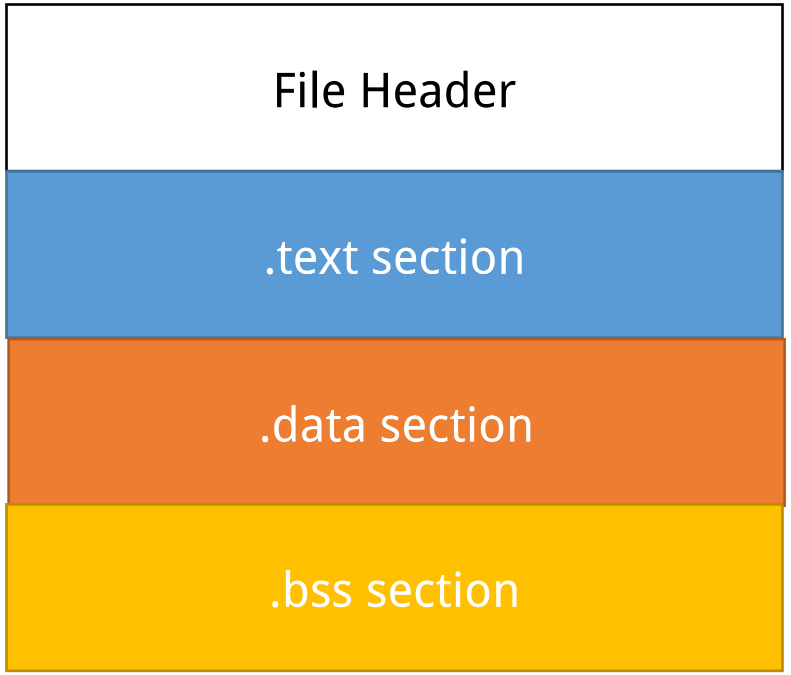


```
int global_init_var = 84;
int global_uninit_var;

void func1(int i) {
    printf(“%d\n”, i)
}

int main(void) {
    static int static_init_var = 85;
    static int static_uninit_var2;

    int a = 1;
    int b;
    func(static_var + static_var2);
}
```

File Content

- File Header
  - Is executable
  - Static Link or Dynamic Link
  - Entry address
  - Target hardware / OS
  - Section Table
- Code & Data
  - Security
  - Cache
  - Share code section(multiple process)

Executable File /
Object File



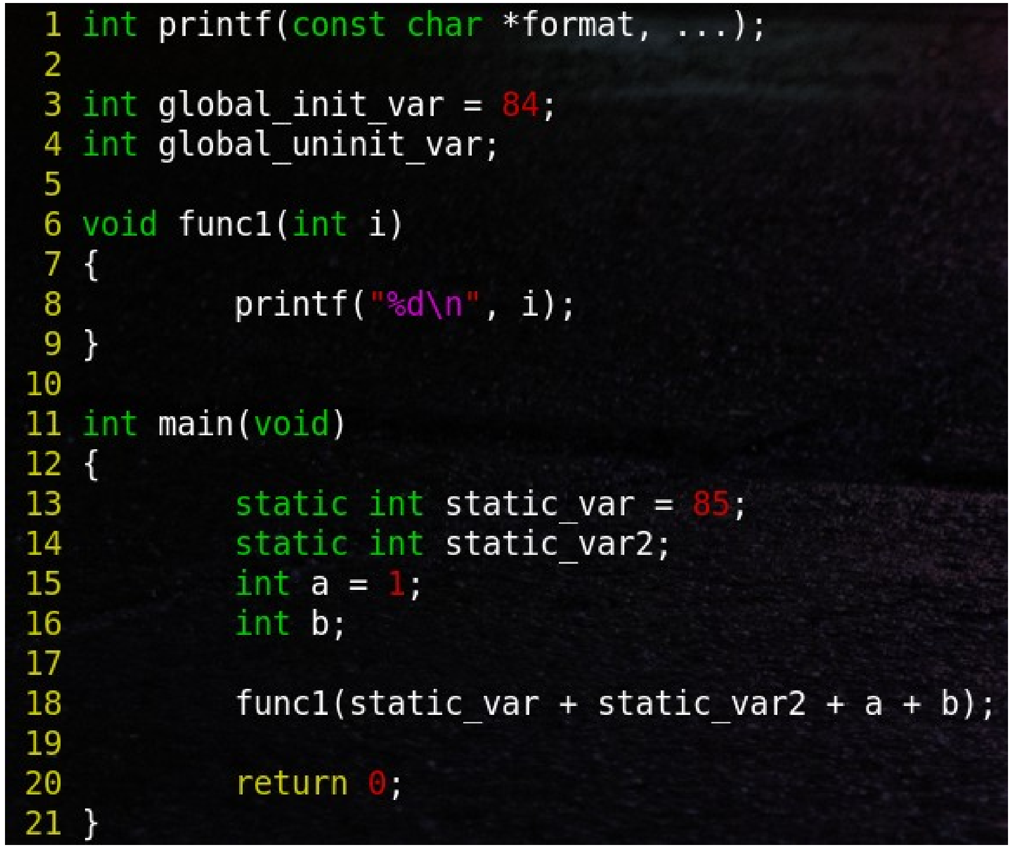
File Header

.text section

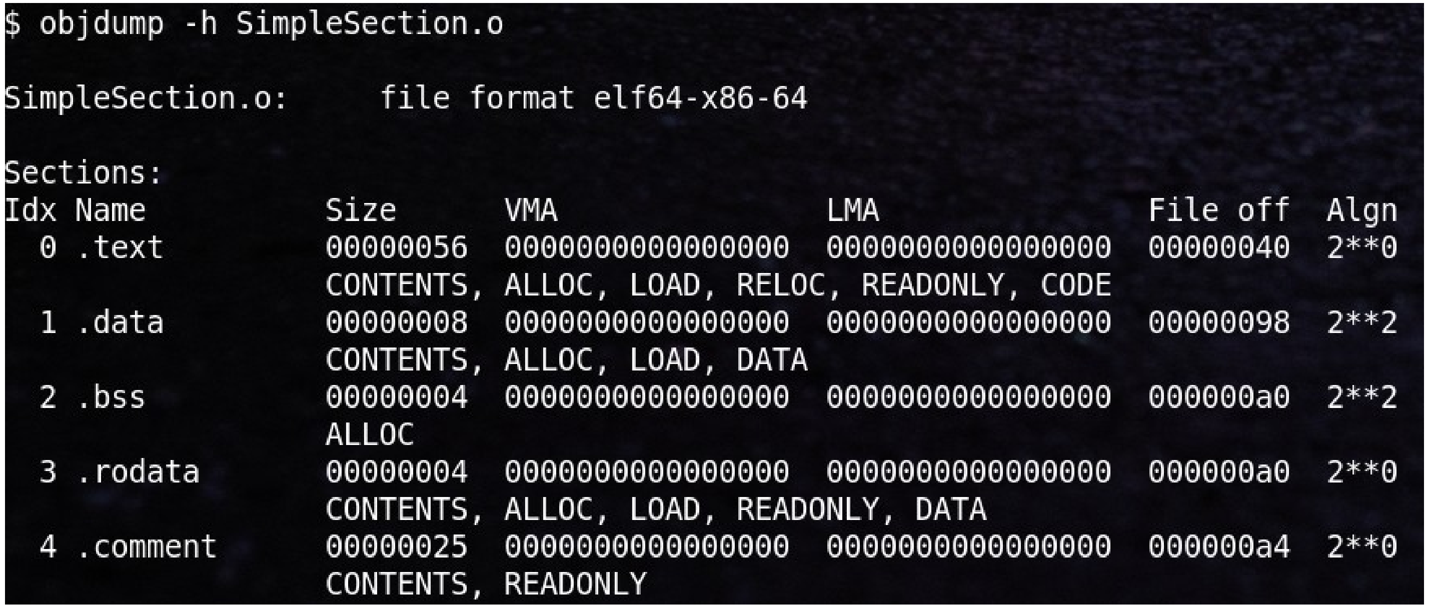
.data section

.bss section

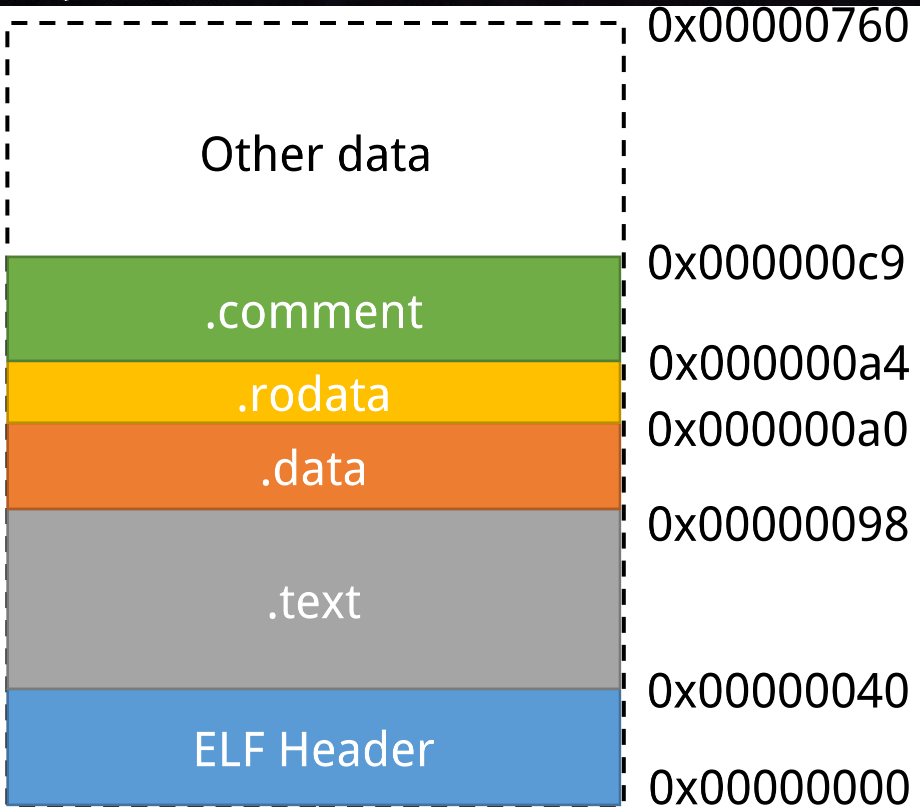
Section



```c
int printf(const char *format, ...);

int global_init_var = 84;
int global_uninit_var;

void func1(int i)
{
        printf("%d\n", i);
}

int main(void)
{
        static int static_var = 85;
        static int static_var2;
        int a = 1;
        int b;

        func1(static_var + static_var2 + a + b);

        return 0;
}
```

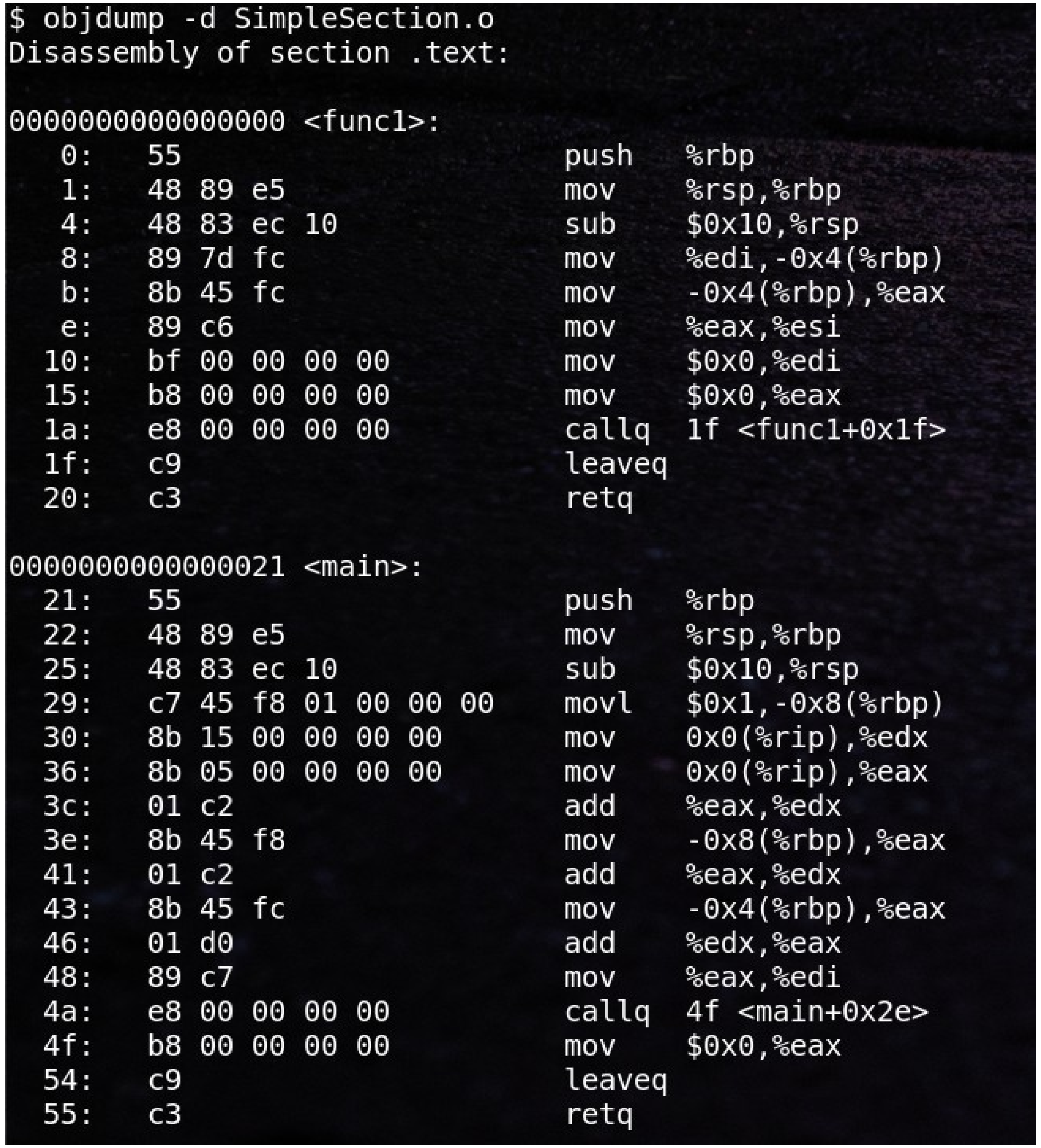


```
$ objdump -h SimpleSection.o

SimpleSection.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000056  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000008  0000000000000000  0000000000000000  00000098  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  0000000000000000  0000000000000000  000000a0  2**2
                  ALLOC
  3 .rodata       00000004  0000000000000000  0000000000000000  000000a0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000025  0000000000000000  0000000000000000  000000a4  2**0
                  CONTENTS, READONLY
```

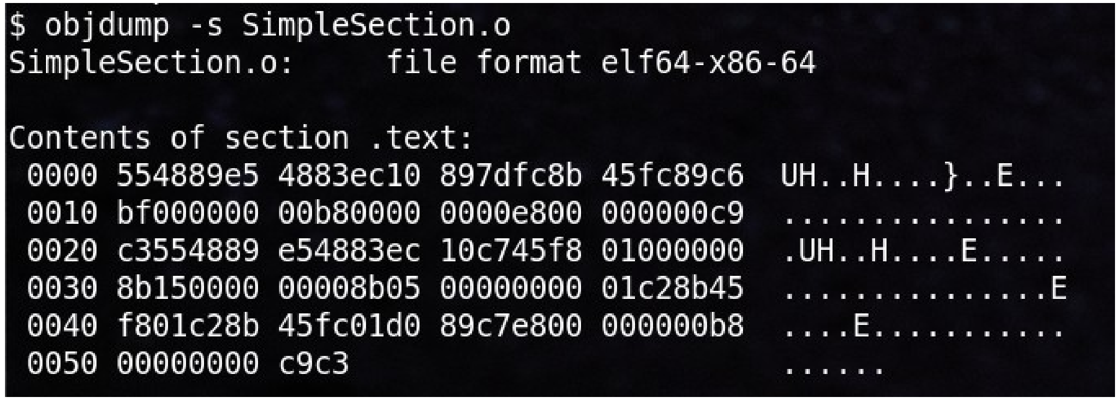


Code Section

- objdump -s
  - Display the full contents of all sections
- objdump -d
  - Display assembler contents of executable sections



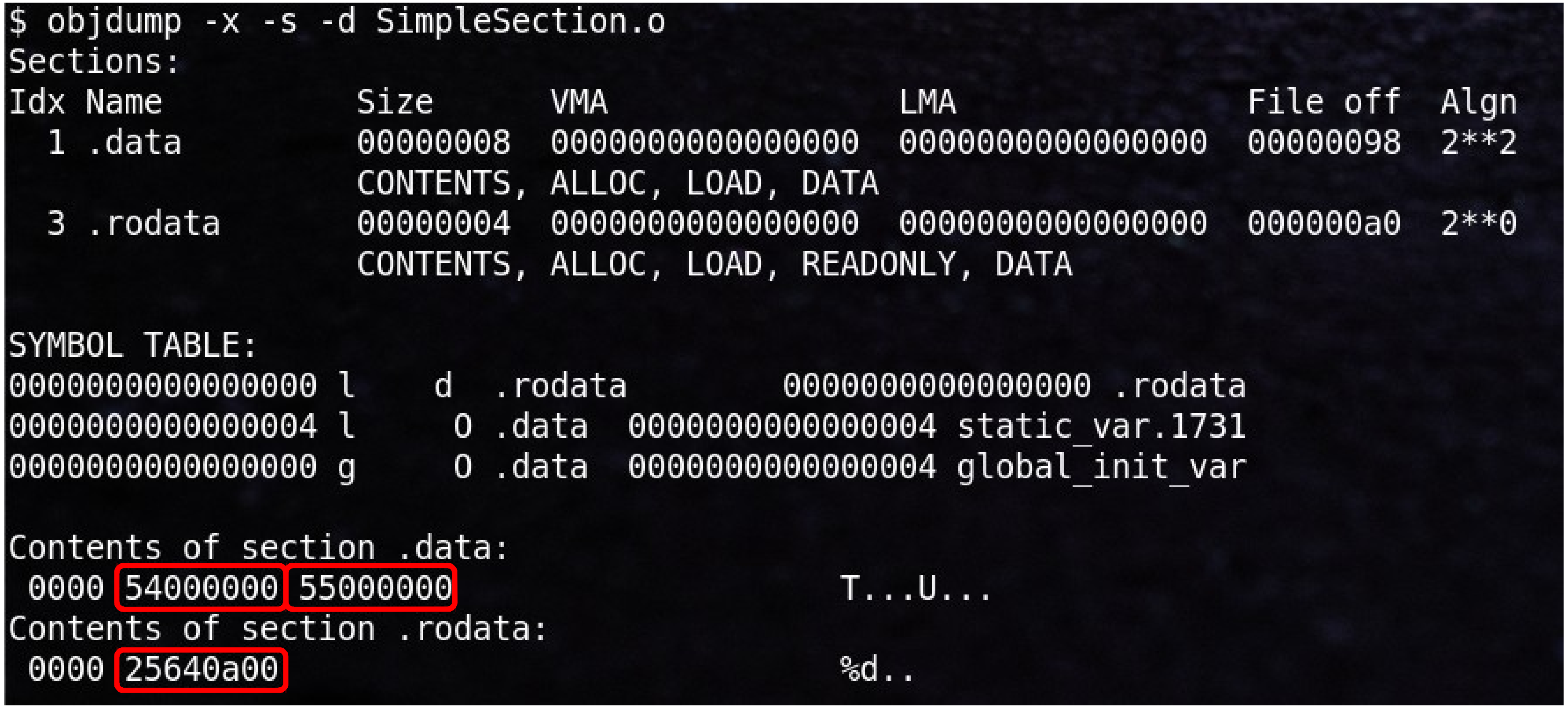
```
$ objdump -d SimpleSection.o
Disassembly of section .text:

0000000000000000 <func1>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   8b 45 fc                mov    -0x4(%rbp),%eax
   e:   89 c6                   mov    %eax,%esi
  10:   bf 00 00 00 00          mov    $0x0,%edi
  15:   b8 00 00 00 00          mov    $0x0,%eax
  1a:   e8 00 00 00 00          callq  1f <func1+0x1f>
  1f:   c9                      leaveq
  20:   c3                      retq

0000000000000021 <main>:
  21:   55                      push   %rbp
  22:   48 89 e5                mov    %rsp,%rbp
  25:   48 83 ec 10             sub    $0x10,%rsp
  29:   c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
  30:   8b 15 00 00 00 00       mov    0x0(%rip),%edx
  36:   8b 05 00 00 00 00       mov    0x0(%rip),%eax
  3c:   01 c2                   add    %eax,%edx
  3e:   8b 45 f8                mov    -0x8(%rbp),%eax
  41:   01 c2                   add    %eax,%edx
  43:   8b 45 fc                mov    -0x4(%rbp),%eax
  46:   01 d0                   add    %edx,%eax
  48:   89 c7                   mov    %eax,%edi
  4a:   e8 00 00 00 00          callq  4f <main+0x2e>
  4f:   b8 00 00 00 00          mov    $0x0,%eax
  54:   c9                      leaveq
  55:   c3                      retq
```



```
$ objdump -s SimpleSection.o
SimpleSection.o:     file format elf64-x86-64

Contents of section .text:
 0000 554889e5 4883ec10 897dfc8b 45fc89c6  UH..H....}..E...
 0010 bf000000 00b80000 0000e800 000000c9  ................
 0020 c3554889 e54883ec 10c745f8 01000000  .UH..H....E.....
 0030 8b150000 00008b05 00000000 01c28b45  ...............E
 0040 f801c28b 45fc01d0 89c7e800 000000b8  ....E...........
 0050 00000000 c9c3                        ......
```

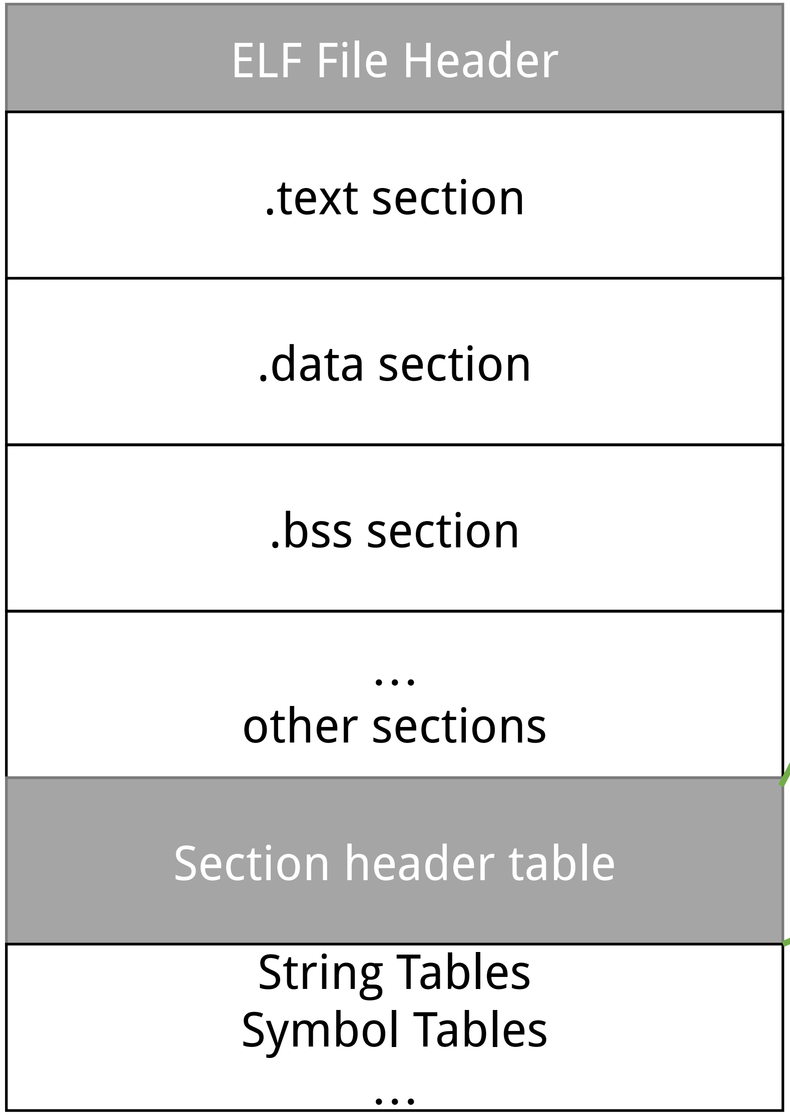
Data Section

- .data → Initialized global variable & static variable
  - global_init_var = 0x54(84)
  - static_var = 0x55(85)



```
$ objdump -x -s -d SimpleSection.o
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  1 .data         00000008  0000000000000000  0000000000000000  00000098  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .rodata       00000004  0000000000000000  0000000000000000  000000a0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA

SYMBOL TABLE:
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000004 l     O .data  0000000000000004 static_var.1731
0000000000000000 g     O .data  0000000000000004 global_init_var

Contents of section .data:
 0000 54000000 55000000                    T...U...
Contents of section .rodata:
 0000 25640a00                             %d..
```

ELF File Structure



ELF File Header

.text section

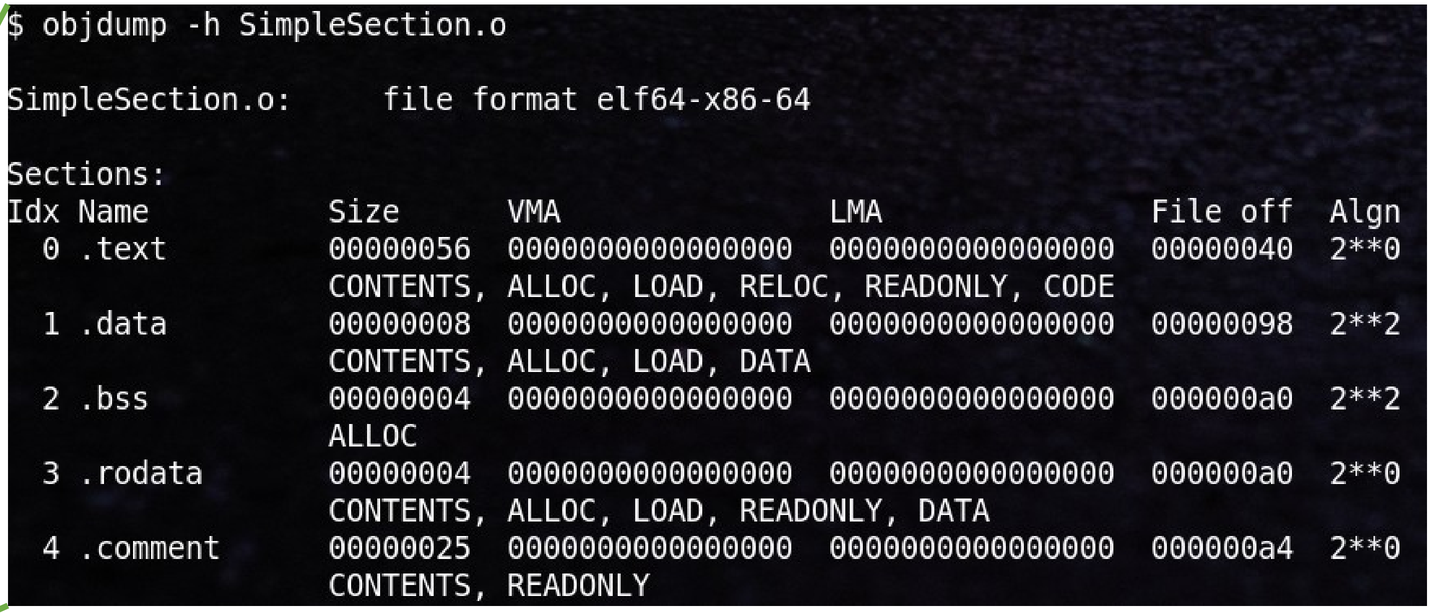
.data section

.bss section

…
other sections

Section header table

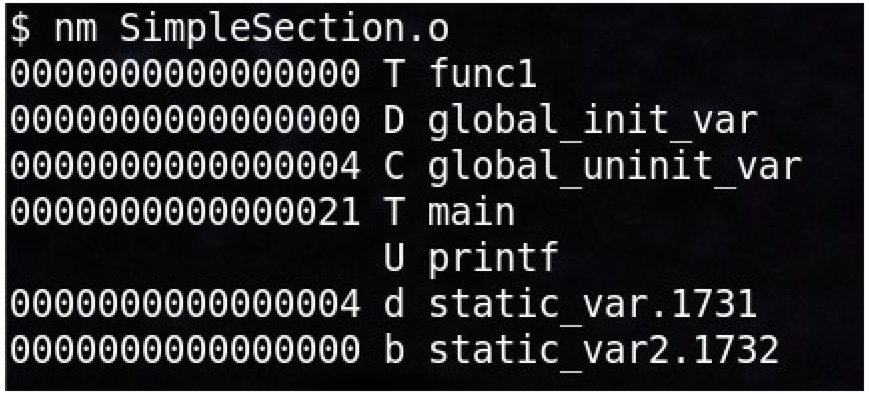
String Tables
Symbol Tables
…



```
$ objdump -h SimpleSection.o

SimpleSection.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000056  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000008  0000000000000000  0000000000000000  00000098  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  0000000000000000  0000000000000000  000000a0  2**2
                  ALLOC
  3 .rodata       00000004  0000000000000000  0000000000000000  000000a0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000025  0000000000000000  0000000000000000  000000a4  2**0
                  CONTENTS, READONLY
```

Symbol

- Object file B use function(variable) “foo” in object file A
  - A defined function(variable) “foo”
  - B reference function(variable) “foo”
- Symbol name(function name, variable name)
- Every object file has a symbol table which record symbol value
- Symbol type
  - Symbol defined in current object file
  - External Symbol
  - …

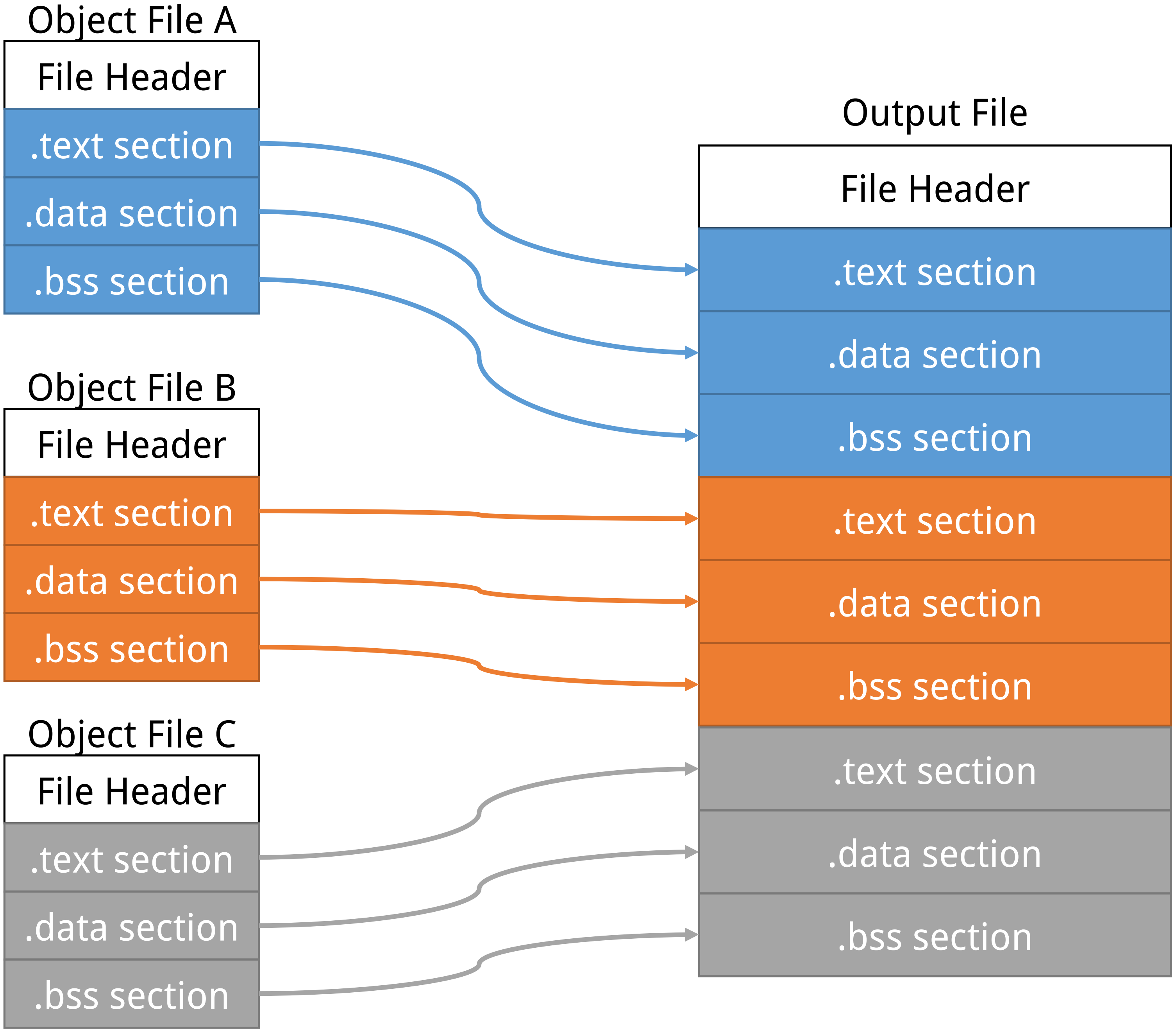


```
$ nm SimpleSection.o
0000000000000000 T func1
0000000000000000 D global_init_var
0000000000000004 C global_uninit_var
0000000000000021 T main
                 U printf
0000000000000004 d static_var.1731
0000000000000000 b static_var2.1732
```

Static Linking

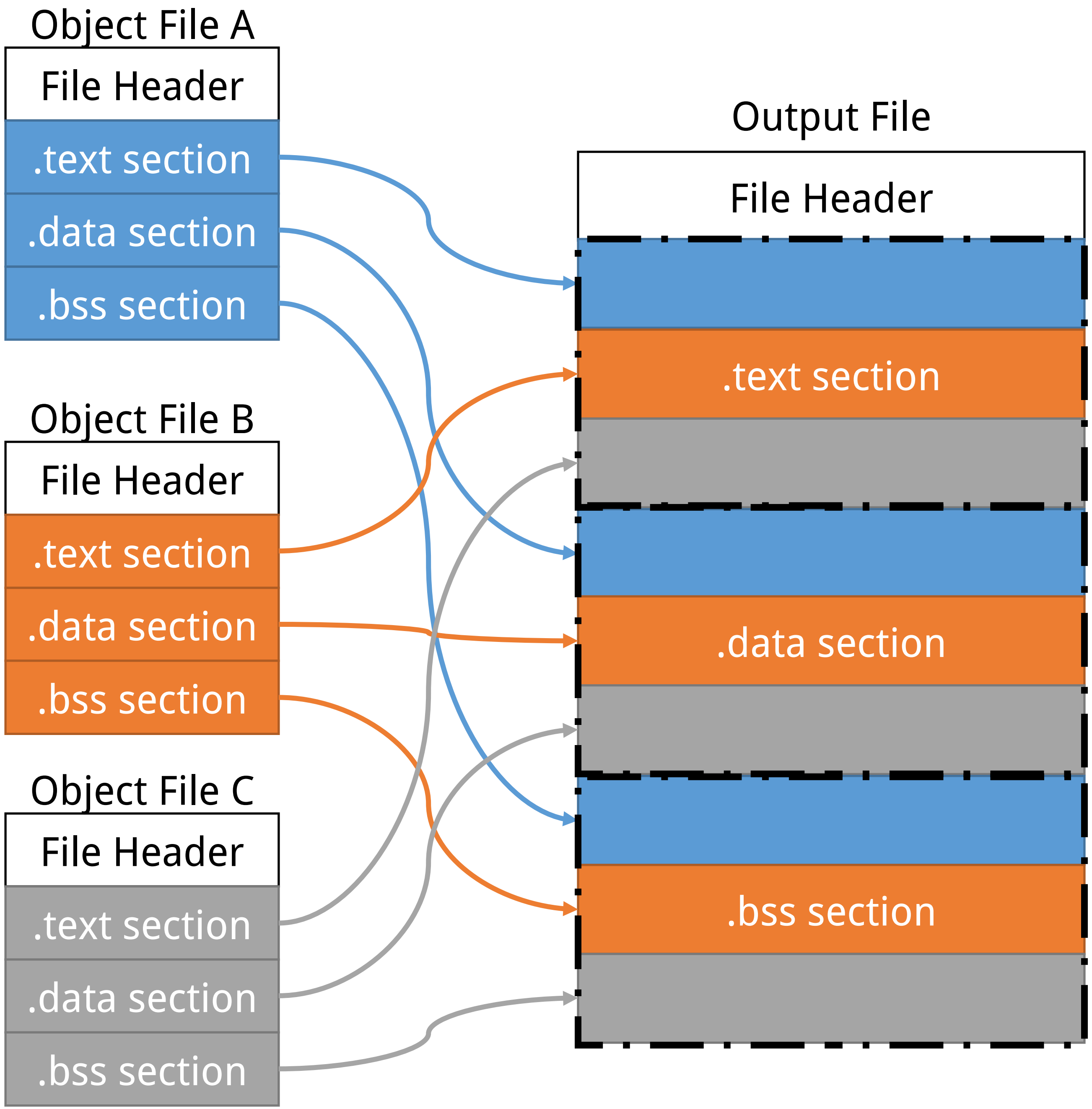
Accumulation

- Put all together
  - Very Simple
- Alignment unit → page(x86)
  - Waste space



Merge Similar Section

- Two-pass Linking
  1. Space & Address Allocation
     - □Fetch section length, attribute an d position
     - □Collect symbol(define, reference) and put to a global table
  2. Symbol Resolution & Relocati on
     - □Modify relocation entry



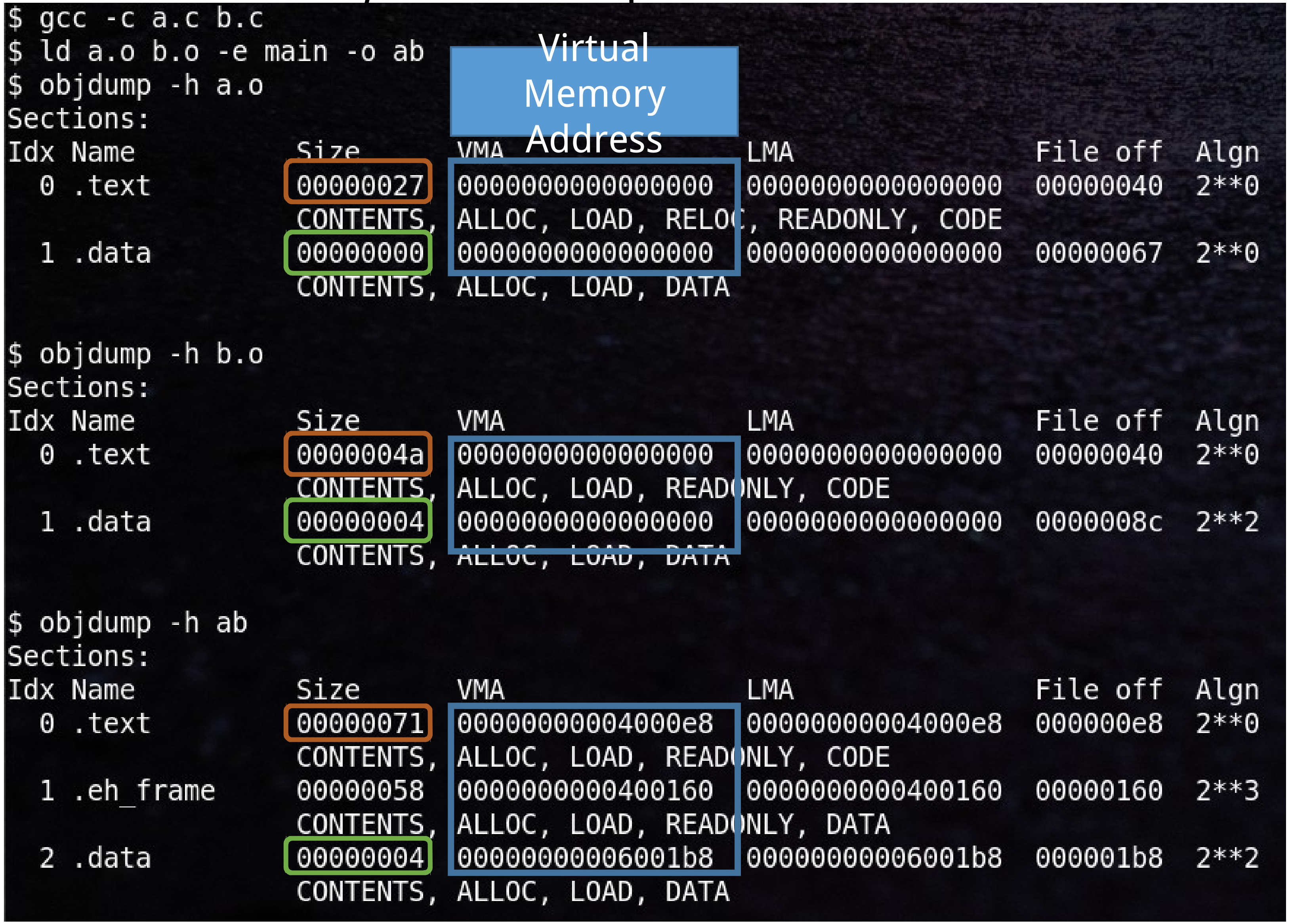
Static Linking Example

Filename: a.c

```
extern int shared;

int main() {
    int a = 100;
    swap(&a, &shared);
}
```

Filename: b.c

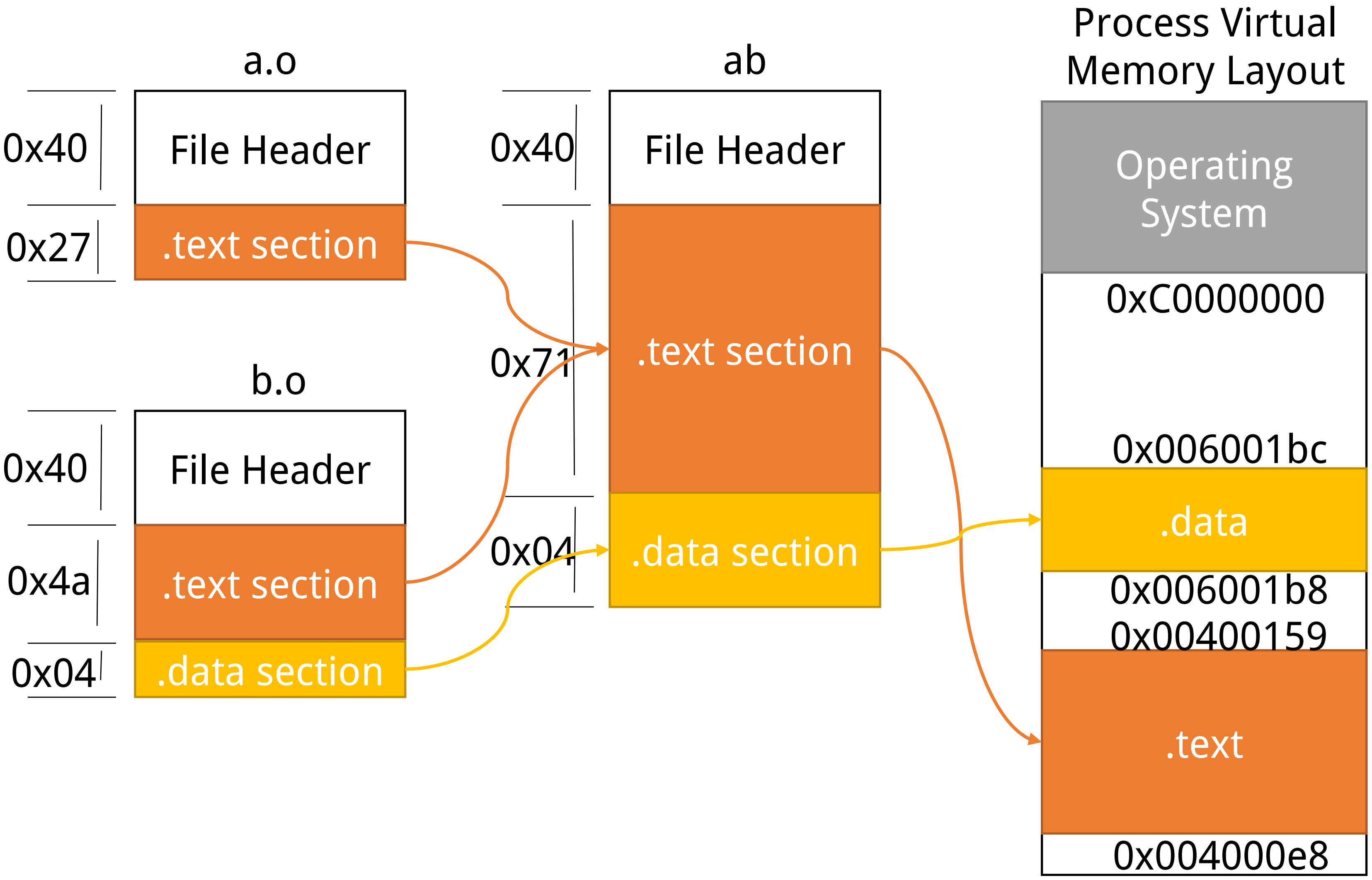
```
int shared = 1;

void swap(int *a, int *b) {
    *a ^= *b ^= *a ^= *b;
}
```



```
$ gcc -c a.c b.c
$ ld a.o b.o -e main -o ab
$ objdump -h a.o
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000027  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000067  2**0
                  CONTENTS, ALLOC, LOAD, DATA

$ objdump -h b.o
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000004a  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  0000000000000000  0000000000000000  0000008c  2**2
                  CONTENTS, ALLOC, LOAD, DATA

$ objdump -h ab
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000071  00000000004000e8  00000000004000e8  000000e8  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .eh_frame     00000058  0000000000400160  0000000000400160  00000160  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000004  00000000006001b8  00000000006001b8  000001b8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Static Linking Example

File	Section	Size	VMA
a.o	.text	0x27	0x00000000
	.data	0x00	0x00000000
b.o	.text	0x4a	0x00000000
	.data	0x04	0x00000000
ab	.text	0x71	0x004000e8
	.data	0x04	0x006001b8

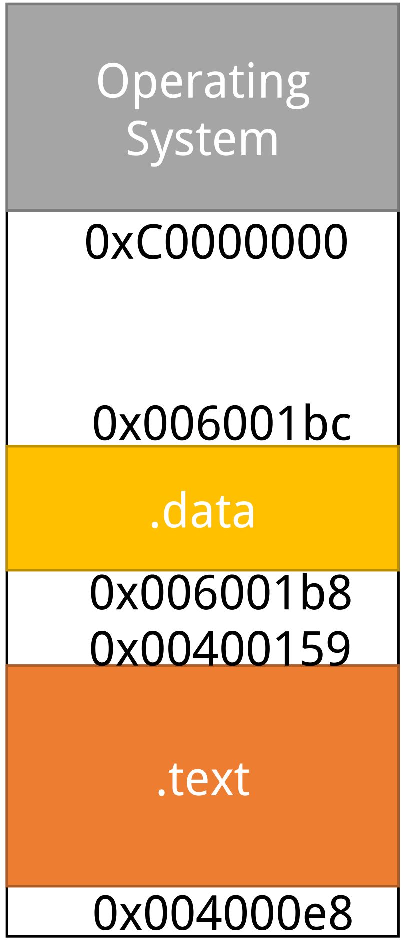


Symbol Address

- Calculation of symbol address
  - function in text section has offset X
  - text section in executable file has offset Y
  - → function in executable file has offset X + Y
- Example:
  - “swap” in “b.o.text” has offset 0x00000000
  - “b.o.text” in “ab” has offset 0x0040010f
  - → “swap” in “ab” has offset 0x00000000 + 0x0040010f = 0x0040010f

Symbol	Type	Virtual Address
main	function	0x004000e8
swap	function	0x0040010f
shared	variable	0x006001b8

Process Virtual Memory Layout



Operating System

0xC0000000

0x006001bc

.data

0x006001b8

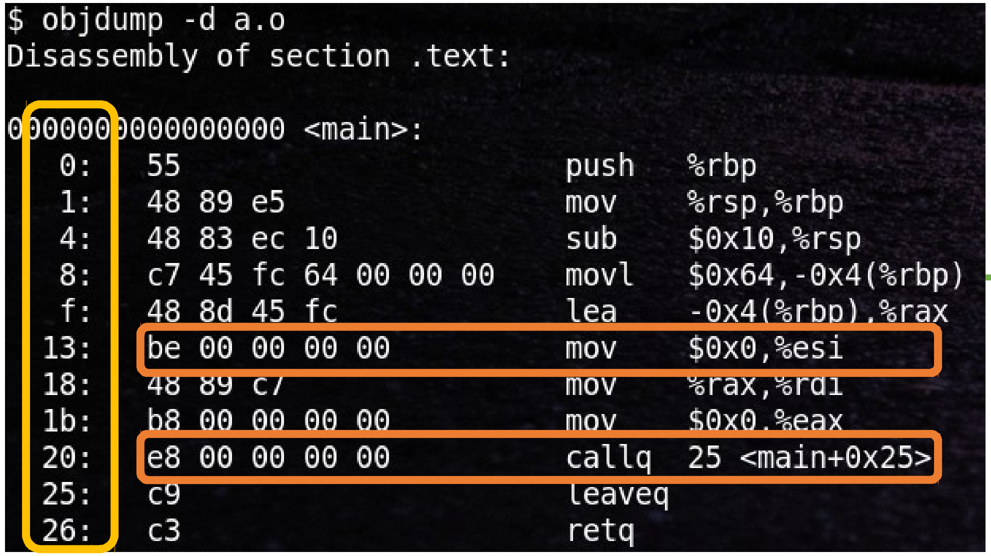
0x00400159

.text

0x004000e8

Relocation

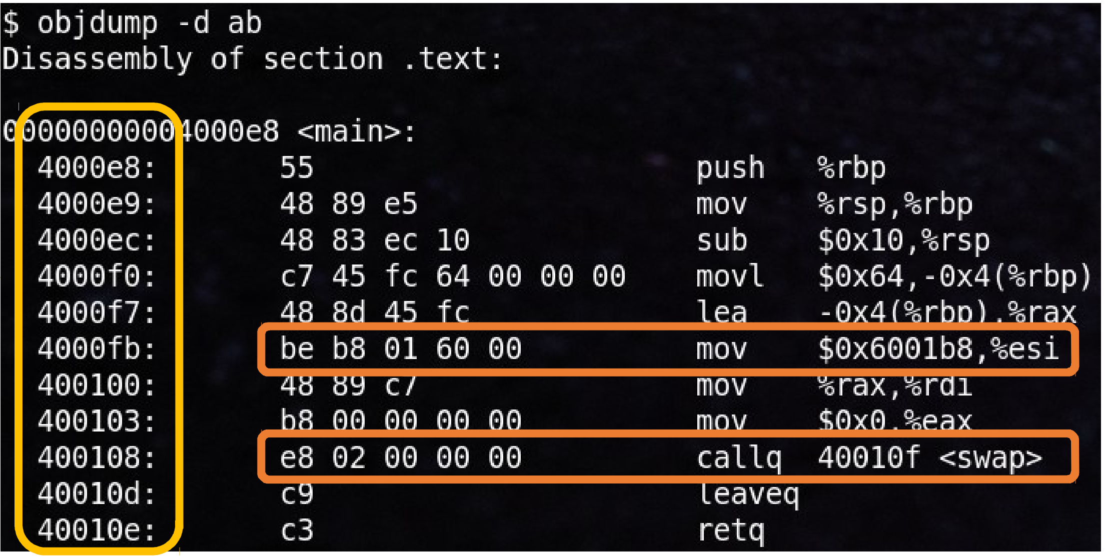
a.o



```
$ objdump -d a.o
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 fc 64 00 00 00    movl   $0x64,-0x4(%rbp)
   f:   48 8d 45 fc             lea    -0x4(%rbp),%rax
  13:   be 00 00 00 00          mov    $0x0,%esi
  18:   48 89 c7                mov    %rax,%rdi
  1b:   b8 00 00 00 00          mov    $0x0,%eax
  20:   e8 00 00 00 00          callq  25 <main+0x25>
  25:   c9                      leaveq
  26:   c3                      retq
```

Linking

ab



```
$ objdump -d ab
Disassembly of section .text:

00000000004000e8 <main>:
  4000e8:       55                      push   %rbp
  4000e9:       48 89 e5                mov    %rsp,%rbp
  4000ec:       48 83 ec 10             sub    $0x10,%rsp
  4000f0:       c7 45 fc 64 00 00 00    movl   $0x64,-0x4(%rbp)
  4000f7:       48 8d 45 fc             lea    -0x4(%rbp),%rax
  4000fb:       be b8 01 60 00          mov    $0x6001b8,%esi
  400100:       48 89 c7                mov    %rax,%rdi
  400103:       b8 00 00 00 00          mov    $0x0,%eax
  400108:       e8 02 00 00 00          callq  40010f <swap>
  40010d:       c9                      leaveq
  40010e:       c3                      retq
```

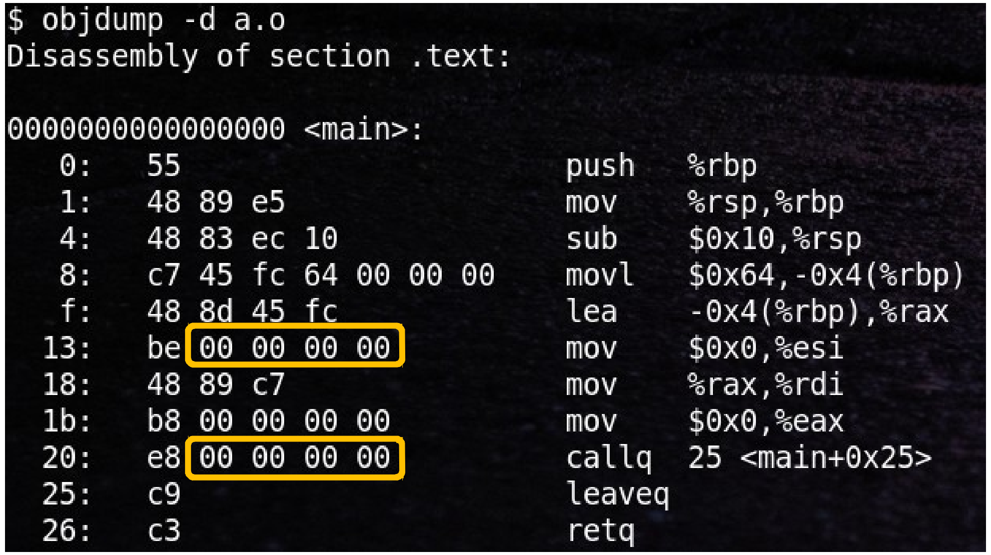
Filename: a.c

```
extern int shared;

int main() {
    int a = 100;
    swap(&a, &shared);
}
```

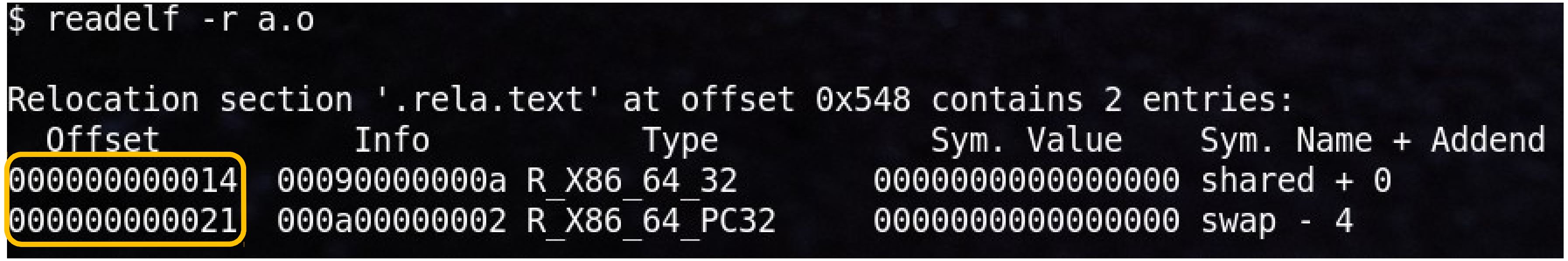
Symbol	Type	Virtual Address
main	function	0x004000e8
swap	function	0x0040010f
shared	variable	0x006001b8

Relocation Table

- Relocatable ELF section will have a .rel section
  - .rel.text
  - .rel.data

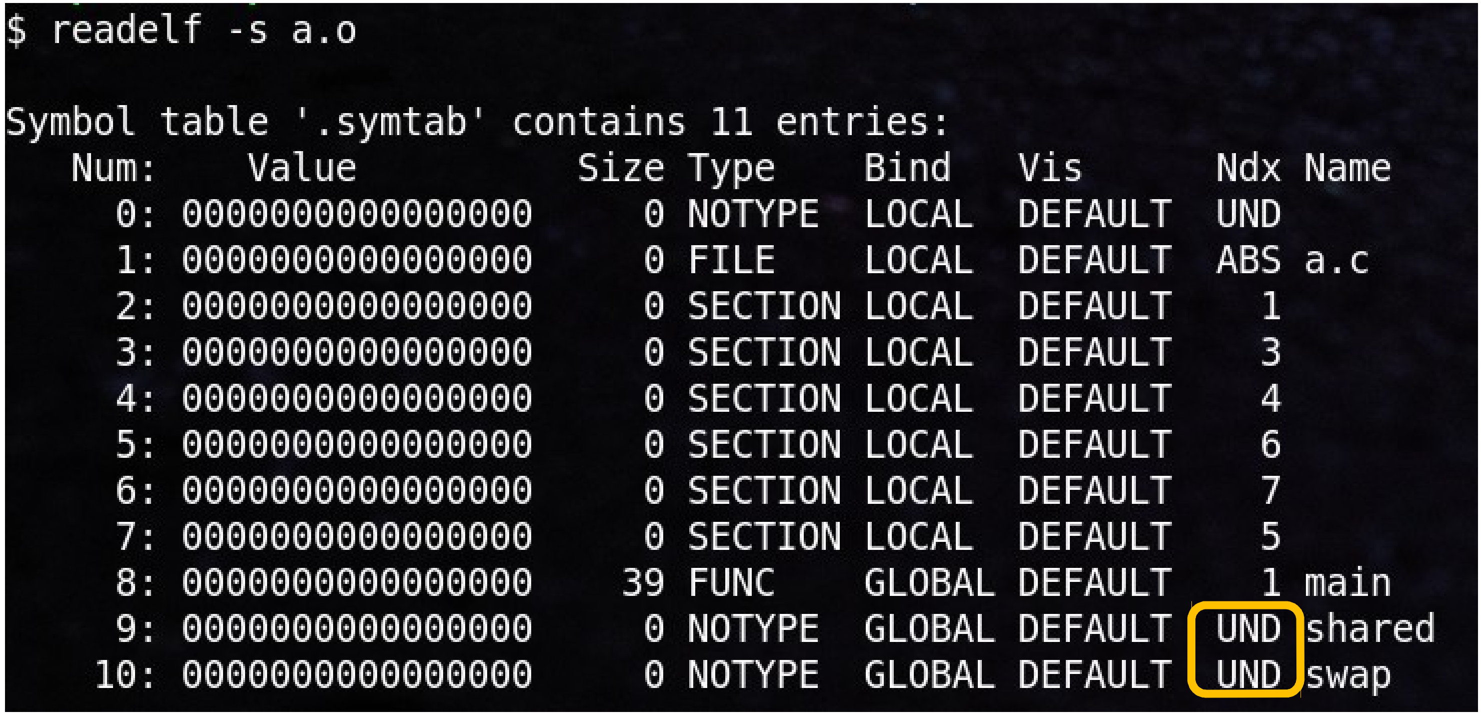


```
$ objdump -d a.o
Disassembly of section .text:

0000000000000000 <main>:
   0:	55                   	push   %rbp
   1:	48 89 e5             	mov    %rsp,%rbp
   4:	48 83 ec 10          	sub    $0x10,%rsp
   8:	c7 45 fc 64 00 00 00 	movl   $0x64,-0x4(%rbp)
   f:	48 8d 45 fc          	lea    -0x4(%rbp),%rax
  13:	be 00 00 00 00       	mov    $0x0,%esi
  18:	48 89 c7             	mov    %rax,%rdi
  1b:	b8 00 00 00 00       	mov    $0x0,%eax
  20:	e8 00 00 00 00       	callq  25 <main+0x25>
  25:	c9                   	leaveq
  26:	c3                   	retq
```



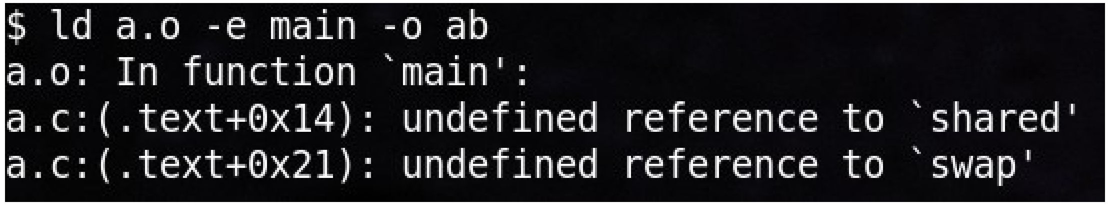
```
$ readelf -r a.o

Relocation section '.rela.text' at offset 0x548 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000014  00090000000a R_X86_64_32       0000000000000000 shared + 0
000000000021  000a00000002 R_X86_64_PC32     0000000000000000 swap - 4
```

Symbol Resolution



```
$ readelf -s a.o

Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS a.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     8: 0000000000000000    39 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND shared
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND swap
```

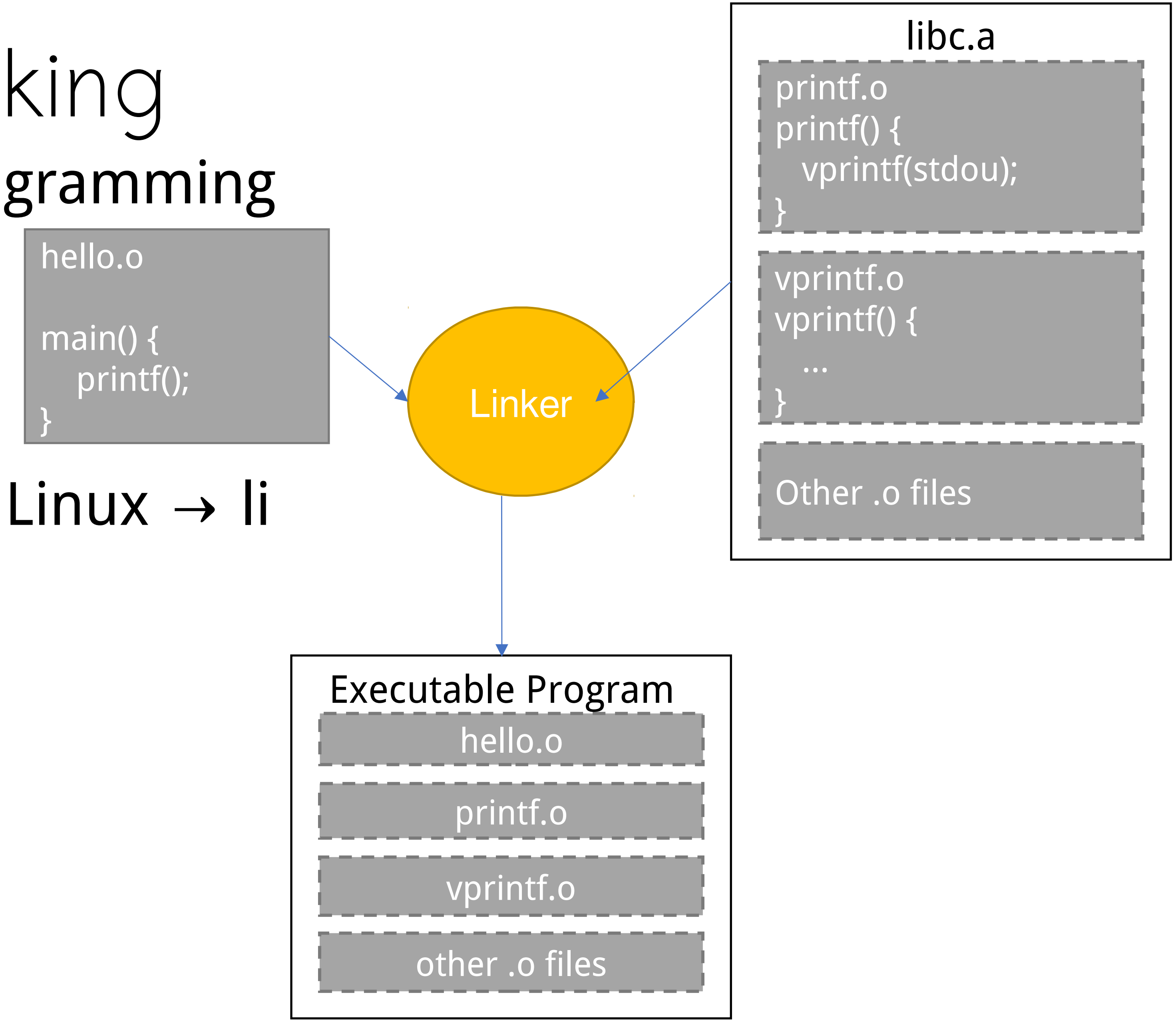
- What will happen if we do not link “b.o”?



```
$ ld a.o -e main -o ab
a.o: In function `main':
a.c:(.text+0x14): undefined reference to `shared'
a.c:(.text+0x21): undefined reference to `swap'
```

Static Library Linking

- OS provide Application Programming Interface(API)
- Language Library
- Collection of object files
- C language static library in Linux → libc.a



```
hello.o

main() {
    printf();
}
```

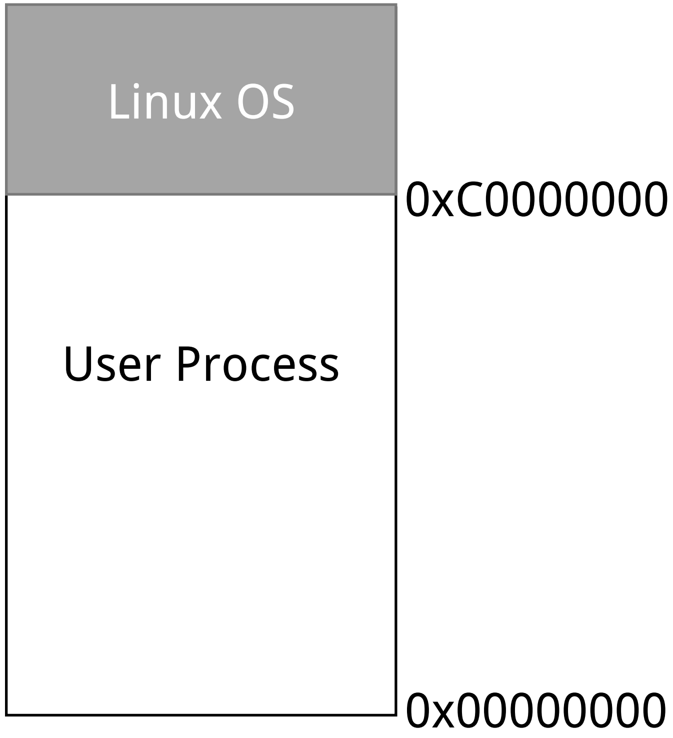
```
printf.o
printf() {
   vprintf(stdou);
}
```

```
vprintf.o
vprintf() {
   ...
}
```

Executable File Loading & Process

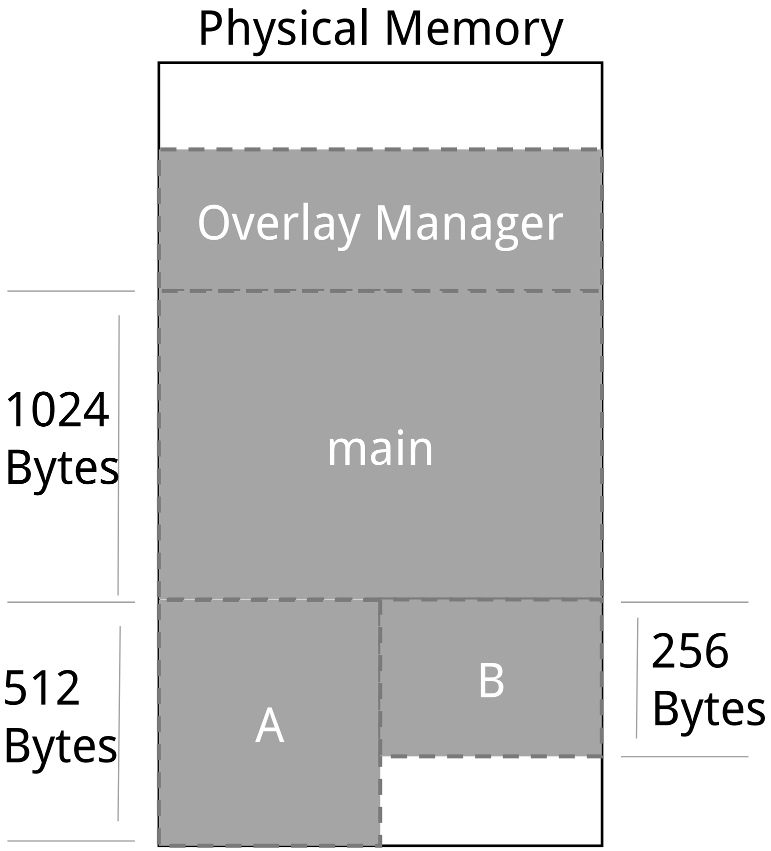
Program & Process

- Analogy
  - Program ↔ Recipe
  - CPU ↔ Man
  - Hardware ↔ Kitchenware
  - Process ↔ Cooking
  - Two CPU can execute the same program
- Process own independent Virtual Address Space
- Process access not allowed address → “Segmentation fault”



Loading

- Overlay
  - ➢Programmer divided progra m
  - ➢Implement Overlay Manager
  - ➢Ex.
    - □Three modules: main, A, B
    - □main → 1024 bytes
    - □A → 512 Bytes
    - □B → 256 Bytes
    - □Total → 1792 Bytes
    - □A will not call B
- Paging

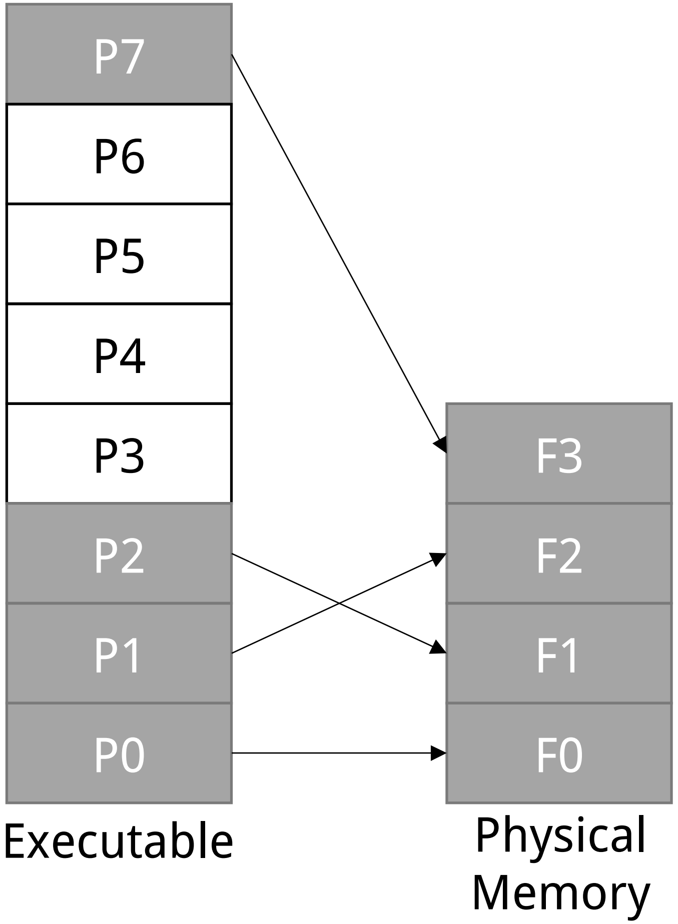


Paging

- Loading & Operation Unit → page
- Example:.
  - 32-bit machine with 16 KB memory
  - page size = 4096 bytes → 4 pages

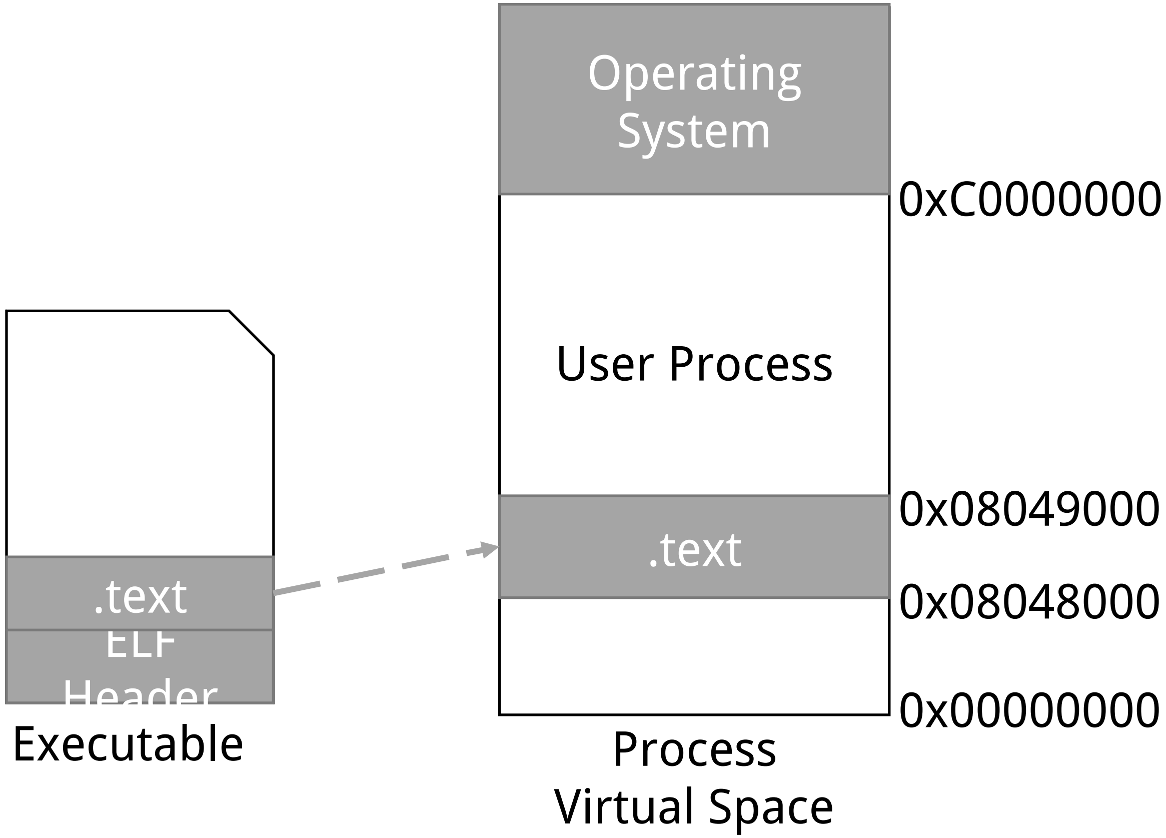
Page Index	Address
F0	0x00000000-0x00000FFF
F1	0x00001000-0x00001FFF
F2	0x00002000-0x00002FFF
F3	0x00003000-0x00003FFF

  - program size = 32 KB → 8 pages
- Page replace
  - FIFO
  - LRU(Least Recently Used)



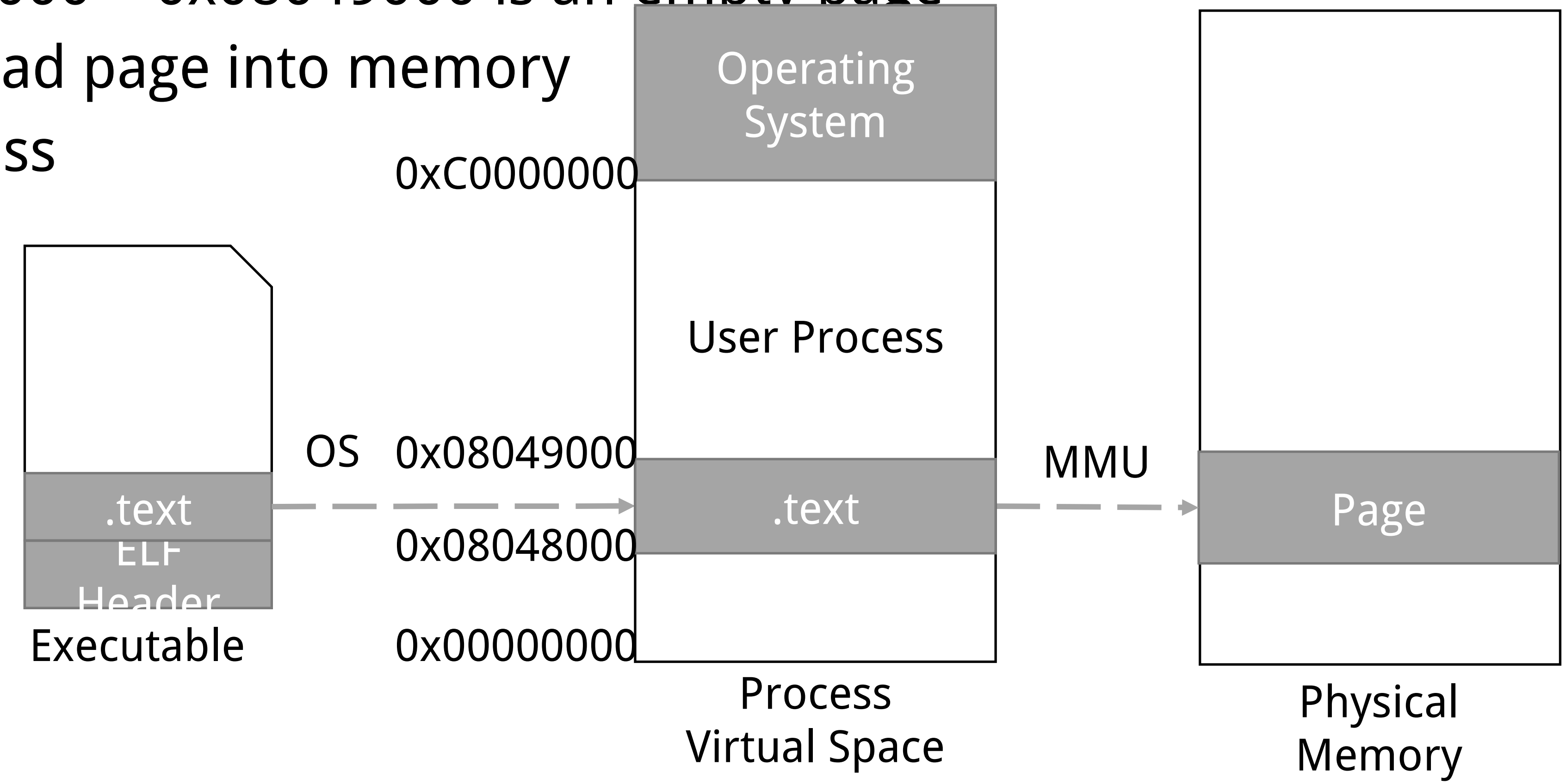
Creation of Process

1. Create a independent virtual AS
   - page directory(Linux)
2. Read executable file header, create mapping between virtual AS and executable file
   - VMA, Virtual Memory Area
3. Assign entry address to program register(PC)
   - Switch between kernel stack and process stack
   - CPU access attribute



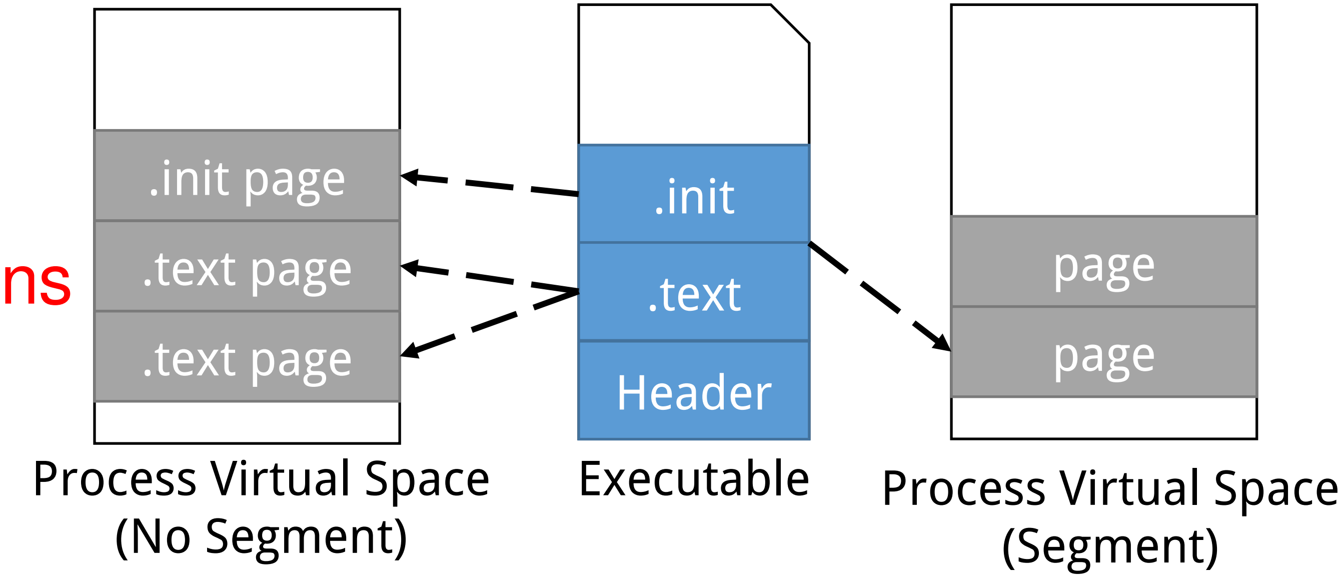
Page Fault

- Executable file has not been loaded into physical memory yet
- Page fault
  1. Found 0x08048000 ~ 0x08049000 is an empty page
  2. Page handler load page into memory
  3. Return to process

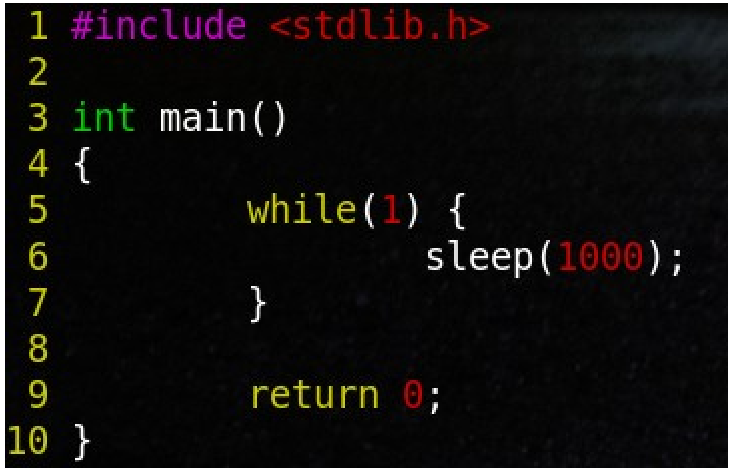


Segment

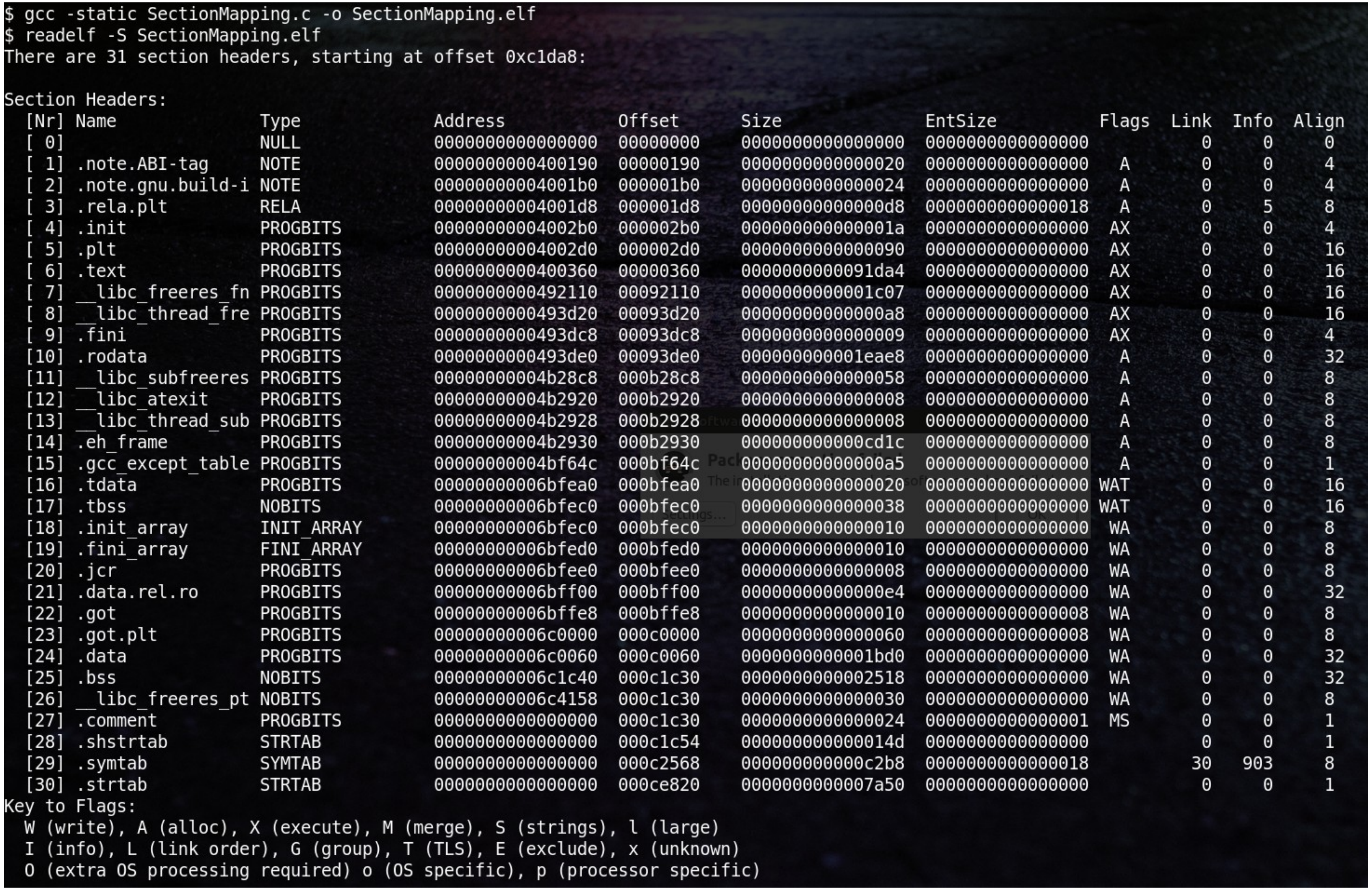
- Page alignment
  - More than a dozen sections
  - Waste space
- OS only cares access rights of sections
  - Readable & Executable(code)
  - Readable & Writable(data)
  - Read Only(rodata)
- Merge the same access rights of sections
  - .text section is 4097 bytes
  - .init section is 512 bytes



Segment Example

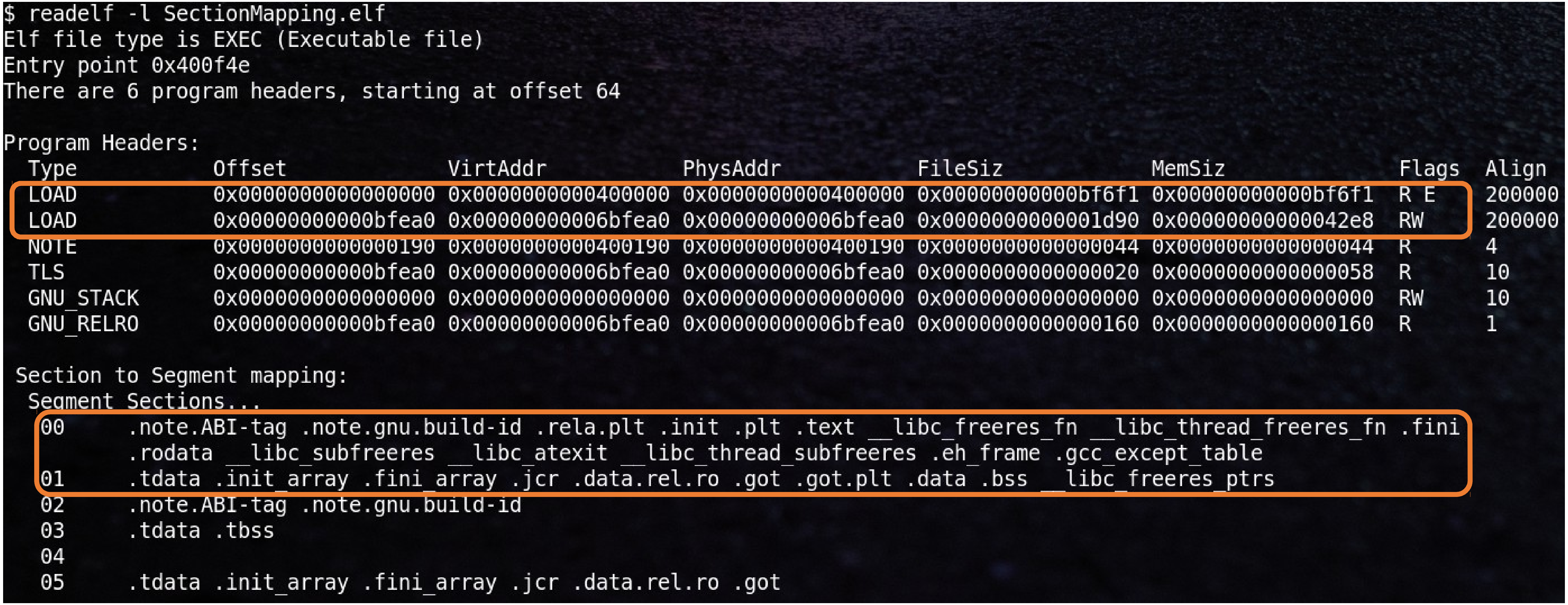


```c
#include <stdlib.h>

int main()
{
        while(1) {
                sleep(1000);
        }

        return 0;
}
```



```
$ gcc -static SectionMapping.c -o SectionMapping.elf
$ readelf -S SectionMapping.elf
There are 31 section headers, starting at offset 0xc1da8:

Section Headers:
  [Nr] Name              Type             Address           Offset      Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000    0000000000000000  0000000000000000           0     0     0
  [ 1] .note.ABI-tag     NOTE             0000000000400190  00000190    0000000000000020  0000000000000000   A       0     0     4
  [ 2] .note.gnu.build-i NOTE             00000000004001b0  000001b0    0000000000000024  0000000000000000   A       0     0     4
  [ 3] .rela.plt         RELA             00000000004001d8  000001d8    00000000000000d8  0000000000000018   A       0     5     8
  [ 4] .init             PROGBITS         00000000004002b0  000002b0    000000000000001a  0000000000000000  AX       0     0     4
  [ 5] .plt              PROGBITS         00000000004002d0  000002d0    0000000000000090  0000000000000000  AX       0     0     16
  [ 6] .text             PROGBITS         0000000000400360  00000360    0000000000091da4  0000000000000000  AX       0     0     16
  [ 7] __libc_freeres_fn PROGBITS         0000000000492110  00092110    0000000000001c07  0000000000000000  AX       0     0     16
  [ 8] __libc_thread_fre PROGBITS         0000000000493d20  00093d20    00000000000000a8  0000000000000000  AX       0     0     16
  [ 9] .fini             PROGBITS         0000000000493dc8  00093dc8    0000000000000009  0000000000000000  AX       0     0     4
  [10] .rodata           PROGBITS         0000000000493de0  00093de0    000000000001eae8  0000000000000000   A       0     0     32
  [11] __libc_subfreeres PROGBITS         00000000004b28c8  000b28c8    0000000000000058  0000000000000000   A       0     0     8
  [12] __libc_atexit     PROGBITS         00000000004b2920  000b2920    0000000000000008  0000000000000000   A       0     0     8
  [13] __libc_thread_sub PROGBITS         00000000004b2928  000b2928    0000000000000008  0000000000000000   A       0     0     8
  [14] .eh_frame         PROGBITS         00000000004b2930  000b2930    000000000000cd1c  0000000000000000   A       0     0     8
  [15] .gcc_except_table PROGBITS         00000000004bf64c  000bf64c    00000000000000a5  0000000000000000   A       0     0     1
  [16] .tdata            PROGBITS         00000000006bfea0  000bfea0    0000000000000020  0000000000000000 WAT       0     0     16
  [17] .tbss             NOBITS           00000000006bfec0  000bfec0    0000000000000038  0000000000000000 WAT       0     0     16
  [18] .init_array       INIT_ARRAY       00000000006bfec0  000bfec0    0000000000000010  0000000000000000  WA       0     0     8
  [19] .fini_array       FINI_ARRAY       00000000006bfed0  000bfed0    0000000000000010  0000000000000000  WA       0     0     8
  [20] .jcr              PROGBITS         00000000006bfee0  000bfee0    0000000000000008  0000000000000000  WA       0     0     8
  [21] .data.rel.ro      PROGBITS         00000000006bff00  000bff00    00000000000000e4  0000000000000000  WA       0     0     32
  [22] .got              PROGBITS         00000000006bffe8  000bffe8    0000000000000010  0000000000000008  WA       0     0     8
  [23] .got.plt          PROGBITS         00000000006c0000  000c0000    0000000000000060  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS         00000000006c0060  000c0060    0000000000001bd0  0000000000000000  WA       0     0     32
  [25] .bss              NOBITS           00000000006c1c40  000c1c30    0000000000002518  0000000000000000  WA       0     0     32
  [26] __libc_freeres_pt NOBITS           00000000006c4158  000c1c30    0000000000000030  0000000000000000  WA       0     0     8
  [27] .comment          PROGBITS         0000000000000000  000c1c30    0000000000000024  0000000000000001  MS       0     0     1
  [28] .shstrtab         STRTAB           0000000000000000  000c1c54    000000000000014d  0000000000000000           0     0     1
  [29] .symtab           SYMTAB           0000000000000000  000c2568    000000000000c2b8  0000000000000018          30   903     8
  [30] .strtab           STRTAB           0000000000000000  000ce820    0000000000007a50  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Segment Example



```
$ readelf -l SectionMapping.elf
Elf file type is EXEC (Executable file)
Entry point 0x400f4e
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr           FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000 0x00000000000bf6f1 0x00000000000bf6f1  R E    200000
  LOAD           0x00000000000bfea0 0x00000000006bfea0 0x00000000006bfea0 0x0000000000001d90 0x00000000000042e8  RW     200000
  NOTE           0x0000000000000190 0x0000000000400190 0x0000000000400190 0x0000000000000044 0x0000000000000044  R      4
  TLS            0x00000000000bfea0 0x00000000006bfea0 0x00000000006bfea0 0x0000000000000020 0x0000000000000058  R      10
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x00000000000bfea0 0x00000000006bfea0 0x00000000006bfea0 0x0000000000000160 0x0000000000000160  R      1

 Section to Segment mapping:
  Segment Sections...
   00     .note.ABI-tag .note.gnu.build-id .rela.plt .init .plt .text __libc_freeres_fn __libc_thread_freeres_fn .fini
          .rodata __libc_subfreeres __libc_atexit __libc_thread_subfreeres .eh_frame .gcc_except_table
   01     .tdata .init_array .fini_array .jcr .data.rel.ro .got .got.plt .data .bss __libc_freeres_ptrs
   02     .note.ABI-tag .note.gnu.build-id
   03     .tdata .tbss
   04
   05     .tdata .init_array .fini_array .jcr .data.rel.ro .got
```

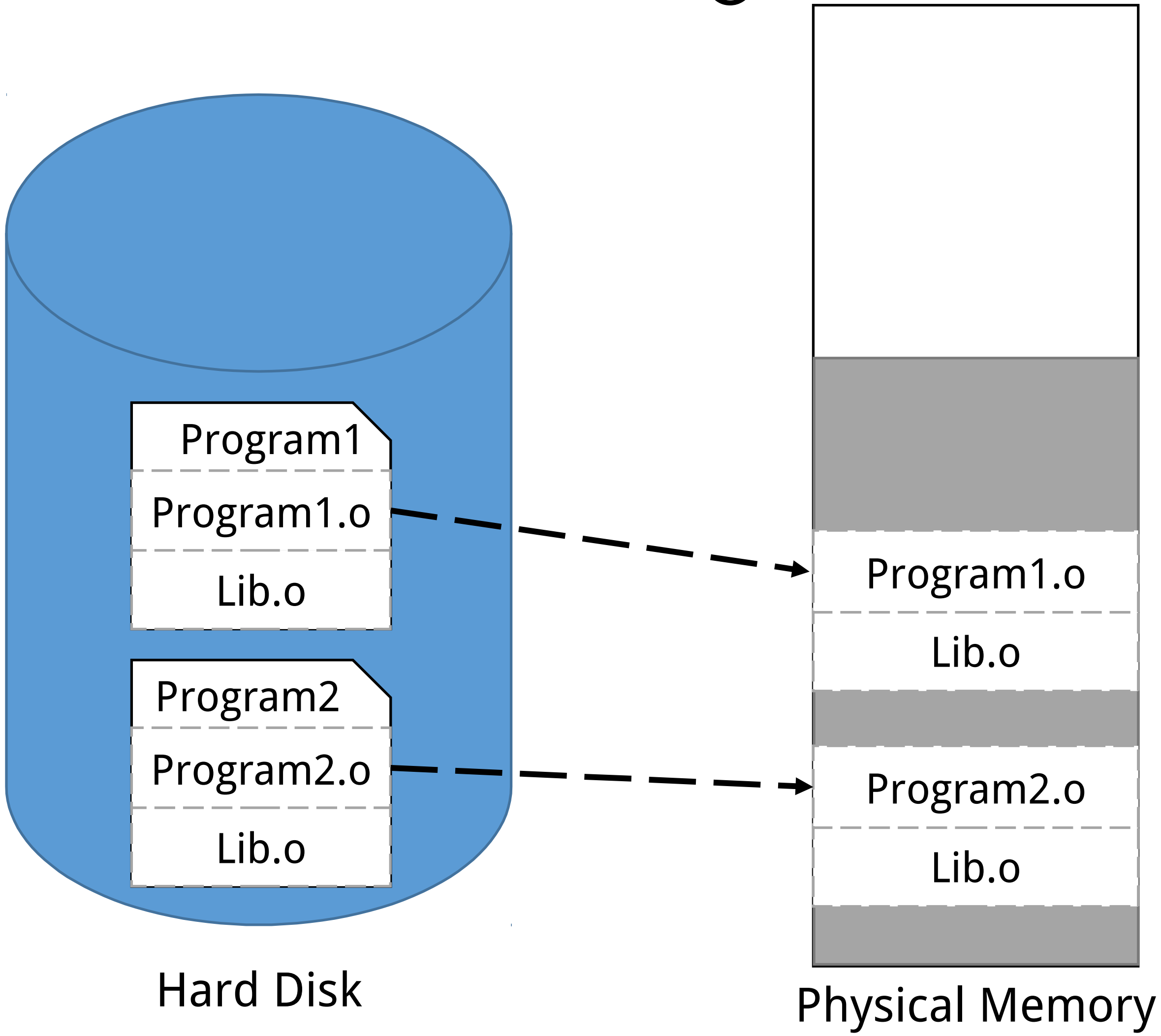
How Linux Kernel Loads ELF File

1. Check file format(magic number, segment, ...)
2. Search dynamic linking section “.interp”
3. According to program header, map ELF file(code, data, rodat a)
4. Initialize ELF context environment
5. Modify return address to program entry

Dynamic Linking

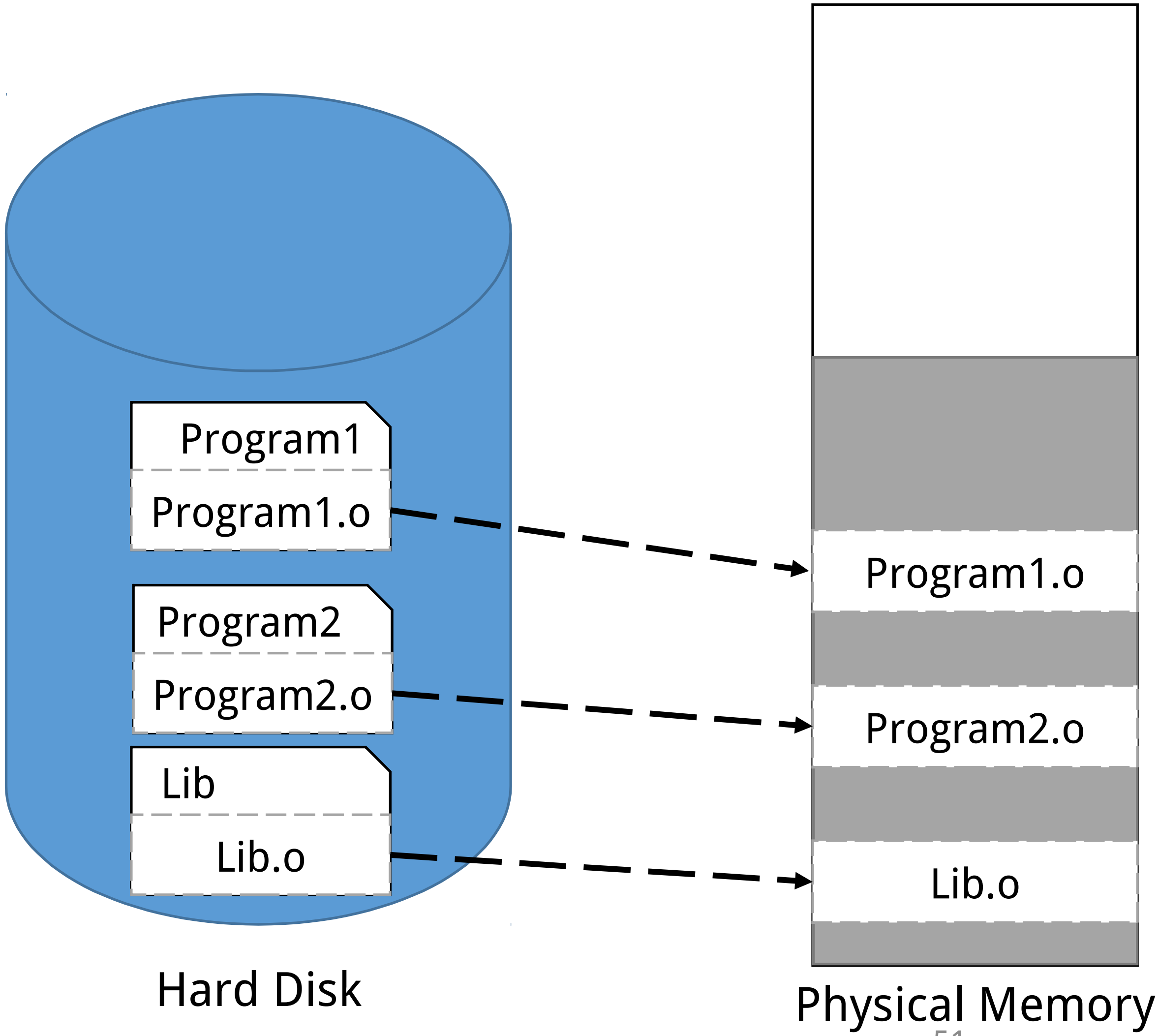
Disadvantage of Static Linking

- Advantage
  - ➢Independent development
  - ➢Test individual modules
- Disadvantage
  - ➢Waste memory and disk space
    - □Every program has a copy of runtime library(printf, scanf, strlen, ...)
  - ➢Difficulty of updating module
    - □Need to re-link and publish to user when a module is updated



Dynamic Linking

- Delay linking until execution
- Example:
  - Program1.o, Program2.o, Lib.o
  - Execute Program1 → Load Program1.o
  - Program1 uses Lib → Load Lib.o
  - Execute Program2 → Load Program2.o
  - Program2 uses Lib → Lib.o has already bee n loaded into physical memory
- Advantage
  - Save space
  - Easier to update modules



Basic Implementation

- Operating system support
  - ➢Process virtual address space allocation
  - ➢Storage manipulation
  - ➢Memory share
- Dynamic Shared Objects, DSO, .so file(in Linux)
- Dynamical Linking Library, .dll file(in Windows)

- Dynamic loader loads all dynamic linking libraries into memory
- Every time we execute the program, the loader will relocate the program
- Slowly
  - ➢Lazy Binding

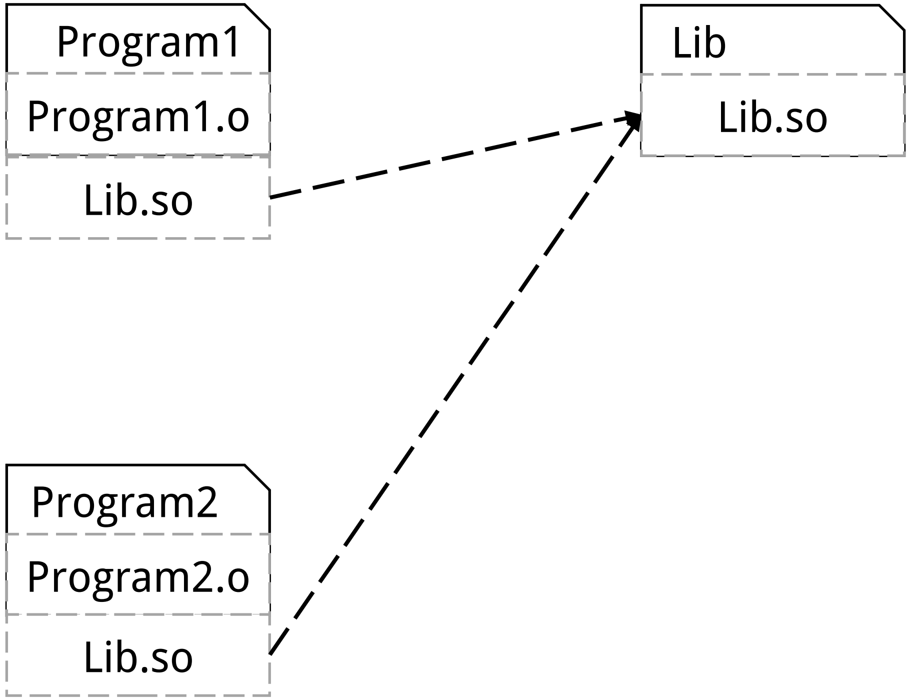
Dynamic Linking Example

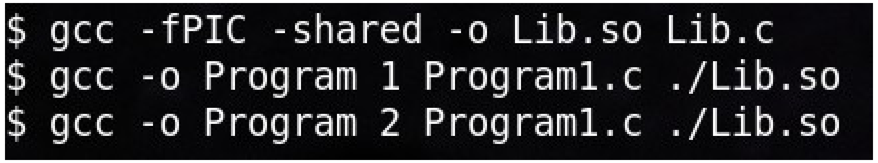
Program1.c

```
#include “Lib.h”

int main() {
    foobar(1);
}
```

Program2.c

```
#include “Lib.h”

int main() {
    foobar(2);
}
```





```
$ gcc -fPIC -shared -o Lib.so Lib.c
$ gcc -o Program 1 Program1.c ./Lib.so
$ gcc -o Program 2 Program1.c ./Lib.so
```

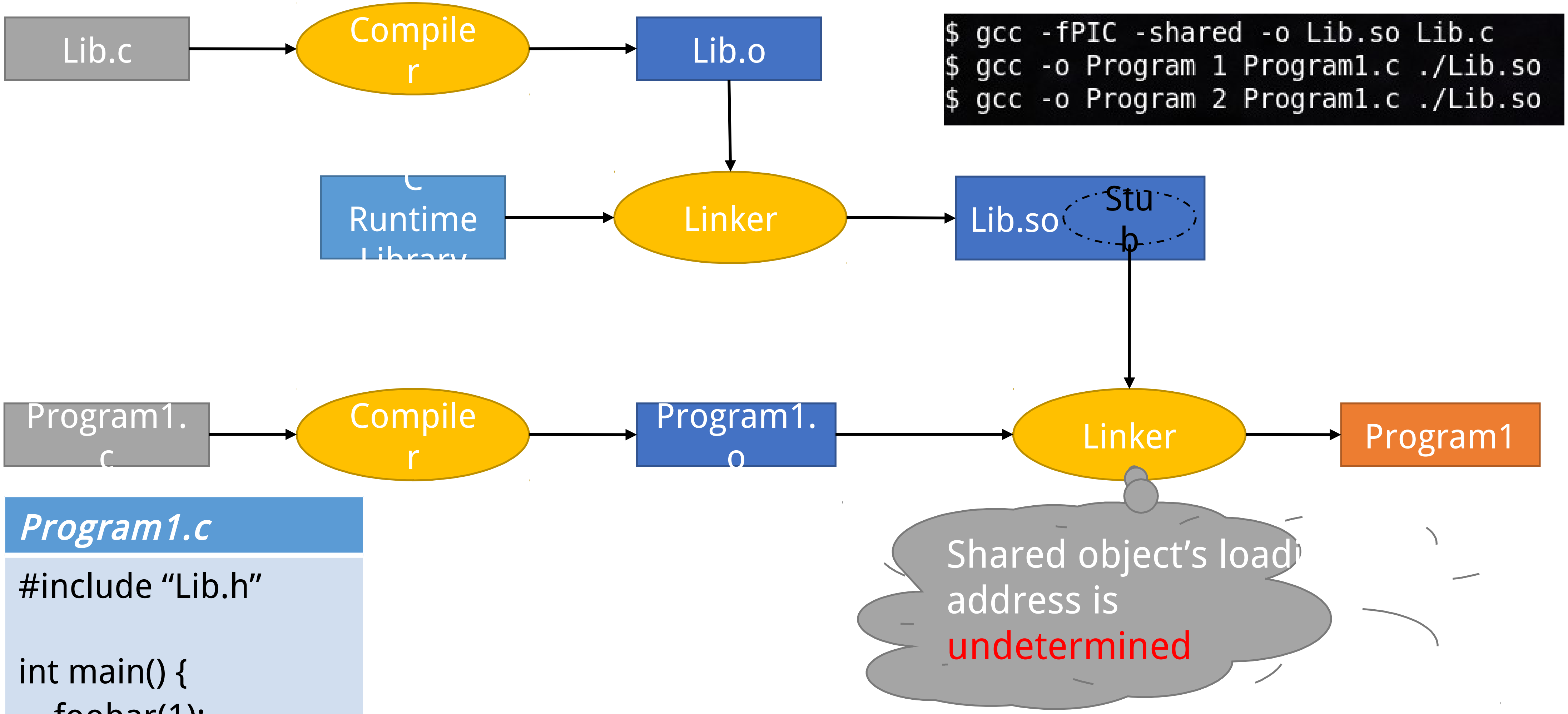
Lib.c

```
#include <stdio.h>

void foobar(int i) {
    printf(“%d\n”, i);
}
```

Lib.h

```
#ifndef LIB_H
#define LIB_H

void foobar(int);

#endif
```

Dynamic Linking Example



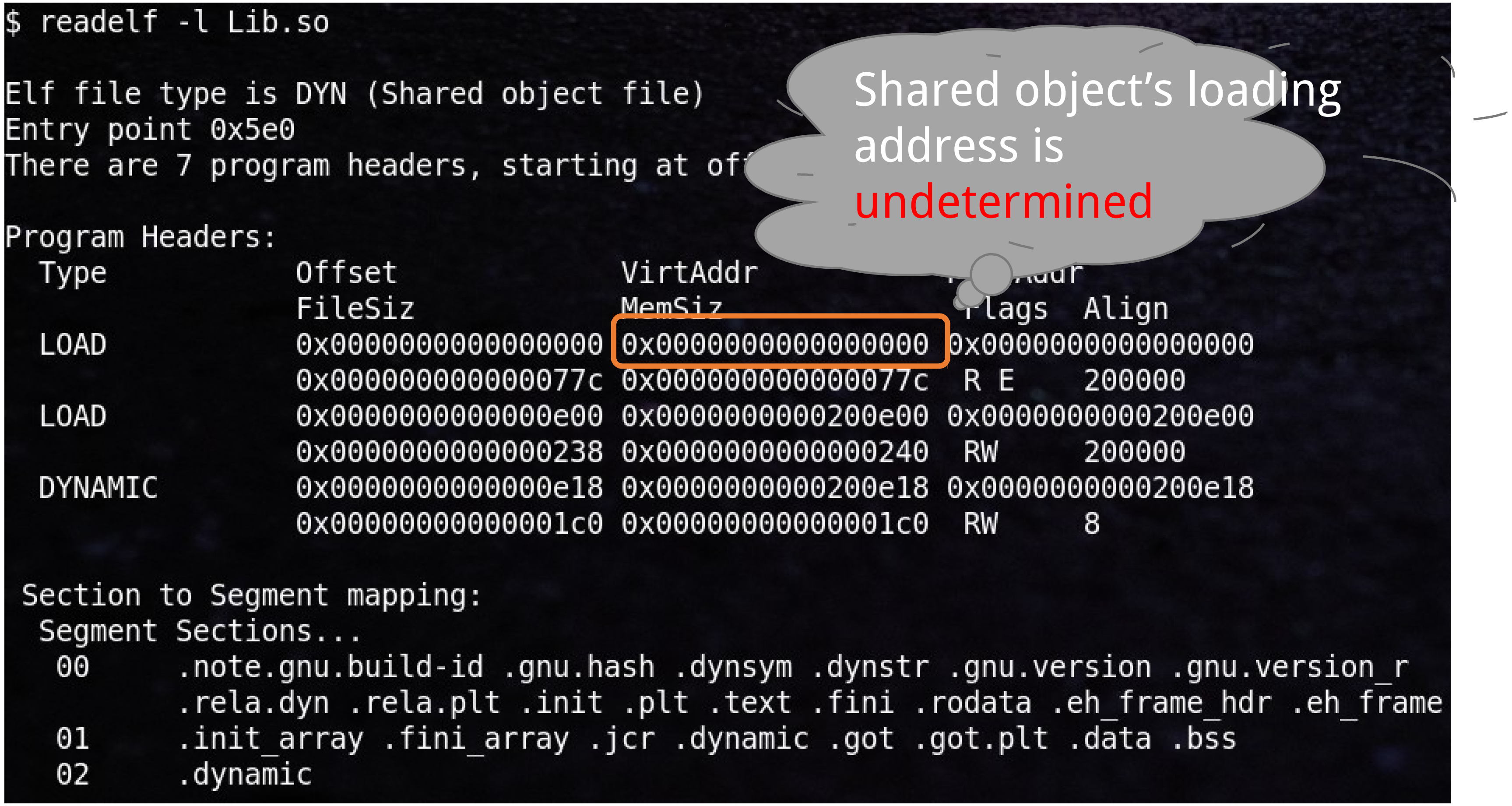
```
$ gcc -fPIC -shared -o Lib.so Lib.c
$ gcc -o Program 1 Program1.c ./Lib.so
$ gcc -o Program 2 Program1.c ./Lib.so
```

Shared object's loadi
address is
undetermined

Program1.c

```
#include "Lib.h"

int main() {
    foobar(1);
}
```

Dynamic Linking Example



```
$ readelf -l Lib.so

Elf file type is DYN (Shared object file)
Entry point 0x5e0
There are 7 program headers, starting at off

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x000000000000077c 0x000000000000077c  R E    200000
  LOAD           0x0000000000000e00 0x0000000000200e00 0x0000000000200e00
                 0x0000000000000238 0x0000000000000240  RW     200000
  DYNAMIC        0x0000000000000e18 0x0000000000200e18 0x0000000000200e18
                 0x00000000000001c0 0x00000000000001c0  RW     8

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r
          .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   01     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   02     .dynamic
```

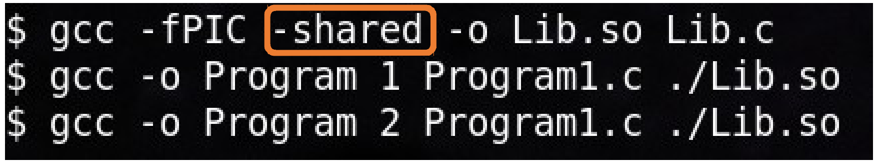
Shared object's loading address is undetermined

Static Shared Library

- Not Static Library
- Load module into particular position
- Ex.
  - Allocate 0x1000~0x2000 to Module A
  - Allocate 0x2000~0x3000 to Module B
- Collision
  - User D allocate 0x1000~0x2000 to Module C
  - Then other people can not use Module A and Module C simultaneously

Load Time Relocation

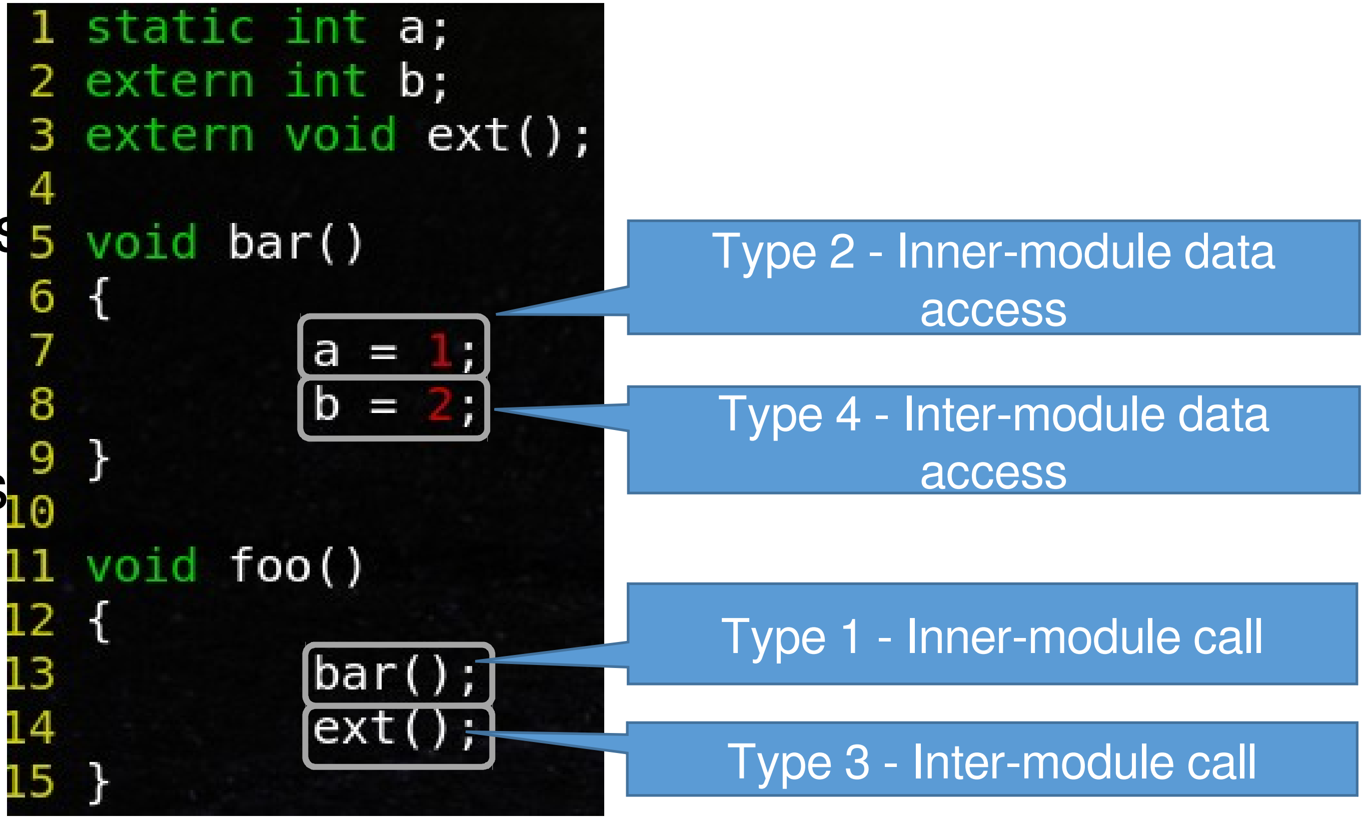
- Relocate absolute address at load time instead of link time
- Example:
  - Function “foobar” has offset 0x100
  - Module is loaded into 0x10000000
  - Then we know function “foobar” at 0x10000100
  - Traverse the relocation table, relocate function “foobar” to 0x10000100
- Multiple processes use the same object, but relocation are different between processes
  - They can not use the same copy of shared object
- Compile with “-shared” argument



```
$ gcc -fPIC -shared -o Lib.so Lib.c
$ gcc -o Program 1 Program1.c ./Lib.so
$ gcc -o Program 2 Program1.c ./Lib.so
```

Position-independent Code (PIC)

- Move the part which should be modified out of normal code section, then every process can have an individual copy of that section
- Address reference type
  - ➢Type 1 - Inner-module call
  - ➢Type 2 - Inner-module data acces
  - ➢Type 3 - Inter-module call
    - □Global Offset Table, GOT
  - ➢Type 4 - Inter-module data acces
    - □Same as type 3
- Compile with “-fPIC” argument



```
static int a;
extern int b;
extern void ext();

void bar()
{
        a = 1;
        b = 2;
}

void foo()
{
        bar();
        ext();
}
```

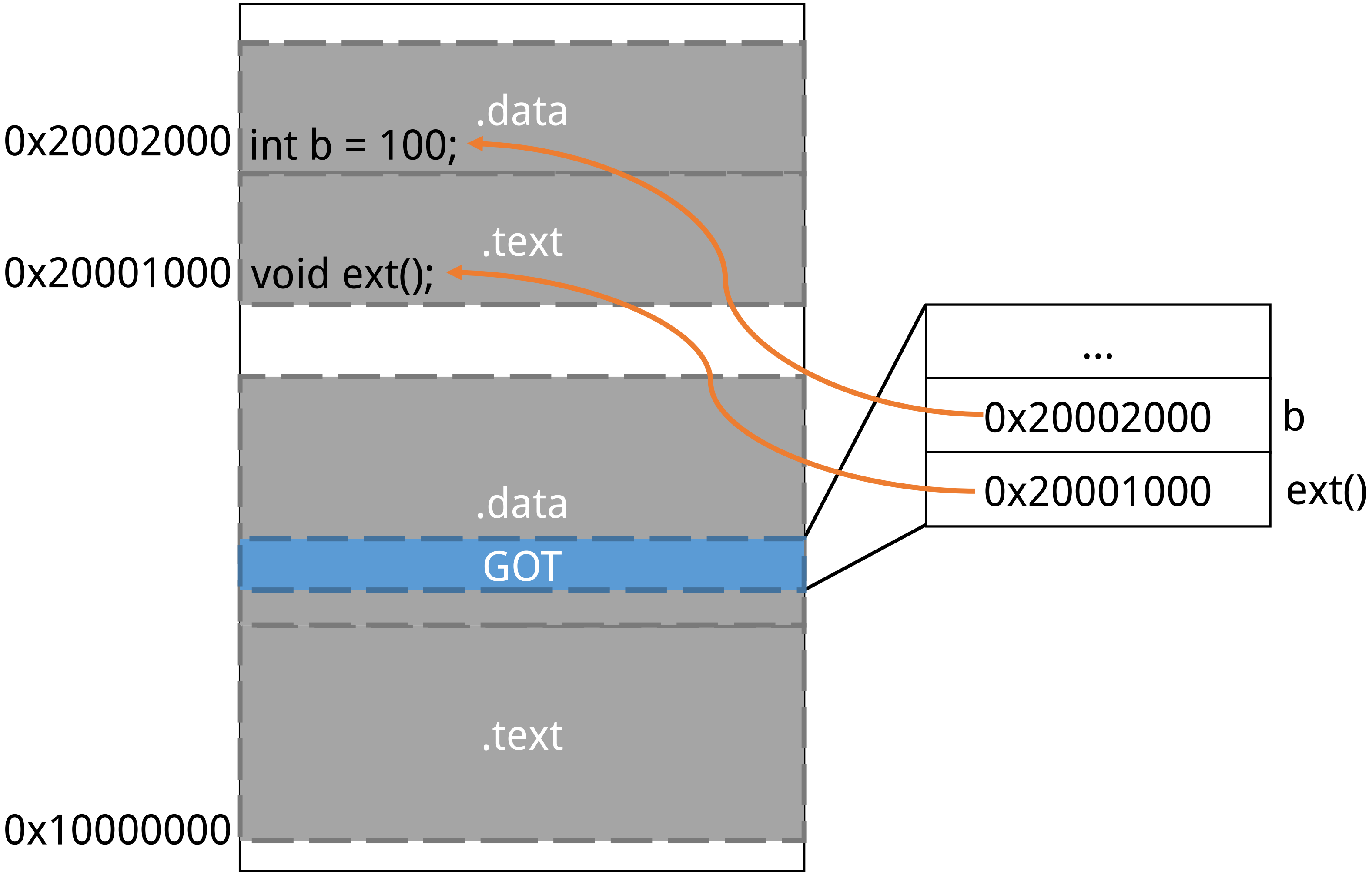
Type 2 - Inner-module data access

Type 4 - Inter-module data access

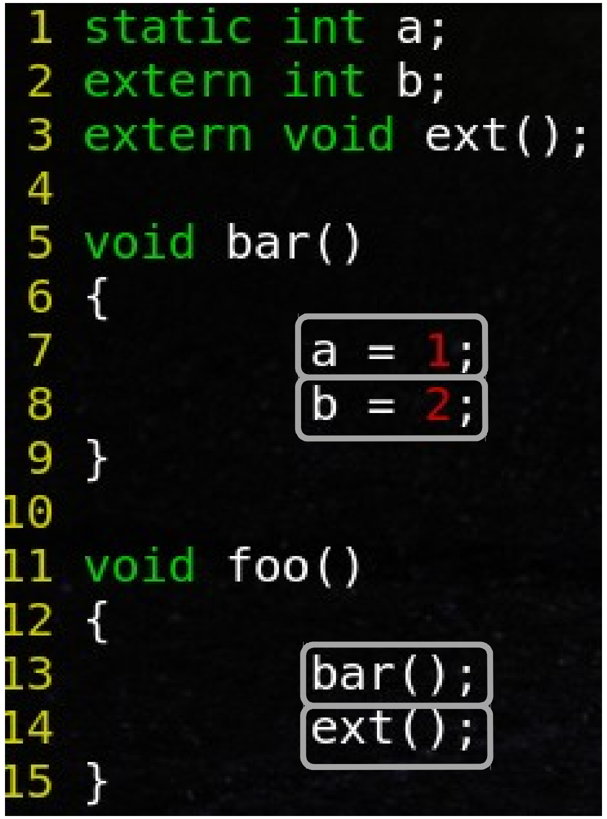
Type 1 - Inner-module call

Type 3 - Inter-module call

Global Offset Table (GOT)



Process Virtual Space



```
static int a;
extern int b;
extern void ext();

void bar()
{
        a = 1;
        b = 2;
}

void foo()
{
        bar();
        ext();
}
```

Dynamic Linking Overhead

- Although dynamic linking program is more flexible, but...
- Static linking is faster than dynamic linking program about 1% to 5%
  - ➢Global , static data access and inter-module calls need complex GOT re-location
  - ➢Load program → Dynamic loader have to link the program

Lazy Binding

- Bind when the first time use the function(relocation, symbol sea rching)
- Dynamic loader view
  - “liba.so” calls function “bar” in “libc.so”
  - We need dynamic loader do address binding, and assume the work is d one by function “lookup”
  - Function “lookup” needs two parameters: module & function
  - “lookup()” in Glibc is “_dl_runtime_resolve()”
- Procedure Linkage Table, PLT

Implementation of PLT

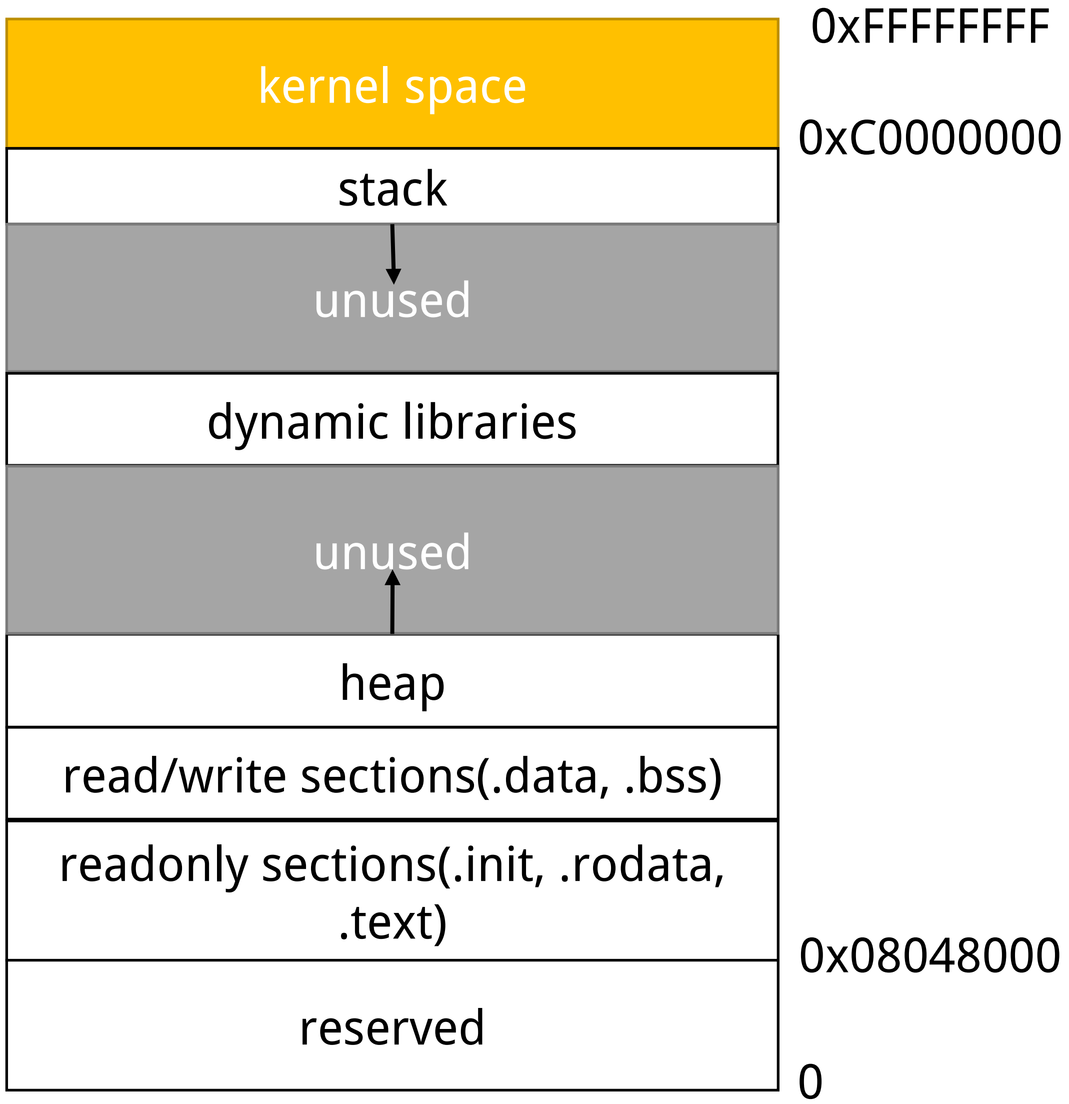
- ~~Inter-module function call → GOT~~
- Inter-module function call → PLT → GOT
- Every inter-module function have a corresponding entry in PLT
  - ➢Function “bar” in PLT → bar@plt
  - ➢bar@GOT = next instruction(push n)
  - ➢n = index of “bar” in “.rel.plt”
- “_dl_runtime_resolve” will modify “bar@GOT” to actual “bar” address

```
bar@plt
jmp *(bar@GOT)
push n
push moduleID
jump _dl_runtime_resolve
```

Memory

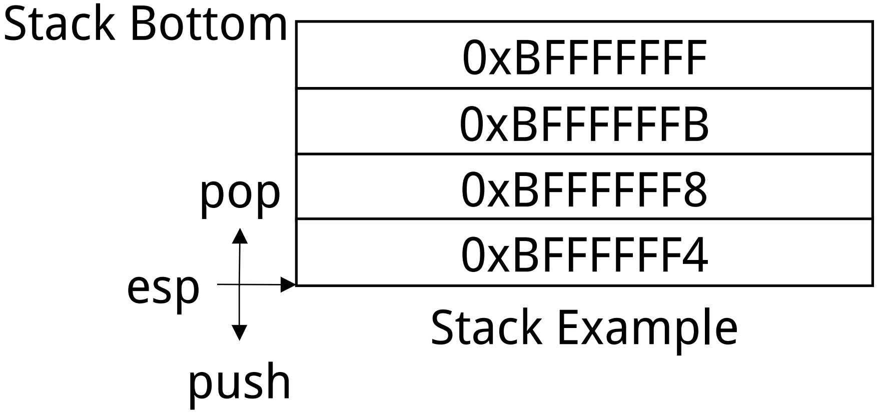
Program Memory Layout

- Flat memory model
- Default regions:
  - ➢stack
  - ➢heap
  - ➢mapping of executable file
  - ➢reserved
  - ➢dynamic libraries



Stack

- Stack Frame(Activate Record)
  - Return address, arguments
  - Temporary variables
  - Context
- Frame Pointer(ebp on i386)
- Stack Pointer(esp on i386)



Stack Bottom

0xBFFFFFFF

0xBFFFFFFB

0xBFFFFFF8

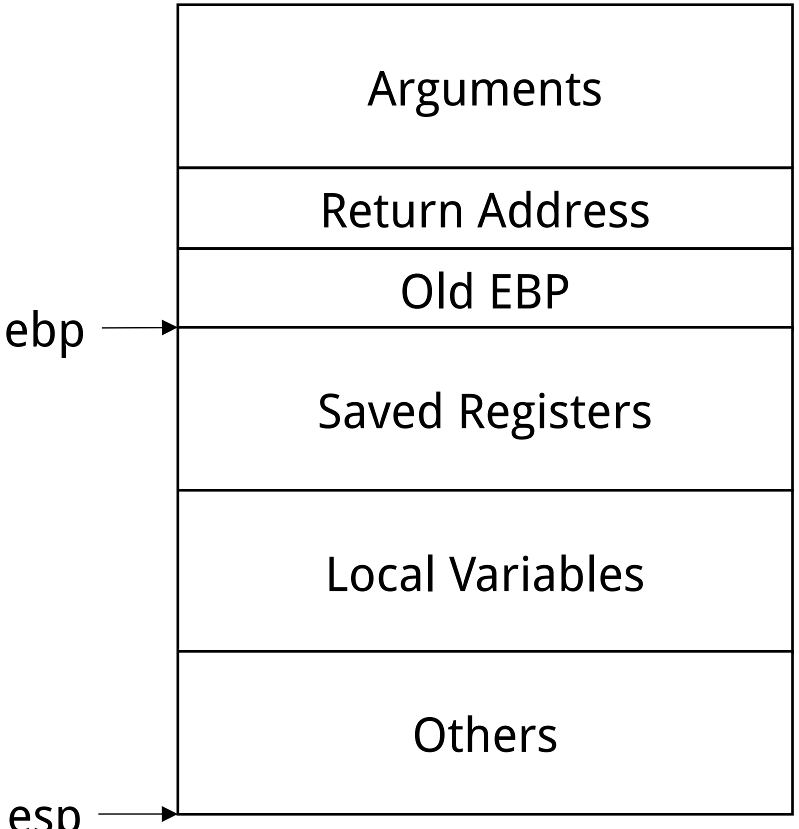
0xBFFFFFF4

pop

esp

push

Stack Example



Arguments

Return Address

Old EBP

ebp

Saved Registers

Local Variables

Others

esp

Activate Record

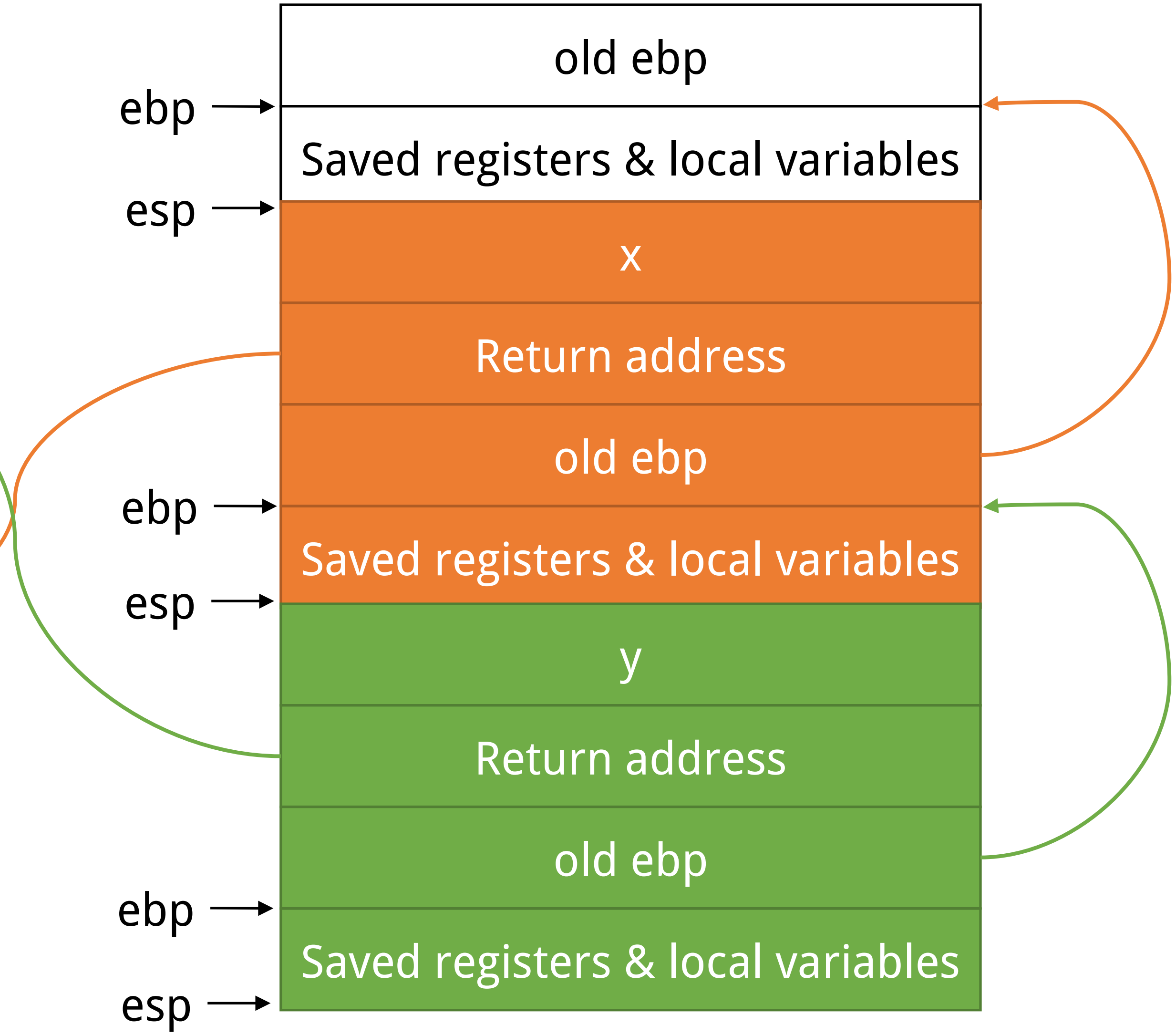
Calling Convention

- Consistency between caller and callee
- Argument passing order and method
  - Stack, Register(eax for return value on i386)
- Stack maintainer
  - Keep consistency before and after function call
  - Responsibility of caller or callee
- Name-mangling
- Default calling convention in C language is “cdecl”

Arguments passing	Stack maintainer	Name-mangling
Push into stack from right to left	Caller	Underscore in front of function name

Calling Convention Example

```
int f(int y) {
    printf("%d", y);
    return 0;
}

int main() {
    int x = 1;
    f(x);
    return 0;
}
```



old ebp

ebp → Saved registers & local variables

esp → x

Return address

old ebp

ebp → Saved registers & local variables

esp → y

Return address

old ebp

ebp → Saved registers & local variables

esp →

Heap

- Dynamic allocate memory

```
int main() {
    char *p = (char *)malloc(1000 * sizeof(char));
    /* use p as an array of size 1000 */
    free(p);
}
```

- Implementation under Linux
  - int brk(void *end_data_segment)
  - void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)
- Algorithms for memory allocation
  - Free List
  - Bitmap
  - Object Collection

System Call & API

System Call?

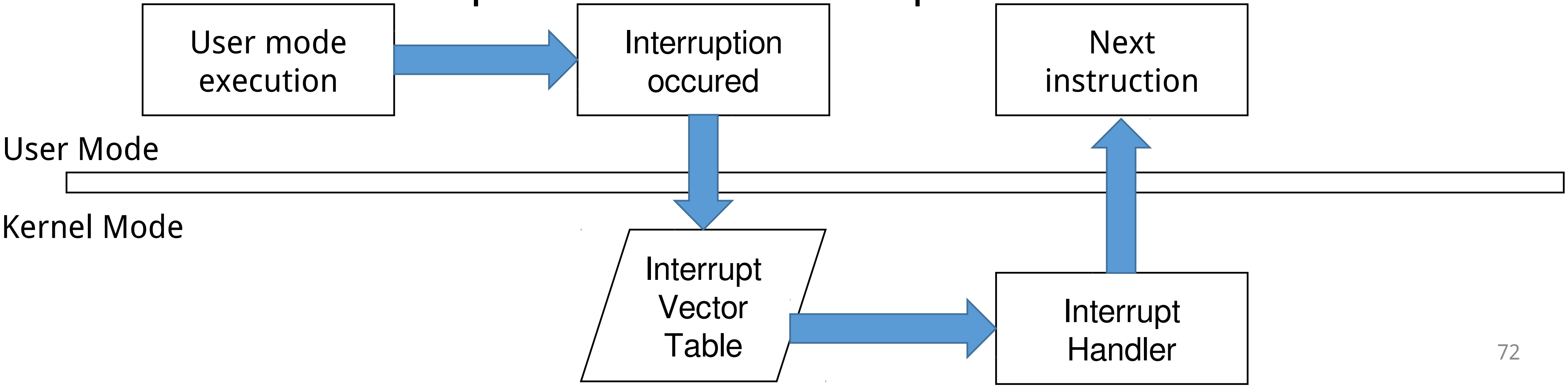
- Process can not access system resource directly
  - File, Network, Input/Output, Device
- Something we need OS help us
  - e.g. `for(int i = 0; i < 10000; i++)`
- Process management, system resource access, GUI operation...
- Drawbacks
  - Too native → Runtime Library
  - Difference between various OSs

Privilege

- Modern CPU architectures usually have multi-level design
  - User Mode
  - Kernel Mode
- high privilege → low privilege is allowed
- low privilege → high privilege is not easy
- Restrict some operations in low privileged mode
  - Stability
  - Security
- OS usually uses interrupt as mode switch signal

Interrupt

- Polling
- Interrupt
  - Interrupt Index
  - Interrupt Service Routine (ISR)
- Hardware interrupt & Software interrupt



System Call Example

- rtenv+
- ARM Cortex-M3
- https://hackpad.com/RTENV-xzo9mDkptBW#

Thinking

```
~$ vim hello.c
~$ gcc hello.c
~$ ./a.out
Hello World!
```

Filename: hello.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf(“Hello World!\n”);

    return 0;
}
```

- Why do we need to compile the program
- What is in an executable file
- What is the meaning of “#include<stdio.h>”
- Difference between
  - Compiler(Microsoft VC, GCC)
  - Hardware architecture(ARM, x86)
- How to execute a program
  - What does OS do
  - Before main function
  - Memory layout
  - If we don’t have OS