National Yang Ming Chiao Tung University
Computer Architecture & System Lab

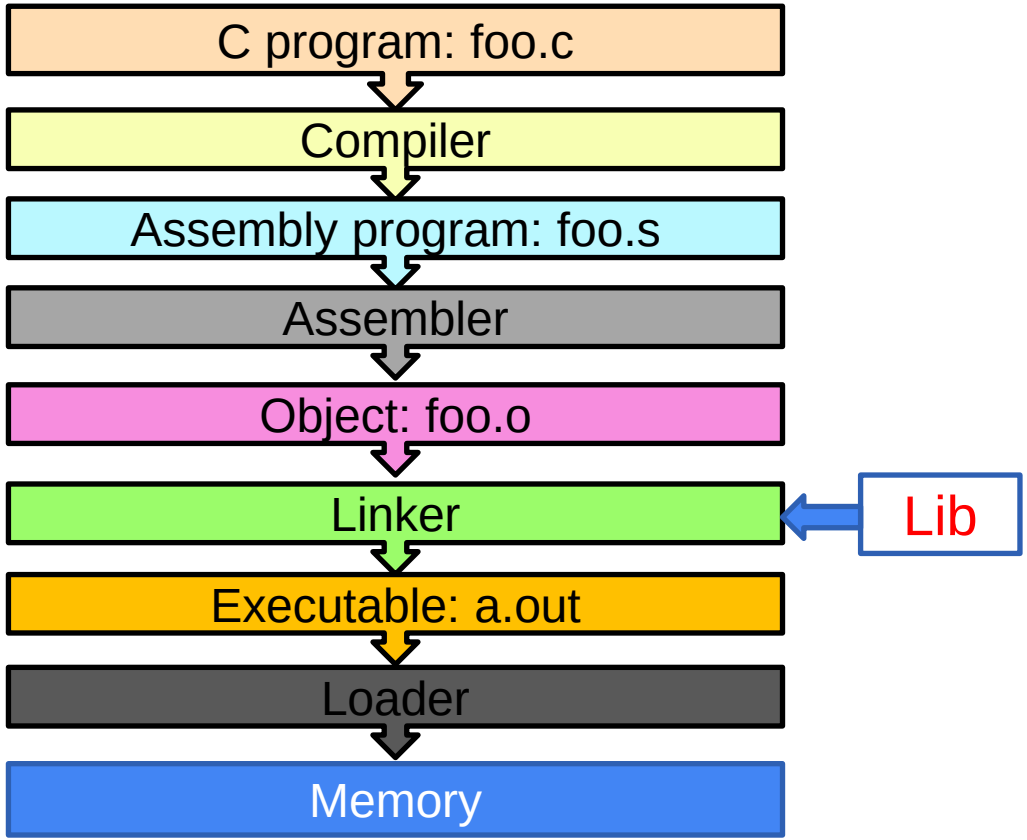
Linker

IOC5226 Operating System Capstone

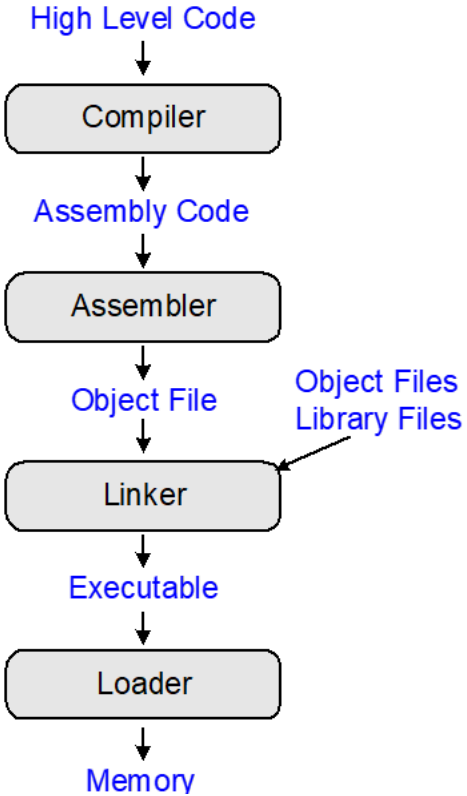
Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Outline

- Translation
- Compiler
- Assembler
- Linker
- Loader
- Example



How to Compile & Run a Program



The Compilation Pipeline

- How source code becomes a running program ?
  - **Preprocessing**
    - The preprocessor handles #include and #define statement
  - **Compilation**
    - The compiler turns C/C++ code into assembly language
  - **Assembly**
    - The assembler converts assembly into object files (.o or .obj)
  - **Linking**
    - The linker combines object files and libraries into an executable

Linker

- **Input**
  - Object code files, information tables (e.g. foo.o, lib.o for RISC-V)
- **Output**
  - Executable code (e.g. a.out for RISC-V)
- Combines several object (.o) files into a single executable (“linking”)
- Enables separate compilation of files
  - The linker’s job is to figure out where every reference and definitions should live in the final executable

What are symbols?

- **A symbol**
  - Any name entity in your program that the linker needs to keep track of. The symbol includes
  - **Functions** (main, printf, malloc)
  - **Global variables** (extern int counter)
  - **Static variables** (those marked with static)
  - **Class methods**
    - Ex. When you write int global_counter = 42; at the top of your C file, you create a symbol called global_counter that points to a specific spot in memory where the value 42 lives

Symbol Types

- The linker categorizes symbols into two types that affect how conflicts are resolved
- **Strong symbols includes**
  - Functions that are defined (have a body)
  - Initialized global variables
- **Weak symbols**
  - Uninitialized global variables
  - Function declarations without definitions

Symbol Table

- The symbol table: A linker's phonebook
  - Every object file contains a symbol table

Symbol Table Structure

Symbol Name	Address	Size	Type	Binding	Section	Description
main	0x08048400	24	FUNC	GLOBAL	.text	Main function entry point
printf	undefined	-	FUNC	GLOBAL	UND	External library function
global_counter	0x08049540	4	OBJECT	GLOBAL	.data	Initialized global variable
static_var	0x08049544	4	OBJECT	LOCAL	.data	Static variable (file scope)
uninitialized_array	0x08049580	1024	OBJECT	WEAK	.bss	Uninitialized global array
helper_function	0x08048450	16	FUNC	GLOBAL	.text	User-defined function

Symbol Table Legend

Type	Binding	Section	Address
FUNC - Function	GLOBAL - Visible globally	.text - Executable code	0x08048xxx - Code segment
OBJECT - Variable	LOCAL - File scope only	.data - Initialized data	0x08049xxx - Data segment
SECTION - Section	WEAK - Can be overridden	.bss - Uninitialized data	undefined - External ref

https://reurl.cc/j6M1zq

Symbol Table

- We can peek at the symbol table using tools
  - “nm” on Unix or “dumpbin” on Windows

```
# Compile a simple C file
gcc -c example.c -o example.o

# Look at the symbol table
nm example.o
```

```
0000000000000000 T main
                 U printf
0000000000000004 D global_var
```

https://reurl.cc/j6M1zq

Symbol Table Example

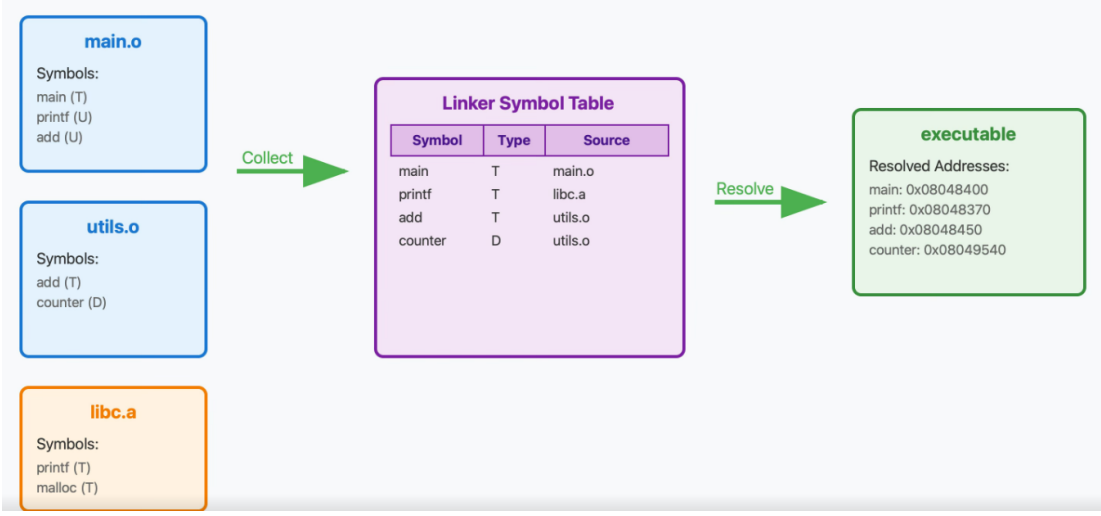
Symbol	Type
bar	U
dec	U
main	T
“Here”	D
num	D
printf	U
“%\n”	D

U: undefined indicates the external file reference
T: .Text section
D: .Data section

```
1  #include <stdio.h>
2  extern int bar();
3  extern int dec;
4  int main() {
5       char *output = “Here”;
6       static int num = 7;
7       int i = 5;
8       while (i > 0) {
9            i --;
10           int temp = bar(num);
11           printf(“%d\n”, temp);
12      }
13}
```

Symbol Table

- What happens when you link multiple object files?
  - **Step 1: Collect references and definitions**
    - Symbols that are referenced but not defined
    - Symbols that are defined



https://reurl.cc/j6M1zq

Symbol Table

- What happens when you link multiple object files?
  - **Step 2: Match references to definitions**
    - For each undefined symbol, the linker searches through all the definitions it has collected
    - If it finds zero definitions, that's an "undefined reference" error
    - If it finds multiple definitions, it applies the strong/weak rules

https://reurl.cc/j6M1zq

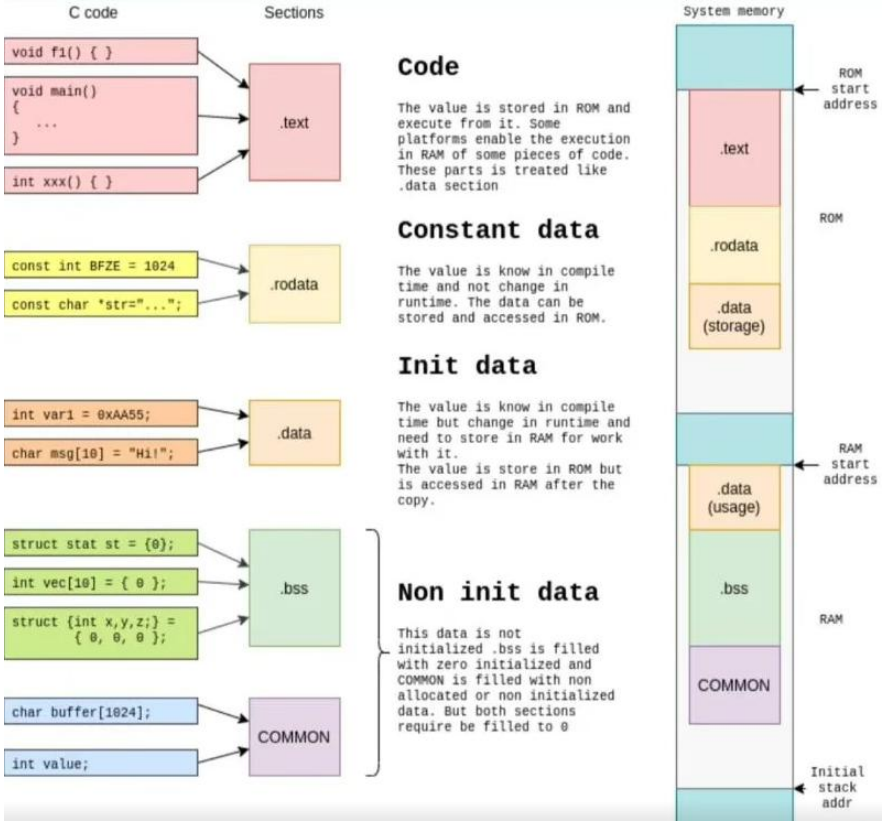
Symbol Table

- What happens when you link multiple object files?
  - **Step 3: Handle special cases**
    - **Common symbols**
      - Uninitialized global variables that might be defined in multiple files
    - **Static symbols**
      - These have local scope and don't participate in global symbol
    - **Library symbols**: only pulled in if they are needed

https://reurl.cc/j6M1zq

Where symbols live: The ELF Format

- On Unix-like systems, object files and executables use **ELF (Executable and Linkable Format) format**



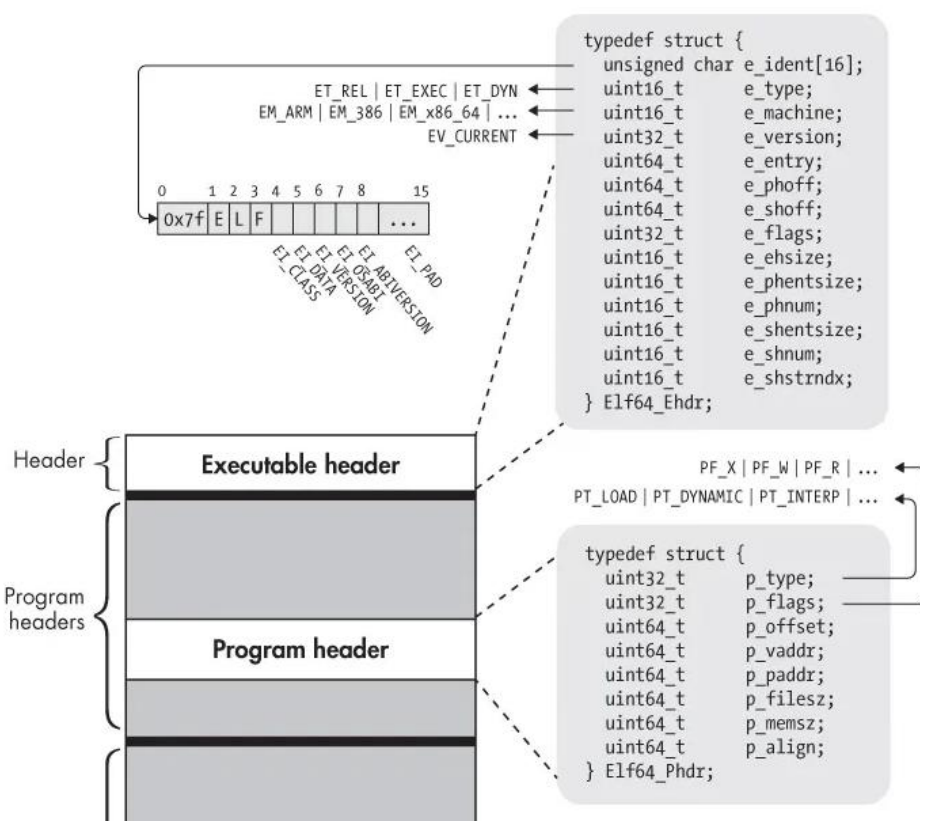
https://reurl.cc/j6M1zq

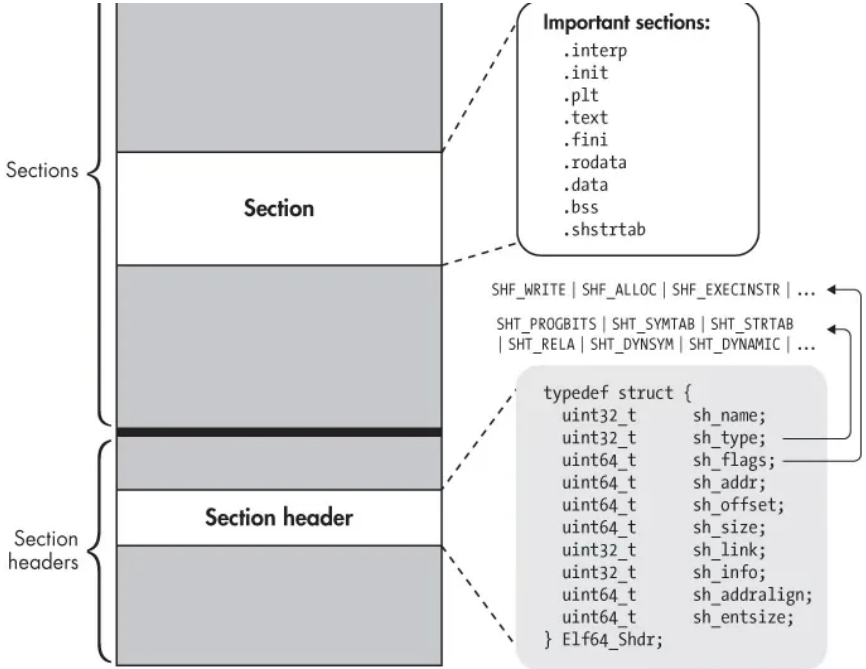
A ELF File

- A ELF file is divided into sections
  - **.text**: The actual machine code
  - **.data**: Initialized global variables
  - **.bss**: Uninitialized global variables
  - **.symtab**: The symbol table
  - **.rel.text and .rel.data**: Relocation entries

https://reurl.cc/j6M1zq

A ELF File





https://reurl.cc/j6M1zq

Relocation

- Symbol resolution
  - Tell the linker which symbols go with which definitions
- Relocation
  - This is where the linker patches up all the addresses in your code

```
# Before linking
call printf    # This is actually "call <placeholder>"
mov eax, [global_var]  # This is "mov eax, [<placeholder>]"

# After linking
call 0x08048370    # Actual address of printf
mov eax, [0x08049540]  # Actual address of global_var
```

https://reurl.cc/j6M1zq

Common Symbol Resolution Problems

- **Undefined References**
- Common causes
  - Forgot to link a required library
  - Misspelled a function name
  - Forgot to compile and link a source file
  - Function is declared but never defined

```
undefined reference to `function_name'
```

https://reurl.cc/j6M1zq

Common Symbol Resolution Problems

- **Link order matters**
  - The linker processes files left to right
  - The linker only pulls in library symbols that are needed to resolve undefined references it has already seen

```
# This might fail
gcc -lmath main.o  # libmath is processed before main.o

# This should work
gcc main.o -lmath  # main.o is processed first, creating references that libmath can satisfy
```

https://reurl.cc/j6M1zq

Shared Libraries and Dynamic Loading

- **Static linking**
  - Create static executables where everything is bundled together
- **With shared libraries**
  - .so file on Unix, .dll files on Windows
  - Some symbols aren’t resolved until the program actually runs
- **The dynamic linker**
  - ld.so on Linux handles this runtime symbol resolution

https://reurl.cc/j6M1zq

Shared Libraries and Dynamic Loading

- **The dynamic linker**
  - ld.so on Linux handles this runtime symbol resolution
  - **Lazy binding:**
    - Function addresses are resolved only when first called
  - **Global symbol interposition**
    - Symbols in the main program can override library symbols

https://reurl.cc/j6M1zq

Debugging Symbol Resolution Issues

- **nm:** List symbols in object files

```
nm -u program.o  # Show undefined symbols
nm -D program    # Show dynamic symbols
```

- **Objdump**: Disassembles object files and shows relocation info

```
objdump -t program.o  # Show symbol table
objdump -r program.o  # Show relocation entries
```

https://reurl.cc/j6M1zq

Debugging Symbol Resolution Issues

- **ldd:** Show shared library dependencies

```
ldd program  # List dynamic dependencies
```

- **readelf**: Detailed ELF file analysis

```
readelf -s program.o  # Symbol table
readelf -r program.o  # Relocation entries
```

https://reurl.cc/j6M1zq

Performance Consideration

- Symbol resolution can be a bottleneck for large programs
- Strategies to improve link time
  - **Minimize symbol visibility**
    - Use static for functions and variables that don't need to be visible outside their compilation unit
    - Reduce the number of symbols the linker needs to process

https://reurl.cc/j6M1zq

Performance Consideration

- Strategies to improve link time
  - **Avoid Header-Only libraries**
    - Header-only libraries can lead to code bloat and slow link times
    - The same code gets compiled into multiple object files
  - **Use Link-Time Optimization (LTO)**
    - Modern compilers can optimize across compilation units during link
    - This make the linking process slower while potentially making the final program faster

https://reurl.cc/j6M1zq

Linker Script

- The main purpose of the linker script
  - Describe how the sections in the input files should be mapped into the output file
  - To control the memory layout of the output file

Linker Script

- Put code section on 0x10000 and data section on 0x8000000
- Line 1: declare the SECTIONs
- Line 3: “.” is **location counter**
  - Location counter represents the current location of the section
  - The value of initial location is 0

```
1  SECTIONS
2  {
3      . = 0x10000;
4      .text :
5      {
6          *(.text)
7      }
8      . = 0x8000000;
9      .data :
10     {
11         *(.data)
12     }
13     .bss :
14     {
15         *(.bss)
16     }
17 }
```

Linker Script

- Put code section on 0x10000 and data section on 0x8000000
- Line 2: set the location counter to 0x10000
- Line 4: Define .text section
- Line 6: put all input files' .text input section here
  * is a wildcard symbol
- Line 8: move location counter to 0x8000000

```
1  SECTIONS
2  {
3      . = 0x10000;
4      .text :
5      {
6          *(.text)
7      }
8      . = 0x8000000;
9      .data :
10     {
11         *(.data)
12     }
13     .bss :
14     {
15         *(.bss)
16     }
17 }
```

https://reurl.cc/0aLmZA

Linker Script

test.ld

```
SECTIONS
{
. = 0x10000;
.text : { *(.text) }
. = 0x8000000;
.data : { *(.data) }
.bss : { *(.bss) }
}
```

main.c

```c
void test(void);

int global_bss;
int global_data = 123;

int main()
{
    global_bss = 0;
    test();
    global_data++;
    return 0;
}
```

test.c

```c
void test(void)
{
    int i;
    // do nothing.
    for (i = 0; i < 10000; i++);
}
```

https://reurl.cc/0aLmZA

Linker Script

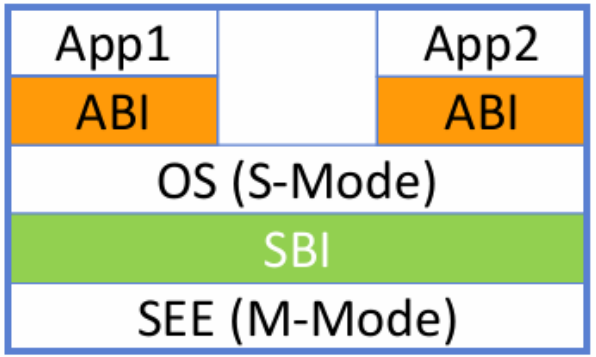
1. Virtual Memory Address (**VMA**): the address the section will have when the output file is run
2. Load Memory Address (**LMA**): the address at which the section will be loaded

```
$ gcc -c main.c test.c
$ ld -T test.ld main.o test.o
$ objdump -h a.out

a.out:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000046  0000000000010000  0000000000010000  00010000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .eh_frame     00000058  0000000000010048  0000000000010048  00010048  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000004  0000000008000000  0000000008000000  00200000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00000004  0000000008000004  0000000008000004  00200004  2**2
                  ALLOC
  4 .comment      00000011  0000000000000000  0000000000000000  00200004  2**0
                  CONTENTS, READONLY
```


https://reurl.cc/0aLmZA

What is SBI?

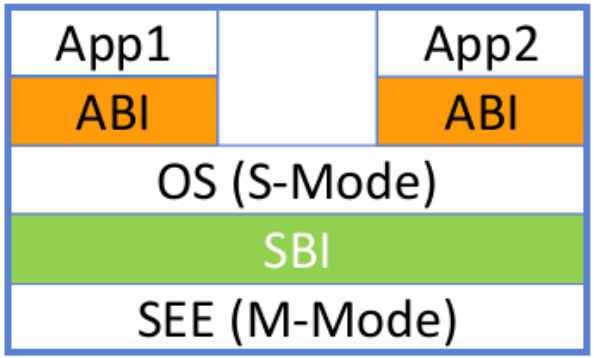
- SBI stands for RISC-V Supervisor Binary Interface
  - System call style calling convention between Supervisor (S-mode) and Supervisor Execution Environment (SEE)
- SEE can be:
  - A M-mode runtime firmware for OS/Hypervisor running in Hypervisor (HS)-mode
  - A HS-mode Hypervisor for Guest OS running in Supervisor (VS)-mode



https://reurl.cc/nlkpGe

What is SBI?

- SBI calls help:
  - Reduce duplicate platform code across OSes (Linux, FreeBSD …)
  - Provide drivers for an OS which can be shared by multiple platforms
  - Provide an interface for direct access to hardware resources (M-mode only resource)



https://reurl.cc/nlkpGe

What is OpenSBI?

- OpenSBI is
  - An open-source implementation of RISC-V Supervisor Binary Interface (SBI) specifications
  - Aim to provide runtime services in M-mode
  - Provide reference platforms for generic simple drivers
    - E.g. PLIC, CLINT, UART …
    - https://github.com/riscv-software-src/opensbi

https://reurl.cc/nlkpGe

Summary

- Symbols are named entities (functions, variables) that need addresses in the final executable
- The linker follows specific rules to resolve conflicts between multiple symbol definitions