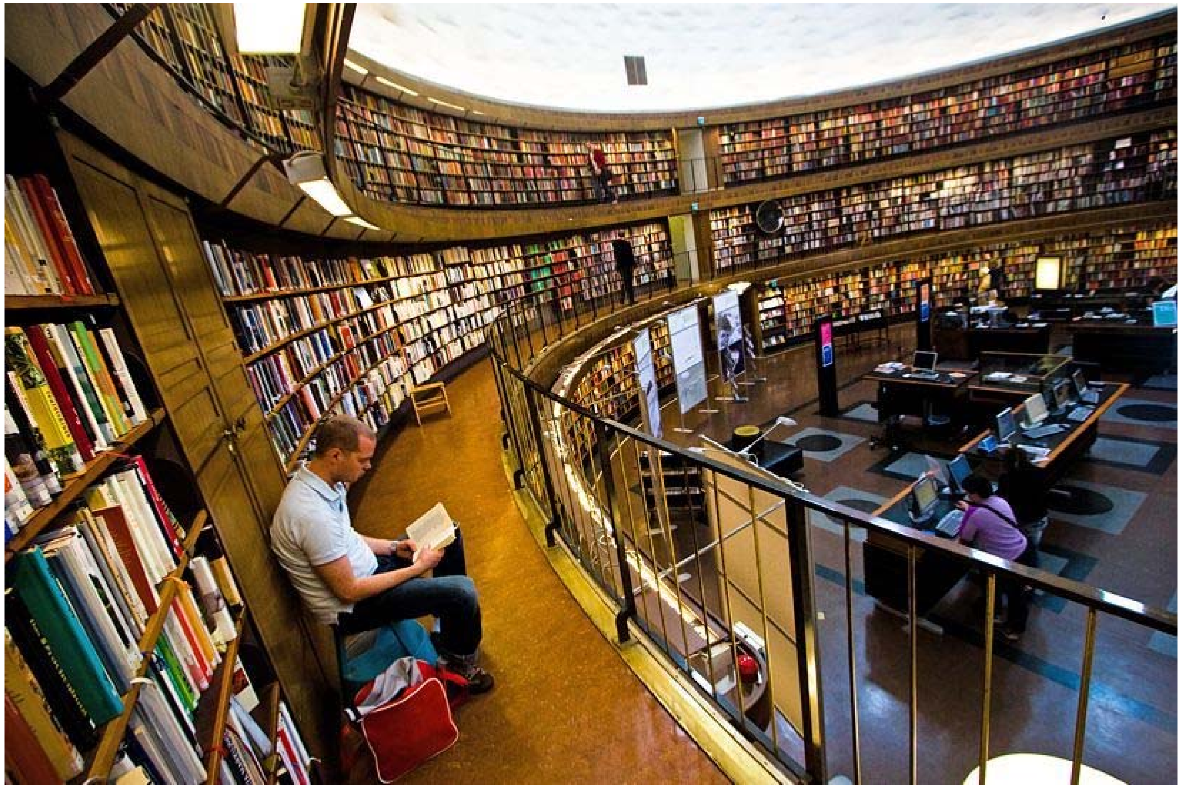
Linux Programming

1. Linux Library
2. Make Utility
3. ELF File Format

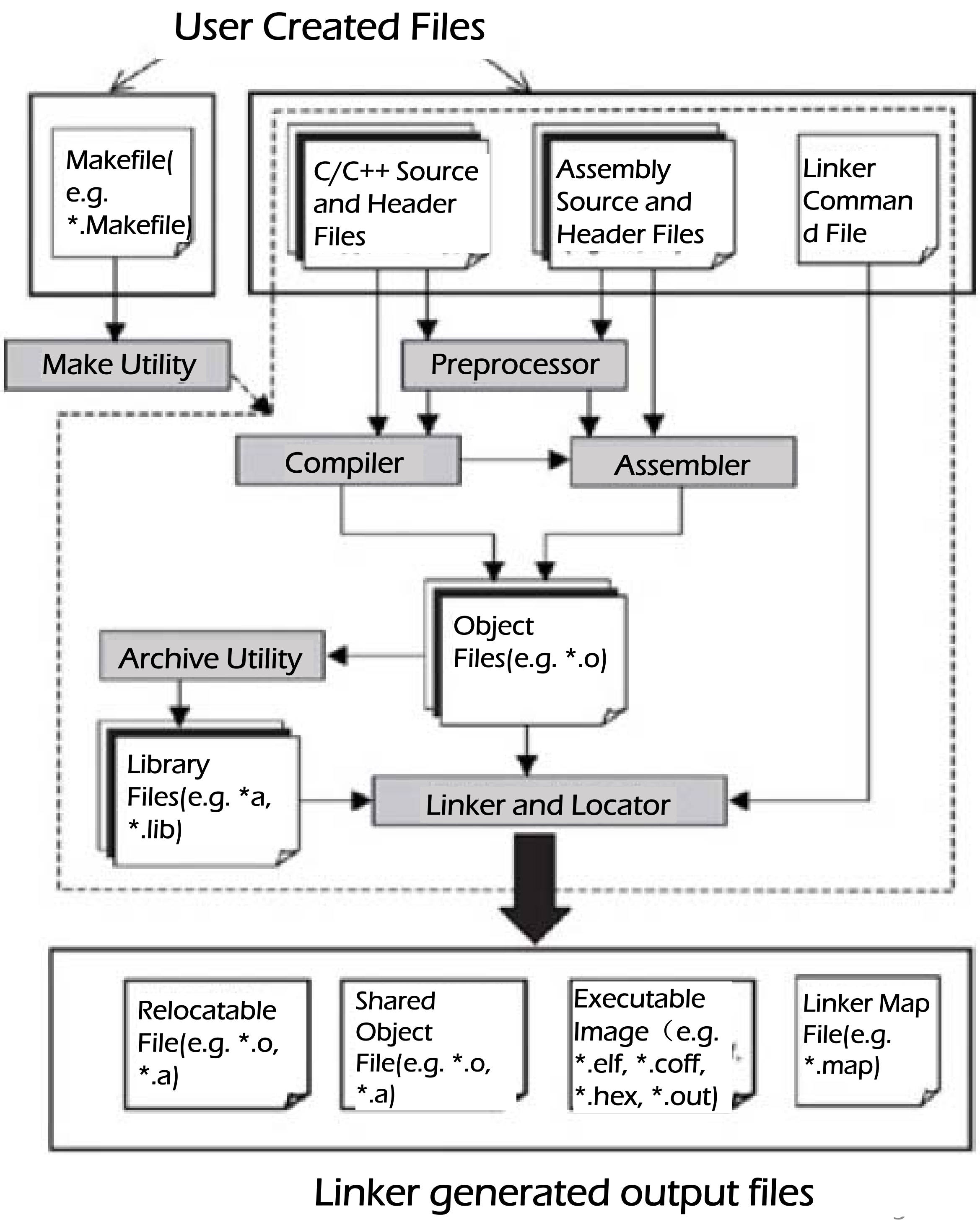
1. 库文件



- 函数库是供开发人员共享使用的函数的集合
- Linux 操作系统把库文件存放在 /lib 和 /usr/lib 目录下
- 库的文件名以 lib 开头

库文件

- 通过预处理、编译、汇编过程生成的多个目标文件
- 通过归档实用工具生成库文件
- 编码时，要想使用库提供的函数，应相关头文件包含进来



User Created Files

Makefile(e.g. *.Makefile)

C/C++ Source and Header Files

Assembly Source and Header Files

Linker Command File

Make Utility

Preprocessor

Compiler

Assembler

Object Files(e.g. *.o)

Archive Utility

Library Files(e.g. *a, *.lib)

Linker and Locator

Relocatable File(e.g. *.o, *.a)

Shared Object File(e.g. *.o, *.a)

Executable Image（e.g. *.elf, *.coff, *.hex, *.out)

Linker Map File(e.g. *.map)

Linker generated output files

1. 库文件

① 静态库

② 动态库（共享库）

- 静态库的后缀使用 .a 扩展名
- 动态库的后缀使用 .so 或 .sa 扩展名
  - 例如 libm.a, libm.so

```
[tes@apollo /guest]$ gcc -o fred fred.c /usr/lib/libm.a
[tes@apollo /guest]$ gcc -o fred fred.c -lm
[tes@apollo /guest]$ gcc -o fred -static fred.c -lm
```

1.1 静态库举例

File:fred.c

```c
#include <stdio.h>

void fred(int arg)
{
    printf("fred: you passed %d\n", arg);
}
```

File:bill.c

```c
#include <stdio.h>

void bill(char* arg)
{
    printf("bill: you passed %s\n", arg);
}
```

1.1 静态库举例

```
[test@apollo guest]$ gcc -c fred.c bill.c
[test@apollo guest]$ ls *.o
bill.o  fred.o
[test@apollo include]$
```

```
File:lib.h
/*
   This is lib.h. It declares the functions
   fred and bill for users
*/

void bill(char *);
void fred(int);
```

1.1 静态库举例

```
File:main.c
#include "lib.h"

int main()
{
   bill("Hello World");
   exit(0);
}
```

```
[test@apollo guest]$ gcc -c main.c
[test@apollo guest]$ gcc -o main main.o bill.o
[test@apollo guest]$ ./main
bill: you passed Hello World
[test@apollo guest]$
```

1.1 静态库举例

- 利用 ar 工具生成静态库
  - ar 是创建、修改、解压归档归档文件的工具
- 利用 ranlib 给归档文件创建索引

```
[test@apollo guest]$ ar crv libfoo.a bill.o fred.o
[test@apollo guest]$ ranlib libfoo.a

[test@apollo guest]$ gcc -o main main.o libfoo.a
[test@apollo guest]$ ./main
bill: you passed Hello World
```

1 .2 动态库举例

- 静态库的问题： 当多个程序同时使用静态的库的时候每个程序都会占有同样的静态库空间，导致浪费内存空间
- Linux 操作系统的 C 共享库存放在 /lib/libc.so.N 目录下。（N 表示主版本号）
- 当程序开始执行时，与动态库建立动态链接

```
[test@apollo guest]$ gcc –c fPIC bill.c fred.c
[test@apollo guest]$
 gcc –shared –o libfoo.so bill.o fred.o

[test@apollo guest]$ gcc main.c –o main –L ./ -lfoo
[test@apollo guest]$ ./main
./main: error while loading shared libraries: libfoo.so:
cannot open shared object file: No such file or directory
```

–f 后面跟一些编译选项，PIC (Position Independent Code)是其中一种，表示生成位置无关代码。

1 .2 动态库举例

- 使用 ldd 命令可以得知可执行文件与哪些共享库连接，每个共享库都在什么路径下，进程加载地址空间

```
[test@apollo guest]$ ldd main
 linux-gate.so.1 =>  (0xb7749000)
 libfoo.so => not found
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb758a000)
 /lib/ld-linux.so.2 (0xb774a000)
[test@apollo guest]$ cp libfoo.so /usr/lib
[test@apollo guest]$./main
bill: you passed Hello World
```

- 把创建的 libfoo.so 文件复制到 /usr/lib 目录下即可

Make Utility

什么是 Make Utility?

一般项目由几百或几千个源文件组成，对源程序部分文件进行修改后，重新编译很费时

1. 编译配置
2. 编译时编译规则
3. 编译后对生成文件的管理和配置

- 用什么配置?
  - 通过编辑 Makefile

- Make Utility 是一个命令解释工具，它解释 Makefile 中的指令（规则）
- Make Utility 可以只针对被修改的源文件进行重新编译

举例

```
/* File name : main.c */
#include “a.h”
…
```

```
/* File name : 2.c */
#include “a.h”
#include “b.h”
…
```

```
/* File name : 3.c */
#include “b.h”
#include “c.h”
…
```

如修改了 c.h 文件，因 main.c 和 2.c 文件没有依赖关系，无需重新编译 main.c 和 2.c 文件

Make Utility 命令选项

- -k : Keep-going, continue as much as possible after an error
- -n : Build-test, print the commands that would be executed, but do not execute.
- -f <filename> : use filename as a Makefile
- Others, you can use “$man make” command

- What is the Makefile?

什么是 Makefile?

- 一个工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中，
- Makefile
- 定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至进行更复杂的功能操作，因为 Makefile 就像一个 Shell Script 一样，其中也可以执行操作系统的命令。
- 定义了源文件编译过程中，编译后，以及生成文件的存放规则等
- Makefile 文件一般存放在源文件的根目录

Makefile Format

1. 指定了依赖关系

- 指定生成的目标文件与源文件的依赖关系

2. 指定了生成规则

- 指定从源文件生成目标文件的生成规则
- 以<Tab> 开始

Makefile 举例

```
/* Filename : Makefile1 */
myapp: main.o 2.o 3.o
       gcc –o myapp main.o 2.o 3.o

main.o: main.c a.h
       gcc –c main.c

2.o: 2.c a.h b.h
       gcc –c 2.c

3.o: 3.c b.h c.h
       gcc –c 3.c
```

- 最终可执行文件 myapp 与 main.o, 2.o, 3.o 文件有依赖性
- Main.o 目标文件 与 main.c, a.h 文件有依赖关系
- 2.O 目标文件 与 2.c, a.h, b.h 文件有依赖关系
- 3.O 目标文件 与 3.c, b.h, c.h 文件有依赖关系
- 目标文件 2.o 文件 创建规则是 gcc – c 2.c

举例

```
/* Filename : b.h */
#include <stdio.h>

void function_two();
```

```
/*Filename:  2.c */
#include “a.h”
#include “b.h”

void function_two() {
             printf(“ This is 2\n”);
}
```

举例

```
/* Filename : c.h */
#include <stdio.h>

void function_three();
```

```
/ * Filename : 3.c */
#include “a.h”
#include “b.h”

void function_three() {
             printf(“ This is 3\n”);
}
```

举例

```
/* Filename : a.h */

#include <stdio.h>

void function_two();
void function_three();
```

```
/* Filename : main.c */
#include “a.h”

extern void function_two();
extern void function_three();

int main()
{
    function_two();
    function_three();
    return 0;
}
```

举例

- 运行

```
$ make -f Makefile1
    gcc -c main.c
    gcc -c 2.c
    gcc -c 3.c
    gcc -o myapp main.o 2.o 3.o
$
```

举例

- 修改 b.h 文件以后， 重新运行 make

```
$ make -f Makefile1
gcc -c 2.c
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
$
```

举例

- 把 object 文件删除后，重新执行 make

```
$ rm 2.o
$ make -f Makefile1
gcc -c 2.c
gcc -o myapp main.o 2.o 3.o
$
```

Makefile 宏指令

- 更一般的形式
- 可以指定编译选项

```
Define:
        MACRONAME = value
Usage:
        $(MACRONAME) or ${MACRONAME}
```

Makefile 宏指令

```
/* File name : Makefile2 */

all: myapp

# Which compiler
CC = gcc

# Where are include files kept
INCLUDE = .

# Options for development
CFLAGS = -g -Wall -ansi

# Options for release
# CFLAGS = -O -Wall -ansi
```

-g : 可调试模式

-O ： 对代码进行基本优化

-Wall ： 设置警告

-ansi： C 标准编译

-c ： 只编译，不连接

Makefile 宏指令

```
myapp: main.o 2.o 3.o
  $(CC) –o myapp main.o 2.o 3.o

main.o: main.c a.h
  $(CC)  –I$(INCLUDE)  $(CFLAGS) –c main.c

2.o: 2.c a.h b.h
  $(CC)  –I$(INCLUDE)  $(CFLAGS) –c 2.c

3.o: 3.c b.h c.h
   $(CC)  –I$(INCLUDE)  $(CFLAGS) –c 3.c
```

Makefile 宏指令

```
$ rm *.o myapp
$ make -f Makefile2
    gcc -I. -g -Wall -ansi -c main.c
    gcc -I. -g -Wall -ansi -c 2.c
    gcc -I. -g -Wall -ansi -c 3.c
    gcc -o myapp main.o 2.o 3.o
$
```

Others

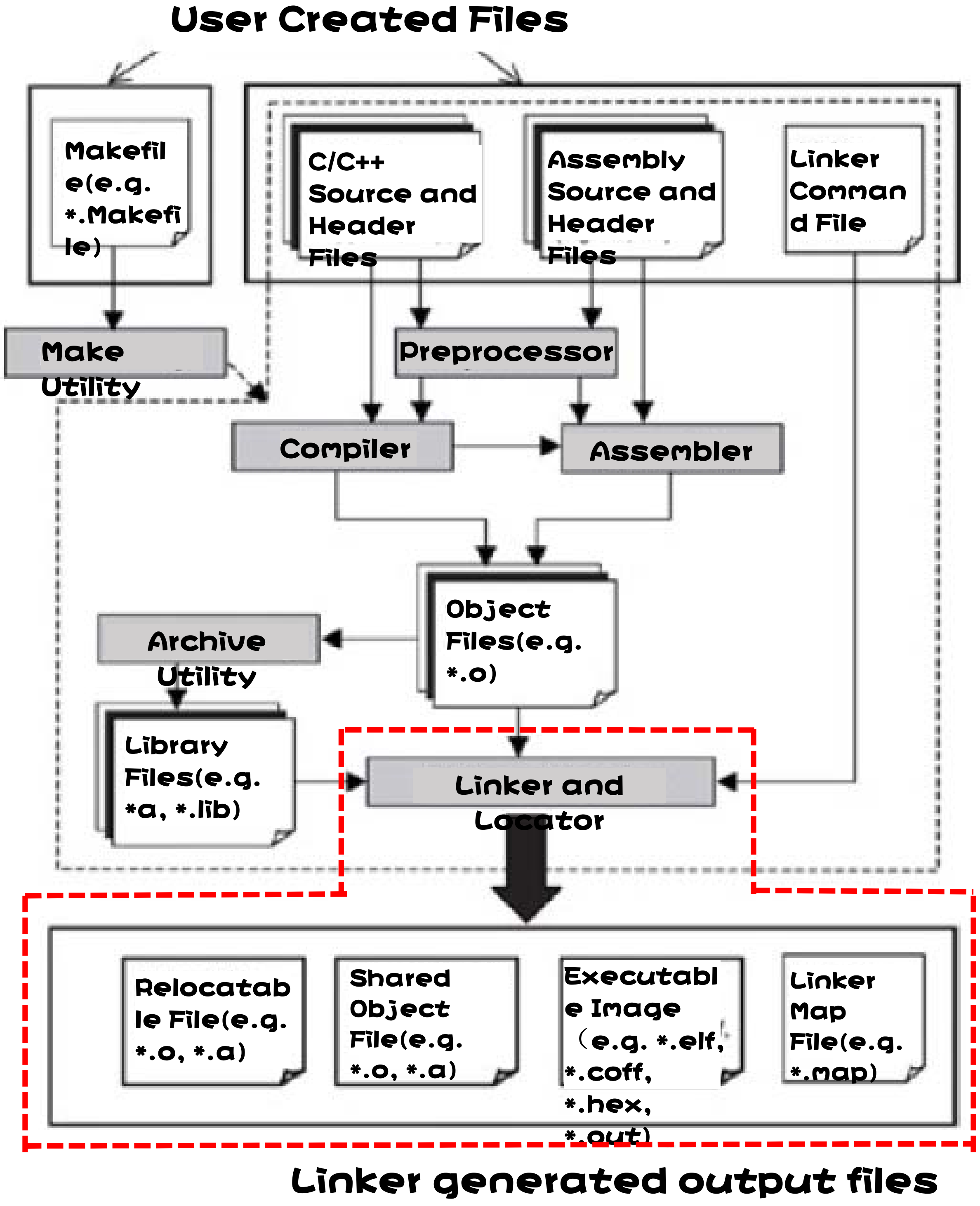
- 通过编译生成的目标文件、库文件、可执行文件等需要方便去管理，
  - 如库文件存放在 /usr/lib 或 /lib 目录下
  - 可执行文件存放在 /bin 目录下
  - 临时生成的 object 文件管理

- $make config
  - 配置编译环境
- $make clean
  - 删除临时生成的目标文件
- $make Install
  - 把可执行文件移动/复制到相应的目录下

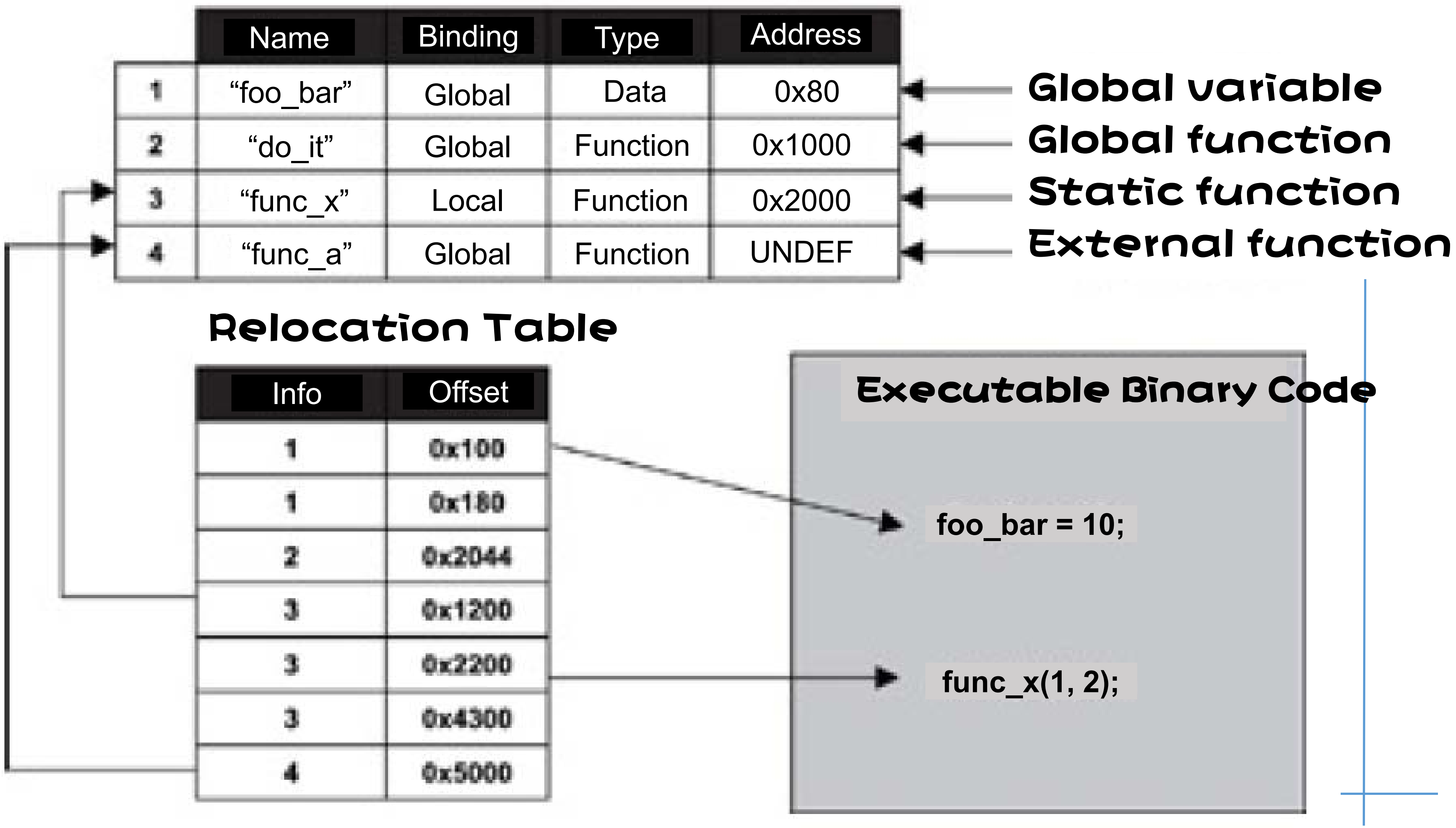
ELF File Format

Linker and the Linking Process

- The main function of the linker is to combine multiple object files into a larger **relocatable object file**, **a shared object file**, or **final executable image**.
- The compiler creates **a symbol table** containing the symbol name to address mapping as part of the object file in produces
- Linking process performed by the linker involves **symbol resolution** and **symbol relocation**


User Created Files
Makefile(e.g. *.Makefile)
C/C++ Source and Header Files
Assembly Source and Header Files
Linker Command File
Make Utility
Preprocessor
Compiler
Assembler
Object Files(e.g. *.o)
Archive Utility
Library Files(e.g. *a, *.lib)
Linker and Locator
Relocatable File(e.g. *.o, *.a)
Shared Object File(e.g. *.o, *.a)
Executable Image （e.g. *.elf, *.coff, *.hex, *.out)
Linker Map File(e.g. *.map)
Linker generated output files

Relationship between the Symbol Table and Relocation Table



Symbol Table

	Name	Binding	Type	Address
1	"foo_bar"	Global	Data	0x80
2	"do_it"	Global	Function	0x1000
3	"func_x"	Local	Function	0x2000
4	"func_a"	Global	Function	UNDEF

Relocation Table

Info	Offset
1	0x100
1	0x180
2	0x2044
3	0x1200
3	0x2200
3	0x4300
4	0x5000

Executable Binary Code

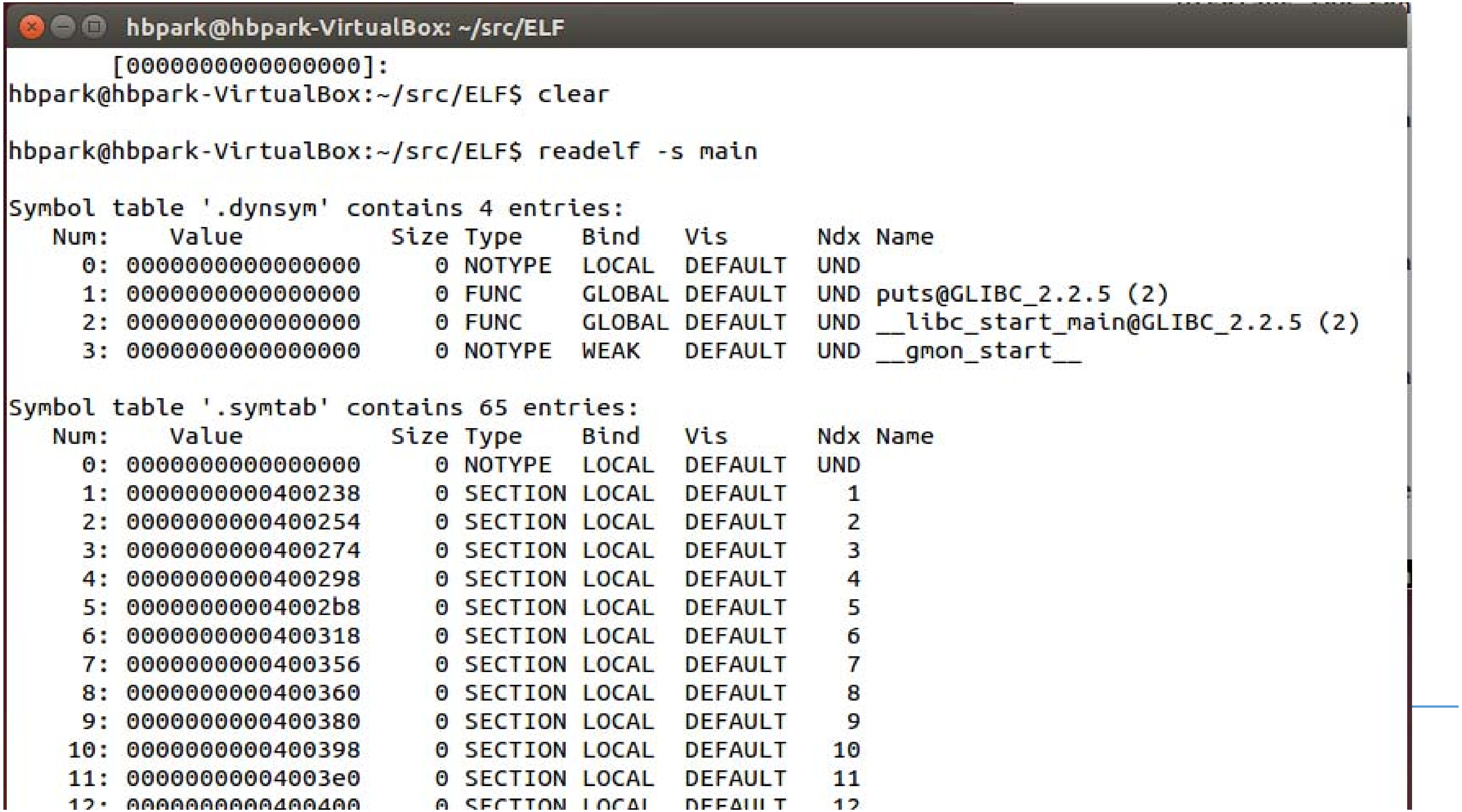
```
foo_bar = 10;

func_x(1, 2);
```

Symbol Table and Relocation Table

Use the following command confirm symbol table of an object file

$readelf –s "objectfile_name"



```
      [0000000000000000]:
hbpark@hbpark-VirtualBox:~/src/ELF$ clear

hbpark@hbpark-VirtualBox:~/src/ELF$ readelf -s main

Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 65 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400254     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400274     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400298     0 SECTION LOCAL  DEFAULT    4
     5: 00000000004002b8     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400318     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000400356     0 SECTION LOCAL  DEFAULT    7
     8: 0000000000400360     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000400380     0 SECTION LOCAL  DEFAULT    9
    10: 0000000000400398     0 SECTION LOCAL  DEFAULT   10
    11: 00000000004003e0     0 SECTION LOCAL  DEFAULT   11
    12: 0000000000400400     0 SECTION LOCAL  DEFAULT   12
```

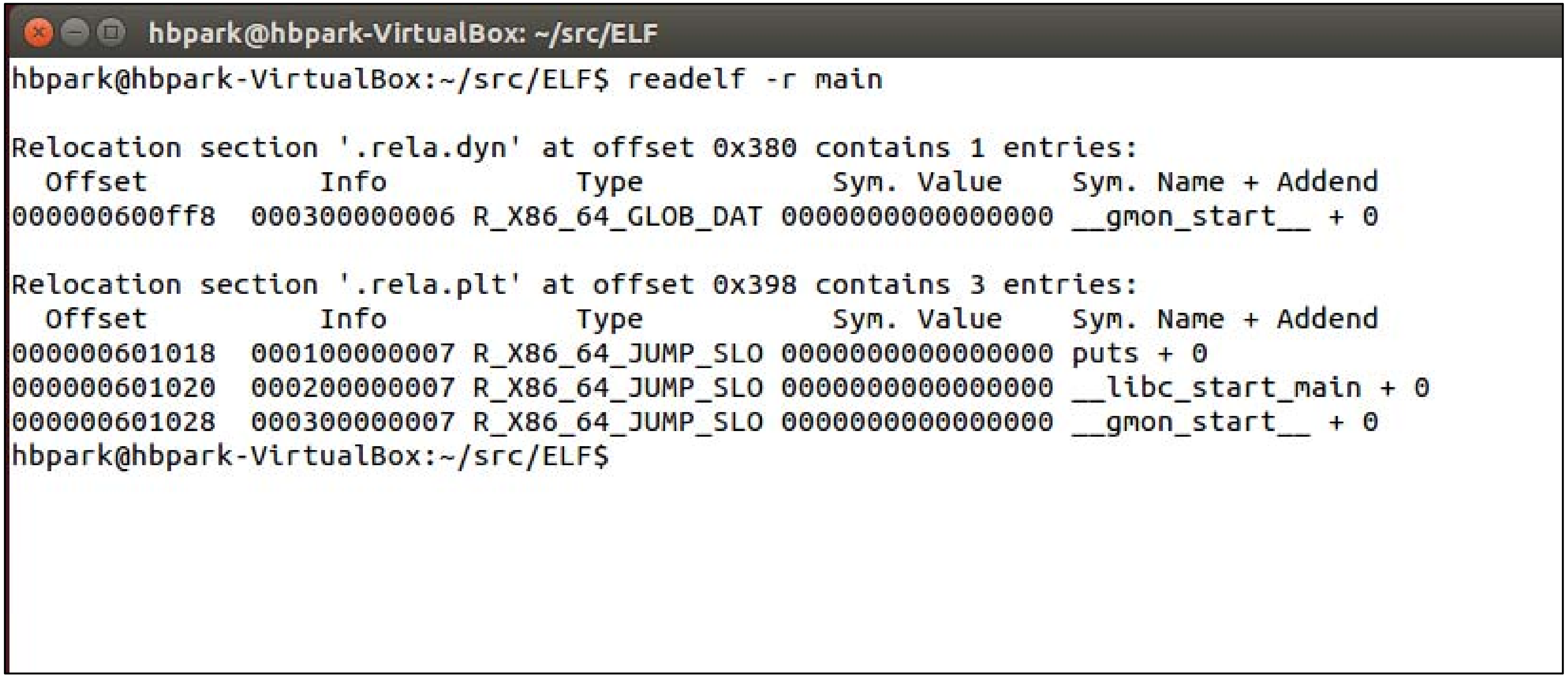
Symbol Table and Relocation Table

- Ndx 列是每个符号所在的 Section 编号
- Value 列是每个符号所代表的地址，在目标文件中，符号地址都是绝对地址
- Bind 列是类型（Type）GLOBAL 或 LOCAL

Symbol Table and Relocation Table

Use the following command confirm relocation table of an object file

$readelf –r “objectfile_name”



```
hbpark@hbpark-VirtualBox:~/src/ELF$ readelf -r main

Relocation section '.rela.dyn' at offset 0x380 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600ff8  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x398 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main + 0
000000601028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 __gmon_start__ + 0
hbpark@hbpark-VirtualBox:~/src/ELF$
```

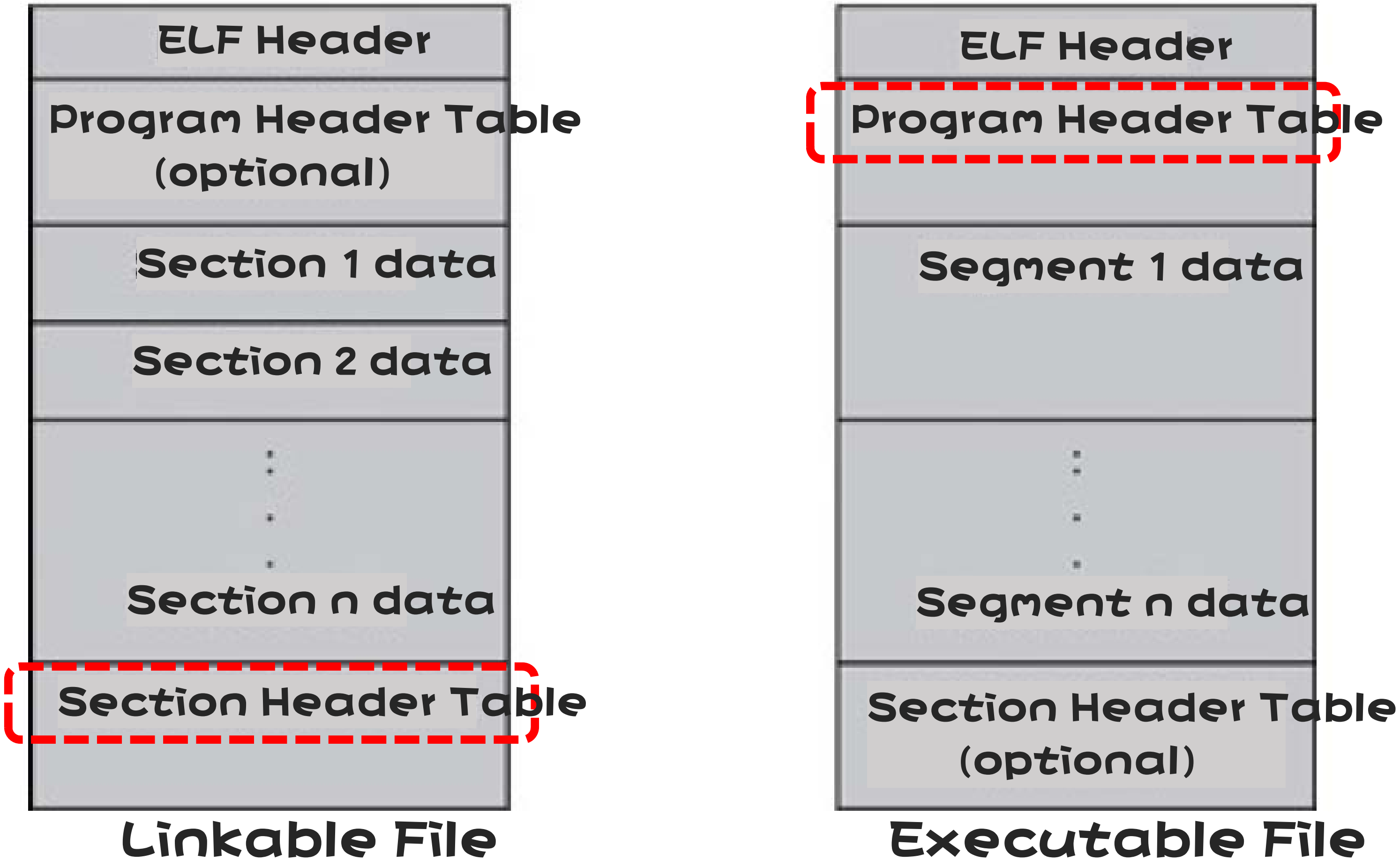
Executable and Linking Format

- There are two types of common object file formats(COFF)
  1. Portable Executable(PE) Format
  2. Executable and Linking Format(ELF)

- Typically an object file contains
  1. General information about the object file, such as file size, binary code and data size, and source file name from which it was created,
  2. Machine-architecture-specific binary instructions and data,
  3. Symbol table and symbol relocation table
  4. And, Debug information

ELF Format

The ELF format has two different interpretations: **Linkable File** and **Executable File**



ELF Header

Program Header Table (optional)

Section 1 data

Section 2 data

Section n data

Section Header Table

Linkable File

ELF Header

Program Header Table

Segment 1 data

Segment n data

Section Header Table (optional)

Executable File

ELF Format

- **A section header table** is an array of section header structures describing the **sections** of an object file
- **A program header table** is an array of program header structures describing a **loadable segment** of an image that allows the loader to prepare the image for execution

Section Header

```
typedef struct {
        Elf32_Word sh_name;
        Elf32_Word sh_type;
        Elf32_Word sh_flags;
        Elf32_Addr sh_addr;
        Elf32_Off sh_offset;
        Elf32_Word sh_size;
        Elf32_Word sh_link;
        Elf32_Word sh_info;
        Elf32_Word
        sh_addralign;
        Elf32_Word
        sh_entsize;
} Elf32_Shdr;
```

Program Header

```
typedef struct {
        Elf32_Word p_type;
        Elf32_Off p_offset;
        Elf32_Addr p_vaddr;
        Elf32_Addr p_paddr;
        Elf32_Word p_filesz;
        Elf32_Word p_memsz;
        Elf32_Word p_flags;
        Elf32_Word p_align;
} Elf32_Phdr;
```

Section Types(sh_type field)

Name	Desc.
NULL	Inactive header without a section
PROGBITS	Code or Initialized data
SYMTAB	Symbol table for static linking
STRTAB	String table
RELA/REL	Relocation entries
HASH	Run-time symbol hash table
DYNAMIC	Information used for dynamic linking
NOBITS	Uninitialized data
DYNSYM	Symbol table for dynamic linking

Section Attributes(sh_flags field)

Name	Desc.
WRITE	Section contains writable data
ALLOC	Section contains allocated data
EXECINSTR	Section contains executable instructions

Pre-defined Sections

Common system-created default sections with predefined names for the PROBITS are .text, .sdata, .data, .sbss, and .bss.

1. .text section: contains program code and constant data, has read only. EXECINSTR attribute.
2. .sdata and .data sections contain initialized data, have WRITE attribute.
3. .sbss and .bss sections contain uninitialized data, have WRITE and ALLOC attribute.

Pre-defined Sections

Other common system-defined section are .symtab, .shstrab, .strtab, .relaname etc.

1. .symtab : contains the symbol table
2. .strtab : contains the string table for the program symbols
3. .shstrab : contains the string table for the section names
4. .relaname : contains the relocation information for the section named name

Each entry in the symbol table(SYMTAB) contains a reference to the string table(STRTAB) where the character representation of the name of stored

User-defined Section

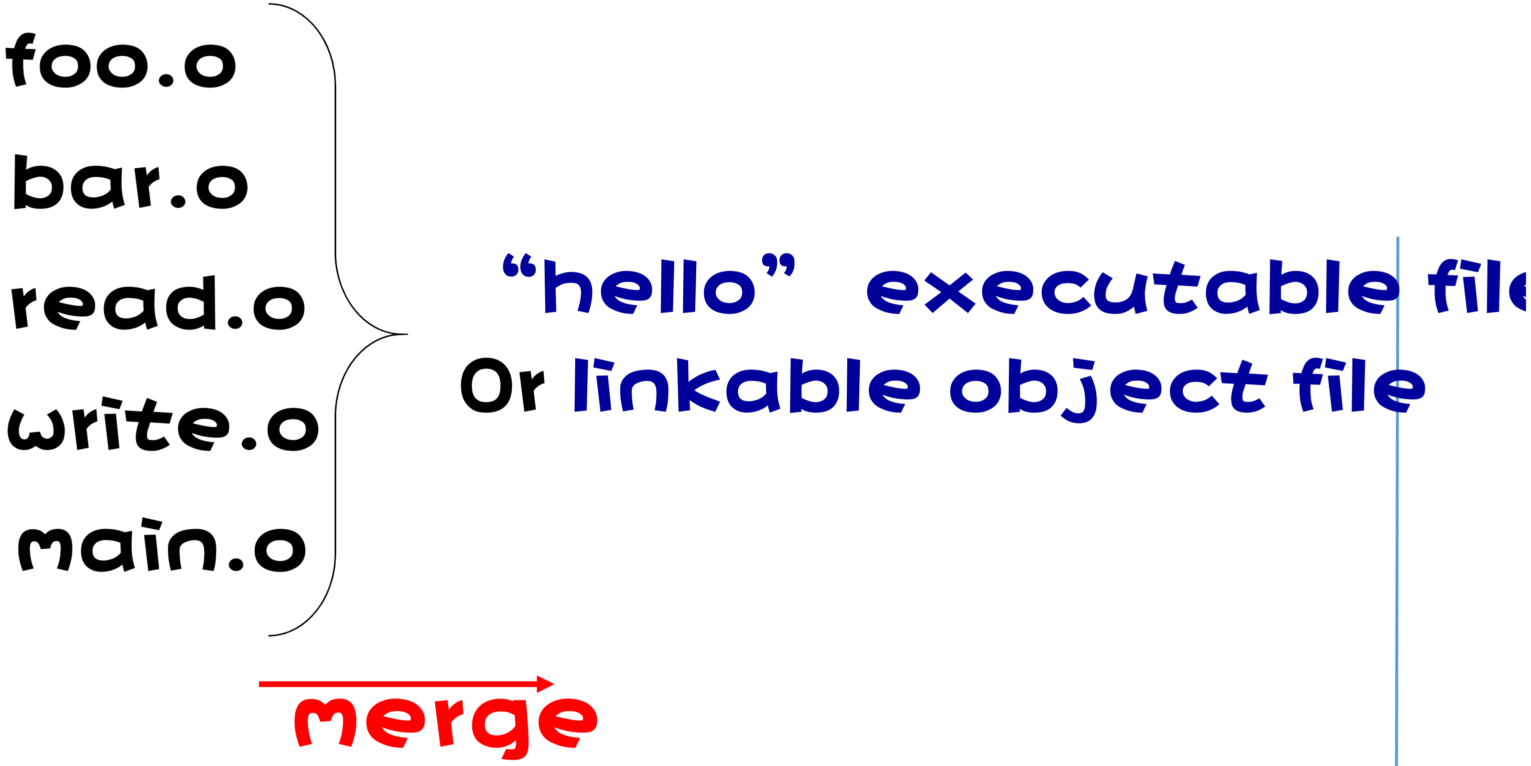
The developer can define custom sections by invoking the linker command .section. For example

- .section my_section

the linker creates a new section called my_section.

1. sh_addr is the address where the program section should reside in the target memory
2. p_paddr is the address where the program segment should reside in the target memory
3. The sh_addr and the p_paddr fields refer to the load addresses.

Example



foo.o

bar.o

read.o

write.o

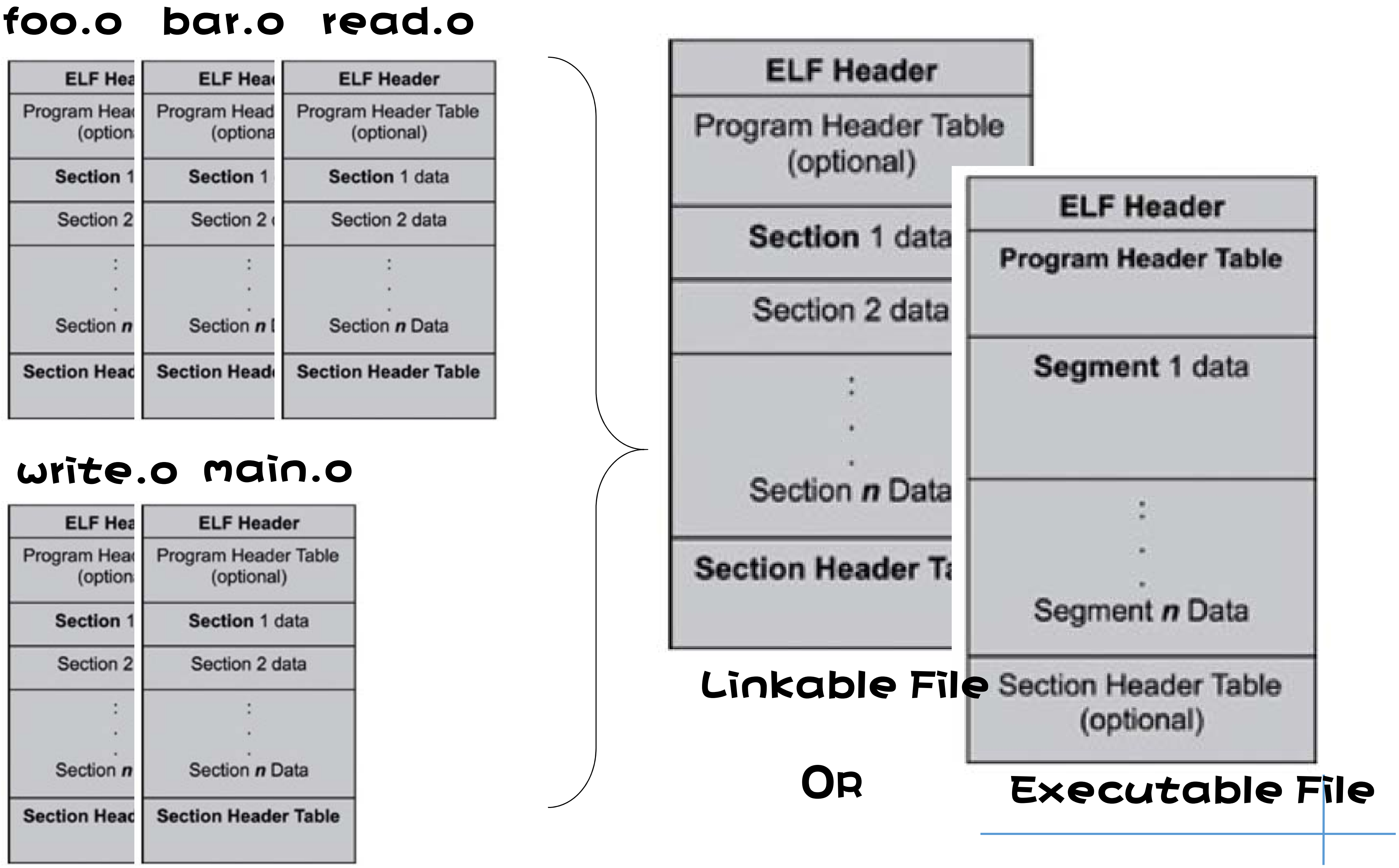
main.o

merge

"hello" executable file

Or linkable object file

Example



foo.o bar.o read.o

write.o main.o

Linkable File

Or

Executable File

Q&A