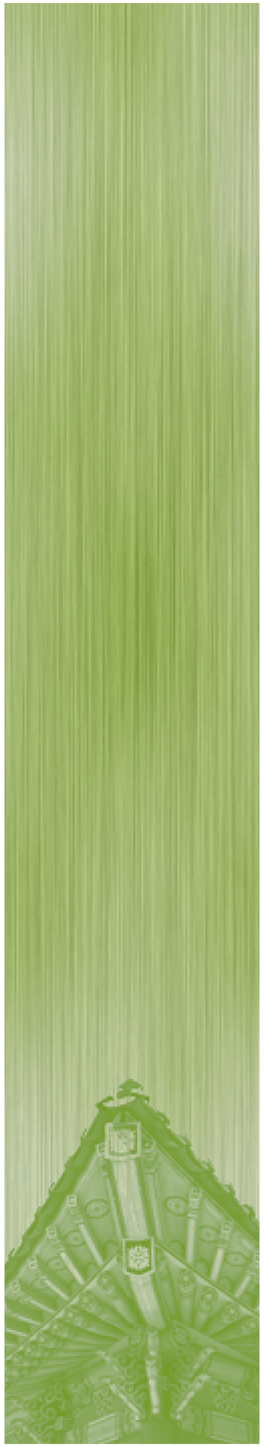


[Chap.7] Linking

Young Ik Eom (yieom@skku.edu, 031-290-7120)
Distributing Computing Laboratory
Sungkyunkwan University
http://dclab.skku.ac.kr

Contents

- **Compiler drivers**
- **Relocatable object files**
- **Static linking**
  - Symbols and symbol tables
  - Symbol resolution
  - Relocation
- **Executable object files**
- **Dynamic linking**
  - Shared libraries
  - Load-time linking and run-time linking
  - Position independent code
- **Tools on object files**

Compiler Drivers

- **Example C programs (C storage classes review)**

```c
/* sub1.c */

extern int x;
void f2(int, double);

int f1()
{
   int a;
   double d = 1.2;
   ...
   a = b + 10;
   ...
   f2(a, d);
   ...
}
```

```c
/* sub2.c */

int x = 0;
static int y = 0;

void f2(int i, double s)
{
  ...
}
```

Compiler Drivers

- Example C programs

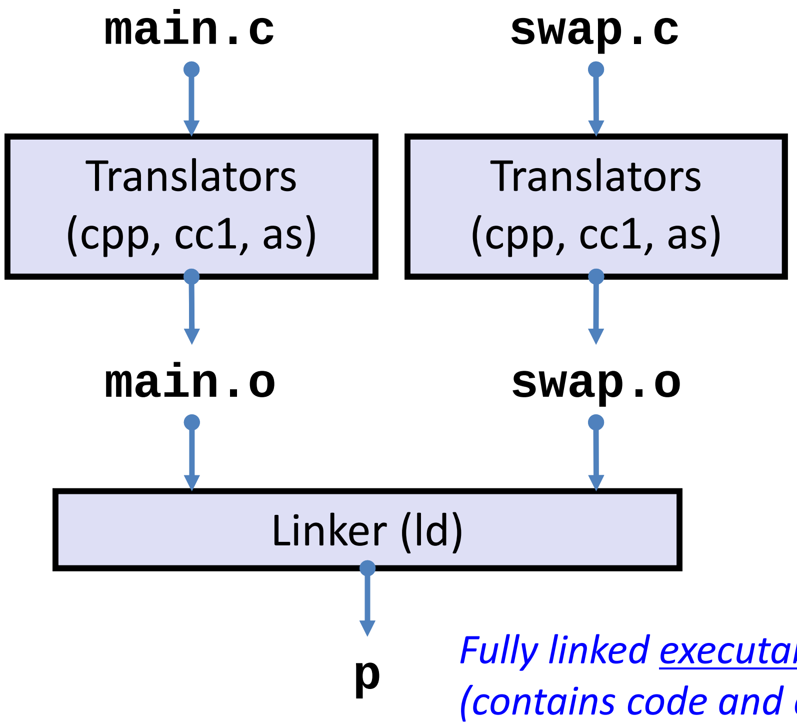
```c
/* main.c */

void swap();
int buf[2] = {1, 2};

int main()
{
   swap();
   return 0;
}
```

```c
/* swap.c */

extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
   int temp;

   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

Compiler Drivers

Compilation and linking

```
unix> gcc -Og -o p main.c swap.c
unix> ./p
```

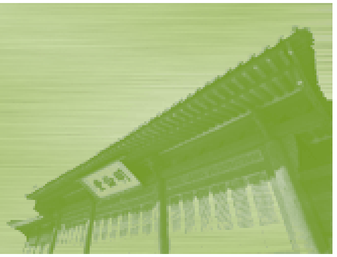


Source files

Separately compiled relocatable object files

Fully linked __executable__ object file (contains code and data for all functions defined in `main.c` and `swap.c`)

Compiler Drivers



Why linkers?

- Modularity
  - Program can be written as a collection of smaller source files, rather than one monolithic mass
  - Can build libraries of common functions (more on this later)
    - ✓ Eg) math library, standard C library, etc
- Efficiency
  - Time (separate compilation)
    - ✓ Change one source file, compile, and then relink
      - No need to recompile other source files
  - Space (libraries)
    - ✓ Common functions can be aggregated into a single file
    - ✓ Executable files and running memory images contain only code for the functions they actually use

Relocatable Object Files

- Object file formats
  - **a.out** format
    - The 1st Unix system from Bell Labs
  - COFF (Common Object File Format)
    - Early versions of Unix System V
  - **ELF** (Executable and Linkable Format)
    - Modern Unix systems including Linux
  - PE (Portable Executable) format
    - MS Windows

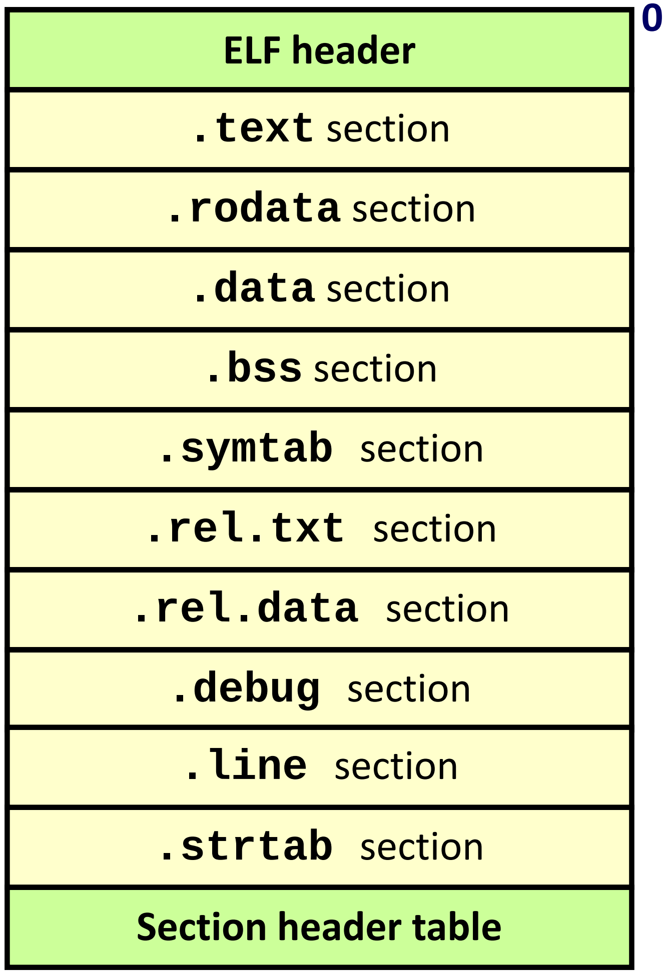
Relocatable Object Files

- **ELF Object files**
  - **Relocatable object files** (**.o** file)
    - Contains code and data in a form that can be combined with other relocatable object files to form an executable object file
      - ✓ Each **.o** file is produced from exactly one source (**.c**) file
  - **Executable object files**
    - Contains code and data in a form that can be copied directly into memory and then executed
  - **Shared object files** (**.so** file)
    - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time
    - Called Dynamic Link Libraries (DLLs) by MS Windows

Relocatable Object Files

Relocatable object files

- ELF header
  - Describes overall format of the file
    - ✓ Word size, byte ordering
    - ✓ ELF header size, object file type, machine type, file offset of the section header table, size and number of entries in the section header table
- Section header table
  - Locations and sizes of the various sections



0

ELF header

.text section

.rodata section

.data section

.bss section

.symtab section

.rel.txt section

.rel.data section

.debug section

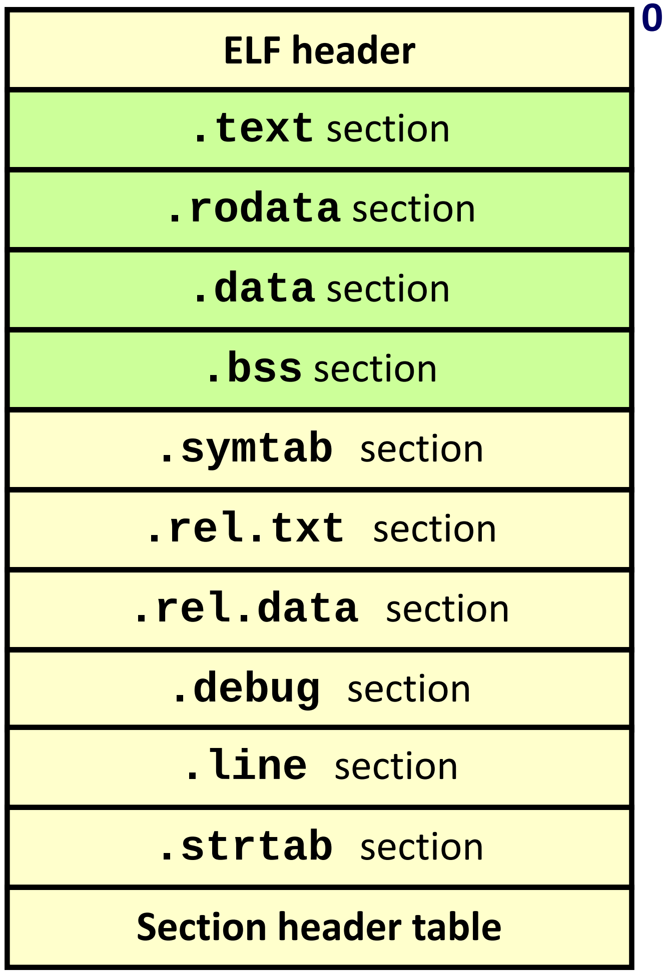
.line section

.strtab section

Section header table

Relocatable Object Files

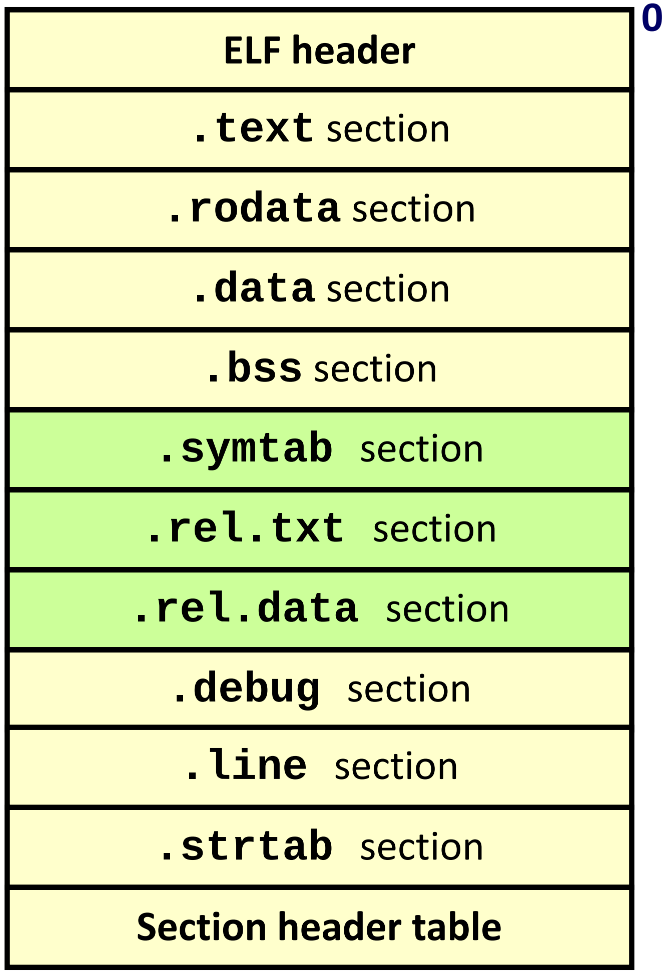
- Relocatable object files
  - **.text**
    - Machine code
  - **.rodata**
    - Read-only data such as jump tables and format strings
  - **.data**
    - Initialized global variables
  - **.bss**
    - Uninitialized global variables
    - Just a place holder
    - "Block Storage Start" "Better Save Space"



Relocatable Object Files

Relocatable object files

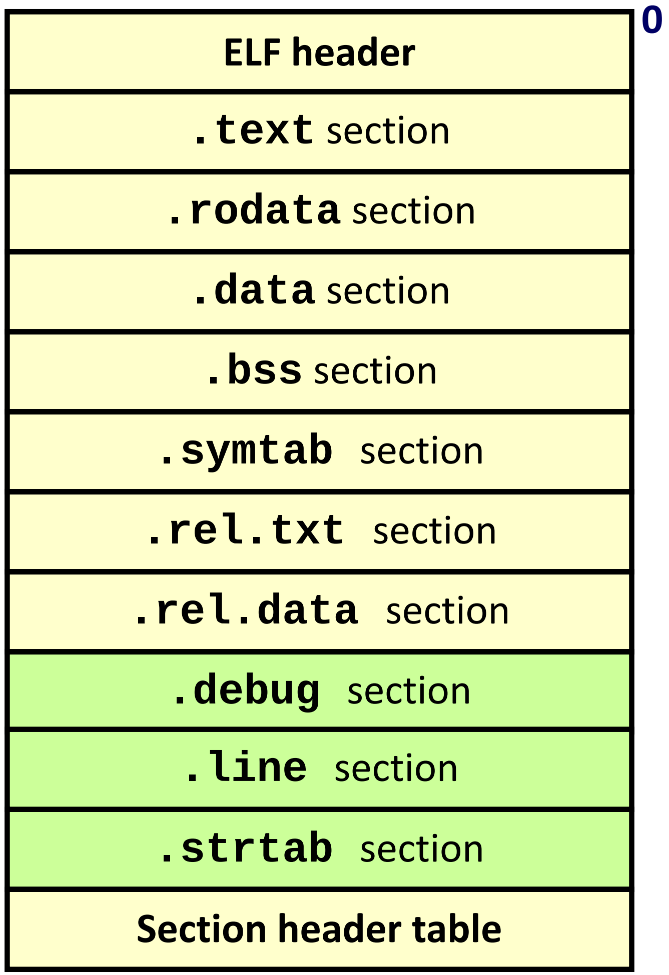
- **.symtab**
  - Symbol table
  - Information on the functions and global variables that are defined or referenced in the program
- **.rel.text**
  - Relocation information for **.text** section
- **.rel.data**
  - Relocation information for **.data** section



Relocatable Object Files

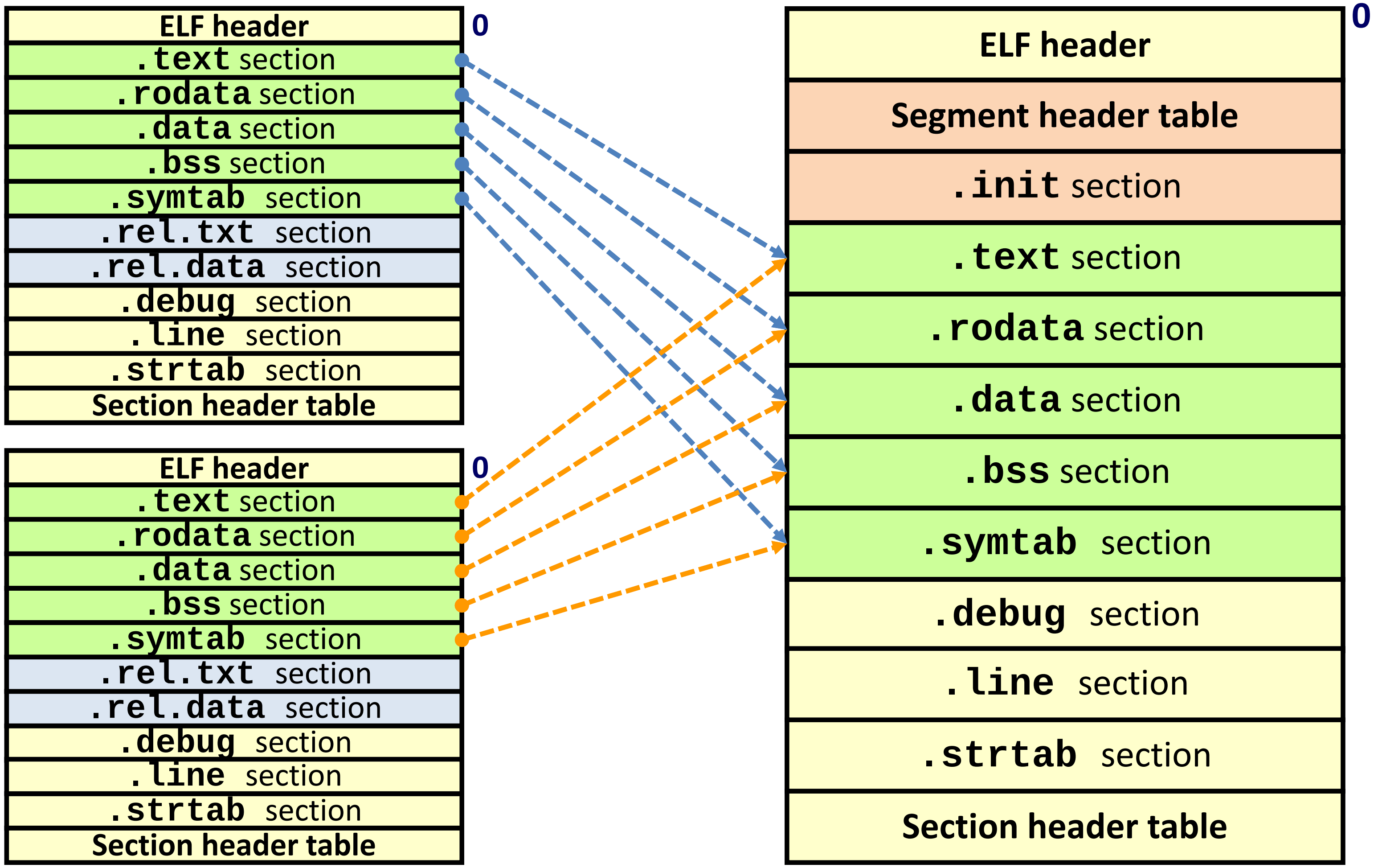
Relocatable object files

- **.debug**
  - Debugging symbol table
  - Created only when the compiler driver is invoked with **–g** option
- **.line**
  - Line number mapping between C source programs and machine code instructions in **.text**
  - Created only when the compiler driver is invoked with **–g** option
- **.strtab**
  - String table for **.symtab**, **.debug**, and section names



Relocatable Object Files

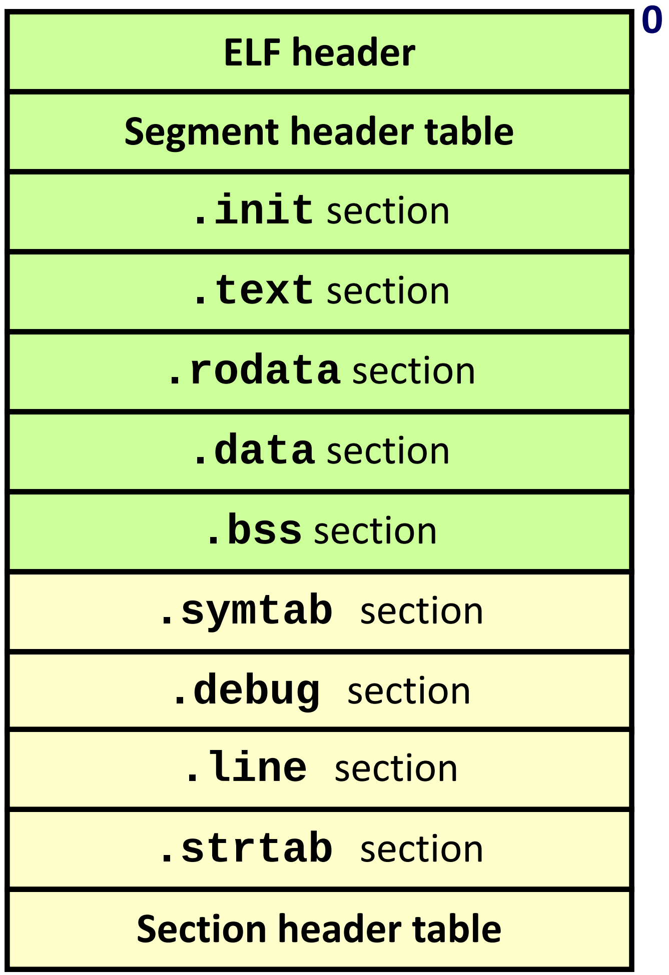
- Relocatable object files vs executable object files



Executable Object Files

Executable object files

- ELF header
  - Describes overall format of the file
  - Includes entry point
- Segment header table
  - Mapping of the contiguous chunks of the executable file to contiguous M segments
- **.init**
  - Defines **_init** function, which will be called by the pgm's initialization code
- **.text**, **.rodata**, **.data**
  - Similar to those in relocatable object files
  - Relocated to their eventual M addresses
- No **.rel** sections



0

ELF header

Segment header table

`.init` section

`.text` section

`.rodata` section

`.data` section

`.bss` section

`.symtab` section

`.debug` section

`.line` section

`.strtab` section

Section header table

Static Linking

- **Static linker (ld in Unix)**
  - Takes as input a collection of relocatable object files and command line arguments and
    generates as output a fully linked executable object file
  - Two main tasks
    - **Symbol resolution**
      - ✓ Associate each symbol reference in the object file with exactly one symbol definition
    - **Relocation**
      - ✓ Associates a memory location with each symbol definition, and modifies all the references to that symbol so that they point to the memory location

Static Linking

Symbols and symbol tables

- Each relocatable object module **m** has a symbol table that contains information about the symbols that are defined or referenced by **m**

- Contained in **.symtab** section
- Built by assemblers

Static Linking

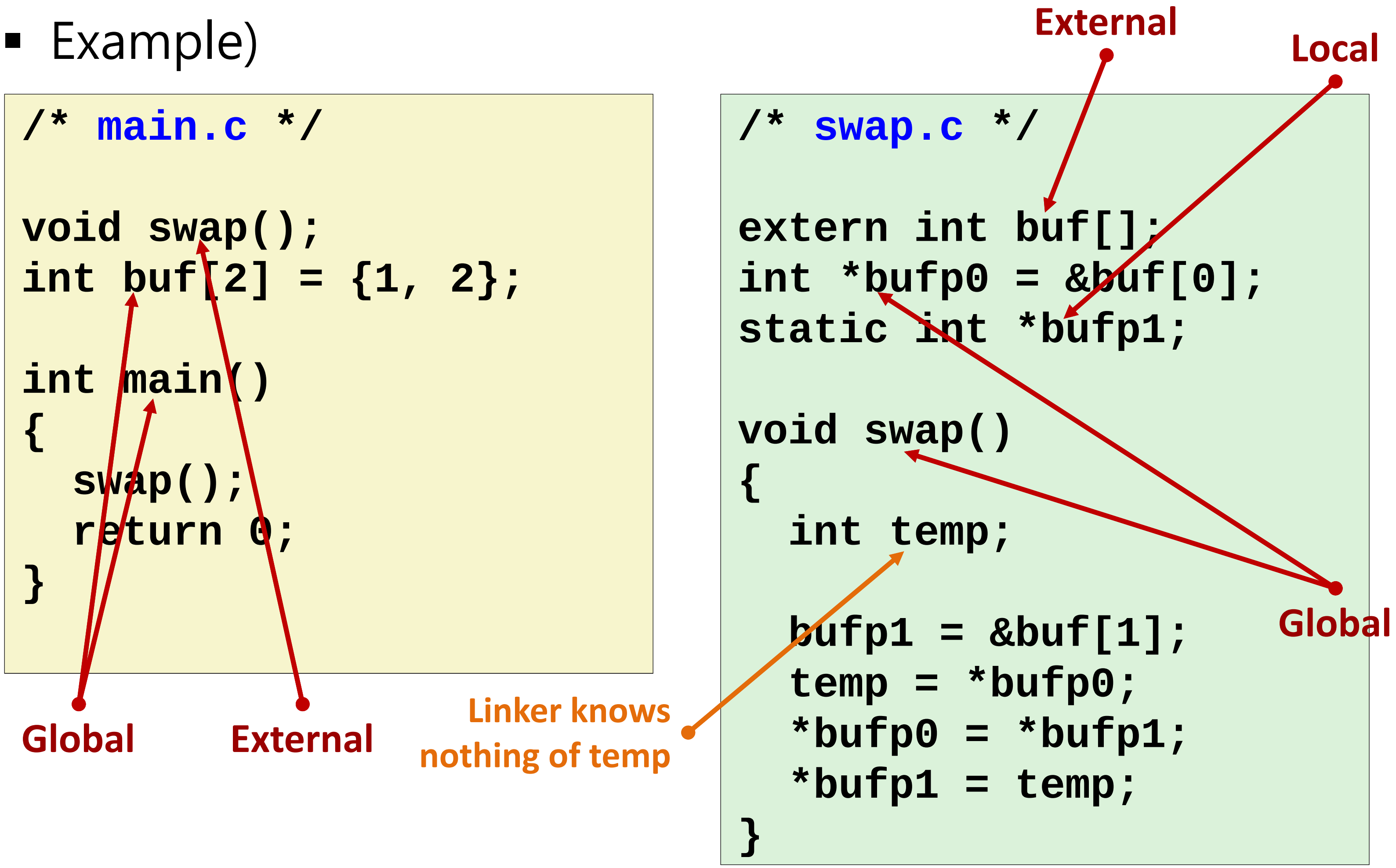
Symbols and symbol tables

- 3 kinds of symbols
  - **Global** symbols
    - ✓ Symbols that are defined by module **m** and that can be referenced by other modules
    - ✓ **Non-static** C functions and global variables
  - **External** symbols
    - ✓ Global symbols that are referenced by module **m** but defined by some other module
  - **Local** symbols
    - ✓ Symbols that are defined and referenced exclusively by module **m**
    - ✓ C functions and variables defined with the **static** attribute
    - ✓ Include the name of the source file, …
    - ✓ Local variables (managed on stack) are not local linker symbols

Static Linking

Symbols and symbol tables

- Example)



```
/* main.c */

void swap();
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

```
/* swap.c */

extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Global

External

Linker knows nothing of temp

External

Local

Global

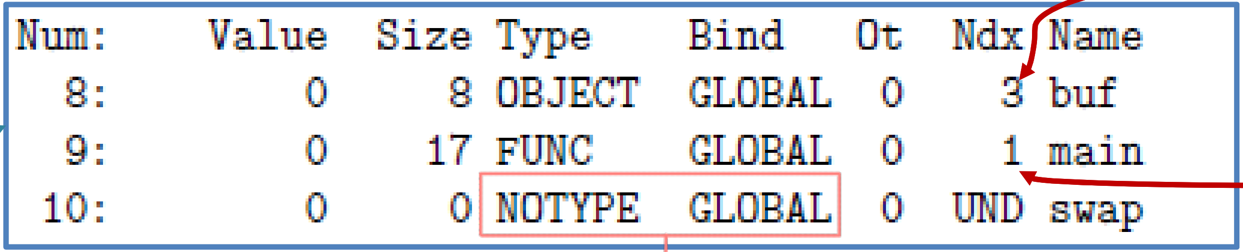
Static Linking

```
/* main.c */
void swap();
int buf[2] = {1, 2};
int main()
{
  swap();
  return 0;
}
```

```
/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;
void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Symbols and symbol tables

- Structure of symbol table: Example)
  - Symbol table of **main.o**



```
Num:    Value  Size Type    Bind   Ot  Ndx Name
  8:        0     8 OBJECT  GLOBAL  0    3 buf
  9:        0    17 FUNC    GLOBAL  0    1 main
 10:        0     0 NOTYPE  GLOBAL  0  UND swap
```

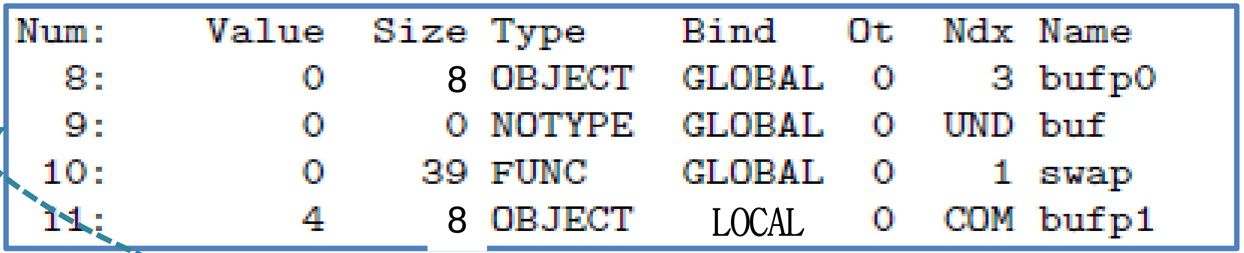
Data section

2 -.radate

Text section

external symbol

  - Symbol table of **swap.o**



```
Num:    Value  Size Type    Bind   Ot  Ndx Name
  8:        0     8 OBJECT  GLOBAL  0    3 bufp0
  9:        0     0 NOTYPE  GLOBAL  0  UND buf
 10:        0    39 FUNC    GLOBAL  0    1 swap
 11:        4     8 OBJECT   LOCAL  0  COM bufp1
```

Sample produced by **readelf** tool

Static Linking

Symbol resolution

- Linker resolves symbol references
  by associating each reference with exactly one symbol definition
  from the symbol tables of its input relocatable object files
  - Straightforward for local symbols
    (defined in the same module as the reference)
  - Trickier for global symbols
    - ✓ When the **compiler** encounters a symbol that is not defined in current module,
      it assumes that it is defined in some other module,
      generates a linker symbol table entry, and leaves it for the **linker** to handle
    - ✓ When the **linker** cannot find the definition of the symbol,
      it reports an error
    - ✓ When the **linker** finds duplicate definitions of the symbol,
      it applies its resolution rules

Static Linking

Symbol resolution

- Strong symbols and weak symbols
  - Strong symbols
    - ✓ Functions and initialized global variables
    - ✓ Eg) **buf**, **main**, **bufp0**, **swap**
  - Weak symbols
    - ✓ Uninitialized global variables
    - ✓ Eg) **bufp1**

```c
/* main.c */

void swap();
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

```c
/* swap.c */

extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Static Linking

- **Symbol resolution**
  - Symbol resolution rules for multiply defined symbols
    - **R1**) Multiple strong symbols are not allowed
    - **R2**) Given a strong symbol and multiple weak symbols,
      choose the strong symbol
    - **R3**) Given multiple weak symbols, choose any of the weak symbols

Static Linking

Symbol resolution: multiply defined symbols

- Example-1)

```
/* foo1.c */
int main()
{
 return 0;
}
```

```
/* bar1.c */
int main()
{
 return 0;
}
```

```
unix> gcc foo1.c bar1.c
```

Error!

Duplicate strong symbol

Static Linking

Symbol resolution: multiply defined symbols

- Example-2)

```
/* foo2.c */
int x = 1;
int main()
{
 return 0;
}
```

초기값 부여

⇓

strong

```
/* bar2.c */
int x = 1;
void f()
{
}
```

```
unix> gcc foo2.c bar2.c
```

Error!

Static Linking

Symbol resolution: multiply defined symbols

- Example-3)

```
/* foo3.c */
...
int x = 100;
int main()
{
 f();
 printf(“%d\n”,x);
 return 0;
}
```

```
/* bar3.c */
...
int x;
void f()
{
 x = 123;
}
```

```
unix> gcc –o fb3 foo3.c bar3.c
unix> ./fb3
```

Static Linking

Symbol resolution: multiply defined symbols

- Example-4)

```
/* foo4.c */
...
int x;
int main()
{
 x = 100;
 f();
 printf(“%d\n”,x);
 return 0;
}
```

```
/* bar4.c */
...
int x;
void f()
{
 x = 123;
}
```

```
unix> gcc –o fb4 foo4.c bar4.c
unix> ./fb4
```

Static Linking

- **Symbol resolution: multiply defined symbols**
  - Example-5)

```
/* foo5.c */
...
int x = 1;
int y = 2;
int main()
{
 f();
 printf(“%x,%x\n”,
  x, y);
 return 0;
}
```

```
/* bar5.c */
...
double x;
void f()
{
 x = -0.0;
}
```

```
unix> gcc –o fb5 foo5.c bar5.c
unix> ./fb5
  0, 80000000
```

↓

Hex

Static Linking

- **Symbol resolution: multiply defined symbols**
  - Examples summary)

`int x;`
`p1(){}`
	`p1(){}`	**Link time error: two strong symbols (p1)**

`int x;`
`p1(){}`
	`int x;`
	`p2(){}`
		References to x will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`p1(){}`
	`double x;`
	`p2(){}`
		Writes to x in p2 might overwrite y!

`int x=7;`
`int y=5;`
`p1(){}`
	`double x;`
	`p2(){}`
		Writes to x in p2 will overwrite y!

`int x=7;`
`p1(){}`
	`int x;`
	`p2(){}`
		References to x will refer to the same initialized variable.

Static Linking

Symbol resolution

- Linking with static libraries
  - Static library
    - ✓ A file that packages related object modules
    - ✓ Can be supplied as input to the linker
    - ✓ File format in Unix → **archive**
  - **Archive** (filenames with **.a** suffix)
    - ✓ A collection of concatenated relocatable object files, with a header that describes the size and location of each member object file
    - ✓ Eg) **libc.a**, **libm.a**, …
  - From the archive, the linker copies only the object modules that are referenced by the program, which reduces the size of the executable on disk and in memory

Static Linking

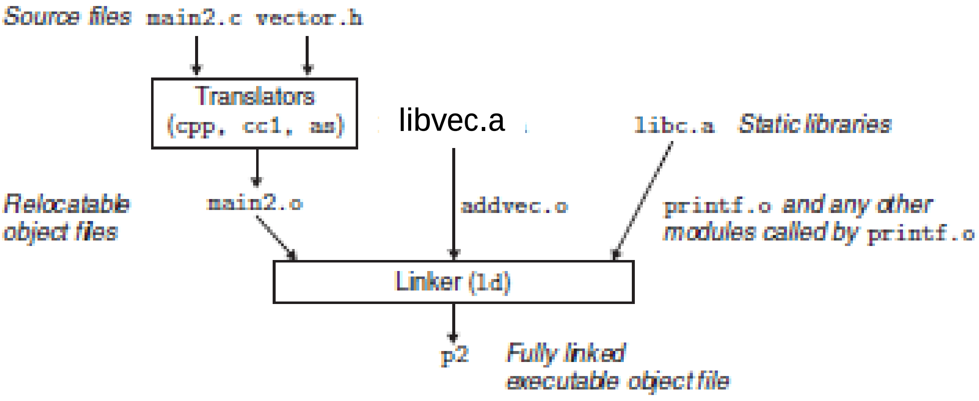
Symbol resolution

- Linking with static libraries
  - Library creation with **ar** tool

```
unix> gcc -c addvec.c mulvec.c
unix> ar rcs libvec.a addvec.o mulvec.o
```

  - Using the library

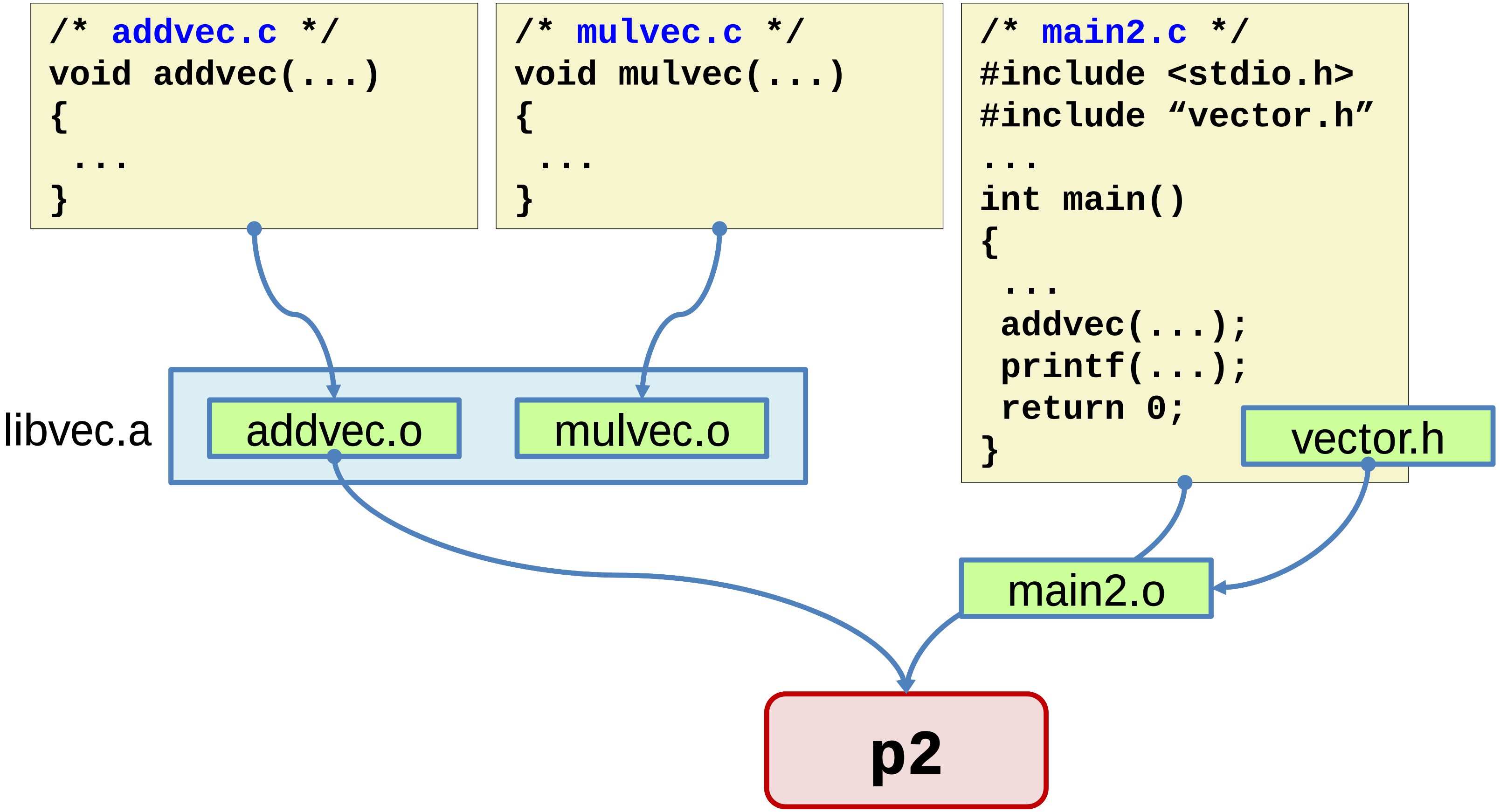
```
unix> gcc -Og -c main2.c
unix> gcc -static -o p2 main2.o ./libvec.a
```



Static Linking

Symbol resolution

- Linking with static libraries



```
/* addvec.c */
void addvec(...)
{
  ...
}
```

```
/* mulvec.c */
void mulvec(...)
{
  ...
}
```

```
/* main2.c */
#include <stdio.h>
#include “vector.h”
...
int main()
{
  ...
  addvec(...);
  printf(...);
  return 0;
}
```

Static Linking

Symbol resolution

- Linking sequence on static libraries
  - During the symbol resolution phase,
    the linker scans the relocatable object files and archives
    left to right in the same sequential order
    that they appear on the compiler driver's command line
    - ✓ Ordering of libraries and object files on the command line is significant in linking

Static Linking

Symbol resolution: linking sequence

- Example)
  - **foo.c** calls functions in **libx.a** and **libz.a** that call functions in **liby.a**

```
unix> gcc -o p1 foo.c libx.a libz.a liby.a
```

  - **foo.c** calls a function in **libx.a** that calls a function in **liby.a** that again calls a function in **libx.a**

```
unix> gcc -o p2 foo.c libx.a liby.a libx.a
```

Static Linking

Relocation

- Merges the input modules and
  assigns run-time addresses to each symbol reference
- 2 steps
  - **Relocating sections and symbol definitions**
    - ✓ Merges all sections of the same type
    - ✓ Assigns run-time addresses to the new aggregate sections,
      to each section defined by the input modules, and
      to each symbol defined by the input modules

Now, every instruction and global variable in the program has a unique run-time address

  - **Relocating symbol references within sections**
    - ✓ Modifies every symbol reference in the code and data sections so that they point to the correct run-time addresses

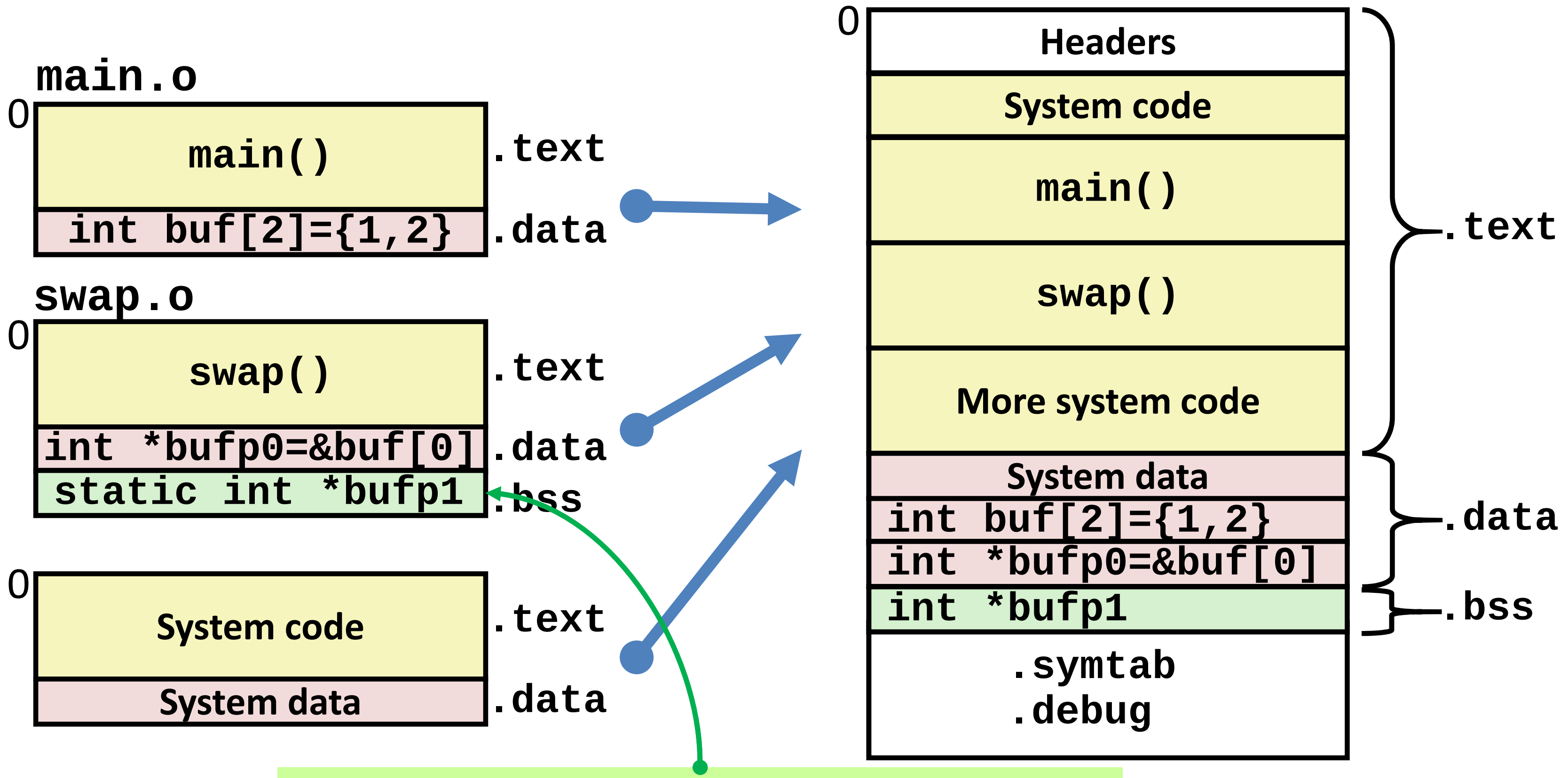
Static Linking

Relocation

- Relocating symbol references

```
/* main.c */
void swap();
int buf[2] = {1, 2};
int main()
{
  swap();
  return 0;
}
```

```
/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;
void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```



main.o

0

main()

int buf[2]={1,2}

.text

.data

swap.o

0

swap()

int *bufp0=&buf[0]

static int *bufp1

.text

.data

.bss

0

System code

System data

.text

.data

0

Headers

System code

main()

swap()

More system code

System data

int buf[2]={1,2}

int *bufp0=&buf[0]

int *bufp1

.symtab
.debug

.text

.data

.bss

Even though private to swap, requires allocation in .bss

Static Linking

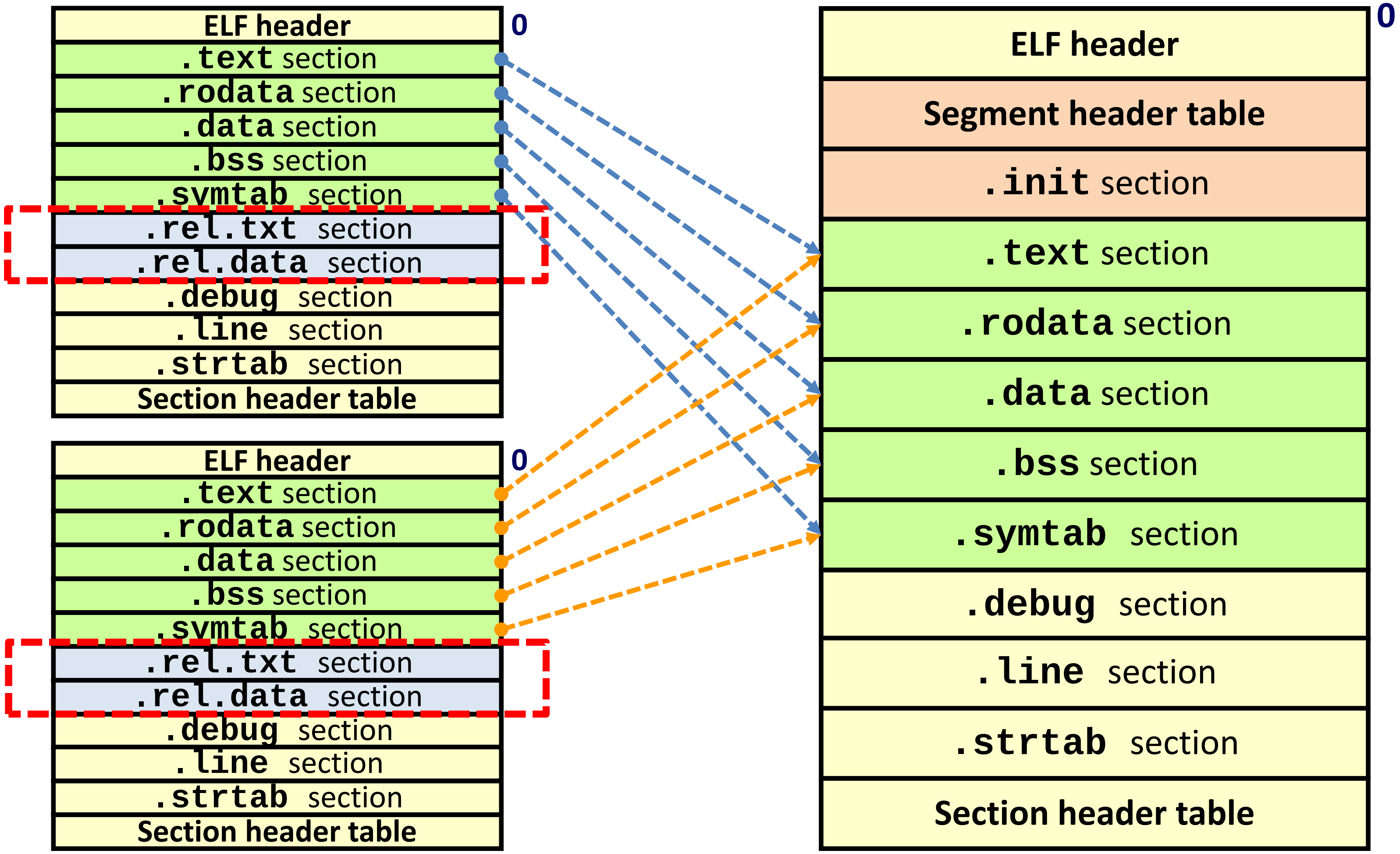
Relocation

- Relocation entries
  - Generated by the assembler
    - ✓ One entry for each reference to an object whose ultimate location is unknown
  - Tells the linker how to modify the reference when it generates an executable
  - Placed in **.rel.text** and **.rel.data**
  - Format of the relocation entry

offset	symbol	type
Where to modify	**Where to point to**	**How to modify**

Static Linking

- Section .rel.text and .rel.data



Static Linking

- Note) commonly used static libraries
  - **libc.a** (C standard library)
    - 8^+MB archive of 900^+ object files
    - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math, etc
  - **libm.a** (C math library)
    - 1^+MB archive of 200^+ object files
    - Floating point math
      - ✓ **sin**, **cos**, **tan**, **log**, **exp**, **sqrt**, etc

Dynamic Linking

- **Loading executables**
  - Loader
    - Invoked by **execve** function
    - Copies the code and data in the executable object file into M and then runs the program by jumping to its **entry point**

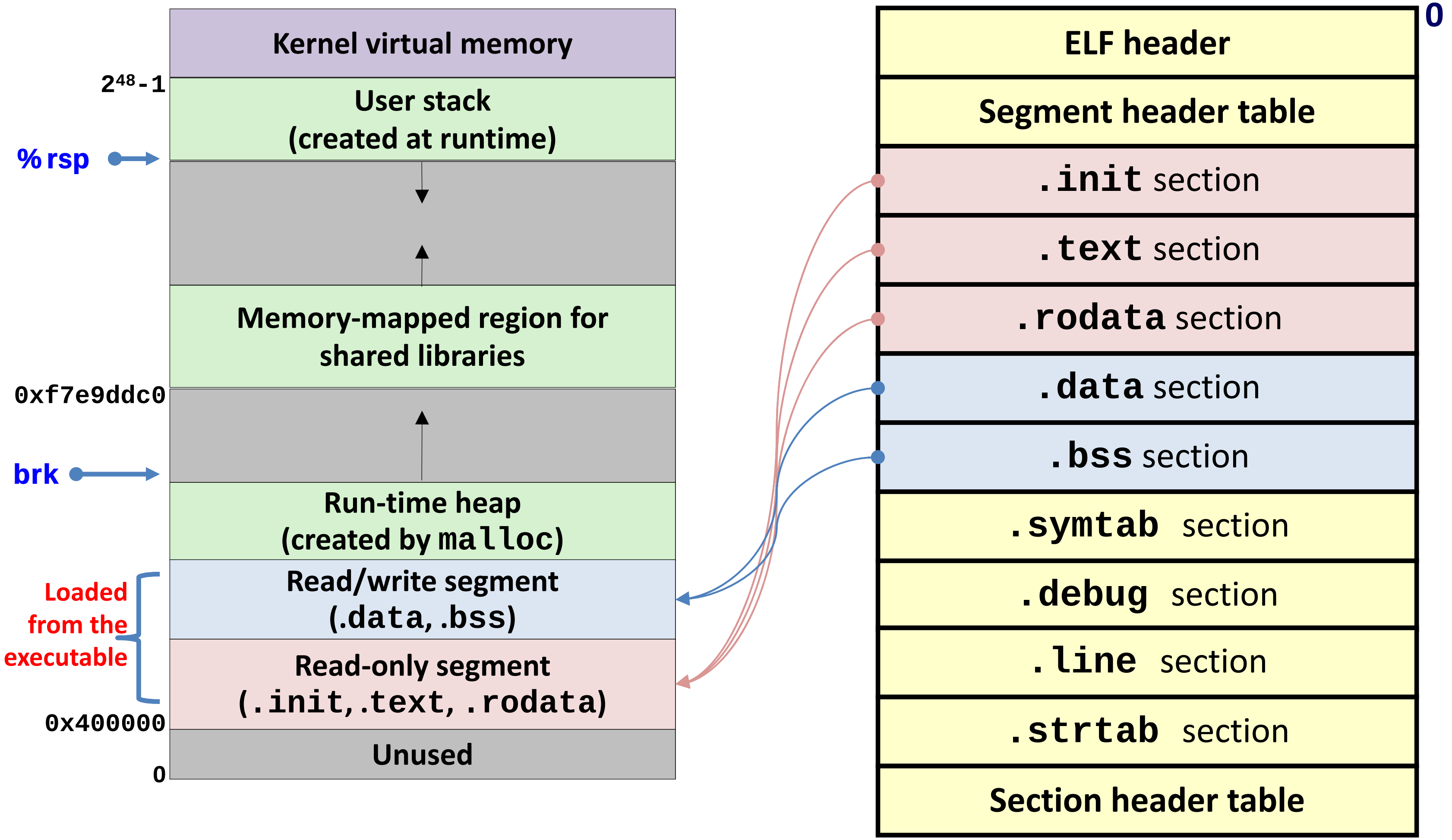
Dynamic Linking

Loading executables

- Loader
  - On x86-64 Linux systems
    - ✓ Code segment starts at address 0x400000
    - ✓ Data segment follows at the next 2MB aligned address
    - ✓ Run-time heap follows on the 2MB aligned address past the data segment
    - ✓ The linker uses ASLR
      when it assigns run-time addresses
      to the stack, shared library, and heap segments

Dynamic Linking

Loading executables



Dynamic Linking

Shared library

- Static libraries have the following disadvantages
  - Duplication in each stored executable
(Every process needs **libc**)
  - Duplication in each running executable (process)
  - Minor changes in the system libraries require relinking on each application

- Modern solution) Shared libraries
  - Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time
  - Also called shared objects (**.so** files in Linux) or DLLs (in MS)
  - Eliminate duplication both in executable and in memory

Dynamic Linking

Shared library

- Building a shared library

```
Linux> gcc -shared -fpic -o libvec.so addvec.c mulvec.c
```

  - Option **–shared**
    - ✓ Directs the linker to create a shared object file
  - Option **–fpic**
    - ✓ Directs the linker to generate position-independent code

Dynamic Linking

Dynamic linking

- Dynamic linking can occur when
  the executable is loaded for running (load-time linking)
  - Linux dynamic linker (**ld-linux.so**)
  - Standard C library (**libc.so**) is usually dynamically linked
- Dynamic linking can also occur
  during running the program (run-time linking)
  - In Linux, this is done by calls to the **dlopen()** interface
- Shared library routines can be shared in memory
  by multiple processes
  - More on this in sections of PIC and virtual memory

Dynamic Linking

Dynamic linking: Load-time linking

- Creating an executable object file

```
Linux> gcc –o proc21 main2.c ./libvec.so
```

  - Does some linking statically when creating the executable
    - ✓ Copies only some relocation and symbol table information
  - Completes linking dynamically when loading the executable

- Loading the executable (for running)
  - Loads the partially linked executable
  - Finds the pathname of the dynamic linker in **.interp** section
  - Runs the dynamic linker (**ld-linux.so** in Linux)

Dynamic Linking

- **Dynamic linking: Load-time linking**
  - Finishing linking (before running)
    - Relocates the text and data of **libc.so** and **libvec.so**, each into separate memory segments
    - Relocates any references in the executable to the symbols defined by **libc.so** and **libvec.so**
  - Running the executable
    - Passes control to the application

Dynamic Linking

Dynamic linking: Run-time linking

- Loading and linking shared libraries from applications
  - Requests the dynamic linker to load and link arbitrary shared libraries while the application is running, without having to link the application against those libraries at compile time
- Usage)
  - Distributing software
  - High-performance web servers
  - Runtime library interpositioning
  - Etc

Dynamic Linking

- **Dynamic linking: Run-time linking**
  - Interfaces for run-time linking in Linux
    - **dlopen**
    - **dlsym**
    - **dlclose**
    - **dlerror**

Tools on Object Files

Tools on Unix/Linux systems

Tools	Functions
ar	Creates static libraries, and inserts, deletes, lists, and extracts members
strings	Lists all of the printable strings contained in an object file
strip	Deletes symbol table information from an object file
nm	Lists the symbols defined in the symbol table of an object file
size	Lists the names and sizes of the sections in an object file
readelf	Displays the complete structure of an object file
objdump	Displays all of the information in an object file, disassembling the binary instructions in the **.text** section
ldd	Lists the shared libraries that an executable needs at run time

Summary