운영체제의 기초:
GNU Linker

2023년 4월 25, 27일

홍 성 수
sshong@redwood.snu.ac.kr
SNU RTOSLab 지도교수
서울대학교 전기정보공학부 교수

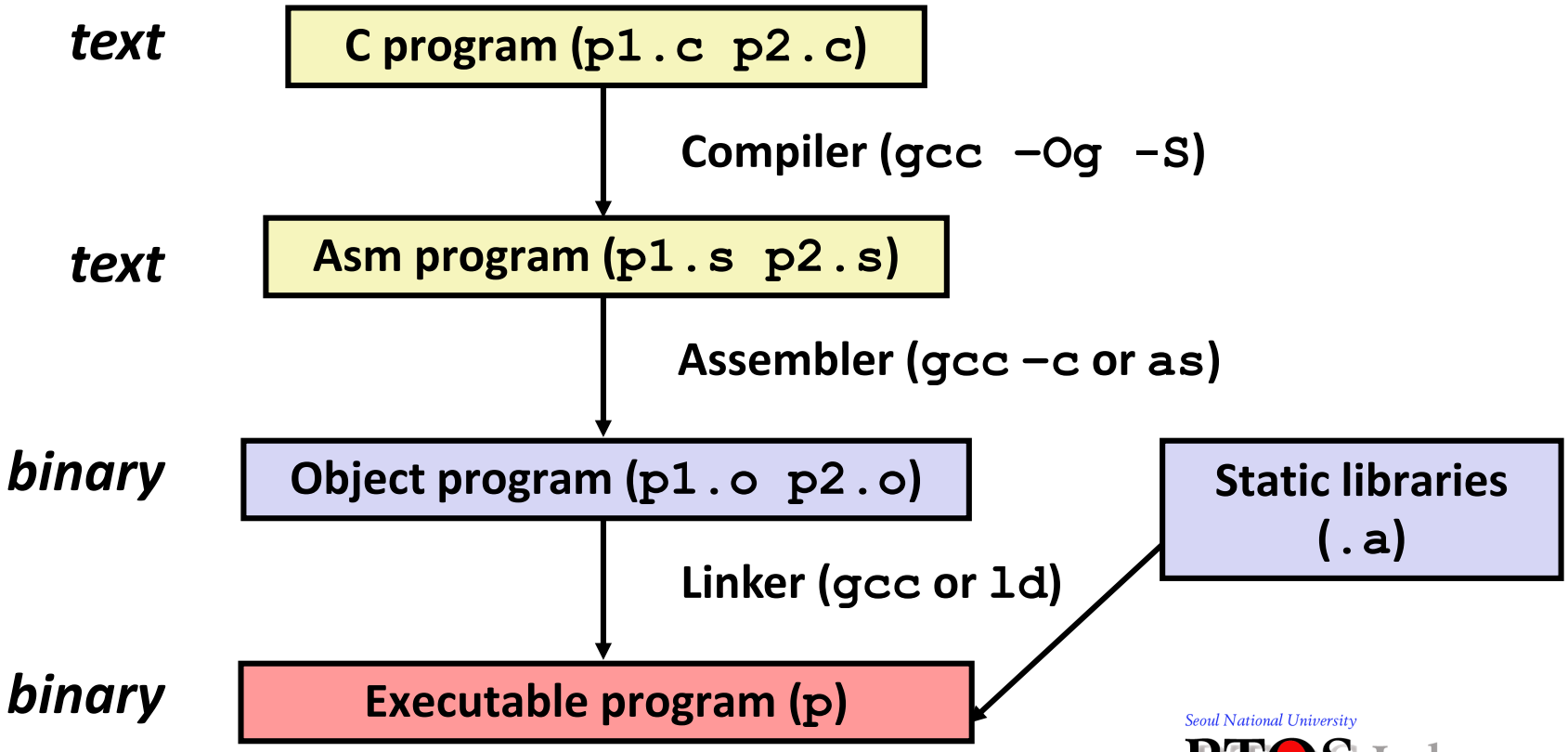
Agenda

I. Compiling Process
II. Linking Process
III. Walk-thru Example
IV. Step 1: Symbol Resolution
V. Step 2: Relocation
VI. Linking Libraries

I. Compiling Process

Turning C into Object Code

- Translates and links programs
  - `linux> gcc -Og -o prog p1.c p2.c`



Example C Code

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

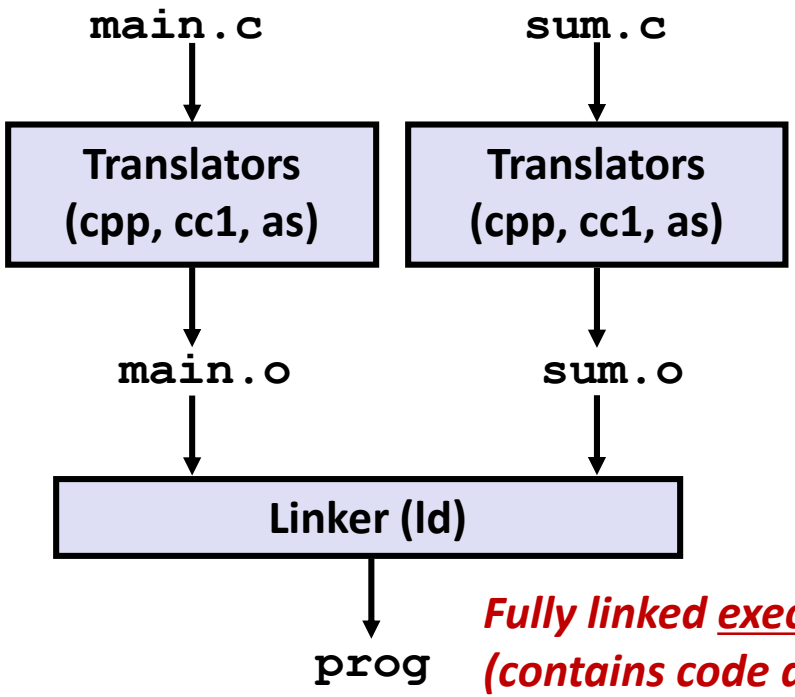
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Compiler Driver

- Translates and links programs
  - `linux> gcc -Og -o prog main.c sum.c`



Source files

Separately compiled relocatable object files

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

Compiler Driver and Loader (1)

- Compiler driver consists of:
  - C-preprocessor, compiler and assembler
    - Convert each *source* file into *object* file
  - Linker
    - Combine all object files into one object file (or *executable* file)
    - Specify memory address to load instructions and data
- Loader
  - Read the executable file
  - Extract addresses from the file to load instructions and data
  - Load instructions and data into those positions
  - OS implements the loader function

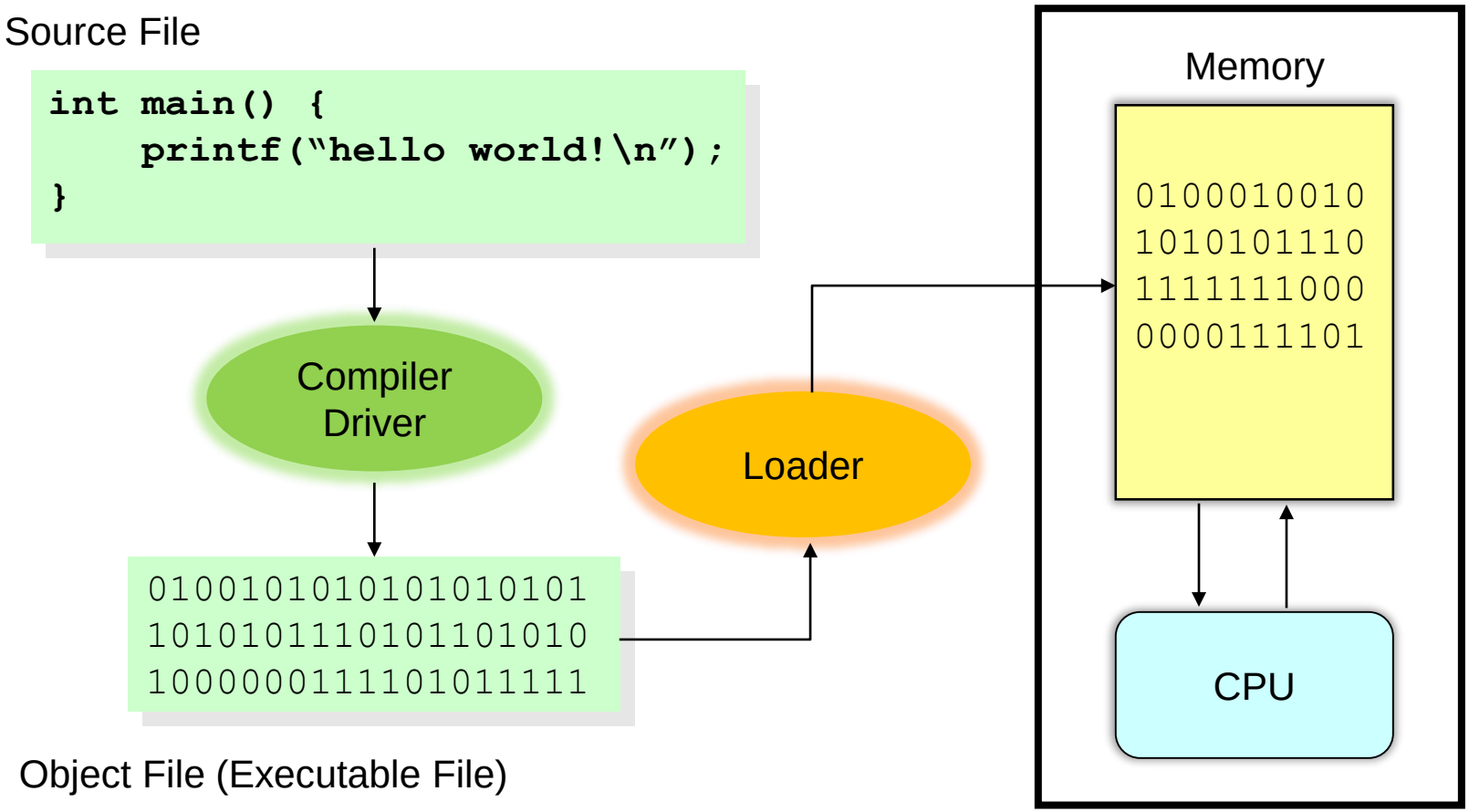
Compiler Driver and Loader (2)

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

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

- Object file
  - *Binary values* to be loaded into memory (instructions, data)
  - *Memory addresses* to load these binary values
  - *Symbol table*: One entry per defined symbol
  - *Relocation table*: One entry per external symbol reference

Compiler Driver and Loader (3)



Three Kinds of Object Files (AKA Modules)

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

Inside Object File (1)

- An object file consists of sections (segments)
  - A formatted unit collectively comprising an object file
    - Each section is considered as having private memory (segment)
    - Assigned a contiguous memory area containing entities with the same property in a given address space
    - Location of a symbol in a section is expressed as [**`section name: offset in section`**]

Inside Object File (2)

- Type of sections
  - Text
    - Contains code to be executed (E.g., `printf("hello");`)
  - Data
    - Contains initialized global variables (E.g., `int a = 0;`)
  - BSS
    - Contains uninitialized global variables (E.g., `int b;`)
  - Other sections
    - Symbol table, relocation table, …
  - Only text and data sections are loaded into memory for execution

Inside Object File (3)

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

II. Linking Process

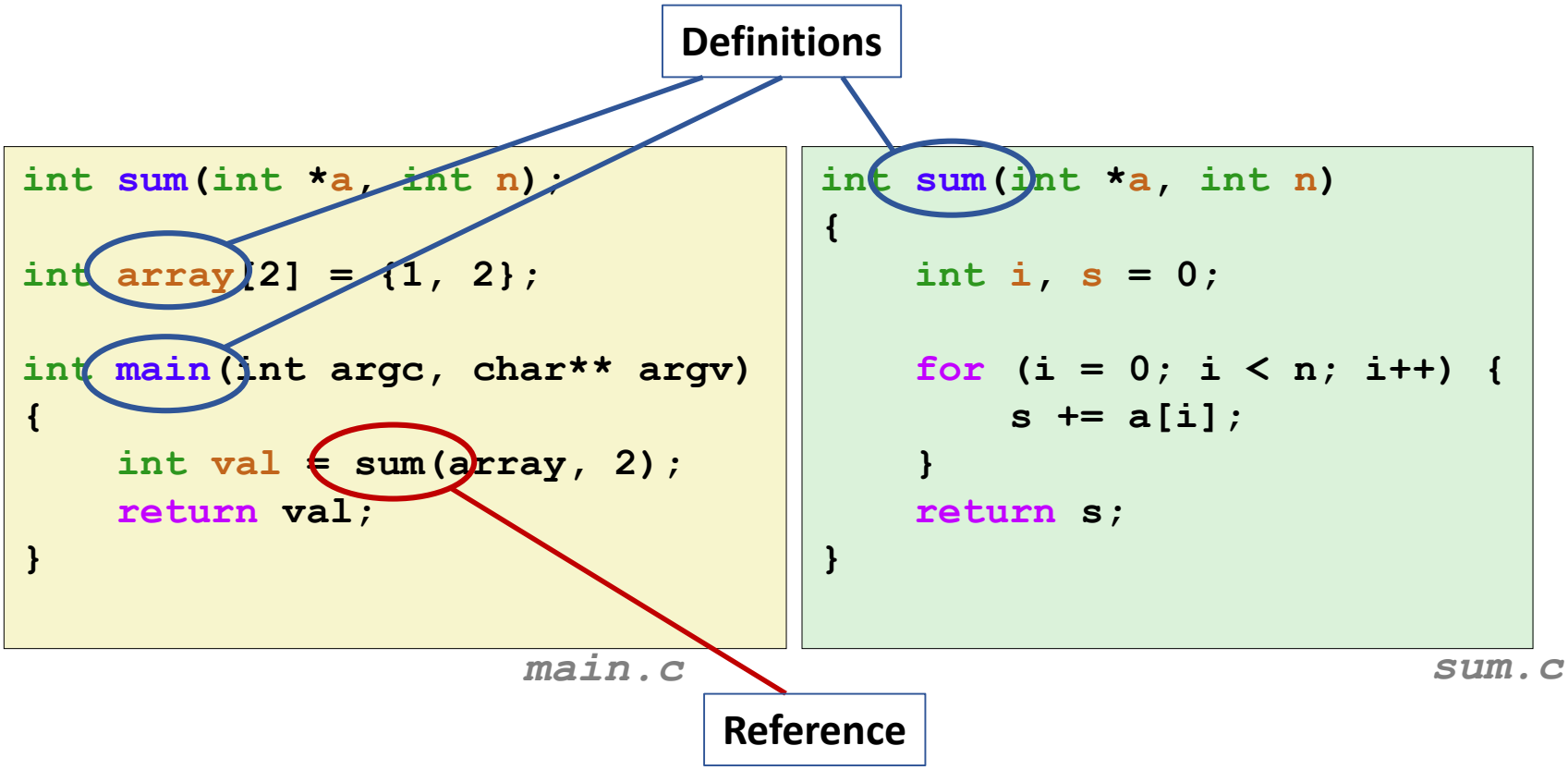
Problems in Compiling and Linking

- *Problem 1*:
  - "*Computer can't recognize symbolic names*"
  - Solution:
    - Translate symbolic names to memory addresses using symbol table
- *Problem 2*:
  - "*Compiler doesn't know addresses of sections to be loaded*"
    - Symbol table alone cannot solve the problem (cross-reference)
  - Solution:
    - Record all instructions that reference symbols of other sections
    - Fix these instructions so as to reference actual memory address during linking using relocation table

What Do Linkers Do? (1)

- Solution to *P1*: *Symbol resolution*
  - Programs define and reference *symbols* (global variables and functions)
    - `void swap(){…} /* define symbol swap */`
    - `swap();  /* reference symbol swap */`
    - `int *xp = &x;  /* define xp, reference x */`
  - Symbol definitions are stored in object file in *symbol table*
    - Symbol table is an array of entries
    - Each entry includes name, size, and location of symbol
  - During the symbol resolution step, the linker associates each symbol reference with exactly one symbol definition

Symbols Example C Code



What Do Linkers Do? (2)

- Solution to *P2*: *Relocation*
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable file
  - Updates all references to these symbols to reflect their new positions

Why Linkers? (1)

- Reason 1: *Modularity*
  - Program can be written as a collection of smaller source files rather than one monolithic mass
  - Can build libraries of common functions
    - E.g., math library, standard C library

Why Linkers? (2)

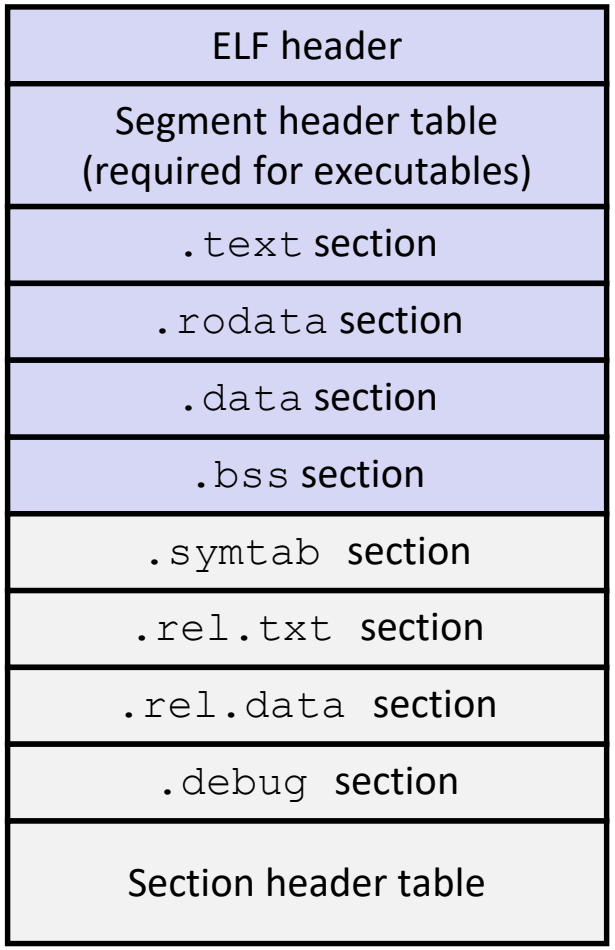
- Reason 2: *Efficiency*
  - *Time*: Separate compilation
    - Change one source file, compile, and then relink
    - No need to recompile other source files
    - Can compile multiple files concurrently
  - *Space*: Libraries
    - Common functions can be aggregated into a single file ...
    - Option 1: *Static Linking*
      - Executable files and running memory images contain only the library code they actually use
    - Option 2: *Dynamic linking*
      - Executable files contain no library code
      - During execution, single copy of library code can be shared across all executing processes

Executable and Linkable Format (ELF)

- Standard binary format for object files
- One unified format for
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Shared object files (`.so`)
- Generic name: ELF binaries

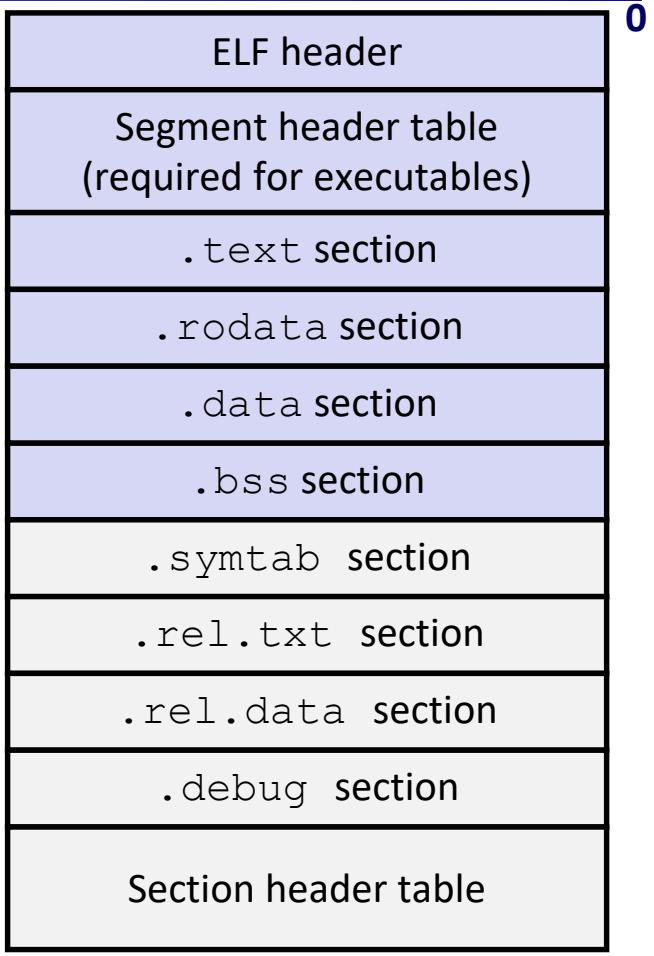
ELF Object File Format (1)

- ELF header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - Page size, virtual address memory segments (sections), segment sizes
- **`.text`** section
  - Code



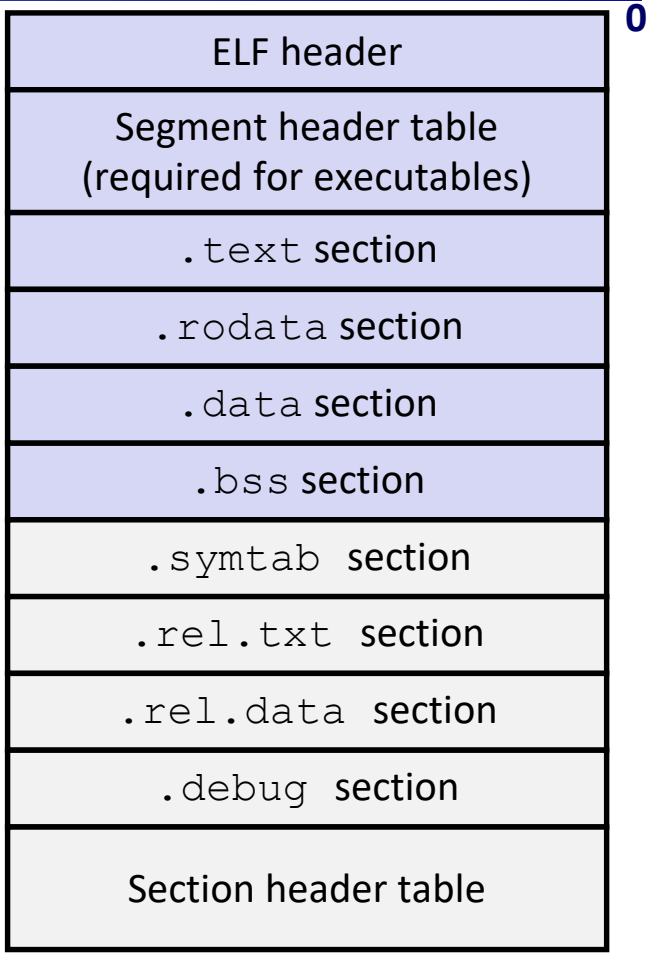
ELF Object File Format (2)

- `.rodata` section
  - Read only data
    - Jump tables, string constants, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space



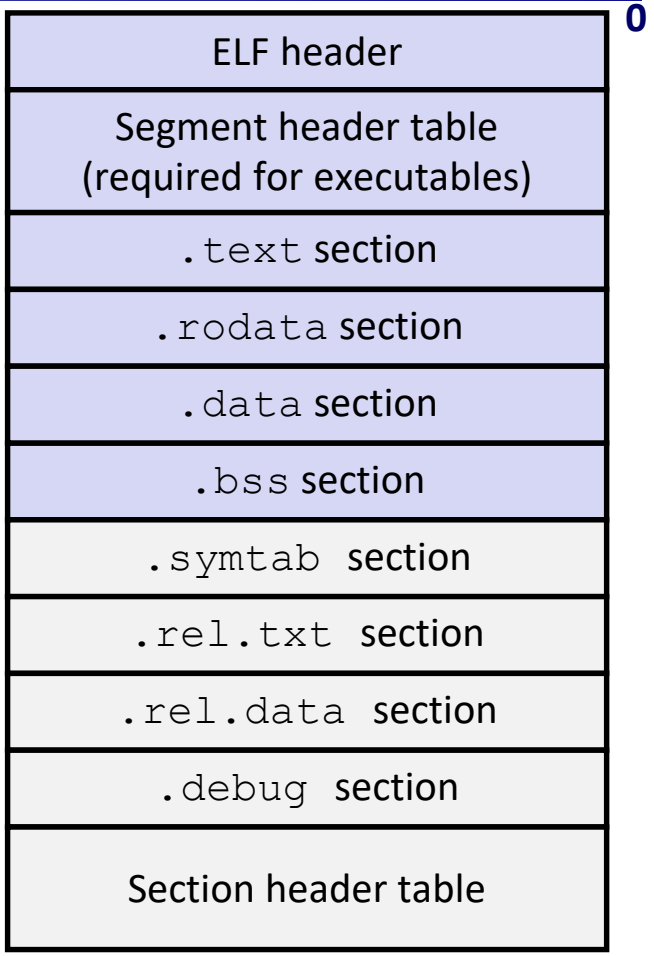
ELF Object File Format (3)

- **`.symtab`** section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text`** section
  - Relocation info for **`.text`** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying



ELF Object File Format (4)

- **`.rel.data`** section
  - Relocation info for **`.data`** section
  - Addresses of pointer data that will need to be modified in the merged executable
- **`.debug`** section
  - Info for symbolic debugging (**`gcc -g`**)
- Section header table
  - Offsets and sizes of each section



III. Walk-thru Example

Generation of Sections

```
int x = 10;
int y = 20;
int z = 30;
int m;

int func(int a) {
    return a+1;
}

void main() {
    m = x + y + z;
    x = func(10);
    return;
}
```

Compiler

Text Section

```
0x0   func:
          ……..

0x10  main:
          ……..
          ……..
```

bss Section

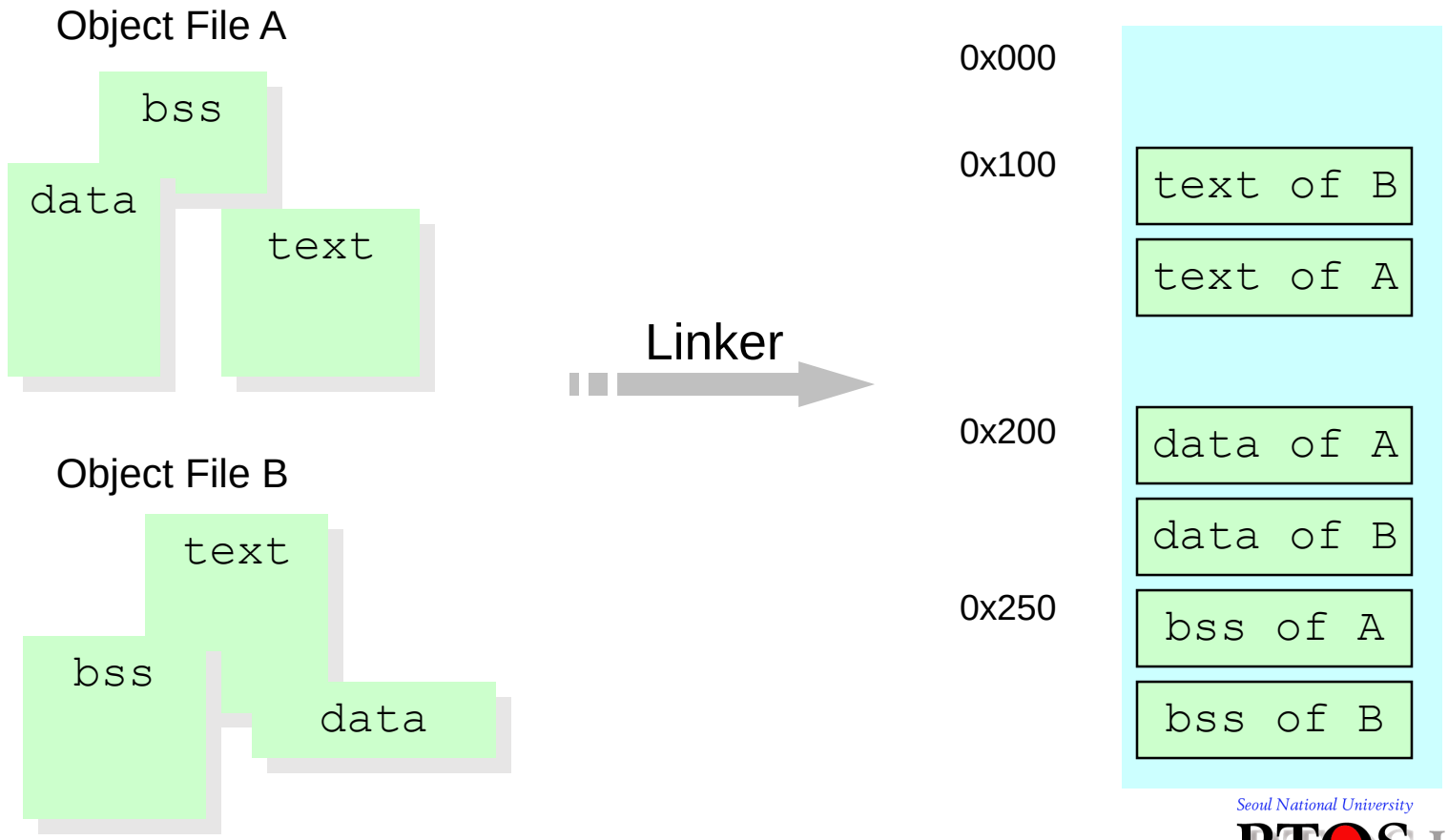
```
0x0   m:
```

Data Section

```
0x0   x: 10
0x4   y: 20
0x8   z: 30
```

Merging Sections Together

- Lays out sections from object files in total order



Specifying Section Layout

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

Loading

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

Symbol Table (1)

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

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

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

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

Symbol Table (2)

- Symbol table example (symbol name: location)

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

Symbol Table (3)

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

Section	Section
text	0x1000
data	0x3000
bss	0x3500

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

section address
+
offset in section

Relocation Table (1)

- Compiler can't convert all symbol references using symbol table
  - Symbols in the same section
  - Symbols in other sections (cross-references)

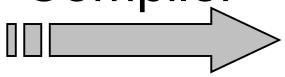
- Symbols in the same section
  - Can be referenced via PC relative addressing
  - Don't need to know memory addresses in this section
  - This type of relocation can be done by compiler
  - Ex) `call local_func` → `call PC+0x1B`

Relocation Table (2)

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

Relocation Table (2)

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

void main() {
    my_var = 10;
    func(10);
}
```



Text Section

```
main:
    ……..
    mov $10 0x0
    ……..
    call 0x0
    ……..
```

0x00
0x16
0x24

Relocation Table

Ref. Section	Ref. Offset	Symbol Name
text	0x16	my_var
text	0x24	func

Relocation Table (3)

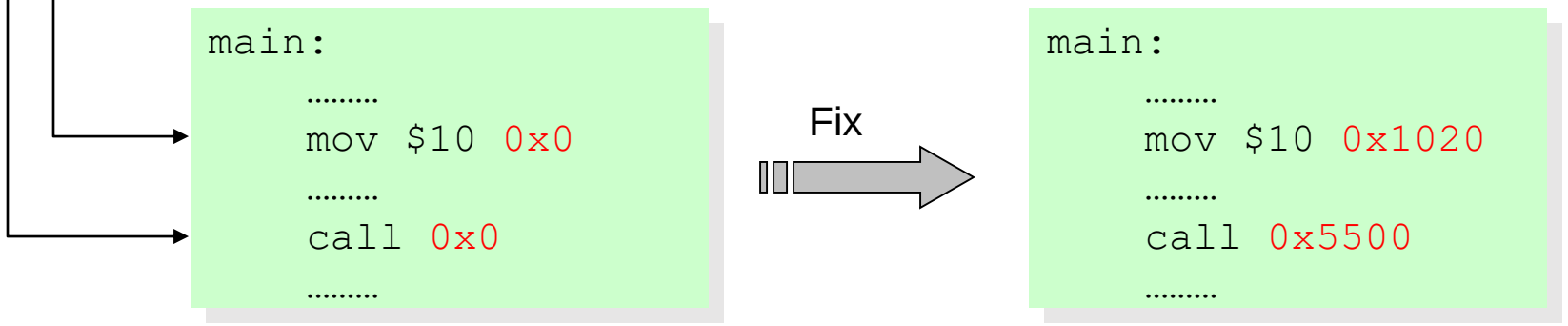
- Linker knows addresses of all symbols
- Fixes addresses of cross referenced symbols (that were zero) to actual addresses

Relocation Table

Ref. Section	Ref. Offset	Symbol Name
text	0x16	my_var
text	0x24	func

Final Symbol Table

Symbol Name	Memory Address
my_var	0x1020
func	0x5500



Overall Example: Source Code

```
// main.c:

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

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

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

```
// lib.c:

int my_var = 100;

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

Overall Example: Compiling "main.c"

Symbol Table

Symbol Name	Section	Offset
local_func	text	0x50
main	text	0x0

Relocation Table

Ref. Section	Ref. Offset	Symbol Name
text	0x20	func
text	0x30	my_var
text	0x38	my_var

Text Section

```
0x00  main:
            .....                 PC = 0x08
0x04        call PC+0x48
            .....
0x20        call 0x0
            .....
0x30        mov 0x0 %eax
            add $10 %eax
0x38        mov %eax 0x0
            .....

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

Overall Example: Compiling “lib.c”

Symbol Table

Symbol Name	Section	Offset
my_var	data	0x0
func	text	0x0

Relocation Table

<None>

Data Section

0x00

```
my_var:
        100
```

Text Section

0x00

```
func:
        .....
        .....
```

Overall Example: Linking "main.o" and "lib.o"

- Step 1: Calculate addresses of all symbols

Layout of Sections

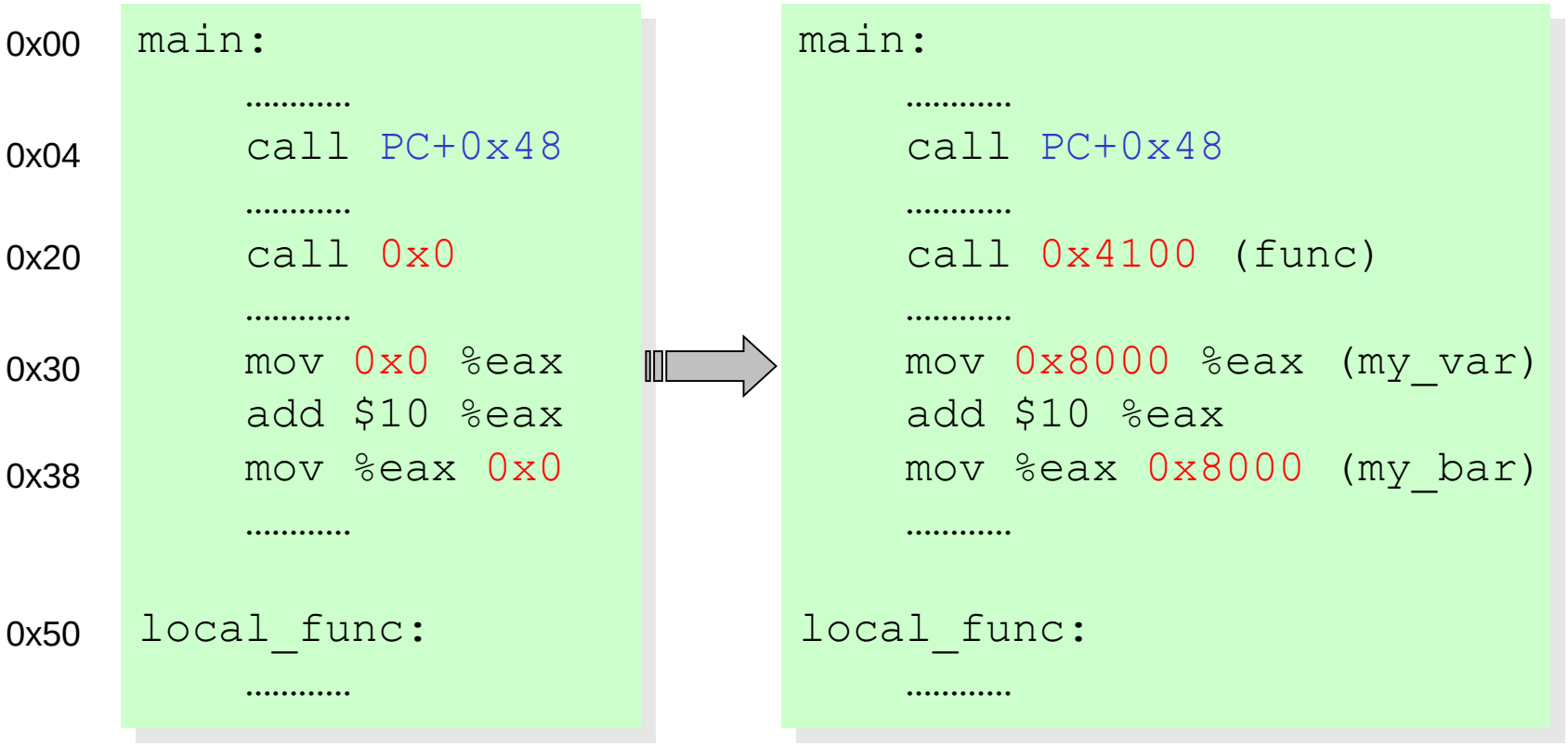
Section	Load Address
main.o (text)	0x4000
lib.o (text)	0x4100
lib.o (data)	0x8000

Final Symbol Table

Symbol Name	Memory Address
main	0x4000 + 0x00 = **0x4000**
local_func	0x4000 + 0x50 = **0x4050**
func	0x4100 + 0x00 = **0x4100**
my_var	0x8000 + 0x00 = **0x8000**

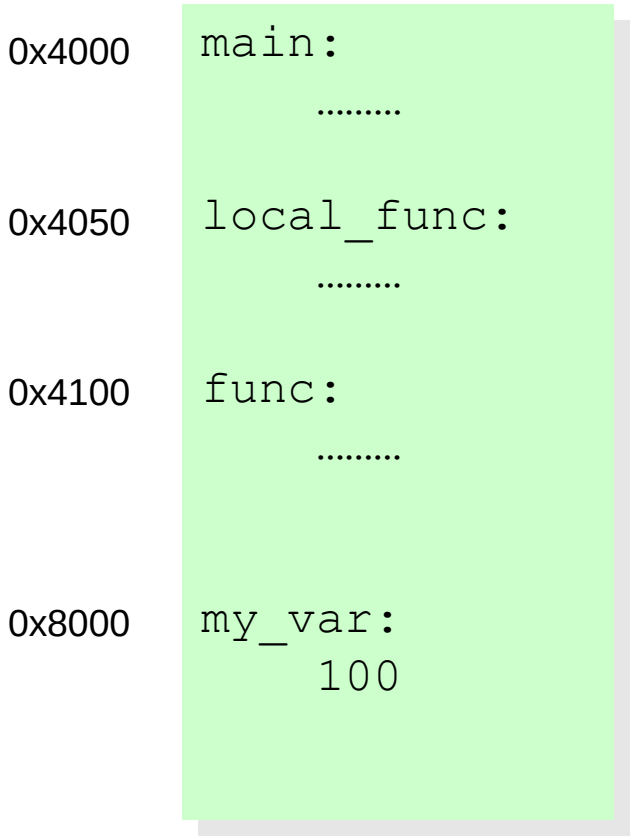
Overall Example: Linking "main.o" and "lib.o"

- Step 2: Fix instructions marked in relocation table

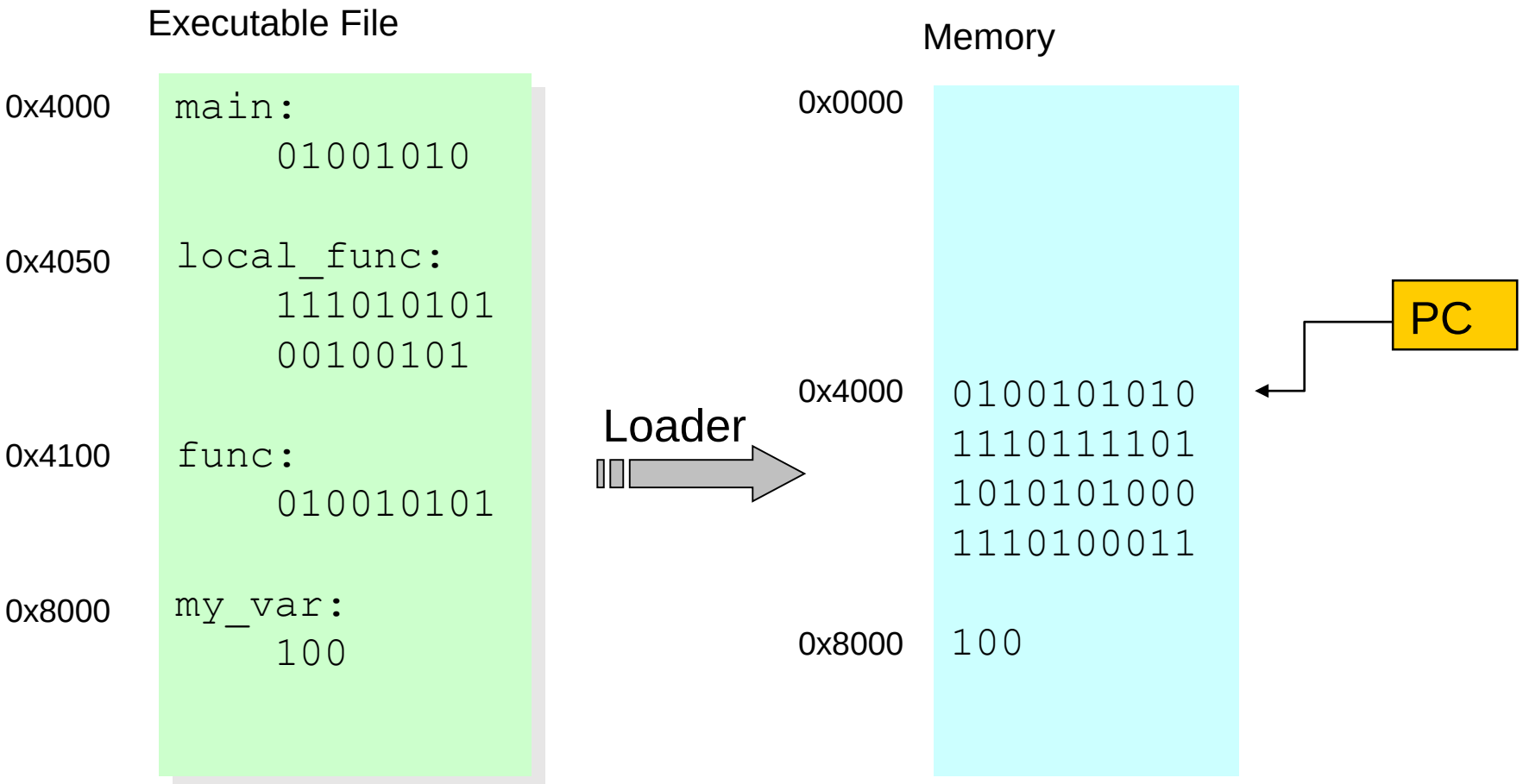


Overall Example: Linking "main.o" and "lib.o"

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



Overall Example: Loading

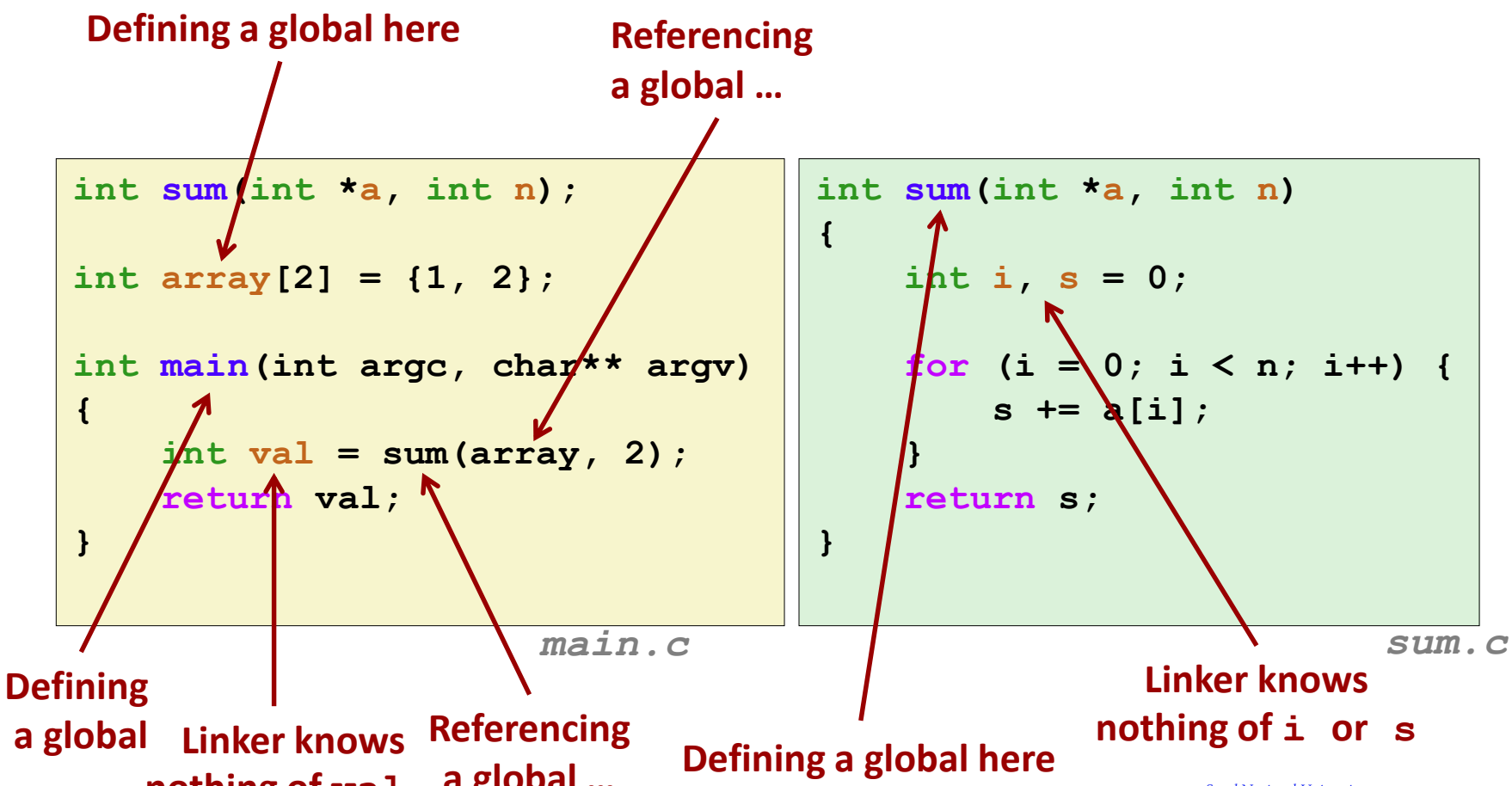


IV. Step 1: Symbol Resolution

Linker Symbols

- *Global* symbols
  - Symbols defined by module *m* that can be referenced by other modules
    - E.g., non-static C functions and non-static 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*
    - E.g., C functions and global variables defined with **`static`**
  - Local linker symbols are not local program variables

Various Symbol Usage



Symbol Identification

- Which of the following names will be in the symbol table of **`symbols.o`**?
  - Names
    - **`incr`**
    - **`foo`**
    - **`a`**
    - **`argc`**
    - **`argv`**
    - **`b`**
    - **`main`**
    - **`printf`**
    - **`"%d\n"`**
  - Can find this with **`readelf`**
    - **`Linux> readelf -s symbols.s`**

symbols.c

```
int incr = 1;
static int foo(int a) {
  int b = a + incr;
  return b;
}

int main(int argc,
         char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

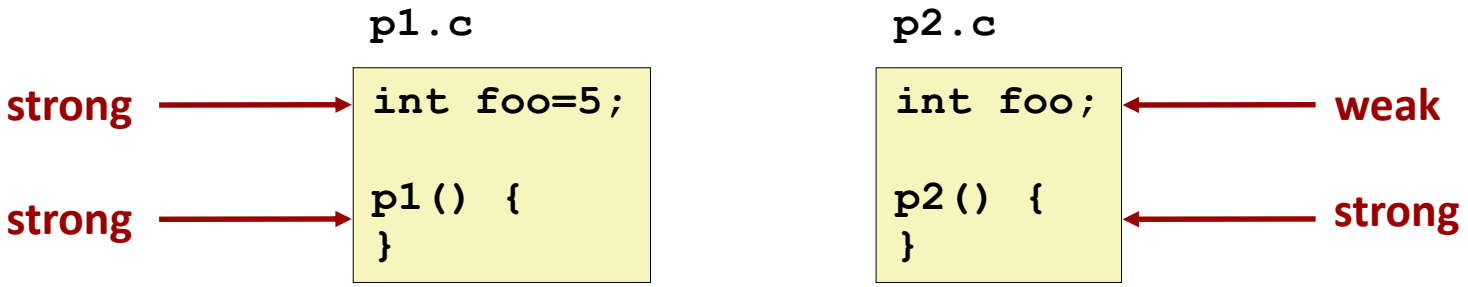
Local Symbols

- Local non-static variables vs. local static variables
  - Local non-static C variables: stored on the stack
  - Local static C variables: stored in either **`.bss`** or **`.data`**
  - Compiler allocates space in **`.data`** for each definition of **`x`**
  - Creates local symbols in the symbol table with unique names, e.g., **`x, x.1721`** and **`x.1724`**

```
static int x = 15;
int f() {
    static int x = 17;
    return x++;
}
int g() {
    static int x = 19;
    return x += 14;
}
int h() {
    return x += 27;
}
```

Duplicate Symbol Definitions (1)

- Program symbols are either strong or weak
  - *Strong*: procedures and initialized globals
  - *Weak*: uninitialized globals
    - Or ones declared with specifier **`extern`**



Duplicate Symbol Definitions (2)

- Linker's symbol rules
  - Rule 1:
    - Multiple strong symbols are not allowed
      - Each item can be defined only once
      - Otherwise: Linker error
  - Rule 2:
    - Given a strong symbol and multiple weak symbols, choose the strong symbol
      - References to the weak symbol resolve to the strong symbol
  - Rule 3:
    - If there are multiple weak symbols, pick an arbitrary one
      - Can override this with **`gcc -fno-common`**

Duplicate Symbol Definitions (3)

`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**!
		Evil!

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

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

Important: Linker does not do type checking

Duplicate Symbol Definitions (4)

- Type mismatch example

```c
long int x;  /* Weak symbol
*/

int main(int argc,
         char *argv[]) {
    printf("%ld\n", x);
    return 0;
}
```

mismatch-main.c

```c
/* Global strong symbol */
double x = 3.14;
```

mismatch-variable.c

- Compiles without any errors or warnings
- What gets printed?

```
-bash-4.2$ ./mismatch
4614253070214989087
```

Global Variables

- Avoid if you can
- Otherwise
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable
    - Treated as weak symbol
    - But also causes linker error if not defined in some file

Global Variables (1)

- Use of **`extern`** in **`.h`** files

`c1.c`

```
#include "global.h"

int f() {
  return g+1;
}
```

`global.h`

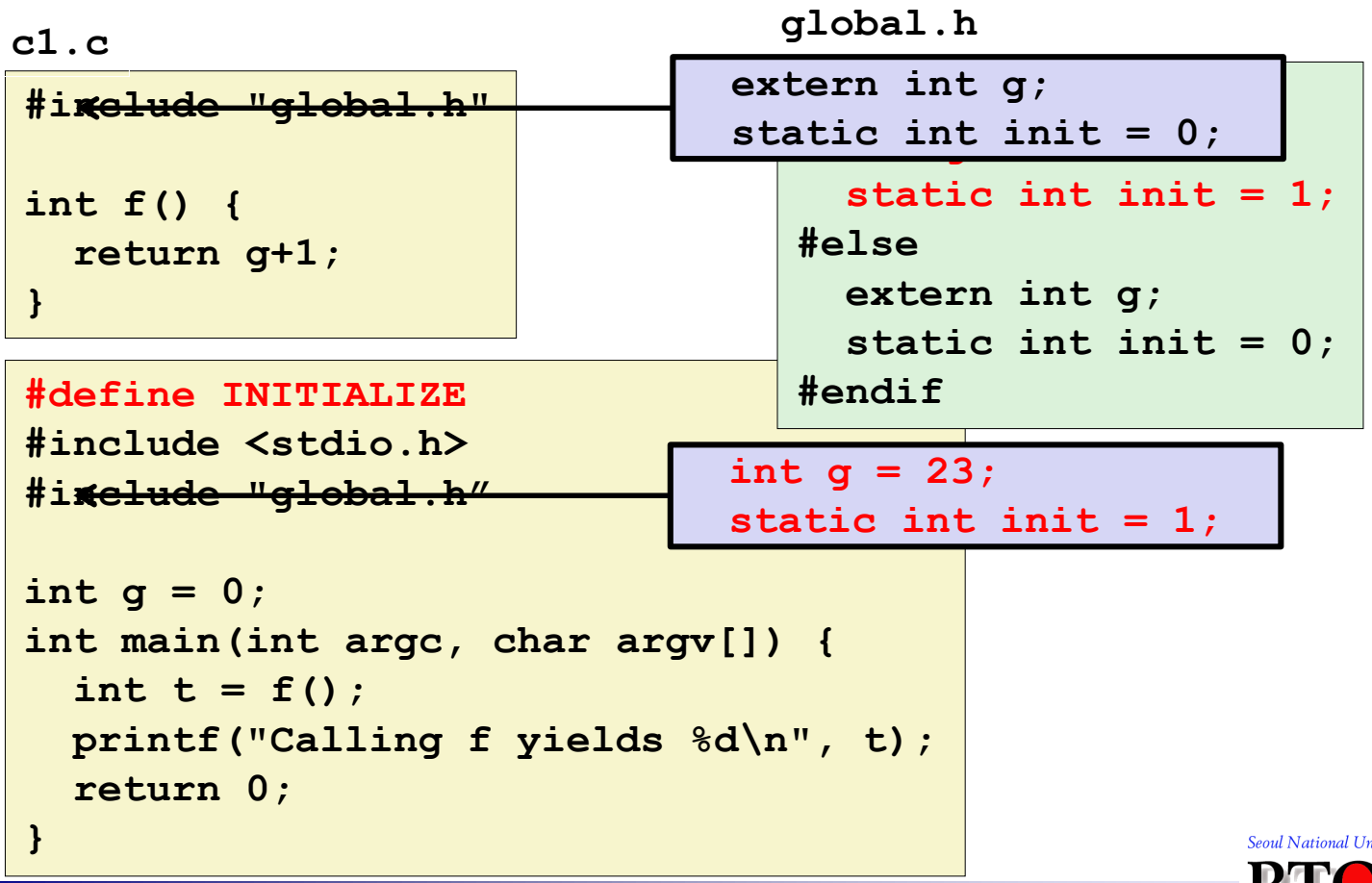
```
extern int g;
int f();
```

`c2.c`

```
#include <stdio.h>
#include "global.h"

int g = 0;

int main(int argc, char argv[]) {
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Global Variables (2)

- Use of **`extern`** in **`.h`** files



V. Step 2: Relocation

Linking Example Code

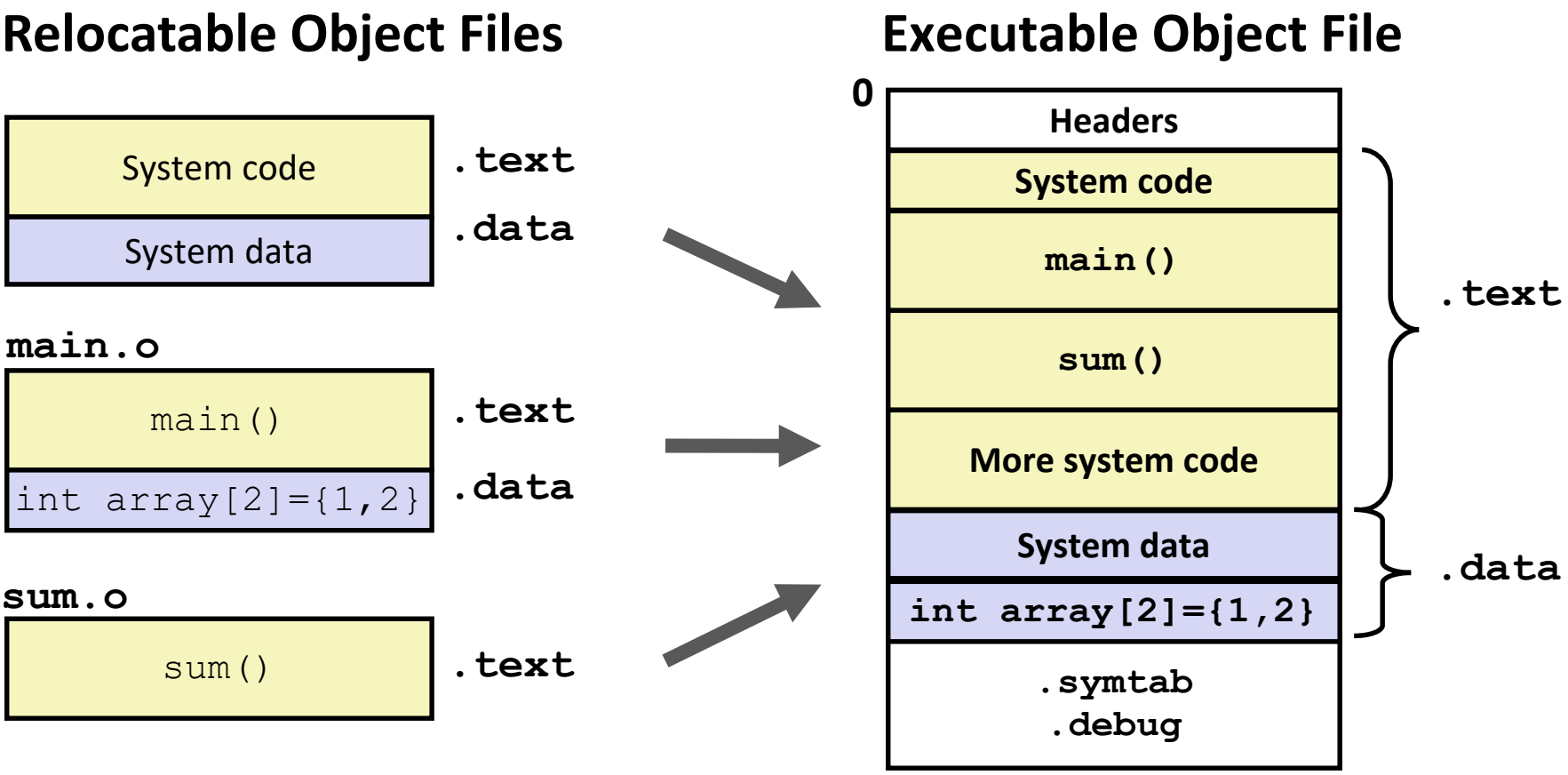
```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Linking Object Files into Executable



Relocation Entries

```c
int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

Source: `objdump -r -d main.o`

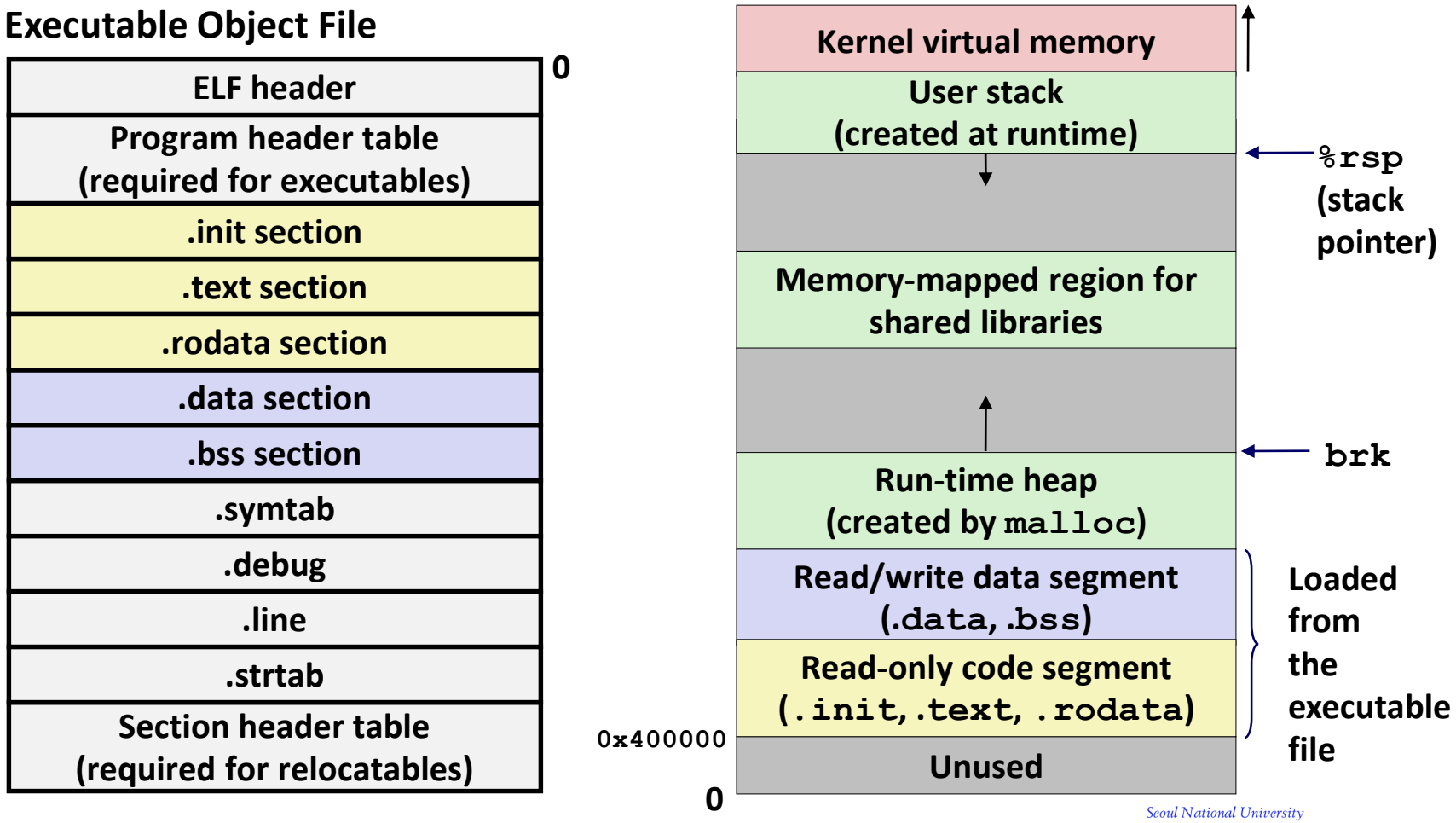
```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi        # %edi = &array
                        a: R_X86_64_32 array           # Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4       # Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

main.o

Relocated .text Section

```
00000000004004d0 <main>:
  4004d0:        48 83 ec 08        sub    $0x8,%rsp
  4004d4:        be 02 00 00 00     mov    $0x2,%esi
  4004d9:        bf 18 10 60 00     mov    $0x601018,%edi   # %edi = &array
  4004de:        e8 05 00 00 00     callq  4004e8 <sum>     # sum()
  4004e3:        48 83 c4 08        add    $0x8,%rsp
  4004e7:        c3                 retq

00000000004004e8 <sum>:
  4004e8:        b8 00 00 00 00           mov    $0x0,%eax
  4004ed:        ba 00 00 00 00           mov    $0x0,%edx
  4004f2:        eb 09                    jmp    4004fd <sum+0x15>
  4004f4:        48 63 ca                 movslq %edx,%rcx
  4004f7:        03 04 8f                 add    (%rdi,%rcx,4),%eax
  4004fa:        83 c2 01                 add    $0x1,%edx
  4004fd:        39 f2                    cmp    %esi,%edx
  4004ff:        7c f3                    jl     4004f4 <sum+0xc>
  400501:        f3 c3                    repz retq
```

Loading Executable Object Files



VI. Linking Libraries

Libraries: Packaging Functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

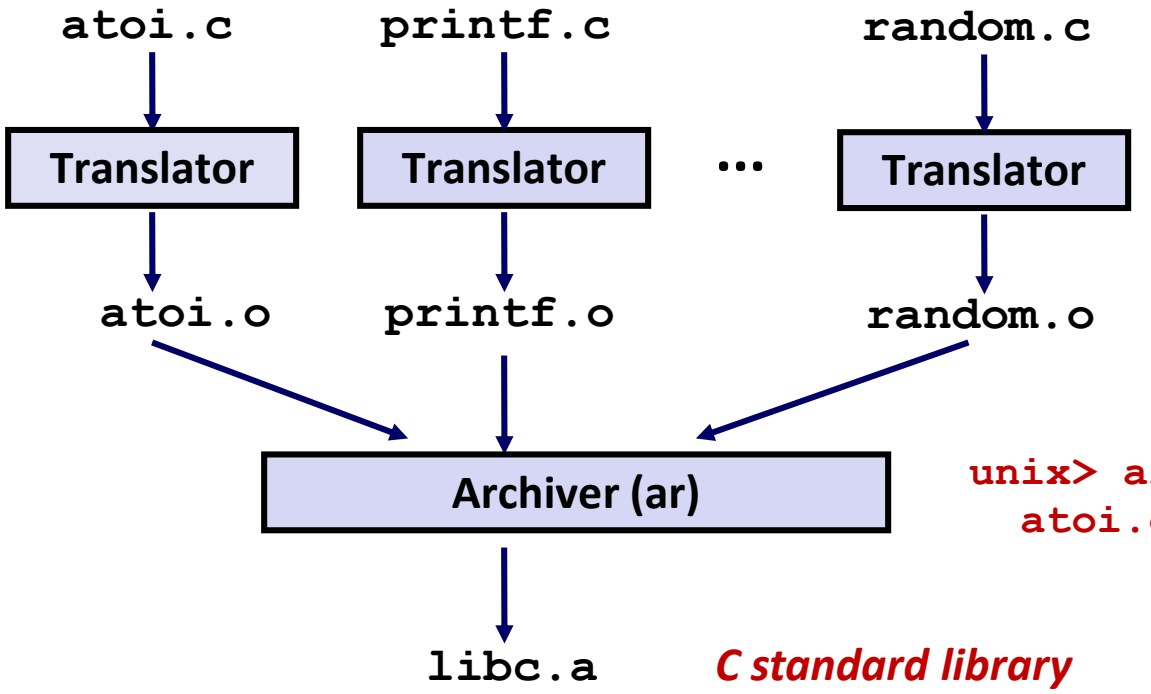
- Awkward, given the linker framework so far:
  - Option 1: Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - Option 2: Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Static Libraries

- Old-Fashioned Solution: `.a` archive files
  - Concatenate related relocatable object files into a single file with an index (called an archive)
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives
  - If an archive member file resolves reference, link it into the executable

Creating Static Libraries

- Archiver allows incremental updates
- Recompile function that changes and replace `.o` file in archive




```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

Commonly Used Libraries (1)

- **`libc.a`** (the C standard library)
  - 4.6 MB archive of 1496 object files
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

Commonly Used Libraries (2)

- **`libm.a`** (the C math library)
  - 2 MB archive of 444 object files.
  - Floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries (1)

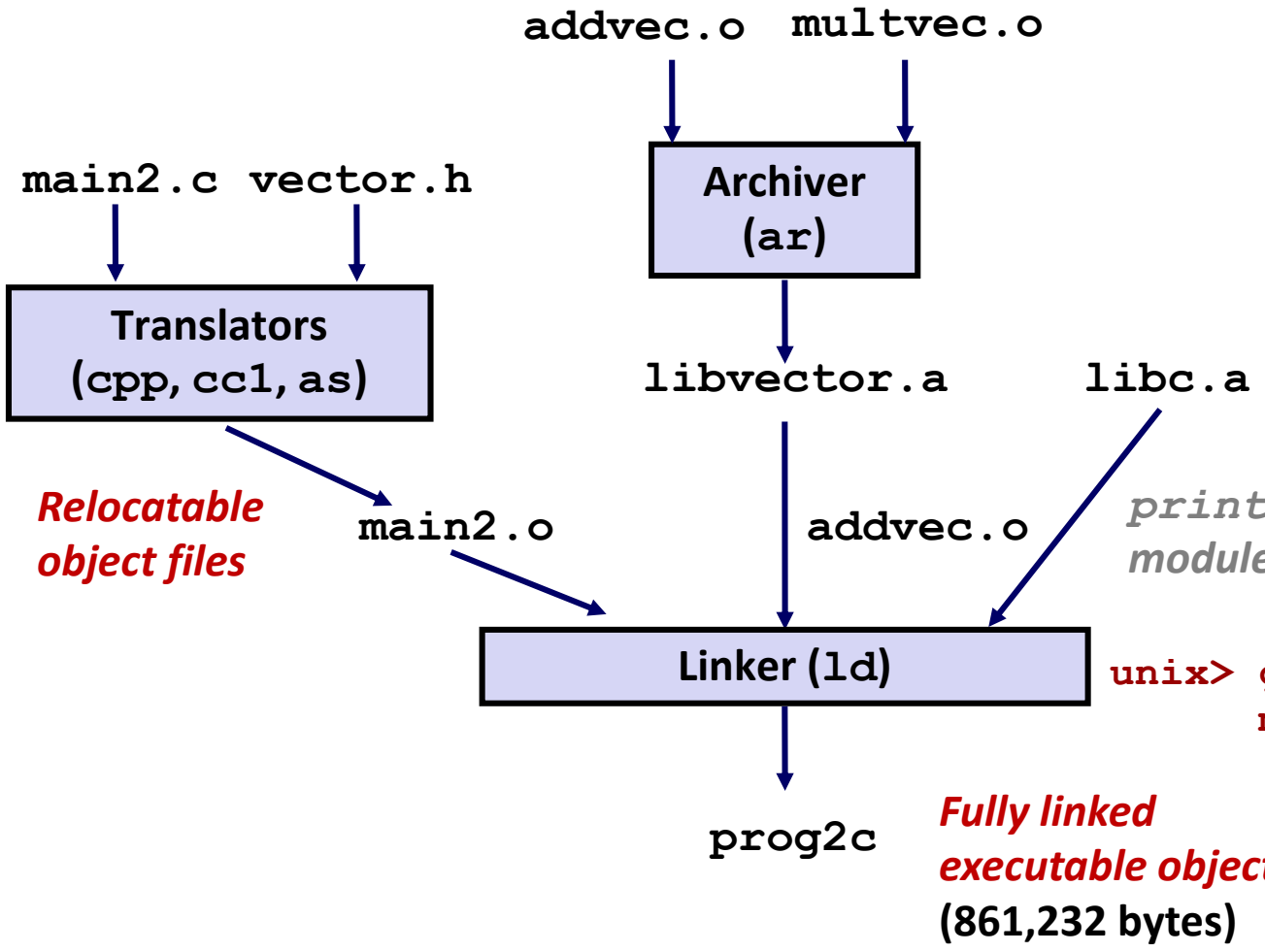
```c
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char**
argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n",
           z[0], z[1]);
    return 0;
}
```
main2.c

libvector.a

```c
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```
addvec.c

```c
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```
multvec.c

Linking with Static Libraries (2)



Using Static Libraries (1)

- Linker's algorithm for resolving external references:
  - Scan `.o` files and `.a` files in the command line order
  - During the scan, keep a list of the current unresolved references
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*
  - If any entries in the unresolved list at end of scan, then error

Using Static Libraries (2)

- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line

```
unix> gcc -static -o prog2c -L. -lvector main2.o
main2.o: In function `main':
main2.c:(.text+0x19): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

Shared Libraries (1)

- Static libraries have the following disadvantages:
  - Duplication in the stored executables
    - E.g., every function needs **`libc`**
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
    - Rebuild everything with **`glibc`**?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

Shared Libraries (2)

- 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 *runtime*
  - Also called: dynamic link libraries (**`DLL`**s) and **`.so`** files
- Shared library routines can be shared by multiple processes

Shared Libraries (3)

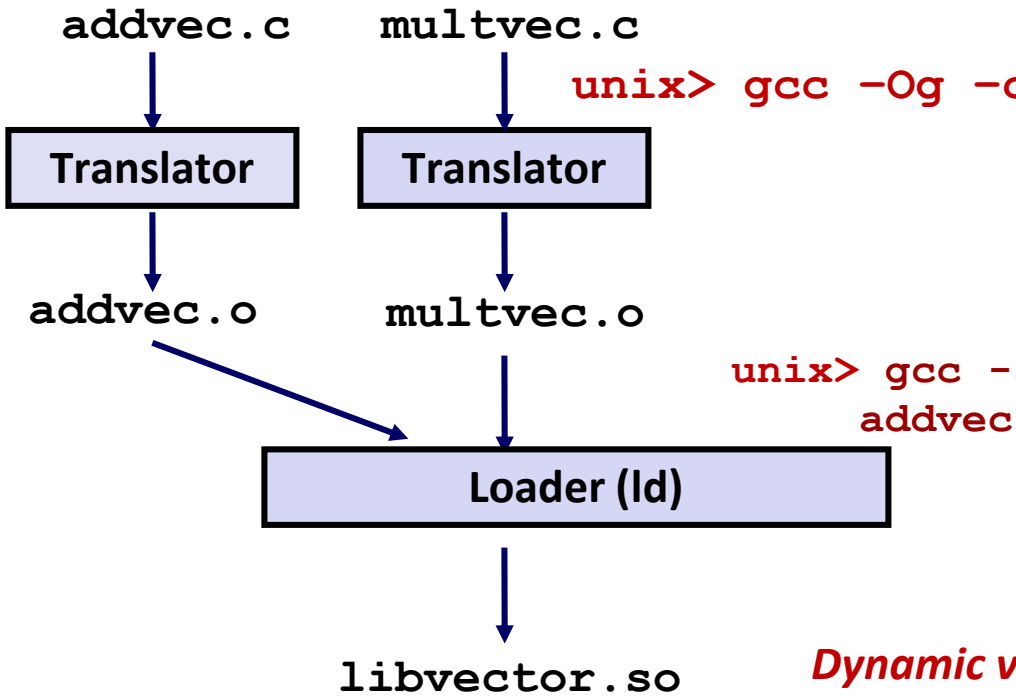
- Dynamic linking can occur when executable is first loaded and run (*load-time linking*)
  - Common case for Linux, handled automatically by the dynamic linker (**`ld-linux.so`**)
  - Standard C library (**`libc.so`**) usually dynamically linked
- Dynamic linking can also occur after program has begun (*runtime linking*)
  - In Linux, this is done by calls to the **`dlopen()`** interface
    - Distributing software
    - High-performance web servers
    - Runtime library interpositioning

Things Required for Dynamic Libraries

- **`.interp`** section
  - Specifies the dynamic linker to use (i.e., **`ld-linux.so`**)
- **`.dynamic`** section
  - Specifies the names, etc. of the dynamic libraries to use
  - Follow an example of **`prog`**
    - (NEEDED) Shared library: [**`libm.so.6`**]
- Where are the libraries found?
  - Use "**`ldd`**" to find out:

```
unix> ldd prog
  linux-vdso.so.1 =>  (0x00007ffcf2998000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

Creating Dynamic Libraries

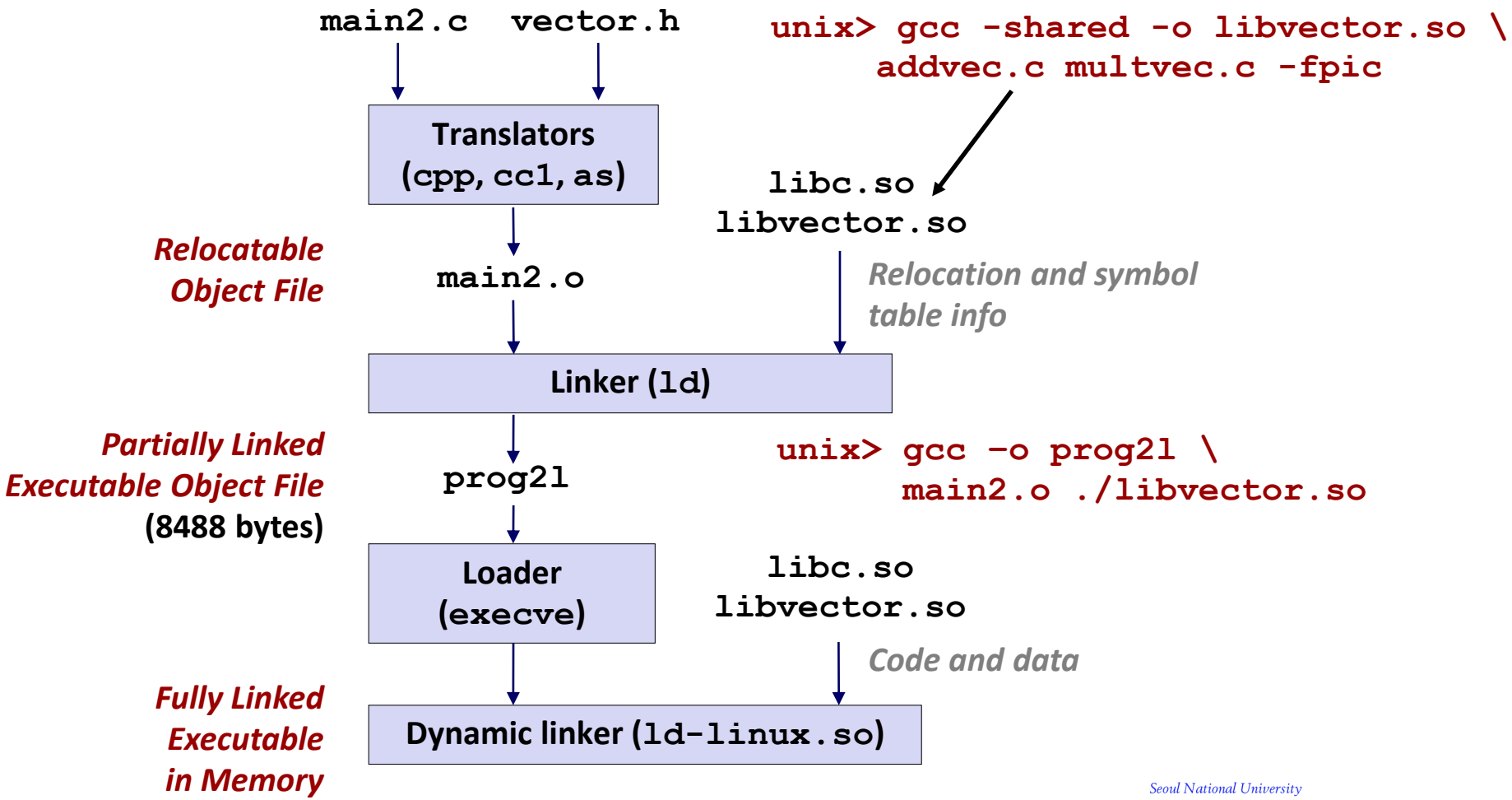



```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

```
unix> gcc -shared -o libvector.so \
      addvec.o multvec.o
```

Dynamic vector library

Dynamic Linking at Load-Time



Dynamic Linking at Runtme (1)

```c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char** argv)
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
  . . .
```

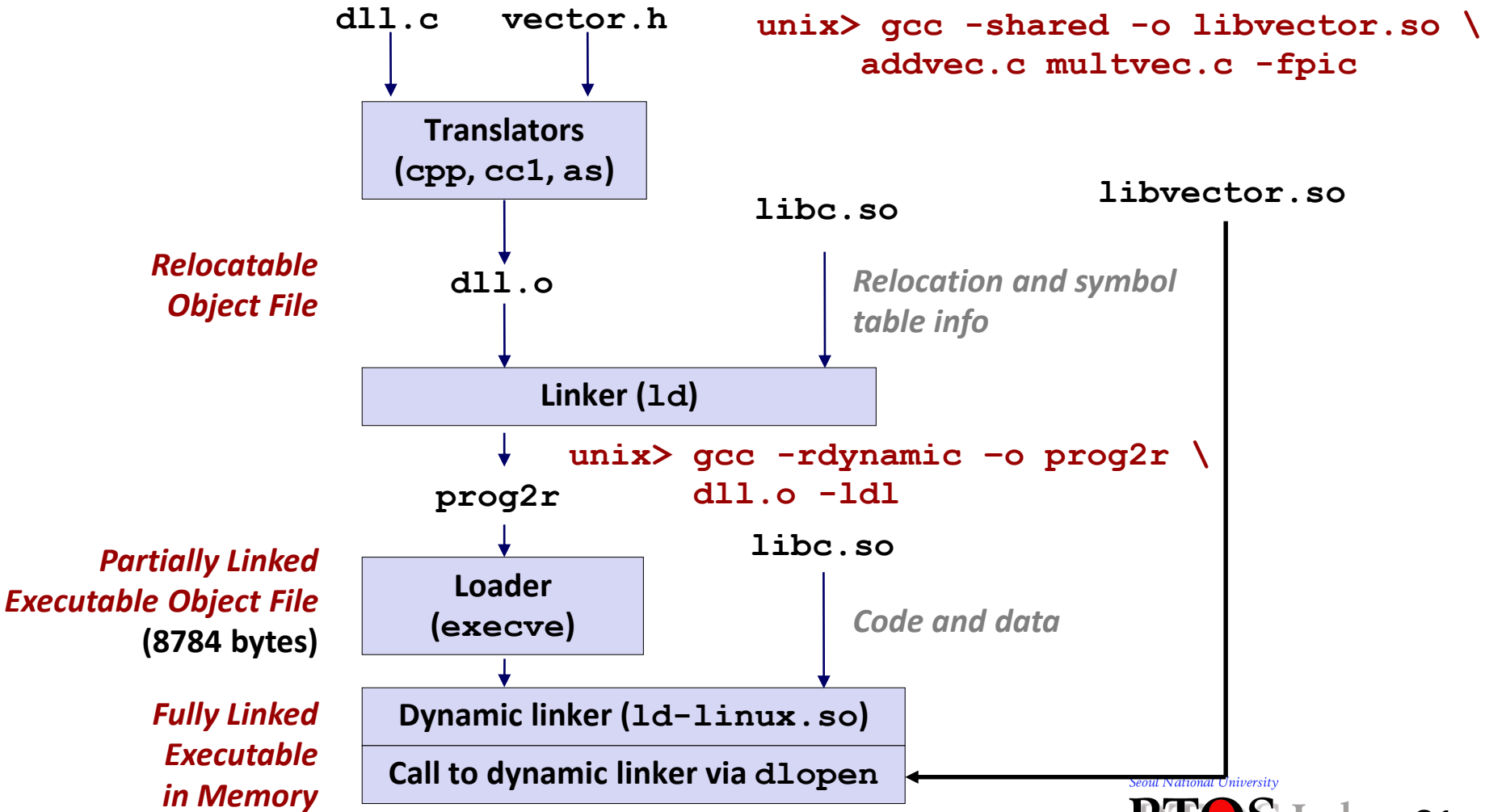
dll.c

Dynamic Linking at Runtme (2)

```
    ...

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

Dynamic Linking at Runtme (3)



Linking Summary

- Linking is a technique that allows programs to be constructed from multiple object files
- Linking can happen at different times in a program's lifetime:
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)
- Understanding linking can help you avoid nasty errors and make you a better programmer

Q & A