

Linking I: Basic Concepts and Procedures

COMP400727: Introduction to Computer Systems

Hao Li
Xi'an Jiaotong University

Today

- **Why Linking**
- **Basic Concepts and Procedures**
 - Basic Procedures
 - ELF formats
- **Procedures in Detail**
 - Symbol Resolution
 - Relocation
- **Walkthrough Example**

Understanding linking can help you avoid nasty errors and make you a better programmer.

Example C Program

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

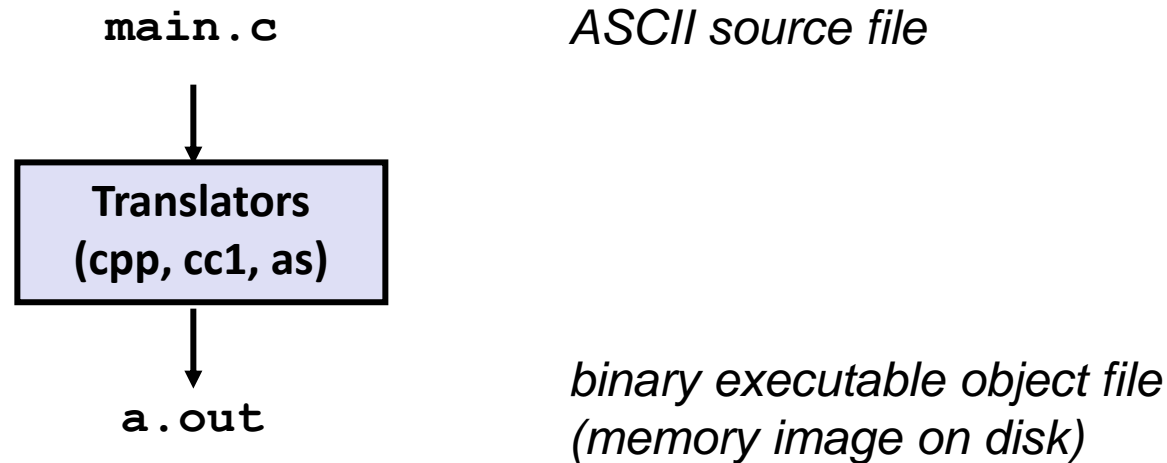
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}

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

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

main.c

Monolithic Compilation and its Problems



- **Huge lines of code in modern software**
 - **Complete (slow) compilation**
- **Multiple developers cooperation**
 - **Hard for code management**
- **Frequent changes**
 - **Frequent complete compilation**

Example C Program

```
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

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

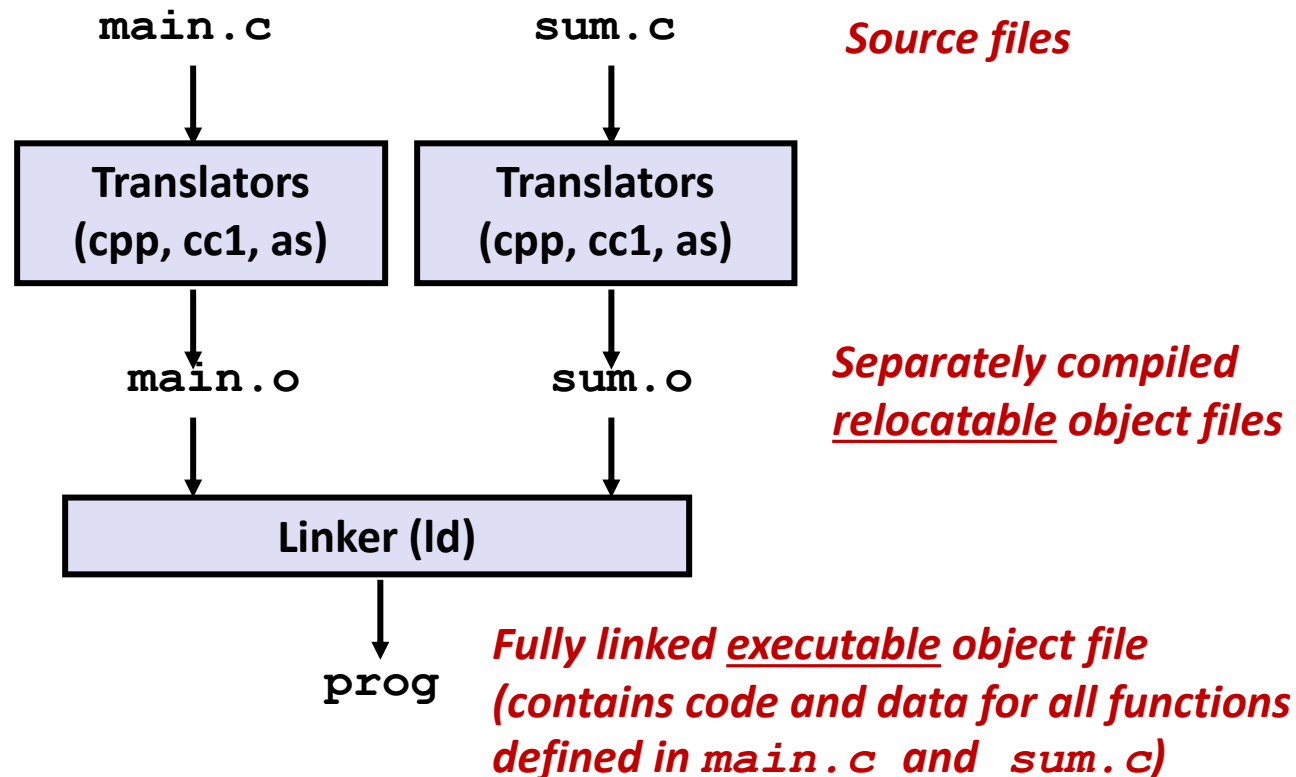
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Separate Compilation + Linking

- Programs are translated and linked using a *compiler driver*:

- `linux> gcc -Og -o prog main.c sum.c`
- `linux> ./prog`



Why Linking: Modularity

- **Program can be written as a collection of smaller source files, rather than one monolithic mass.**
 - Code from multiple developers can be isolated
- **Can build libraries of common functions**
 - e.g., Math library, standard C library

Why Linking: Efficiency

■ Time: Separate compilation. How does that save time?

- Change one source file, compile, and then relink.
- No need to recompile other source files.
- Can compile multiple files concurrently.

■ Space: Libraries. How do libraries save space?

- 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

Today

- Why Linking
- **Basic Concepts and Procedures**
 - Basic Procedures
 - Object files and ELF formats
- **Procedures in Detail**
 - Symbol Resolution
 - Relocation
- **Walkthrough Example**

Step 1: Symbol Resolution

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

Symbols in Example C Program

Definitions

```
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

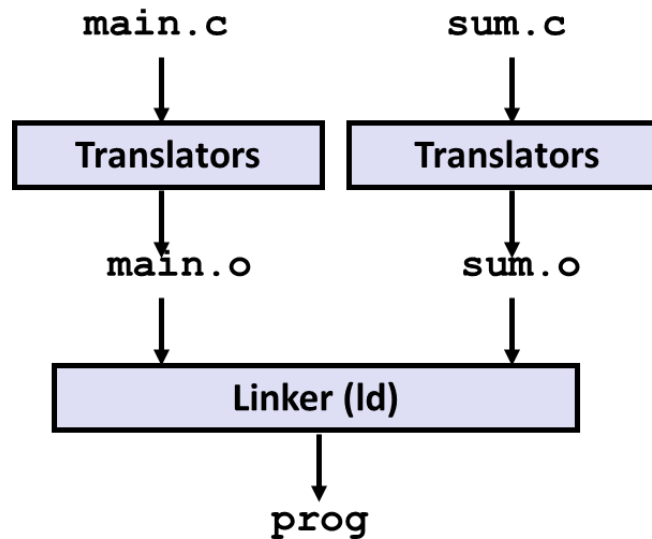
```
int sum(int *a, int n)  
{  
    int i, s = 0;  
  
    for (i = 0; i < n; i++) {  
        s += a[i];  
    }  
    return s;  
}
```

sum.c

Reference

Step 2: Relocation

- Merges separate code and data sections into single sections
- Relocates symbols from their relative locations to their final absolute memory locations in the executable.
- Updates all references to these symbols



Three Kinds of Object Files (Modules)

■ 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

■ Executable object file (`a.out` file)

- Contains code and data in a form that can be copied directly into memory and then executed.

■ 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 run-time.
- Called *Dynamic Link Libraries* (DLLs) by Windows

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

■ 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

■ .rodata section

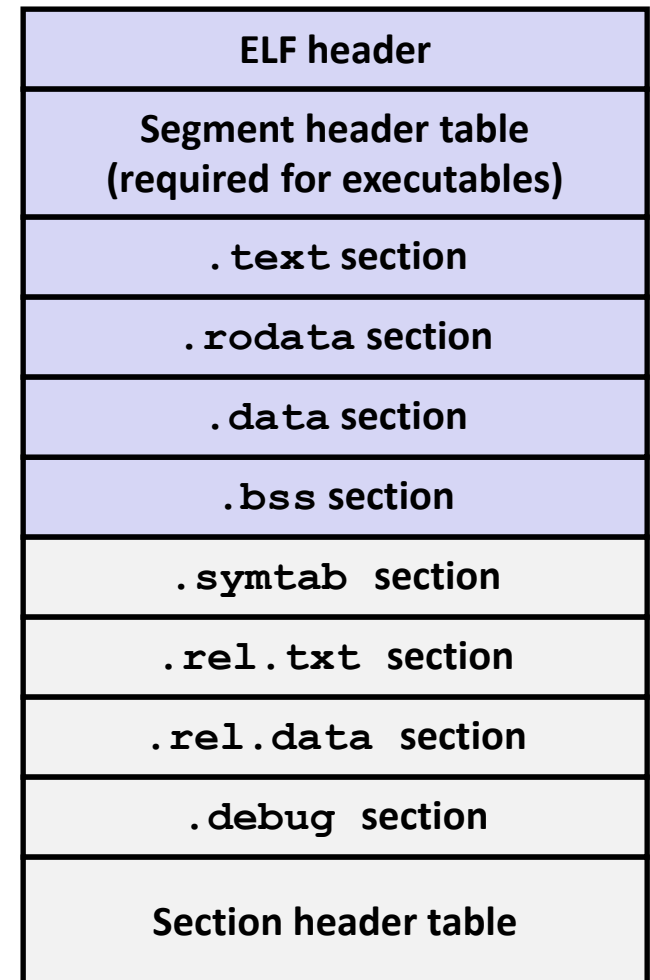
- Read only data: jump tables, string constants, ...

■ .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 (cont.)

- **.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.
- **.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

ELF header
Segment header table (required for executables)
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.text section
.rel.data section
.debug section
Section header table

0

Today

- **Why Linking**
- **Basic Concepts and Procedures**
 - Basic Procedures
 - Object files and ELF formats
- **Procedures in Detail**
 - Symbol Resolution
 - Relocation
- **Walkthrough Example**

ELF Symbol Tables

- Each relocatable object file has a symbol table in **.symtab section**
- A symbol table contains information about the symbols that are defined and referenced in the file
 - Symbol table contains an array of entries
- The symbol table records the linker symbols (next slide)

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 (Referenced global 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 the **static** attribute.
- **Local linker symbols are *not* local program variables**

Symbol Identification

Which of the following names will be in the symbol table of `symbols.o`?

`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;
}
```

Names:

- `incr`
- `foo`
- `a`
- `argc`
- `argv`
- `b`
- `main`
- `printf`
- `"%d\n"`

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```

Symbol Identification

```
• toney@DESKTOP-SGS2CTH:~/xjtu-ics/linking$ readelf -s sym.o
```

Symbol table '.symtab' contains 8 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	sym.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	.text
3:	0000000000000000	30	FUNC	LOCAL	DEFAULT	1	foo
4:	0000000000000000	0	SECTION	LOCAL	DEFAULT	5	.rodata
5:	0000000000000000	4	OBJECT	GLOBAL	DEFAULT	3	incr
6:	0000000000000001e	58	FUNC	GLOBAL	DEFAULT	1	main
7:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	printf

ELF Symbol Tables

■ Name

- byte offset into the string table that points to the null-terminated string name of the symbol.

■ Value (symbol's address)

- For relocatable modules
 - the value is an offset from the beginning of the section where the object is defined
- For executable object files
 - the value is an absolute run-time address.

■ Size

- the size (in bytes) of the object

ELF Symbol Tables

■ Type

- usually either data or function
- The symbol table can also contain entries
 - for the individual sections
 - for the path name of the original source file.
- So there are distinct types

■ Binding

- indicates whether the symbol is local or global

ELF Symbol Tables

■ Section

- Each symbol is assigned to some section of the object file, denoted by the section field, which is an index into the section header table.
- There are three special pseudosections that don't have entries in the section header table
 - ABS: symbols that should not be relocated
 - UNDEF: symbols that are referenced in this object module but defined elsewhere
 - COMMON: uninitialized data objects
 - these pseudosections exist only in relocatable object files and do not exist in executable object files

Local Symbols

■ Local non-static C variables vs. local static C variables

- local non-static C variables: stored on the stack
- local static C variables: stored in either `.bss`, or `.data`

```
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;
}
static-local.c
```

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`.

Linker Symbol Example

```
1.  extern int a ;
2.  int f()
3.  {
4.      static int x=1 ;    //x.1
5.      int b = 2;
6.      return x+b;
7.  }
8.
9.  int g()
10. {
11.     static int x = 1;    //x.2
12.     return x + a ;
13. }
```

Linker Symbol Examples

```
• toney@DESKTOP-SGS2CTH:~/xjtu-ics/linking$ readelf -s sym2.o
```

Symbol table '.symtab' contains 9 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FILE	LOCAL	DEFAULT	ABS	sym2.c
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	.text
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	.data
4:	0000000000000000	4	OBJECT	LOCAL	DEFAULT	3	x.1
5:	0000000000000004	4	OBJECT	LOCAL	DEFAULT	3	x.0
6:	0000000000000000	28	FUNC	GLOBAL	DEFAULT	1	f
7:	000000000000001c	24	FUNC	GLOBAL	DEFAULT	1	g
8:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	a

Step 1: Symbol Resolution

...that's defined here

Referencing
a global...

```
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
```

Defining
a global

Linker knows
nothing of val

Referencing
a global...

...that's defined here

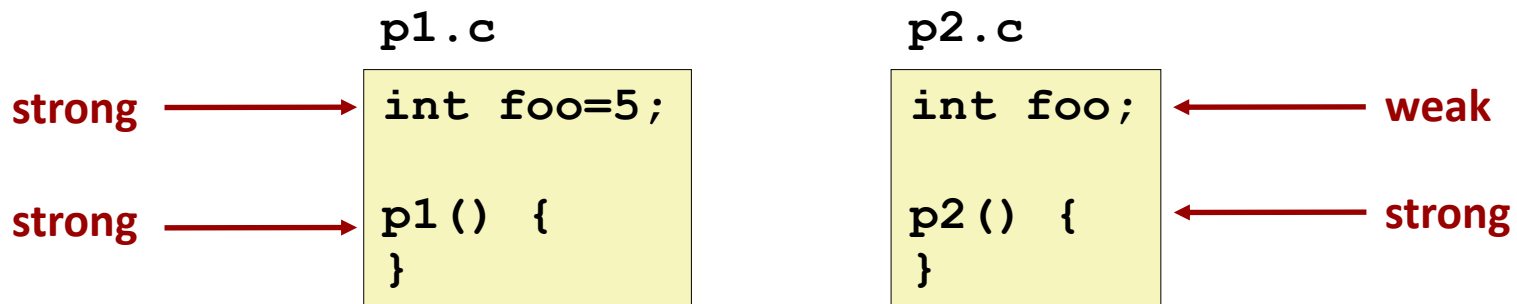
```
int sum(int *a, int n)  
{  
    int i, s = 0;  
    for (i = 0; i < n; i++) {  
        s += a[i];  
    }  
    return s;  
}  
  
sum.c
```

Linker knows
nothing of i or s

Resolve Duplicate Symbol Definitions

■ Program symbols are either *strong* or *weak*

- **Strong**: procedures and initialized globals
- **Weak**: uninitialized globals
 - Or ones declared with specifier **extern**



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`
- **Puzzles on the next slide**

Linker Puzzles

```
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.

Type Mismatch Example

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

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

mismatch-main.c

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

mismatch-variable.c

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

Global Variables

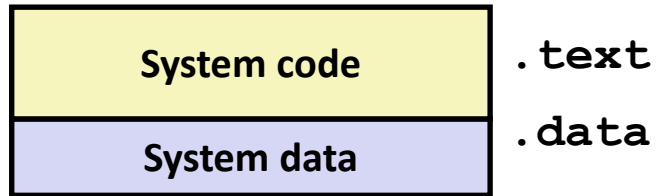
- **Avoid if you can**
- **Otherwise**
 - Use **static** if you can
 - Initialize if you define a global variable
 - Use **extern** if you reference an external global variable
 - Treated as weak symbol
 - But also causes linker error if not defined in some file

Step 2: Relocation

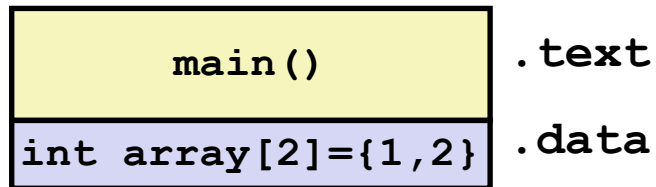
- **For each reference to an object with unknown location**
 - Assembler generates a relocation entry
 - Relocation entries for code are placed in .rel.text
 - Relocation entries for data are placed in .rel.data

Step 2: Relocation

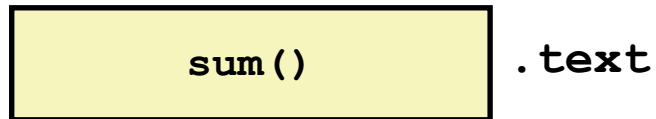
Relocatable Object Files



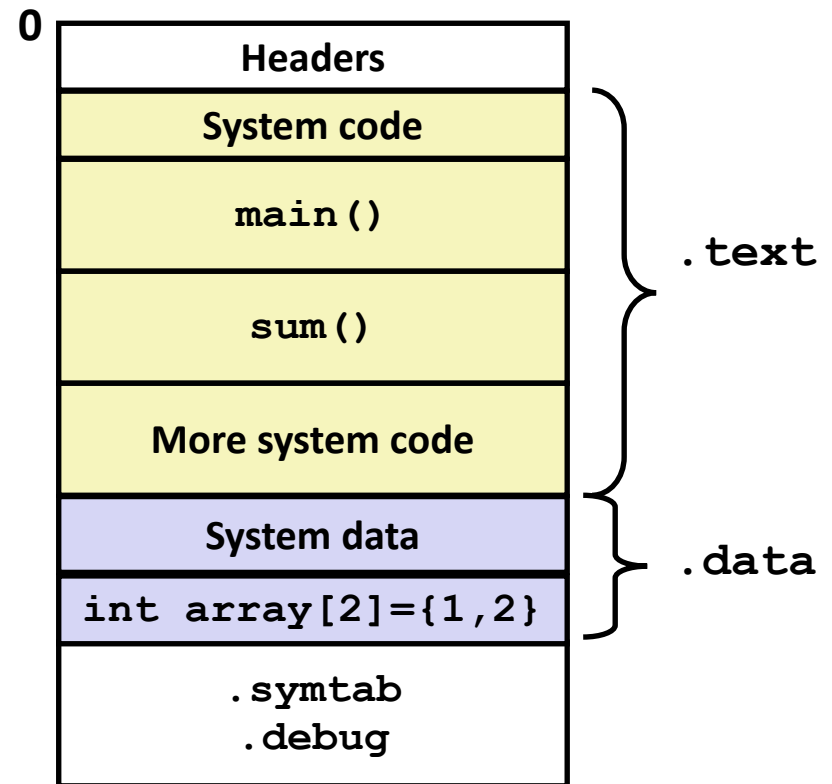
main.o



sum.o



Executable Object File



Relocation Entries

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

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

main.c

000000000000000000 <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	j1	4004f4 <sum+0xc>	
400501:	f3 c3	repz retq		

callq instruction uses PC-relative addressing for sum():

0x4004e8 = **0x4004e3** + **0x5**

Today

- **Why Linking**
- **Basic Concepts and Procedures**
 - Basic Procedures
 - Object files and ELF formats
- **Procedures in Detail**
 - Symbol Resolution
 - Relocation
- **Walkthrough Example**

Example

main.c	swap.c
<pre>1. /*main.c */ 2. void swap() ; 3. 4. int buf[2] = { 1, 2 }; 5. 6. int main() 7. { 8. swap() ; 9. return 0 ; 10. }</pre>	<pre>1. /*swap.c */ 2. extern int buf[] ; 3. 4. int *bufp0 = &buf[0] ; 5. int *bufp1 ; 6. 7. void swap() 8. { 9. int temp ; 10. 11. bufp1 = &buf[1] ; 12. temp = *bufp0 ; 13. *bufp0 = *bufp1 ; 14. *bufp1 = temp ; 15. }</pre>

Step 1: Symbol Resolution

main.c	swap.c
<pre>1. /*main.c */ 2. void swap() ; 3. 4. int buf[2] = { 1, 2 }; 5. 6. int main() 7. { 8. swap(); 9. return 0 ; 10. }</pre>	<pre>1. /*swap.c */ 2. extern int buf[] ; 3. 4. int *bufp0 = &buf[0] ; 5. int *bufp1 ; 6. 7. void swap() 8. { 9. int temp ; 10. 11. bufp1 = &buf[1] ; 12. temp = *bufp0 ; 13. *bufp0 = *bufp1 ; 14. *bufp1 = temp ; 15. }</pre>

Global

External

Local

Step 2: Relocation (main.o)

Define (main)

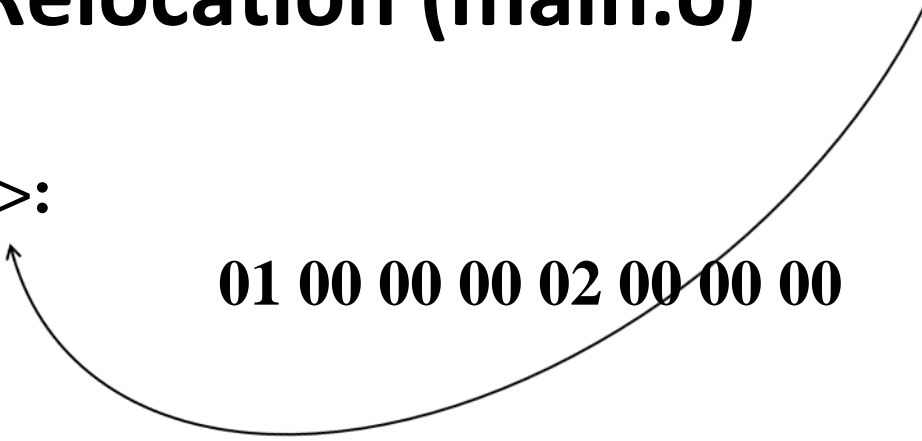
1	00 <main>:	
2	00: 55	push %rbp
3	01: 48 89 e5	mov %rsp,%rbp
4	04: b8 00 00 00 00	mov \$0x0,%eax
5	09: e8 00 00 00 00	callq e <main+0xe>
6	0e: b8 00 00 00 00	mov \$0x0,%eax
7	13: 5d	pop %rbp
8	14: c3	retq

Reference
(swap)

Step 2: Relocation (main.o)

Define (buf)

1 **00** <buf>:
2 00: 01 00 00 00 02 00 00 00

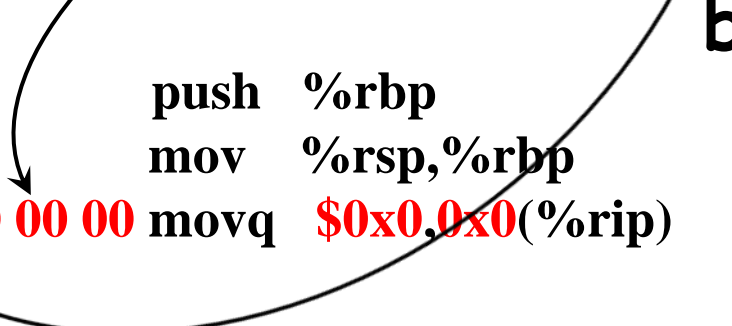


bufp1

References

Step 2: Relocation (swap.o)

1. **00**<swap>:
2. 00: 55 push %rbp
3. 01: 48 89 e5 mov %rsp,%rbp
4. 04: 48 c7 05 **00 00 00 00** movq \$0x0,0x0(%rip)
5. 0b: **00 00 00 00** ←
6. 0f: 48 8b 05 **00 00 00 00** mov 0x0(%rip),%rax bufp1= &buf[1]
7. 16: 8b 00 mov (%rax),%eax get bufp0
8. 18: 89 45 fc mov %eax,-0x4(%rbp)
9. 1b: 48 8b 05 **00 00 00 00** mov 0x0(%rip),%rax get bufp0
10. 22: 48 8b 15 **00 00 00 00** mov 0x0(%rip),%rdx get bufp1
11. 29: 8b 12 mov (%rdx),%edx
12. 2b: 89 10 mov %edx,(%rax) *bufp0 = *bufp1
13. 2d: 48 8b 05 **00 00 00 00** mov 0x0(%rip),%rax get bufp1
13. 34: 8b 55 fc mov -0x4(%rbp),%edx
14. 37: 89 10 mov %edx,(%rax)



Step 2: Relocation (swap.o)

Define (buf)

```
15. 39: 90      nop
16. 3a: 5d      pop    %rbp
17. 3b: c3      retq
```

Reference
(buf)

1 00<bufp0>:

2 00:

00 00 00 00 00 00 00 00 00

1 00<bufp1>:

Loading Executable Object Files

Executable Object File

0	ELF header
	Program header table (required for executables)
	.init section
	.text section
	.rodata section
	.data section
	.bss section
	.symtab
	.debug
	.line
	.strtab
	Section header table (required for relocatables)

