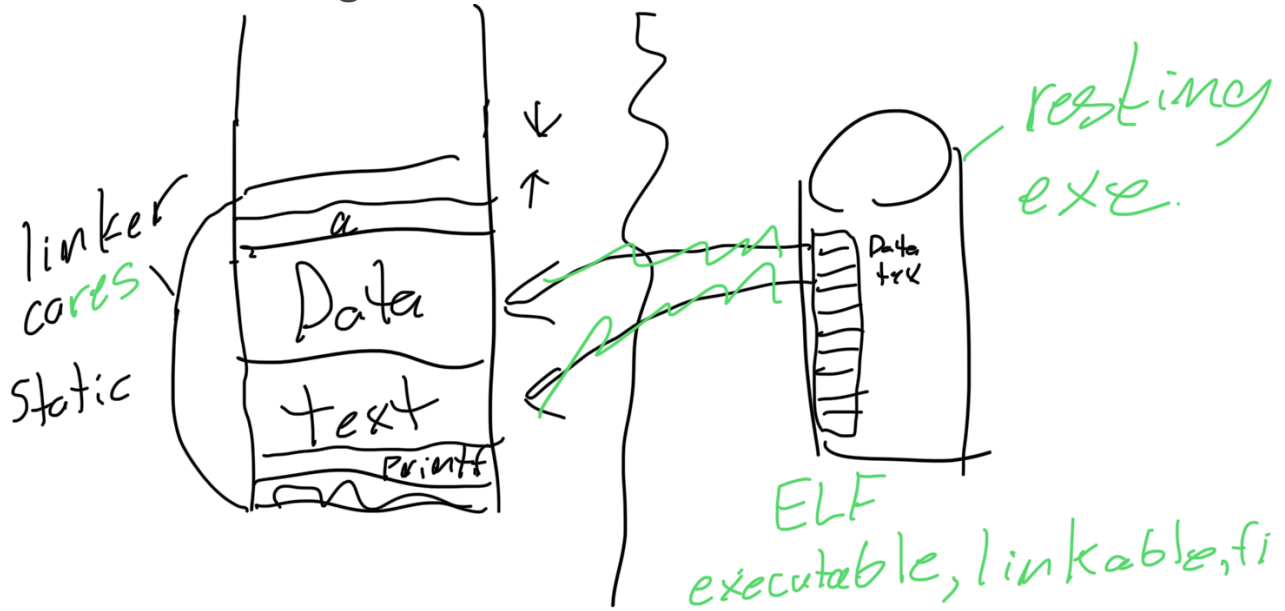


## Slides 3 - Linking



`printf()` → `jal printf` - external reference  
*\$a0 = "Hello world"*

`printf.c` → `printf()`

Symbol address never changes  
↳ Stored in static memory

# Executable and Linkable Format

- ELF header
  - Word size
  - Byte ordering
  - Machine type (e.g., IA32)
  - ELF header size
  - Object file type
    - relocatable, executable, or shared
  - Offset of the section header
  - Number of entries
- Section header table
  - Section locations and sizes
  - A fixed size entry for each

ELF header	}	Sections
.text		
.rodata		
.data		
.bss		
.symtab		
.rel.text		
.rel.data		
.debug		
.line		
.strtab		
Section header table		Describes object file sections

main.c → main.o

a.out

.dll, .so

a.out is just a file, when executed,  
it loads data into memory

# Executable and Linkable Format

- **.text section**
  - Machine instructions
- **.data section**
  - Initialized global variables
  - Static local variables
- **.bss (Block storage start) section**
  - Uninitialized global variables
- **.symtab**
  - Symbol table
  - Functions and global variables

ELF header	}	Sections
.text		
.rodata		
.data		
.bss		
.symtab		
.rel.text		
.rel.data		
.debug		
.line		
.strtab		
Section header table		Describes object file sections

## ELF Format

- **.rel.text section** *- offset from zero*
  - Relocation info for .text section
- **.rel.data section**
  - Relocation info for .data section
- **.debug section (gcc -g)**
  - Symbolic debugging information
- **.line section**
  - Mapping line # in source code and machine instruction in the .text section
- **.strtab section**
  - String table for .symtab and .debug

ELF header	}	Sections
.text		
.rodata		
.data		
.bss		
.symtab		
.rel.text		
.rel.data		
.debug		
.line		
.strtab		
Section header table		Describes object file sections

# ELF Symbol Examples

	Num:	Value	Size	Type	Bind	Ndx	Name
/* main.c */							
void swap();	8:	0	8	OBJECT	GLOBAL	3	buf
int buf[2] = {1, 2};	9:	0	21	FUNC	GLOBAL	1	main
int main() {	10:	0	0	NOTYPE	GLOBAL	UND	swap
swap();							
return 0;							
}							
/* swap.c */							
extern int buf[];	8:	0	4	OBJECT	GLOBAL	3	bufp0
int *bufp0 = &buf[0];	9:	0	0	NOTYPE	GLOBAL	UND	buf
int *bufp1;	10:	8	4	OBJECT	GLOBAL	COM	bufp1
void swap(){	11:	0	59	FUNC	GLOBAL	1	swap
int temp;							
bufp1 = &buf[1];							
temp = *bufp0;							
*bufp0 = *bufp1;							
*bufp1 = temp;							
}							

## Additional Example

```

<swap.c>
extern int buf[];

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

static void incr () {
    static int count=0;
    int i=0;
    count++;
}

void swap () {
    int temp;

    incr();
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}

```

buf:	external global, undefined
bufp0:	global, .data
bufp1:	local, .bss
count:	local, .data
incr:	local, .text
swap:	global, .text

objdump -S \*.o — gives instructions

Cross-compiler

1. for C global

# Strong and Weak Symbols

- Symbols are either *strong* and *weak*
  - Strong – Functions and initialized global variables
  - Weak – uninitialized global variables
  - Compiler exports global symbols to the assembler (weak or strong)
  - Assembler encode symbols information in .symtab

	p1.c		p2.c	
Strong	← int foo = 5		int foo;	→ Weak
Strong	← int p1 () {		int p2 () {	→ Strong
Weak?	← int a;		int b = 5	→ Strong?
	}		}	

If there's two of the same strong symbols, there will be a linking error

## Symbol Resolution

- **Rule 1** – Multiple strong symbols are not allowed
- **Rule 2** – Given a strong symbol and multiple weak symbols, choose the strong symbol
- **Rule 3** – Given multiple weak symbols, choose **any of the weak symbols**

# Relocation

- After the symbol resolution
  - Linker associates each symbol reference in the code with exactly one symbol definition
  - Linker knows the exact sizes of the code and data sections
- 1. Relocating sections and symbol definitions
  - Merging all sections of the same type
  - Assigning run-time memory address
- 2. Relocating symbol reference within sections
  - Update External references to point to the correct address

## Relocation Example

p1.c

```
int e = 7;

int main () {
    int r = a();
    exit (0);
}
```

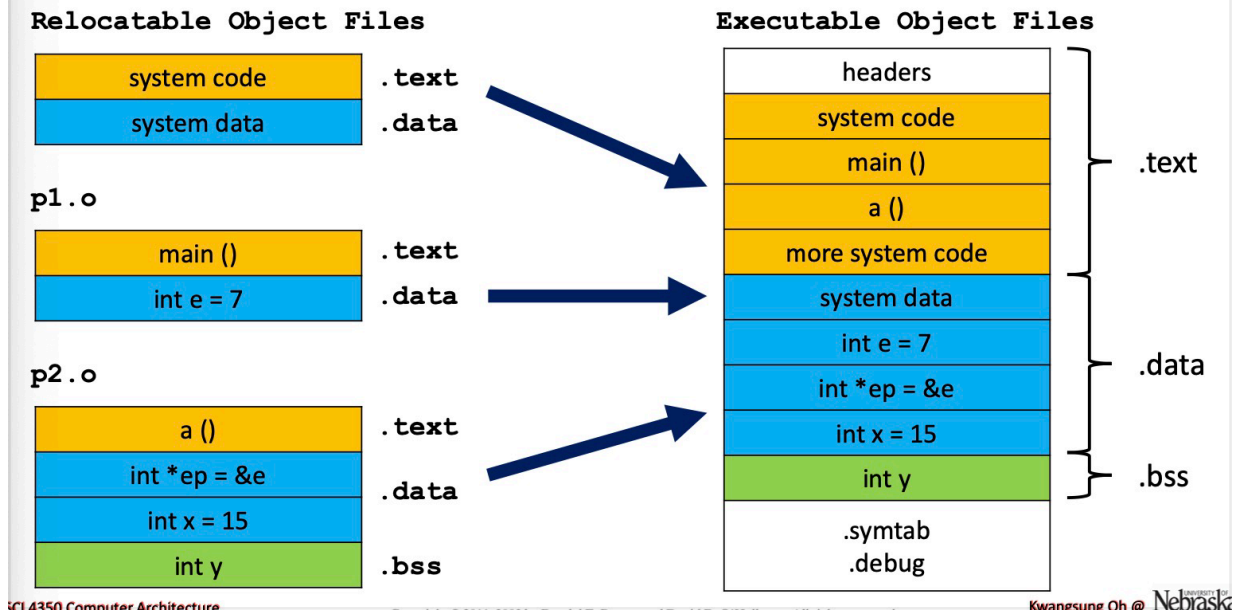
p2.c

```
extern int e;

int *ep = &e;
int x = 15;
int y;

int a () {
    return *ep+x+y;
}
```

# Relocating Sections



## Executable Object Format

- **Fully linked** (relocated)
  - no .rel sections needed
- **Segment header Table**
  - Page size
  - Virtual and physical address of memory segments (sections)
  - Segment sizes
- **.init**
  - Called by programs' initialization code

ELF header	Read-only segment (Code)
Segment header table	
.init	
.text	Read/Write (Data)
.rodata	
.data	
.bss	Not loaded into memory
.symtab	
.debug	
.line	
.strtab	
Section header table	

# Why Shared Libraries?

- **Disadvantages** of static libraries
  - **Duplicated common functions** in many programs (e.g., printf)
  - **Space inefficient** for duplicated codes in text segment
  - **Requirement of relinking** all programs if a function changes
- Shared libraries (\*.so or \*.dll)
  - Dynamically loaded and linked at run-time
  - Exactly one shared library for a particular library
  - Sharing libraries in memory by different processes
  - By loader (ld-linux.so) at **load-time**
  - By user (dlopen() function) at **run-time**

Dynamic library can be used at runtime