Linker

Why linkers?

- Understanding linkers will help you build large programs.
- Understanding linkers will help you avoid dangerous programming errors.
- Understanding linking will help you understand how language scoping rules are implemented.
- Understanding linking will help you understand other important systems concepts.

Why linkers?

- **Modularity**
  - Large program can be written as a collection of smaller files, rather than one monolithic mass.
  - Can build libraries of common functions (more on this later) e.g., Math library, standard C library
- **Efficiency**
  - Time:
    - Change one source file, compile, and then re-link
    - No need to recompile other source files
  - Space:
    - Libraries of common functions can be put in a single file...
    - Yet executable files and running memory images contain only code for the functions they actually use

What does a linker do?

Step 1: Symbol resolution

- Programs define and reference symbols (variables and functions)
- Symbol definitions are stored (by compilers) in a *symbol table*
  - Symbol table is an array of struct
  - Each entry includes name, type, size, and location of symbol
- Linker associates each symbol reference with exactly one symbol definition

What does a linker do?

Step 2: 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
- Updates all references to these symbols to reflect their new positions

Three kinds of object files

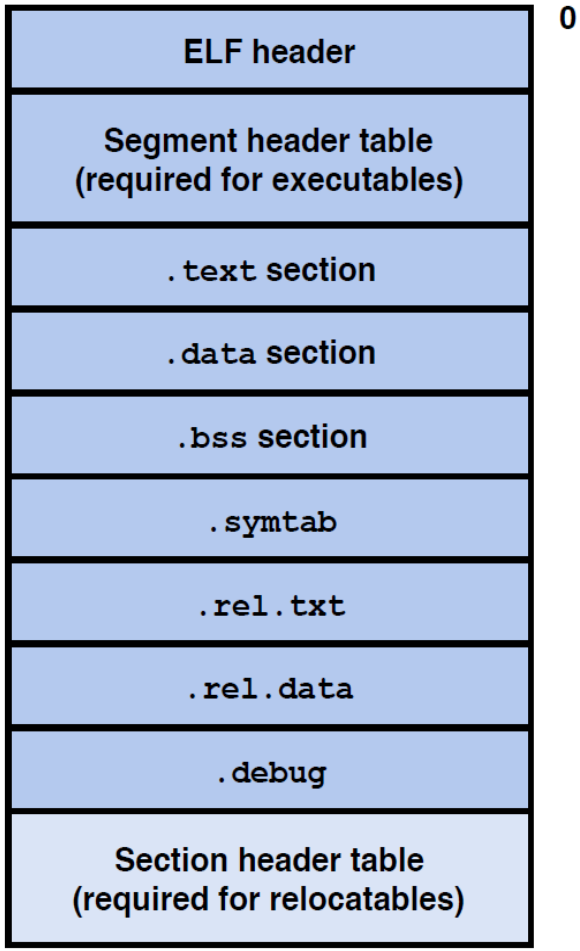
- Generated by compilers and assemblers
  - Relocatable object file
    - Contains code and data in a form that can be combined with other relocatable object files to form an executable
    - Each .o file is produced from exactly one source (.c) file
  - Shared object file
    - Special type of relocatable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

Executable and Linkable Format (ELF)

- Standard binary format for object files
- Derives from AT&T System V Unix (Common Object File Format – COFF)
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - Executable object files
  - Relocatable object files (.o),
  - Shared object files (.so)
- Generic name: ELF binaries

ELF object file format

- **ELF header**
  - Magic number, type (.o, exec, .so),machine, byte ordering, offset of section header table, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- ***.text* section**
  - Code
- ***.data* section**
  - Initialized (static) data
- ***.bss* section**
  - Uninitialized (static) data
  - Originally an IBM 704 assembly instruction; "Block Started by Symbol" ("Better Save Space")
  - Has section header but occupies no space



ELF object file format

- *.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

Linker Symbols

Every relocatable object module has a symbol table

- **Global symbols**
  - Symbols defined by a module 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 a module but defined by some other module
- **Local symbols**
  - Symbols that are defined and referenced exclusively by a module
  - E.g. C functions and variables defined with the static attribute
  - *Local linker symbols are not local program variables* (no symbols for local nonstatic program variables that are managed at runtime)

Examples

Example 1

a.c

```
int x = 3;
static y=4;
int main()
{
fun();
y++;
return 0;
}
```

Def of a GLOBAL symbol **x**

Ref to external symbol **fun()**

b.c

```
extern int x;

void fun(void)
{
x++;
P(x);
}
```

Ref to external symbol **x**

Def of a GLOBAL symbol **fun()**

a.o symbol table

Symbol	.symtab entry ?	Symbol Type	Module where defined	Section
x	yes	GLOBAL	a.o	.data (D)
y	yes	LOCAL	a.o	.data (d)
fun	yes	EXTERN	b.o	.text (t)
main	yes	GLOBAL	a.o	.text (T)

Example 1

a.c

```
int x = 3;
static y=4;
int main()
{
fun();
y++;
return 0;
}
```

Def of a Local symbol **x**

Ref to external symbol **fun()**

b.c

```
extern int x;

void fun(void)
{
x++;
P(x);
}
```

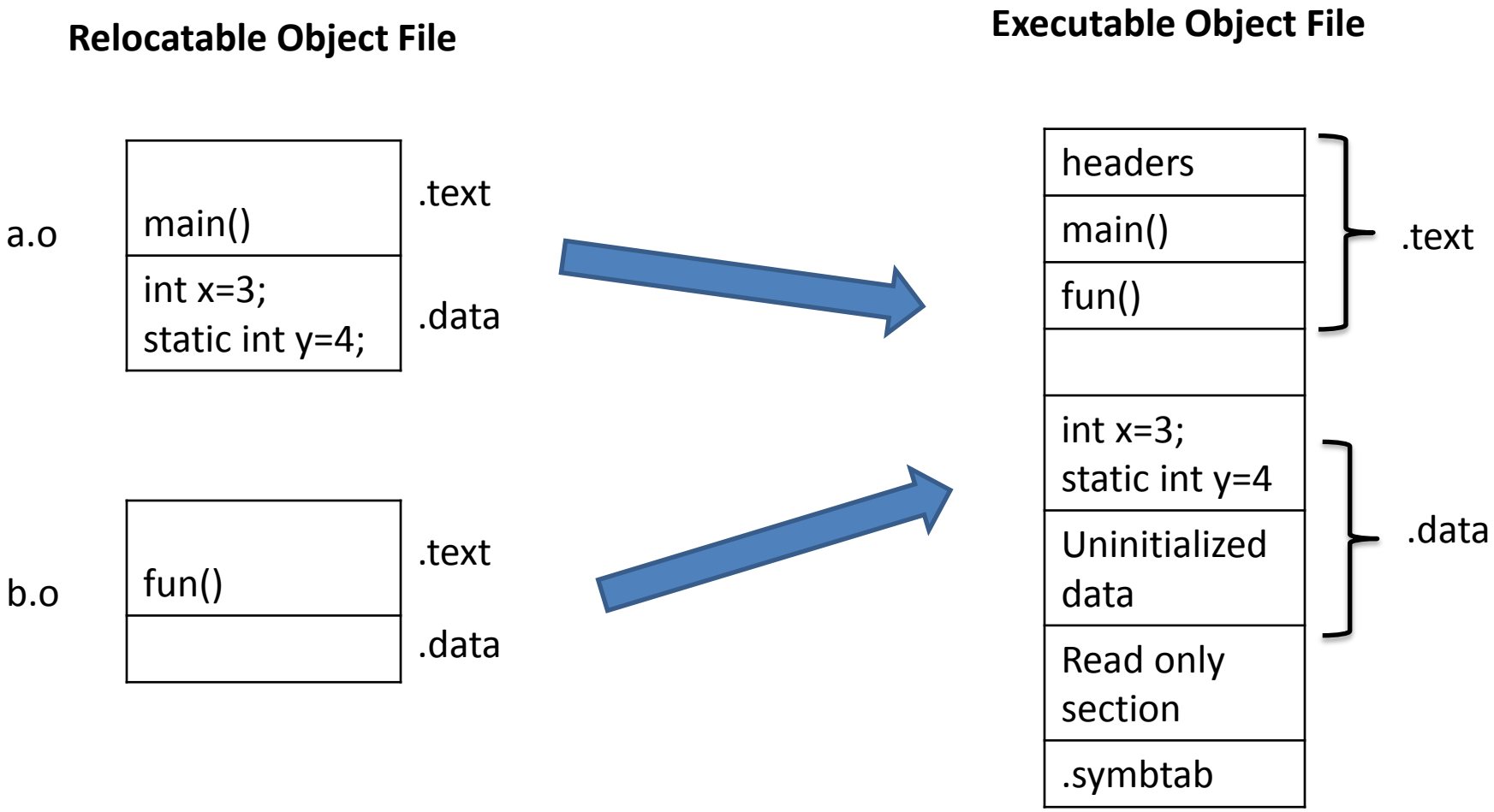
Ref to external symbol **x**

Def of a Local symbol **fun()**

b.o symbol table

Symbol	.symtab entry ?	Symbol Type	Module where defined	Section
x	yes	EXTERN	a.o	.data (d)
fun	yes	GLOBAL	b.o	.text (T)

Example 1


Relocatable Object File
Executable Object File
a.o
main()
int x=3;
static int y=4;
.text
.data
b.o
fun()
.text
.data
headers
main()
fun()
int x=3;
static int y=4
Uninitialized data
Read only section
.symbtab
.text
.data

Example 2:

main.c

```
void swap();
int buf[2] = {1, 2};
static var=20;
int main()
{
swap();
return 0;
}
```

Def of a Global symbol buf

Ref to external symbol swap

swap.c

```
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;
void swap()
{
int temp;
bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
```

Ref to external symbol buf

Def of a Local symbol bufp0, bufp1

temp is a Local variable not a symbol.

main.o

Symbol	.symtab entry ?	Symbol Type	Module where defined	Section
buf	Yes	GLOBAL	main.o	.data
var	Yes	LOCAL	main.o	.data
main	Yes	GLOBAL	main.o	.text
swap	Yes	EXTERNAL	swap.o	.text

Example 2:

main.c

```
void swap();
int buf[2] = {1, 2};
static var=20;
int main()
{
swap();
return 0;
}
```

Def of a Local symbol buf

Ref to external symbol swap

swap.c

```
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;
void swap()
{
int temp;
bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
```

Ref to external symbol buf

Def of a Local symbol bufp0, bufp1

temp is a Local variable not a symbol.

swap.o

Symbol	.symtab entry ?	Symbol Type	Module where defined	Section
buf	Yes	EXTERN	main.o	.data
bufp0	Yes	GLOBAL	swap.o	.data
bufp1	Yes	GLOBAL	swap.o	.bss
swap	Yes	GLOBAL	swap.o	.text
temp	no	**temp** A local variable; not a local symbol.		

GNU BinUtils

- **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.

GNU BinUtils

- **readelf:** Displays the complete structure of an object file, including all of the information encoded in the ELF header; subsumes the functionality of size and nm.
- **objdump:** The mother of all binary tools. Can display all of the information in an object file. Its most useful function is disassembling the binary instructions in the .text section.
- **ldd:** Lists the shared libraries that an executable needs at run time.