Linking

Introduction to Computer Systems
15th Lecture, Nov. 7, 2024

Instructors:

Class 1: Chen Xiangqun, Liu Xianhua

Class 2: Guan Xuetao

Class 3: Lu Junlin

Outline of Linking

- **Linking: combining object files into programs**
  - Object files
  - Linking mechanism
    - Symbols and symbol resolution
    - Relocation
- **Libraries**
- **Dynamic linking, loading & execution**
- **Library inter-positioning**

Example C Program

```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;
}
```

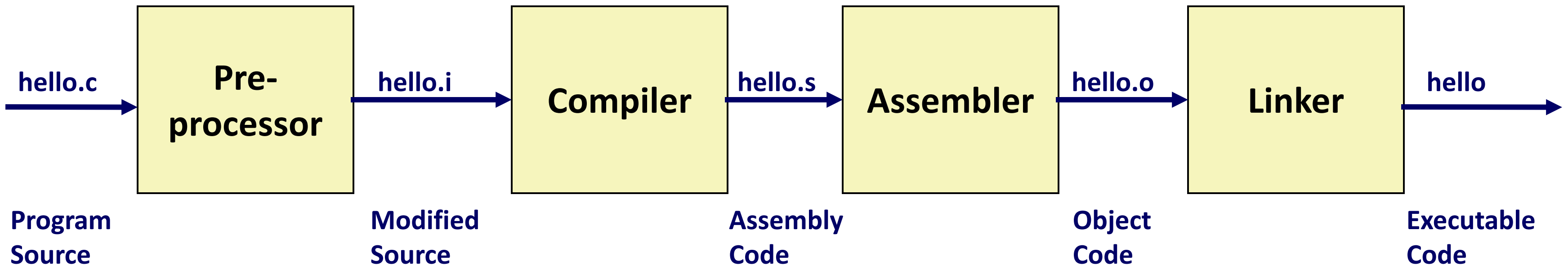
main.c

```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, GCC as an Example

- **Gcc is the compiler driver in compilation toolchain.**
- **Gcc invokes several other compilation phases**
  - cpp, the preprocessor
  - cc1, the compiler
  - as/gas, the assembler
  - ld, the linker
- **What does each one do? What are their outputs?**



Preprocessor

- **First, *gcc* compiler driver invokes *cpp* to generate expanded source**
  - Preprocessor just does text substitution/ gcc with option “-E”
  - Converts the C source file to another C source file
  - Expands “#” directives

```
#include <stdio.h>
#define FOO 4
int main(){
     printf("hello, world %d\n", FOO);
}
```

```
…
extern int printf (const char *__restrict
    __format, ...);
…
int main() {
 printf("hello, world %d\n", 4);
}
```

Compiler

- **Next, *gcc* invokes *cc1* to generate assembly code**
  - Translates high-level C code into assembly

```
…
extern int printf (const char *__restrict
    __format, ...);
…
int main() {
 printf("hello, world %d\n", 4);
}
```

```
        .section        .rodata
    .LC0:
        .string "hello, world %d\n"

        .text
    main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $4, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        popq    %rbp
        ret
```

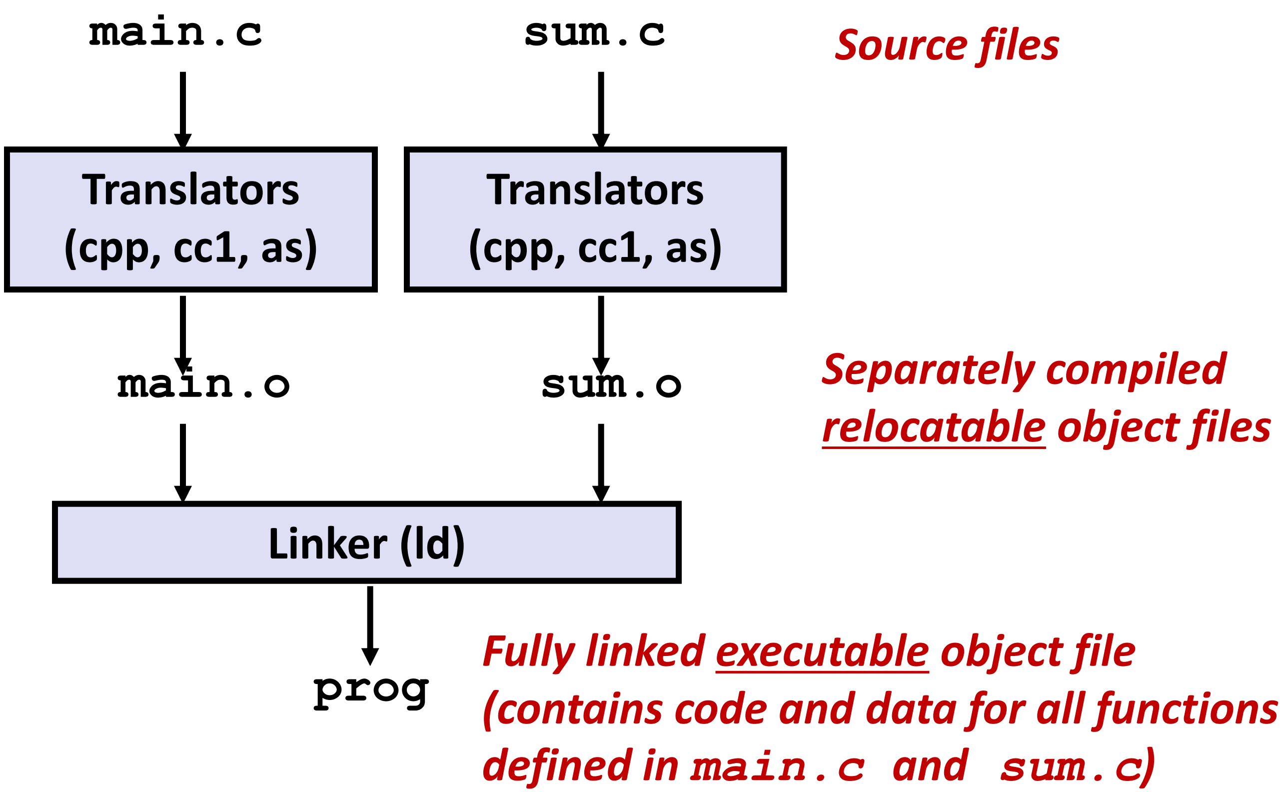
Assembler

- **Furthermore, *gcc* invokes *gas* to generate object code**
  - Translates assembly code into binary object code

```
#  readelf -a hello | grep rodata
    [10] .rodata           PROGBITS         0000000000495d40  00095d40

#  readelf -a hello | grep -E "GLOBAL.* main"
    1591: 0000000000401190    31 FUNC    GLOBAL DEFAULT    6 main

#  readelf -x .rodata hello
    Hex dump of section '.rodata':
    0x00495d40 01000200 68656c6c 6f2c2077 6f726c64 ....hello, world
    0x00495d50 2025640a 00464154 414c3a20 6b65726e  %d..FATAL: kern

#  objdump -d hello
    0000000000401190 <main>:
    401190:       55                      push   %rbp
    401191:       48 89 e5                mov    %rsp,%rbp
    401194:       be 04 00 00 00          mov    $0x4,%esi
    401199:       bf 44 5d 49 00          mov    $0x495d44,%edi
    40119e:       b8 00 00 00 00          mov    $0x0,%eax
    4011a3:       e8 d8 0e 00 00          callq  402080 <_IO_printf>
    4011a8:       b8 00 00 00 00          mov    $0x0,%eax
    4011ad:       5d                      pop    %rbp
    4011ae:       c3                      retq
    4011af:       90                      nop
```

(Static) Linking

- Programs are translated and linked using a *compiler driver*:
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`

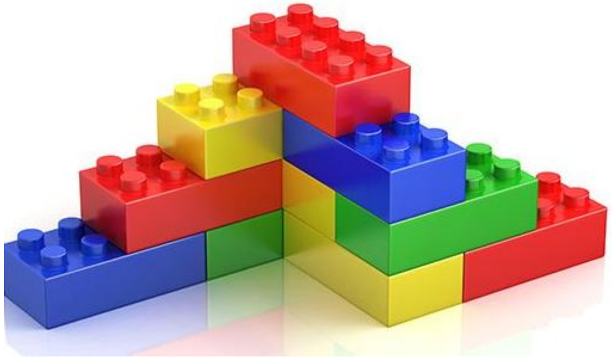


Why Linkers?

- **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 (more on this later)
    - e.g., Math library, standard C library



Why Linkers? (cont)

- **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

What Do Linkers Do?

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.**

What Do Linkers Do? (cont)

- **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.

Let's look at these two steps in more detail....

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**

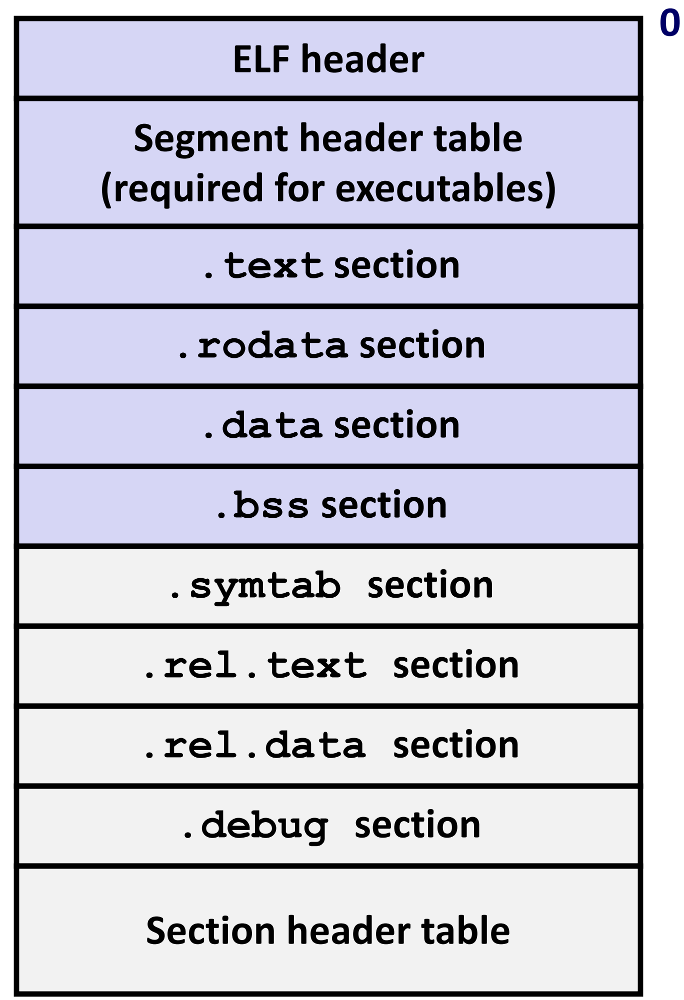
- **One unified format for**
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Shared object files (`.so`)

- **Generic name: ELF binaries**

- **First appeared in System V Release 4 Unix, c. 1989**
- **Linux switched to ELF c. 1995, BSD later at c. 1998-2000**

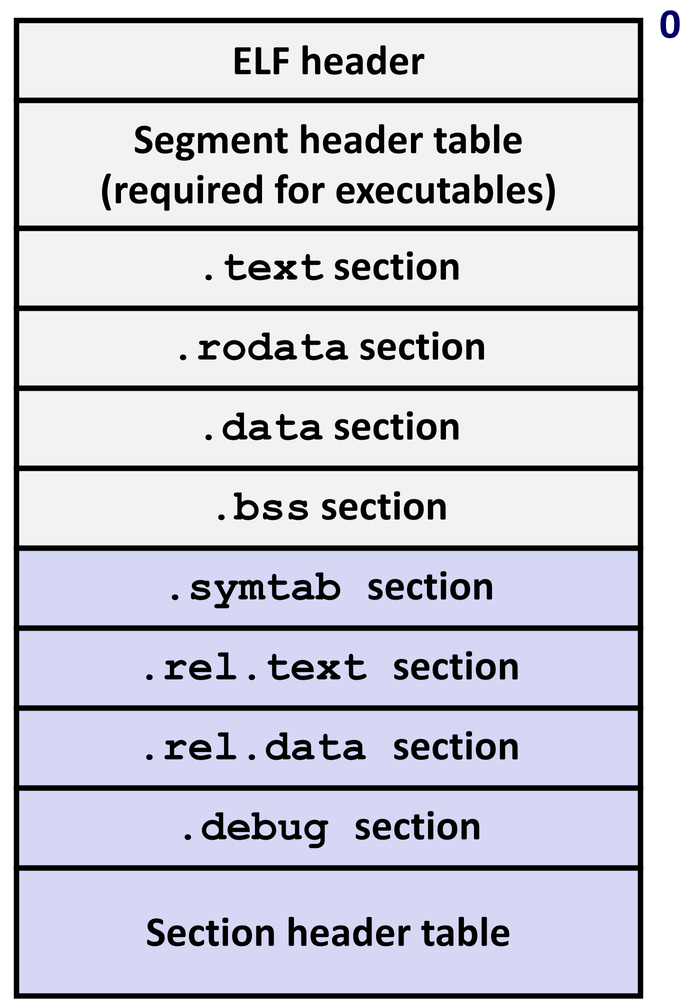
ELF Object File Format

- **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



Parallel Views of a ELF File

- ***Program header table/Segments*** is used to build a process image (execute a program); relocatable files don't need it.
- Files used during linking must have a ***section header table/Sections***.

ELF Header
Program header table *optional*
Section 1
...
Section n
...
Section header table required

Linking View

ELF Header
Program header table required
Segment 1
Segment 2
Segment 3
...
Section header table *optional*

Execution View

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 the **`static`** attribute.
  - **Local linker symbols are *not* local program variables**

Step 1: Symbol Resolution

...that's defined here

Referencing a global...

```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

Defining a global

Linker knows nothing of `val`

Referencing a global...

...that's defined here

Linker knows nothing of `i` or `s`

How Linker Resolves Duplicate Symbol Definitions (such as sum, array)?

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`.

How Linker Resolves Duplicate Symbol Names

- **Program symbols are either *strong* or *weak***
  - ***Strong***: procedures and initialized global variables
  - ***Weak***: uninitialized global variables
    - Or ones declared with specifier **`extern`**
- **Compiler exports such kind of information and assembler encodes it implicitly in the symbol table of ELF files.**

`p1.c`

strong → `int foo=5;`

strong → `p1() {`

```
int foo=5;

p1() {
}
```

`p2.c`

`int foo;` ← weak

`p2() {` ← strong

```
int foo;

p2() {
}
```

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`

Linker Puzzles

Module 1	Module 2	Result
`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.

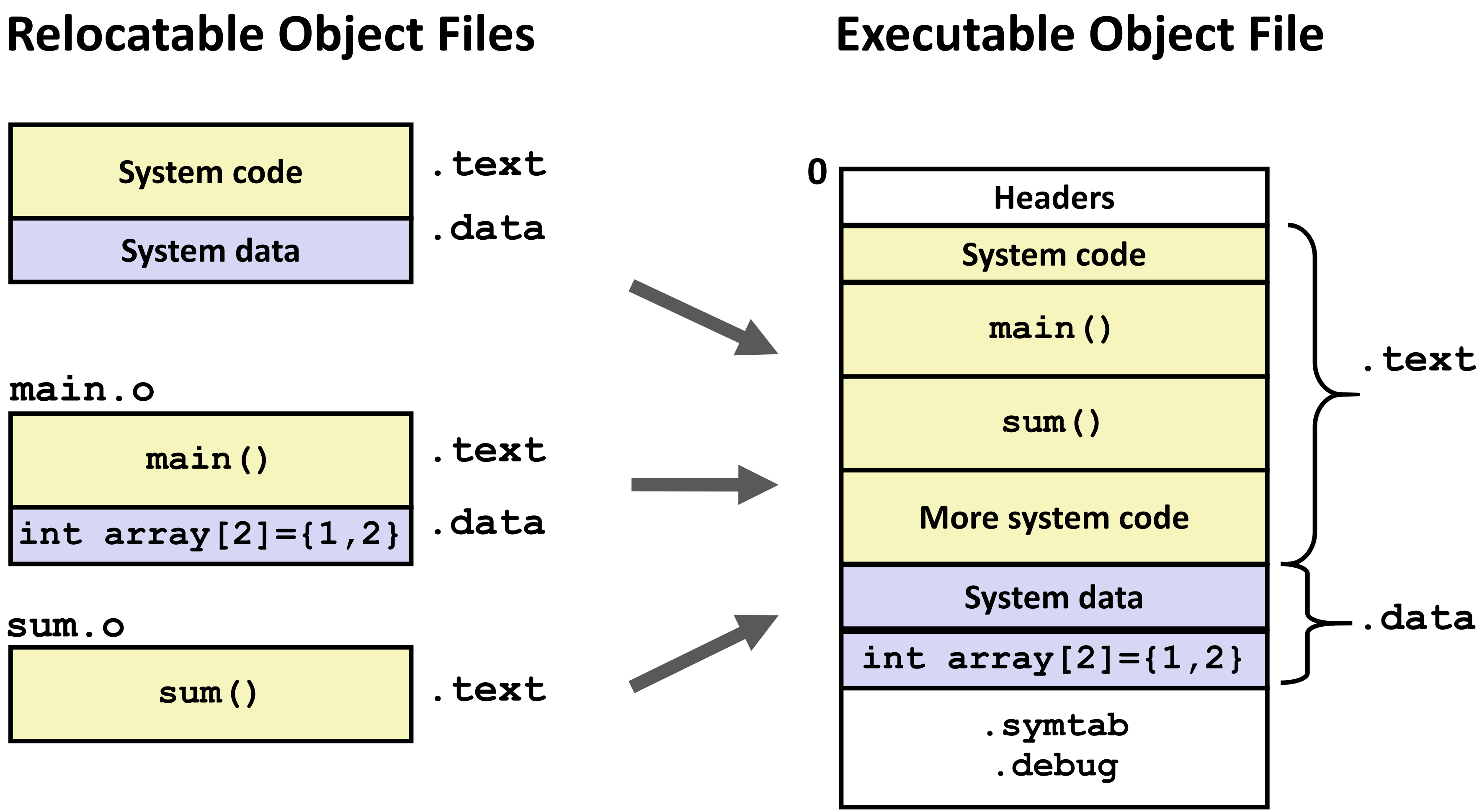
Important: Linker does not do type checking.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Rules for avoiding type mismatches

- **Avoid global variables as much as possible**
- **Use `static` as much as possible**
- **Declare *everything* that's not `static` in a header file**
  - Make sure to include the header file everywhere it's relevant
  - Including the files that define those symbols
- **Always put `extern` on declarations in header files**
  - Unnecessary but harmless for function declarations
  - Avoids the quirky behavior of extern-less global variables
- **Always write (void) when a function takes no arguments**
  - `extern void no_args(void);`
  - Leaving out the **`void`** means "I'm *not saying* what argument list this function takes." Turns off argument type checking!

Step 2: Relocation



2-Step Relocation in Static Linking

- **Relocating sections and symbol definitions**
  - Merges all sections of the same type into a new aggregate section of the same type.
  - Assigns run-time memory addresses to
    - The new aggregate section.
    - Each section defined by the input modules.
    - Each symbol defined by the input modules.
- **Relocating symbol references with sections**
  - Modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses.
  - It relies on data structures in the relocatable modules known as **relocation entries.**

Relocation Entries

- A ***relocation entry*** generates from reference with unknown location.

```
/* Relocation table entry with addend
(in section of type SHT_RELA). */
typedef struct
{
Elf64_Addr r_offset; /* Address */
Elf64_XWord r_info; /* Relocation type and symbol index
*/
Elf64_Sxword r_addend; /* Addend */
} Elf64_Rela;
#define ELF64_R_SYM(i)  ((i) >> 32)
#define ELF64_R_TYPE(i) ((i) & 0xffffffff)
```

  - ***r_offset*** is the section offset of the reference that will be modified.
  - ***ELF_64_R_SYM*** identifies the symbol that the reference should point to.
  - ***ELF_64_R_TYPE*** tells the linker how to modify the new reference.
  - ***r_addend*** is a constant used for offset adjustment in some kind of relocation.

Two Most Basic Relocation Types

- **R_X86_64_PC32**
  - Relocates a reference that uses a 32-bit PC-relative address.
- **R_X86_64_32/R_X86_64_32S**
  - Relocates a reference that uses a 32-bit absolute address.

```
for each section s {
    foreach relocation entry r {
        refptr = s + r.offset;  /* ptr to reference to be relocated */

        /* Relocate a PC-relative reference */
        if (r.type == R_X86_64_PC32){
            refaddr = ADDR(s) + r.offset; /* ref's run-time address */
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
        }

        /* Relocate an absolute reference */
        if (r.type ==R_X86_64_32)
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
    }
}
```

Relocation Entries

```
int array[2] = {1, 2};
int main(int argc, char** argv){
    int val = sum(array, 2);
    return val;
}
```

main.c

```
# readelf -r main.o
Relocation section '.rela.text' at offset 0x560 contains 2 entries:
Offset        Info          Type            Sym.Value         Sym.Name+Addend
000000000015  00080000000a  R_X86_64_32     0000000000000000  array + 0
00000000001a  000a00000002  R_X86_64_PC32   0000000000000000  sum - 4

Relocation section '.rela.eh_frame' at offset 0x590 contains 1 entries:
Offset        Info          Type            Sym.Value         Sym.Name+Addend
000000000020  000200000002  R_X86_64_PC32   0000000000000000  .text + 0
```

offset

type

symbol name & addend

Totally 3 symbols to be relocated.

Relocation Entries (in main.o)

```
int array[2] = {1, 2};
int main(int argc, char** argv){
    int val = sum(array, 2);
    return val;
}
```

main.c

```
# readelf -r main.o
Relocation section '.rela.text' at offset 0x560 contains 2 entries:
Offset        Info          Type           Sym.Value         Sym.Name+Addend
000000000015  00080000000a  R_X86_64_32    0000000000000000  array + 0
00000000001a  000a00000002  R_X86_64_PC32  0000000000000000  sum - 4
```

Dear Linker,

Please patch the .rela.text section at offsets 0x15. Patch in a 32-bit value like following steps. When you determine the addr of .data, compute [addr of array] + [addend, which equals 0] and place the result at the prescribed place.

Sincerely,
Assembler

Relocation Entries (in main.o)

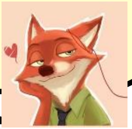
```
int array[2] = {1, 2};
int main(int argc, char** argv){
    int val = sum(array, 2);
    return val;
}                                   main.c
```

```
# readelf -r main.o
Relocation section '.rela.text' at offset 0x560 contains 2 entries:
Offset       Info         Type            Sym.Value         Sym.Name+Addend
000000000015 00080000000a R_X86_64_32     0000000000000000 array + 0
00000000001a 000a00000002 R_X86_64_PC32   0000000000000000 sum - 4
```

Dear Linker,

Please patch the .rela.text section at offsets 0x1a. Patch in a 32-bit "PC-relative" value like following steps. When you determine the addr of sum, compute [addr of sum] + [addend, which equals -4] – [addr of section + offset] and place the result at the prescribed place.

Sincerely,
Assembler



Relocation Entries (in sum.o)

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

```
# readelf -r sum.o
Relocation section '.rela.eh_frame' at offset 0x4f8 contains 1 entries:
Offset        Info          Type           Sym.Value         Sym.Name+Addend
000000000020  000200000002  R_X86_64_PC32  0000000000000000  .text + 0
```

offset

type

symbol name & addend

1 symbol to be relocated (.text)

Original Object File of main.o

```
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:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 20             sub    $0x20,%rsp
   8:   89 7d ec                mov    %edi,-0x14(%rbp)
   b:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
   f:   be 02 00 00 00          mov    $0x2,%esi
  14:   bf 00 00 00 00          mov    $0x0,%edi         # %edi = &array
                        15: R_X86_64_32 array            # Relocation entry
  19:   e8 00 00 00 00          callq  1e <main+0x1e>    # sum()
                        1a: R_X86_64_PC32 sum-0x4        # Relocation entry
  1e:   89 45 fc                mov    %eax,-0x4(%rbp)
  21:   8b 45 fc                mov    -0x4(%rbp),%eax
  24:   c9                      leaveq
  25:   c3                      retq
```

Original Object File of sum.o

```
int sum(int *a, int n){
```

```
0000000000000000 <sum>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
   8:   89 75 e4                mov    %esi,-0x1c(%rbp)
   b:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  12:   c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  19:   eb 1d                   jmp    38 <sum+0x38>
  1b:   8b 45 f8                mov    -0x8(%rbp),%eax
  1e:   48 98                   cltq
  20:   48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  27:   00
  28:   48 8b 45 e8             mov    -0x18(%rbp),%rax
  2c:   48 01 d0                add    %rdx,%rax
  2f:   8b 00                   mov    (%rax),%eax
  31:   01 45 fc                add    %eax,-0x4(%rbp)
  34:   83 45 f8 01             addl   $0x1,-0x8(%rbp)
  38:   8b 45 f8                mov    -0x8(%rbp),%eax
  3b:   3b 45 e4                cmp    -0x1c(%rbp),%eax
  3e:   7c db                   jl     1b <sum+0x1b>
  40:   8b 45 fc                mov    -0x4(%rbp),%eax
  43:   5d                      pop    %rbp
  44:   c3                      retq
```

.text=0xbabe00

main.o

```
0000000000000000 <main>:
   0:   55
   1:   48 89 e5
   4:   48 83 ec 20
   8:   89 7d ec
   b:   48 89 75 e0
   f:   be 02 00 00 00
  14:   bf 00 00 00 00
  19:   e8 00 00 00 00
  1e:   89 45 fc
  21:   8b 45 fc
  24:   c9
  25:   c3
```

sum.o

```
0000000000000000 <sum>:
   0:   55
   1:   48 89 e5
   4:   48 89 7d e8
   8:   89 75 e4
   b:   c7 45 fc 00 00 00 00
  12:   c7 45 f8 00 00 00 00
  19:   eb 1d
  1b:   8b 45 f8
  1e:   48 98
  20:   48 8d 14 85 00 00 00
  27:   00
  28:   48 8b 45 e8
  2c:   48 01 d0
  2f:   8b 00
  31:   01 45 fc
  34:   83 45 f8 01
  38:   8b 45 f8
  3b:   3b 45 e4
  3e:   7c db
  40:   8b 45 fc
  43:   5d
  44:   c3
```

executable

```
0000000000babe00 <_start>:
...
0000000000babf18 <main>:
 babf18:   55
 babf19:   48 89 e5
 babf1c:   48 83 ec 20
 babf20:   89 7d ec
 babf23:   48 89 75 e0
 babf27:   be 02 00 00 00
 babf2c:   bf 10 fe ca 00
 babf31:   e8 0a 00 00 00
 babf36:   89 45 fc
 babf39:   8b 45 fc
 babf3c:   c9
 babf3d:   c3
0000000000babf40 <sum>:
 babf40:   55
 babf41:   48 89 e5
 babf44:   48 89 7d e8
 babf48:   89 75 e4
 babf4b:   c7 45 fc 00 00 00 00
 babf52:   c7 45 f8 00 00 00 00
 babf59:   eb 1d
 babf5b:   8b 45 f8
 babf5e:   48 98
 babf60:   48 8d 14 85 00 00 00
 babf67:   00
 babf68:   48 8b 45 e8
 babf6c:   48 01 d0
 babf6f:   8b 00
 babf71:   01 45 fc
 babf74:   83 45 f8 01
 babf78:   8b 45 f8
 babf7b:   3b 45 e4
 babf7e:   7c db
 babf80:   8b 45 fc
 babf83:   5d
 babf84:   c3
```

```
Disassembly of section .data:
0000000000cafe00 <__data_start>:
   ...
0000000000cafe10 <array>:
  cafe10:   01 00
  cafe12:   00 00
  cafe14:   02 00
```

.data=0xcafe00

addr_of_array=0xcafe10

Using the value 0xcafe10 to modify the content here

addr_of_main=0xbabf18

addr_of_sum=0xbabf40

offset = 0x1a

addend = -4

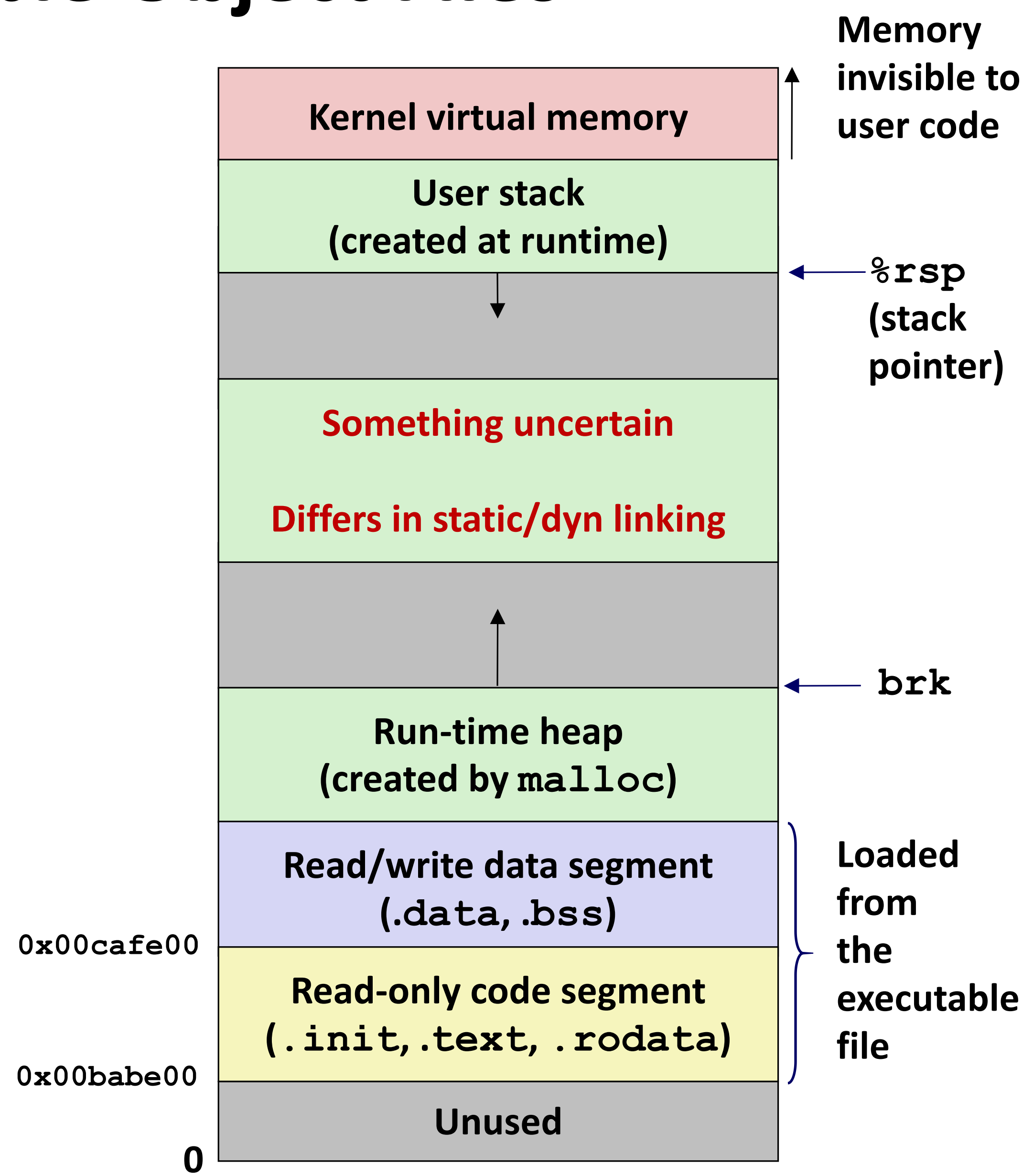
refptr
= 0xbabf18 + 0x1a
= 0xbabf32

*refptr(content)
=0xbabf40-4-0xbabf32
=0x0a

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)



Libraries: Packaging a Set of Functions

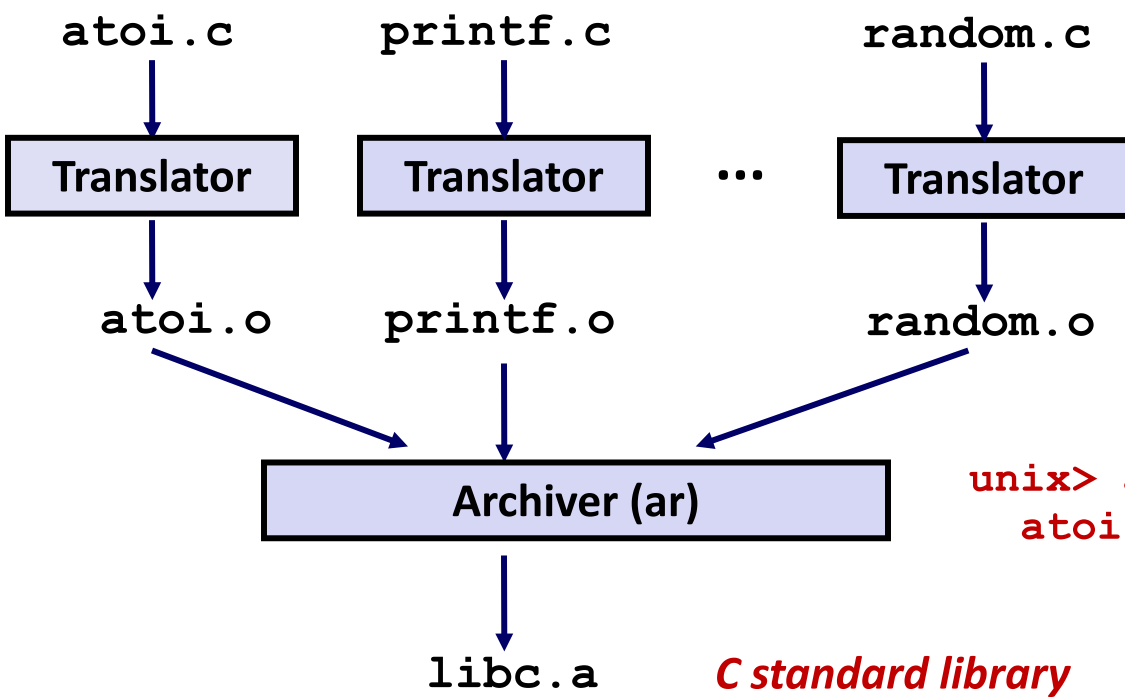
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **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

Old-fashioned Solution: Static Libraries

- **Static libraries (`.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




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

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

Commonly Used Libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files. (differs in different versions)
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 2 MB archive of 444 object files. (differs in different versions)
- floating point math (sin, cos, tan, log, exp, sqrt, …)

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

```
% 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

```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

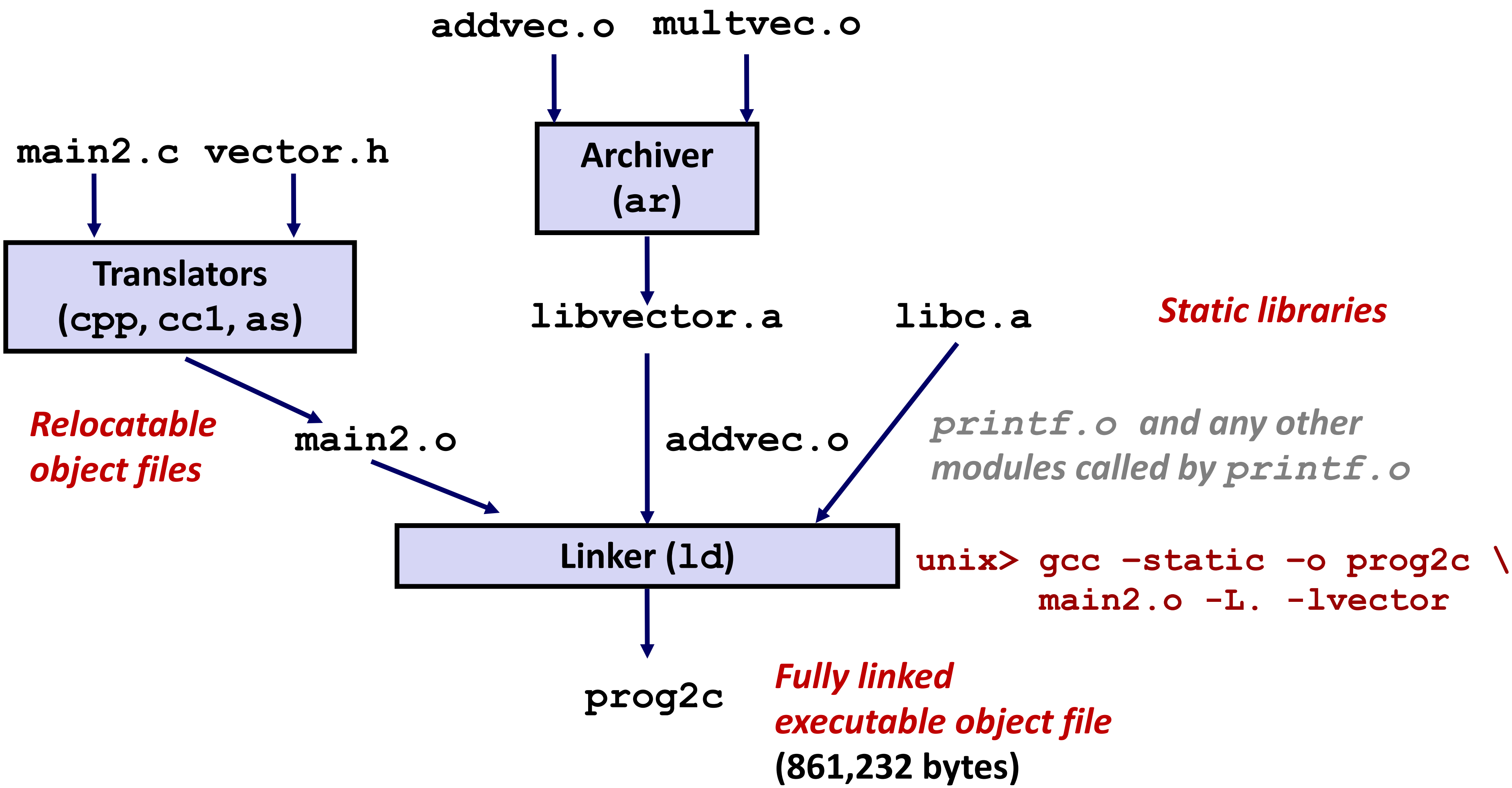
```c
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    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



"c" for "compile-time"

Using Static Libraries

- **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.

- **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
```

Modern Solution: Shared Libraries

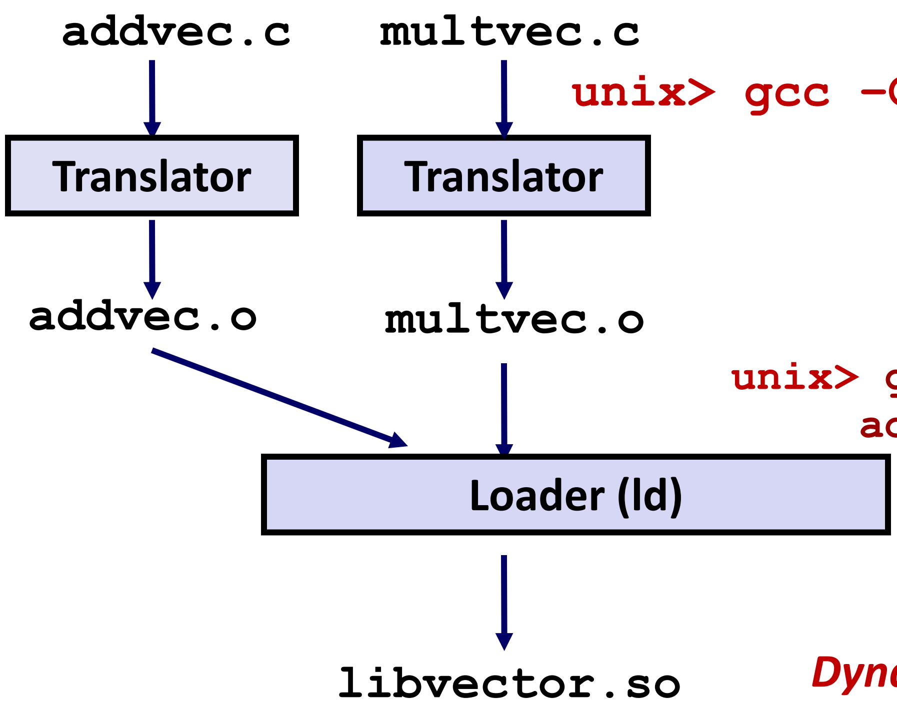
- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
    - Rebuild everything with glibc?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getad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

- **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: dynamic link libraries, DLLs, `.so` files

Shared Libraries (cont.)

- **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 (run-time linking).**
  - In Linux, this is done by calls to the **`dlopen()`** interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.
- **Shared library routines can be shared by multiple processes.**
  - More on this when we learn about virtual memory

Dynamic Library Example

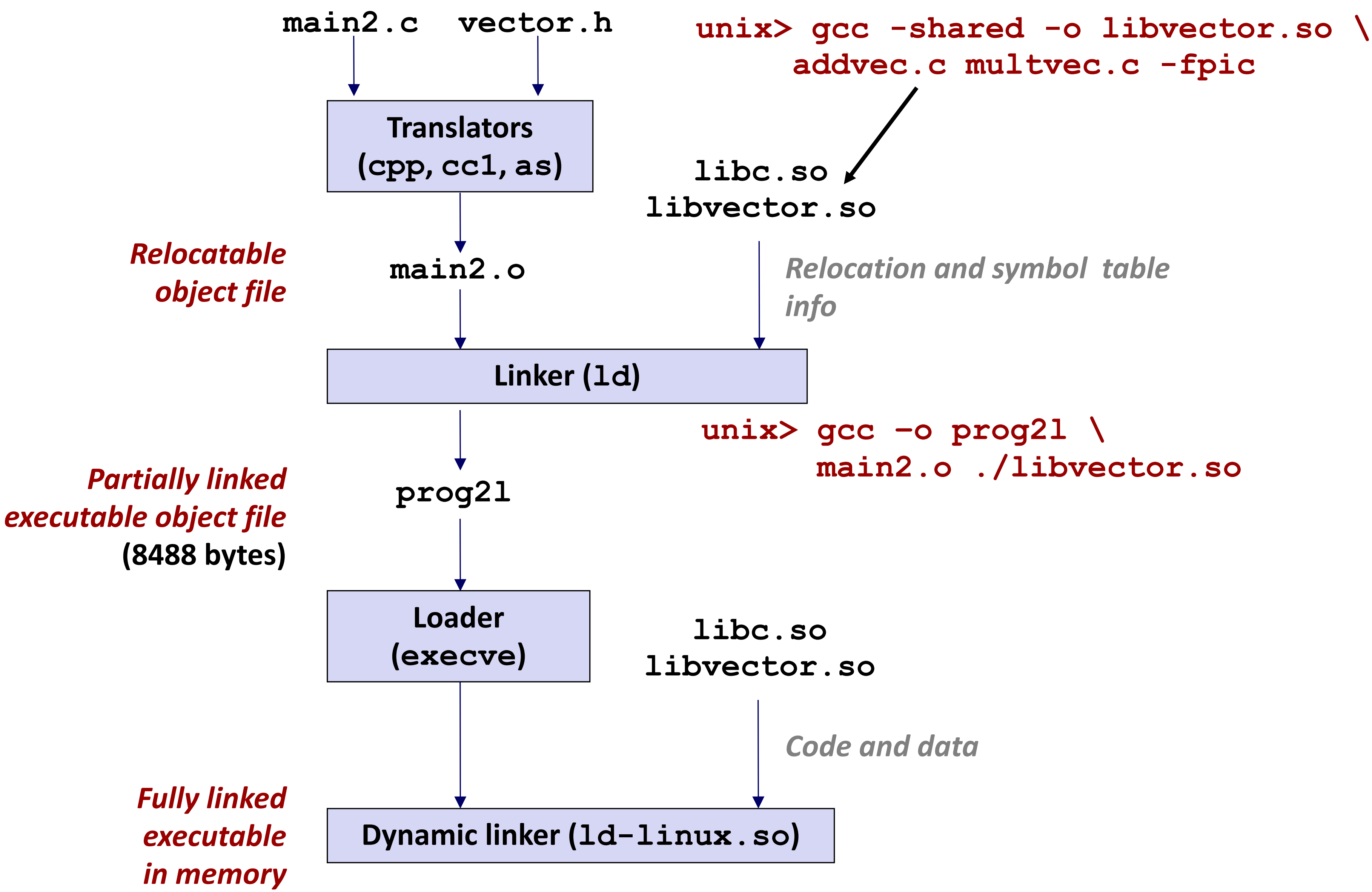



```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

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

Dynamic vector library

Dynamic Linking at Load-time



Dynamic Linking at Run-time

```
#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);
    }
  . . .
```

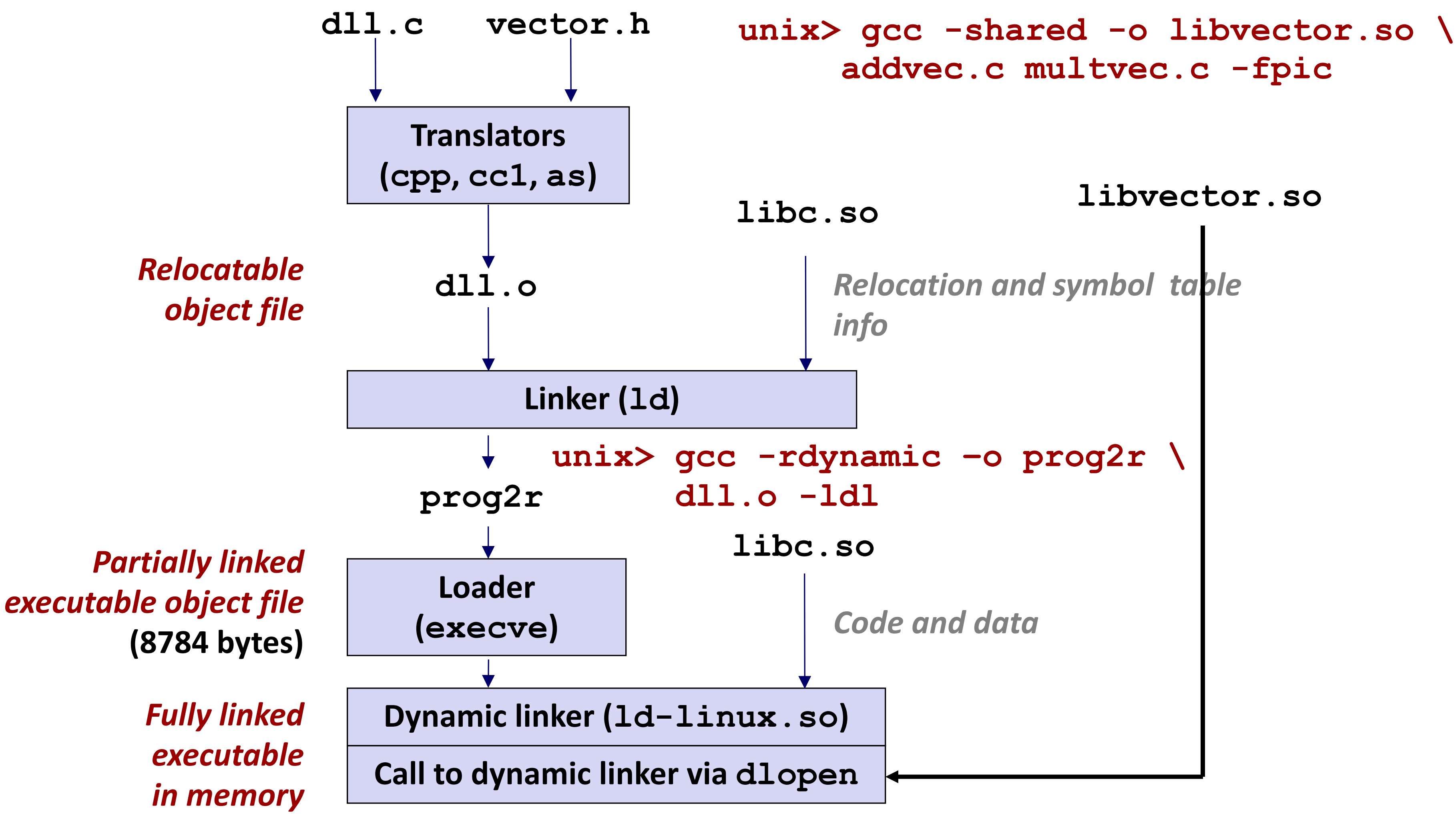
dll.c

Dynamic Linking at Run-time (cont)

```
    ...

    /* 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 Run-time



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.**

Case Study: Library Interpositioning

- **Documented in Section 7.13 of textbook**
- **Library interpositioning : powerful linking technique that allows programmers to intercept calls to arbitrary functions**
- **Interpositioning can occur at:**
  - Compile time: When the source code is compiled.
  - Link time: When the relocatable object files are statically linked to form an executable object file.
  - Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

Some Interpositioning Applications

- **Security**
  - Confinement (sandboxing)
  - Behind the scenes encryption
- **Debugging**
  - In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
  - Code in the SPDY networking stack was writing to the wrong location
  - Solved by intercepting calls to POSIX write functions (write, writev, pwrite)

Source: Facebook engineering blog post at:

https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/

Some Interpositioning Applications

- **Monitoring and Profiling**
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - Malloc tracing
    - Detecting memory leaks
    - **Generating address traces**
- **Error Checking**
  - C Programming Lab used customized versions of malloc/free to do careful error checking
  - Other labs (malloc, shell, proxy) also use interpositioning to enhance checking capabilities

Example program

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

int main(int argc,
         char *argv[])
{
  int i;
  for (i = 1; i < argc; i++) {
    void *p =
          malloc(atoi(argv[i]));
    free(p);
  }
  return(0);
}
```

int.c

- Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.
- Three solutions: interpose on the library `malloc` and `free` functions at compile time, link time, and load/run time.

Compile-time Interpositioning

```
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Compile-time Interpositioning

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
                                                  malloc.h
```

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc 10 100 1000
malloc(10)=0x1ba7010
free(0x1ba7010)
malloc(100)=0x1ba7030
free(0x1ba7030)
malloc(1000)=0x1ba70a0
free(0x1ba70a0)
linux>
```

Search for <malloc.h> leads to /usr/include/malloc.h

Search for <malloc.h> leads to

Link-time Interpositioning

```
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Link-time Interpositioning

```
linux> make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl \
    int.o mymalloc.o
linux> make runl
./intl 10 100 1000
malloc(10) = 0x91a010
free(0x91a010)
. . .
```

Search for `<malloc.h>` leads to `/usr/include/malloc.h`

- **The "`-Wl`" flag passes argument to linker, replacing each comma with a space.**
- **The "`--wrap,malloc`" `arg` instructs linker to resolve references in a special way:**
  - Refs to `malloc` should be resolved as `__wrap_malloc`
  - Refs to `__real_malloc` should be resolved as `malloc`

Load/Run-time Interpositioning

```
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Observe that DON'T have `#include <malloc.h>`

Load/Run-time Interpositioning

```
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c

Load/Run-time Interpositioning

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr 10 100 1000)
malloc(10) = 0x91a010
free(0x91a010)
. . .
linux>
```

Search for `<malloc.h>` leads to `/usr/include/malloc.h`

- The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `mymalloc.so` first.
- Type into (some) shells as:

```
env LD_PRELOAD=./mymalloc.so ./intr 10 100 1000)
```

Interpositioning Recap

- **Compile Time**
  - Apparent calls to **`malloc/free`** get macro-expanded into calls to **`mymalloc/myfree`**
  - Simple approach. Must have access to source & recompile
- **Link Time**
  - Use linker trick to have special name resolutions
    - **`malloc`** → **`__wrap_malloc`**
    - **`__real_malloc`** → **`malloc`**
- **Load/Run Time**
  - Implement custom version of **`malloc/free`** that use dynamic linking to load library **`malloc/free`** under different names
  - Can use with ANY dynamically linked binary

```
env LD_PRELOAD=./mymalloc.so gcc -c int.c)
```