UNM SCHOOL of ENGINEERING
Department of Computer Science

Lecture #22:
Chap. 7: Linking (2)

Prof. Soraya Abad-Mota, PhD

From the index cards

- When to make a variable static?
- “manual linking” (what I meant)
- Separate modules, i.e. .c and .h files, must be *linked* to produce an appropriate executable. The linker might give errors, in which case the program cannot execute

Questions

- Name the two tasks of linking
- What is the ELF format?
- Can you name 3 sections it contains?

An example

- Practice problem 7.1 p. 678
  - provides two .c files (`m.c` and `swap.c`)
  - variables and functions defined on each
  - for each symbol, the exercise asks for the type, if it has an entry in .symtab, where it is defined, and the section it belongs to.
  - uses definitions on section 7.5
  - Solution on p. 717, but try it out first

(a) m.c

code/link/m.c

```c
void swap();

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

int main()
{
    swap();
    return 0;
}
```

code/link/m.c

(b) swap.c

code/link/swap.c

```c
extern int buf[];

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

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

code/link/swap.c

Figure 7.5 **Example program for Practice Problem 7.1.**

Declaration and definition in C

- For functions:
  - Declaration: signature (prototype) only (by default: `extern`)
  - Definition: body of the function
- For variables:
  - Declaration: only the type, no memory allocation
    - Attained with `extern int var;`
  - Definition: type and memory allocation
    - E.g. `int var;`
    - Those declared as extern must be defined somewhere

Answers here

Symbol	.symtab entry?	Symbol type	Module where defined	Section
buf	_____	_____	_____	_____
bufp0	_____	_____	_____	_____
bufp1	_____	_____	_____	_____
swap	_____	_____	_____	_____
temp	_____	_____	_____	_____

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`

```
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

Static vars preserve their value even after out of scope

Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x.1` (x in f) and `x.2` (x in g).

How Linker Resolves Duplicate Symbol Definitions

- Program symbols are either *strong* or *weak*
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals

p1.c

strong → `int foo=5;`

strong → `p1() { }`

p2.c

`int foo;` ← weak

`p2() { }` ← strong

(Linux) Linker's Symbol Rules

- Rule 1: Multiple strong symbols[1] are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error
- Rule 2: Given a strong symbol and multiple weak symbols[1], choose the strong symbol
  - References to the weak symbol resolve to the strong symbol
- Rule 3: If there are multiple weak symbols[1], pick an arbitrary one
  - Can override this with **`gcc -fno-common`** (triggers error if encounters multiply defined global symbols) or –Werror (to turn warnings into errors)

[1] with the same name

Practice Problem 7.2 (solution page 718)

In this problem, let REF(x.i) → DEF(x.k) denote that the linker will associate an arbitrary reference to symbol x in module i to the definition of x in module k. For each example that follows, use this notation to indicate how the linker would resolve references to the multiply-defined symbol in each module. If there is a link-time error (rule 1), write "ERROR". If the linker arbitrarily chooses one of the definitions (rule 3), write "UNKNOWN".

A.

```
/* Module 1 */
int main()
{
}
```

```
/* Module 2 */
int main;
int p2()
{
}
```

(a) REF(main.1) → DEF(_____._____)

(b) REF(main.2) → DEF(_____._____)

B.

```
/* Module 1 */
void main()
{
}
```

```
/* Module 2 */
int main = 1;
int p2()
{
}
```

(a) REF(main.1) → DEF(_____._____)

(b) REF(main.2) → DEF(_____._____)

C.

```
/* Module 1 */
int x;
void main()
{
}
```

```
/* Module 2 */
double x = 1.0;
int p2()
{
}
```

(a) REF(x.1) → DEF(_____._____)

(b) REF(x.2) → DEF(_____._____)

Topics (3)

- Chapter 7: Linking
  - ELF format, linker symbols.
  - The linking process in more detail
    - Linker puzzles
    - The relocation phase of the Linking process

A note about

- Static global variables (not visible from outside their module) are still of interest to the Linker:
  - They need to be stored in the .data section of the object code; they should not go in the stack, because all the functions in that module would not be able to use them if they are stored on the stack.

- A reflection on what is stored in the object files vs what's stored in the stack:
  - Only the local variables are the ones stored in the stack and of no interest to the Linker.

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

```
int x;
p1() {}
```

```
int x;
p2() {}
```

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

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! (both are weak, unknown which chosen)

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` will overwrite `y`!
Nasty! (and subtle) (quietly chooses x=7 def of x)

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same initialized variable.

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

Global Variables

- Avoid if you can

- Otherwise
  - Use **`static`** if you can (makes it invisible outside of its module)
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable

Topics

- Chapter 7: Linking
  - ELF format, linker symbols.
  - The linking process in more detail
    - Linker puzzles
    - The relocation phase of the Linking process
  - Libraries

After symbol resolution (phase 1)

- symbol reference 1:1 one symbol definition (association established as table entry in the symbol table)
- the linker knows:
  - exact size of code sections
  - exact size of data sections

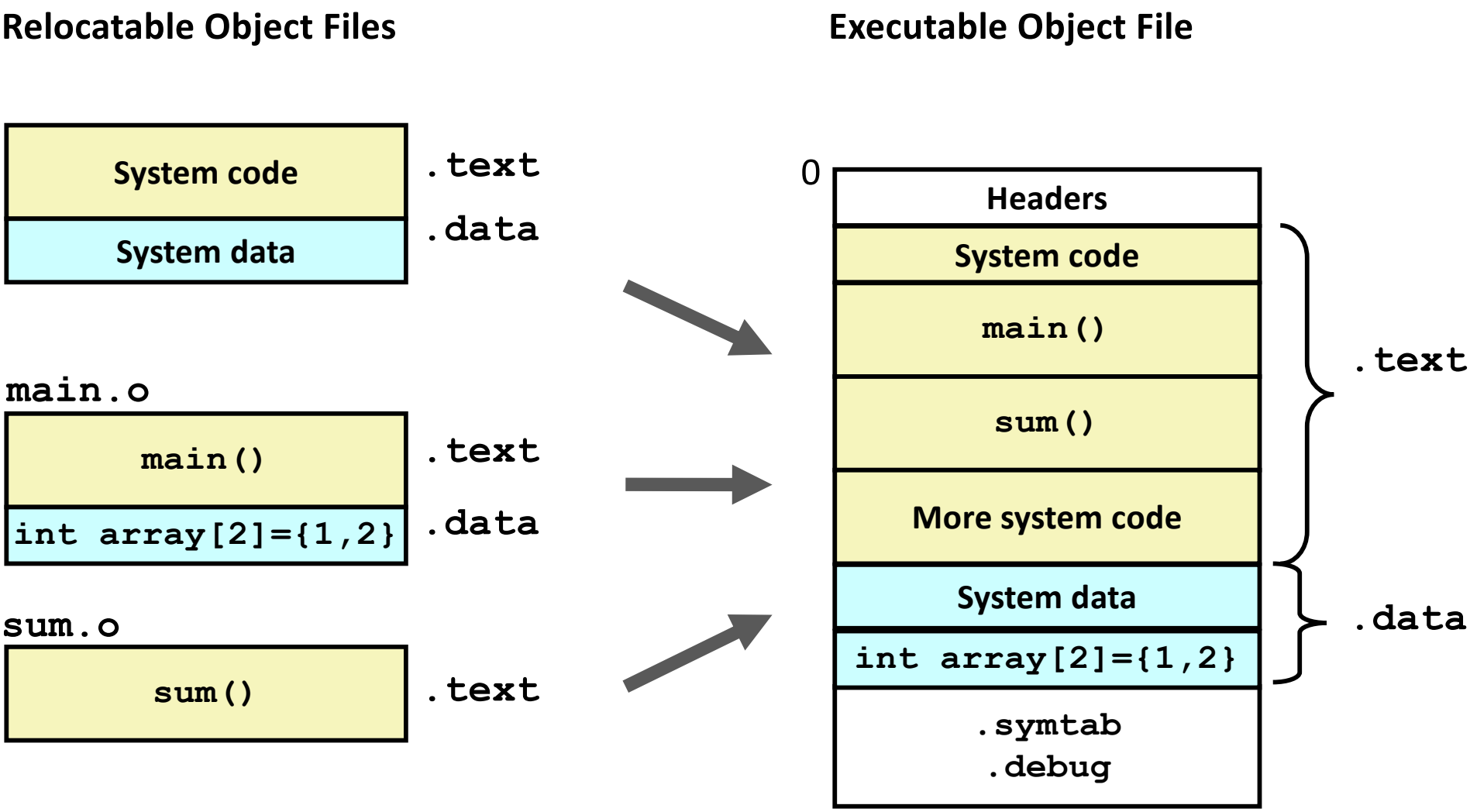
  in the object modules provide as input to it.

Linking process (2nd phase)

- Relocation
  - merges the input modules and assigns run-time addresses to each symbol.
  - Two steps:

1. Relocating sections and symbol definitions
   - output: each instruction and global var has unique run-time memory address
2. Relocating symbol references within sections
   - output: modifies every symbol ref to point to correct run-time address. Uses relocation entries data structures.

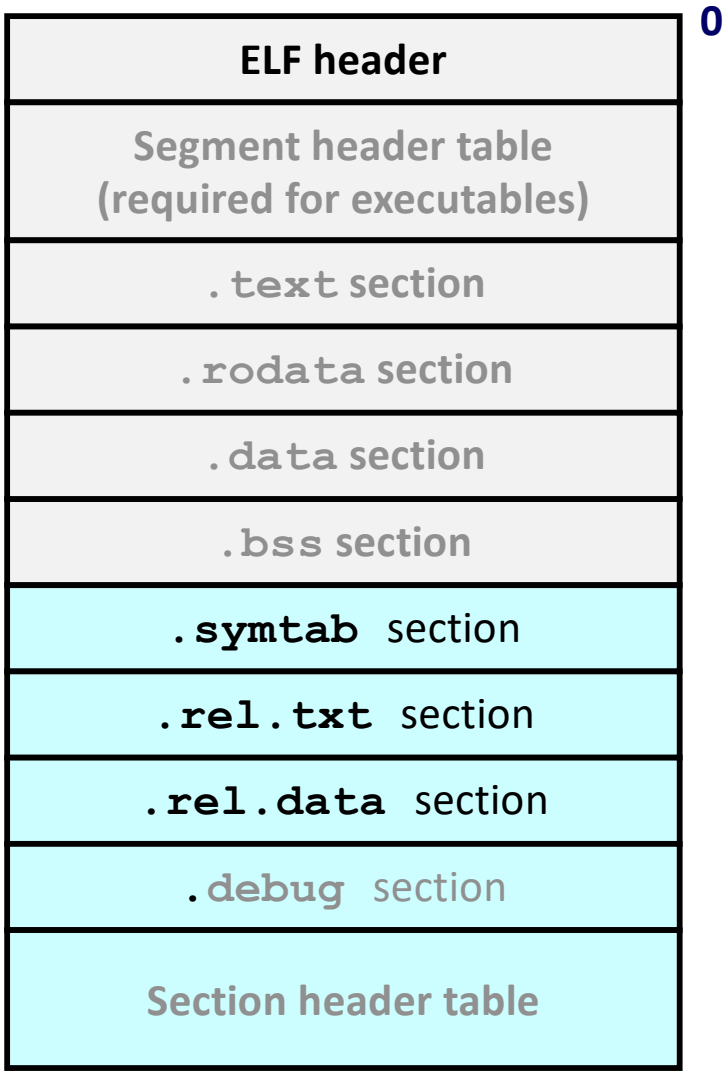
Step 2: Relocation (sec. 7.7)



Step 2.1: Relocating sections and symbol definitions

Relocation entries in ELF

- `.symtab` section
  - Symbol table (no entries for local vars)
  - 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



Relocation Entries

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

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

Step 2.2: Relocating symbol refs within sections
Relocation entries for code & data

Absolute address
PC-relative address

```
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

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

Relocated .text section

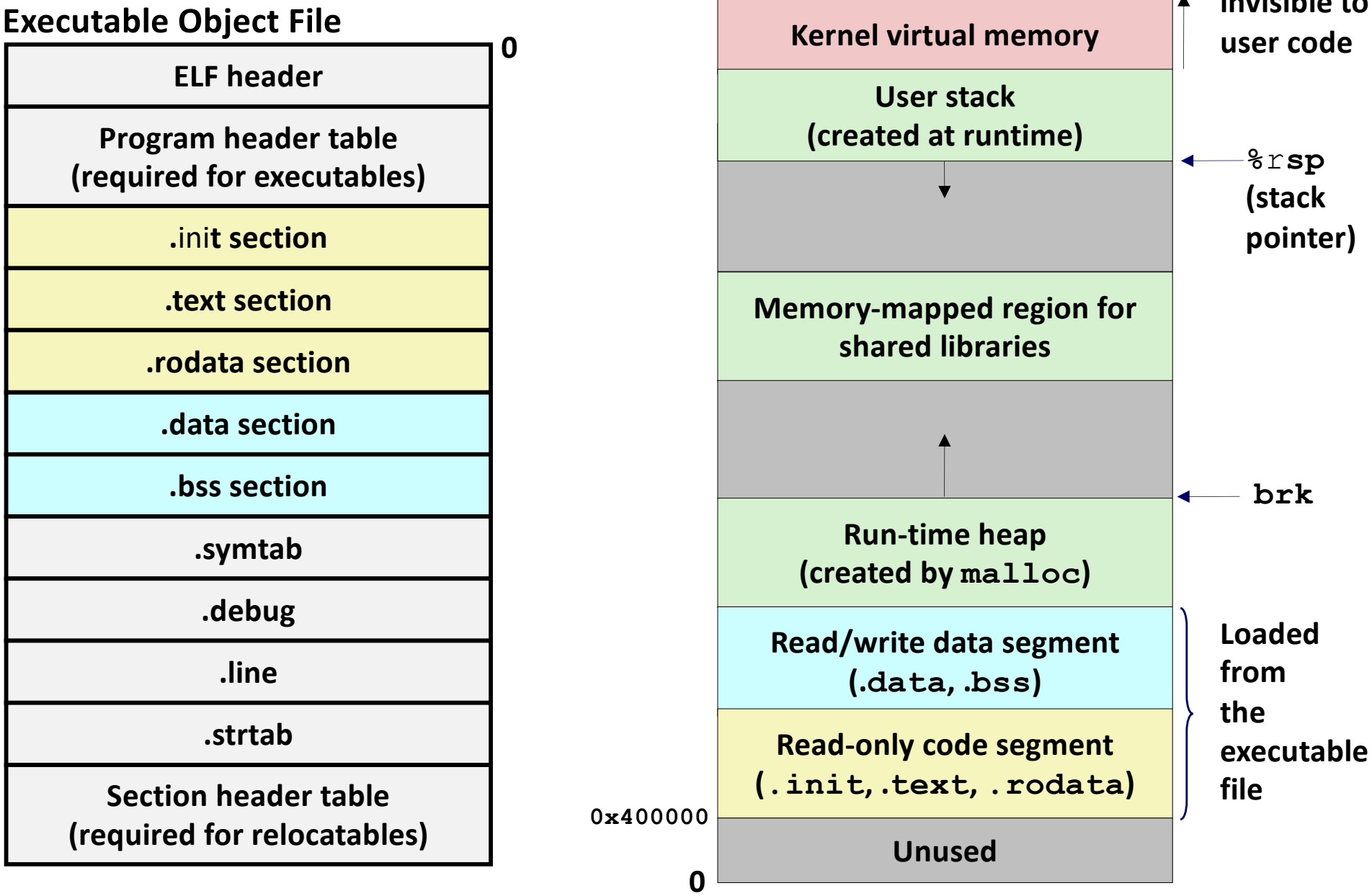
```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08       sub    $0x8,%rsp
  4004d4:       be 02 00 00 00    mov    $0x2,%esi
  4004d9:       bf 18 10 60 00    mov    $0x601018,%edi  # %edi = &array
  4004de:       e8 05 00 00 00    callq  4004e8 <sum>    # sum()
  4004e3:       48 83 c4 08    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

Using PC-relative addressing for sum(): 0x4004e8 = 0x4004e3 + 0x5

Source: objdump -dx prog

Loading Executable Object Files



UNM SCHOOL of ENGINEERING
Department of Computer Science

Chap. 7: Linking (end)

Prof. Soraya Abad-Mota, PhD

Topics

- Ch. 7: Linking
  - Libraries (Secs. 7.6.2 and 7.6.3)
    - static libraries (this is where the order matters)
    - shared libraries
    - dynamic libraries

NOTE: will not cover this semester, but here are 12 slides which explain historical background and current approach: Shared Libraries. Go to slide 48 for a brief overview of shared libraries.

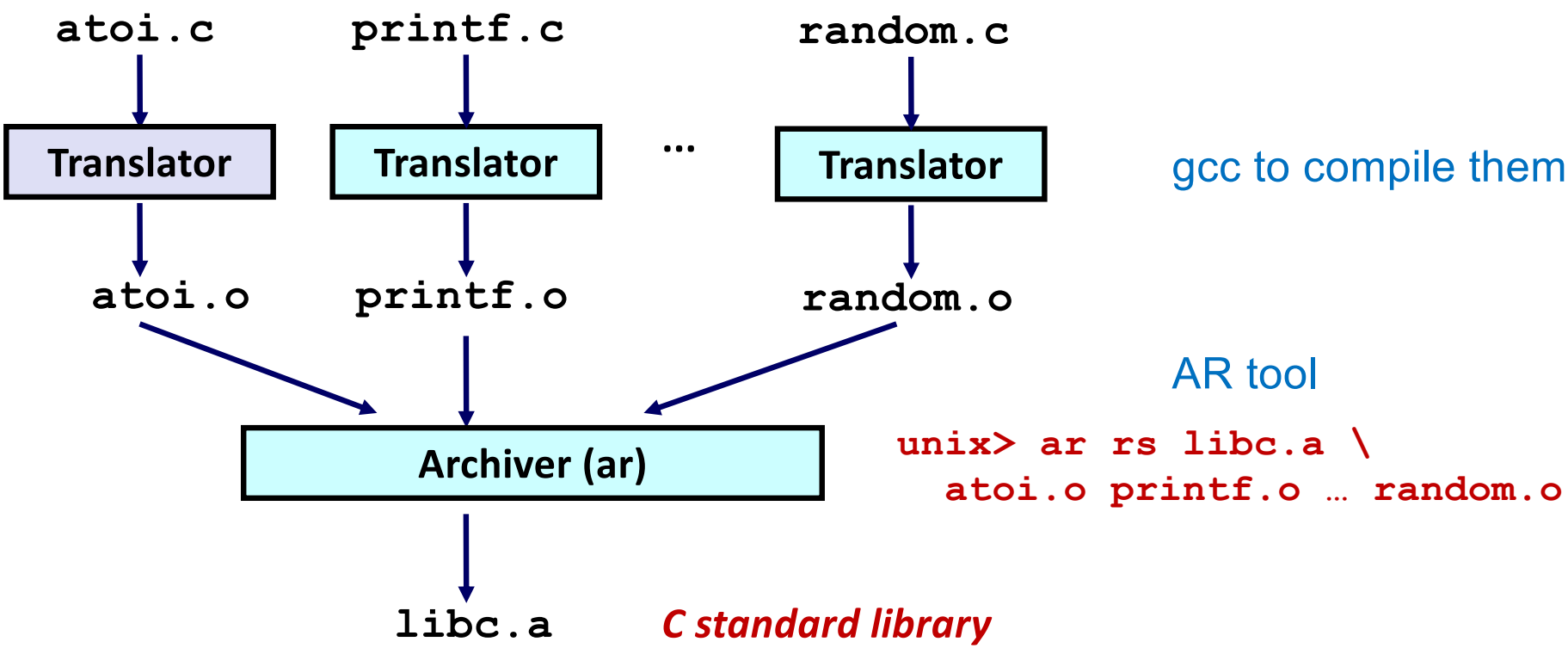
Packaging Commonly Used Functions
(sections 7.6.2 and 7.6.3)

- 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)
  - To resolve the inconvenience of previous approaches.
  - 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.

Commonly Used Libraries

`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

`libm.a` (the C math library)

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

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

```
% ar -t 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()
{
    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];
}
```

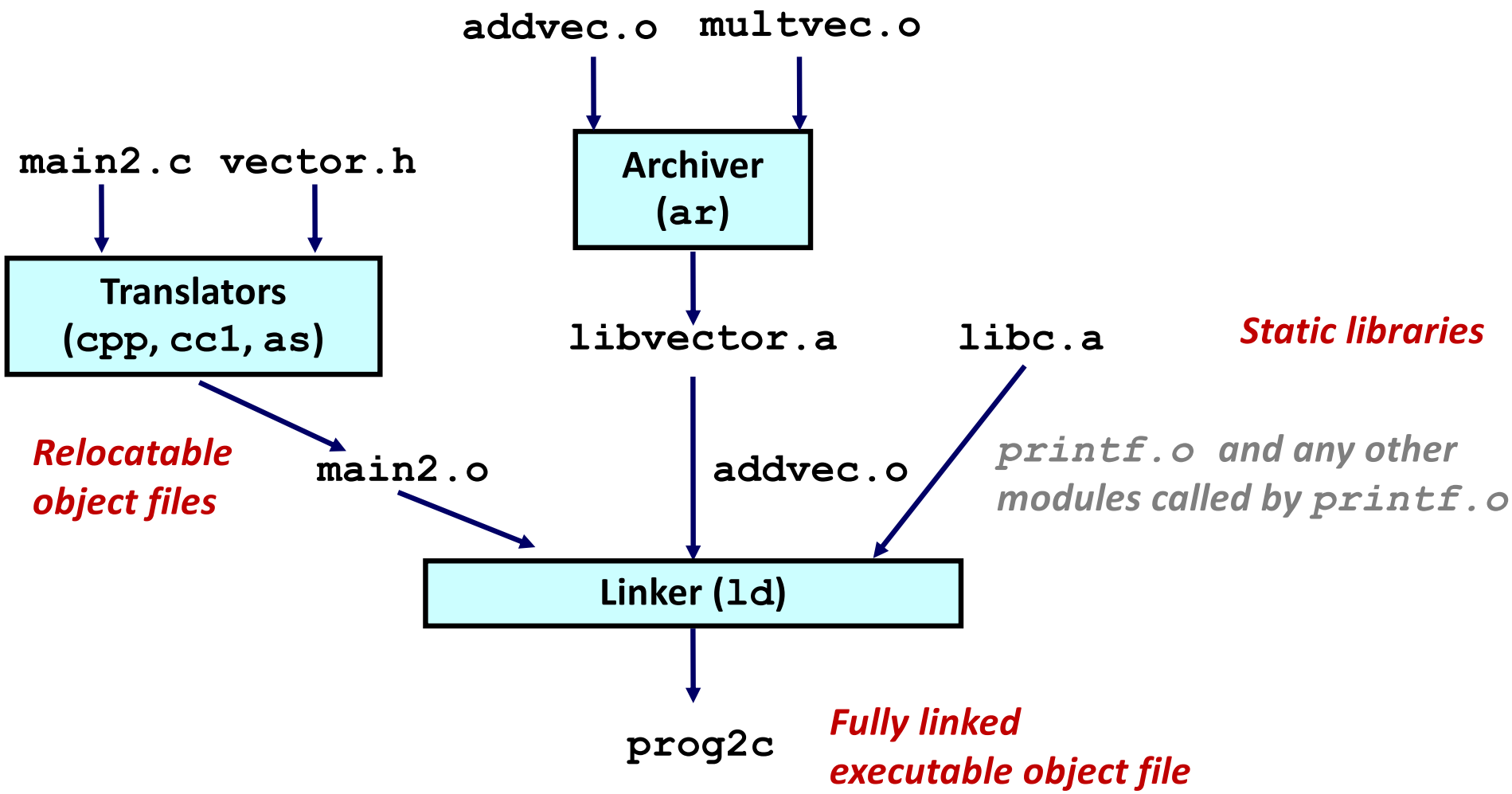
addvec.c

```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 -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Modern Solution: Shared Libraries (section 7.10)

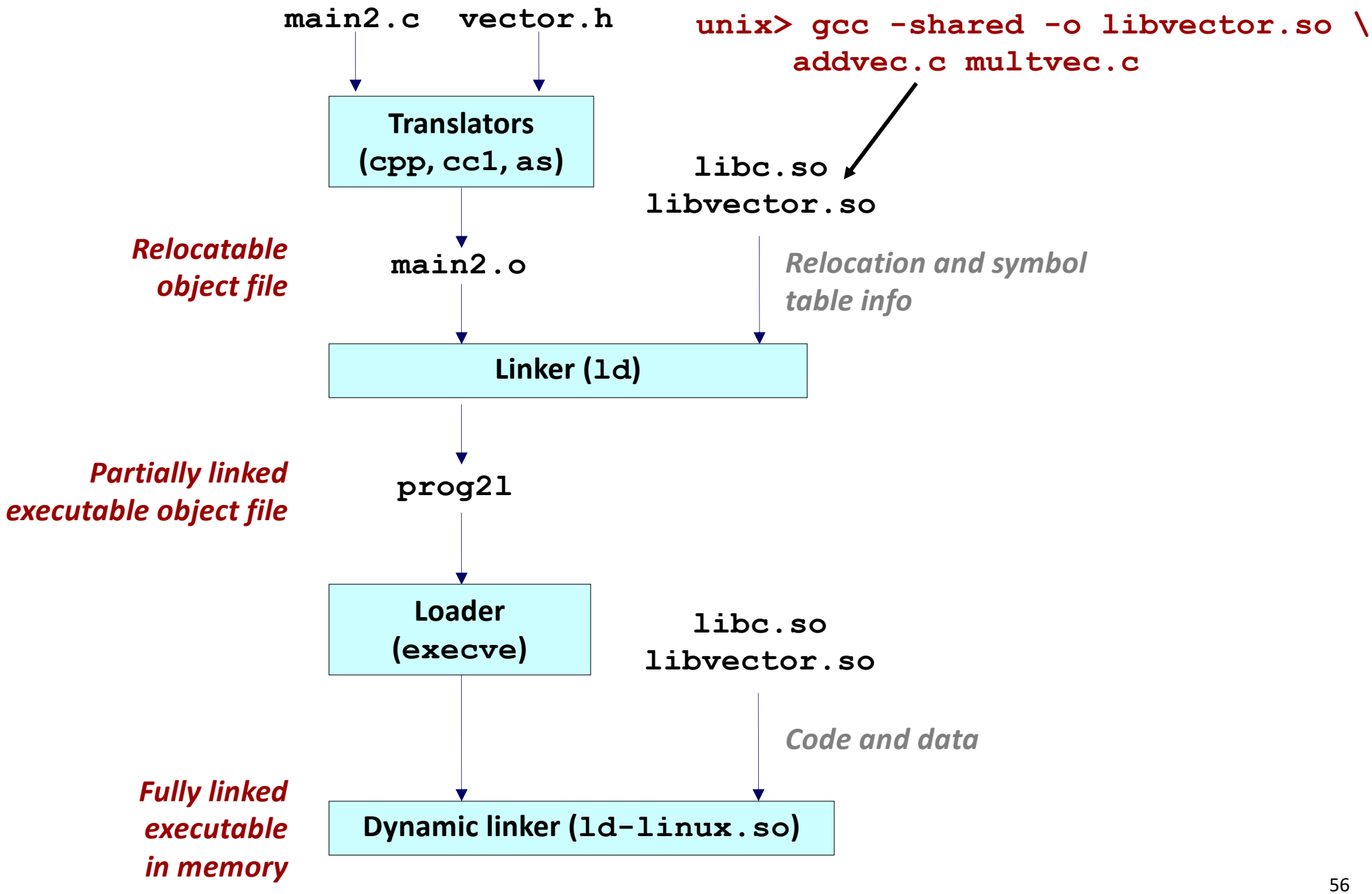
- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- 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.) (goto slide 53)

- 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 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()
{
    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

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.

Notes about chapter 7

- Do not cover section 7.12
- We will not cover Library Interpositioning, sec. 7.13, either, but for those interested in reading and learning about a case study of its use in Facebook in 2014, I will copy those slides on canvas.

The Cachelab Project

Part A: Build a cache simulator

- Divided in two parts:
  - A1 - Design (mostly of the data structures but also the algorithm) record a video explaining + 1-page document (due in 1 week)
  - A2 - Implementation (2 more weeks)

Part B: optimize matrix transpose