



Linking

18-213/18-613: Introduction to Computer Systems
11th Lecture, Spring 2026

Today

■ Linking

- Motivation
- What it does
- How it works
- Shared libraries and dynamic linking

CSAPP 7.1

CSAPP 7.2-7.4

CSAPP 7.5-7.9

CSAPP 7.10-7.12

■ Case study: Library interpositioning

CSAPP 7.13

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

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;

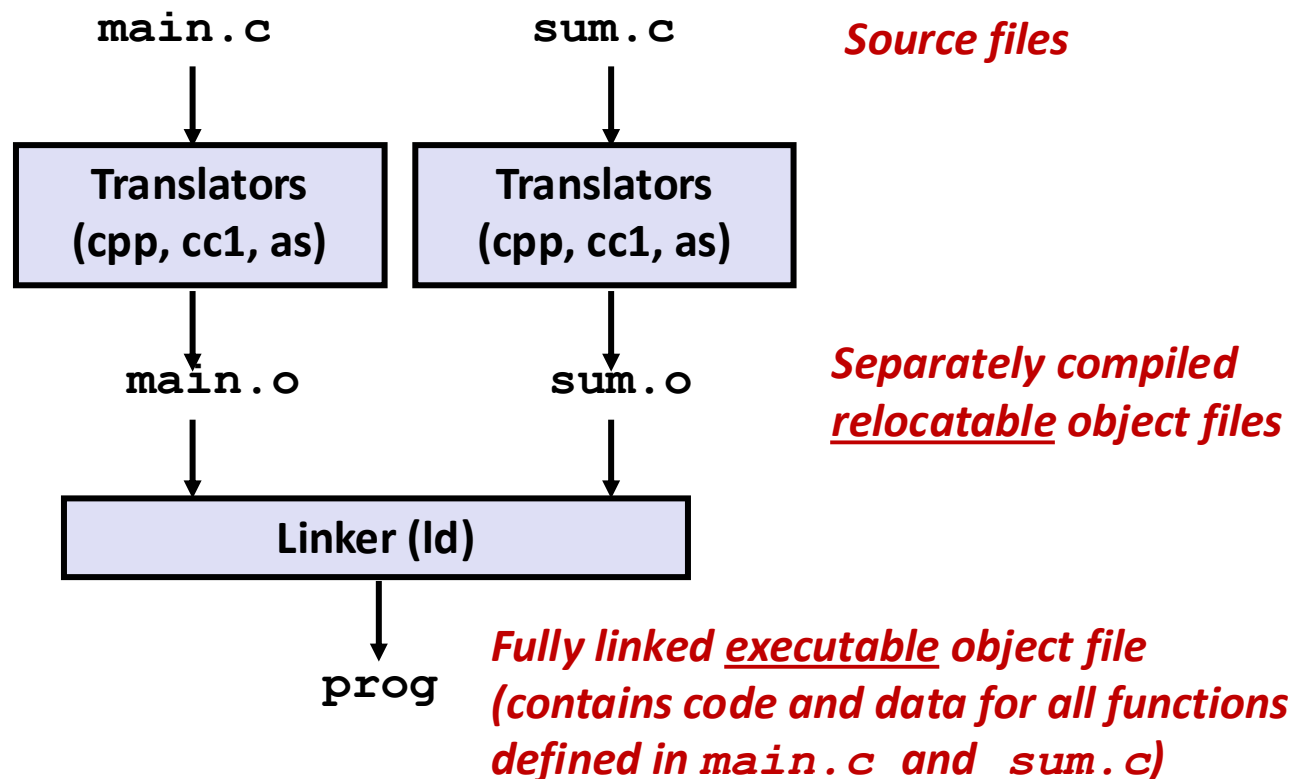
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

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.**

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

What Do Linkers Do? (cont'd)

■ 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**

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

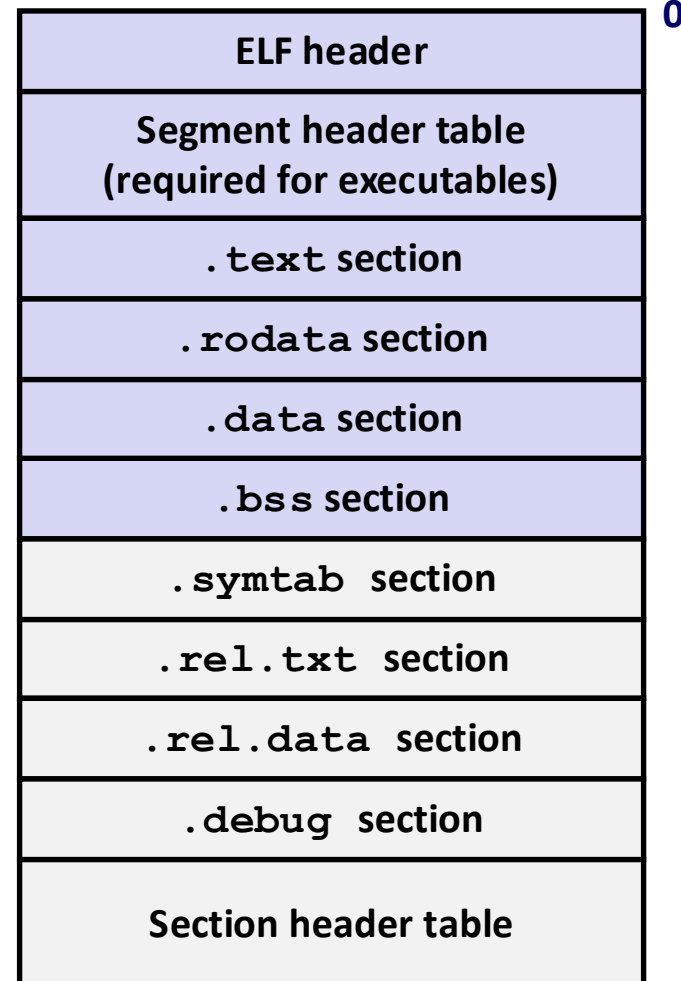
- Read only data: jump tables, string constants, ...

■ .data section

- Initialized global variables

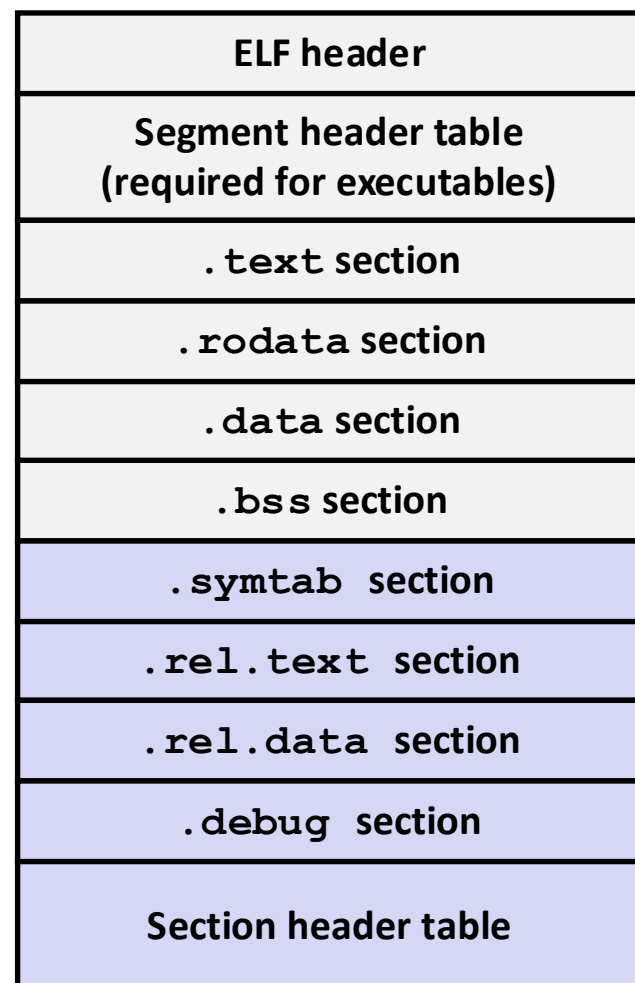
■ .bss section

- 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



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...

```
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
```

Defining
a global

Linker knows
nothing of val

Referencing
a global...

...that's defined here

Linker knows
nothing of i or s

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
```

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;
}

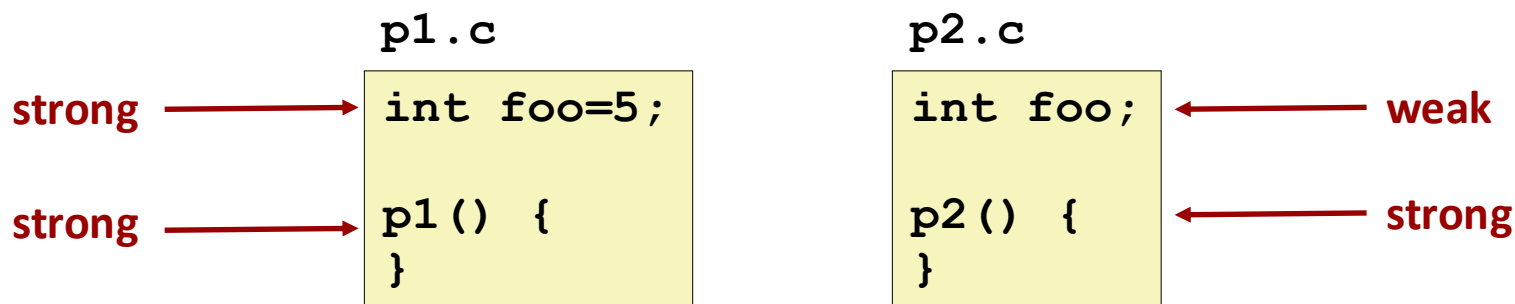
int h() {
    return x += 27;
}
static-local.c
```

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 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?

```
-bash-4.2$ ./mismatch  
4614253070214989087
```

Global Variables

- **Avoid if you can**

- **Otherwise**
 - Use **static** if you can (why?)
 - 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

Use of extern in .h Files

c1.c

```
#include "global.h"

int f() {
    return g+1;
}
```

global.h

```
extern int g;
int f();
```

c2.c

```
#include <stdio.h>
#include "global.h"

int g = 0;

int main(int argc, char argv[]) {
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

Step 2: Relocation

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

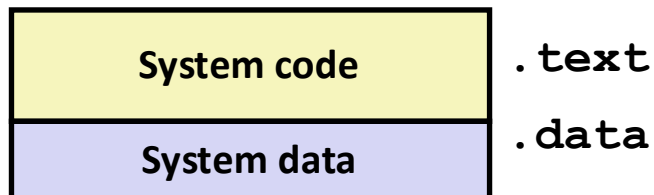
int array[2] = {1, 2};

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

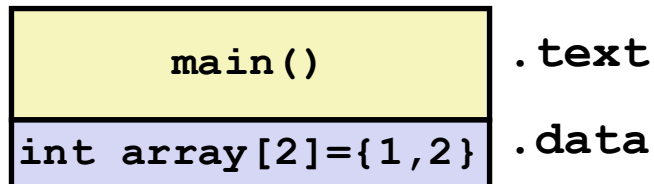
```
int sum(int *a, int n)
{
    int i, s = 0;

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

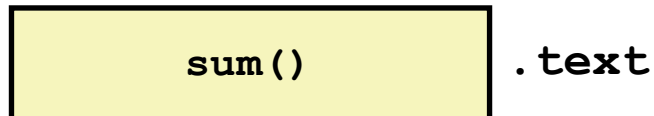
Relocatable Object Files



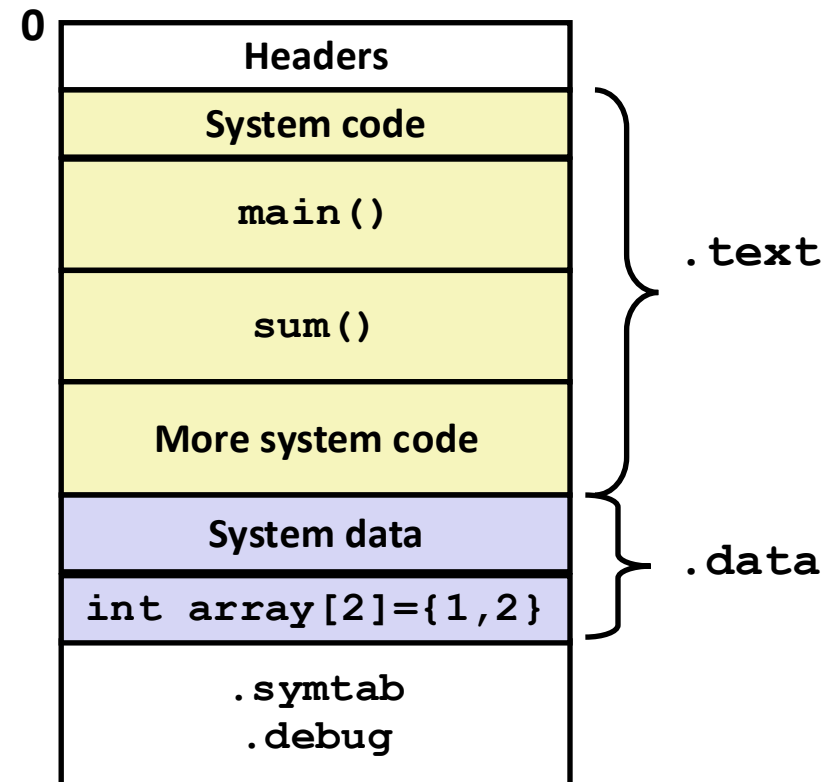
main.o



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

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
                          # Relocation entry
a:  R_X86_64_32 array

e:  e8 00 00 00 00      callq 13 <main+0x13>     # sum()
                          # Relocation entry
f:  R_X86_64_PC32 sum-0x4

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

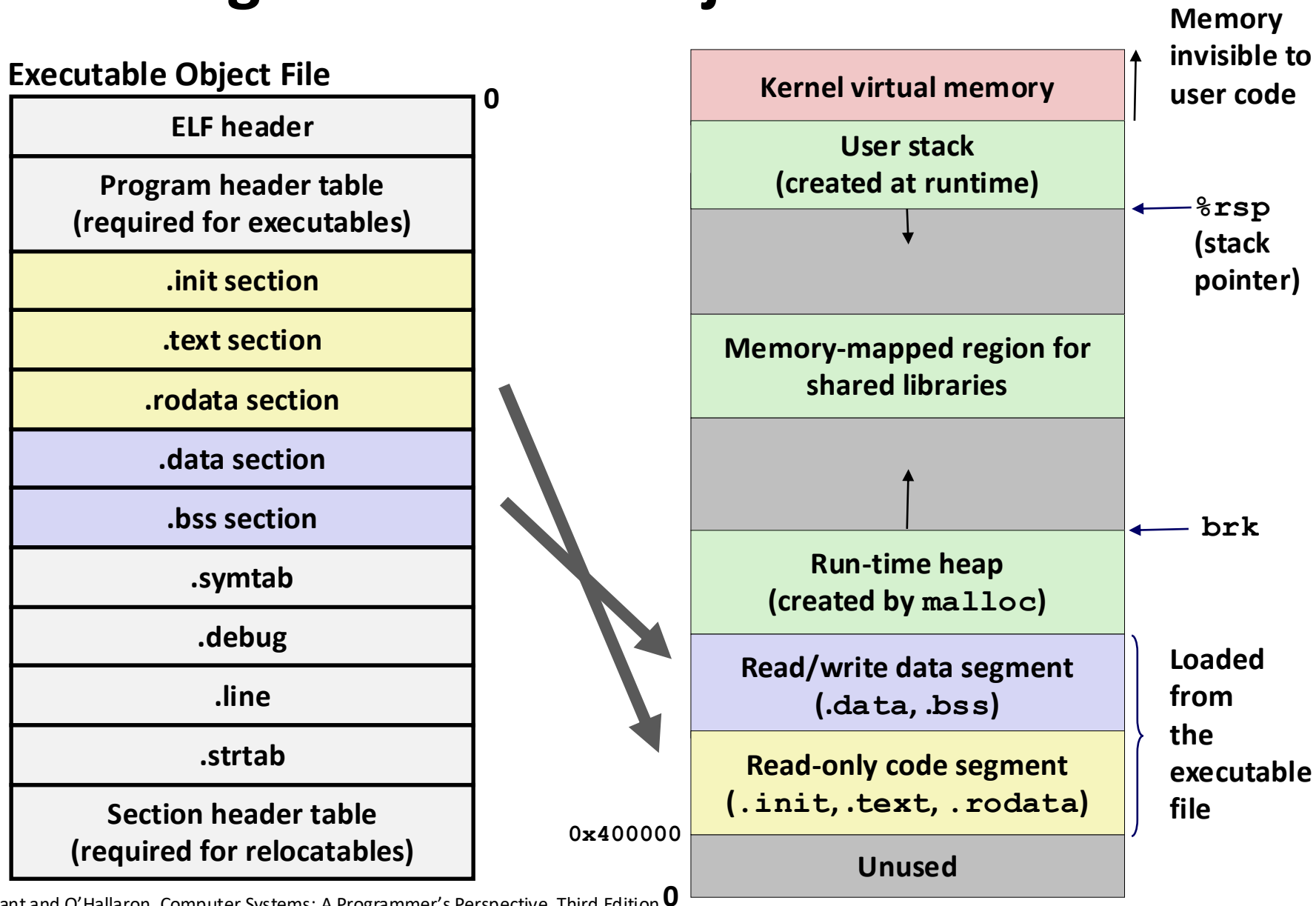
00000000004004e8 <sum>:
4004e8:      b8 00 00 00 00      mov    $0x0,%eax
4004ed:      ba 00 00 00 00      mov    $0x0,%edx
4004f2:      eb 09              jmp    4004fd <sum+0x15>
4004f4:      48 63 ca          movslq %edx,%rcx
4004f7:      03 04 8f          add    (%rdi,%rcx,4),%eax
4004fa:      83 c2 01          add    $0x1,%edx
4004fd:      39 f2            cmp    %esi,%edx
4004ff:      7c f3            jl    4004f4 <sum+0xc>
400501:      f3 c3          repz retq

```

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

$$0x4004e8 = 0x4004e3 + 0x5$$

Loading Executable Object Files



Quiz Time!

Canvas Quiz: Day 11 – Linking

Today

- **Linking**
 - Motivation
 - What it does
 - How it works
 - **Shared libraries and dynamic linking**
- **Case study: Library interpositioning**

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 /usr/lib/libc.a | sort
...
fork.o
...
fprintf.o
fpu_control.o
putc.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
...
```

Libraries: Packaging a Set of Functions

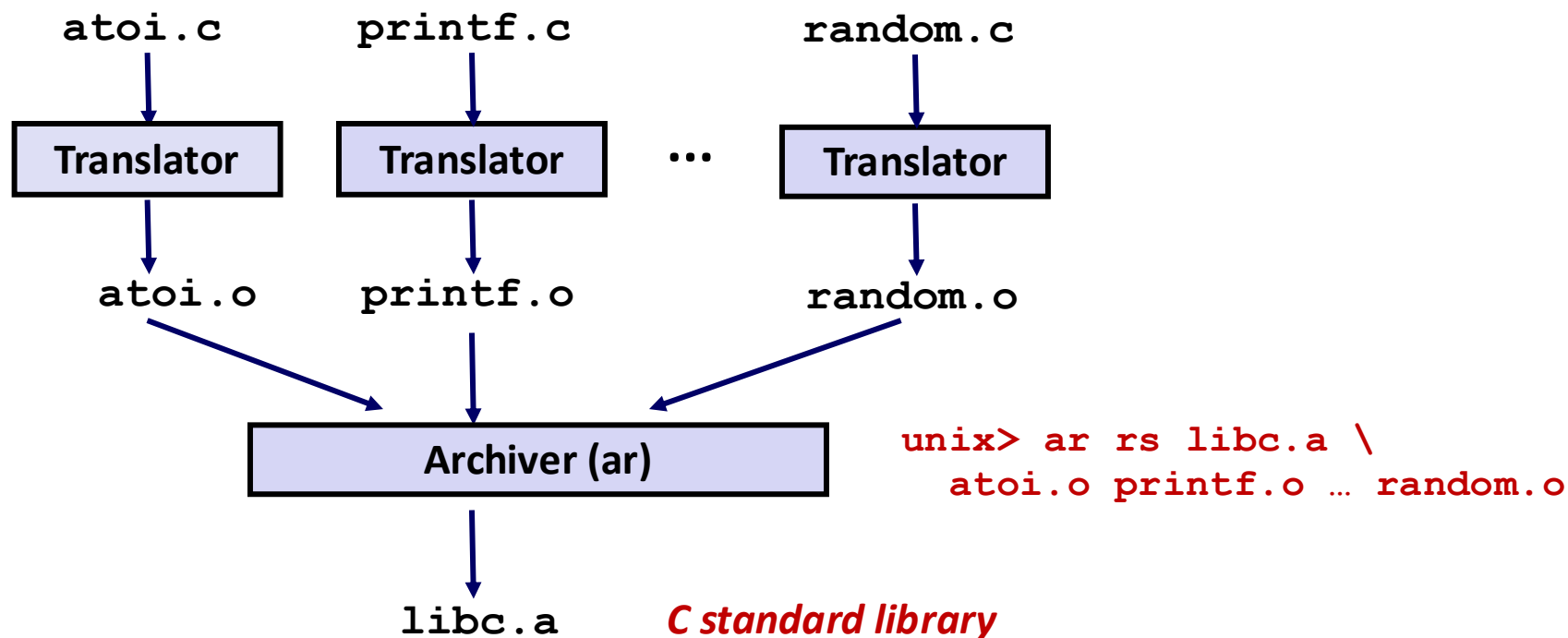
- **How to package functions commonly used by programmers?**
 - Math, I/O, memory management, string manipulation, etc.
- **Awkward, given the linker framework so far:**
 - **Option 1:** Put all functions into a single source file
 - Programmers link big object file into their programs
 - Space and time 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



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

Linking with Static Libraries

```
#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

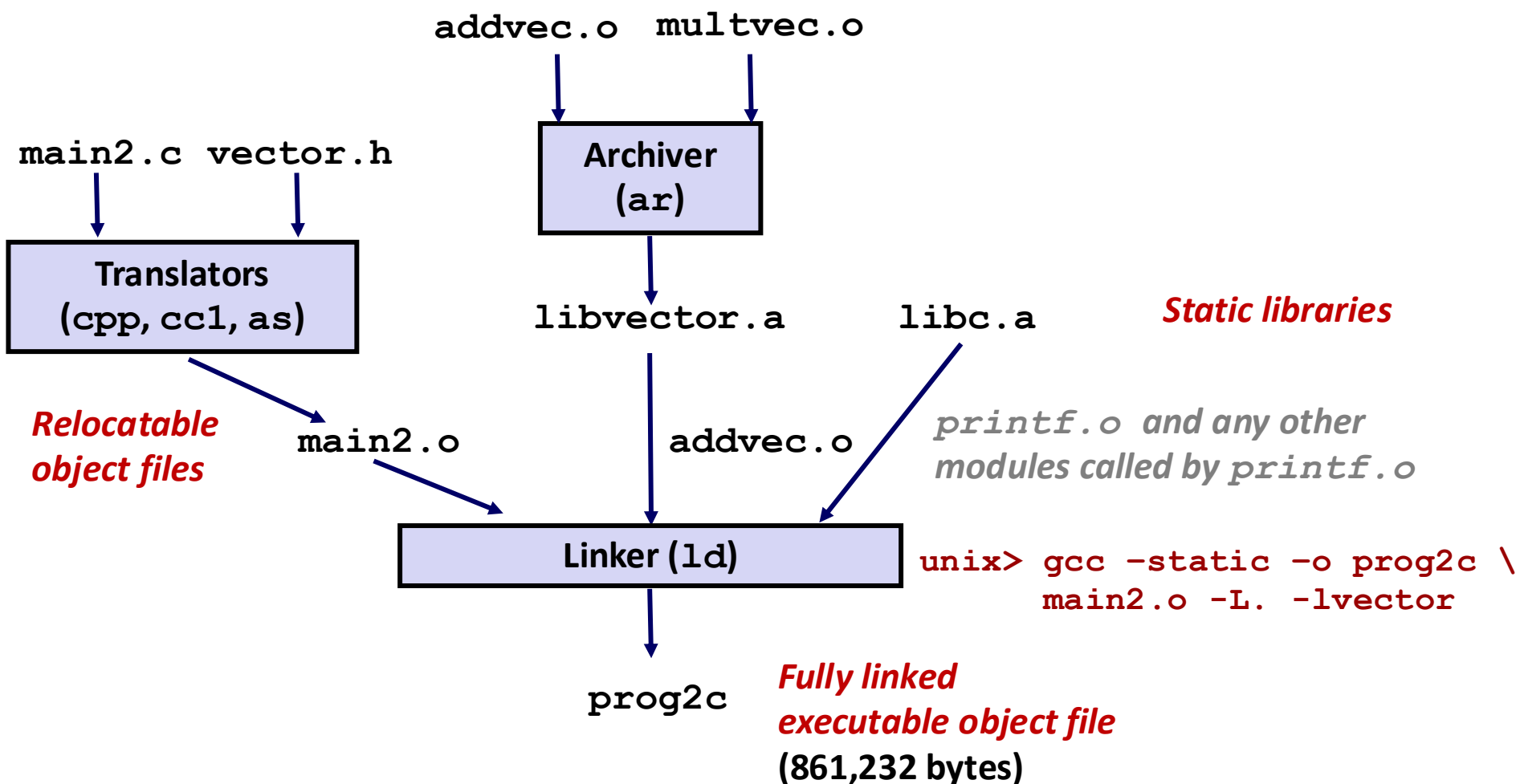
```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
addvec.c
```

```
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**
 - Using the virtual-to-physical memory mapping

What dynamic libraries are required?

■ `.interp` section

- Specifies the dynamic linker to use (i.e., `ld-linux.so`)

■ `.dynamic` section

- Specifies the names, etc of the dynamic libraries to use
- Follow an example of `prog`

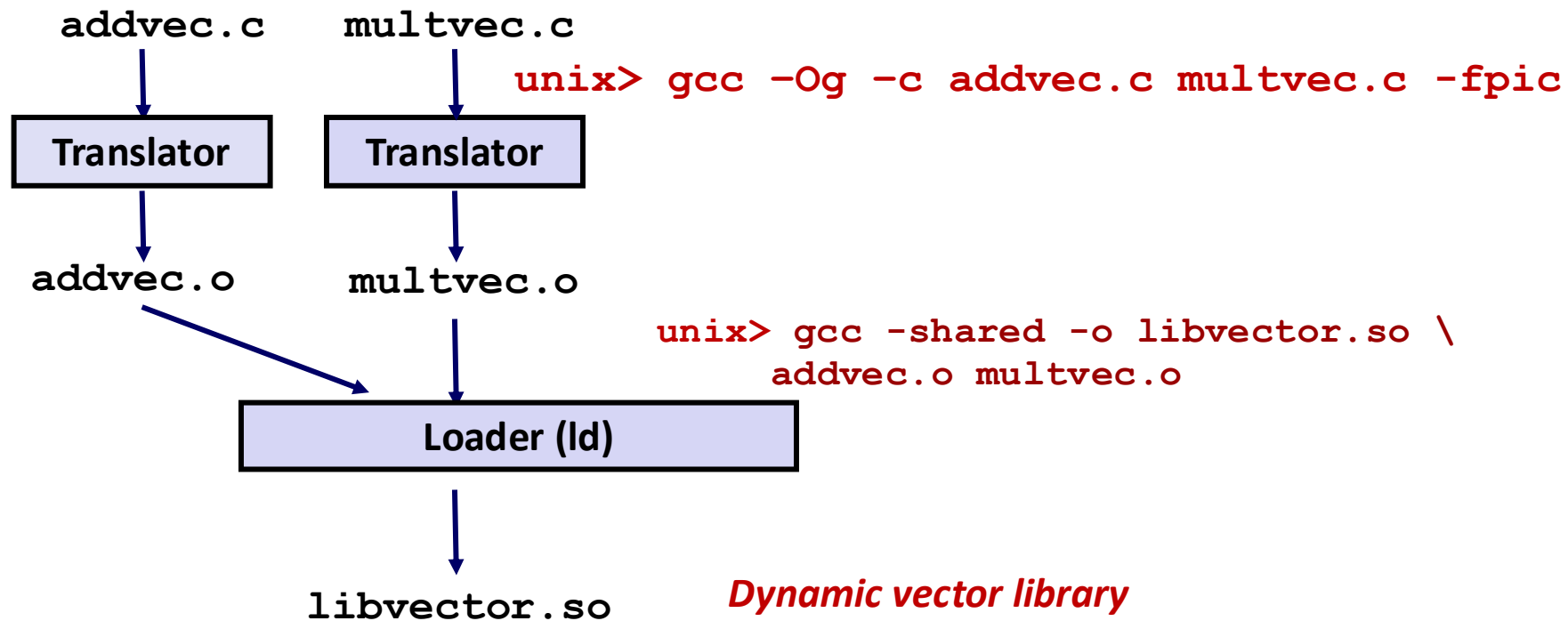
```
(NEEDED) Shared library: [libm.so.6]
```

■ Where are the libraries found?

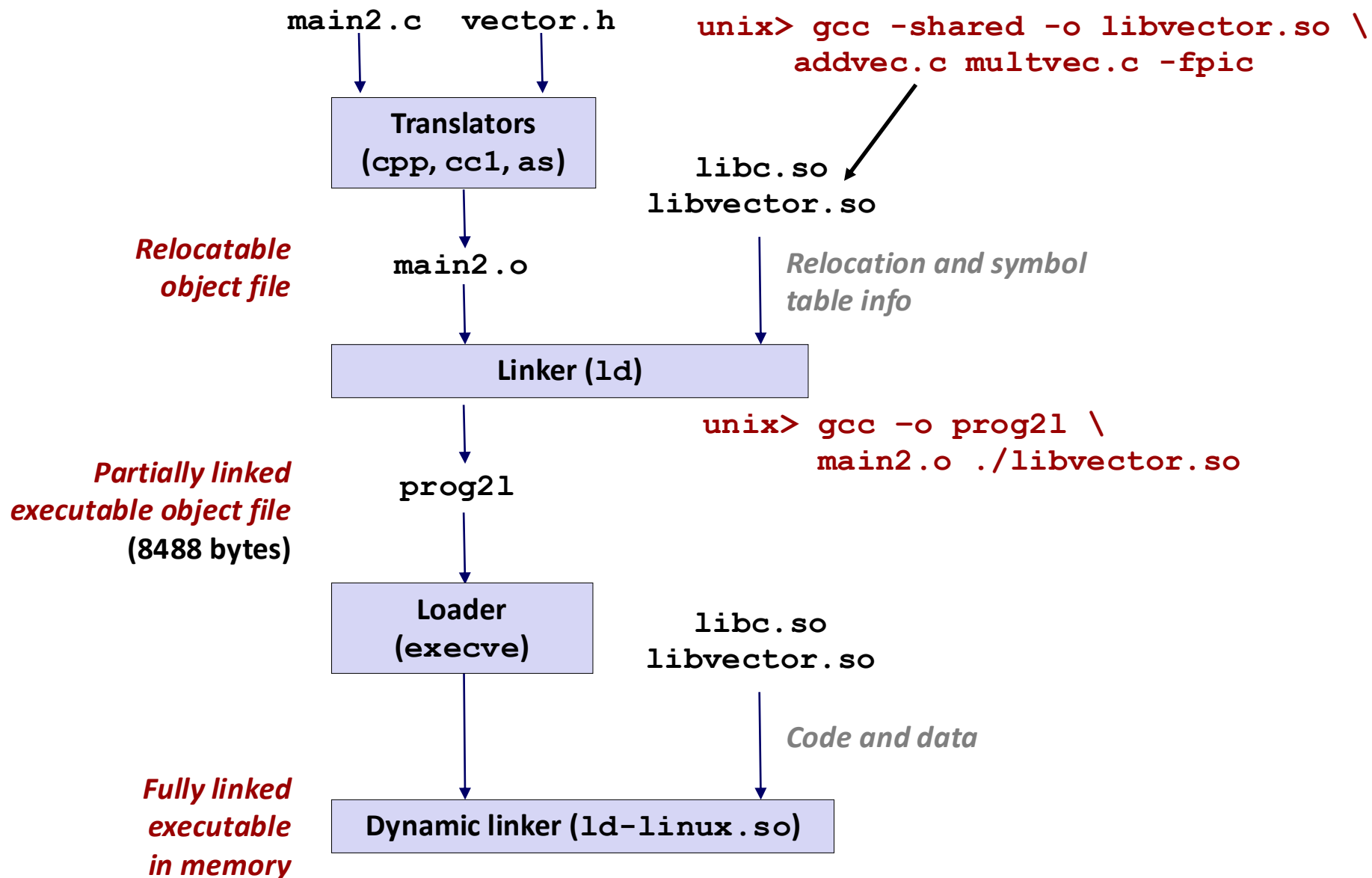
- Use “`ldd`” to find out:

```
unix> ldd prog
linux-vdso.so.1 => (0x00007ffcf2998000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
/lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

Dynamic Library Example

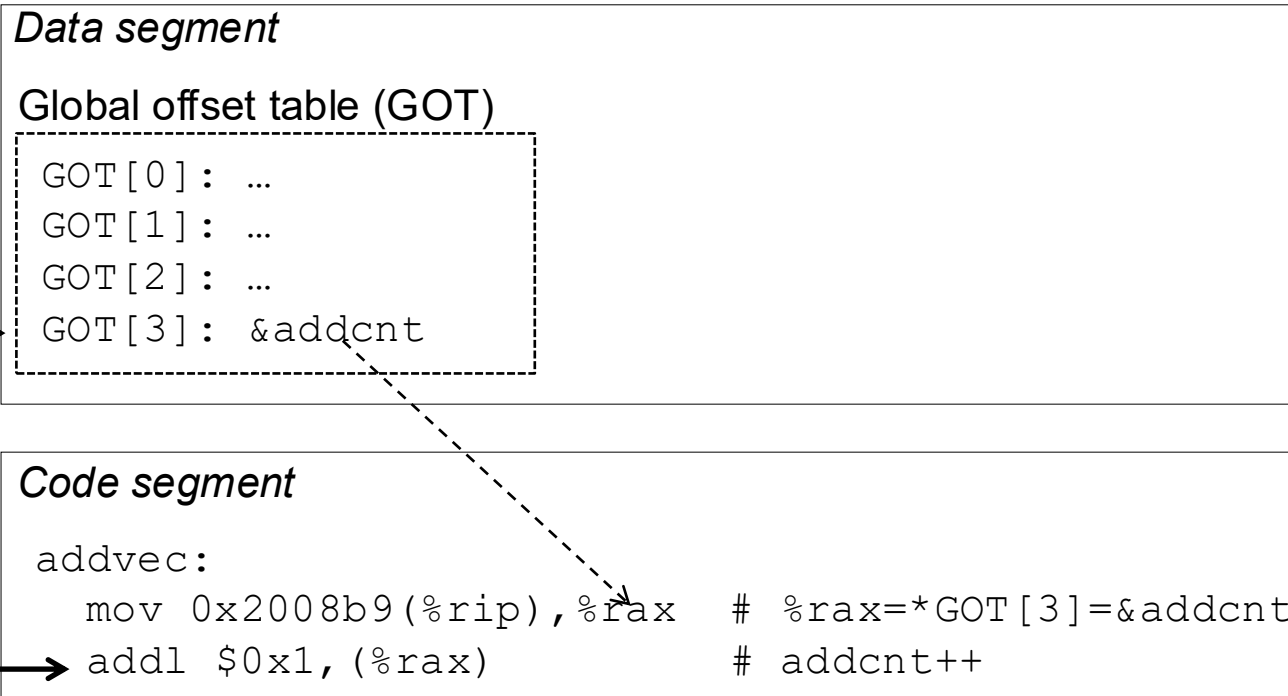


Dynamic Linking at Load-time



Dynamic Linking:

Global Offset Table(GOT)

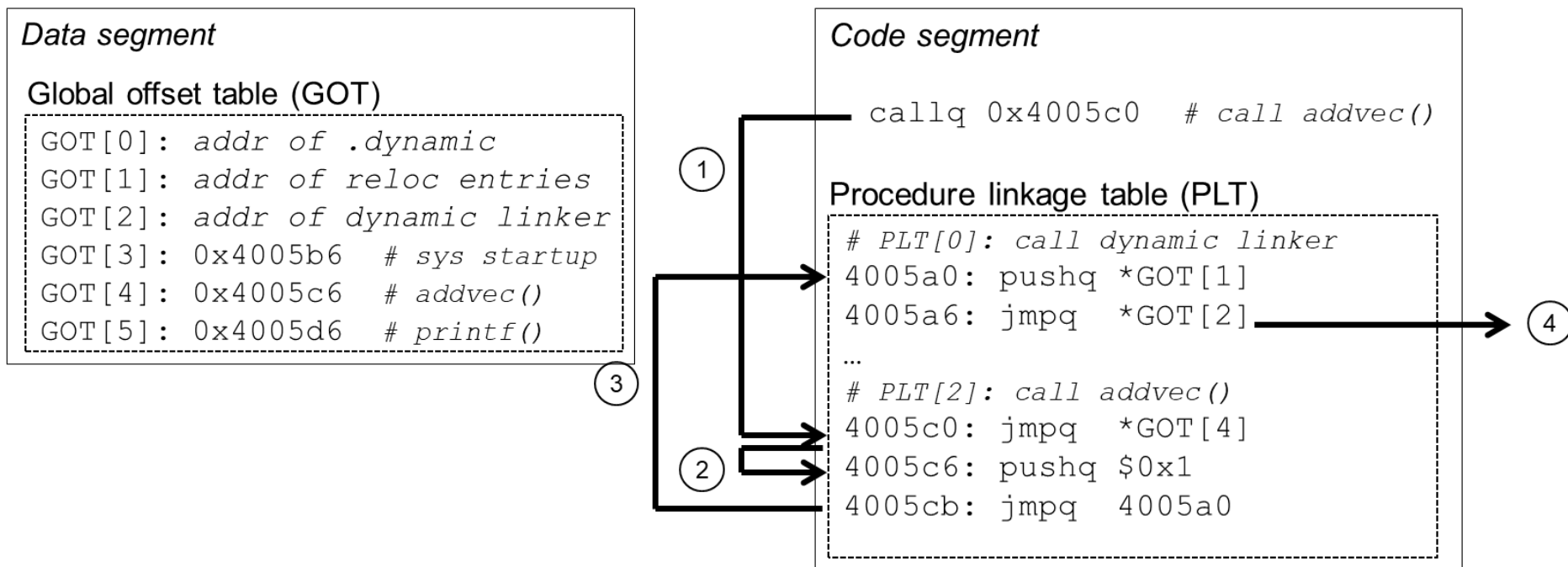


Fixed distance of
0x2008b9 bytes
at run time
between GOT[3]
and addl
instruction.

- The GOT is an array of pointers, with one entry per dynamically linked function (and then a few)
- The dynamic linker populates this table as it links each such function at runtime
- Calls to dynamically linked functions are made using indirection via the corresponding entry in the GOT

Dynamic Linking:

Procedure Linkage Table (PLT): Initial



- Initially, all entries in the GOT point right back to the next instruction in the PLT
- This instruction, and the one that follows it, call the dynamic linker, which in turn maps in the function and adjusts the GOT to point to it

Dynamic Linking:

Procedure Linkage Table (PLT): Steady State

Data segment

Global offset table (GOT)

```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6 # sys startup
GOT[4]: &addvec ()
GOT[5]: 0x4005d6 # printf()
```

①

Code segment

```
callq 0x4005c0 # call addvec()
```

Procedure linkage table (PLT)

```
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq *GOT[2]
...
# PLT[2]: call addvec()
4005c0: jmpq *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq 4005a0
```

②

- Once the dynamic linker is done, the GOT contains the address of the function
- The dynamic linker calls the newly linked function
- Subsequent calls will succeed without the intervention of the dynamic linker.

Dynamic Linking API for Run-time Linking

```
#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 API (cont'd)

```
...

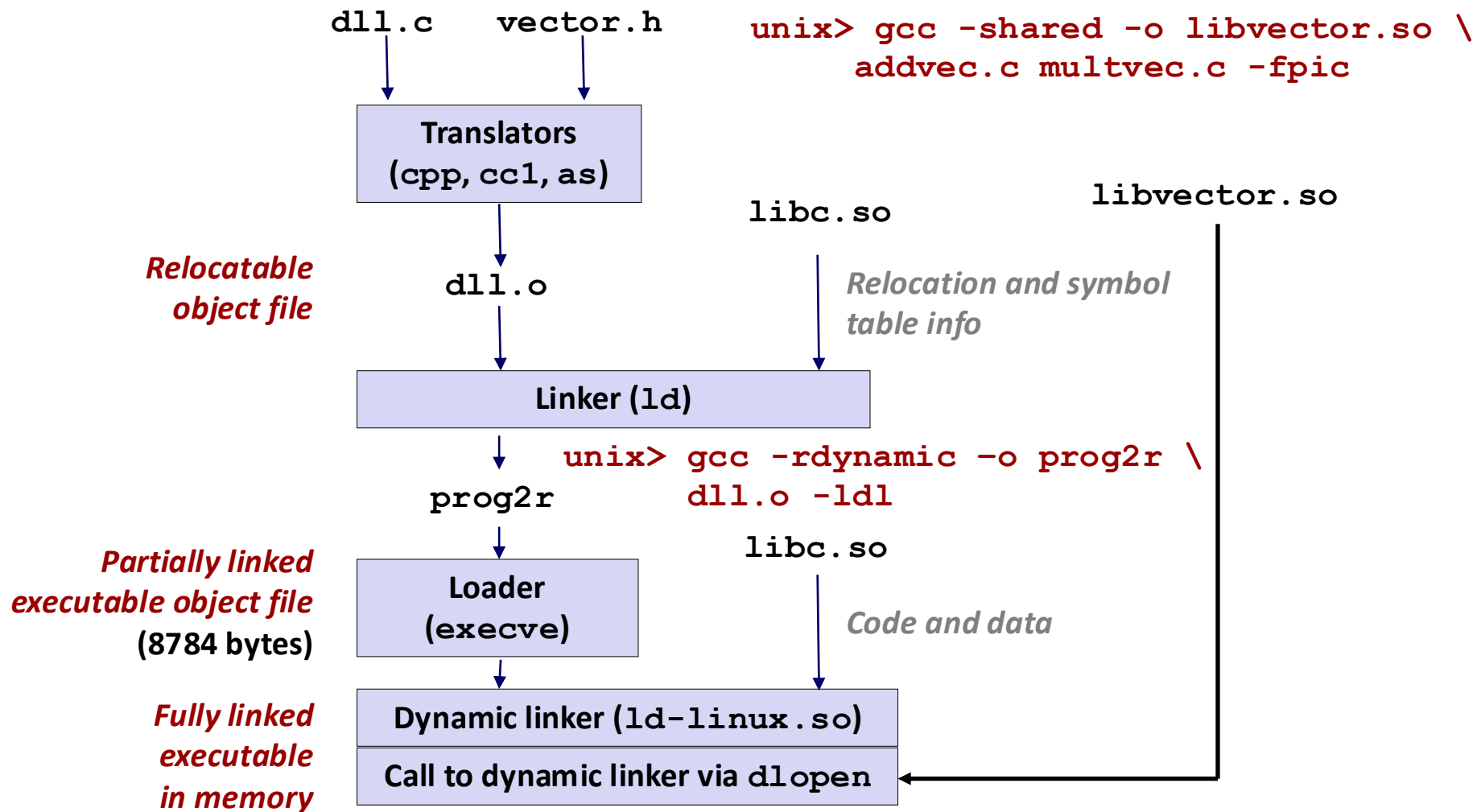
/* Get a pointer to the addvec() function we just loaded */
addvec = dlsym(handle, "addvec");
if ((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(1);
}

/* 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 API



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**

Today

- **Linking**
 - Motivation
 - What it does
 - How it works
 - Shared libraries and dynamic linking
- **Case study: Library interpositioning**

Case Study: Library Interpositioning

- Documented in Section 7.13 of book
- **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 (used to create malloc lab 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

```
#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;
}
```

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

mymalloc.c

Load/Run-time Interpositioning

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

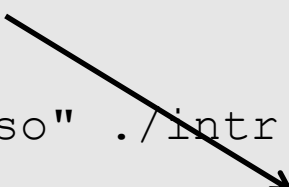
    if (!ptr)
        return;

    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
```

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)
```

Linking Recap

- **Usually: Just happens, no big deal**
- **Sometimes: Strange errors**
 - Bad symbol resolution
 - Ordering dependence of linked .o, .a, and .so files
- **For power users:**
 - Interpositioning to trace programs with & without source