Shared Libraries & Dynamic Linking

Introduction to Computer Systems
16th Lecture, Nov. 18, 2021

Instructors:

Class 1: Chen Xiangqun, Sun Guangyu , 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**
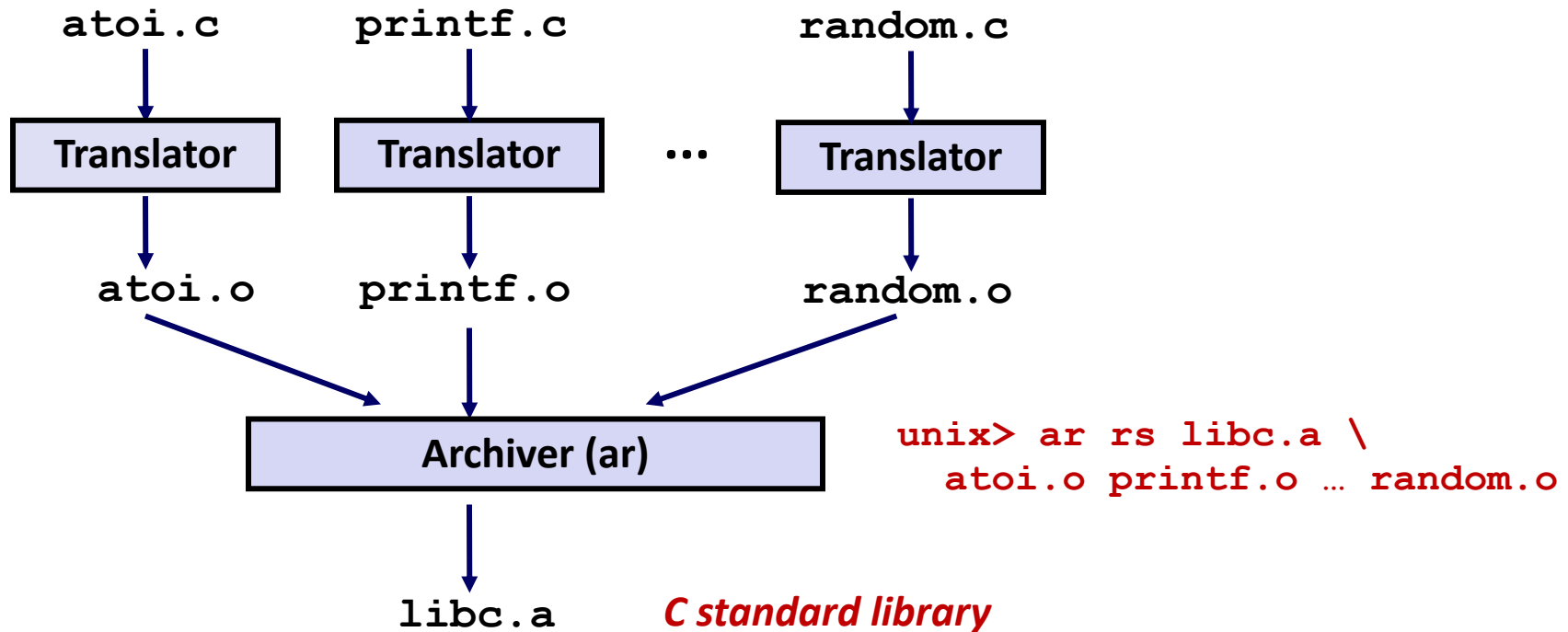
Libraries: Packaging a Set of Functions

- **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

atoi.c   **printf.c**   **random.c**

Translator   Translator **...** Translator

atoi.o   **printf.o**   **random.o**

Archiver (ar)

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

libc.a   *C standard library*

- **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 /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

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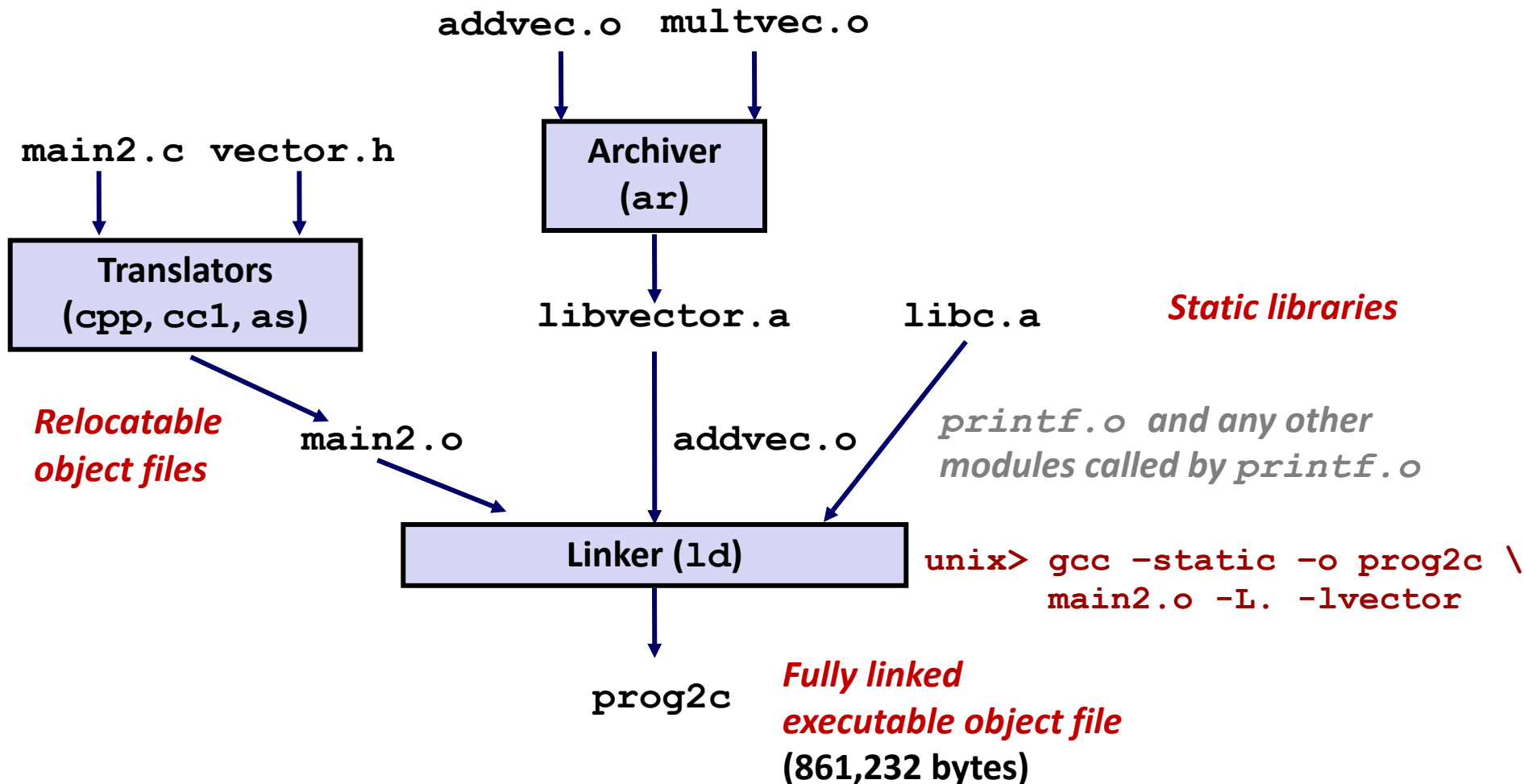
```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    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

addvec.o   **multvec.o**

main2.c vector.h

Archiver (ar)

Translators (cpp, cc1, as)

libvector.a   **libc.a**   *Static libraries*

Relocatable object files   **main2.o**   **addvec.o**   *printf.o and any other modules called by printf.o*

Linker (ld)

unix> gcc –static –o prog2c \
        main2.o -L. -lvector

prog2c   *Fully linked executable object file* (861,232 bytes)

"c" for "compile-time"

8

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
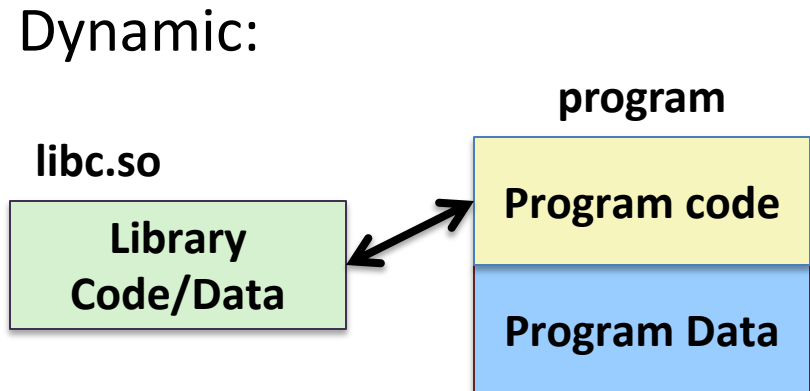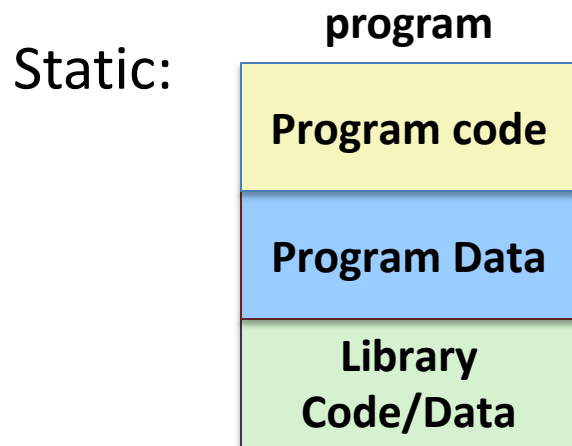
- **Modern solution: Shared Libraries**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, `.so` files

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

Static vs. Dynamic Linking

- **Static linking – required code and data is copied into executable at compile time**

- **Dynamic linking – required code and data is linked to executable at runtime**

Static:

program

Program code
Program Data
Library Code/Data

Dynamic:

libc.so

Library Code/Data

program

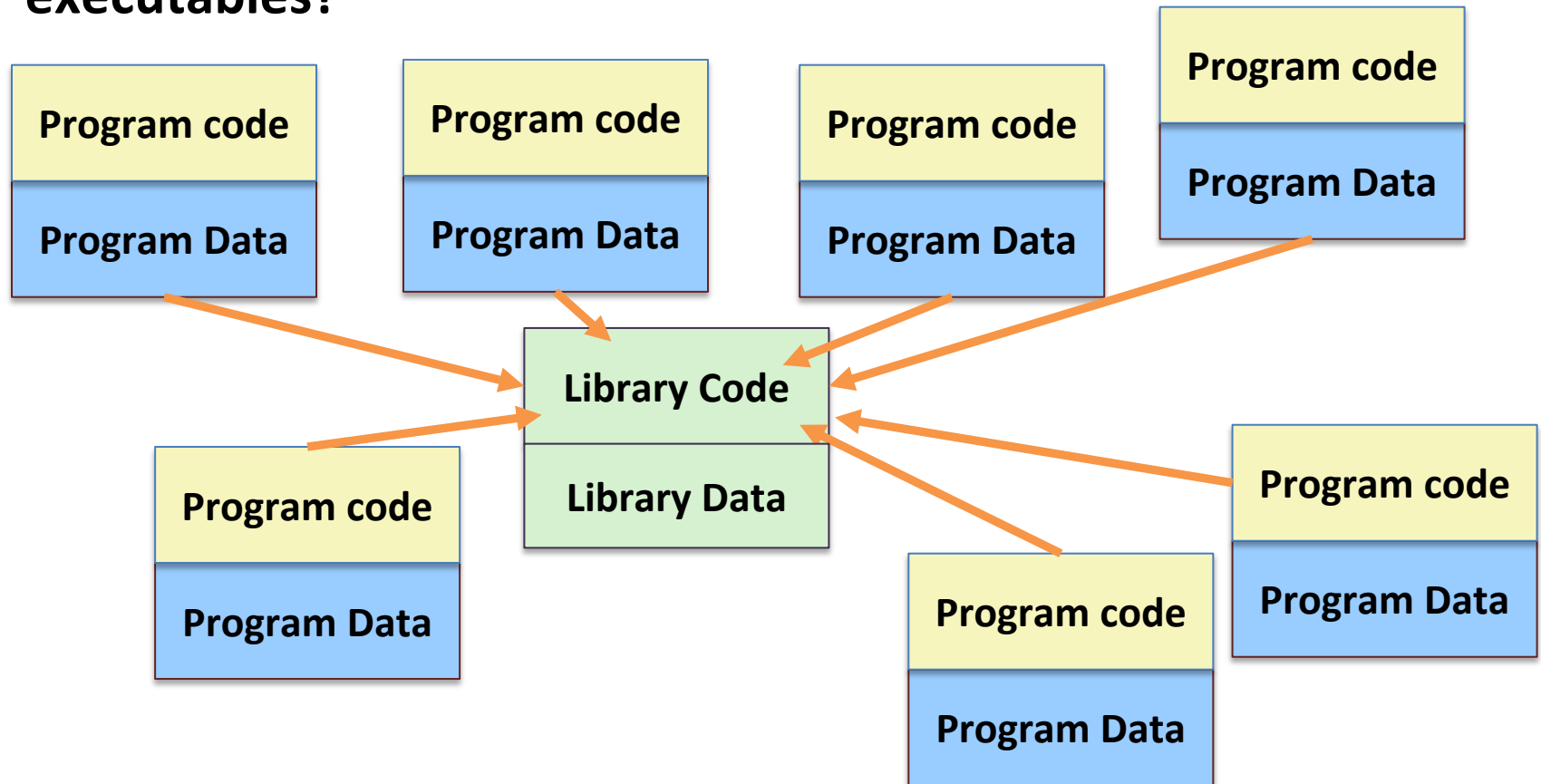| |
|---|
| **Program code** |
| **Program Data** |

Static Linking

- **With multiple programs, what happens to the size of the executables?**
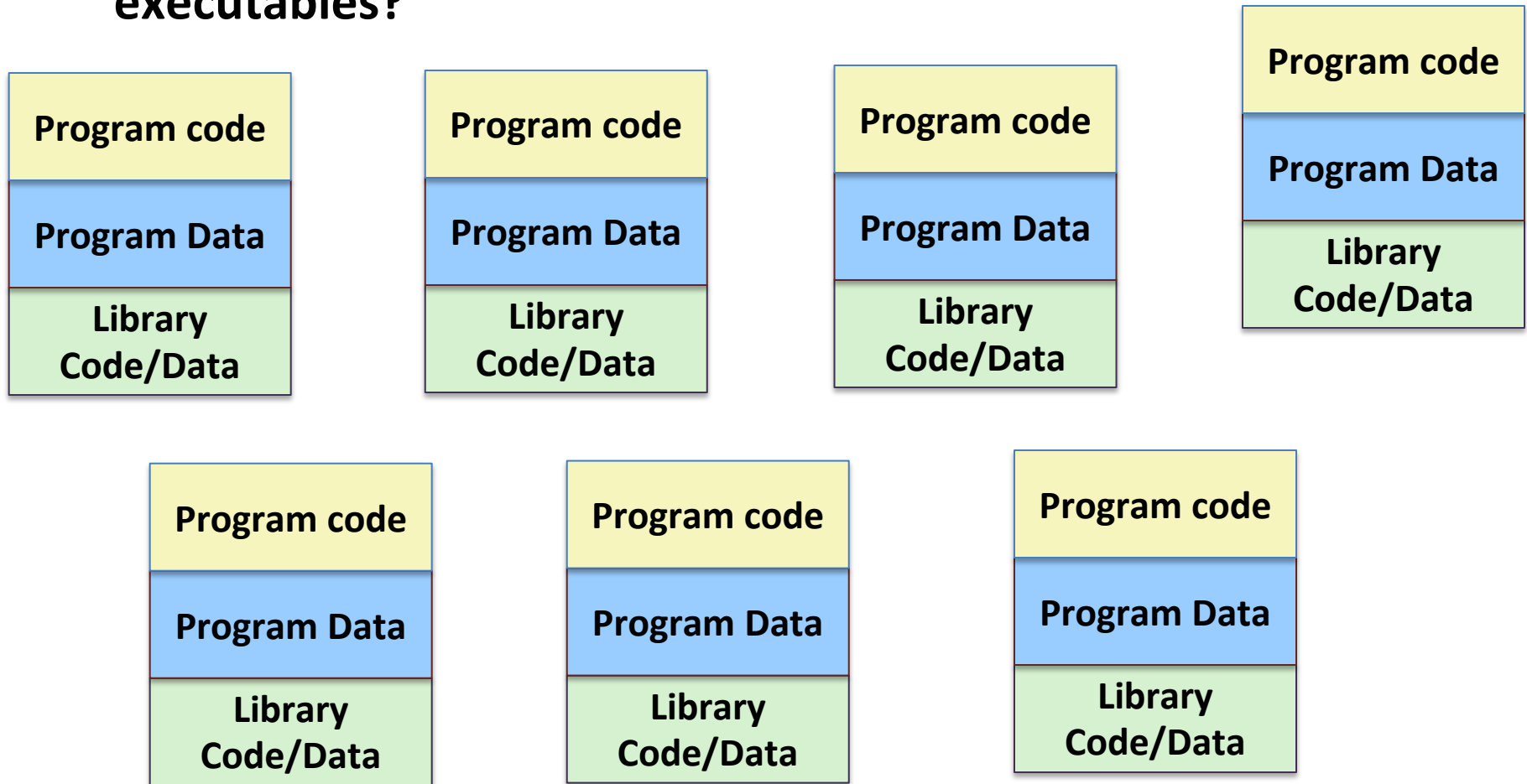
Dynamic Linking

- **With multiple programs, what happens to the size of the executables?**

- **How about global data of libraries? Copy on write.**

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
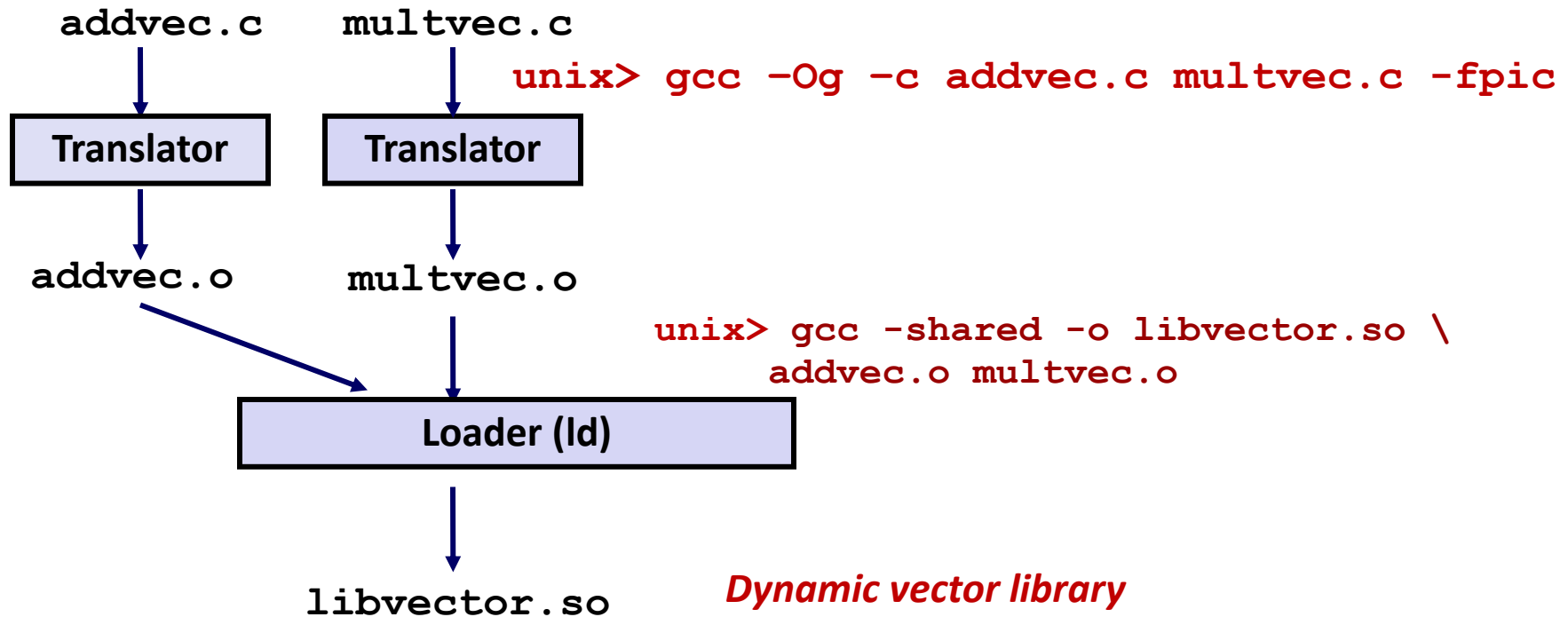 - Follow an example of `prog`

  `(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

```
addvec.c    multvec.c
```

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

Translator   **Translator**

```
addvec.o    multvec.o
```

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

Loader (ld)

```
libvector.so
```

Dynamic vector library

Dynamic Linking at Load-time

```
main2.c    vector.h
```

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

Translators
(cpp, cc1, as)

```
libc.so
libvector.so
```

Relocatable
object file

```
main2.o
```

Relocation and symbol
table info

Linker (ld)

```
unix> gcc –o prog2l \
           main2.o ./libvector.so
```

Partially linked
executable object file
(8488 bytes)

```
prog2l
```

Loader
(execve)

```
libc.so
libvector.so
```

Code and data

Fully linked
executable
in memory

Dynamic linker (ld-linux.so)

17

Relevant GCC Options for Dynamic Linking

- **–shared, -fpic: To create position independent code (next slide)**
  - -fpic is similar to -fPIC, with restrictions on some platforms. Lowercase version is preferred here.
  - -fpie/-fPIE is similar to -fpic/-fPIC, just for executables.
  - Using "*readelf –d foo.so | grep TEXTREL*" to check if a DSO is PIC.

- **–o something.so: To output result as a DLL**

- **–rdynamic: Includes dynamic symbol names for gprof, gdb**

- **–ldir: "dir" is the directory to look for the .so file in**
  - Alternative: -L<dir> -llibname

Position-Independent Code

- **If the load address for a program is not fixed (e.g., shared libraries), we use *position independent code*.**

- **Basic idea: separate code from data; generate code that doesn't depend on where it is loaded.**

- **PC-relative addressing can give position-independent code references.**
  - *This may not be enough, e.g.: data references, instruction peculiarities (e.g., **call** instruction in Intel IA32 x86) may not permit the use of PC-relative addressing.*

- **Slightly less efficient than absolute addresses**
- **Commonly used today**

Position-Independent Code

- **ELF executable file characteristics:**
  - data pages follow code pages;
  - the offset from the code to the data does not depend on where the program is loaded.

- **The linker creates a *global offset table* (GOT) that contains offsets to all global data used.**

- **If a program can load its own address into a register, it can then use a fixed offset to access the GOT, and thence the data.**

PIC code on ELF(x86 & x86-64)

- **Code to figure out its own address (x86)**

  call L   /* push address of next instruction on stack */

  L: pop %ebx /* pop address of this instruction into %ebx */

  RIP-relative addressing is new for x64 and allows accessing data tables and such in the code relative to the current instruction pointer, making position independent code easier to implement.

- **Accessing a global variable x in PIC:**

  - GOT has an entry, say at position k, for x. The dynamic linker fills in the address of x into this entry at load time.

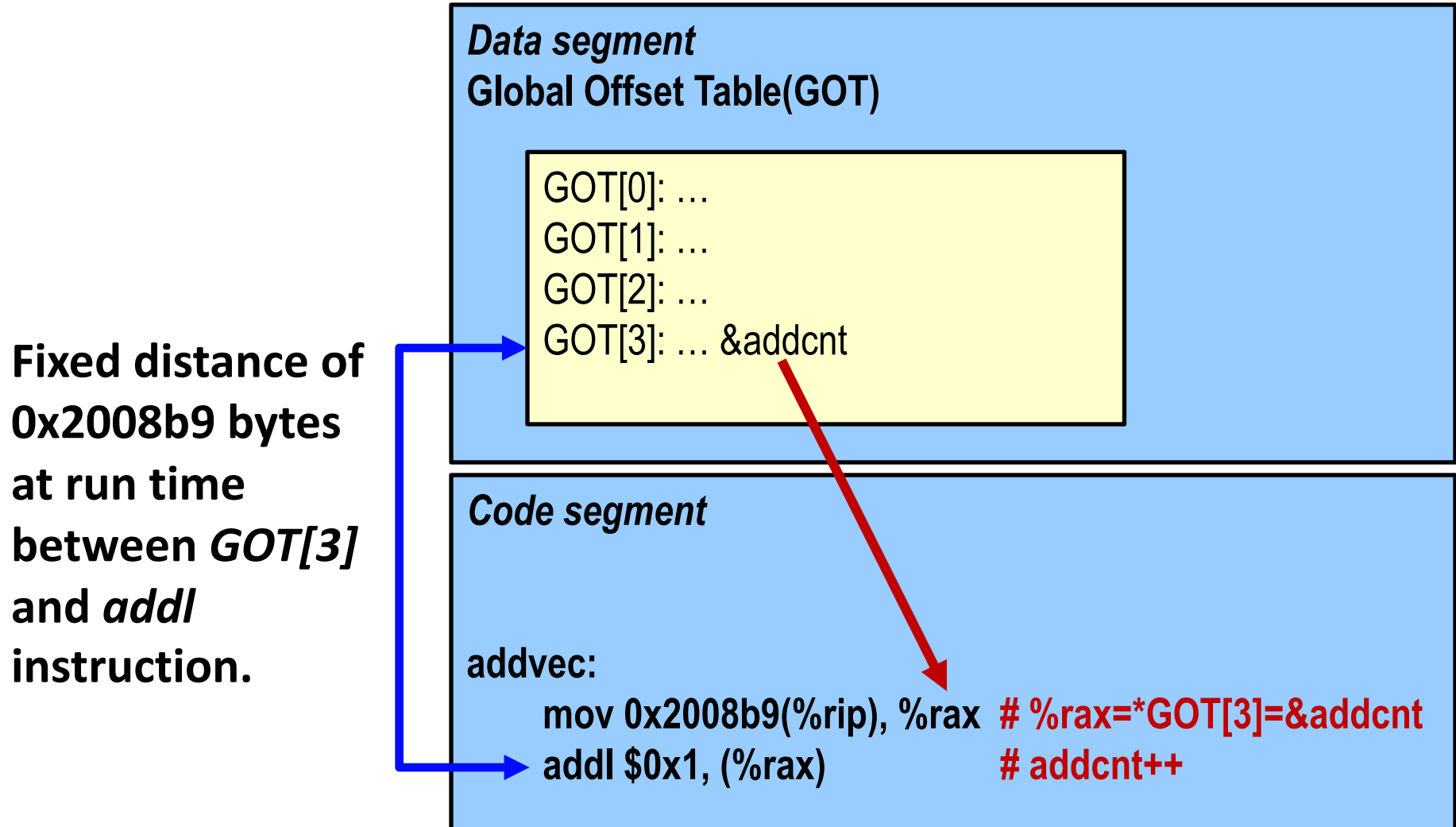 Compute "current address" into a register, say %ebx (above)

  %ebx += offset_to_GOT;   /* fixed for a given program */

  %eax = contents of location k (%ebx) /* %eax = addr. of x */

  Or directly mov offset_to_GOT(%rip), %rax in x86-64

  - access memory location pointed at by %eax/%rax(x86-64)

Using the GOT to Reference a Global Variable

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

Data segment
Global Offset Table(GOT)

GOT[0]: …
GOT[1]: …
GOT[2]: …
GOT[3]: … &addcnt

Code segment

addvec:
    mov 0x2008b9(%rip), %rax # %rax=*GOT[3]=&addcnt
    addl $0x1, (%rax)   # addcnt++

PIC: Advantages and Disadvantages

■ **Advantages:**

- ▪ Code does not have to be relocated when loaded.
  (However, data still need to be relocated.)

- ▪ Different processes can share the memory pages of code, even if they don't have the same address space allocated.

■ **Disadvantages:**

- ▪ GOT needs to be relocated at load time.

  In big libraries, GOT can be very large, so this may be slow.

- ▪ PIC code is bigger and slower than non-PIC code.
  The slowdown is architecture dependent (in an architecture with few registers, using one to hold GOT address can affect code quality significantly.)

Dynamic loading requires that the shared library be relocatable, but more…

- **With mapped files (Linux mmap API), the segment can be a different base address in each process.**

- **So… not only does each process see the DLL at a different location in memory, the DLL sees *itself* there too!**

- **And in fact each also has its own data segment**

Solution Involves Two Aspects

- We compile the library with –shared –fpic. This tells the compiler to generate "register offset" addressing

- Then, at runtime, whenever we call into the shared library, we need to put the code segment base address in a specific register (save the old value to the stack!), and the data segment base into a second register. Restore the original values when the method returns.

- With –fpic, all jumps and data accesses in the DLL are "relativized" as offsets with respect to these registers.

Linking at Load-time/Run-time

- **Procedure Linkage Table (PLT)**
  - Used by dynamic linker to resolve locations of library functions
  - All function calls to shared libraries are replaced by stub functions that query the PLT at runtime for the address of the function

- **Global Offset Table (GOT)**
  - Used by dynamic linker to resolve locations of library data
  - All references to variables in shared libraries are replaced with references to the GOT
  - Shared libraries may be compiled with Position Independent Code (allows branch and jump instructions to **relative** addresses with respect to GOT)

Linking at Load-time/Run-time

- **At compile & assembly time:**
  - The linker (ld) location is embedded in program
  - Addresses of dynamic functions are replaced with calls to the linker

- **At runtime the linker does lazy-binding:**
  - The program runs as normal until it encounters an unresolved function
  - Execution jumps to the linker
  - The linker maps the shared library into the process' address space and replaces the unresolved address with the resolved address
  - "lazy update of GOT"

Load-time & Run-time Relocations

- **If a program uses a library, the operating system maps it into memory. The single copy can then be shared**

- **Then a "dynamic linking" module runs to connect the executable to the mapped library segment.**
  - It may have a different base address in each address space, creating a need for dynamic relocation.
  - We also create a copy of the data segments of the library for each process using it, so that any changes are private.

PLT & GOT

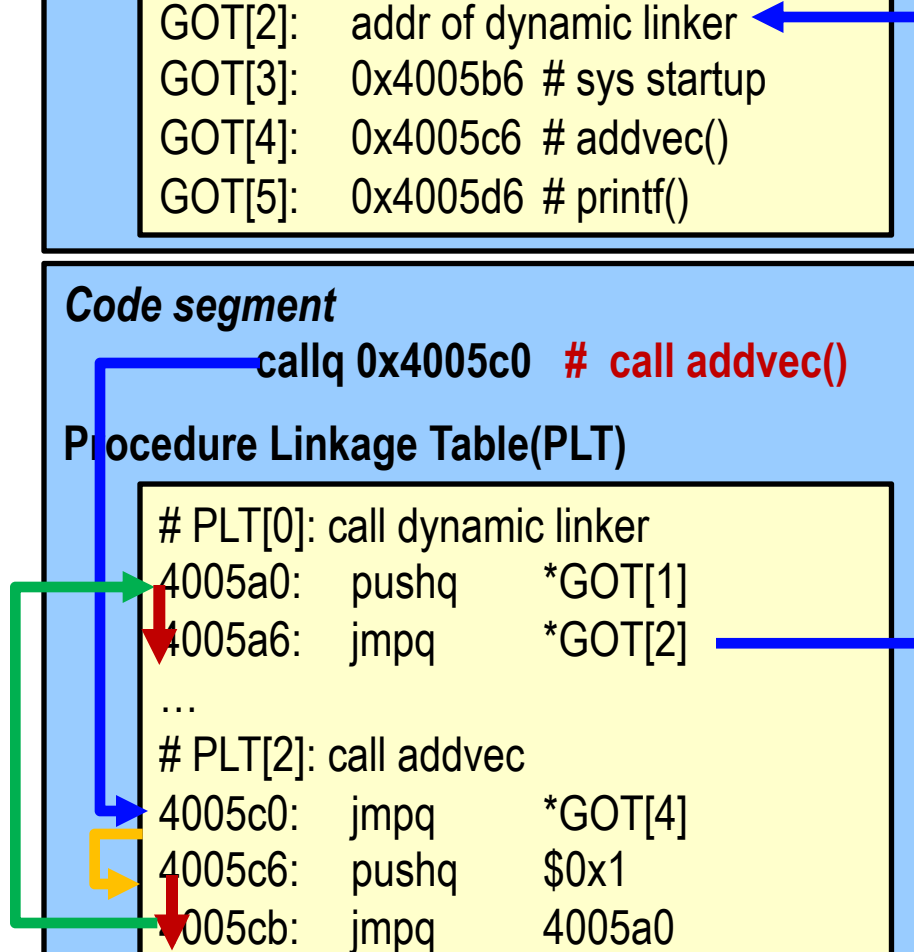**Data segment**
Global Offset Table(GOT)

GOT[0]:   addr of .dynamic
GOT[1]:   addr of reloc entries
GOT[2]:   addr of dynamic linker
GOT[3]:   0x4005b6 # sys startup
GOT[4]:   0x4005c6 # addvec()
GOT[5]:   0x4005d6 # printf()

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

29

PLT & GOT

Data segment （**WRITABLE!**）
Global Offset Table(GOT)

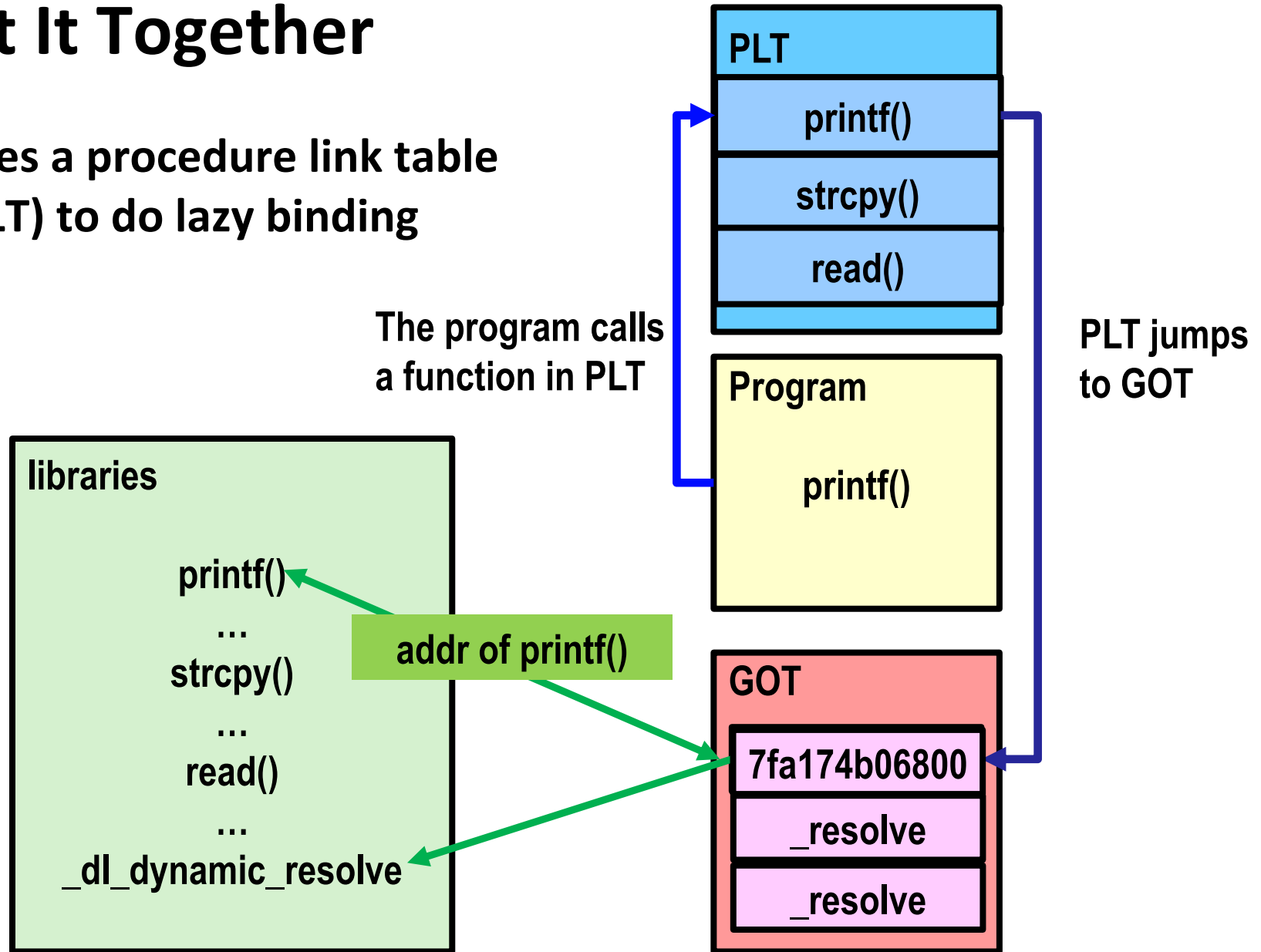|	
GOT[0]:	addr of .dynamic
GOT[1]:	addr of reloc entries
GOT[2]:	addr of dynamic linker
GOT[3]:	0x4005b6 # sys startup
GOT[4]:	**&addvec**
GOT[5]:	0x4005d6 # printf()

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
```

Put It Together

- **Uses a procedure link table (PLT) to do lazy binding**

PLT

| printf() |
| strcpy() |
| read() |

The program calls a function in PLT

PLT jumps to GOT

Program

printf()

libraries

printf()
…
strcpy()
…
read()
…
_dl_dynamic_resolve

addr of printf()
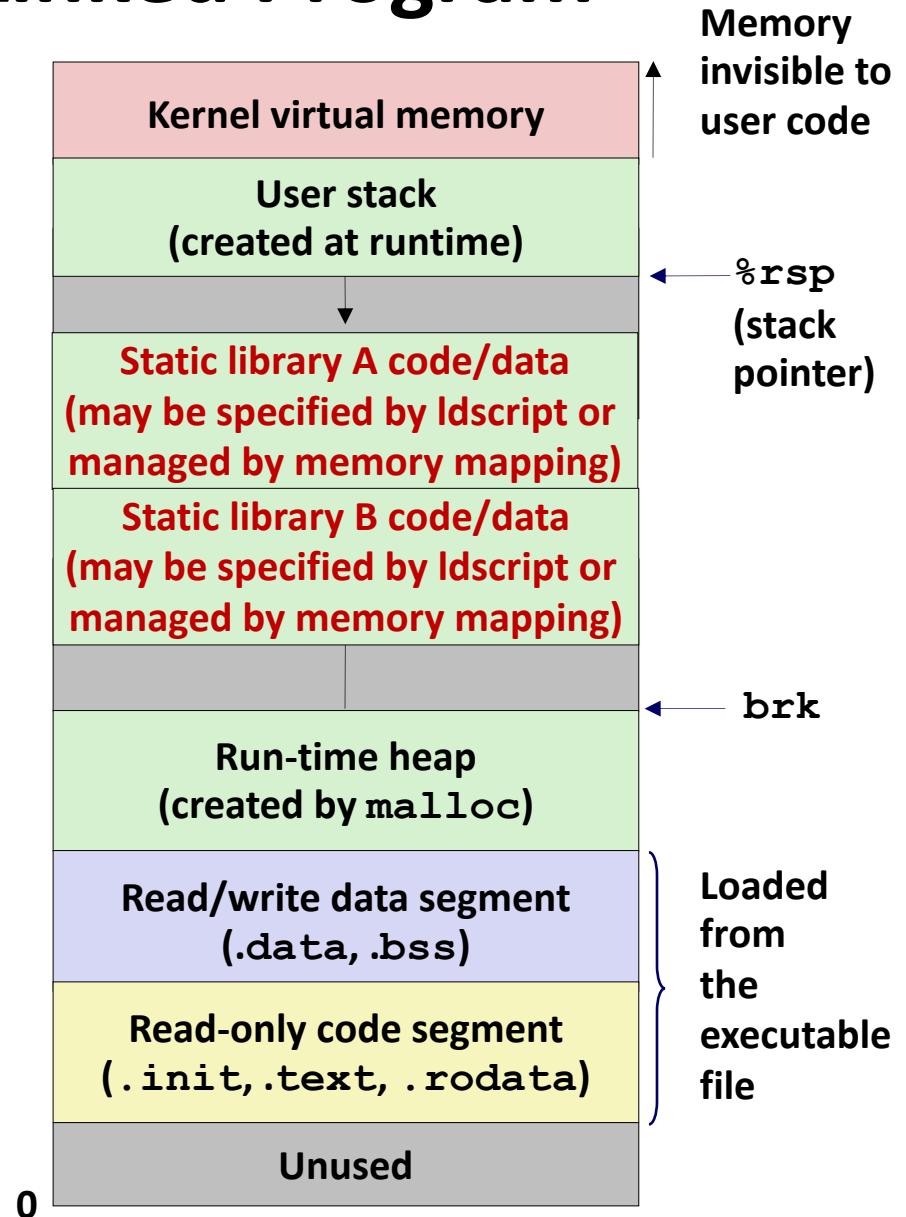
GOT

| 7fa174b06800 |
| _resolve |
| _resolve |

GOT stores addr of _dl_dynamic_resolve

31

Running a Statically Linked Program

- **All functions and data needed by the process space are linked as the last step of the compiler**

- **Only that code/data needed by the program are loaded into virtual memory**

- **ASLR is not considered**

Default 0x400000

Kernel virtual memory	Memory invisible to user code
User stack (created at runtime)	
	`%rsp` (stack pointer)
Static library A code/data (may be specified by ldscript or managed by memory mapping)	
Static library B code/data (may be specified by ldscript or managed by memory mapping)	
	`brk`
Run-time heap (created by `malloc`)	
Read/write data segment (`.data`, `.bss`)	Loaded from the executable file
Read-only code segment (`.init`, `.text`, `.rodata`)	
Unused	

0

32

```c
int main(void) {
    ...
    printf("main is : %p\n", &main);
    return 0;
}
```

./hello
  Hello World!
  main is : 0x400544
readelf --headers ./hello | grep "Entry point"
  Entry point address: 0x400460
objdump --disassemble –M intel ./hello

...

0000000000400460 <_start>:

- Entry point != &main
- Most compilers insert extra code into compiled programs
- Those code typically runs before and after main()
- Such as .init & .fini for C++

```
  400460:       31 ed                   xor    ebp,ebp
  400462:       49 89 d1                mov    r9,rdx
  400465:       5e                      pop    rsi
  400466:       48 89 e2                mov    rdx,rsp
  400469:       48 83 e4 f0             and    rsp,0xfffffffffffffff0
  40046d:       50                      push   rax
  40046e:       54                      push   rsp
  40046f:       49 c7 c0 20 06 40 00    mov    r8,0x400620
  400476:       48 c7 c1 90 05 40 00    mov    rcx,0x400590
  40047d:       48 c7 c7 44 05 40 00    mov    rdi,0x400544
  400484:       e8 c7 ff ff ff          call   400450 <__libc_start_main@plt>
```
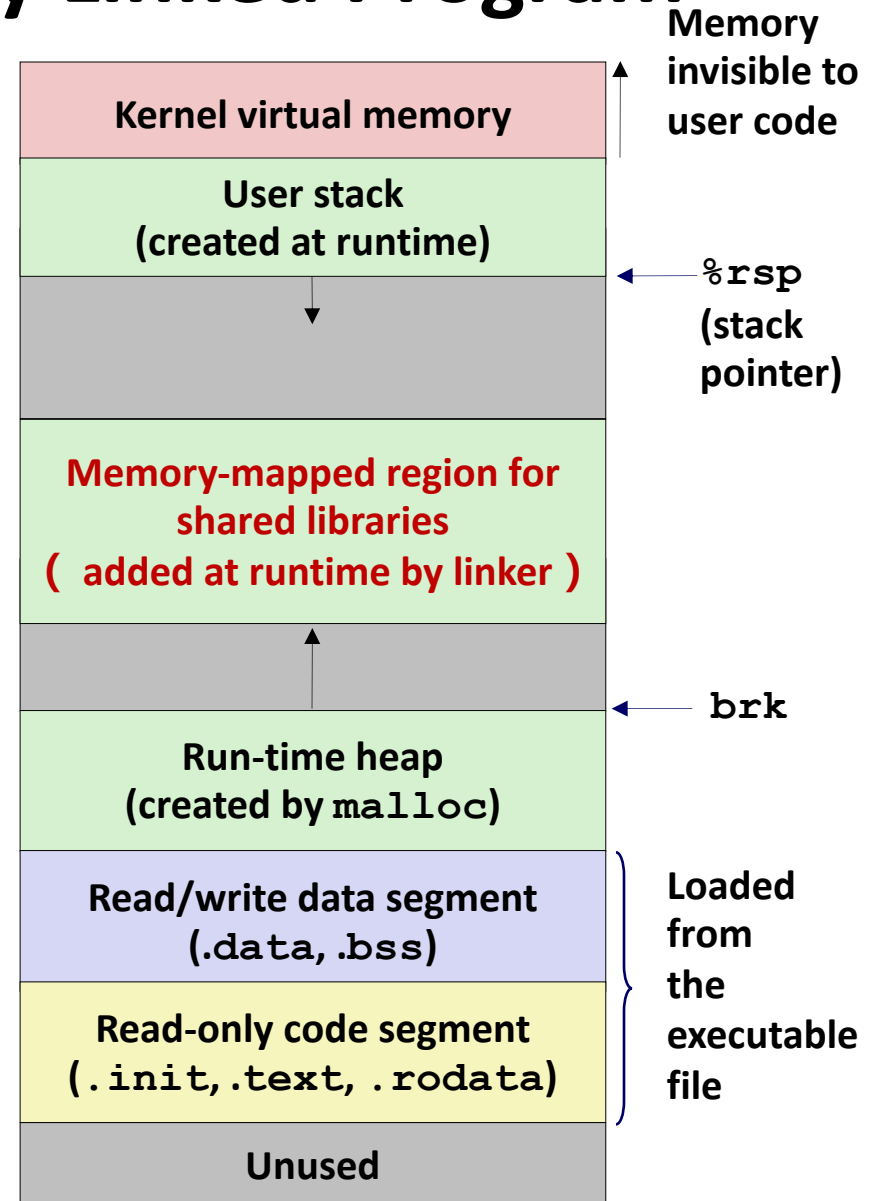
...

Running a Dynamically Linked Program

- **Some functions and data do not exist in process space at runtime**

- **The dynamic linker (called *ld*) maps these into the memory map segment on-demand**

Memory invisible to user code

Kernel virtual memory
User stack **(created at runtime)**

`%rsp` (stack pointer)

Memory-mapped region for shared libraries **(added at runtime by linker)**

`brk`

Run-time heap **(created by `malloc`)**
Read/write data segment **(`.data`, `.bss`)**
Read-only code segment **(`.init`, `.text`, `.rodata`)**
Unused

Loaded from the executable file

Default 0x400000

How about ld.so Itself?

- **Bootstrapping**

  - ld.so helps other to relocate, who helps ld.so?

  - Could not use global variables and functions before GOT/PLT relocated.

  - Implemented in glibc, elf/rtld.c

```
/* Now life is sane; we can call functions and access global data.
```

- **Shared library or executable?**

  - Both, ".interp" section could be regarded as entry point for DLLs.

- **Dynamic linking or static linking?**

```
#ldd /lib/x86_64-linux-gnu/ld-2.15.so
        statically linked
#file /lib/x86_64-linux-gnu/ld-2.15.so
/lib/x86_64-linux-gnu/ld-2.15.so: ELF 64-bit LSB shared object,
x86-64, version 1 (SYSV), dynamically linked,
BuildID[sha1]=0x40cd912b6d6235ceee7f82ab12678fe0887bfa53, stripped
```

Dynamic Linking at Run-time

```c
#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

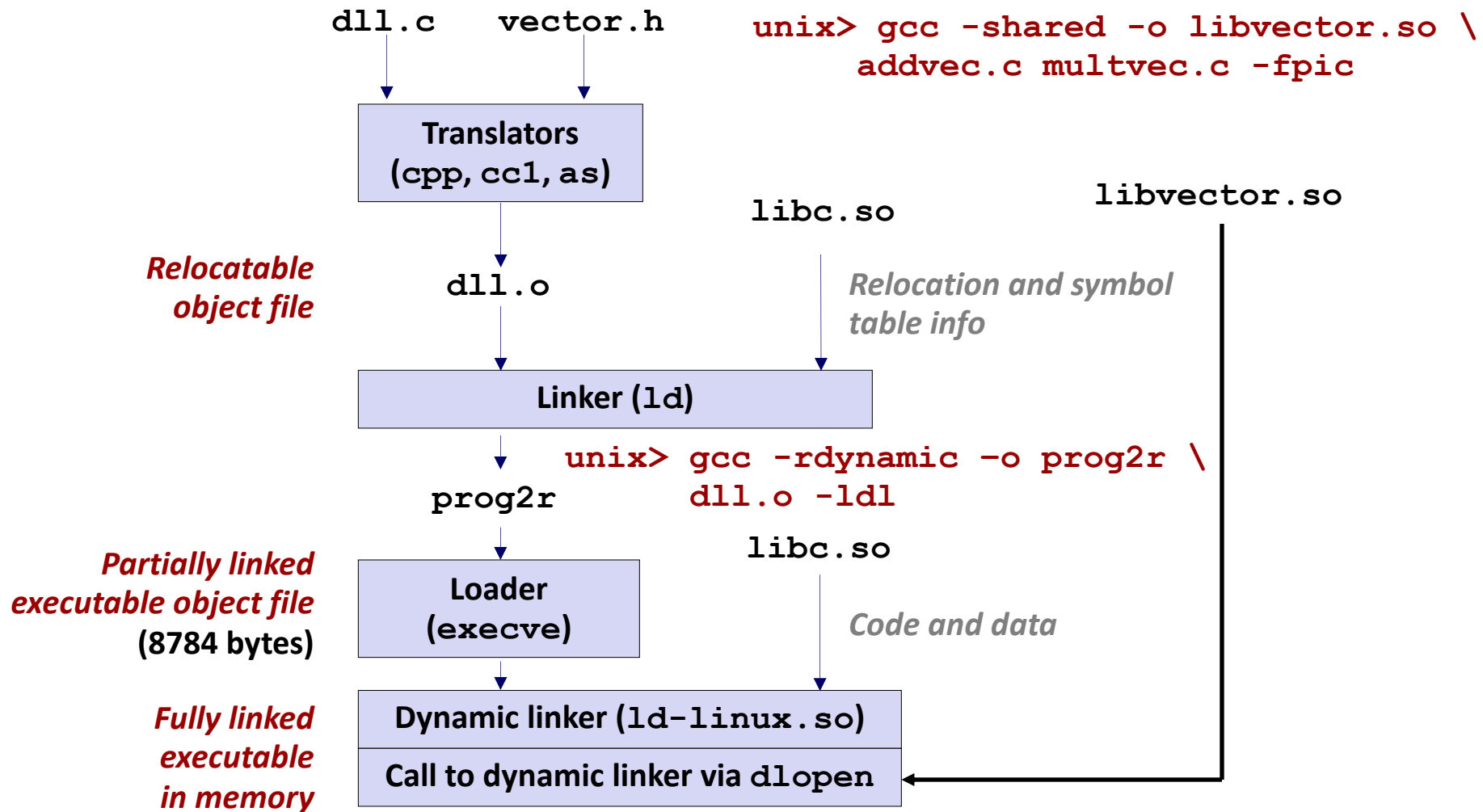# Dynamic Linking at Run-time (cont)

```c
    ...

    /* 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

```
dll.c     vector.h
```

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

Translators
(`cpp, cc1, as`)

`libc.so`

`libvector.so`

*Relocatable
object file*

`dll.o`

*Relocation and symbol
table info*

Linker (`ld`)

```
unix> gcc -rdynamic -o prog2r \
         dll.o -ldl
```

`prog2r`

`libc.so`

*Partially linked
executable object file*
(8784 bytes)

Loader
(`execve`)

Code and data

*Fully linked
executable
in memory*

Dynamic linker (`ld-linux.so`)

Call to dynamic linker via `dlopen`

Static Linking Relocation (GDB View)

■ **gcc -static -g -o prog2c main2.o -L. -lvector**

```
# gdb ./prog2c
...
This GDB was configured as "x86_64-linux-gnu".
Reading symbols from /root/prog2c...done.
(gdb) p addvec          #print the address of function addvec
$1 = {<text variable, no debug info>} 0x4011d4 <addvec>
(gdb) r                 #running the program
Starting program: /root/prog2c
z = [4 6]
[Inferior 1 (process 19506) exited normally]
(gdb) p addvec
$2 = {<text variable, no debug info>} 0x4011d4 <addvec>
```

Statically linked address.
Relocated at link time

Load-time Relocation (GDB View)

- **gcc -g -o prog2l main2.o ./libvector.so**

```
# gdb ./prog2c
Reading symbols from /root/prog2l...done.
(gdb) p addvec          #print the address of function addvec
$1 = {<text variable, no debug info>} 0x4005b0 <addvec@plt>
(gdb) b _start          #set breakpoint at the beginning
Breakpoint 1 at 0x4005c0 in _start ()
(gdb) r                 #running the program
Starting program: /root/prog2l
Breakpoint 1, 0x00000000004005c0 in _start()
(gdb) p addvec
$2 = {<text variable, no debug info>} 0x7ffff7bd8720 <addvec>
(gdb) c                 #continues the program
Continuing.
z = [4 6]
[Inferior 1 (process 19542) exited normally]
(gdb) p addvec
$2 = {<text variable, no debug info>} 0x7ffff7bd8720 <addvec>
```

Dynamically linked address. Relocated at load time.
Address changed after loading

Run-time Relocation (GDB View)

- **gcc -g -rdynamic -o prog2r dll.c -ldl**

```
# gdb ./prog2r
Reading symbols from /root/prog2r...done.
(gdb) p addvec          #print the address of function addvec
No symbol "addvec" in current context.
(gdb) b main    #set breakpoint at main()
Breakpoint 1 at 0x400a0c
(gdb) b 16      # handle = dlopen("./libvector.so", RTLD_LAZY);
Breakpoint 2 at 0x400a10: file dll.c, line 16.
(gdb) b 23      # addvec = dlsym(handle, "addvec");
Breakpoint 3 at 0x400a55: file dll.c, line 23.
(gdb) b 30      # addvec(x, y, z, 2);
Breakpoint 4 at 0x400aa1: file dll.c, line 30.
(gdb) r                 #running the program
Starting program: /root/prog2l
Breakpoint 1, 0x0000000000400a0c in main ()
(gdb) p addvec          #print the address of function addvec
No symbol "addvec" in current context.
```

Dynamically linked address. Relocated at run time.
Symbol not found in context until dlopen().

Runtime Relocation (GDB View)

```
(gdb) c                    #continues
Breakpoint 2, 0x0000000000400a10 in main ()
(gdb) p addvec
No symbol "addvec" in current context.
(gdb) c                    #continues
Breakpoint 3, 0x0000000000400a55 in main ()
(gdb) p addvec
$1 = {<text variable, no debug info>} 0x7ffff7613720 <addvec>
(gdb) c                    #continues
Breakpoint 4, 0x0000000000400aaf in main ()
(gdb) p addvec
$2 = {<text variable, no debug info>} 0x7ffff7613720 <addvec>
(gdb) c                    #continues
Continuing.
z = [4 6]
[Inferior 1 (process 19720) exited normally]
(gdb) p addvec
No symbol "addvec" in current context.
```

Dynamically linked address. Relocated at run time.
Symbol relocated after dlopen(), removed after dlclose().

Runtime Errors

- **At runtime, your program searches for the .so file**
  - $LD_PRELOAD, $LD_LIBRARY_PATH

- **What if it can't find it?**
  - You will get an error message during execution, and the executable will terminate. Depending on the version of Linux, this occurs when you launch the program, or when it tries to access something in the dll.

- **Some dll files also have "versioning" data. On these, your program might crash because of an "incompatible dll version number"**

Static vs. Dynamic Linking Tradeoffs

Static:

- **Does not need to look up libraries at runtime**

- **Does not need extra PLT indirection**

- **Consumes more memory with copies of each library in every program**

Dynamic:

- **Less disk space/memory (7K vs 571K for hello world)**

- **Shared libraries already in memory and in hot cache**

- **Incurs lookup and indirection overheads**

Key Concepts

■ **Overview – module inclusion and linking choices**

	Compile-time	Link-time	Load-time	Run-time
Static linking	Textual source inclusion	OBJ/LIB inclusion	-	-
Dynamic linking	-	-	Reference to an external DLL	Loading an external DLL

■ **How to decide between them?**

- ▪ Small ASM-only program? #include is an option, although, it's not recommended…
- ▪ Self-contained stand-alone program? Static linking.
- ▪ For every other case dynamic linking is recommended.
- ▪ Load-time dynamic linking is easier to use and less error-prone.
- ▪ A mix of load-time + run-time dynamic linking should be used for faster application load time and/or for supporting plugin-able modules or on-the-fly updates.

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 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**
- **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

```c
#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

```c
#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

```c
#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

```c
/* 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
```
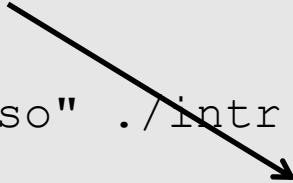
`mymalloc.c`

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)
```

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

Linux Binary Utilities (GNU binutils)

- ar – alternative to ar, to create, manipulate archive files

- strings – display textual strings (similar to strings utility)

- strip – strip ELF file from symbol tables

- nm – display symbols from object/executable files

- size – display size of each section (text, data, bss, etc)

- readelf – human-readable display of ELF files

- objdump – show information of object files

- ldd – list the shared libraries that an executable needs at runtime.

- ld – combining object and archive files

- ranlib – create index for archives for performance

- addr2line – convert addresses into line number/file name pairs.

Additional Slides

Compiler, Assembler & Linker/Loader

- **The compiler generates symbols, but regardless of the specific address.**
  - Detect name conflicts in the source code.
  - Name mangling
- **The assembler is responsible for marking the relocation entries and computes some simple relative addresses.**
  - Relative offset of local/global symbols could be determined now.
  - Weak symbols? Not the right time yet.
- **The linker and loader are responsible for relocation which relies on the final address.**
  - Link-time relocation
  - Load-time relocation
  - Run-time relocation

Compiler, Assembler & Linker/Loader

```
# cat t2.s                    local
.local foo
.type foo,@function
foo: jmp foo
     call foo
# gcc -c -o t2.o t2.s
# objdump -D t2.o
...
0000000000000000 <foo>:
0: eb fe              jmp      0 <foo>
2: e8 f9 ff ff ff callq   0 <foo>
```

```
# cat t1.s                    weak
.weak foo
.type foo,@function
foo: jmp foo
     call foo
# gcc -c -o t1.o t1.s
# objdump -D t1.o
...
0000000000000000 <foo>:
0: e9 00 00 00 00 jmpq   5 <foo+0x5>
5: e8 00 00 00 00 callq a <foo+0xa>
```

```
# cat t3.s           global/binutils-2.22
.global foo
.type foo,@function
foo: jmp foo
     call foo
# gcc -c -o t1.o t1.s
# objdump -D t1.o
...
0000000000000000 <foo>:
0: e9 00 00 00 00 jmpq   5 <foo+0x5>
5: e8 00 00 00 00 callq a <foo+0xa>
```

```
# cat t3.s           global/binutils-2.34
.global foo
.type foo,@function
foo: jmp foo
     call foo
# gcc -c -o t2.o t2.s
# objdump -D t2.o
...
0000000000000000 <foo>:
0: eb fe              jmp      5 <foo>
2: e8 00 00 00 00 callq   7 <foo+0x7>
```