CSE 2421

Linking and Relocation

- Required Reading: *Computer Systems: A Programmer's Perspective, 3rd Edition*
  - Chapter 7 through 7.6.3 (inclusive)

What are linking and relocation?

•*Linking* is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed (that is, *an executable*).

•*Relocation* is the process of *adjusting addresses* in object modules when the modules are linked with other modules to create an executable.

Why should I care?

- Will help you build large programs
  - Will help with missing modules/linker error resolution
- Will help you avoid dangerous programming errors
  - Should you choose to use global variables
- Will help you understand language scoping
- Will help you understand important system concepts
  - Virtual memory/paging/memory mapping(Systems II)
- Will help you exploit shared libraries

Linking can be done:

- At compile time
- At load time
- At run time

Related OS concepts

- When a process is running, it enhances security if the address space of the process is divided into parts which are only known to the OS:
  - Read only space:
    - Read only data (such as format strings used with printf or scanf in C) and
    - Code (i.e., instructions)
  - Read-write space: data which can be both read and written.
- Therefore, when the linker does linking and relocation, it makes a division of the address space of the executable into these parts.

Example C Program

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

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

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

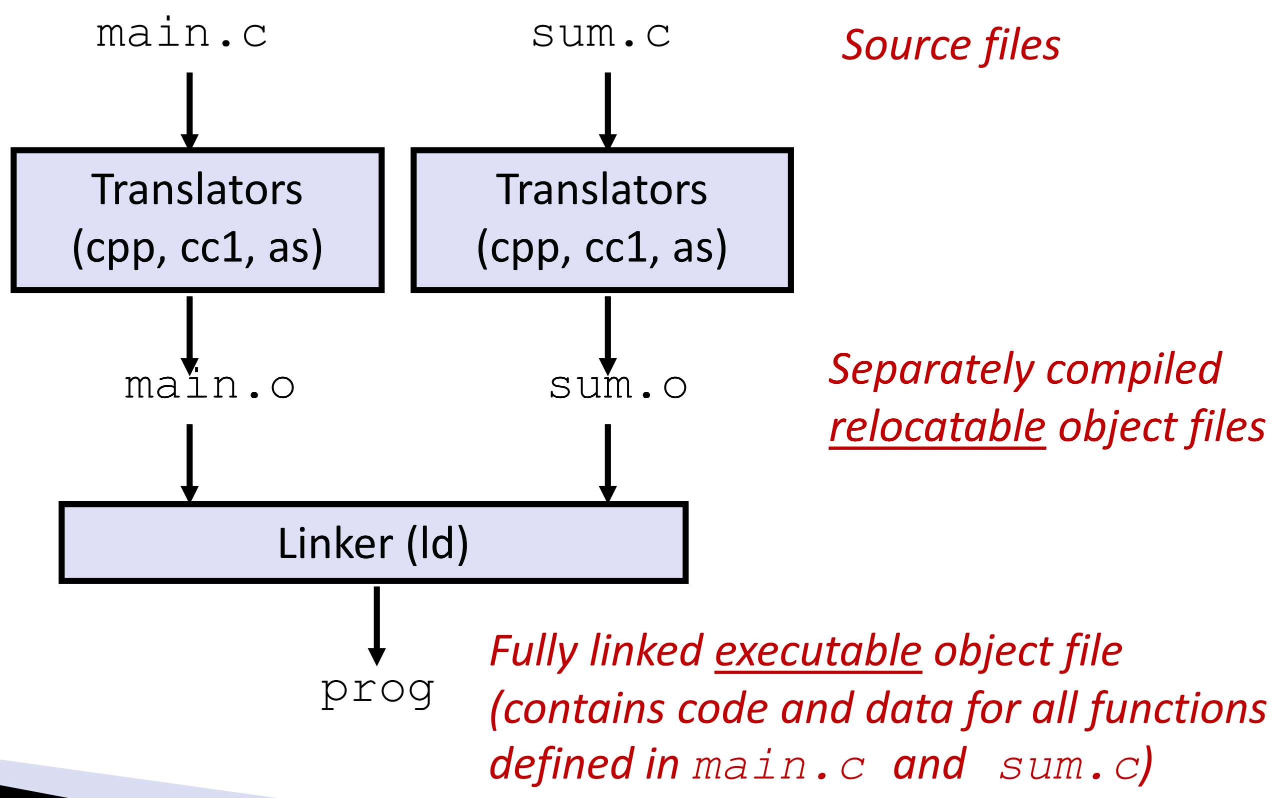
`main.c`

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

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

`sum.c`

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.
    - Consider the function of makefiles…
  - Space: Libraries
    - Common functions can be aggregated into a single file...
    - Yet executable files and running memory images contain only code for the functions they actually use.

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 `structs`
    - 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

Object File Format/Organization – ELF Object File Format (used in Unix/Linux)

•The object file formats provide parallel views of a file's contents, reflecting the differing needs of *the linker* and *the loader*

•ELF header (*Executable and Linkable Format*)

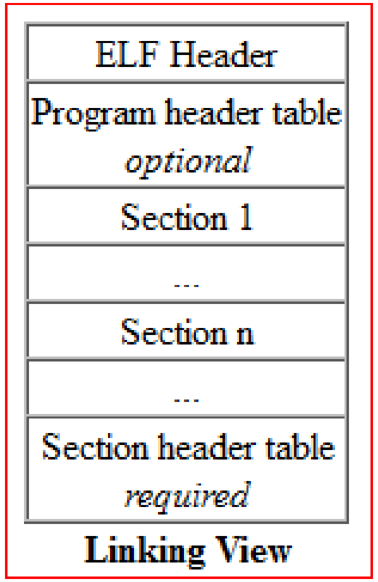
 -Resides at the beginning and holds a "road map" describing the file's organization.

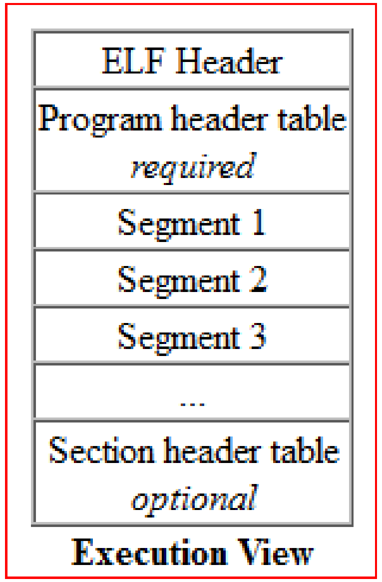
•Program (or Segment) header table

 -Tells the system how to create a process image

  -Object files used to build a process image (used by the loader), i.e., executables ***must*** have a program header table; relocatable files do not need one.

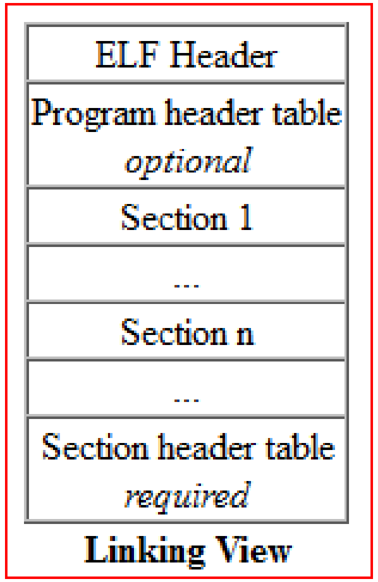
  -Object files used to do linking must have a Section header table (because it has location and size information for each section); executable object files do not need one.

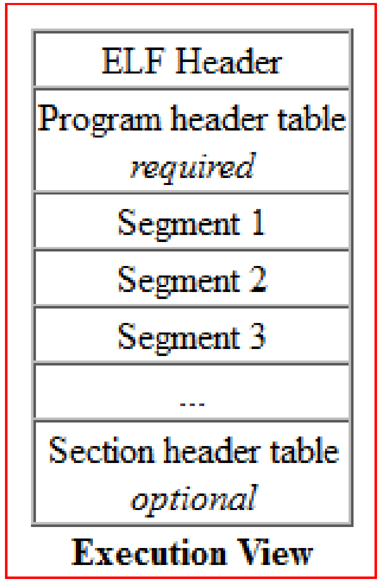




Object File Format/Organization (cont)

- Section header table
  - -Contains information describing the file's sections
  - -Every section has an entry in the table
    - -Each entry gives information such as the section name, the section size (needed to compute address information), and so on.
- Sections
  - -Hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, etc.
  - -Object files used during linking ***must*** have a section header table; other object files may or may not have one.



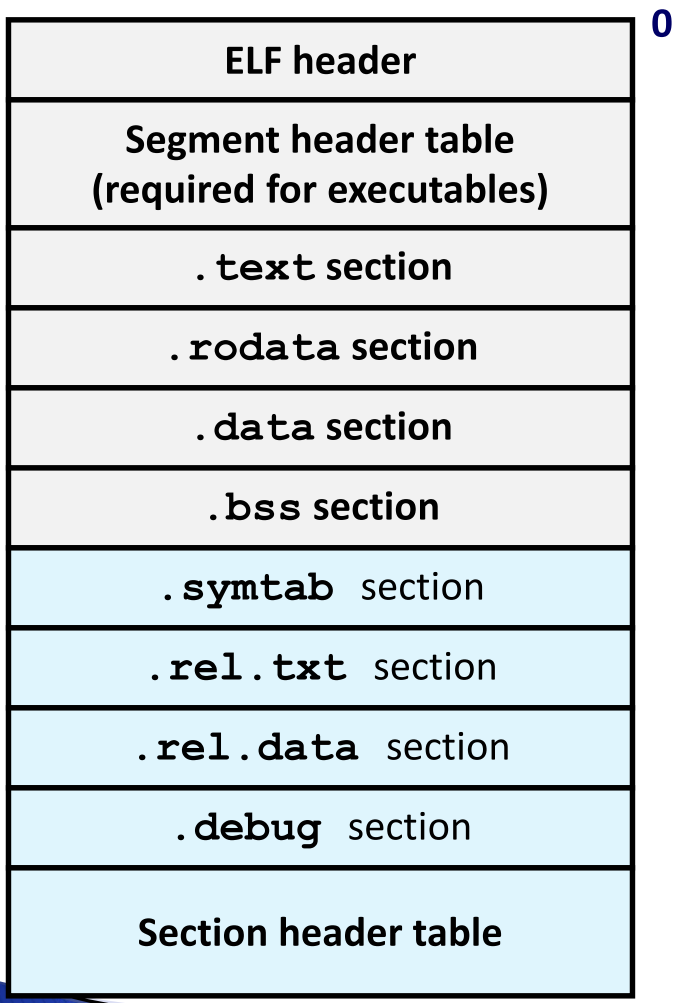


ELF Object File Format

ELF header	0
Segment header table (required for executables)	
`.text` section	
`.rodata` section	
`.data` section	
`.bss` section	
`.symtab` section	
`.rel.txt` section	
`.rel.data` section	
`.debug` section	
Section header table	

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, ...
- `.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

Referencing a global…

…that's defined here

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

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

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

main.c

Defining a global

Linker knows nothing of `val`

Referencing a global…

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

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

sum.c

Linker knows nothing of `i` or `s`

…that's defined here

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;
}
```

Compiler allocates space in .data for each definition of x

C variables in .bss aren't allocated space until execution time

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

How Linker Resolves Duplicate Symbol Definitions

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

	`p1.c`	`p2.c`	
strong	`int foo=5;`	`int foo;`	weak
strong	`p1() { }`	`p2() { }`	strong

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

`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` will overwrite `y`! Nasty!

`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
  - Initialize if you define a global variable
  - Use `extern` if you reference an external global variable

Step 2: Relocation

•Relocation merges the input modules and assigns run-time addresses to each symbol
•When an assembler generates an object module, it does not know where the code and data will ultimately be stored in memory or the locations of any externally defined functions or global variables referenced by the module
•A “relocation entry” is generated when the assembler encounters a reference to an data object, function, or jump label whose ultimate location is unknown
•2 example types (there are many such types)

- R_X86_64_PC32 For PC relative relocation, 32 bit offset
- R_X86_64_64 Absolute relocation, 64 bit address

•A PC relative “address” is not an address at all! It is *a displacement* which is added to the current PC to get the PC for the next instruction. Jump instructions and some calls use PC relative addressing.
•Absolute relocation, which is used to relocate addresses for data in the .data section, and for labels in certain call instructions, actually uses a 64 bit address. (R_X86_64_32 uses 32 bits to hold addresses low enough to fit in 32 bits)

The assembler homework as an example

- The code is shown twice on the next 2 slides
  - Absolute addresses are marked on one slide
  - Relative addressing is marked on the other
  - The objdump output shows how the addresses are actually coded in hex
- What the code does:
  - Prints a prompt
  - Uses scanf to read a value and put it into **x,** which is a quad in the .data section
  - Adds 5 to x
  - Prints x

Absolute Address Example (ahw)

```
000000000040055d <main>:
  40055d:      55                        push    %rbp
  40055e:      48 89 e5                  mov     %rsp,%rbp
  400561:      48 c7 c7 50 06 40 00      mov     $0x400650,%rdi
  400568:      48 c7 c0 00 00 00 00      mov     $0x0,%rax
  40056f:      e8 bc fe ff ff            callq   400430 <printf@plt>
  400574:      48 c7 c6 34 10 60 00      mov     $0x601034,%rsi
  40057b:      48 c7 c7 8e 06 40 00      mov     $0x40068e,%rdi
  400582:      48 c7 c0 00 00 00 00      mov     $0x0,%rax
  400589:      e8 c2 fe ff ff            callq   400450 <scanf@plt>

000000000040058e <here>:
  40058e:      48 83 04 25 34 10 60      addq    $0x5,0x601034
  400595:      00 05
  400597:      48 8b 34 25 34 10 60      mov     0x601034,%rsi
  40059e:      00
  40059f:      48 c7 c7 91 06 40 00      mov     $0x400691,%rdi
  4005a6:      48 c7 c0 00 00 00 00      mov     $0x0,%rax
  4005ad:      e8 7e fe ff ff            callq   400430 <printf@plt>
  4005b2:      48 c7 c0 00 00 00 00      mov     $0x0,%rax
  4005b9:      c9                        leaveq
  4005ba:      c3                        retq
```

The format strings are in the .rodata section at **0x400650, 0x40068e, and 0x400691**.
The absolute addresses are coded as a 4 byte address rather than an 8 byte address.

The quad **x** is in the .data section at 0x601034.
The absolute address of x is coded as a 4 byte address rather than an 8 byte address.

Relative Address Example (ahw)

```
000000000040055d <main>:
  40055d:       55                      push   %rbp
  40055e:       48 89 e5                mov    %rsp,%rbp
  400561:       48 c7 c7 50 06 40 00    mov    $0x400650,%rdi
  400568:       48 c7 c0 00 00 00 00    mov    $0x0,%rax
  40056f:       e8 bc fe ff ff          callq  400430 <printf@plt>
  400574:       48 c7 c6 34 10 60 00    mov    $0x601034,%rsi
  40057b:       48 c7 c7 8e 06 40 00    mov    $0x40068e,%rdi
  400582:       48 c7 c0 00 00 00 00    mov    $0x0,%rax
  400589:       e8 c2 fe ff ff          callq  400450 <scanf@plt>

000000000040058e <here>:
  40058e:       48 83 04 25 34 10 60    addq   $0x5,0x601034
  400595:       00 05
  400597:       48 8b 34 25 34 10 60    mov    0x601034,%rsi
  40059e:       00
  40059f:       48 c7 c7 91 06 40 00    mov    $0x400691,%rdi
  4005a6:       48 c7 c0 00 00 00 00    mov    $0x0,%rax
  4005ad:       e8 7e fe ff ff          callq  400430 <printf@plt>
  4005b2:       48 c7 c0 00 00 00 00    mov    $0x0,%rax
  4005b9:       c9                      leaveq
  4005ba:       c3                      retq
```

The function calls use 32 bit signed relative offsets instead of 8 byte addresses.
`0xfffffebc` is –324 decimal
`0xfffffec2` is –318 decimal
`0xfffffe7e` is –386 decimal

Can check the math:
400430 – 400574=fffebc or –324 decimal

Printf and scanf are loaded above this code at lower numbered addresses – note that here address grow as we go down the page

(Stored numbers are little endian)

How did those addresses get there?

- The addresses in the executable were put there by the linker
- The linker used the relocation entries that were placed in the .o file by the assembler
- In the relocation table that follows we find 9 entries:
  - 3 **`R_X86_64_PC32`** entries for the relative addressing used in the function calls
  - 6 **`R_X86_64_PC32`** entries that for absolute addressing
    - 3 are references to the variable **`x`** in the .data section
    - 3 are references to the format strings in the .rodata section

Relocation table in ahw.o (via the readelf tool)

```
Relocation section '.rela.text' at offset 0x240 contains 9 entries:
  Offset          Info           Type           Sym. Value    Sym. Name +
Addend
000000000007  00050000000b R_X86_64_32S      0000000000000000 .rodata + 0
000000000013  000900000002 R_X86_64_PC32     0000000000000000 printf - 4
00000000001a  00030000000b R_X86_64_32S      0000000000000000 .data + 0
000000000021  00050000000b R_X86_64_32S      0000000000000000 .rodata + 3e
00000000002d  000a00000002 R_X86_64_PC32     0000000000000000 scanf - 4
000000000035  00030000000b R_X86_64_32S      0000000000000000 .data + 0
00000000003e  00030000000b R_X86_64_32S      0000000000000000 .data + 0
000000000045  00050000000b R_X86_64_32S      0000000000000000 .rodata + 41
000000000051  000900000002 R_X86_64_PC32     0000000000000000 printf - 4
```

This code references the .rodata and .data sections and it calls 2 library functions. There are absolute and relative relocation entries, but all are 32 bit.

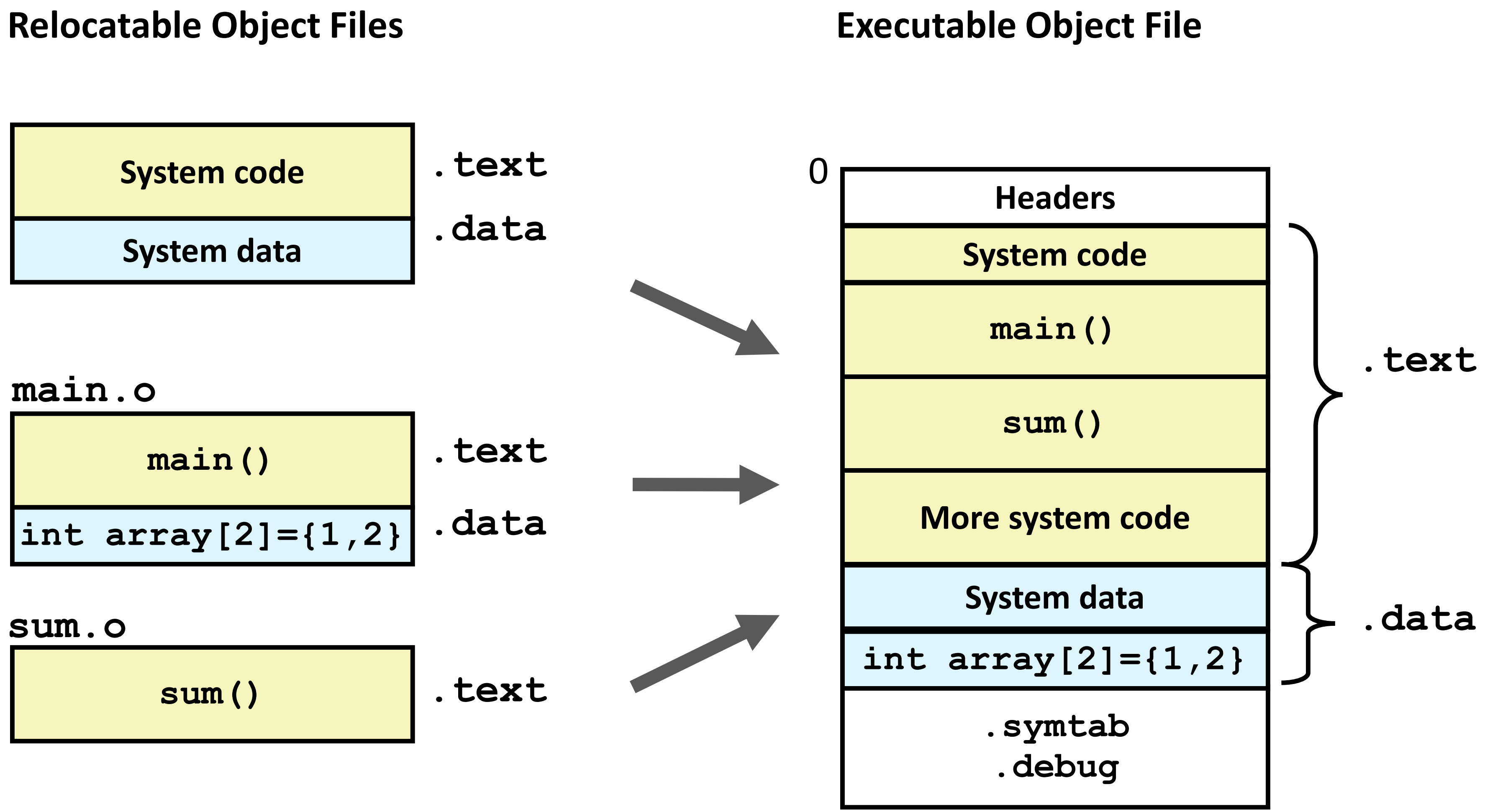
Sidebar: How many address bits do we need?

- 32 bits handles many addresses:
  - The heap starts around **`0x 0060 0000`** (spaces added for clarity)
  - Code lives between **`0x 0040 0000`** and the heap
- We need more bits for stack-based addresses:
  - A typical user stack address is **`0x 7fff ffff e300`**
  - Linux user processes have a 48 bit virtual address space
  - Current processors support 48 bit physical addresses
- Having 64 bit addresses that fit in 32 bits saves many bytes in the instruction stream

Static linking – What do linkers do?

- Step 2. Relocation
  - -Merges separate code and data sections into single sections
    - -Take the code section from each of the relocatable object files, main.o and swap.o, and merge them into a single code section.
    - -Take the .rodata sections from each of the relocatable object files, and merge them into a single .rodata section.
    - -Take the .data sections from each of the relocatable object files, and merge them into a single .data section.
    - -Take the .bss (unitialized file scope variables) sections from individual relocatable object files, and merge them into a single .bss section
  - -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 (i.e., any encoded instructions which have the addresses of these symbols) to reflect their new positions.

Relocation


Relocatable Object Files
System code
System data
.text
.data
main.o
main()
int array[2]={1,2}
.text
.data
sum.o
sum()
.text
Executable Object File
0
Headers
System code
main()
sum()
More system code
System data
int array[2]={1,2}
.symtab
.debug
.text
.data

Packaging Commonly Used Functions

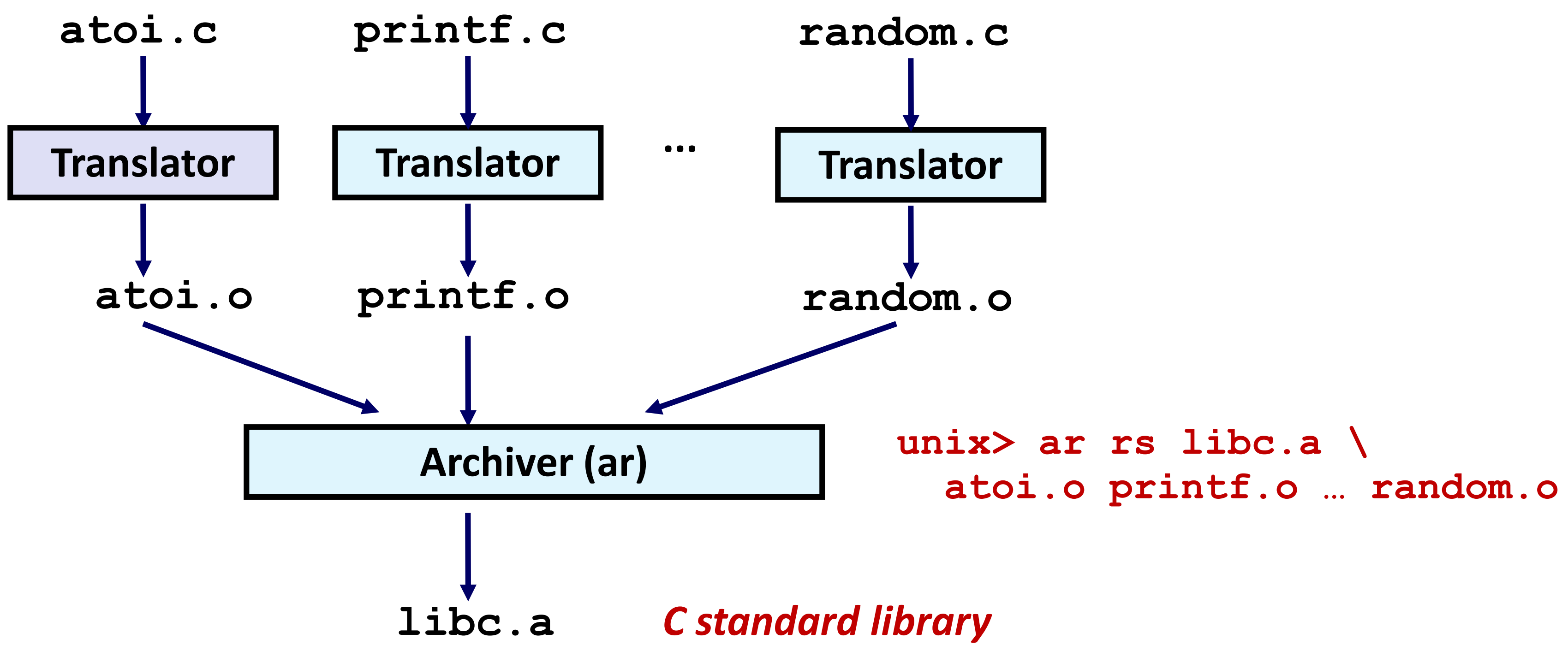
- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- 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.

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

libvector.a

```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

```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

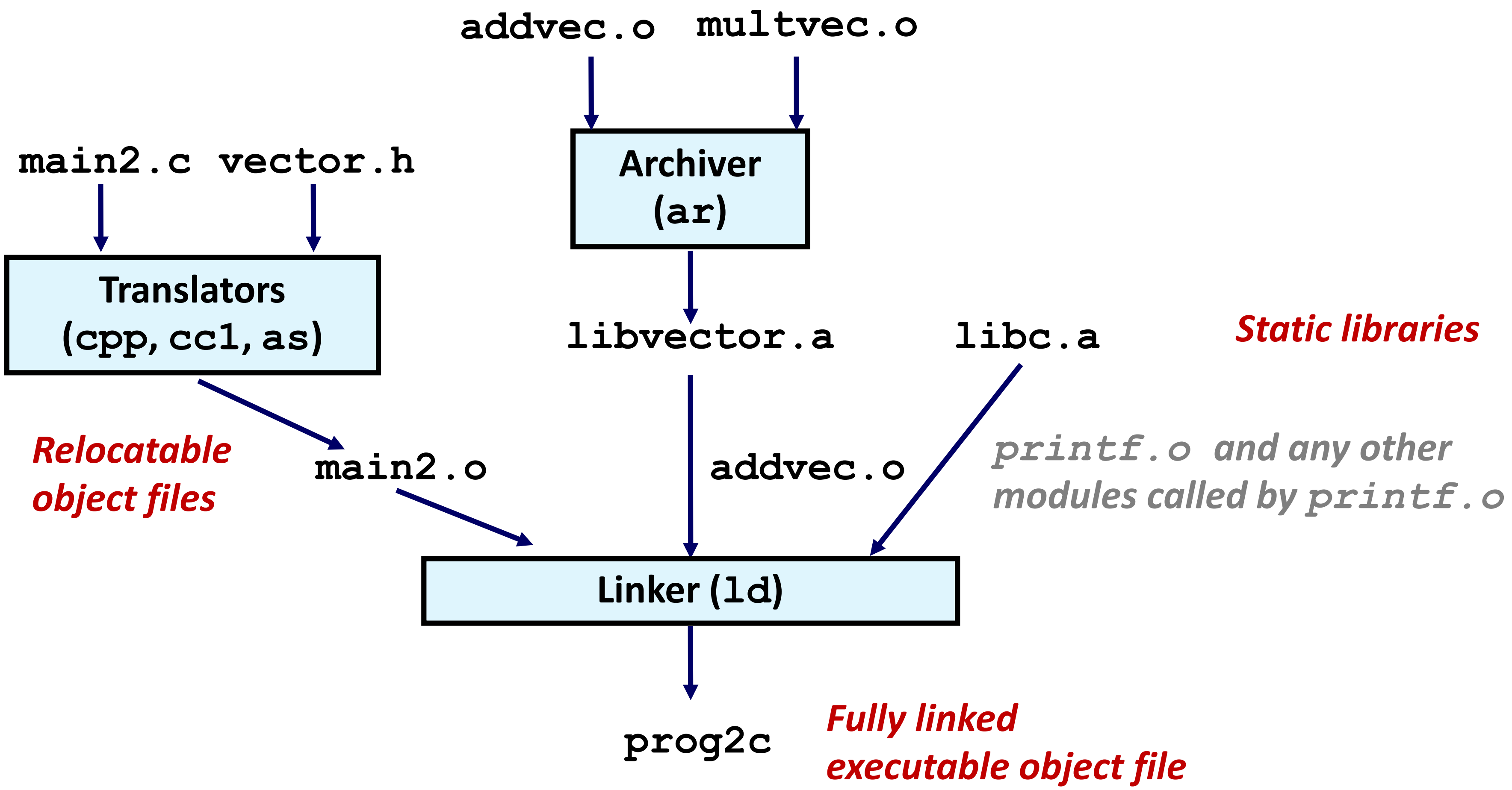
```c
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

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

multvec.c

Contents of a more familiar static library:

```
[kirby.249@cse-sl1 list]$ ar -t liblinkedlist.a
iterate.o
insert.o
deleteSome.o
any.o
```

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

- 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

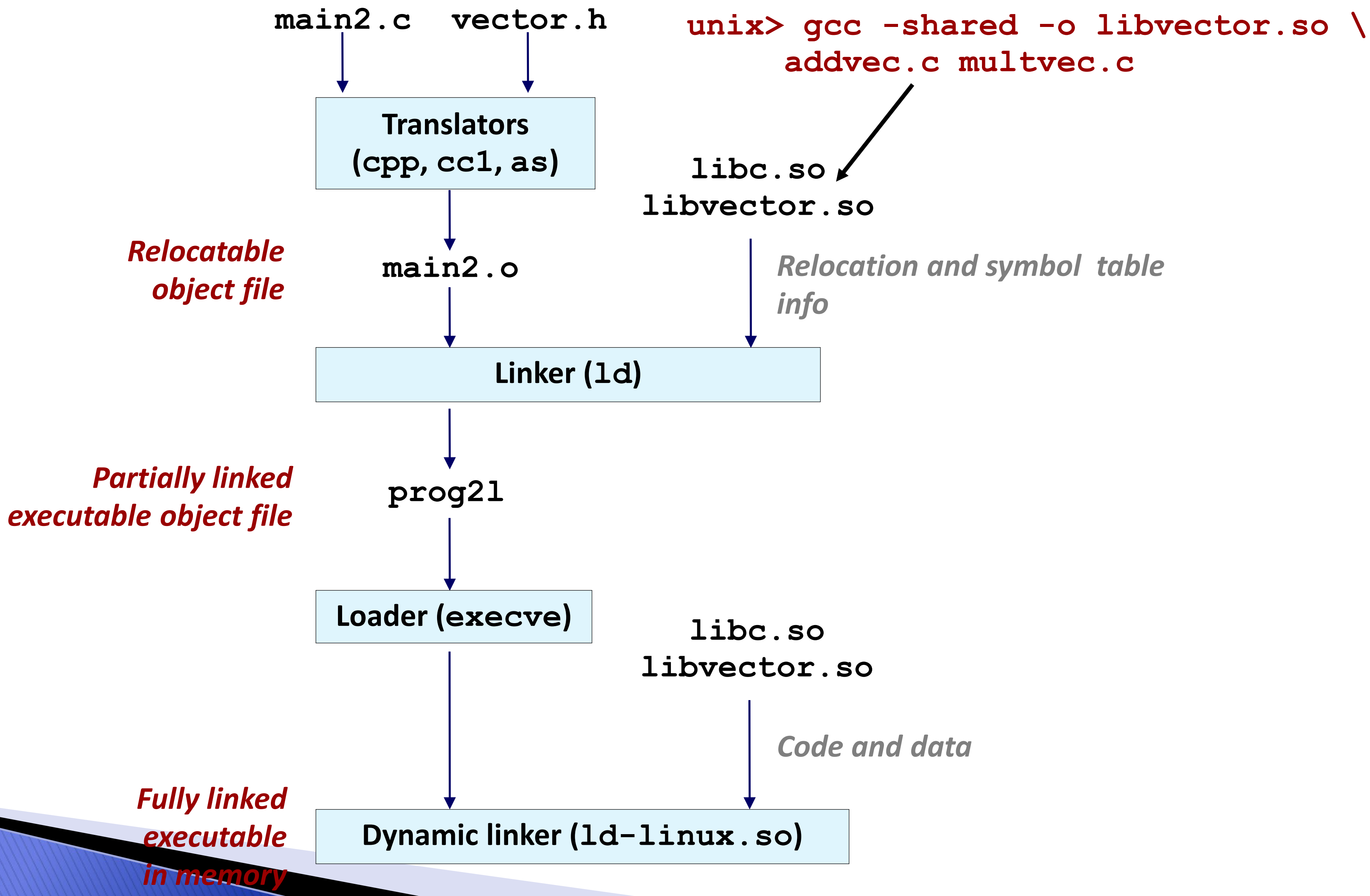
Shared Libraries (cont.)

- 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 you learn about virtual memory in Systems II

Dynamic Linking at Load-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.