# CS201 – Lecture 3
# Linking

RAOUL RIVAS

PORTLAND STATE UNIVERSITY

# Announcements

# Multiple Source files

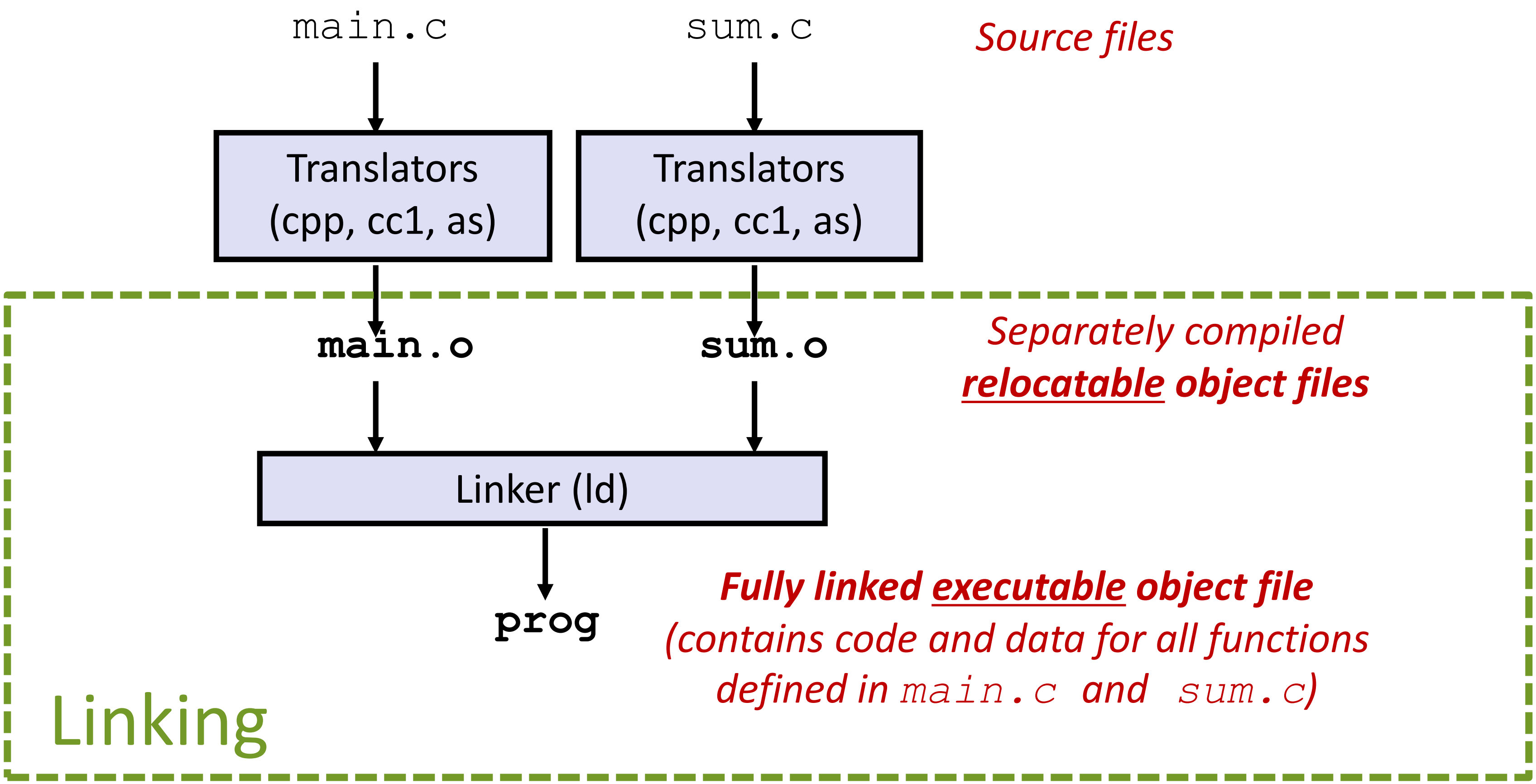
***main.c***

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

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

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

***sum.c***

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

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

# Why Linkers?


main.c
sum.c
Source files
Translators
(cpp, cc1, as)
Translators
(cpp, cc1, as)
main.o
sum.o
Separately compiled
relocatable object files
Linker (ld)
prog
Fully linked executable object file
(contains code and data for all functions
defined in main.c and sum.c)
Linking

# Why Linkers?

- Reason 1: Modularity
  - Applications can be very large
    - Linux: 15 million lines of code
    - Windows: 50 million lines of code
    - Office: 40 million lines of code
  - 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?

- Reason 2: Efficiency

  - Time: Separate compilation
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.

  - 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.

# Object Files

- 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.
  - This are *.EXE and *.COM files in Windows
  - Non Relocatable!
- 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

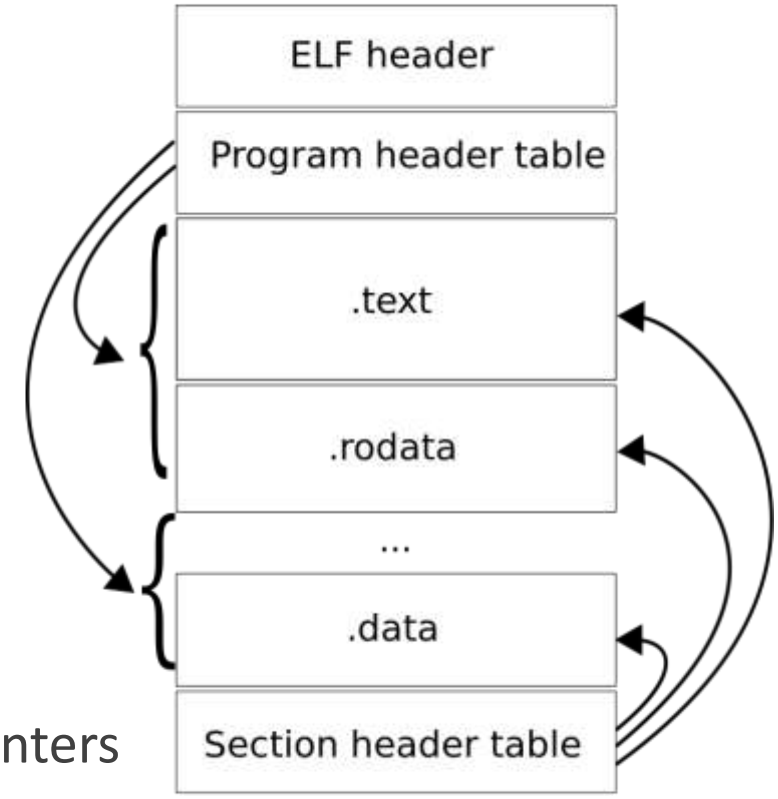
# Executable and Linkable Format (ELF)

- Standard binary format for object files in Unix and Linux

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

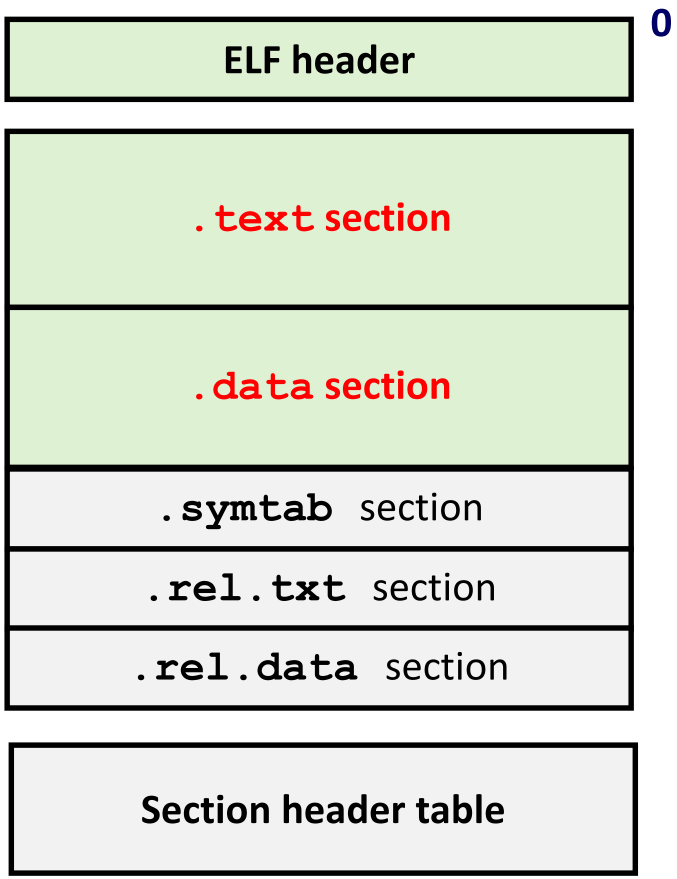
- Generic name: ELF binaries

- Modular design
  - Program header table contains pointers to each section



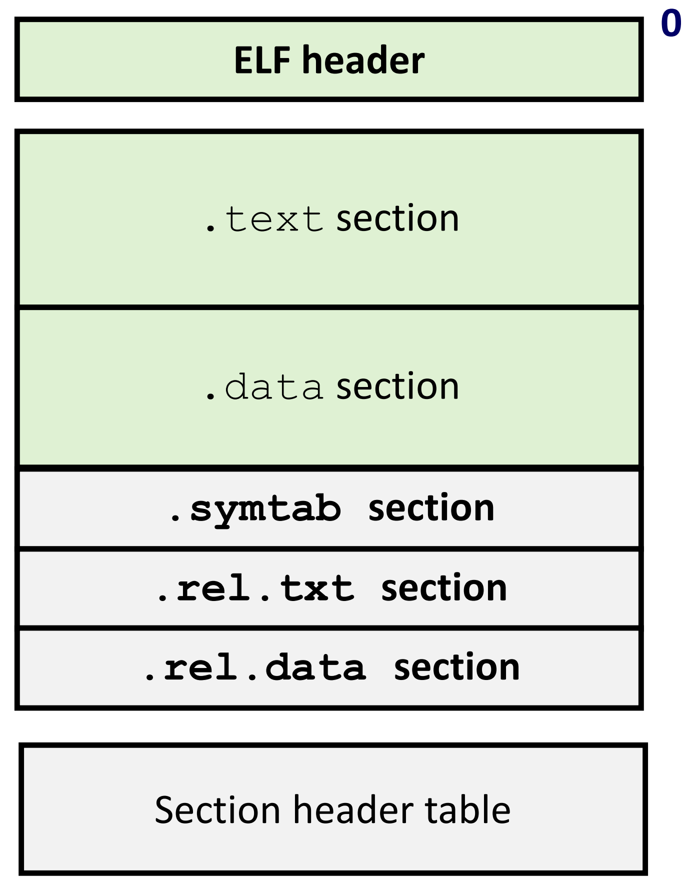
# ELF Structure

- Elf header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **`.text` section**
  - Code
- **`.data` section**
  - Initialized global variables
- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- Section header table
  - Offsets and sizes of each section for Relocatable Object Files



# ELF Structure – Relocation Structures

- Relocation data is stored in the
- `.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



# Step 1: Symbol Resolution

- 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.**

# 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

Reference to a Local Symbol

Local Symbol

*main.c*

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

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

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

*sum.c*

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

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

Definition of a Global Symbol

Linker knows nothing of `val`

Reference to an External Symbol

Definition of a Global symbol

Linker knows nothing of `i` or `s`

# Local Symbols

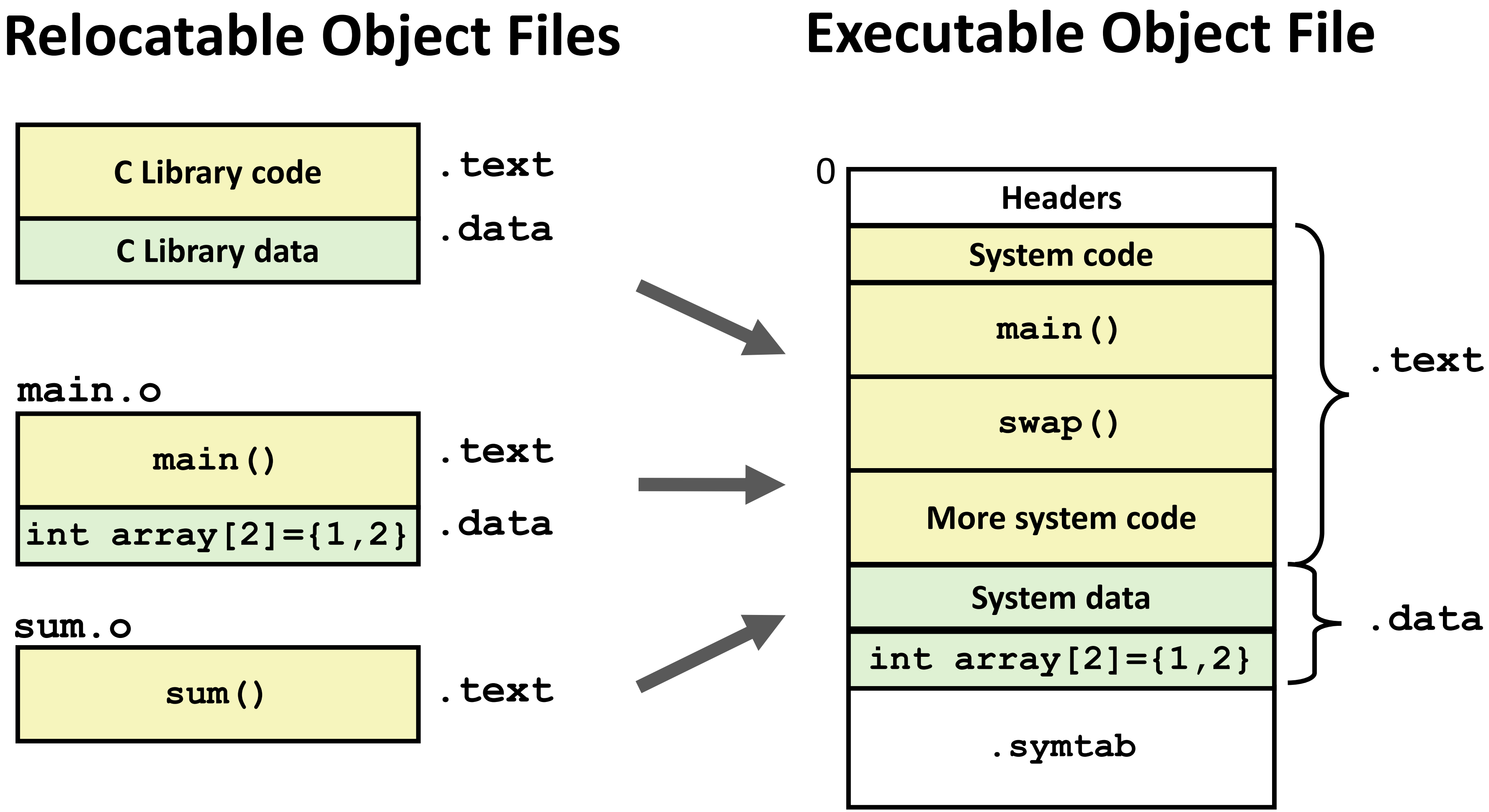
- Local non-static C variables vs. local static C variables
  - local non-static C variables: stored on the stack
  - local static C variables: stored in `.data`

```
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`

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

# Step 2: Relocation



# Relocation Entries

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

int main()
{
    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
                        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
```

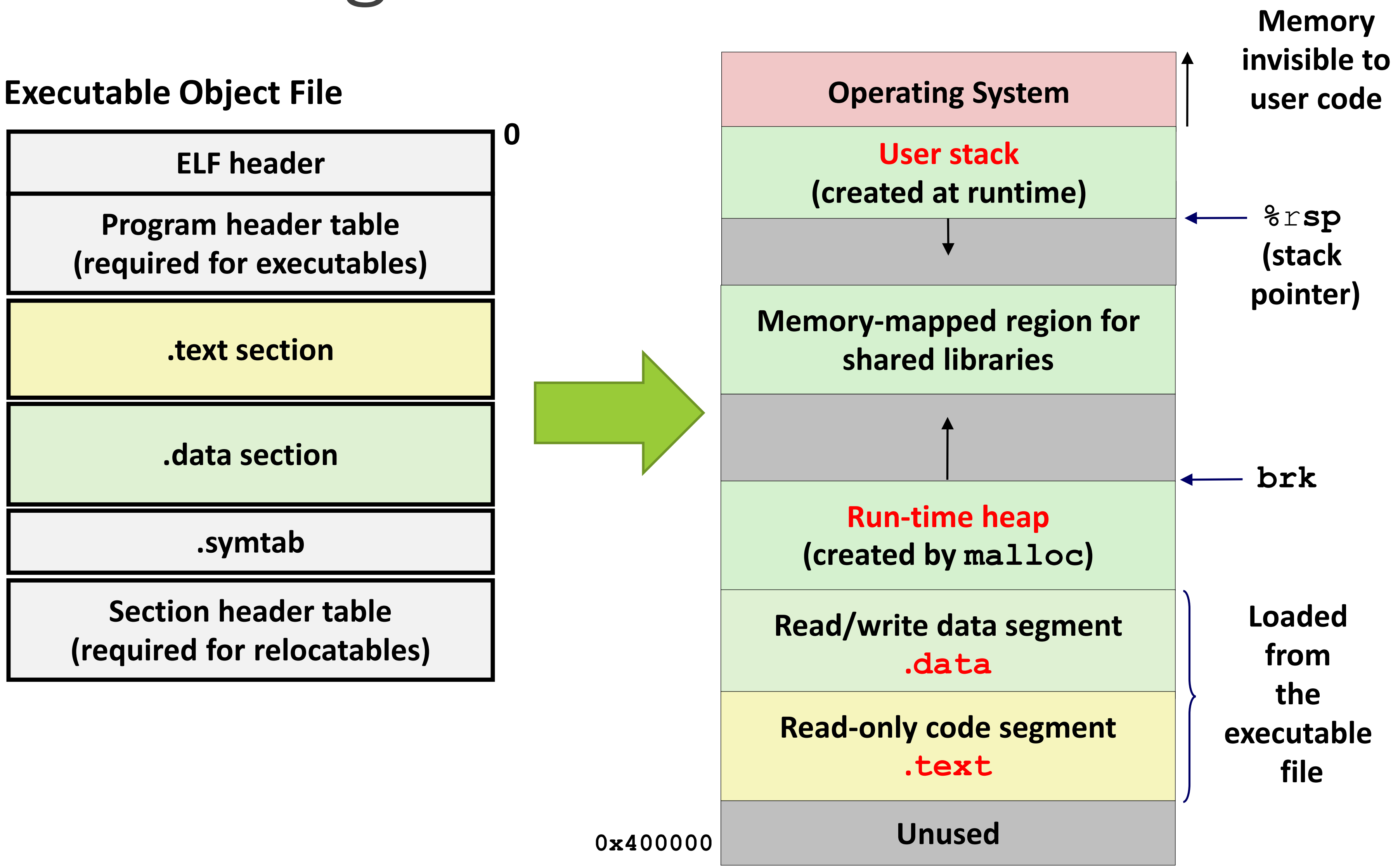
main.o

# 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

# Loading an Executable

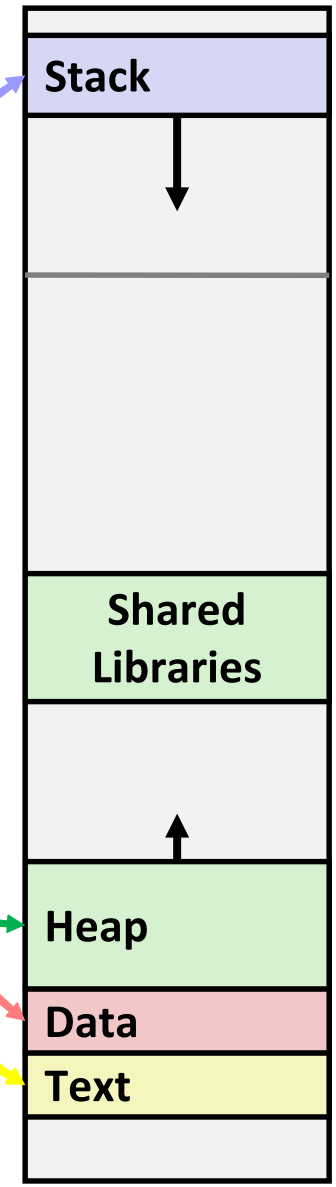


# Loading an Executable

- User Stack
  - Allocate local variables
  - Store the state of the registers of the caller including the return address of the caller
- Run-Time Heap
  - Used to allocate dynamic memory (malloc)
- Read Write Segment
  - Store statically allocated globals (.data segment)
- Read-Only Segment
  - Store program code (.text segment)

# Memory Allocation

```
char big_array[1L<<24];  /* 16 MB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1,*p4;
    int local = 0;

    p1 = malloc(1L << 28); /* 256 MB */
    p4 = malloc(1L << 8);  /* 256  B */

 /* Some print statements ... */
}
```

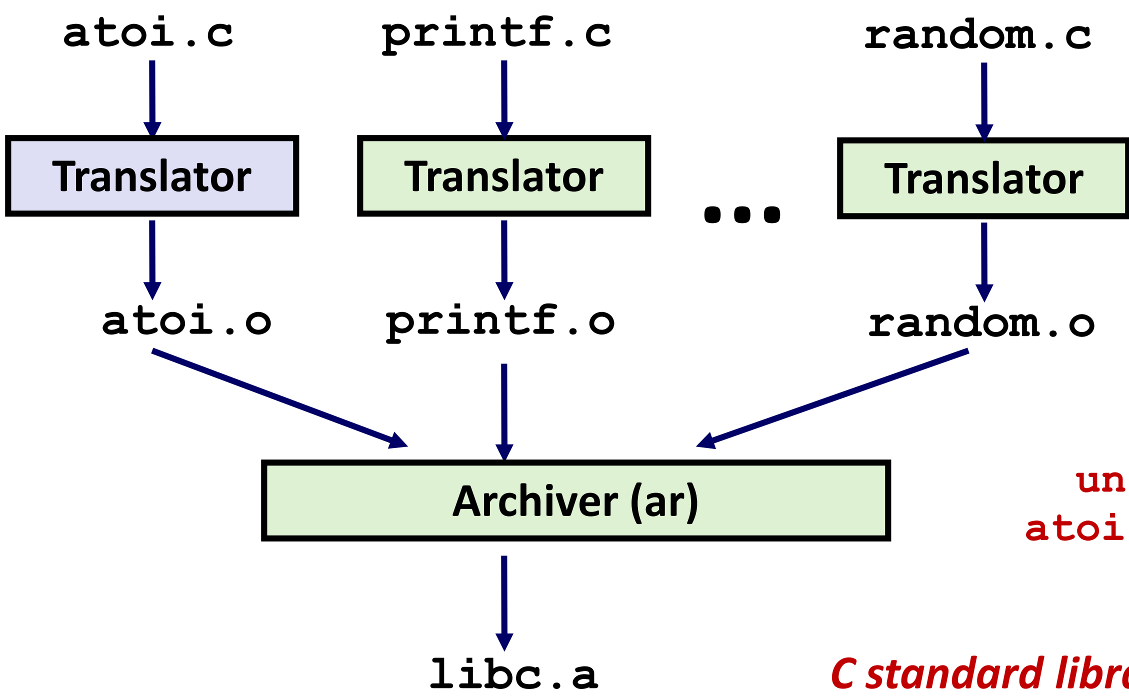



***Where does everything go?***

# 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.

# Static Libraries




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

*C standard library*

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

# The C library

## `libc.a` (the C standard library)

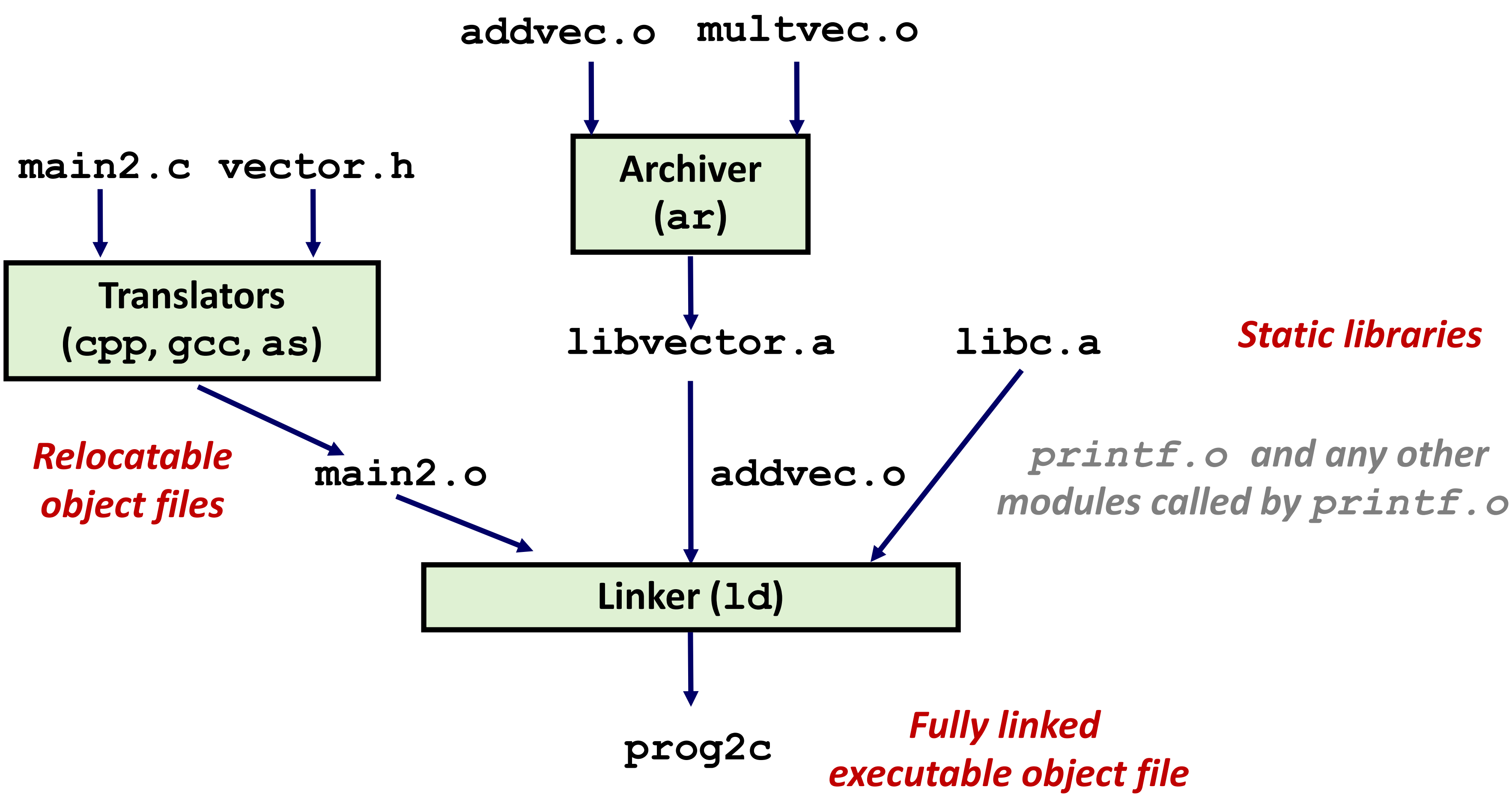
- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

## `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
```

# Linking Static Libraries



# Shared Libraries

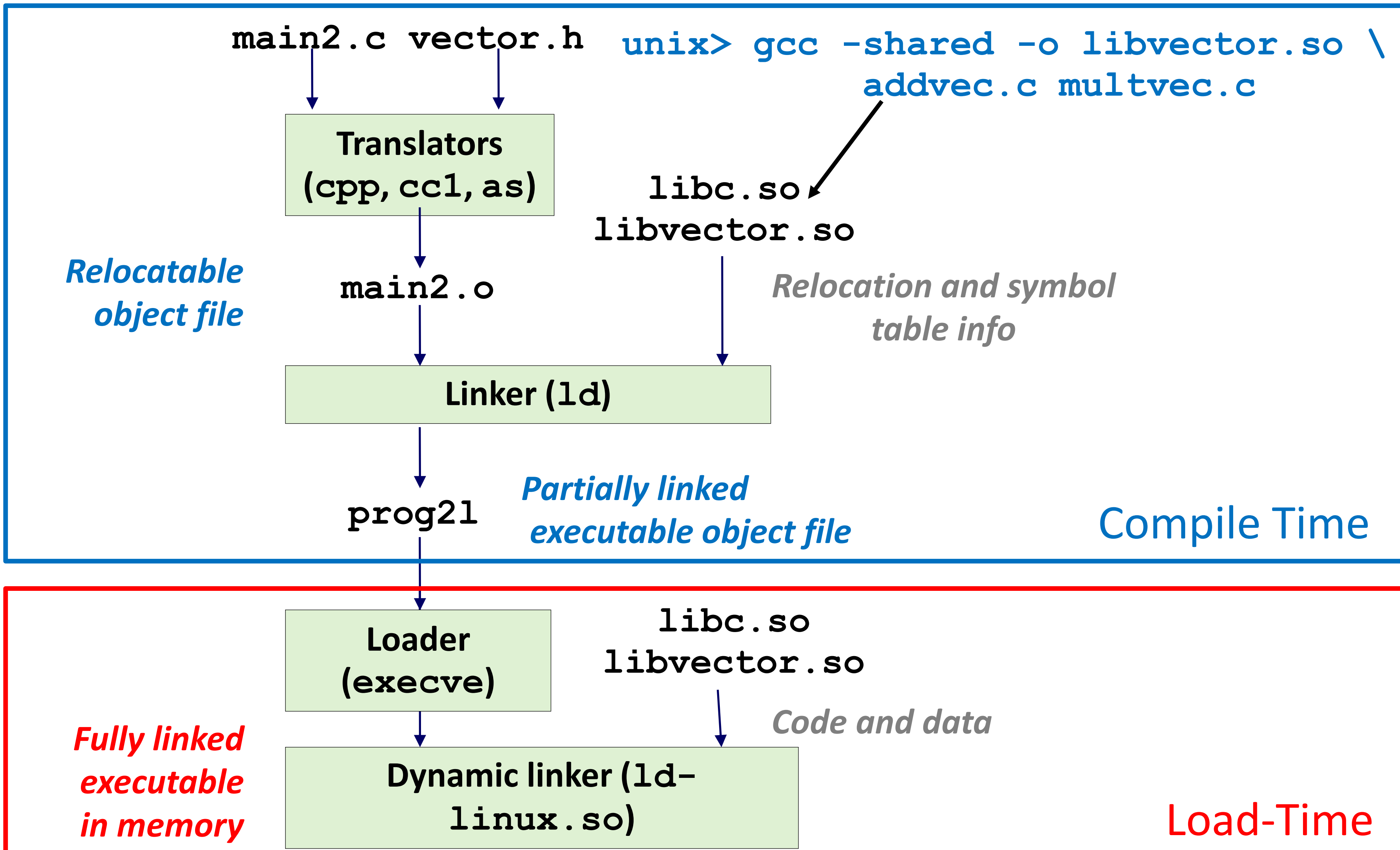
- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- 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

# Dynamic Linking

- 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.
    - Add-ins and Plug-ins
    - High-performance web servers.
    - Runtime library inter-positioning.

# Dynamic Linking at Load-time

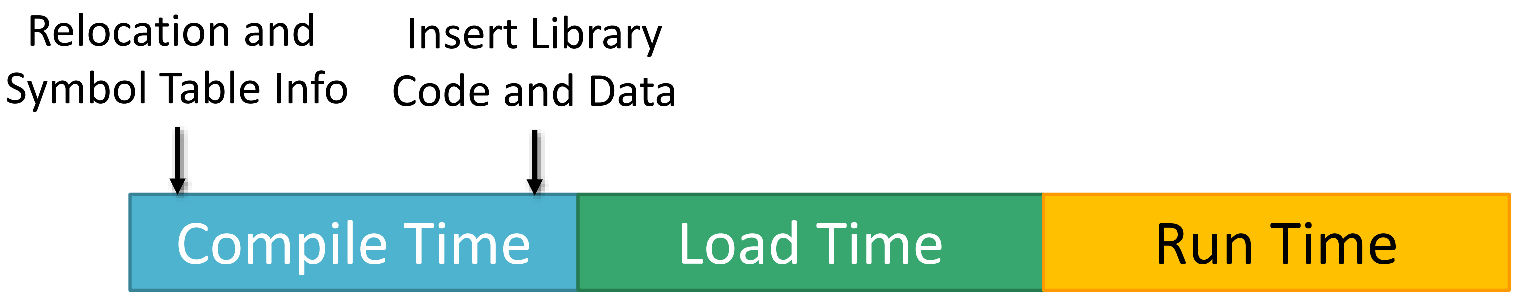


# Dynamic Linking at Run-time

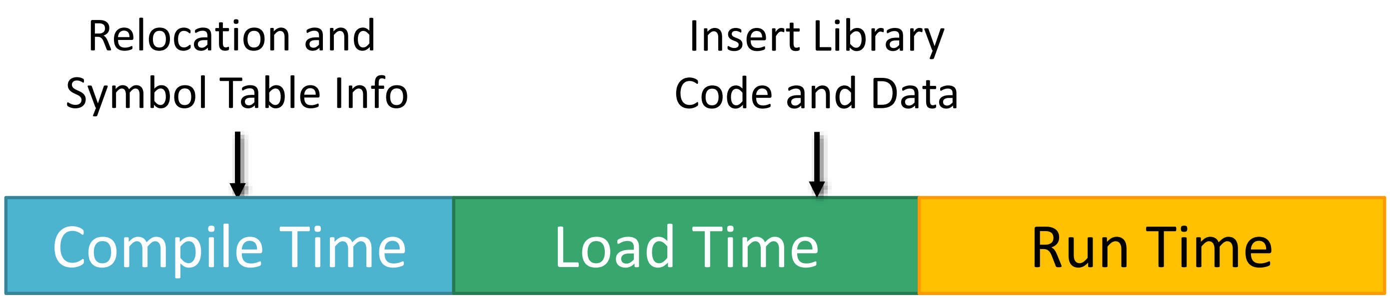
```
int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);

/* Dynamically load the shared library that contains
addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
/* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
/* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
/* Unload the shared library */
    dlclose(handle);
    return 0;
}
```

# Library Linking Timeline

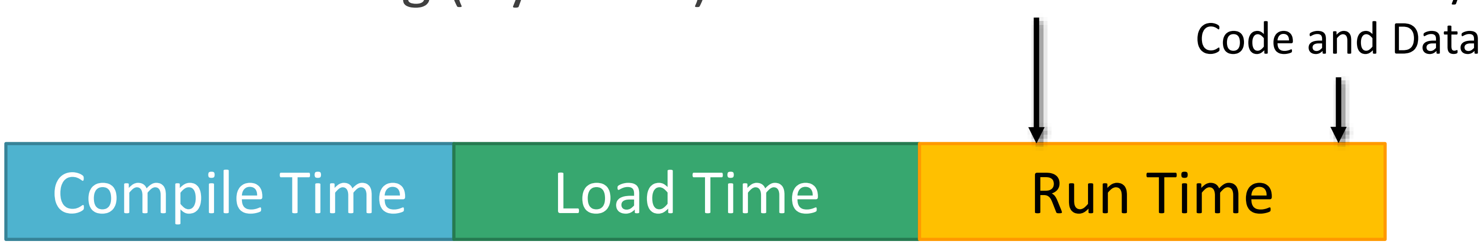
- Compile-Time Linking (Static)




- Load-Time Linking (Dynamic)




- Run-Time Linking (Dynamic)



# 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.

- Read http://www.ibm.com/developerworks/library/l-dynamic-libraries/ to learn more about Dynamic Linking in Linux