ECE20019 Open Source Software, Spring 2018

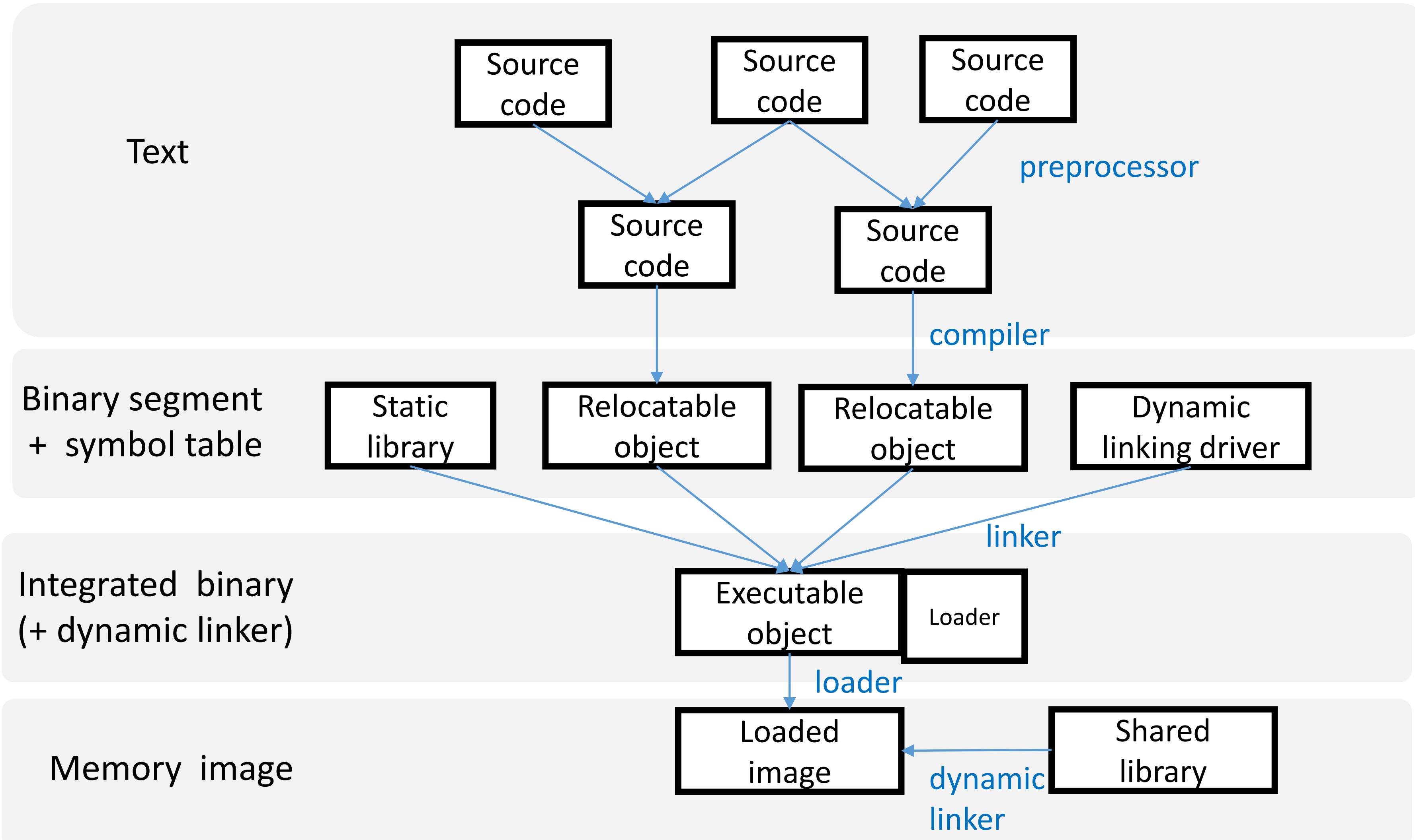
Build: Preprocessing, Compiling & Linking

Overview

- A build is a process of translating a high-level program into a machine-readable and executable form
  - operation into instructions
  - symbols into memory addresses
    - variables, functions

- Modern computer systems have different mechanisms for binding symbols with memory addresses
  - statically (code) vs. dynamically (execution)
  - trade-off between runtime performance and portabiilty

Building Flow



Binding Time

- Coding

```
printf(“Hello, Application”) ;
```

- Preprocessing

```
#ifdef ARCH x86
    printf(“Hello, x86”) ;
#endif
```

- Linking

```
extern char * libversion ;
printf(“Hello, library of version %s”, libversion) ;
```

- Runtime (execution time)

```
scanf(″%s”, name) ;
printf(“Hello, user %s”, name) ;
```

C preprocessing overview

- A text processor employed by a compiler framework to manipulate source code text before compilation
  - inclusion of header files
  - macro expansion
  - conditional compilation
  - removal of unnecessary text
  - encoding conversion

- Preprocessing directives command most preprocessing tasks for source code files
  - in a form of '#*name*'
  - some directives receive arguments
  - a directive cannot cover more than one line except the one continued with backslash-newlines

Marco expansion (basic)

- A macro is a code fragment which has been given a name.
  Whenever the name is used, it is replaced by the content of the macro.
- `#define` *x y*
  - the preprocessor replaces all *x* token instances to *y*
  - *x* must not begin with '_'
- `#define` *x*
  - to notify a certain control message to the preprocessor
    (i.e., controlling macro or guard macro)
- A preprocessor can be invoked with macro definitions
  - e.g. `gcc -D name=value` … starts a preprocessing with *name* being *value*
- `#undef` *x*
  - Cancel the x macro definition
  - Macro definitions are interpreted in the order that they appear in a text

Header file inclusion

- A header file is a file containing C declarations, structure definitions and macro definitions to be shared between several source files
  - `.h` files
- `#include` copies the whole texts of the given file name to the current source file
  - A header file shall not include another header file in order to avoid the possibility of including the same file twice
- `#include` <*name*> (standard system directory first)
  `#include` "*name*" (current dir first)
- A computed include enables including one of many header files depending on a controlling condition
  - E.g.,

```
#if SYSTEM_1
#  include "system_1.h"
#elif SYSTEM_2
#  include "system_2.h"
          …
#endif
```

This slide is taken from a class material of 15-213 Intro. to Computer System conducted in CMU at Fall 2015
http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

Linking

15-213: Introduction to Computer Systems
13th Lecture, Oct. 13, 2015

Instructors:

Randal E. Bryant and David R. O'Hallaron

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

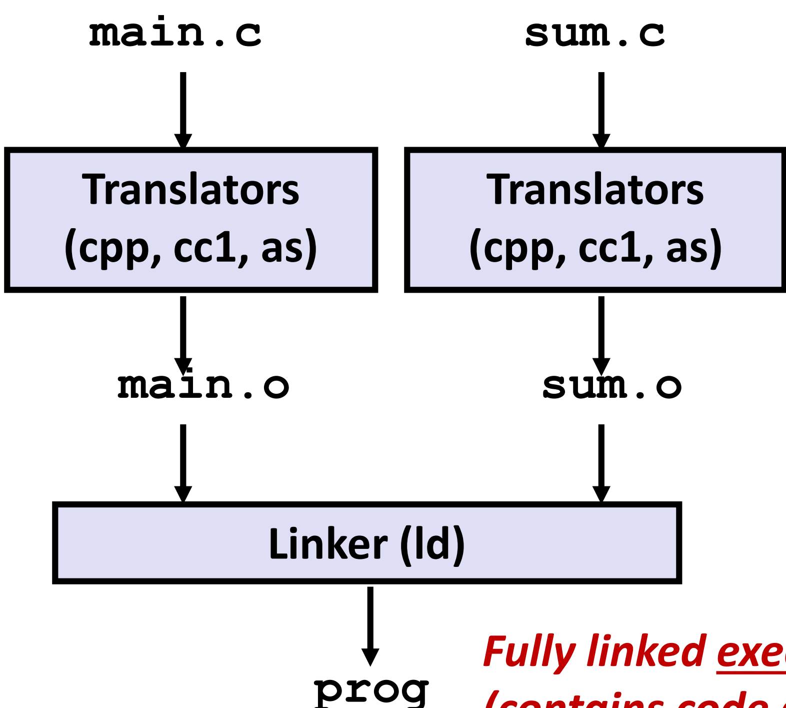
```c
int sum(int *a, int n)
{
    int i, s = 0;

    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`



Source files

Separately compiled <u>relocatable</u> object files

Fully linked <u>executable</u> object file (contains code and data for all functions defined in `main.c` and `sum.c`)

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.
  - 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
- **Static library (`.a` file)**
  - Collection of relocatable objects of a certain module
- **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)

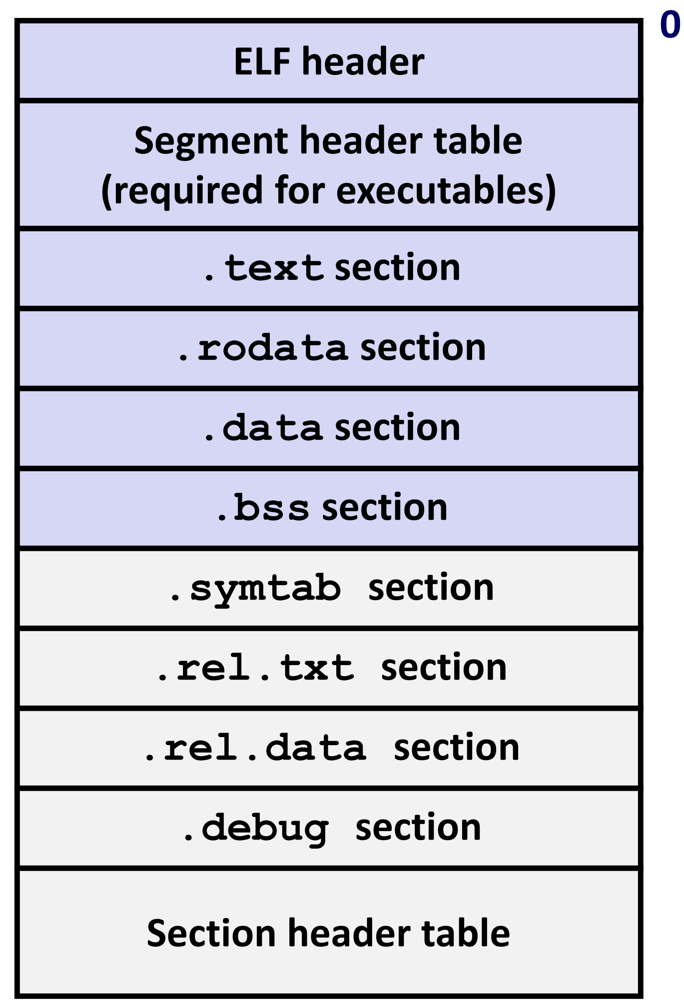
- **Standard binary format for object files**

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

- **Generic name: ELF binaries**

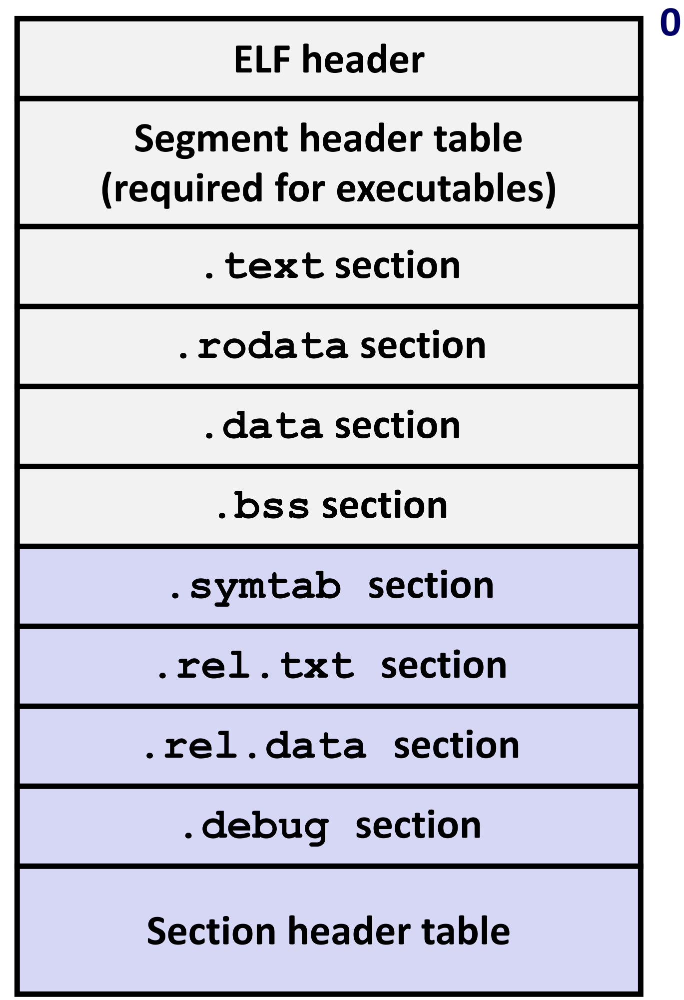
ELF Object File Format

- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual 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

```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`

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;
}
```

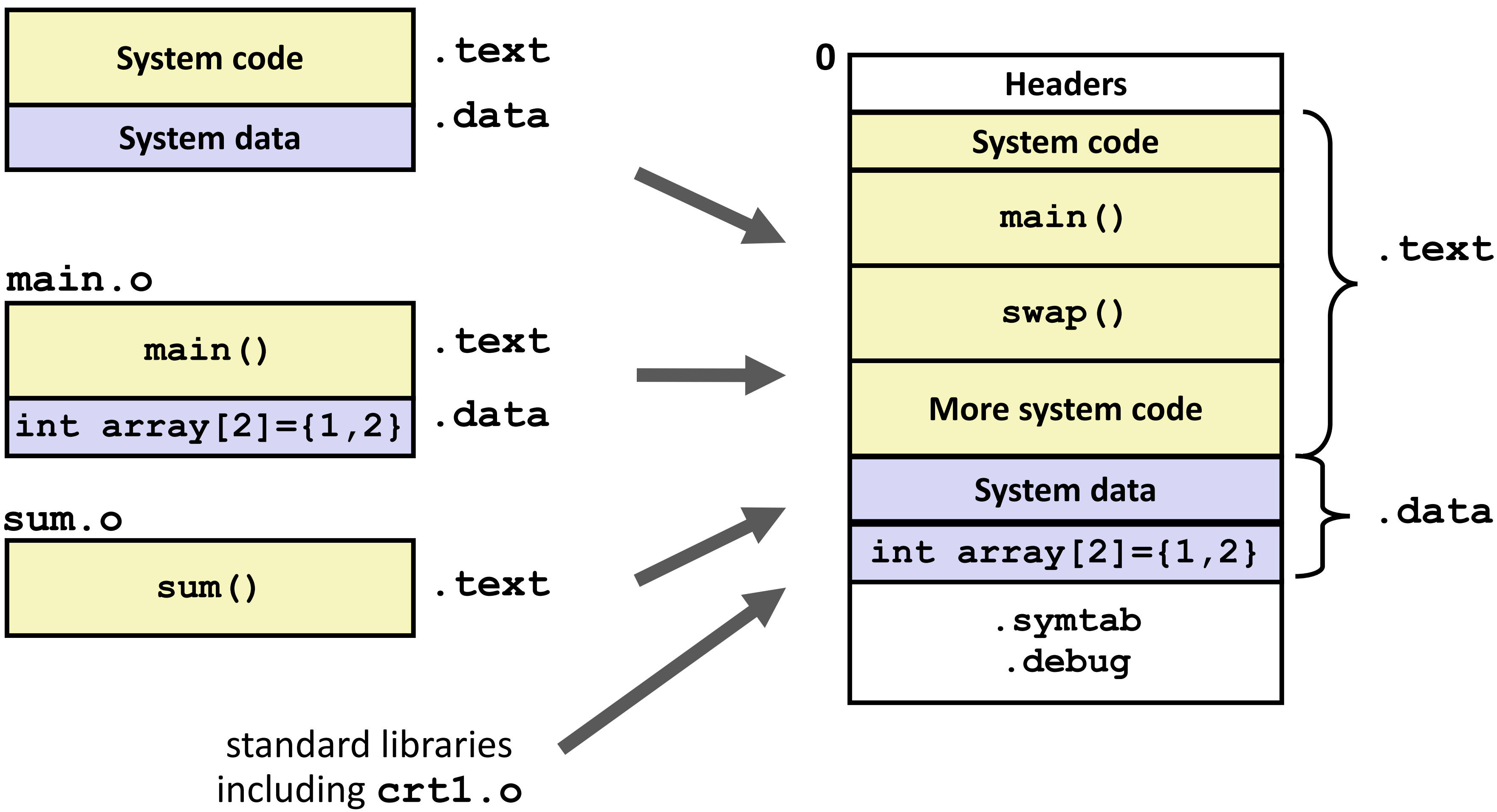
Compiler allocates space in `.data` for each definition of `x`

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

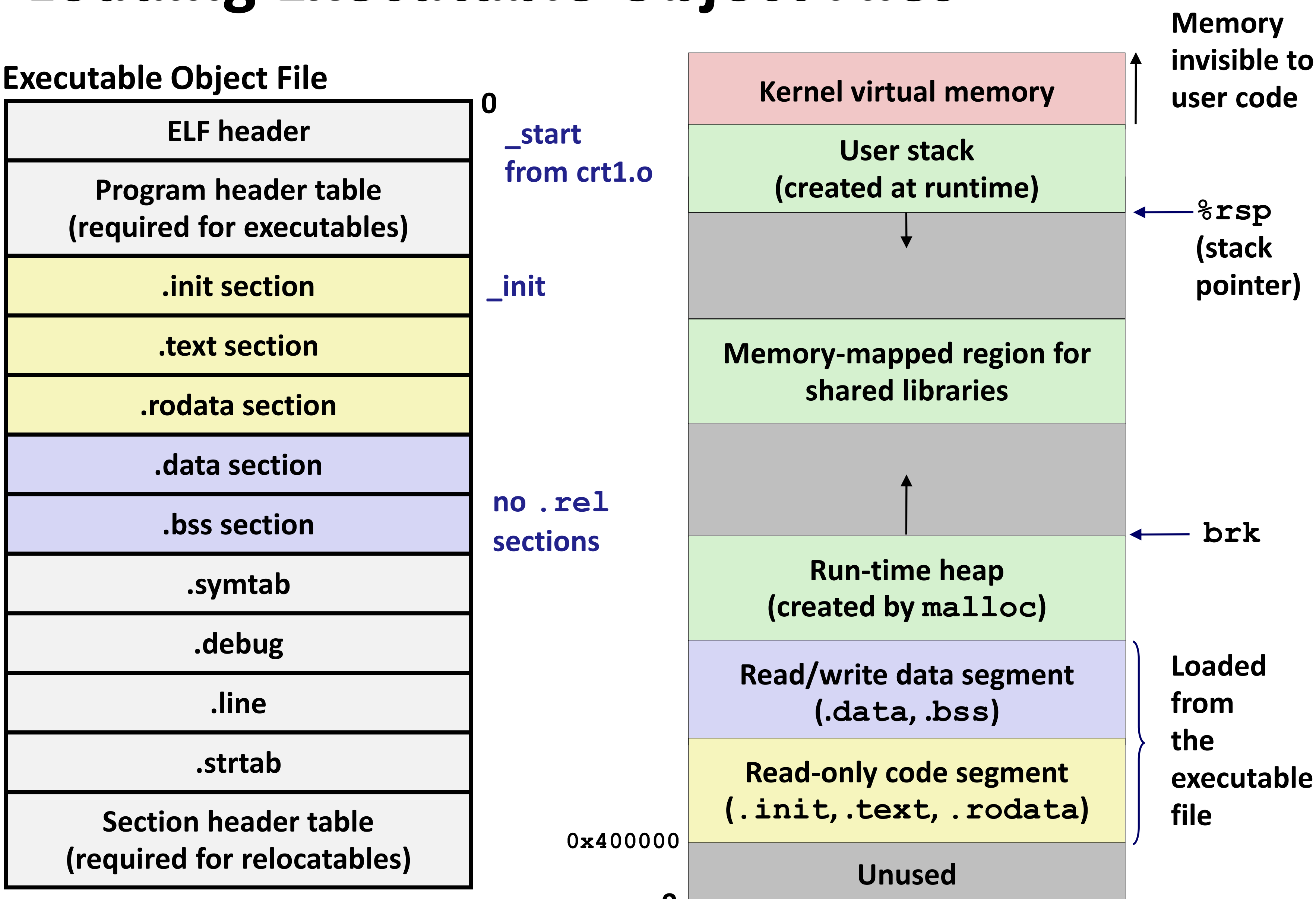
Step 2: Relocation

Relocatable Object Files

Executable Object File



Loading Executable Object Files



Packaging Commonly Used Functions

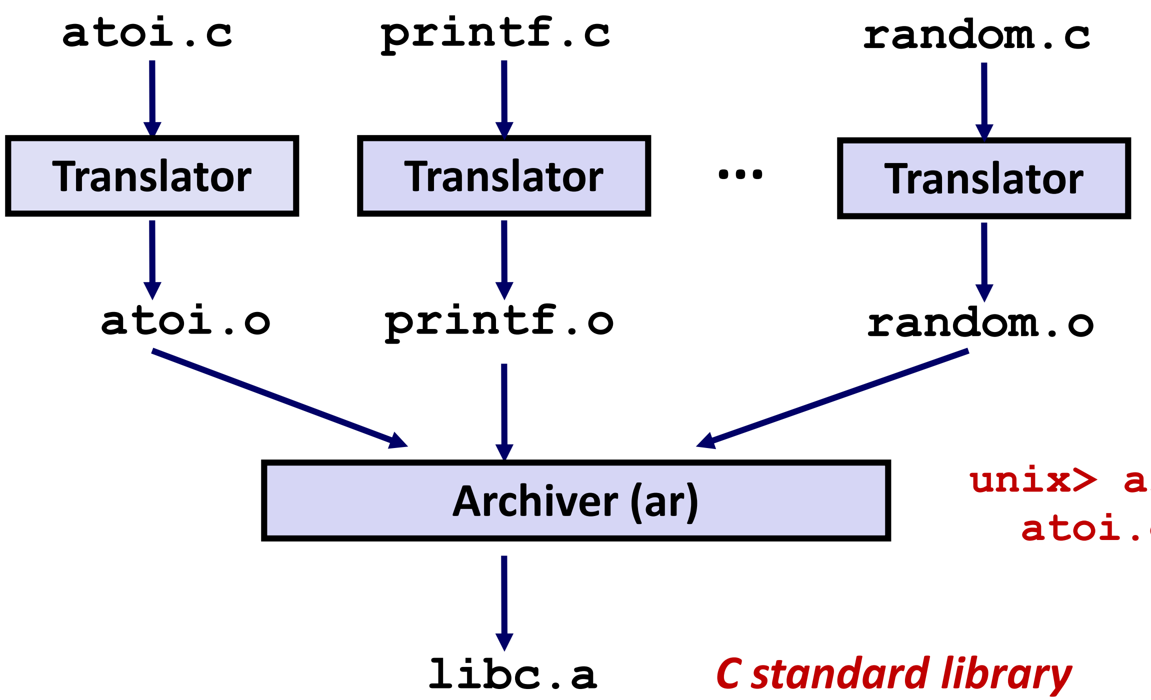
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **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




```
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.**

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];
}
```

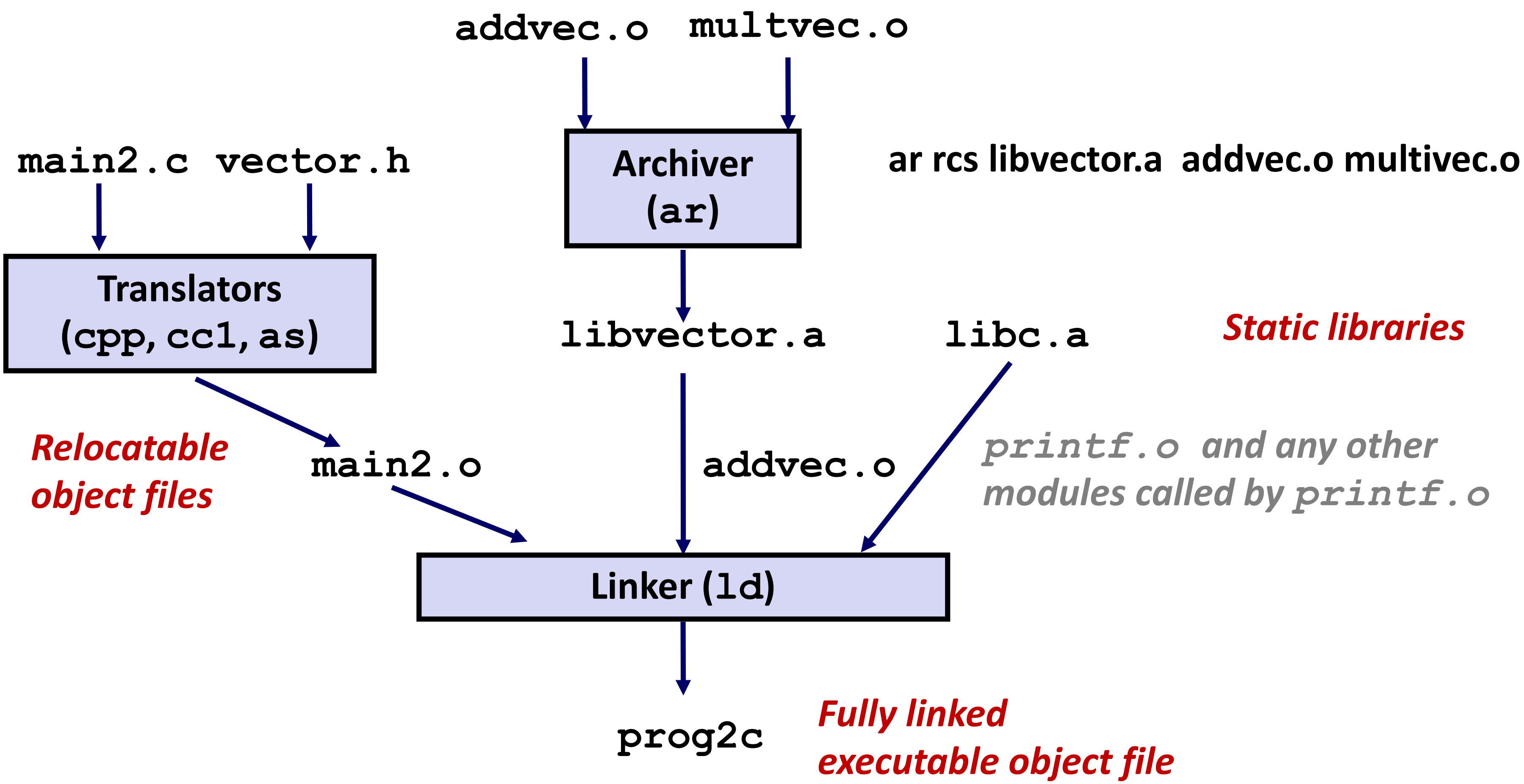
addvec.c

```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

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

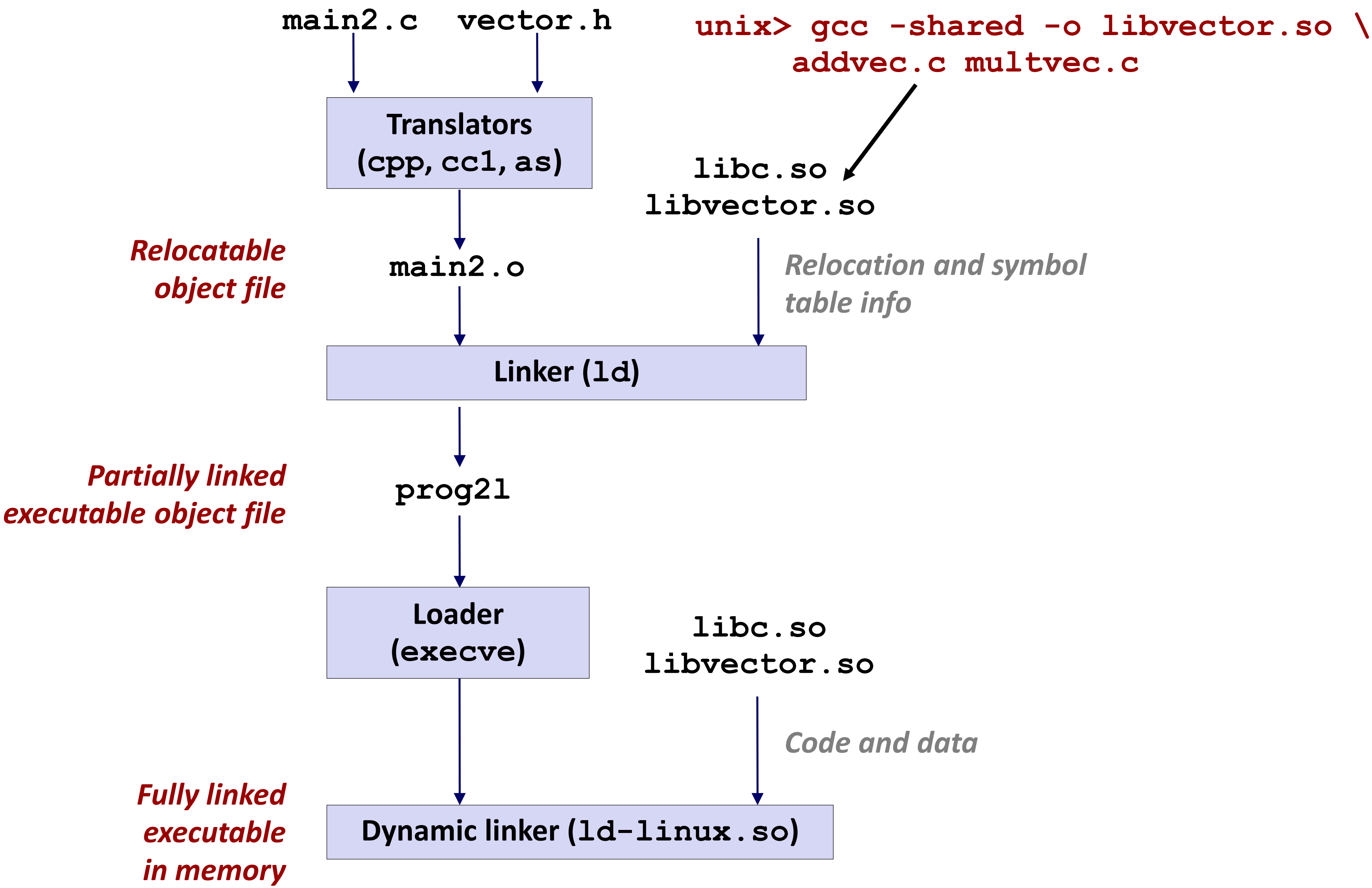
- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- **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 we learn about virtual memory

Dynamic Linking at Load-time



Dynamic Linking at Run-time

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
   void *handle;
   void (*addvec)(int *, int *, int *, int);
   char *error;

   /* Dynamically load the shared library that contains addvec() */
   handle = dlopen("./libvector.so", RTLD_LAZY);
   if (!handle) {
      fprintf(stderr, "%s\n", dlerror());
      exit(1);
   }
```

dll.c

Dynamic Linking at Run-time

```
    . . .

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

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

Linking Summary

- **Linking is a technique that allows programs to be constructed from multiple object files.**

- **Linking can happen at different times in a program's lifetime:**
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

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