# Today

- **Linking**

- Case study: Library interpositioning

# Example C Program

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

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

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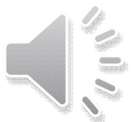*main.c*

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

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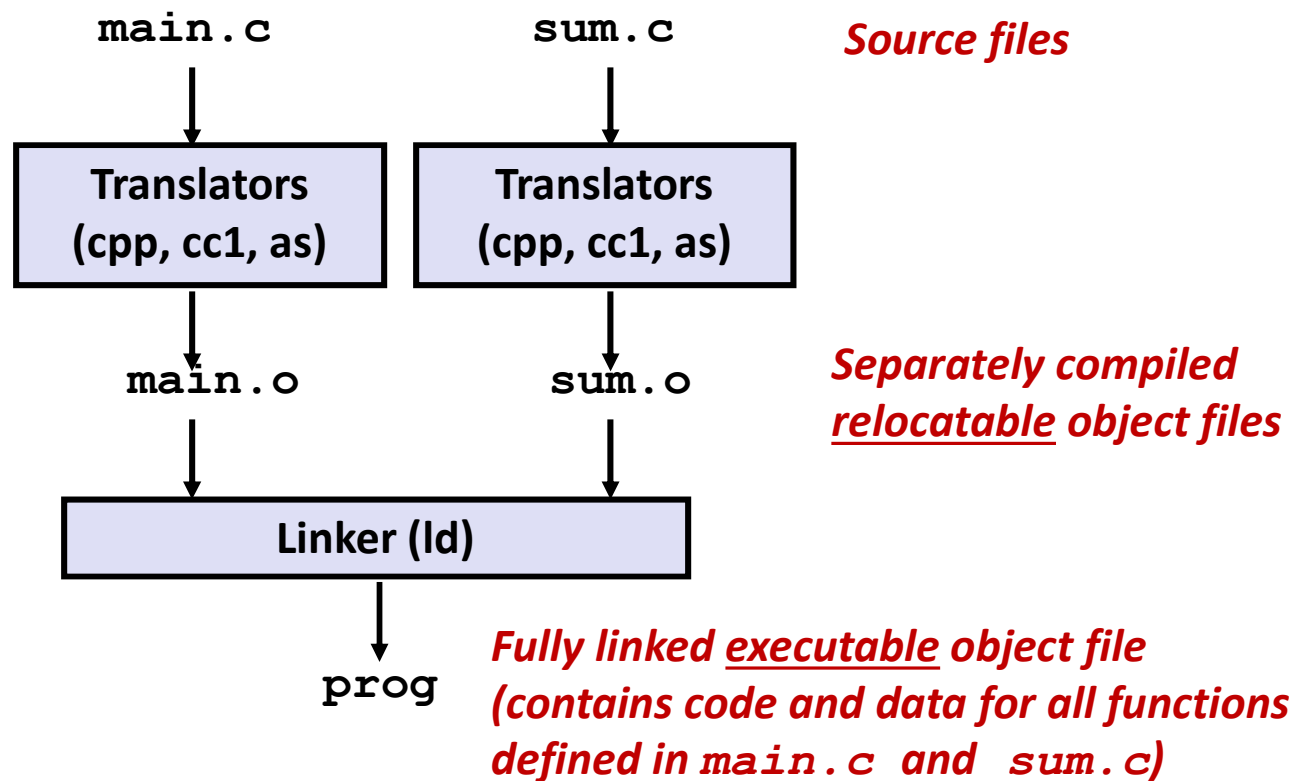*sum.c*

# Static Linking

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`

```
main.c                  sum.c                 Source files

   │                      │
   ▼                      ▼
┌──────────────┐    ┌──────────────┐
│ Translators  │    │ Translators  │
│ (cpp, cc1,as)│    │ (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)
```

# 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 .○ files to their final absolute memory locations in the executable.

  - Updates all references to these symbols to reflect their new positions.

# 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**

# 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.

# Step 1: Symbol Resolution

Referencing a global...

...that's defined here

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

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

int main()
{
    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
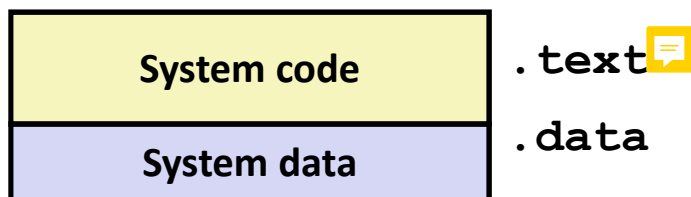a global

Linker knows
nothing of `val`

Referencing
a global...

...that's defined here
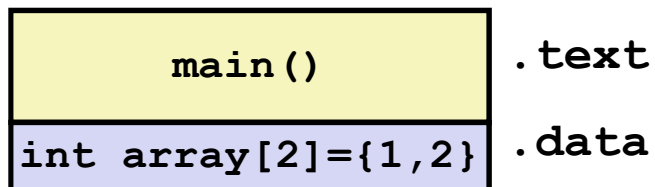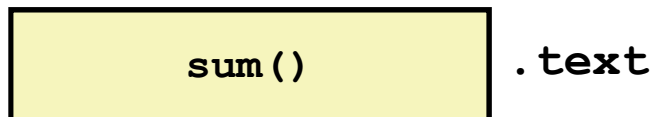
Linker knows
nothing of `i` or `s`

# Step 2: Relocation

**Relocatable Object Files**

| System code | `.text` |
| System data | `.data` |

**main.o**

| main() | `.text` |
| int array[2]={1,2} | `.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`
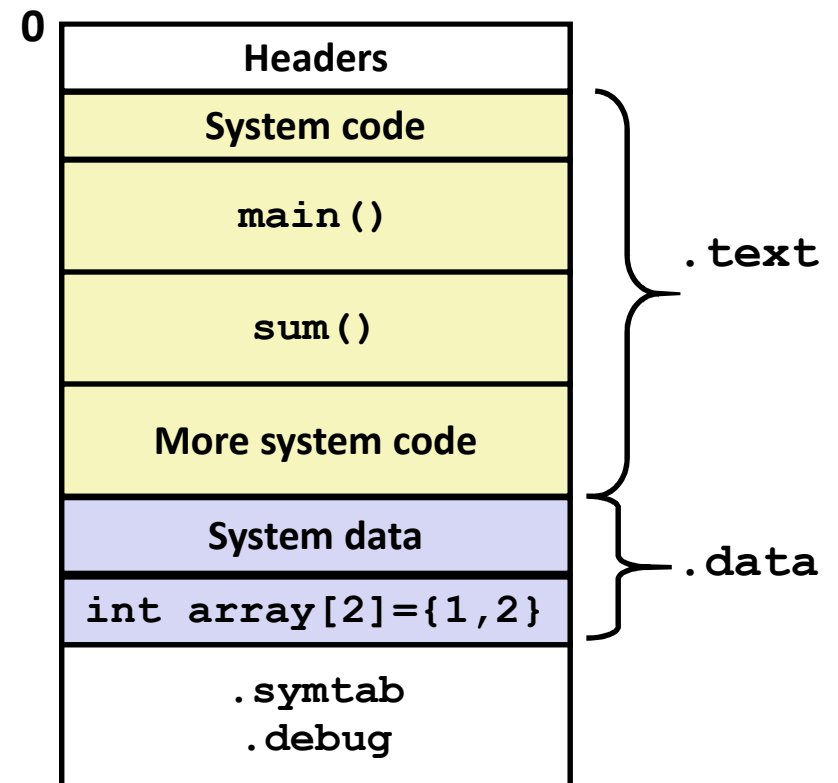
# Packaging Commonly Used Functions

- **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
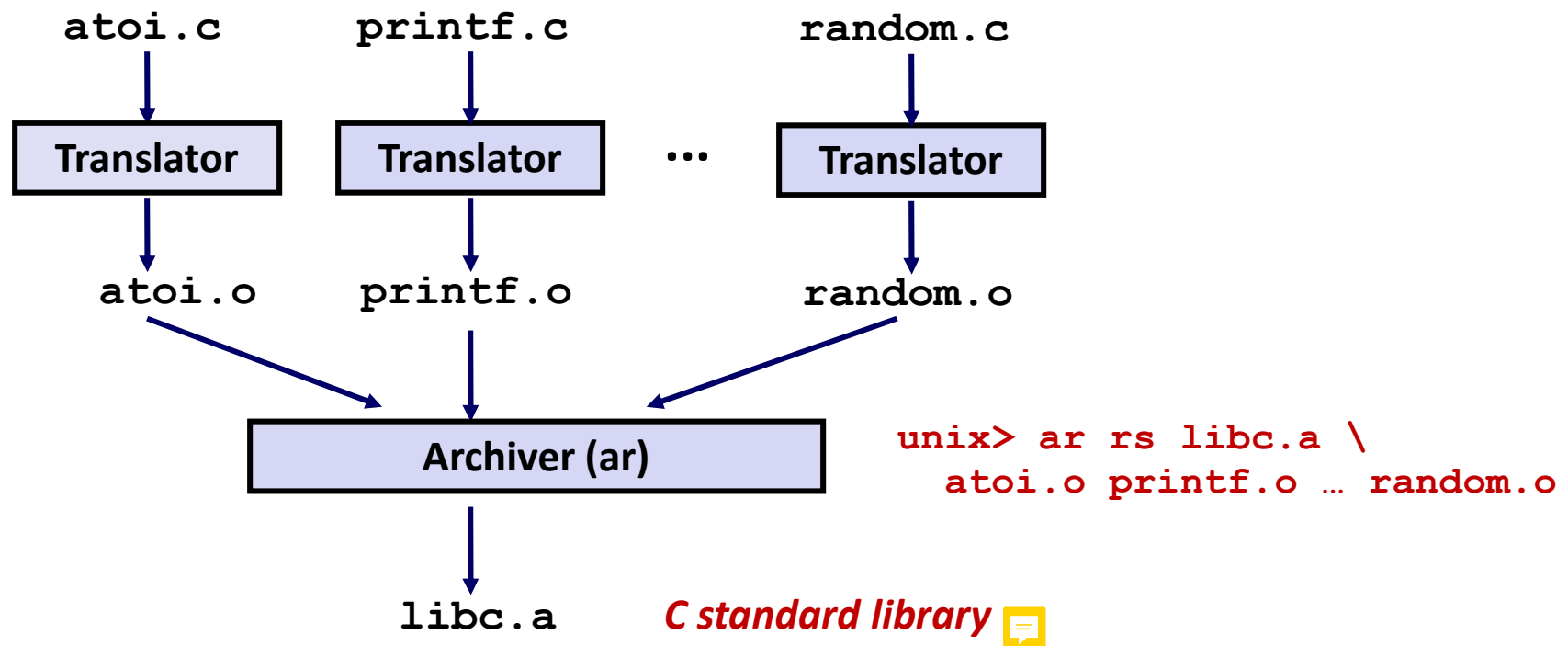        - More efficient, but burdensome on the programmer

# 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



```
atoi.c          printf.c          random.c
  |                |                  |
  v                v                  v
Translator      Translator   ...   Translator
  |                |                  |
  v                v                  v
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 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];
}
```
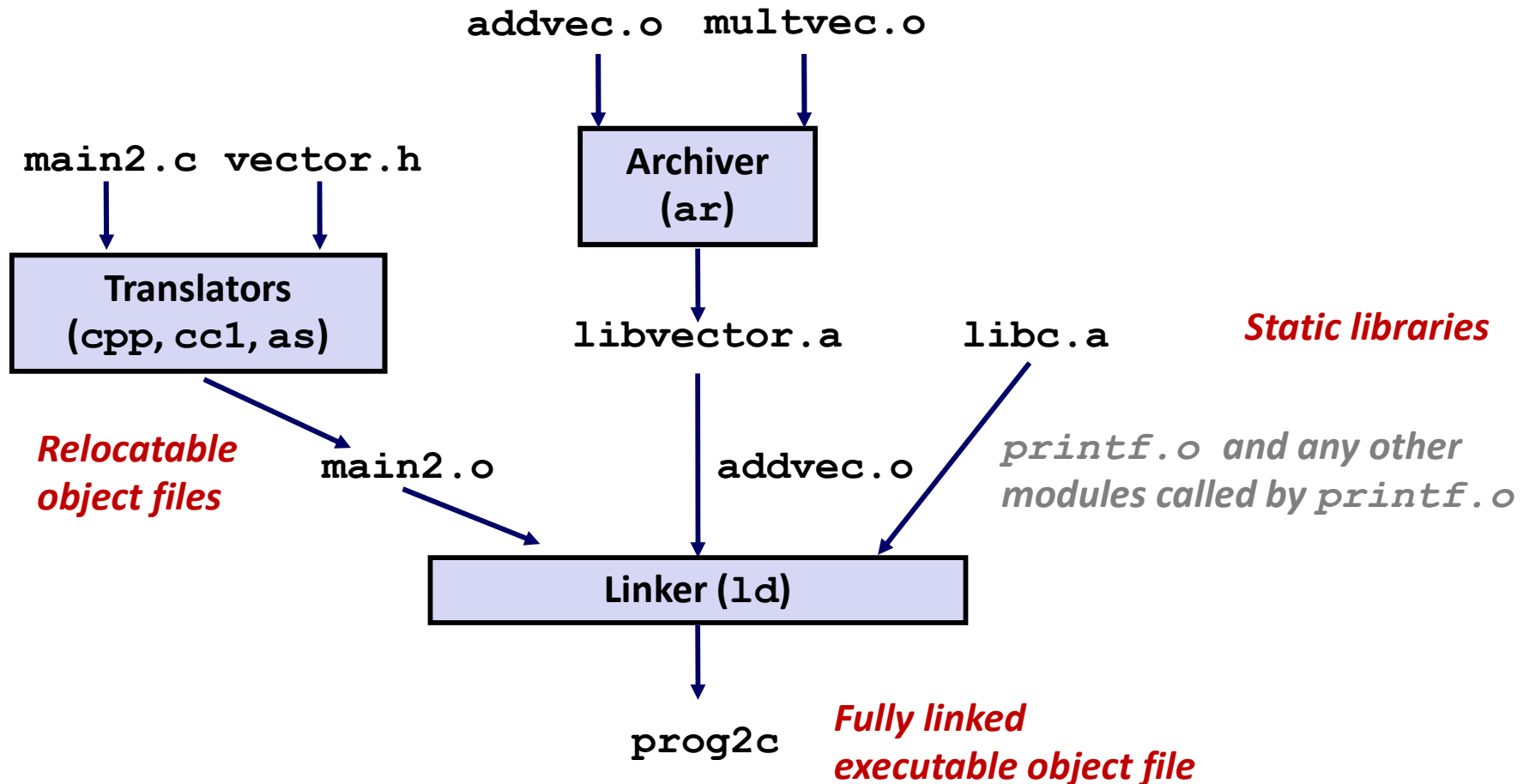addvec.c

```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*
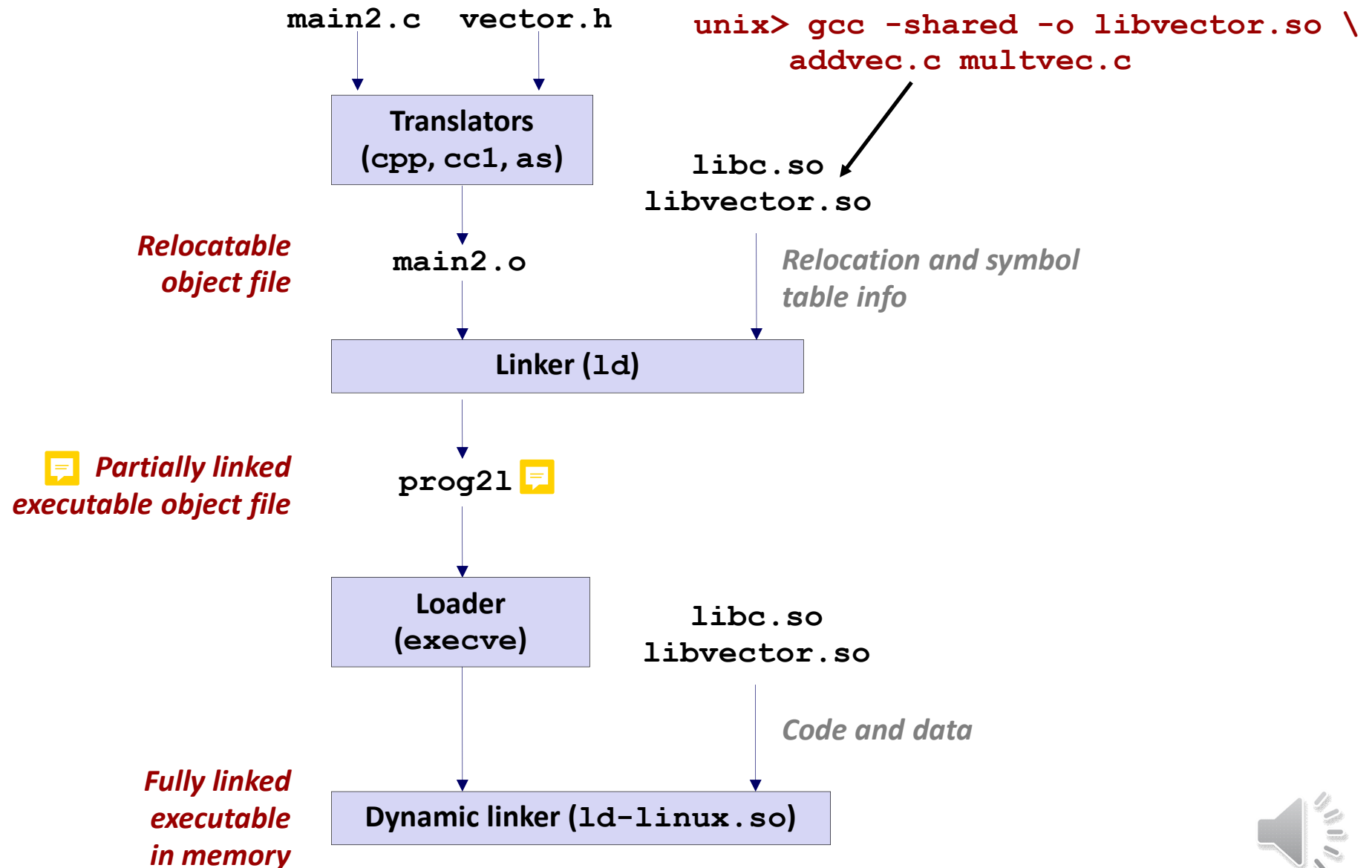  - Also called: dynamic link libraries, DLLs, `.so` files

# 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

main2.c  vector.h

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

**Translators
(cpp, cc1, as)**

libc.so
libvector.so

*Relocatable
object file*

main2.o

*Relocation and symbol
table info*

**Linker (ld)**

*Partially linked
executable object file*

prog21

**Loader
(execve)**

libc.so
libvector.so

*Code and data*

*Fully linked
executable
in memory*

**Dynamic linker (ld-linux.so)**

# 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**

- **Case study: Library interpositioning**

# Case Study: Library Interpositioning

- **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**