Linking

15-213: Introduction to Computer Systems
11th Lecture, Sept. 30, 2010

Instructors:

Randy Bryant and Dave O'Hallaron

Today

- **Linking**
- **Case study: Library interpositioning**

Example C Program

main.c

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```
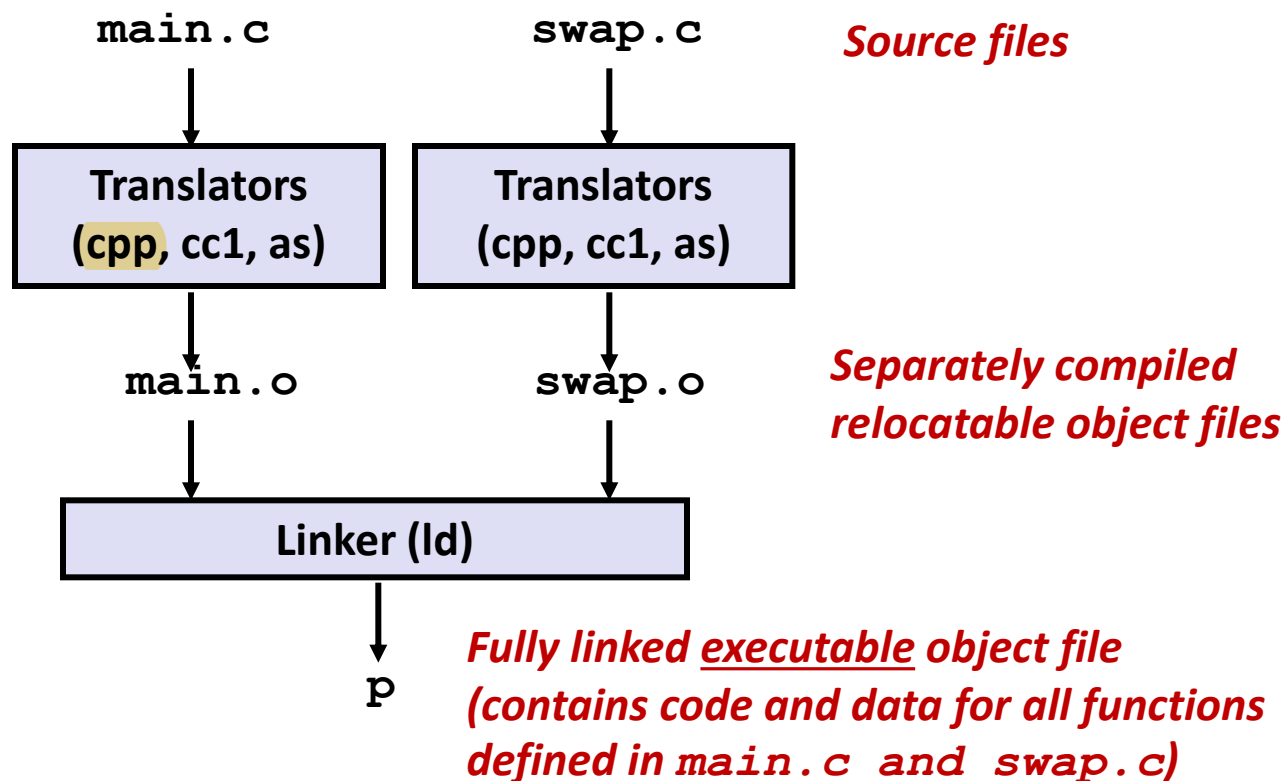
Static Linking

■ **Programs are translated and linked using a *compiler driver*:**
  ▪ unix> *gcc -O2 -g -o p main.c swap.c*
  ▪ unix> *./p*

```
         main.c              swap.c          Source files
            │                   │
            ▼                   ▼
    ┌───────────────┐   ┌───────────────┐
    │  Translators  │   │  Translators  │
    │ (cpp, cc1, as)│   │ (cpp, cc1, as)│
    └───────────────┘   └───────────────┘
            │                   │
            ▼                   ▼
         main.o              swap.o       Separately compiled
                                          relocatable object files
            │                   │
            ▼                   ▼
    ┌───────────────────────────────┐
    │         Linker (ld)           │
    └───────────────────────────────┘
                    │
                    ▼
                    p          Fully linked executable object file
                               (contains code and data for all functions
                               defined in main.c and swap.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* (variables and functions):
    - `void swap() {…}    /* define symbol swap */`
    - `swap();            /* reference symbol a */`
    - `int *xp = &x;      /* define symbol xp, reference x */`

  - Symbol definitions are stored (by compiler) in *symbol table*.
    - Symbol table is an array of structs
    - Each entry includes name, size, and location of symbol.

  - 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.

Relocation Entries

```
typedef struct {
    int offset;
    int symbol:24,
        type:8;
} Elf32_rel;
```

- Offset : section offset of the references to relocate
- Symbol : identifies the symbol that the modified reference should point to.
- Type : tells the linker how to modify the new reference

- ELF defines 11 relocation types.
- two most widely used :
  - R_386_PC32 : 32 bit PC_relative address
    - Add the 32 bit value to the current PC value (address of the next instruction)
  - R_386_32 : 32 bit absolute address

Relocation Algorithm

```
foreach section s {
    foreach relocation entry r {
        refptr = s + r.offset;  /* ptr to reference to be relocated */

        /* Relocate a PC-relative reference */
        if (r.type == R_386_PC32) {
            refaddr = ADDR(s) + r.offset; /* ref's run-time address */
            *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
        }

        /* Relocate an absolute reference */
        if (r.type == R_386_32)
            *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
    }
}
```

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**

■ **Originally proposed by AT&T System V Unix**

- Later adopted by BSD Unix variants and Linux

■ **One unified format for**

- Relocatable object files (`.o`),
- 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

0

ELF header
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 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

0

ELF header
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

Linker Symbols

- **Global symbols**
  - Symbols defined by module *m* that can be referenced by other modules.
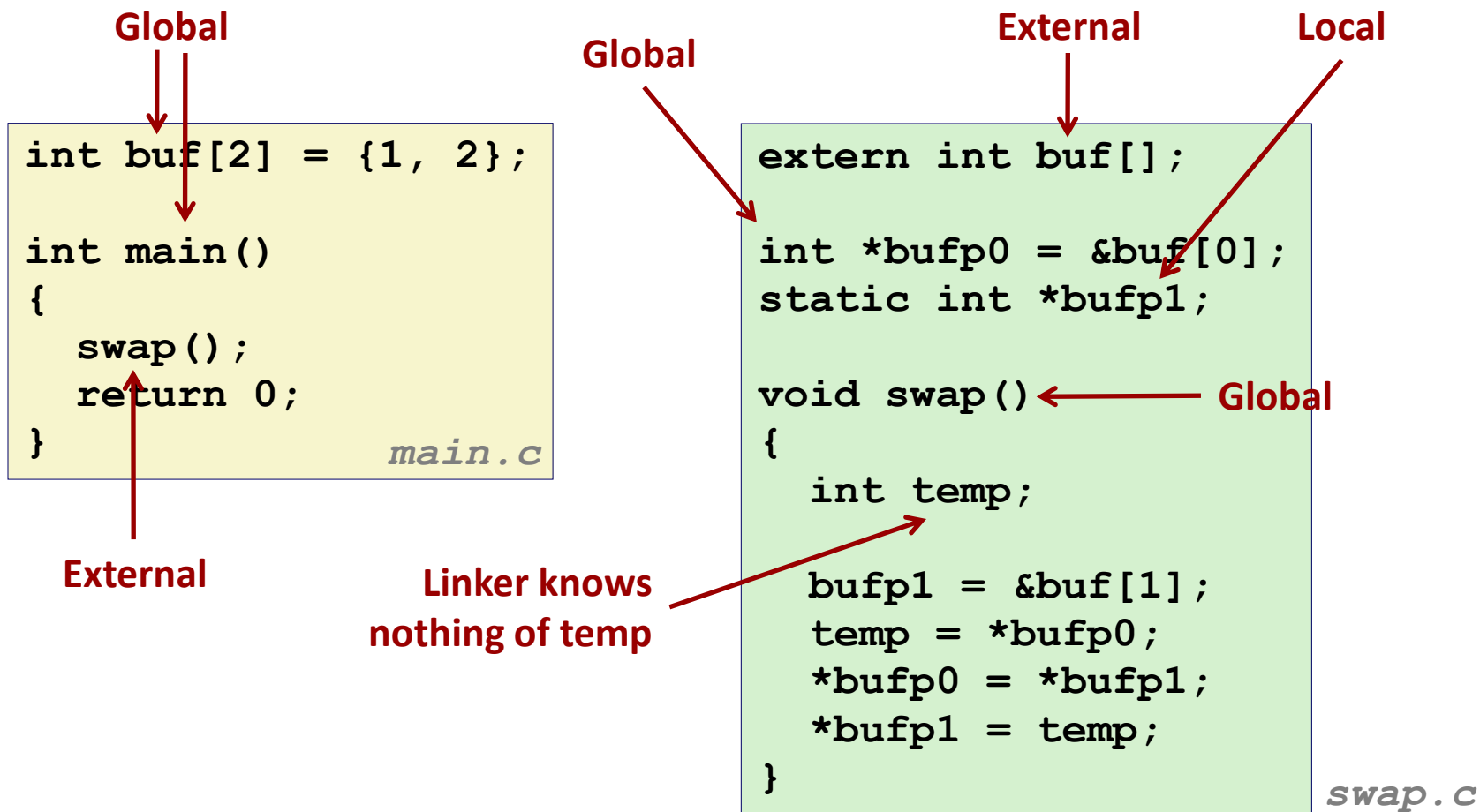 - E.g.: non-`static` C functions and non-`static` global variables.

- **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 variables defined with the `static` attribute.
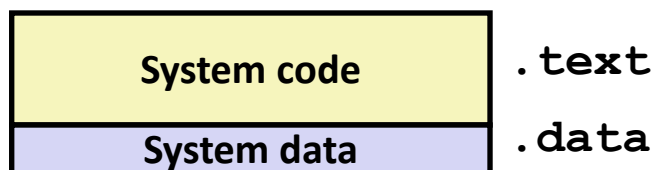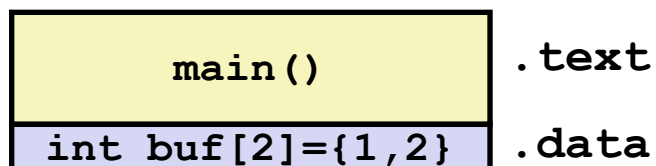 - **Local linker symbols are *not* local program variables**
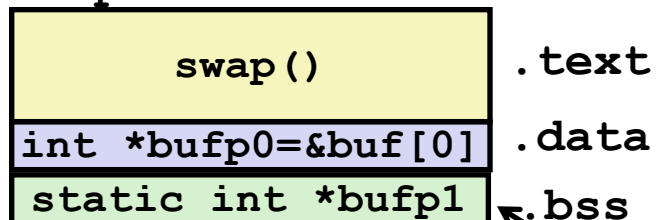
Resolving Symbols

Global

Global

External

Local

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}                    main.c
```

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}                    swap.c
```

Global

External

Linker knows nothing of temp

Relocating Code and Data

Relocatable Object Files

System code	.text
System data	.data

main.o

main()	.text
int buf[2]={1,2}	.data

swap.o

swap()	.text
int *bufp0=&buf[0]	.data
static int *bufp1	.bss

Executable Object File

0

Headers	
System code	
main()	.text
swap()	
More system code	
System data	
int buf[2]={1,2}	.data
int *bufp0=&buf[0]	
int *bufp1	.bss
.symtab .debug	
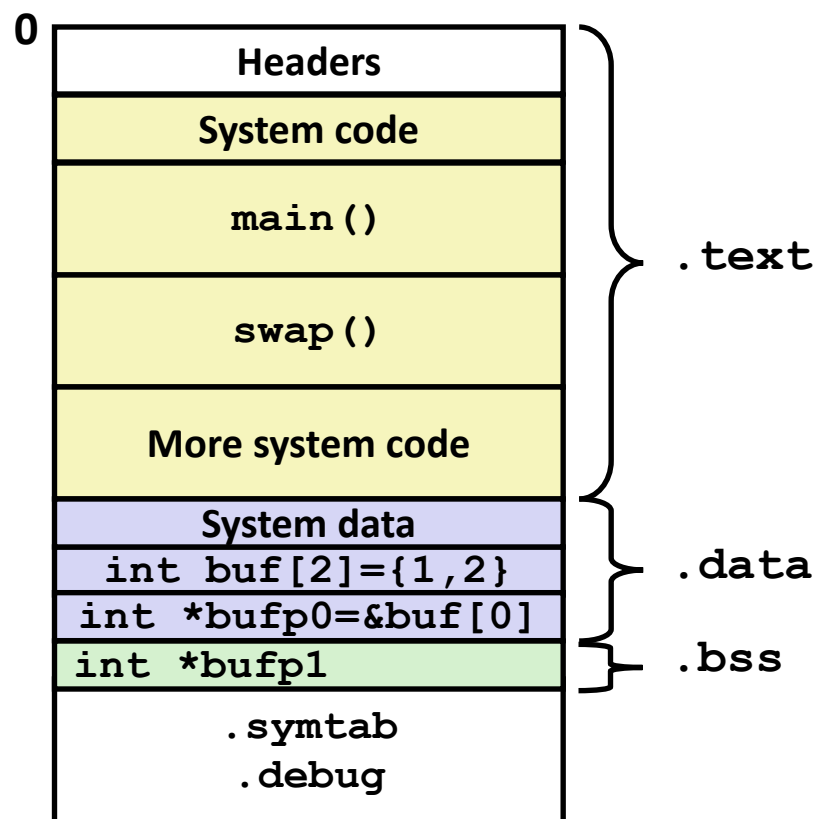
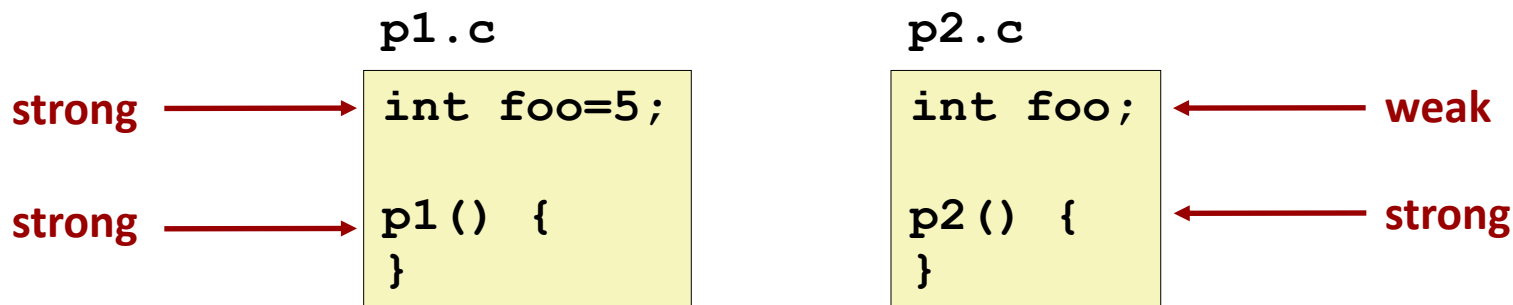**Even though private to swap, requires allocation in .bss**

17

Static variables

- **Static variables** preserve their values even after they are out of their scope **!**

  - A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

  - Static variables are allocated memory in data segment, not stack segment.

  - Static variables (like global variables) are initialized as 0 if not initialized explicitly.

  - Static variables can only be initialized using constant literals.

  - Static global variables and functions are also possible in. The purpose of these is to limit scope of a variable or function to a file.

Strong and Weak Symbols

■ **Program symbols are either strong or weak**

    ■ *Strong*: procedures and initialized globals

    ■ *Weak*: uninitialized globals

p1.c

```
int foo=5;

p1() {
}
```

strong ⟶ `int foo=5;`

strong ⟶ `p1() {`

p2.c

```
int foo;

p2() {
}
```

`int foo;` ⟵ weak

`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 symbol, 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.

Role of .h Files

c1.c

```
#include "global.h"

int f() {
  return g+1;
}
```

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

c2.c

```
#include <stdio.h>
#include "global.h"

int main() {
  if (!init)
    g = 37;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Running Preprocessor

c1.c

```
#include "global.h"

int f() {
   return g+1;
}
```

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

–DINITIALIZE

no initialization

```
int g = 23;
static int init = 1;
int f() {
   return g+1;
}
```

```
int g;
static int init = 0;
int f() {
   return g+1;
}
```

#include causes C preprocessor to insert file verbatim

Role of .h Files (cont.)

c1.c

```
#include "global.h"

int f() {
   return g+1;
}
```

global.h

```
#ifdef INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```

c2.c

```
#include <stdio.h>
#include "global.h"

int main() {
  if (!init)
    g = 37;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

What happens:

```
gcc -o p c1.c c2.c
    ??
gcc -o p c1.c c2.c \
   -DINITIALIZE
    ??
```

Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you use external global variable
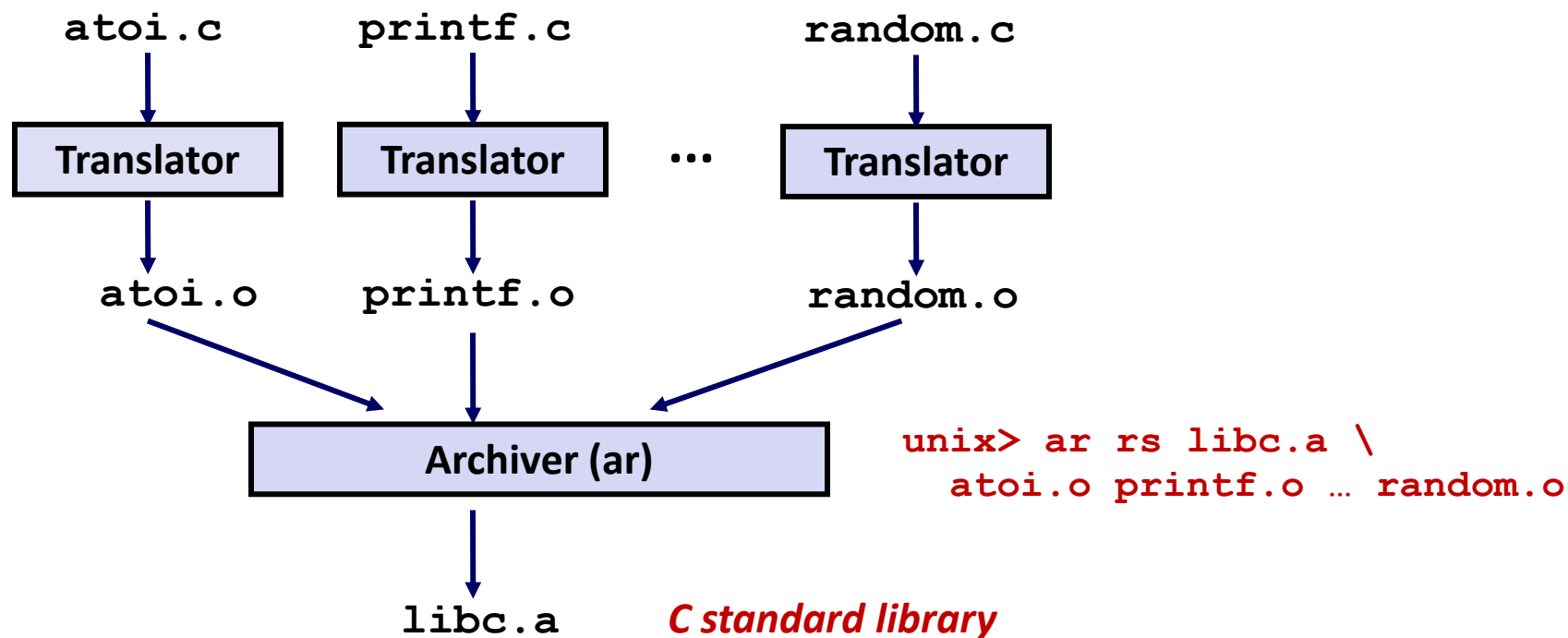
Packaging Commonly Used Functions

■ **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

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)

- 8 MB archive of 1392 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
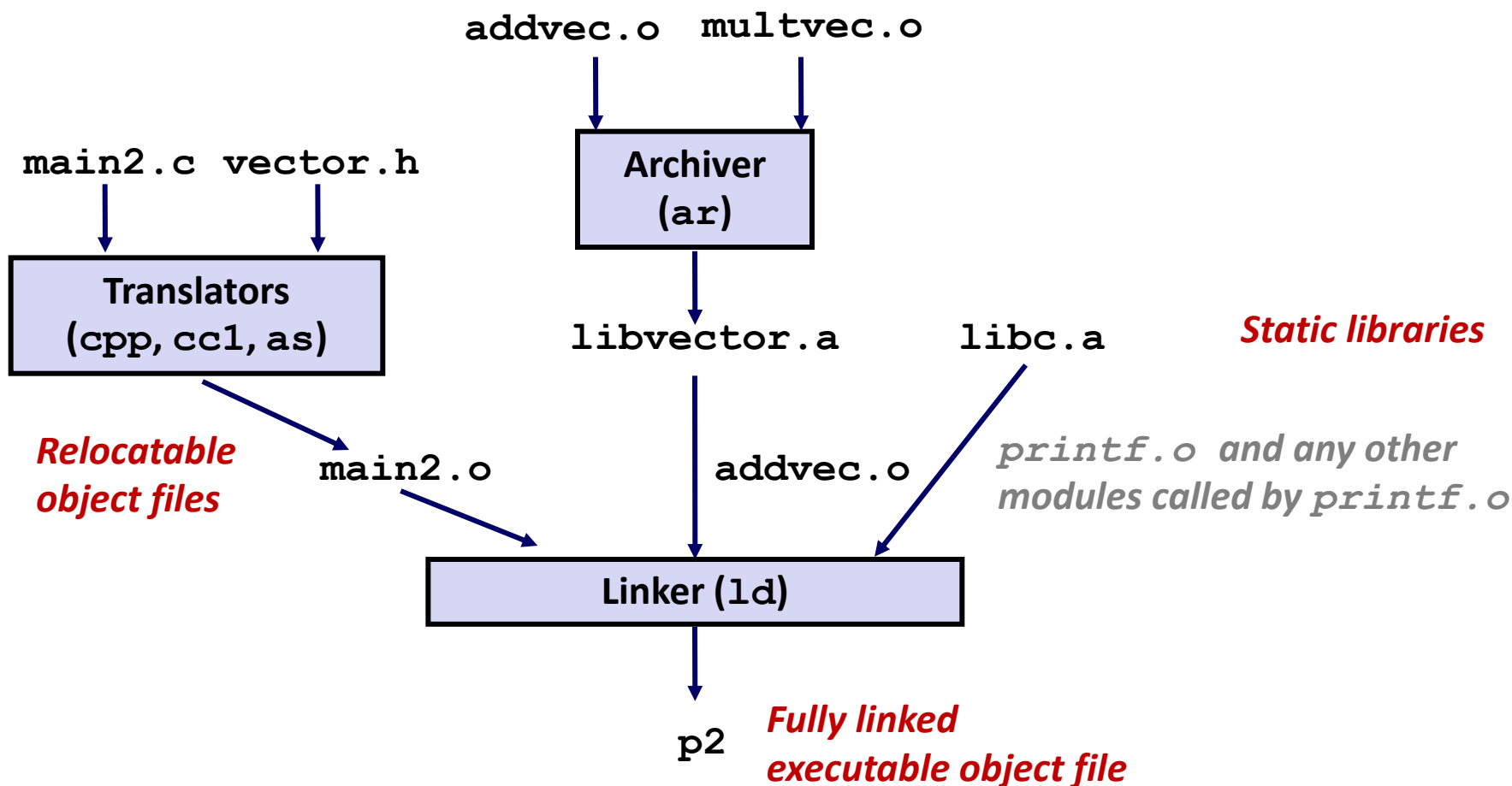
`libm.a` (the C math library)

- 1 MB archive of 401 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

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)

p2 *Fully linked
executable object file*

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*.
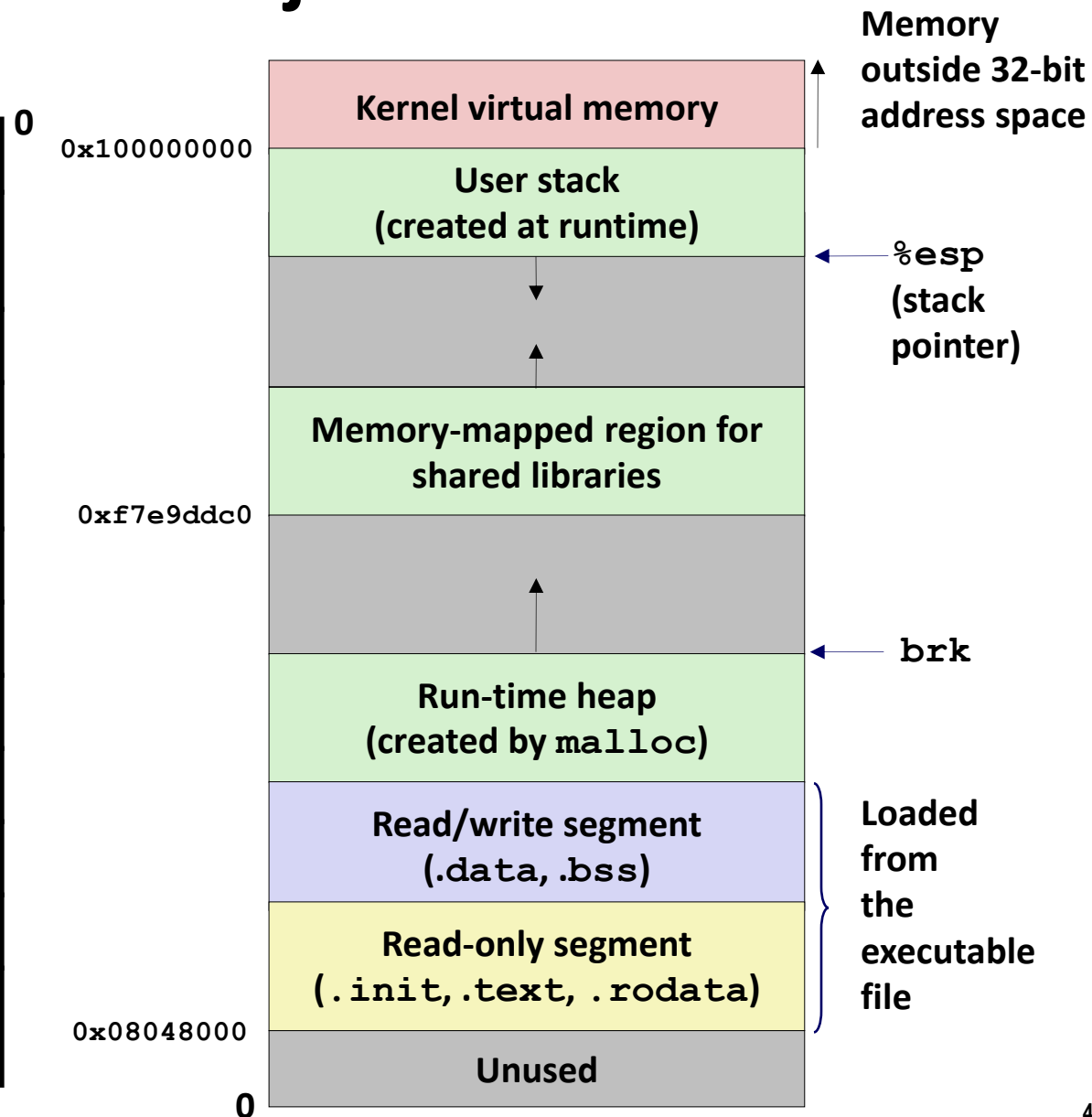 - If any entries in the unresolved list at end of scan, then error.

- **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'
```

Loading Executable Object Files

Executable Object File

ELF header
Program header table (required for executables)
.init section
.text section
.rodata section
.data section
.bss section
.symtab
.debug
.line
.strtab
Section header table (required for relocatables)

0

0x100000000	Kernel virtual memory
	User stack (created at runtime)
	Memory-mapped region for shared libraries
0xf7e9ddc0	
	Run-time heap (created by `malloc`)
	Read/write segment (`.data`,`.bss`)
0x08048000	Read-only segment (`.init`,`.text`, `.rodata`)
0	Unused

Memory outside 32-bit address space

`%esp` (stack pointer)

`brk`

Loaded from the executable file

Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function need std 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
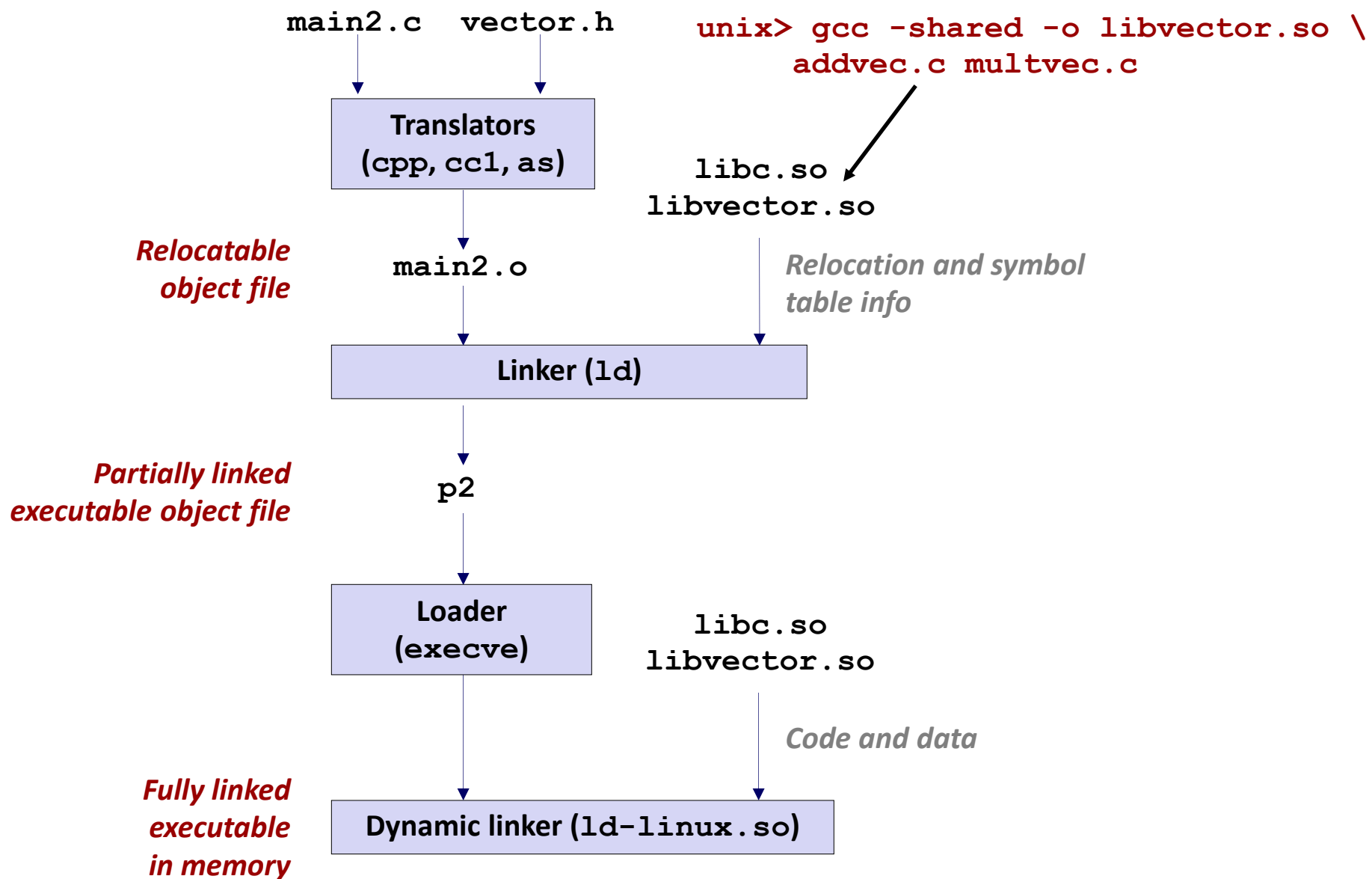
Dynamic Linking at Run-time

```c
#include <stdio.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 lib that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

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;
}
```

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)
    - Interpose calls to libc functions.
  - Behind the scenes encryption
    - Automatically encrypt otherwise unencrypted network connections.
- **Monitoring and Profiling**
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - Malloc tracing
    - Detecting memory leaks
    - **Generating address traces**

Example program

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

int main()
{
    free(malloc(10));
    printf("hello, world\n");
    exit(0);
}
                              hello.c
```

- **Goal: trace the addresses and sizes of the allocated and freed blocks, without modifying the source code.**

- **Three solutions: interpose on the `lib malloc` and `free` functions at compile time, link time, and load/run time.**

Compile-time Interpositioning

```
#ifdef COMPILETIME
/* Compile-time interposition of malloc and free using C
 * preprocessor. A local malloc.h file defines malloc (free)
 * as wrappers mymalloc (myfree) respectively.
 */


#include <stdio.h>
#include <malloc.h>


/*
 * mymalloc - malloc wrapper function
 */
void *mymalloc(size_t size, char *file, int line)
{
    void *ptr = malloc(size);
    printf("%s:%d: malloc(%d)=%p\n", file, line, (int)size,
ptr);
    return ptr;
}
```
mymalloc.c

Compile-time Interpositioning

```
#define malloc(size) mymalloc(size, __FILE__, __LINE__ )
#define free(ptr) myfree(ptr, __FILE__, __LINE__ )

void *mymalloc(size_t size, char *file, int line);
void myfree(void *ptr, char *file, int line);
                                                        malloc.h
```

```
linux> make helloc
gcc -O2 -Wall -DCOMPILETIME -c mymalloc.c
gcc -O2 -Wall -I. -o helloc hello.c mymalloc.o
linux> make runc
./helloc
hello.c:7: malloc(10)=0x501010
hello.c:7: free(0x501010)
hello, world
```

Link-time Interpositioning

```
#ifdef LINKTIME
/* Link-time interposition of malloc and free using the
static linker's (ld) "--wrap symbol" flag. */

#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);


/*
 * __wrap_malloc - malloc wrapper function
 */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

Link-time Interpositioning

```
linux> make hellol
gcc -O2 -Wall -DLINKTIME -c mymalloc.c
gcc -O2 -Wall -Wl,--wrap,malloc -Wl,--wrap,free \
-o hellol hello.c mymalloc.o
linux> make runl
./hellol
malloc(10) = 0x501010
free(0x501010)
hello, world
```

- **The "–Wl" flag passes argument to linker**

- **Telling linker "--wrap,malloc" tells it 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`

```
#ifdef RUNTIME
 /* Run-time interposition of malloc and free based on
  * dynamic linker's (ld-linux.so) LD_PRELOAD mechanism */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

void *malloc(size_t size)
{
    static void *(*mallocp)(size_t size);
    char *error;
    void *ptr;

    /* get address of libc malloc */
    if (!mallocp) {
        mallocp = dlsym(RTLD_NEXT, "malloc");
        if ((error = dlerror()) != NULL) {
            fputs(error, stderr);
            exit(1);
        }
    }
    ptr = mallocp(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

Load/Run-time Interpositioning

`mymalloc.c`

Load/Run-time Interpositioning

```
linux> make hellor
gcc -O2 -Wall -DRUNTIME -shared -fPIC -o mymalloc.so mymalloc.c
gcc -O2 -Wall -o hellor hello.c
linux> make runr
(LD_PRELOAD="/usr/lib64/libdl.so ./mymalloc.so" ./hellor)
malloc(10) = 0x501010
free(0x501010)
hello, world
```

■ **The LD_PRELOAD environment variable tells the dynamic linker to resolve unresolved refs (e.g., to malloc) by looking in libdl.so and mymalloc.so first.**

- **libdl.so** necessary to resolve references to the **dlopen** functions.

Interpositioning Recap

- **Compile Time**
  - Apparent calls to malloc/free get macro-expanded into calls to mymalloc/myfree

- **Link Time**
  - Use linker trick to have special name resolutions
    - malloc $\rightarrow$ __wrap_malloc
    - __real_malloc $\rightarrow$ malloc

- **Compile Time**
  - Implement custom version of malloc/free that use dynamic linking to load library malloc/free under different names