Linking

CS 47: Introduction to Computer Systems
April 9 and 14, 2014

Original lecture by:
Randy Bryant and Dave O'Hallaron

Modified for SJSU by Thomas D. Howell

Example C Program

main.c

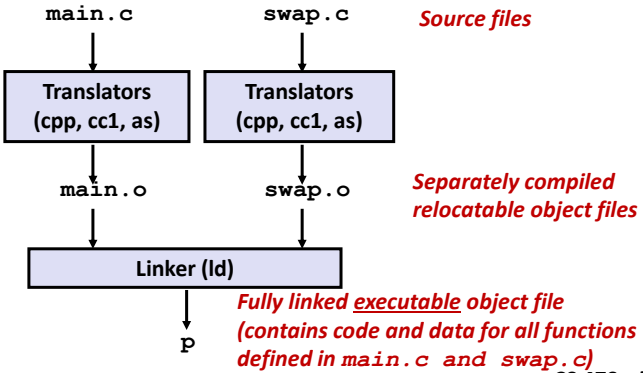
```
int buf[2] = {1, 2};

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`



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.

Three Kinds of Object Files (Modules)

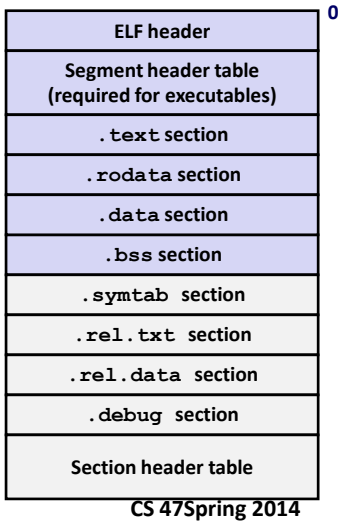
- **Relocatable object file (`.o` file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each `.o` file is produced from exactly one source (`.c`) file
- **Executable object file (`a.out` [`a.exe`] file)**
  - Contains 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**
- **Windows version is called COFF (Common Object File Format). Similar but incompatible.**

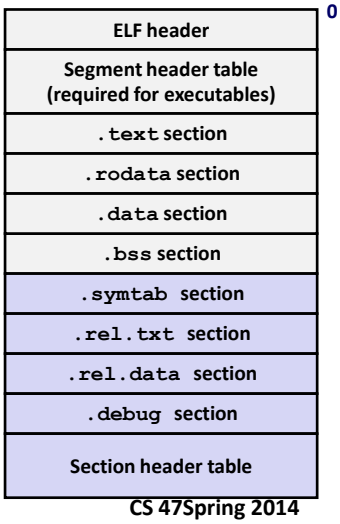
ELF Object File Format

- **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 Storage Start"
  - "Better Save Space"
  - Has section header but occupies no space



ELF Object File Format (cont.)

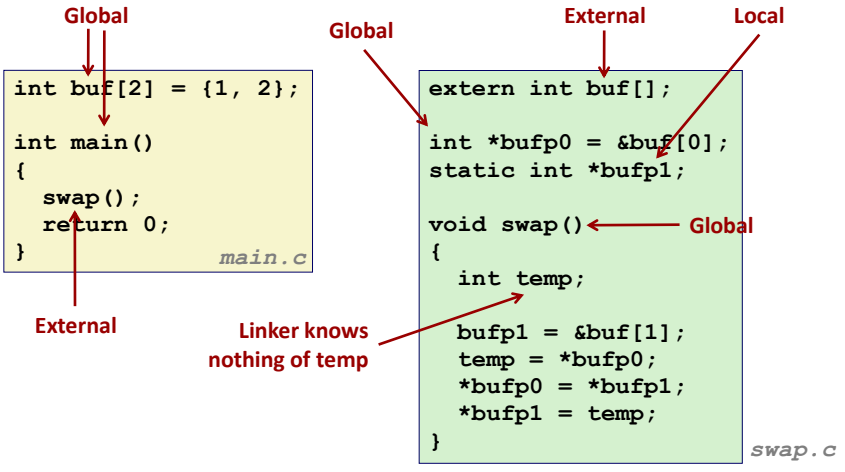
- **`.symtab` section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text` section**
  - Relocation info for **`.text`** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **`.rel.data` 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. (e.g. sleep)
- **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



Practice Problem 7.1

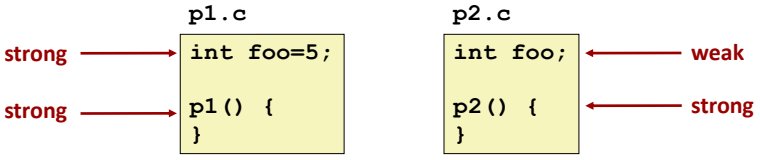
Using program from previous slide:

Symbol	In Swap.o symtab?	Symbol type	Module where defined	Section
buf		External		
bufp0		Global		.data
bufp1	Yes	Local	swap.o	.bss
swap				.text
temp	No			

Note: bufp1 is *not* declared "static" in Fig. 7.1(b), p. 656

Strong and Weak Symbols

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



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;
}
```

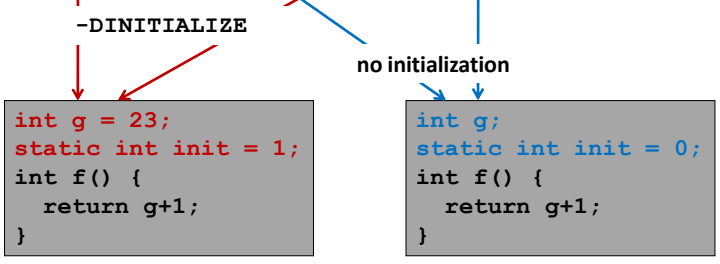
Running Preprocessor

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
```




```
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:

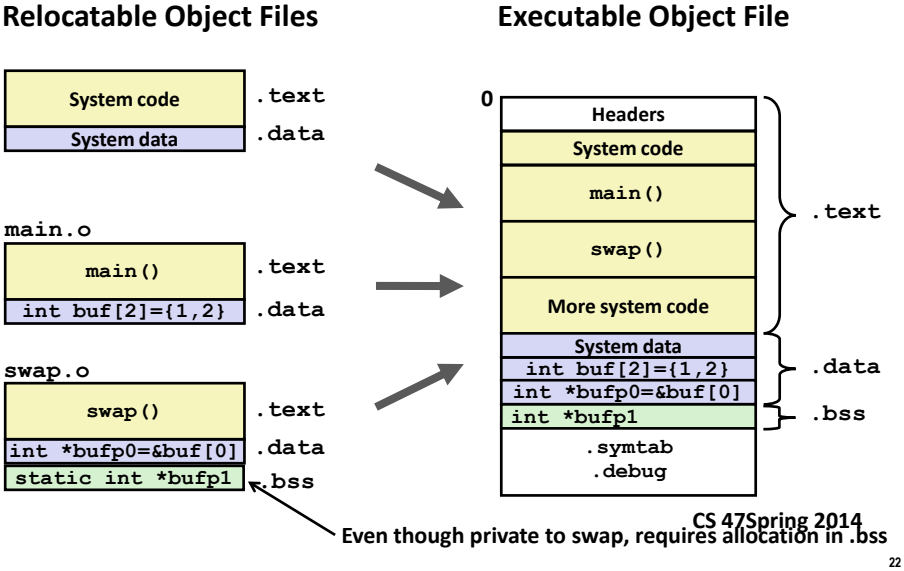
```
gcc -o p c1.c c2.c
   ??
gcc -o p c1.c c2.c \
  -DINITIALIZE
   ??
```

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

Relocating Code and Data



Relocation Info (main)

main.c

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

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

main.o

```
00000000 <_main>:
   0:   55                      push   %ebp
   1:   b8 10 00 00 00          mov    $0x10,%eax
   6:   89 e5                   mov    %esp,%ebp
   8:   83 ec 08                sub    $0x8,%esp
   b:   83 e4 f0                and    $0xfffffff0,%esp
   e:   e8 00 00 00 00          call   13 <_main+0x13>
                        f: DISP32       __alloca
  13:   e8 00 00 00 00          call   18 <_main+0x18>
                        14: DISP32      ___main
  18:   e8 00 00 00 00          call   1d <_main+0x1d>
                        19: DISP32      _swap
  1d:   c9                      leave
  1e:   31 c0                   xor    %eax,%eax
  20:   c3                      ret
```

```
Disassembly of section .data:

00000000 <_buf>:
   0:   01 00 00 00
   4:   02 00 00 00
```

Source: `objdump -r -D`

Relocation Info (swap, `.text`)

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;
}
```

swap.o

```
Disassembly of section .text:

00000000 <_swap>:
 0:     55                  push  %ebp
 1:     b8 04 00 00 00      mov   $0x4,%eax   # &buf[1]
                            2: dir32    _buf
 6:     8b 15 04 00 00 00   mov   0x4,%edx    # buf[1]
                            8: dir32    _buf
 c:     a3 00 00 00 00      mov   %eax,0x0    # bufp1
                            d: dir32    .bss
 11:    a1 00 00 00 00      mov   0x0,%eax    # bufp0
                            12: dir32   .data
 16:    89 e5               mov   %esp,%ebp
 18:    8b 08               mov   (%eax),%ecx # temp
 1a:    89 10               mov   %edx,(%eax) # *bufp0
 1c:    5d                  pop   %ebp
 1d:    89 0d 04 00 00 00   mov   %ecx,0x4    # *bufp1
                            1f: dir32   _buf
 23:    c3                  ret
```

Relocation Info (swap, .data)

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;
}
```

```
Disassembly of section .data:

00000000 <_bufp0>:
   0:   00 00 00 00
        0: dir32        _buf
        ...

Disassembly of section .bss:

00000000 <_bufp1>:
```


Executable Before/After Relocation (.text)

```
0000000 <main>:
   . . .
  18:   e8 00 00 00 00          call   1d <_main+0x1d>
                        19: DISP32      _swap
  1d:   c9                      leave
  . . .
```

0x40105d + 0x13 = 0x401070

```
00401040 <_main>:
  401040:       55                      push   %ebp
  401041:       b8 10 00 00 00          mov    $0x10,%eax
  401046:       89 e5                   mov    %esp,%ebp
  401048:       83 ec 08                sub    $0x8,%esp
  40104b:       83 e4 f0                and    $0xfffffff0,%esp
  40104e:       e8 4d 00 00 00          call   4010a0 <___chkstk>
  401053:       e8 a8 00 00 00          call   401100 <___main>
  401058:       e8 13 00 00 00          call   401070 <_swap>
  40105d:       c9                      leave
  40105e:       31 c0                   xor    %eax,%eax
  401060:       c3                      ret
```

```
   1:   b8 04 00 00 00          mov    $0x4,%eax
                        2: dir32       _buf
   6:   8b 15 04 00 00 00       mov    0x4,%edx
                        8: dir32       _buf
   c:   a3 00 00 00 00          mov    %eax,0x0
                        d: dir32       .bss
  11:   a1 00 00 00 00          mov    0x0,%eax
                        12: dir32      .data
 …
  1d:   89 0d 04 00 00 00       mov    %ecx,0x4
                        1f: dir32      _buf
  23:   c3                      ret
```

```
00401070 <_swap>:
  401070:       55                      push   %ebp
  401071:       b8 04 20 40 00          mov    $0x402004,%eax
  401076:       8b 15 04 20 40 00       mov    0x402004,%edx
  40107c:       a3 08 30 40 00          mov    %eax,0x403008
  401081:       a1 08 20 40 00          mov    0x402008,%eax
  401086:       89 e5                   mov    %esp,%ebp
  401088:       8b 08                   mov    (%eax),%ecx
  40108a:       89 10                   mov    %edx,(%eax)
  40108c:       5d                      pop    %ebp
  40108d:       89 0d 04 20 40 00       mov    %ecx,0x402004
  401093:       c3                      ret
```


Executable After Relocation (.data)

```
Disassembly of section .data:

 402000 <buf>:
 402000:       01 00 00 00 02 00 00 00
 402008 <bufp0>:
 402008:       00 20 40 00
```

Packaging Commonly Used Functions

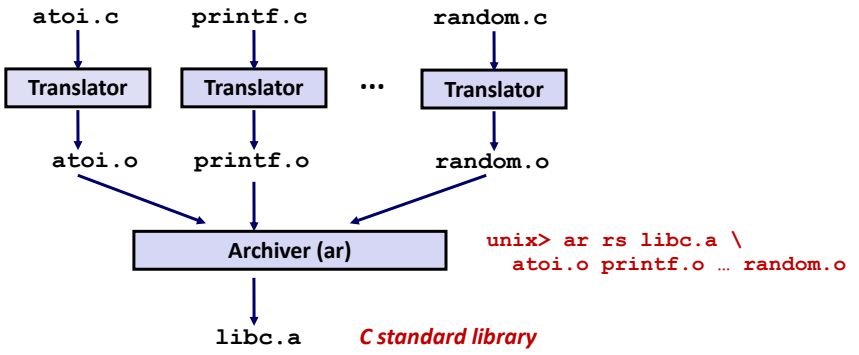
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **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



- 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

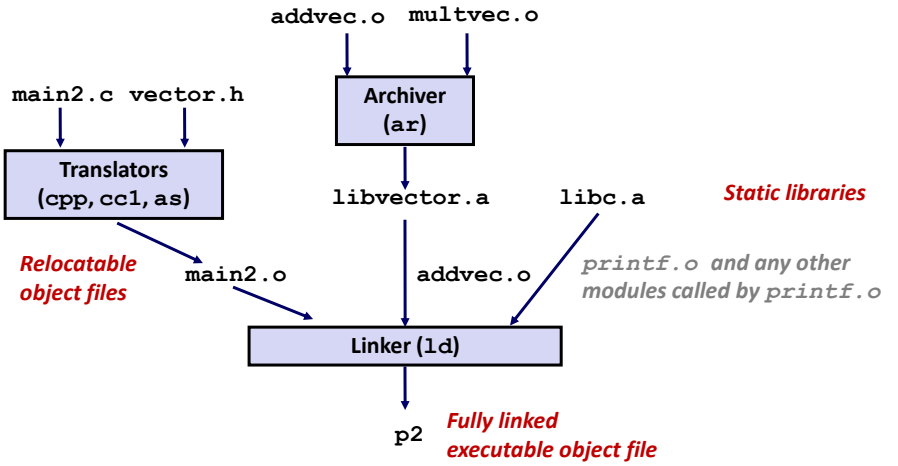
`libm.a` (the C math library)

- 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



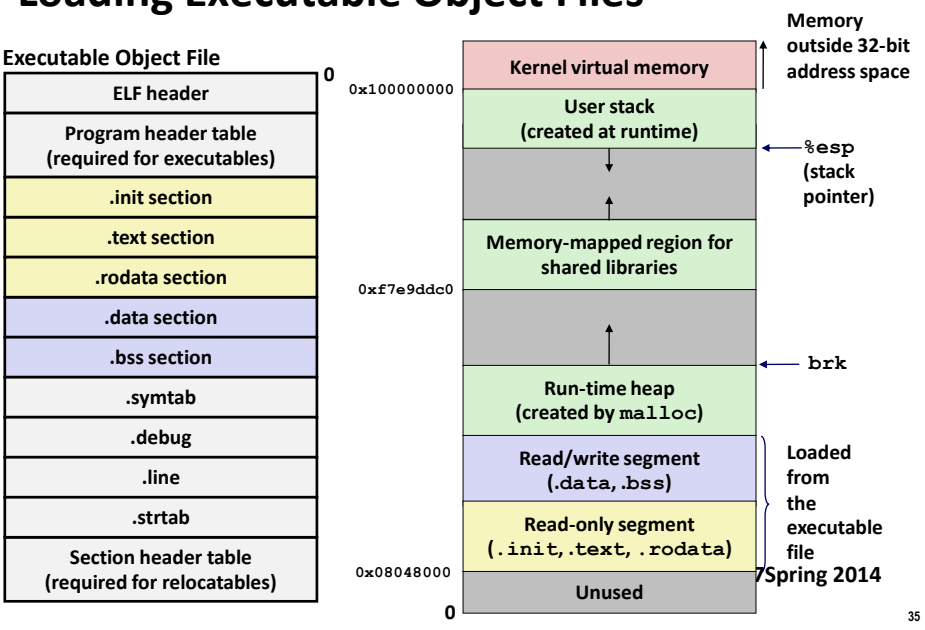
Using Static Libraries

- **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



Shared Libraries

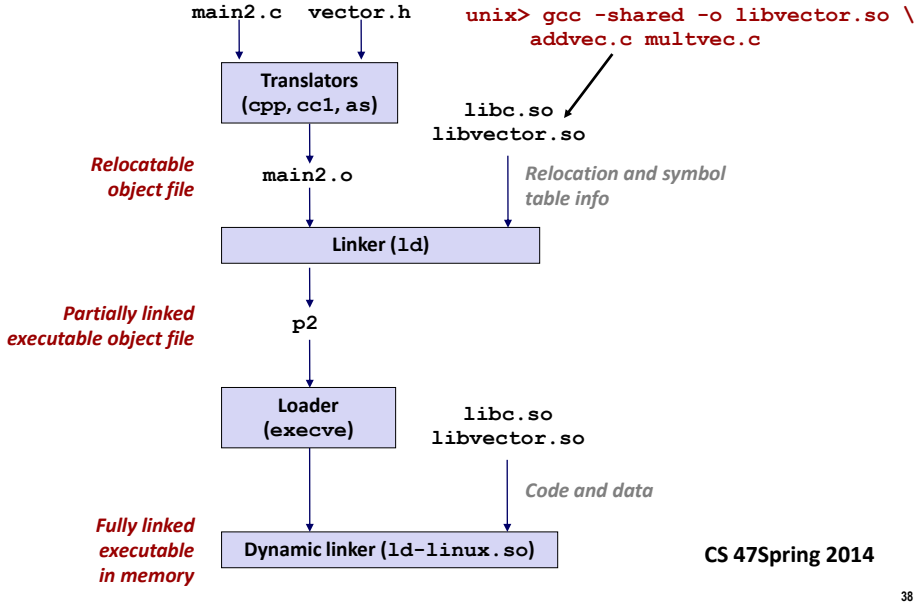
- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function need std libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- **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

```
#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;
}
```