

## Linking

15-213 / 18-213: Introduction to Computer Systems  
12<sup>th</sup> Lecture, Feb. 23, 2012

### Instructors:

Todd C. Mowry & Anthony Rowe

1

## Today

- Linking
- Case study: Library interpositioning

2

## 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;

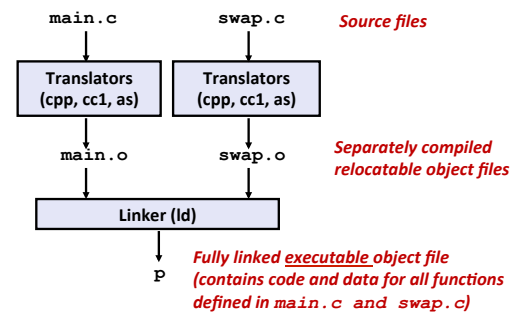
void swap()
{
    int temp;

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

3

## Static Linking

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



4

## 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

5

## 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.

6

## 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.

7

## 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.

8

## 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

9

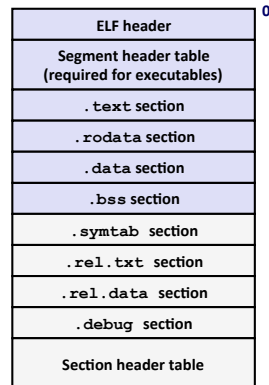
## Executable and Linkable Format (ELF)

- **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**

10

## ELF Object File Format

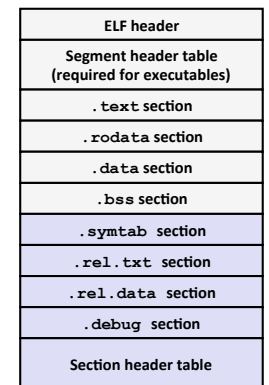
- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **.text section**
  - Code
- **.rodata section**
  - Read only data: jump tables, ...
- **.data section**
  - Initialized global variables
- **.bss section**
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space



11

## 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



12

## 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

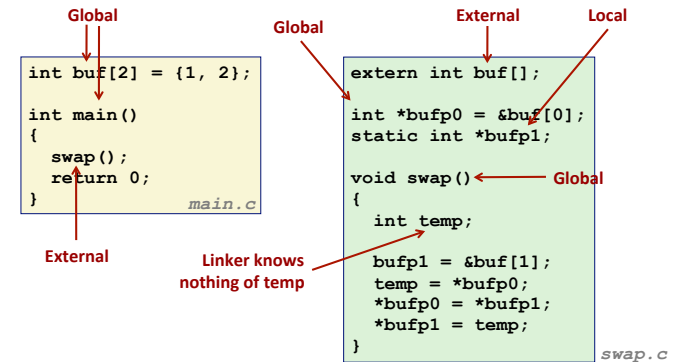
- Global symbols that are referenced by module *m* but defined by some other module.

### 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**

13

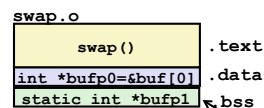
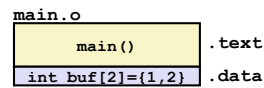
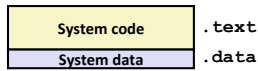
## Resolving Symbols



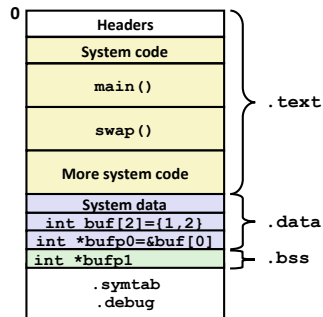
14

## Relocating Code and Data

### Relocatable Object Files



### Executable Object File



Even though private to swap, requires allocation in .bss

15

## Relocation Info (main)

main.c	main.o
<code>int buf[2] = {1,2};</code>	00000000 <main>: 0: 8d 4c 24 04 lea 0x4(%esp),%ecx
<code>int main() { swap(); return 0; }</code>	4: 83 e4 f0 and \$0xffffffff0,%esp
	7: ff 71 fc pushl 0xffffffffc(%ecx)
	a: 55 push %ebp
	b: 89 e5 mov %esp,%ebp
	d: 51 push %ecx
	e: 83 ec 04 sub \$0x4,%esp
	11: e8 fc ff ff call 12 <main+0x12>
	12: R_386_PC32 swap
	16: 83 c4 04 add \$0x4,%esp
	19: 31 c0 xor %eax,%eax
	1b: 59 pop %ecx
	1c: 5d pop %ebp
	1d: 8d 61 fc lea 0xffffffffc(%ecx),%esp
	20: c3 ret

Disassembly of section .data:

Source: objdump -r -d

```
00000000 <buf>:
0: 01 00 00 00 02 00 00 00
```

16

## Relocation Info (swap, .text)

<pre>swap.c extern int buf[];  int *bufp0 = &amp;buf[0];  static int *bufp1;  void swap() {     int temp;      bufp1 = &amp;buf[1];     temp = *bufp0;     *bufp0 = *bufp1;     *bufp1 = temp; }</pre>	<pre>swap.o Disassembly of section .text: 00000000 &lt;swap&gt;: 0: 8b 15 00 00 00 00    mov     0x0,%edx                 2: R_386_32    buf 6: a1 04 00 00 00 00    mov     0x4,%eax                 7: R_386_32    buf b: 55                  push   %ebp c: 89 e5              mov     %esp,%ebp e: c7 05 00 00 00 00 04 movl    \$0x4,0x0 15: 00 00 00                 10: R_386_32    .bss                 14: R_386_32    buf 18: 8b 08              mov     (%eax),%ecx 1a: 89 10              mov     %edx,(%eax) 1c: 5d                  pop     %ebp 1d: 89 0d 04 00 00 00    mov     %ecx,0x4                 1f: R_386_32    buf 23: c3                  ret</pre>
--	--

17

## Relocation Info (swap, .data)

<pre>swap.c extern int buf[];  int *bufp0 =     &amp;buf[0]; static int *bufp1;  void swap() {     int temp;      bufp1 = &amp;buf[1];     temp = *bufp0;     *bufp0 = *bufp1;     *bufp1 = temp; }</pre>	<pre>Disassembly of section .data: 00000000 &lt;bufp0&gt;: 0: 00 00 00 00                 0: R_386_32    buf</pre>
---	--

18

## Executable Before/After Relocation (.text)

<pre>00000000 &lt;main&gt;: . . . e: 83 ec 04          sub     \$0x4,%esp 11: e8 fc ff ff ff    call   12 &lt;main+0x12&gt;                 12: R_386_PC32    swap 16: 83 c4 04          add     \$0x4,%esp . . .</pre>	<pre>0x8048396 + 0x1a = 0x80483b0</pre>
<pre>08048380 &lt;main&gt;: 8048380: 8d 4c 24 04        lea    0x4(%esp),%ecx 8048384: 83 e4 f0           and    \$0xfffffffff0,%esp 8048387: ff 71 fc           pushl 0xfffffffffc(%ecx) 804838a: 55                push   %ebp 804838b: 89 e5             mov    %esp,%ebp 804838d: 51                push   %ecx 804838e: 83 ec 04         sub    \$0x4,%esp 8048391: e8 1a 00 00 00    call  80483b0 &lt;swap&gt; 8048396: 83 c4 04         add    \$0x4,%esp 8048399: 31 c0             xor    %eax,%eax 804839b: 59                pop    %ecx 804839c: 5d                pop    %ebp 804839d: 8d 61 fc         lea   0xfffffffffc(%ecx),%esp 80483a0: c3                ret</pre>	

19

```
0: 8b 15 00 00 00 00    mov     0x0,%edx
                2: R_386_32    buf
6: a1 04 00 00 00 00    mov     0x4,%eax
                7: R_386_32    buf
. . .
e: c7 05 00 00 00 00 04 movl    $0x4,0x0
15: 00 00 00
                10: R_386_32    .bss
                14: R_386_32    buf
. . .
1d: 89 0d 04 00 00 00    mov     %ecx,0x4
                1f: R_386_32    buf
23: c3                ret
```

```
080483b0 <swap>:
80483b0: 8b 15 20 96 04 08    mov     0x8049620,%edx
80483b6: a1 24 96 04 08      mov     0x8049624,%eax
80483bb: 55                  push   %ebp
80483bc: 89 e5              mov     %esp,%ebp
80483be: c7 05 30 96 04 08 24 movl    $0x8049624,0x8049630
80483c5: 96 04 08
80483c8: 8b 08              mov     (%eax),%ecx
80483ca: 89 10              mov     %edx,(%eax)
80483cc: 5d                  pop     %ebp
80483cd: 89 0d 24 96 04 08    mov     %ecx,0x8049624
80483d3: c3                ret
```

20

## Executable After Relocation (.data)

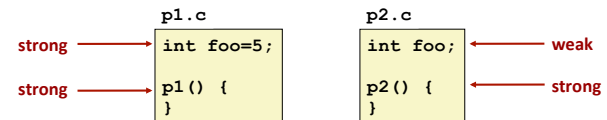
Disassembly of section .data:

```
08049620 <buf>:
8049620: 01 00 00 00 02 00 00 00
08049628 <bufp0>:
8049628: 20 96 04 08
```

21

## Strong and Weak Symbols

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



22

## 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`

23

## Linker Puzzles

<code>int x;</code> <code>p1() {}</code>	<code>p1() {}</code>	Link time error: two strong symbols (p1)
<code>int x;</code> <code>p1() {}</code>	<code>int x;</code> <code>p2() {}</code>	References to <code>x</code> will refer to the same uninitialized int. Is this what you really want?
<code>int x;</code> <code>int y;</code> <code>p1() {}</code>	<code>double x;</code> <code>p2() {}</code>	Writes to <code>x</code> in <code>p2</code> might overwrite <code>y</code> ! Evil!
<code>int x=7;</code> <code>int y=5;</code> <code>p1() {}</code>	<code>double x;</code> <code>p2() {}</code>	Writes to <code>x</code> in <code>p2</code> will overwrite <code>y</code> ! Nasty!
<code>int x=7;</code> <code>p1() {}</code>	<code>int x;</code> <code>p2() {}</code>	References to <code>x</code> will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

24

## Role of .h Files

c1.c

```
#include "global.h"

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

global.h

```
#ifndef 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;
}
```

25

## Running Preprocessor

c1.c

```
#include "global.h"

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

global.h

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

-DINITIALIZE

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

no initialization

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

#include causes C preprocessor to insert file verbatim

26

## Role of .h Files (cont.)

c1.c

```
#include "global.h"

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

global.h

```
#ifndef 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
??
```

27

## 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

28

## 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

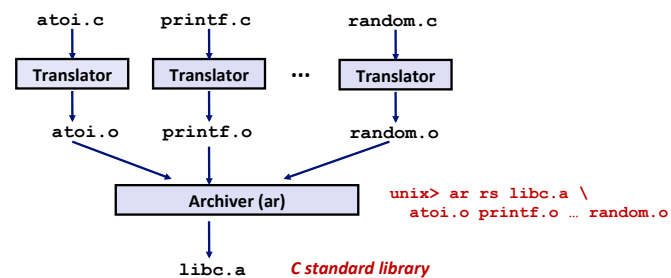
29

## 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.

30

## Creating Static Libraries



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

31

## 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
...
  
```

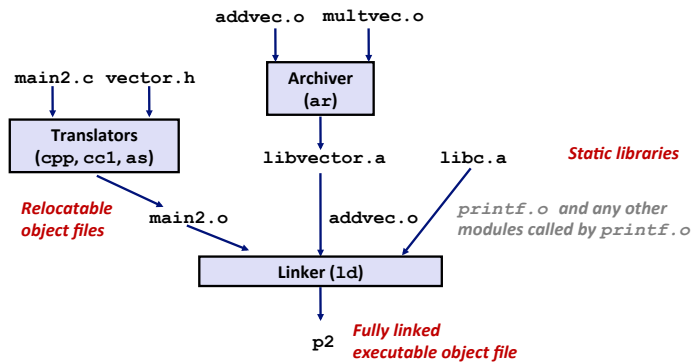
```

% 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
...
  
```

32



## Linking with Static Libraries



33

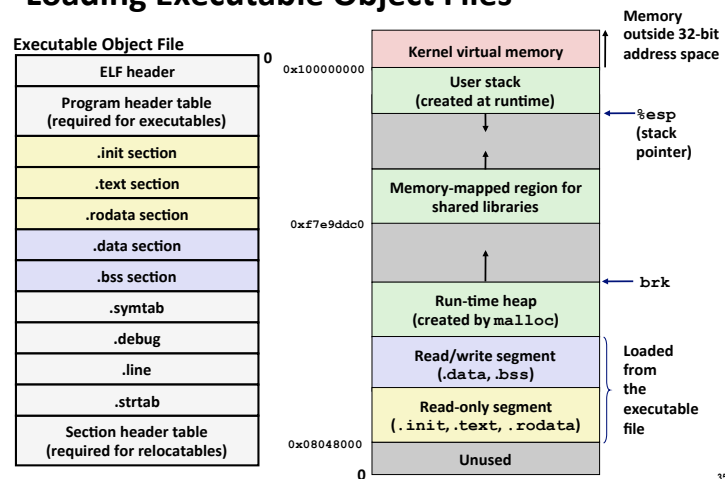
## Using Static Libraries

- **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'
```

34

## Loading Executable Object Files



35

## 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

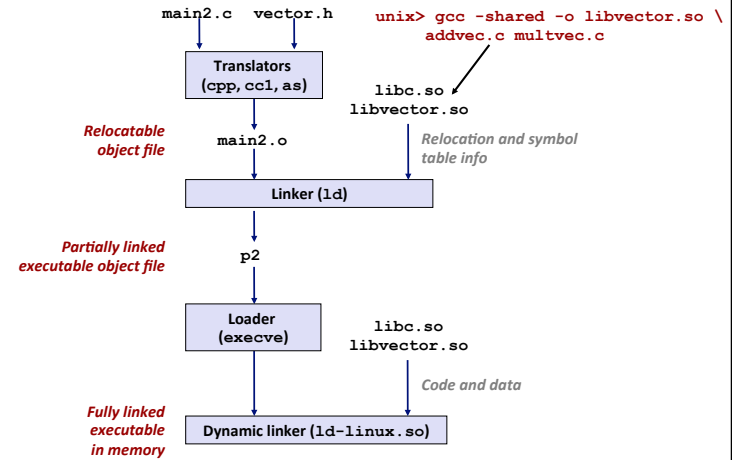
36

## 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

37

## Dynamic Linking at Load-time



38

## Dynamic Linking at Run-time

```

#include <stdio.h>
#include <dlopen.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);
    }
  
```

39

## 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;
}
  
```

40

## Today

- Linking
- Case study: Library interpositioning

41

## 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.

42

## 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**

43

## 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.**

44

## 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

45

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

```

46

## 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

47

## Link-time Interpositioning

```

linux> make hello1
gcc -O2 -Wall -DLINKTIME -c mymalloc.c
gcc -O2 -Wall -Wl,--wrap,malloc -Wl,--wrap,free \
-o hello1 hello.c mymalloc.o
linux> make run1
./hello1
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

48

```

#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

49

## 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.

50

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

51