LINKING

Spring 2012

Euiseong Seo

(euiseong@gmail.com)
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;
}
Programs are translated and linked using a *compiler driver*:

- `unix> gcc -O2 -g -o p main.c swap.c`
- `unix> ./p`

---

**Static Linking**

- **Source files**
- **Separately compiled relocatable object files**
- **Fully linked executable object file (contains code and data for all functions defined in main.c and swap.c)**
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
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.
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 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
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-\texttt{static} C functions and non-\texttt{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 \texttt{static} attribute.
  - \textbf{Local linker symbols are not local program variables}
```c
int buf[2] = {1, 2};
int main()
{
    swap();
    return 0;
}

void swap()
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```
Relocatable Object Files

- **System code**
  - `main()`
  - `int buf[2]={1,2}`
- **System data**
  - `int *bufp0=&buf[0]`

Executable Object File

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

Disassembly of section .data:

```
00000000 <buf>:
  0:   01 00 00 00 02 00 00 00
```
extern int buf[];

int *bufp0 = &buf[0];

static int *bufp1;

void swap()
{
    int temp;

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

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

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
### Executable Before/After Relocation

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Arguments</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>08048380</td>
<td>8d 4c 24 04</td>
<td>lea</td>
<td>0x4(%esp),%ecx</td>
<td></td>
</tr>
<tr>
<td>08048384</td>
<td>83 e4 f0</td>
<td>and</td>
<td>$0xffffffff0,%esp</td>
<td></td>
</tr>
<tr>
<td>08048387</td>
<td>ff 71 fc</td>
<td>pushl</td>
<td>0xfffffffffc(%ecx)</td>
<td></td>
</tr>
<tr>
<td>0804838a</td>
<td>55</td>
<td>push</td>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>0804838b</td>
<td>89 e5</td>
<td>mov</td>
<td>%esp,%ebp</td>
<td></td>
</tr>
<tr>
<td>0804838d</td>
<td>51</td>
<td>push</td>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>0804838e</td>
<td>83 ec 04</td>
<td>sub</td>
<td>$0x4,%esp</td>
<td></td>
</tr>
<tr>
<td>08048391</td>
<td>e8 1a 00 00 00</td>
<td>call</td>
<td>80483b0 &lt;swap&gt;</td>
<td></td>
</tr>
<tr>
<td>08048396</td>
<td>83 c4 04</td>
<td>add</td>
<td>$0x4,%esp</td>
<td></td>
</tr>
<tr>
<td>08048399</td>
<td>31 c0</td>
<td>xor</td>
<td>%eax,%eax</td>
<td></td>
</tr>
<tr>
<td>0804839b</td>
<td>59</td>
<td>pop</td>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>0804839c</td>
<td>5d</td>
<td>pop</td>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>0804839d</td>
<td>8d 61 fc</td>
<td>lea</td>
<td>0xfffffffffc(%ecx),%esp</td>
<td></td>
</tr>
<tr>
<td>080483a0</td>
<td>c3</td>
<td>ret</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

0x8048396 + 0x1a = 0x80483b0
0: 8b 15 00 00 00 00 mov 0x0,%edx
2: R_386_32 buf
6: a1 04 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
### Executable After Relocation (.data)

**Disassembly of section .data:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>08049620 &lt;buf&gt;:</td>
<td></td>
</tr>
<tr>
<td>8049620</td>
<td>01 00 00 00 02 00 00 00</td>
</tr>
<tr>
<td>08049628 &lt;bufp0&gt;:</td>
<td></td>
</tr>
<tr>
<td>8049628</td>
<td>20 96 04 08</td>
</tr>
</tbody>
</table>
Program symbols are either strong or weak

- **Strong**: procedures and initialized globals
- **Weak**: uninitialized globals

```c
int foo=5;

int foo;

p1.c

p1() {
    }

p2.c

p2() {
    }
```
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 symbols, 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**

1. int x;  
   p1(); {}  
   p1(); {}  
   Link time error: two strong symbols (p1)

2. int x;  
   p1(); {}  
   p2(); {}  
   References to x will refer to the same uninitialized int. Is this what you really want?

3. int x;  
   int y;  
   p1(); {}  
   double x;  
   p2(); {}  
   Writes to x in p2 might overwrite y!  
   Evil!

4. int x=7;  
   int y=5;  
   p1(); {}  
   double x;  
   p2(); {}  
   Writes to x in p2 will overwrite y!  
   Nasty!

5. 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.
```c
#include "global.h"

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

```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;
}
```
#include "global.h"

int f() {
    return g+1;
}

-DINITIALIZE

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

```
unix> ar rs libc.a \n   atoi.o printf.o ... random.o
```

C standard library
**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 fputc.o fopen.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

- Translators (cpp, cc1, as)
- 
- main2.c vector.h
- 
- Archiver (ar)
  - libvector.a
  - libc.a
- 
- Relocatable object files
- 
- Linker (ld)
  - addvec.o
  - multvec.o
- 
- Static libraries
  - printf.o and any other modules called by printf.o
- 
- Fully linked executable object file
- 
- p2
- 
- Sungkyunkwan University
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

<table>
<thead>
<tr>
<th>ELF Header</th>
<th>0x00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
<td>0x10000000</td>
</tr>
<tr>
<td>Section header table</td>
<td>0x20000000</td>
</tr>
<tr>
<td>.init section</td>
<td>0x30000000</td>
</tr>
<tr>
<td>.text section</td>
<td>0x40000000</td>
</tr>
<tr>
<td>.rodata section</td>
<td>0x50000000</td>
</tr>
<tr>
<td>.data section</td>
<td>0x60000000</td>
</tr>
<tr>
<td>.bss section</td>
<td>0x70000000</td>
</tr>
<tr>
<td>.symtab</td>
<td>0x80000000</td>
</tr>
<tr>
<td>.debug</td>
<td>0x90000000</td>
</tr>
<tr>
<td>.line</td>
<td>0xA0000000</td>
</tr>
<tr>
<td>.strtab</td>
<td>0xB0000000</td>
</tr>
<tr>
<td>Section header table (required for relocatables)</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

#### Memory-happed region for shared libraries
- Run-time heap
  - Created by `malloc`

#### Kernel virtual memory
- User stack
  - Created at runtime

#### Memory outside 32-bit address space
- `brk`

#### Read/write segment
- `data`, `bss`

#### Read-only segment
- `init`, `text`, `rodata`

#### Unused

---

*Source: Sungkyunkwan University*
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
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

Translators (cpp, cc1, as)

main2.o

Linker (ld)

p2

Loader (execve)

Dynamic linker (ld-linux.so)

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

Relocatable object file

Partially linked executable object file

Fully linked executable in memory

Relocation and symbol table info

Code and data

Sungkyunkwan University
#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);
    }
/* 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;
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
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.

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

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

hello.c
```
#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;
}
#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
#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;
}
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

```
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
```
#ifndef 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 = dlset(RTLD_NEXT, "malloc");
        if ((error = dlerror()) != NULL) {
            fputs(error, stderr);
            exit(1);
        }
    }
    ptr = mallocp(size);
    printf("malloc(%d) = %p
", (int)size, ptr);
    return ptr;
}

mymalloc.c
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