LINKING

Spring 2014
Euiseong Seo
(euiseong@gmail.com)
Linking

Case study: Library interpositioning
main.c

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

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

swap.c

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

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

extern int buf[];

int *bufp0 = &buf[0];

static int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
Relocatable Object Files

- System code
  - .text
  - .data
- System data

Executable Object File

- Headers
  - System code
    - main()
    - swap()
  - More system code
    - System data
      - int buf[2]={1,2}
      - int *bufp0=&buf[0]
      - int *bufp1
      - static int *bufp1
      - .symtab
      - .debug

Even though private to swap, requires allocation in .bss
int buf[2] = {1,2};

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

Disassembly of section .data:

Source: objdump -r -d

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;
}
Disassembly of section .data:

00000000 <bufp0>:
  0: 00 00 00 00

0: R_386_32 buf

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;
}
```
Executable Before/After Relocation

```
0000000 <main>:

    e: 83 ec 04     sub    $0x4,%esp
   11: e8 fc ff ff ff  call  12 <main+0x12>
   12: R_386_PC32  swap
   16: 83 c4 04    add    $0x4,%esp

0x8048396 + 0x1a = 0x80483b0
```

```
08048380 <main>:

08048380:     8d 4c 24 04        lea    0x4(%esp),%ecx
08048384:     83 e4 f0           and    $0xfffffffff0,%esp
08048387:     ff 71 fc           pushl   0xfffffffffc(%ecx)
0804838a:     55                push   %ebp
0804838b:     89 e5             mov    %esp,%ebp
0804838d:     51                push   %ecx
0804838e:     83 ec 04           sub    $0x4,%esp
08048391:     e8 1a 00 00 00     call   80483b0 <swap>
08048396:     83 c4 04           add    $0x4,%esp
08048399:     31 c0             xor    %eax,%eax
0804839b:     59                pop    %ecx
0804839c:     5d                pop    %ebp
0804839d:     8d 61 fc           lea    0xfffffffffc(%ecx),%esp
080483a0:     c3                ret
```
### Executable After Relocation (.data)

Disassembly of section .data:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>08049620</td>
<td>&lt;buf&gt;</td>
</tr>
<tr>
<td>8049620</td>
<td>01 00 00 00 02 00 00 00</td>
</tr>
<tr>
<td>08049628</td>
<td>&lt;bufp0&gt;</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;
p1() {
}
```

```c
int foo;
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

<table>
<thead>
<tr>
<th>int x;</th>
<th>p1() {}</th>
<th>p1() {}</th>
<th><strong>Link time error: two strong symbols (p1)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>p1() {}</td>
<td>int x;</td>
<td>p2() {}</td>
</tr>
<tr>
<td>int x;</td>
<td>double x;</td>
<td>int x;</td>
<td>p2() {}</td>
</tr>
<tr>
<td>int x=7; int y=5;</td>
<td>double x;</td>
<td>p1() {}</td>
<td>p2() {}</td>
</tr>
<tr>
<td>int x=7;</td>
<td>int x;</td>
<td>p1() {}</td>
<td>p2() {}</td>
</tr>
</tbody>
</table>

**Nightmare scenario:** two identical weak structs, compiled by different compilers with different alignment rules.
c1.c

```c
#include "global.h"

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

c2.c

```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;
}
```

global.h

```c
#define INITIALIZE
int g = 23;
static int init = 1;
#else
int g;
static int init = 0;
#endif
```
Sungkyunkwan University

```c
#include "global.h"

int f() {
    return g+1;
}

#include causes C preprocessor to insert file verbatim
```

### global.h

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

### c1.c

```c
#include "global.h"

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
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**

- **Translator**
  - atoi.c
  - printf.c
  - random.c

- **Archiver (ar)**
  - atoi.o
  - printf.o
  - random.o

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

- **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             
fpcontrol.o           
fputc.o               
freopen.o             
fsprintf.o            
fsopen.o              
fsread.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)
- addvec.o
- multvec.o
- libvector.a
- libc.a

**Linker** (ld)
- main2.o
- addvec.o
- printf.o
- and any other modules called by printf.o

- **Relocatable object files**
- **Fully linked executable object file**
- **Static libraries**

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

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)

Kernel virtual memory

User stack (created at runtime)

Memory-mapped region for shared libraries

Run-time heap (created by malloc)

Read/write segment (.data, .bss)

Read-only segment (.init, .text, .rodata)

Unused

Memory outside 32-bit address space

%esp (stack pointer)

brk

Loaded from the executable file

Memory-mapped region for shared libraries

Loaded from the executable file

Run-time heap (created by malloc)

Read/write segment (.data, .bss)

Read-only segment (.init, .text, .rodata)

Unused

Memory outside 32-bit address space

%esp (stack pointer)

brk
Shared Libraries

Static libraries have the following disadvantages:

- Duplication in the stored executables (every function needs 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

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

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

**Linker**
- (ld)

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

**Reloca:tion and symbol table info**

**Code and data**

**Fully linked executable in memory**

**Partially linked executable object file**

**Relocatable object file**

- **main2.c**
- **vector.h**
- **main2.o**
- **libc.so**
- **libvector.so**
#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;  
}
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
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.
#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);  

```c
#include "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  
```
```c
#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
#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 = 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;
}
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 → __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