Linking

15-213 / 18-213: Introduction to Computer Systems
12\textsuperscript{th} Lecture, Oct. 3, 2013

\textbf{Instructors:}
Randy Bryant, Dave O’Hallaron, and Greg Kesden
Today

- Linking
- Case study: Library interpositioning
Example C Program

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;
}
```
Static Linking

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

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`)

![Diagram of program compilation and linking process]
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* (global variables and functions):
  - `void swap() {...} /* define symbol swap */`
  - `swap(); /* reference symbol swap */`
  - `int *xp = &x; /* define symbol xp, reference x */`

- Symbol definitions are stored in object file (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

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

```c
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;
}
```
Relocating Code and Data

Relocatable Object Files

- System code
- System data

<table>
<thead>
<tr>
<th>main.o</th>
<th>swap.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>swap()</td>
</tr>
<tr>
<td>int buf[2]={1,2}</td>
<td>int *bufp0=&amp;buf[0]</td>
</tr>
<tr>
<td>static int *bufp1</td>
<td></td>
</tr>
</tbody>
</table>

Executable Object File

0

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

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

Disassembly of section .data:

Source: objdump -j .data -d main.o

Disassembly of section .data:

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

Source: objdump -r -d main.o

Relocation Info (main)

00000000 <main>:
  0: 8d 4c 24 04          lea    0x4(%esp),%ecx
  4: 83 e4 f0            and    $0xfffffffff0,%esp
  7: ff 71 fc          pushl  0xfffffffffc(%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 ff     call   12 <main+0x12>
12: R_386_PC32  swap
 16: b8 00 00 00 00       mov    $0x0,%eax
 1b: 83 c4 04          add    $0x4,%esp
 1e:  59                pop    %ecx
 1f:  5d                pop    %ebp
 20: 8d 61 fc        lea    0xfffffffffc(%ecx),%esp
 23: c3                ret

Source: objdump –r –d main.o
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

00000000 <swap>:
    0:   55                     push   %ebp
    1:   89 e5                  mov    %esp,%ebp
    3:   53                     push   %ebx
    b:  00 00 00

        6:  R_386_32 .bss
        a:  R_386_32 buf
        e:  8b 0d 00 00 00 00 00     mov   0x0,%ecx

        10: R_386_32 bufp0
       14:  8b 19                  mov    (%ecx),%ebx
       16:  ba 04 00 00 00 00     mov   $0x4,%edx

       17: R_386_32 buf
       1b:  8b 02                  mov    (%edx),%eax
       1d:  89 01                  mov    %eax,(%ecx)
       1f:  89 1a                  mov    %ebx,(%edx)
       21:  5b                     pop    %ebx
       22:  5d                     pop    %ebp
       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;
}
Executable Before/After Relocation (.text)

00000000 <main>:

... e: 83 ec 04 sub $0x4,%esp
11: e8 fc ff ff ff call 12 <main+0x12>
16: b8 00 00 00 00 mov $0x0,%eax
...

Link time:
0x8048398 + (-4)
- 0x8048386 = 0xe

Runtime:
0x804838a + 0xe
= 0x8048398

08048374 <main>:

8048374: 8d 4c 24 04 lea 0x4(%esp),%ecx
8048378: 83 e4 f0 and $0xfffffffff0,%esp
804837b: ff 71 fc pushl 0xfffffffffc(%ecx)
804837e: 55 push %ebp
8048382: 83 ec 04 sub $0x4,%esp
8048385: e8 0e 00 00 00 call 0x8048398 <swap>
804838a: b8 00 00 00 00 mov $0x0,%eax
804838f: 83 c4 04 add $0x4,%esp
8048392: 59 pop %ecx
8048393: 5d pop %ebp
8048394: 8d 61 fc lea 0xfffffffffc(%ecx),%esp
8048397: c3 ret
Before relocation

After relocation

00000000 <swap>:
...
4: c7 05 00 00 00 00 04 movl $0x4,0x0
b: 00 00 00

6: R_386_32 .bss
a: R_386_32 buf
e: 8b 0d 00 00 00 00 00 mov 0x0,%ecx

10: R_386_32 bufp0
14: 8b 19 mov (%ecx),%ebx
16: ba 04 00 00 00 mov $0x4,%edx

17: R_386_32 buf

08048398 <swap>:
8048398: 55 push %ebp
8048399: 89 e5 mov %esp,%ebp
804839b: 53 push %ebx
804839c: c7 05 14 96 04 08 04 movl $0x8049604,0x8049614
80483a3: 96 04 08
80483a6: 8b 0d 08 96 04 08 mov 0x8049608,%ecx
80483ac: 8b 19 mov (%ecx),%ebx
80483ae: ba 04 96 04 08 mov $0x8049604,%edx
80483b3: 8b 02 mov (%edx),%eax
80483b5: 89 01 mov %eax,(%ecx)
80483b7: 89 1a mov %ebx,(%edx)
80483b9: 5b pop %ebx
80483ba: 5d pop %ebp
80483bb: c3 ret
Executable After Relocation (.data)

Disassembly of section .data:
08049600 <buf>:
  8049600: 01 00 00 00 02 00 00 00

08049608 <bufp0>:
  8049608: 00 96 04 08
Strong and Weak Symbols

- 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 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

**Link time error: two strong symbols (p1)**

```c
int x;
p1() {}
```

```c
p1() {}
```

**References to x will refer to the same uninitialized int. Is this what you really want?**

```c
int x;
p1() {}
```

```c
int x;
p2() {}
```

**Writes to x in p2 might overwrite y! Evil!**

```c
int x;
int y;
p1() {}
```

```c
double x;
p2() {}
```

**Writes to x in p2 will overwrite y! Nasty!**

```c
int x=7;
int y=5;
p1() {}
```

```c
double x;
p2() {}
```

**References to x will refer to the same initialized variable.**

```c
int x=7;
p1() {}
```

```c
int x;
p2() {}
```

**Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.**
Role of .h Files

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

c1.c

```c
#include "global.h"

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

```c
-DINITIALIZE

int g = 23;
static int init = 1;
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;
static int init = 0;
int f() {
    return g+1;
}
```

#include causes C preprocessor to insert file verbatim (Use gcc –E to view result)
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

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

```
unix> ar rs libc.a \ 
    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 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 e_asinl.o ...


Linking with Static Libraries

- **Translators** (cpp, cc1, as)
- **Archiver** (ar)
- **Linker** (ld)

- Relocatable object files:
  - `main2.o`
  - `addvec.o`

- **Static libraries**:
  - `libc.a`
  - `libvector.a`

- Other modules called by `printf.o`:
  - `multvec.o`
  - `addvec.o`

- Fully linked executable object file:
  - `p2`

- Files:
  - `main2.c`    `vector.h`    `printf.o`

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.

```bash
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

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
<td>Program header table (required for executables)</td>
</tr>
<tr>
<td>.init section</td>
<td>.text section</td>
</tr>
<tr>
<td>.rodata section</td>
<td>.data section</td>
</tr>
<tr>
<td>.bss section</td>
<td>.symtab</td>
</tr>
<tr>
<td>.debug</td>
<td>.line</td>
</tr>
<tr>
<td>.strtab</td>
<td>Section header table (required for relocatables)</td>
</tr>
</tbody>
</table>

### Memory Layout

- **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**
  - brk
  - %esp (stack pointer)

- **Loaded from the executable file**
  - 0x08048000
  - 0xf7e9ddc0
  - 0x100000000

- **Memory-mapped region for shared libraries**
  - 0x08048000

- **Run-time heap**
  - 0xf7e9ddc0

- **User stack**
  - 0x100000000

- **Kernel virtual memory**
  - 0x0
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 `dllopen()` 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

libc.so
libvector.so

Relocation and symbol table info

Relocatable object file

Partially linked executable object file

Fully linked executable in memory

Code and data
#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 Summary

- Linking is a technique that allows programs to be constructed from multiple object files.

- Linking can happen at different times in a program’s lifetime:
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

- Understanding linking can help you avoid nasty errors and make you a better programmer.
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

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

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

- **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

```c
#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;
}
```
Compile-time Interpositioning

```c
#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
```
#ifndef 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

- 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
```
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

The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., `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

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