Linking

15-213 / 18-213: Introduction to Computer Systems
12\textsuperscript{th} Lecture, Oct. 2, 2014

Instructors:
Greg Ganger, Greg Kesden, and Dave O’Hallaron
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 -o p main.c swap.c`
  - `unix> ./p`

![Diagram of static linking process]

- **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
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 assembler) 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.

Let’s look at these two steps in more detail....
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 (`.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-\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 global variables defined with the \texttt{static} attribute.
  - \texttt{Local linker symbols are not local program variables}
Step 1: Symbol Resolution

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;
}

Global

Question: How do linkers resolve duplicate names?
Strong and Weak Symbols

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

```
int foo=5;
p1();
}

int foo;
p2();
}
```

```
p1.c
```

```
p2.c
```
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)

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

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

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

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

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.
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
Step 2: Relocation

Relocatable Object Files

| System code | .text |
| System data | .data |

| main.o       | .text |
|             | .data |
| main()       | .text |
| int buf[2]={1,2} | .data |

| swap.o       | .text |
|             | .data |
| swap()       | .text |
| int *bufp0=&buf[0] | .data |
| static int *bufp1 | .bss |

Executable Object File

<table>
<thead>
<tr>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>System code</td>
</tr>
<tr>
<td>main()</td>
</tr>
<tr>
<td>swap()</td>
</tr>
</tbody>
</table>

More system code

| System data |
| int buf[2]={1,2} |
| int *bufp0=&buf[0] |
| 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:

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

Source: objdump –r –d main.o

Source: objdump –j .data –d main.o
Executable Before/After Relocation (.text)

00000000 <main>:
...
  
  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
...

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 0xfffffffff0(%ecx)
804837e:  55  push %ebp
804837f:  89 e5  mov %esp,%ebp
8048381:  51  push %ecx
8048382:  83 ec 04  sub $0x4,%esp
8048385:  e8 0e 00 00 00  call 8048398 <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 0xfffffffff0(%ecx),%esp
8048397:  c3  ret
## Loading Executable Object Files

<table>
<thead>
<tr>
<th>Executable Object File</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
<td>0xbffffffff</td>
</tr>
<tr>
<td>Program header table (required for executables)</td>
<td>0xf7e9ddc0</td>
</tr>
<tr>
<td>.init section</td>
<td>0x08048000</td>
</tr>
<tr>
<td>.text section</td>
<td>0x0</td>
</tr>
<tr>
<td>.rodata section</td>
<td>Unused</td>
</tr>
<tr>
<td>.data section</td>
<td>0</td>
</tr>
<tr>
<td>.bss section</td>
<td>0x08048000</td>
</tr>
<tr>
<td>.symtab</td>
<td>0xbffffffff</td>
</tr>
<tr>
<td>.debug</td>
<td>0xf7e9ddc0</td>
</tr>
<tr>
<td>.line</td>
<td>0x08048000</td>
</tr>
<tr>
<td>.strtab</td>
<td>0xbffffffff</td>
</tr>
<tr>
<td>Section header table (required for relocatables)</td>
<td>0</td>
</tr>
</tbody>
</table>

**Memory Invisible to User Code**

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

**Loaded from the Executable File**

- Read-only code segment (.init, .text, .rodata)
- Read/write data segment (.data, .bss)
- Run-time heap (created by malloc)
- Memory-mapped region for shared libraries
- User stack (created at runtime)
- Kernel virtual memory

**Memory-Mapped Region for Shared Libraries**

- Memory-mapped region for shared libraries

**Run-time Heap**

- Run-time heap (created by malloc)

**Memory Inaccessible to User Code**

- Memory invisible to user code

**User Stack**

- User stack (created at runtime)

**Kernel Virtual Memory**

- Kernel virtual memory
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
Old-fashioned 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 (ar) 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, ...)

```bash
% ar -t /usr/lib/libc.a | sort
... fork.o ...
fprintf.o fpu_control.o fputc.o freopen.o fscanf.o fseek.o fstab.o ...
```

```bash
% 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_asin.f.o e_asinf.o e_asinl.o ...
```
Linking with Static Libraries

Main2.c vector.h

Translators (cpp, cc1, as)

Archiver (ar)

main2.o, printf.o

Linker (ld)

addvec.o, multvec.o

libvector.a, libc.a

printf.o and any other modules called by printf.o

Relocatable object files

Fully linked executable object file

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.

```shell
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'
```
Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs 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

Translators (cpp, cc1, as)

main2.c  vector.h

Translations

main2.o

Linker (ld)

p2

Relocatable object file

Relocation and symbol table info

Loader (execve)

libc.so libvector.so

Code and data

Dynamic linker (ld-linux.so)

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

libc.so libvector.so
Dynamic Linking at Run-time

```c
#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 = dlload("./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;
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)
  - Behind the scenes encryption

- **Debugging**
  - In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
  - Code in the SPDY networking stack was writing to the wrong location
  - Solved by intercepting calls to Posix write functions (write, writev, pwrite)

Source: Facebook engineering blog post at https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/
Some Interpositioning Applications

- 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 <malloc.h>

int main()
{
    int *p = malloc(32);
    free(p);
    return(0);
}
```

- **Goal:** trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and 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
#define COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void *mymalloc(size_t size)
{
    void *ptr = malloc(size);
    printf("malloc(%d)=%p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void *ptr)
{
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c
Compile-time Interpositioning

```c
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc
malloc(32)=0x1edc010
free(0x1edc010)
linux>
```
Link-time Interpositioning

```c
#define LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

mymalloc.c
The "-Wl" flag passes argument to linker, replacing each comma with a space.

The "--wrap,malloc" arg instructs linker 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 intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl
int.o mymalloc.o
linux> make runl
./intl
malloc(32) = 0x1aa0010
free(0x1aa0010)
linux>
```
```c
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Addr of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```
/* free wrapper function */

void free(void *ptr) {
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Addr of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
Load/Run-time Interpositioning

- The LD_PRELOAD environment variable tells the dynamic linker to resolve unresolved refs (e.g., to malloc) by looking in mymalloc.so first.

```bash
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr)
malloc(32) = 0xe60010
free(0xe60010)
linux>
```
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 \(\rightarrow\) \texttt{\_\_wrap\_malloc}
    - \texttt{\_\_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
Supplemental slides
extern int buf[];

int
  *bufp0 = &buf[0];

static int *bufp1;

void swap()
{
  int temp;

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

```c
swap.c

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
  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 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)

```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: R_386_32 buf
```
Before relocation

A: R_386_32 .bss
4: c7 05 00 00 00 00 04 movl $0x4,0x0
6: R_386_32 buf
B: R_386_32 buf

After relocation

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
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

#include "global.h"

int f() {
    return g+1;
}

-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;
}

global.h

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

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