Linking

15-213: Introduction to Computer Systems
13th Lecture, Oct. 13, 2015

Instructors:
Randal E. Bryant and David R. O’Hallaron
Today

- Linking
- Case study: Library interpositioning
Example C Program

\begin{align*}
\textbf{main.c} \\
\textbf{sum.c}
\end{align*}

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}

int sum(int *a, int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

* main.c
* sum.c
Static Linking

- Programs are translated and linked using a compiler driver:
  - \texttt{linux\> gcc -Og -o prog main.c sum.c}
  - \texttt{linux\> ./prog}

Source files

\begin{itemize}
  \item main.c
  \item sum.c
\end{itemize}

Translators
\texttt{(cpp, cc1, as)}

\begin{itemize}
  \item main.o
  \item sum.o
\end{itemize}

Fully linked executable object file
(contains code and data for all functions defined in \texttt{main.c} and \texttt{sum.c})

Separately compiled relocatable object files
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.

  - During symbol resolution step, the 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

---

### ELF header

- Segment header table (required for executables)
  - `.text` section
  - `.rodata` section
  - `.data` section
  - `.bss` section
  - `.symtab` section
  - `.rel.text` section
  - `.rel.data` section
  - `.debug` section

### Section header table
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**
Step 1: Symbol Resolution

int sum(int *a, int n);
int array[2] = {1, 2};
int main()
{
    int val = sum(array, 2);
    return val;
}

main.c

int sum(int *a, int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}

sum.c
Local Symbols

- Local non-static C variables vs. local static C variables
  - local non-static C variables: stored on the stack
  - local static C variables: stored in either `.bss`, or `.data`

```c
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x.1` and `x.2`. 
How Linker Resolves Duplicate Symbol Definitions

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

```
int foo=5;
p1() {
}
p2.c
int foo;
p2() {
}
p1.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 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>Link time error: two strong symbols (p1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td>p1() {}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int x;</th>
<th>int x;</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1() {}</td>
<td>p2() {}</td>
</tr>
<tr>
<td>References to x will refer to the same uninitialized int. Is this what you really want?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int x;</th>
<th>double x;</th>
</tr>
</thead>
<tbody>
<tr>
<td>int y;</td>
<td>p2() {}</td>
</tr>
<tr>
<td>Writes to x in p2 might overwrite y! Evil!</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int x=7;</th>
<th>double x;</th>
</tr>
</thead>
<tbody>
<tr>
<td>int y=5;</td>
<td>p2() {}</td>
</tr>
<tr>
<td>Writes to x in p2 will overwrite y! Nasty!</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int x=7;</th>
<th>int x;</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1() {}</td>
<td>p2() {}</td>
</tr>
<tr>
<td>References to x will refer to the same initialized variable.</td>
<td></td>
</tr>
</tbody>
</table>

**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 reference an external global variable
Step 2: Relocation

Relocatable Object Files

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

- main.o
  - main()
  - int array[2] = {1,2}

- sum.o
  - sum()
  - int array[2] = {1,2}

Executable Object File

| Headers |
| System code |
| main() |
| swap() |
| More system code |
| System data |
| int array[2] = {1,2} |
| .symtab |
| .debug |
Relocation Entries

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

int main()
{
    int val = sum(array, 2);
    return val;
}
```

```
0000000000000000 <main>:
  0:   48 83 ec 08             sub    $0x8,%rsp
  4:   be 02 00 00 00           mov    $0x2,%esi
  9:   bf 00 00 00 00           mov    $0x0,%edi
      # %edi = &array
     a: R_X86_64_32 array        # Relocation entry

     e: e8 00 00 00 00           callq   13 <main+0x13>    # sum()
     f: R_X86_64_PC32 sum-0x4   # Relocation entry

 13:   48 83 c4 08             add    $0x8,%rsp
 17:   c3                       retq
```

Source: objdump -r -d main.o
Using PC-relative addressing for sum(): $0x4004e8 = 0x4004e3 + 0x5$

Source: objdump -dx prog
## 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>Memory-mapped region for shared libraries</td>
</tr>
<tr>
<td>.text section</td>
<td>Run-time heap (created by malloc)</td>
</tr>
<tr>
<td>.rodata section</td>
<td>Read/write data segment (.data, .bss)</td>
</tr>
<tr>
<td>.data section</td>
<td>Read-only code segment (.init, .text, .rodata)</td>
</tr>
<tr>
<td>.bss section</td>
<td>Unused</td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
</tr>
<tr>
<td>.line</td>
<td></td>
</tr>
<tr>
<td>.strtab</td>
<td></td>
</tr>
<tr>
<td>Section header table</td>
<td></td>
</tr>
</tbody>
</table>

### Kernel virtual memory

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

### Memory invisible to user code

- %rsp (stack pointer)
- brk

### Memory-mapped region for shared libraries

- Loaded from the executable file

### User stack

- (created at runtime)

### Run-time heap

- (created by malloc)

### Unused

- 0x400000

---

Bryant and O'Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition
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 allows incremental updates
- Recompile function that changes and replace .o file in archive.
Commonly Used Libraries

**libc.a (the C standard library)**
- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

**libm.a (the C math library)**
- 2 MB archive of 444 object files.
- Floating point math (sin, cos, tan, log, exp, sqrt, ...)

```
% ar –t libc.a | sort
...
  fork.o
  ...
  fprintf.o
  fpu_control.o
  fputc.o
  freopen.o
  fscanf.o
  fseek.o
  fstab.o
...

% ar –t 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

```c
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);
    return 0;
}

void addvec(int *x, int *y, int *z, int n) {
    int i;
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}

void multvec(int *x, int *y, int *z, int n) {
    int i;
    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

- `addvec.c`
- `multvec.c`
- `main2.c`
- `libvector.a`
Linking with Static Libraries

- Translators (cpp, cc1, as)
  - main2.c, vector.h
- Archiver (ar)
  - main2.o, addvec.o
  - libvector.a, libc.a
- Linker (ld)
  - prog2c
- Static libraries
  - printf.o and any other modules called by printf.o
- Relocatable object files
  - main2.o, addvec.o
- Fully linked executable object file
  - "c" for "compile-time"
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'
```
Modern Solution: 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

```
main2.c  vector.h

Translators (cpp, cc1, as)

main2.o

Linker (ld)

prog21

Loader (execve)

Dynamic linker (ld-linux.so)

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

libc.so
libvector.so

Relocatable object file
Partially linked executable object file
Fully linked executable in memory

Relocation and symbol table info
Code and data
```
Dynamic Linking at Run-time

```c
#include <stdio.h>
#include <stdlib.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 library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s
", 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;
}  
dll.c
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);
}

int.c
```

- **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.
Compile-time Interpositioning

```c
#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
```
Compile-time Interpositioning

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

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

```bash
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
#ifndef 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
```
## Link-time Interpositioning

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>linux&gt; make intl</code></td>
<td>Compile <code>mymalloc.c</code> with link-time interposition.</td>
</tr>
<tr>
<td><code>gcc -Wall -DLINKTIME -c mymalloc.c</code></td>
<td></td>
</tr>
<tr>
<td><code>gcc -Wall -c int.c</code></td>
<td></td>
</tr>
<tr>
<td><code>gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o</code></td>
<td>Compile <code>int.c</code> and <code>mymalloc.c</code> with the linker flag.</td>
</tr>
<tr>
<td><code>linux&gt; make runl</code></td>
<td>Compile and link the program.</td>
</tr>
<tr>
<td><code>. /intl</code></td>
<td>Run the compiled program.</td>
</tr>
<tr>
<td><code>malloc(32) = 0x1aa0010</code></td>
<td>Display the result of calling <code>malloc(32)</code></td>
</tr>
<tr>
<td><code>free(0x1aa0010)</code></td>
<td>Display the result of calling <code>free(0x1aa0010)</code></td>
</tr>
<tr>
<td><code>linux&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

- 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`
```c
#define 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"); /* Get 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;
}
```

mymalloc.c
```c
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if (error = dlerror()) !- NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
```

mymalloc.c
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 → __wrap_malloc
    - __real_malloc → malloc

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