15-213
“The course that gives CMU its Zip!”

Linking
October 14, 2004

Topics
- Static linking
- Dynamic linking
- Case study: Library interpositioning
Example C Program

```c
main.c
int buf[2] = {1, 2};
int main()
{
    swap();
    return 0;
}
```

```c
swap.c
extern int buf[];
int *bufp0 = &buf[0];
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

```
main.c  swap.c
   |     |
   v     v
Translators (cpp, ccl, as) Translators (cpp, ccl, as)
   |     |
   v     v
main.o  swap.o
   |     |
   v     v
Linker (ld)
   |     |
   v     v
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)
Why Linkers?

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

Efficiency

- Time:
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of 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 Does a Linker Do?

Symbol resolution

- Programs define and reference *symbols* (variables and functions):
  - code: `a();`  /* reference to symbol `a` */
  - data: `int *xp=&x;`  /* reference to symbol `x` */
- Symbol definitions are stored (by compiler) in *symbol table*:
  - Each entry includes name, size, and location of object.
- Linker associates each symbol reference with exactly one symbol definition.

Relocation

- Merges separate code and data sections into single sections
- Relocates symbols from their relative locations in the `.o` files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions.
Object Files (Modules)

Relocatable object file (.o file)
- Contains code and data in a form that can be combined with other relocatable object files at compile time to form an executable.

Executable object 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 runtime.
- 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
  - Shared object files (.so)

Generic name: ELF binaries
ELF Object File Format

Elf header
- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

Program header table
- Page size, virtual addresses memory segments (sections), segment sizes.

.text section
- Code

.data section
- Initialized (static) data

.bss section
- Uninitialized (static) data
  - “Block Started by Symbol”
  - “Better Save Space”
- Has section header but occupies no space

<table>
<thead>
<tr>
<th>ELF header</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
<td></td>
</tr>
<tr>
<td>.text section</td>
<td></td>
</tr>
<tr>
<td>.data section</td>
<td></td>
</tr>
<tr>
<td>.bss section</td>
<td></td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
</tr>
<tr>
<td>.rel.txt</td>
<td></td>
</tr>
<tr>
<td>.rel.data</td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
</tr>
</tbody>
</table>

Section header table
(required for relocatables)
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)
Linker Symbols

Global symbols
- Symbols defined by module $m$ that can be referenced by other modules.
- Ex: 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 $m$.
- Ex: C functions and variables that are defined with the static attribute.

Key Point: Local linker symbols are not local program variables
Resolving Symbols

**Def of global symbol buf**

```c
int buf[2] = {1,2};
int main()
{
    swap();
    return 0;
}
```

**Ref to external symbol buf**

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

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

**Ref to external symbol buf**

**Def of local symbol bufp0**

**Linker knows nothing of temp**
Relocating Symbol Definitions and References

Relocatable Object Files
- system code
  - .text
  - .data
- system data
- main()
  - .text
  - .data
  - int e = 7
- a()
  - .text
  - .data
  - int *ep = &e
  - int x = 15
  - int y
- a.o

Executable Object File
- 0
- headers
- system code
- main()
- a()
- more system code
- system data
- int e = 7
- int *ep = &e
- int x = 15
- uninitialized data
- .symtab
- .debug
- .text
- .data
- .bss
main.o Relocation Info

```c
int buf[2] = {1,2};
int main()
{
    swap();
    return 0;
}
```

```
00000000 <main>:
0:  55      push %ebp
1:  89 e5   mov %esp,%ebp
3:  83 ec 08 sub $0x8,%esp
6:  e8 fc ff ff ff call 7 <main+0x7>
7:  R_386_PC32 swap
b:  31 c0   xor %eax,%eax
d:  89 ec   mov %ebp,%esp
f:  5d      pop %ebp
10: c3      ret
```

Disassembly of section .data:
```
00000000 <buf>:
0:  01 00 00 00 02 00 00 00
```

Source: objdump
swap.o Relocation Info (.text)

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

void swap()
{
    int temp;

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

Disassembly of section .text:

00000000 <swap>:
  0: 55         push  %ebp
  1: 8b 15 00 00 00 00     mov  0x0,%edx
  3: R_386_32 bufp0
  7: a1 00 00 00          mov  0x4,%eax
  8: R_386_32 buf
  c: 89 e5          mov  %esp,%ebp
  e: c7 05 00 00 00 00 00 04 movl $0x4,0x0
 15: 00 00 00
 10: R_386_32 bufp1
 14: R_386_32 buf
 18: 89 ec          mov  %ebp,%esp
 1a: 8b 0a         mov  (%edx),%ecx
 1c: 89 02          mov  %eax,(%edx)
 1e: a1 00 00 00 00     mov  0x0,%eax
 1f: R_386_32 bufp1
 23: 89 08          mov  %ecx,(%eax)
 25: 5d           pop  %ebp
 26: c3            ret
extern int buf[];

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

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
Executable After Relocation (.text)

08483b4 <main>:
  80483b4:   55 push %ebp
  80483b5:   89 e5 mov %esp,%ebp
  80483b7:   83 ec 08 sub $0x8,%esp
  80483ba:   e8 09 00 00 00 call 80483c8 <swap>
  80483bf:   31 c0 xor %eax,%eax
  80483c1:   89 ec mov %ebp,%esp
  80483c3:   5d pop %ebp
  80483c4:   c3 ret

08483c8 <swap>:
  80483c8:   55 push %ebp
  80483c9:   8b 15 5c 94 04 08 mov 0x804945c,%edx
  80483cf:   a1 58 94 04 08 mov 0x8049458,%eax
  80483d4:   89 e5 mov %esp,%ebp
  80483d6:   c7 05 48 95 04 08 58 movl $0x8049458,0x8049548
  80483dd:   94 04 08
  80483e0:   89 ec mov %ebp,%esp
  80483e2:   8b 0a mov (%edx),%ecx
  80483e4:   89 02 mov %eax,(%edx)
  80483e6:   a1 48 95 04 08 mov 0x8049548,%eax
  80483eb:   89 08 mov %ecx,(%eax)
  80483ed:   5d pop %ebp
  80483ee:   c3 ret
Executable After Relocation (.data)

Disassembly of section .data:

08049454 <buf>:
  8049454: 01 00 00 00 02 00 00 00

0804945c <bufp0>:
  804945c: 54 94 04 08
Strong and Weak Symbols

Program symbols are either strong or weak

- **strong**: procedures and initialized globals
- **weak**: uninitialized globals

```c
#include <stdio.h>

int main() {
    int foo = 5;
    return 0;
}
```
Linker’s Symbol Rules

Rule 1. A strong symbol can only appear once.

Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
- references to the weak symbol resolve to the strong symbol.

Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.
Linker Puzzles

int x;
p1() {}

int x;
p1() {}

Link time error: two strong symbols (p1)

int x;
p1() {}

int x;
p2() {}

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

double x;
p1() {}

double x;
p2() {}

Writes to x in p2 might overwrite y! Evil!

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

double x;
p2() {}

Writes to x in p2 will overwrite y! Nasty!

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.
Packaging Commonly 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 in 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 (.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 into executable.
Creating Static Libraries

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

Archiver (ar)

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

C standard library
Commonly Used Libraries

**libc.a (the C standard library)**
- 8 MB archive of 900 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 226 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, ...)

```
% ar -t /usr/lib/libc.a | sort
...
fork.o
...
fprintf.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstat.o
...
```
```
% ar -t /usr/lib/libm.a | sort
...
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acosl.o
e_acosl1.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

Static libraries

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

Relocatable object files

main2.o

Linker (ld)

addvec.o

Fully linked executable object file

p2
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> gcc -L. libtest.o -lmime
bash> gcc -L. -lmime 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>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF header</td>
<td>0</td>
</tr>
<tr>
<td>Program header table</td>
<td>0xc0000000</td>
</tr>
<tr>
<td>(required for executables)</td>
<td></td>
</tr>
<tr>
<td>.text section</td>
<td>0x40000000</td>
</tr>
<tr>
<td>.data section</td>
<td></td>
</tr>
<tr>
<td>.bss section</td>
<td></td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
</tr>
<tr>
<td>.rel.text</td>
<td></td>
</tr>
<tr>
<td>.rel.data</td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
</tr>
<tr>
<td>Section header table</td>
<td>0x08048000</td>
</tr>
<tr>
<td>(required for relocatables)</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 segment
(.data, .bss)

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

Unused

Memory invisible to user code

%esp
(stack pointer)

brk

Loaded from the executable file

- 26 -
Shared Libraries

Static libraries have the following disadvantages:

- Potential for duplicating lots of common code in the executable files on a filesystem.
  - e.g., every C program needs the standard C library
- Potential for duplicating lots of code in the virtual memory space of many processes.
- 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*
- 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 Unix, this is done by calls to the `dlopen()` interface.
  - 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)

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

Relocatable
object file

```
libc.so
libvector.so
```

Main2.o

```
Relocation and symbol
table info
```

Linker (ld)

```
p2
```

Partially linked
executable object file

```
Loader
(execlve)
```

```
libc.so
libvector.so
```

```
Code and data
```

Dynamic linker (ld-linux.so)

Fully linked
executable
in memory
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 = dlopen("./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() it 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;
}
Some Interpositioning Applications

Security

- Confinement (sandboxing)
  - Autolab interposes calls to libc functions.
  - Running `/bin/xterm` from student code on server a no-no.

- 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 malloc traces
Case Study: Library Interpositioning

Library interpositioning is a 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 linked to form an executable object file

- **load/run time**
  - When an executable object file is loaded into memory, dynamically linked, and then executed.
Example: `malloc()` Statistics

Find out how much memory is allocated by a function

```c
void *malloc(size_t size) {
    static void *(*fp)(size_t) = 0;
    void *mp;
    char *errorstr;

    if (!fp) { /* get a pointer to the real malloc() */
        fp = dlsym(RTLD_NEXT, "malloc");
        if ((errorstr = dlerror()) != NULL) {
            fprintf(stderr, "%s(): %s\n", fname, errorstr);
            exit(EXIT_FAILURE);
        }
    }

    mp = fp(size);
    if (!mp) { /* report malloc failure */
    }

    mem_used += size;

    return mp;
}
```