Linking

Alan L. Cox
alc@rice.edu

Some slides adapted from CMU 15.213 slides
x86-64 Assembly

Brief overview last time

- Lecture notes available on the course web page and x86-64 assembly overview in the textbook

You will not write any x86-64 assembly

- Need to be able to recognize/understand code fragments
- Need to be able to correlate assembly code to source code

More assembly examples today
Objectives

Be able to do your homework assignment on linking 😊

Understand how C type attributes (e.g. static, extern) control memory allocation for variables

Be able to recognize some of the pitfalls when developing modular programs

Appreciate how programs can optimize for efficiency, modularity, evolvability
Example Program (2 .c files)

/* main.c */
void swap(void);
int buf[2] = {1, 2};

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

/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;
void swap(void)
{
    int temp;
    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
An Analogy for Linking
Linking

Linking: collecting and combining various pieces of code and data into a single file that can be loaded into memory and executed

Why learn about linking?

- Make you a better jigsaw puzzle solver!
- It will help you build large programs
- It will help you avoid dangerous program errors
- It will help you understand how language scoping rules are implemented
- It will help you understand other important systems concepts (that are covered later in the class)
- It will enable you to exploit shared libraries
Compilation

UNIX% gcc -v -O -g -o p main.c swap.c

Compiler: .c C source code to .s assembly code
Assembler: .s assembly code to .o relocatable object code
Linker: .o to executable
Compilation

UNIX% gcc -O -g -o p main.c swap.c

C source code

main.c

cc1

main.s

as

main.o

swap.c

cc1

swap.s

as

swap.o

Linking step

Id (collect2)

ELF Format Files

Executable

p
Order & existence of segments is arbitrary, except ELF header must be present and first

ELF (Executable Linkable Format)

| 0 | ELF header          |
|   | Program header table |
|   | .text               |
|   | .rodata             |
|   | .data               |
|   | .bss                |
|   | .symtab             |
|   | .rel.text           |
|   | .rel.data           |
|   | .rel.data           |
|   | .debug              |
|   | ...                 |
|   | Section header table |

Order & existence of segments is arbitrary, except ELF header must be present and first
ELF Header

Basic description of file contents:

- File format identifier
- Endianness
- Alignment requirement for other sections
- Location of other sections
- Code’s starting address
- ...

<table>
<thead>
<tr>
<th>0</th>
<th>ELF header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td></td>
</tr>
<tr>
<td>.rodata</td>
<td></td>
</tr>
<tr>
<td>.data</td>
<td></td>
</tr>
<tr>
<td>.bss</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>...</td>
<td></td>
</tr>
</tbody>
</table>

Section header table
Program and Section Headers

Info about other sections necessary for loading
- Required for executables & libraries

Info about other sections necessary for linking
- Required for relocatables

Diagram:
- ELF header
- Program header table
  - .text
  - .rodata
  - .data
  - .bss
  - .symtab
  - .rel.text
  - .rel.data
  - .debug
  - ...
- Section header table
Text Section

Machine instruction code
  * read-only

Diagram of ELF header and section headers:

0

ELF header

Program header header table

.text

.rodata

.data

.bss

.symtab

.rel.text

.rel.data

.debug

...

Section header table
Data Sections

Static data
- initialized, read-only
- initialized, read/write
- uninitialized, read/write (BSS = “Block Started by Symbol” pseudo-op for IBM 704)

Initialized
- Initial values in ELF file

Uninitialized
- Only total size in ELF file

Writable distinction enforced at run-time
- Why? Protection; sharing
- How? Virtual memory
## Relocation Information

Describes where and how symbols are used

- A list of locations in the `.text` section that will need to be modified when the linker combines this object file with others
- Relocation information for any global variables that are referenced or defined by the module
- Allows object files to be easily relocated

<table>
<thead>
<tr>
<th>ELF header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
</tr>
<tr>
<td><code>.text</code></td>
</tr>
<tr>
<td><code>.rodata</code></td>
</tr>
<tr>
<td><code>.data</code></td>
</tr>
<tr>
<td><code>.bss</code></td>
</tr>
<tr>
<td><code>.symtab</code></td>
</tr>
<tr>
<td><code>.rel.text</code></td>
</tr>
<tr>
<td><code>.rel.data</code></td>
</tr>
<tr>
<td><code>.debug</code></td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

| Section header table |
Debug Section

Relates source code to the object code within the ELF file
Other Sections

Other kinds of sections also supported, including:

- Other debugging info
- Version control info
- Dynamic linking info
- C++ initializing & finalizing code

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

Section header table
Symbol Table

Describes where global variables and functions are defined

- Present in all relocatable ELF files

```c
/* main.c */
void swap(void);
int buf[2] = {1, 2};

int main(void)
{
    swap();
    return (0);
}
```
Linker Symbol Classification

Global symbols
- Symbols defined by module $m$ that can be referenced by other modules
- C: non-static functions & global variables

External symbols
- Symbols referenced by module $m$ but defined by some other module
- C: extern functions & variables

Local symbols
- Symbols that are defined and referenced exclusively by module $m$
- C: static functions & variables

Local linker symbols ≠ local function variables!
/* main.c */
void swap(void);
int buf[2] = {1, 2};

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

/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

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

Definition of global symbols bufp0 and bufp1 (even though not used outside file)

Reference to external symbol swap

Reference to external symbol buf

Linker knows nothing about local variables
What’s missing?

* swap - where is it?

buf is a C type object to the text set 0 offset of (UND) 1 (.text)

use readelf -S to see sections

UNIX% gcc -O -c main.c
UNIX% readelf -s main.o

Symbol table '.symtab' contains 11 entries:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>00000000000000000000</td>
<td>19</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>1</td>
<td>main</td>
</tr>
<tr>
<td>9</td>
<td>00000000000000000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td>swap</td>
</tr>
<tr>
<td>10</td>
<td>00000000000000000000</td>
<td>8</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>3</td>
<td>buf</td>
</tr>
</tbody>
</table>
## Linker Symbols

```c
/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void swap(void)
{
    int temp;

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

### What’s missing?
- **buf** - where is it?

### Symbol table `.symtab' contains 12 entries:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0000000000000000</td>
<td>38</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>1</td>
<td>swap</td>
</tr>
<tr>
<td>9</td>
<td>0000000000000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td>buf</td>
</tr>
<tr>
<td>10</td>
<td>0000000000000008</td>
<td>8</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>COM</td>
<td>bufp1</td>
</tr>
<tr>
<td>11</td>
<td>0000000000000000</td>
<td>8</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>3</td>
<td>bufp0</td>
</tr>
</tbody>
</table>

*bufp1* is a 8-byte initialized (COMMON) object with an alignment requirement.
Linking Steps

Symbol Resolution

- Determine where symbols are located and what size data/code they refer to

Relocation

- Combine modules, relocate code/data, and fix symbol references based on new locations
Problem: Undefined Symbols

Missing symbols are not compiler errors

- May be defined in another file
- Compiler just inserts an undefined entry in the symbol table

During linking, any undefined symbols that cannot be resolved cause an error

UNIX% gcc -O -o p main.c
/tmp/cccpTy0d.o: In function `main':
main.c:(.text+0x5): undefined reference to `swap'
collect2: ld returned 1 exit status
UNIX%
Problem: Multiply Defined Symbols

Different files could define the same symbol

• Is this an error?
• If not, which one should be used? One or many?
Linking: Example

int x = 3;
int y = 4;
int z;

int foo(int a) {...}
int bar(int b) {...}

extern int x;
static int y = 6;
int z;

int foo(int a);
static int bar(int b) {...}

Note: Linking uses object files
Examples use source-level for convenience
Linking: Example

```
int  x = 3;
int  y = 4;
int  z;

int foo(int a) {...}
int bar(int b) {...}
```

Defined in one file

```
extern int  x;
static int  y = 6;
int         z;

int foo(int a);
static int bar(int b) {...}
```

Declared in other files

```
int  x = 3;

int foo(int a) {...}
```

Only one copy exists
Linking: Example

```
int x = 3;
int y = 4;
int z;

int foo(int a) {…}
int bar(int b) {…}
```

```
extern int x;
static int y = 6;
int z;

int foo(int a);
static int bar(int b) {…}
```

Private names not in symbol table. Can’t conflict with other files’ names.

Renaming is a convenient source-level way to understand this.

```
int x = 3;
int y = 4;
int y’ = 6;

int foo(int a) {…}
int bar(int b) {…}
int bar’(int b) {…}
```
C allows you to omit "extern" in some cases – **Don’t!**
Strong & Weak Symbols

Program symbols are either strong or weak

**strong** procedures & initialized globals

**weak** uninitialized globals

```
p1.c
int foo=5;
p1() {}

p2.c
int foo;
p2() {}
```
Strong & Weak Symbols

A strong symbol can only appear once

A weak symbol can be overridden by a strong symbol of the same name
  * References to the weak symbol resolve to the strong symbol

If there are multiple weak symbols, the linker can pick an arbitrary one
Linker Puzzles: What Happens?

- **Int x;**
  - **p1() {}**
  - **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?

- **Int x;**
  - **Int y;**
  - **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: replace r.h.s. int with a struct type, each file then compiled with different alignment rules
Advanced Note: Name Mangling

Other languages (i.e. Java and C++) allow overloaded methods

- Functions then have the same name but take different numbers/types of arguments
- How does the linker disambiguate these symbols?

Generate unique names through *mangling*

- Mangled names are compiler dependent
- Example: class “Foo”, method “bar(int, long)”:
  - bar__3Fooil
  - _ZN3Foo3BarEil
- Similar schemes are used for global variables, etc.
Linking Steps

Symbol Resolution

• Determine where symbols are located and what size data/code they refer to

Relocation

• Combine modules, relocate code/data, and fix symbol references based on new locations

main.o   swap.o

ld (collect2)

p

Relocatable object code

Executable
UNIX% gcc -0 -c main.c
UNIX% readelf -s main.o

Symbol table `.symtab' contains 11 entries:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0000000000000000</td>
<td>19</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>1</td>
<td>main</td>
</tr>
<tr>
<td>9</td>
<td>0000000000000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td>swap</td>
</tr>
<tr>
<td>10</td>
<td>0000000000000000</td>
<td>8</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>3</td>
<td>buf</td>
</tr>
</tbody>
</table>

```
.file "main.c"
.text
.globl main
.type main, @function
main:
.LFB2:
  subq $8, %rsp
.LCFI0:
  call swap
    movl $0, %eax
    addq $8, %rsp
  ret
.LFE2:
    .size main, .-main
.globl buf
    .data
      .align 4
      .type buf, @object
    .size buf, 8
buf:
    .long 1
    .long 2
    ....
```
Symbol table `.symtab' contains 12 entries:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0000000000000000</td>
<td>38</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>1</td>
<td>swap</td>
</tr>
<tr>
<td>9</td>
<td>0000000000000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td>buf</td>
</tr>
<tr>
<td>10</td>
<td>00000000000000008</td>
<td>8</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>COM</td>
<td>bufp1</td>
</tr>
<tr>
<td>11</td>
<td>00000000000000000</td>
<td>8</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>3</td>
<td>bufp0</td>
</tr>
</tbody>
</table>

```assembly
.file   "swap.c"
.text
.globl swap
.type   swap, @function
swap:
.LFB2:
movq    $buf+4, bufp1(%rip)
movq    bufp0(%rip), %rdx
movl    (%rdx), %ecx
movl    buf+4(%rip), %eax
movl    %eax, (%rdx)
movq    bufp1(%rip), %rax
movl    %ecx, (%rax)
ret

.LFE2:
.size   swap, .-swap
.globl bufp0
.data
.align 8
.type   bufp0, @object
.size   bufp0, 8
bufp0:
.quad   buf
.comm   bufp1,8,8
....
```
Symbol Resolution

Undefined symbols must be resolved
- Where are they located
- What size are they?

Linker looks in the symbol tables of all relocatable object files
- Assuming every unknown symbol is defined once and only once, this works well

main.o
.text
.data
...
.symtab

swap.o
.text
.data
...
.symtab

where’s swap?
where’s buf?
Relocation

Once all symbols are resolved, must combine the input files

- Total code size is known
- Total data size is known
- All symbols must be assigned run-time addresses

Sections must be merged

- Only one text, data, etc. section in final executable
- Final run-time addresses of all symbols are defined

Symbol references must be corrected

- All symbol references must now refer to their actual locations
Relocation: Merging Files

main.o

.text
.data
...
symtab

swap.o

.text
.data
...
symtab

p

.text
.data
...
symtab
Linking: Relocation

```c
/* main.c */
void swap(void);
int buf[2] = {1, 2};

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

UNIX% objdump -r -d main.o

main.o: file format elf64-x86-64

Disassembly of section .text:

```assembly
0000000000000000 <main>:
  0:   48 83 ec 08       sub    $0x8,%rsp
  4:   e8 00 00 00 00    callq  9 <main+0x9>
  8:   b8 00 00 00 00    mov    $0x0,%eax
  c:   48 83 c4 08       add    $0x8,%rsp
  10:  c3                retq
```

can also use readelf -r to see relocation information
/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

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

UNIX% objdump -r -D swap.o

swap.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <swap>:
    0:   48 c7 05 00 00 00 00 movq $0x0,0(%rip)
    7:   00 00 00 00

3: R_X86_64_PC32
bufp1+0xfffffffffffffff8
7: R_X86_64_32S buf+0x4

Disassembly of section .data:

0000000000000000 <bufp0>:
    ... 

0: R_X86_64_64 buf

Need relocated address of bufp1 to initialize bufp0 with &buf[0] (== buf)
After Relocation

0000000000000000 <main>:
  0:  48 83 ec 08      sub    $0x8,%rsp
  4:  e8 00 00 00 00   callq  9 <main+0x9>

R_X86_64_PC32 swap+0xfffffffffffffffc

  5:                   swap+0xfffffffffffffffc
  9:  b8 00 00 00 00   mov    $0x0,%eax
 e:  48 83 c4 08      add    $0x8,%rsp
12:  c3                retq

400000 <main>:

00000000000400448 <main>:
  400448:  48 83 ec 08      sub    $0x8,%rsp
  40044c:  e8 0b 00 00 00   callq  40045c <swap>
  400451:  b8 00 00 00 00   mov    $0x0,%eax
  400456:  48 83 c4 08      add    $0x8,%rsp
  40045a:  c3                retq
  40045b:  90                nop

0000000000040045c <swap>:
  40045c:  48 c7 05 01 04 20 00   movq    $0x600848,2098177(%rip)
After Relocation

0000000000000000 <swap>:
  0:  48 c7 05 00 00 00 00 movq $0x0,0(%rip)
  7:  00 00 00 00
       3: R_X86_64_PC32 bufp1+0xfffffffffffffff8
       7: R_X86_64_32S buf+0x4

<..snip..>
0000000000000000 <bufp0>:

          ...

          0: R_X86_64_64 buf

0000000000000000 <swap>:
  40045c:  48 c7 05 01 04 20 00 movq $0x600848,2098177(%rip)
  400463:  48 08 60 00
        # 600868 <bufp1>

<..snip..>
0000000000000000 <bufp0>:
  600850:  44 08 60 00 00 00 00 00
Libraries

How should functions commonly used by programmers be provided?

- Math, I/O, memory management, string manipulation, etc.
- **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 object files into their programs
  - More efficient, but burdensome on the programmer

**Solution: static libraries (.a archive files)**

- Multiple relocatable files + index ≠ single archive file
- Only links the subset of relocatable files from the library that are used in the program
- Example: `gcc -o fpmath main.c float.c -lm`
Two Common Libraries

**libc.a (the C standard library)**
- 4 MB archive of 1395 object files
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- Usually automatically linked

**libm.a (the C math library)**
- 1.3 MB archive of 401 object files
- Floating point math (sin, cos, tan, log, exp, sqrt, ...)
- Use "-lm" to link with your program

UNIX% ar t /usr/lib64/libc.a
  ...
  fprintf.o
  ...
  feof.o
  ...
  fputc.o
  ...
  strlen.o
  ...

UNIX% ar t /usr/lib64/libm.a
  ...
  e_sinh.o
  e_sqrt.o
  e_gamma_r.o
  k_cos.o
  k_rem_pio2.o
  k_sin.o
  k_tan.o
  ...
Creating a Library

/* vector.h */
void addvec(int *x, int *y, int *z, int n);
void multvec(int *x, int *y, int *z, int n);

/* addvec.c */
#include "vector.h"
void addvec(int *x, int *y, int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}

/* multvec.c */
#include "vector.h"
void multvec(int *x, int *y, int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}

UNIX% gcc -c addvec.c multvec.c
UNIX% ar rcs libvector.a addvec.o multvec.o
Using a library

```c
/* main.c */
#include <stdio.h>
#include "vector.h"

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

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

UNIX% gcc –O –c main.c
UNIX% gcc –static –o program main.o ./libvector.a
How to Link: Basic Algorithm

Keep a list of the current unresolved references.

For each object file (.o and .a) in command-line order
  • Try to resolve each unresolved reference in list to objects defined in current file
  • Try to resolve each unresolved reference in current file to objects defined in previous files
  • Concatenate like sections (.text with .text, etc.)

If list empty, output executable file, else error

**Problem:** Command line order matters! Link libraries last:

```
UNIX% gcc main.o libvector.a
UNIX% gcc libvector.a main.o
main.o: In function `main':
main.o(.text+0x4): undefined reference to `addvec'
```
Why UNIX% gcc libvector.a main.o Doesn’t Work

Linker keeps list of currently unresolved symbols and searches an encountered library for them

If symbol(s) found, a .o file for the found symbol(s) is obtained and used by linker like any other .o file

By putting libvector.a first, there is not yet any unresolved symbol, so linker doesn’t obtain any .o file from libvector.a!
## Dynamic Libraries

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked at compile-time</td>
<td>Linked at run-time</td>
</tr>
<tr>
<td>UNIX: foo.a</td>
<td>UNIX: foo.so</td>
</tr>
<tr>
<td>Relocatable ELF File</td>
<td>Shared ELF File</td>
</tr>
</tbody>
</table>

What are the differences?
Static & Dynamic Libraries

**Static**
- Library code added to executable file
- Larger executables
- Must recompile to use newer libraries
- Executable is self-contained
- Some time to load libraries at compile-time
- Library code shared only among copies of same program

**Dynamic**
- Library code not added to executable file
- Smaller executables
- Uses newest (or smallest, fastest, ...) library without recompiling
- Depends on libraries at run-time
- Some time to load libraries at run-time
- Library code shared among all uses of library
**Static & Dynamic Libraries**

### Static

**Creation**

```
ar rcs libfoo.a bar.o baz.o
ranlib libfoo.a
```

**Use**

```
gcc –o zap zap.o -lfoo
```

Adds library’s code, data, symbol table, relocation info, ...

### Dynamic

**Creation**

```
gcc –shared –Wl,-soname,libfoo.so -o libfoo.so bar.o baz.o
```

**Use**

```
gcc –o zap zap.o -lfoo
```

Adds library’s symbol table, relocation info
Loading

Linking yields an executable that can actually be run

Running a program

- `unix% ./program`
- Shell does not recognize “program” as a shell command, so assumes it is an executable
- Invokes the *loader* to load the executable into memory (any unix program can invoke the loader with the `execve` function - more later)
Creating the Memory Image (sort of...)

Create code and data segments
- Copy code and data from executable into these segments

Create initial heap segment
- Grows up from read/write data

Create stack
- Starts near the top and grows downward

Call dynamic linker to load shared libraries and relocate references
Starting the Program

Jump to program’s entry point (stored in ELF header)
  • For C programs, this is the _start symbol

Execute _start code (from crt1.o - same for all C programs)
  • call __libc_init_first
  • call _init
  • call atexit
  • call main
  • call _exit
Position Independent Code

Static libraries compile with **unresolved** global & local addresses

- Library code & data concatenated & addresses resolved when linking
Position Independent Code

By default (in C), dynamic libraries compile with resolved global & local addresses

- E.g., libfoo.so starts at 0x400000 in every application using it

- Advantage: Simplifies sharing

- Disadvantage: Inflexible - must decide ahead of time where each library goes, otherwise libraries can conflict
Position Independent Code

Can compile dynamic libraries with **unresolved** global & local addresses

- gcc -shared -fPIC ...

- **Advantage:** More flexible - no conflicts

- **Disadvantage:** Code less efficient - referencing these addresses involves indirection
Library Interpositioning

Linking with non-standard libraries that use standard library symbols

- “Intercept” calls to library functions

Some applications:

- Security
  - Confinement (sandboxing)
  - Behind the scenes encryption
    - Automatically encrypt otherwise unencrypted network connections

- Monitoring & Profiling
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - malloc tracing
    - Detecting memory leaks
    - Generating malloc traces
Dynamic Linking at Run-Time

Application access to dynamic linker via API:

```c
#include <dlfcn.h>

void
dlink(void)
{
    void *handle = dlopen("mylib.so", RTLD_LAZY);
    /* type */ myfunc = dlsym(handle, "myfunc");
    myfunc(...);
    dlcose(handle);
}
```

Symbols resolved at first use, not now

Error-checking omitted for clarity
Next Time

Lab: Hash tables and linked Lists

Exceptions