CS429: Computer Organization and Architecture

Linking II

Dr. Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: January 13, 2017 at 08:55
Symbols are lexical entities that name functions and variables.
- Each symbol has a value (typically a memory address).
- Code consists of symbol definitions and references.
- References can be either local or external.

```c
int e = 7; // def of global e

int main() {
    int r = a(); // ref to external symbol a
    exit(0); // ref to external symbol exit
    // (defined in libc.so)
}
```

Note that e is locally defined, but global in that it is visible to all modules. Declaring a variable static limits its scope to the current file module.
```c
extern int e;

int *ep = &e; // def of global ep, ref to extern symbol e
int x = 15;    // def of global x
int y;         // def of global y

int a() {
    return *ep + x + y; // refs of globals ep, x, y
}
```
Disassembly of section .text

00000000 <main>:
  0:  55   pushl %ebp
  1:  89 e5 movl %esp, %ebp
  3: e8 fc ff ff ff call 4<main+0x4>
     4: R_386_PC32 a
  8:  6a 00 pushl $0x0
 a: e8 fc ff ff ff call b<main+0xb>
     b: R_386_PC32 exit
  f  90   nop

Disassembly of section .data

00000000 <e>:
  0:  07 00 00 00
a.c

extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return
    *ep + x + y;
}
extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}

Disassembly of section .data

00000000 <ep>:
    0: 00 00 00 00
    0: R_386_32 e

00000004 <x>:
    4: 0f 00 00 00
<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048530</td>
<td>pushl %ebp</td>
<td></td>
</tr>
<tr>
<td>8048531</td>
<td>movl %esp, %ebp</td>
<td>call 8048540 &lt;a&gt;</td>
</tr>
<tr>
<td>8048533</td>
<td>call 8048474 &lt;_init+0x94&gt;</td>
<td></td>
</tr>
<tr>
<td>8048538</td>
<td>pushl $0x0</td>
<td></td>
</tr>
<tr>
<td>804853a</td>
<td>call 8048474 &lt;_init+0x94&gt;</td>
<td></td>
</tr>
<tr>
<td>804853f</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>8048540</td>
<td>pushl %ebp</td>
<td></td>
</tr>
<tr>
<td>8048541</td>
<td>movl 0x804a01c, %edx</td>
<td></td>
</tr>
<tr>
<td>8048546</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>8048547</td>
<td>movl 0x804a020, %eax</td>
<td></td>
</tr>
<tr>
<td>804854c</td>
<td>movl %esp, %ebp</td>
<td></td>
</tr>
<tr>
<td>804854e</td>
<td>addl (%edx), %eax</td>
<td></td>
</tr>
<tr>
<td>8048550</td>
<td>movl %ebp, %esp</td>
<td></td>
</tr>
<tr>
<td>8048552</td>
<td>addl 0x804a3d0, %eax</td>
<td></td>
</tr>
<tr>
<td>8048557</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>8048558</td>
<td>popl %ebp</td>
<td></td>
</tr>
<tr>
<td>8048559</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
m.c

```c
int e = 7;

int main() {
    int r = a();
    exit(0);
}
```

a.c

```c
extern int e;

int *ep = &e;
int x = 15;
int y;

int a() {
    return *ep + x + y;
}
```

Disassembly of section .data

```
0804a018 <e>:
  804a018: 07 00 00 00

0804a01c <ep>:
  804a01c: 18 a0 04 08

0804a020 <x>:
  804a020: 0f 00 00 00
```
Strong and Weak Symbols

Program symbols are either strong or weak.

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

This doesn’t apply to purely local variables.

### p1.c
```
int foo = 5; // foo: strong
p1() {
    // p1: strong
}
```

### p2.c
```
int foo; // foo: weak here
p2() {
    // p2: strong
}
```
Linker 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 one arbitrarily.
Linker Puzzles

What happens in each case?

<table>
<thead>
<tr>
<th>File 1</th>
<th>File 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x; p1() {}</td>
<td>p1() {}</td>
<td>int x; p1() {}</td>
</tr>
<tr>
<td>int x; p1() {}</td>
<td>int x; p2() {}</td>
<td>double x; p2() {}</td>
</tr>
<tr>
<td>int x; int y; p1() {}</td>
<td>double x; p2() {}</td>
<td></td>
</tr>
<tr>
<td>int x=7; int y=5; p1() {}</td>
<td>double x; p2() {}</td>
<td></td>
</tr>
<tr>
<td>int x=7; p1() {}</td>
<td>int x; p2() {}</td>
<td></td>
</tr>
</tbody>
</table>
Linker Puzzles

Think carefully about each of these.

<table>
<thead>
<tr>
<th>File 1</th>
<th>File 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x; p1() {}</td>
<td>p1() {}</td>
<td>Link time error: two strong symbols (p1)</td>
</tr>
<tr>
<td>int x; p1() {}</td>
<td>int x; p2() {}</td>
<td>References to x will refer to the same uninitialized int. What you wanted?</td>
</tr>
<tr>
<td>int x; int y; p1() {}</td>
<td>double x; p2() {}</td>
<td>Writes to x in p2 might overwrite y!</td>
</tr>
<tr>
<td></td>
<td></td>
<td>That’s just evil!</td>
</tr>
<tr>
<td>int x=7; int y=5; p1() {}</td>
<td>double x; p2() {}</td>
<td>Writes to x in p2 might overwrite y!</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Very nasty!</td>
</tr>
<tr>
<td>int x=7; p1() {}</td>
<td>int x; p2() {}</td>
<td>References to x will refer to the same initialized variable.</td>
</tr>
</tbody>
</table>

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.
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 (.a archive files)

- Concatenate related relocatable object files into a single repository with an index (called an archive).
- Enhance the linker so that it tries to resolve unresolved external reference by looking for symbols in one or more archives.
- If an archive member resolves the reference, link into the executable.
libc.a is a static library (archive) of relocatable object files concatenated into one file.

The output p is an executable object file that only contains code and data for libc functions called from p1.c and p2.c.

This further improves modularity and efficiency by packaging commonly used functions, e.g., C standard library (libc) or math library (libm).

The linker extracts only those .o files from the archive that are actually needed by the program.
Command: `ar rs libc.a atoi.o printf.o ... random.o`

Archiver allows incremental updates: Recompile a function that changes and replace the .o file in the archive.
Commonly Used Libraries

**libc.a** (the C standard library)
- 8MB 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)
- 1MB archive of 226 object files
- Floating point math (sin, cos, tan, log, exp, sqrt, ...)

```bash
% ar -t /usr/lib/libc.a | sort
... 
fork.o
... 
printf.o
fpu_control.o
fputc.o
freopen.o
fscanf.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
...
```
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 in obj.
- If there are any entries in the unresolved list at the end of the scan, then error.

Problem:
- Command line order matters.
- Moral: put libraries at the end of the command line.

```
> gcc -L libtest.o -lmine
> gcc -L -lmine libtest.o
libtest.o: In function ‘main’:
libtest.o(.text+0x4): undefined reference to ‘libfun’
```
Executable object file for example program p:

<table>
<thead>
<tr>
<th>ELF header</th>
<th>Program header tables (required for executables)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text section</td>
<td>.data section</td>
</tr>
<tr>
<td>.bss section</td>
<td>.symtab</td>
</tr>
<tr>
<td>.symtab</td>
<td>.rel.text</td>
</tr>
<tr>
<td>.rel.data</td>
<td>.debug</td>
</tr>
<tr>
<td>Section header table (required for relocatables)</td>
<td></td>
</tr>
</tbody>
</table>

Loaded segments:

- Process image: init and shared lib segments
  - Virtual addr: 0x080483e0
- .text segment (r/o)
  - Virtual addr: 0x08048494
- .data segment (initialized r/w)
  - Virtual addr: 0x0804a010
- .bss segment (uninitialized r/w)
  - Virtual addr: 0x0804a3b0
Static libraries have some disadvantages:

- Potential for duplicating lots of common code in the executable files on a file system. (e.g., every 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.
Solution:

- **Shared libraries** (dynamic link libraries DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.

- Dynamic linking can occur when an executable is first loaded and run. (The common case for Linux, handled automatically by `ld-linux.so`.)

- Dynamic linking can also occur after the program has begun.
  - In Linux, this is done explicitly by user with `dlopen()`.
  - Basis for High-Performance Web Servers.

- Shared library routines can be shared by multiple processes.
Dynamically Linked Shared Libraries

Translators (cc1, as)
m.c → m.o

Translators (cc1, as)
a.c → a.o

Linker (ld)
p

Loader/Dynamic Linker (ld−linux.so)

Partially linked executable p (on disk)

Fully linked executable p’ (in memory)

Shared library of dynamically relocatable object files

libc.so functions called by m.c and a.c are loaded, linked, and (potentially) shared among processes.
The Complete Picture

Translators (cc1, as)

m.o

Translators (cc1, as)

a.o

linker (ld)

p

link whatever.a

Loader/Dynamic Linker (ld–linux.so)

p'

libc.so

libm.so