Linking

Lectures 7, 8
CS:APP2e Chapter 7

Linker Puzzles

```
int x;
p1 () {}
p2 () {}
```

```
int x;
p1 () {}
p2 () {}
```

```
int x;
int y;
p1 () {}
p2 () {}
```

```
int x=7;
int y=5;
p1 () {}
p2 () {}
```

```
int x=7;
P1 () {}
P2 () {}
```

A Simplistic Program Translation Scheme

```
Translator
```

```
ASCII source file
```

```
Binary executable object file
(memory image on disk)
```

Problems:
- Efficiency: small change requires complete recompilation
- Modularity: hard to share common functions (e.g. `printf`)

Solution:
- **Static linker (or linker)**
A Better Scheme Using a Linker

Separately compiled relocatable object files

Executable object file (contains code and data for all functions defined in m.c and a.c)

What are all these files?

- **Source code**: plain-text, human-readable
  - what you edit directly, e.g., C source code
- **Assembly code**: plain-text, somewhat readable
  - plain-text representation of machine code using some readable “instruction” mnemonics (e.g., MULT for the multiplication operation);
  - what you worked on in CIS314
- **Object code**: binary, readable with tools (readelf, objdump)
  - partial implementation
  - contains additional information needed for relocation and linking
- **Machine code**: binary, not readable
  - the only code that runs!
  - executed directly by the CPU (or other processor)
- **Library**: binary
  - A collection of object files

What Does a Linker Do?

Merges object files
- Merges multiple relocatable (.o) object files into a single executable object file that can loaded and executed by the loader.

Resolves external references
- As part of the merging process, resolves external references.
  - **External reference**: reference to a symbol defined in another object file.

Relocates symbols
- 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.
  - References can be in either code or data
    ```c
    code: a();        // reference to symbol a */
    data: int *x=6x;  // reference to symbol x */
    ```

Compiler driver

- What you usually think of as the compiler (e.g., gcc) is actually a **compiler driver**, which invokes several other tools:
  ```bash
  gcc -g -o test test.c square1.c
  ```
  ```bash
  cpp [args] test.c /tmp/test.i
  cc1 /tmp/test.i [args] -o /tmp/test.s
  as [args] -o /tmp/test.o /tmp/test.s
  ... similarly for square1.c ... then finally:
  ld -o test [sys. objs] /tmp/test.o /tmp/square1.o
  ```
  ```text
  preprocessor
  compiler
  assembler
  linker
  ```
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.

Executable and Linkable Format (ELF)

Standard binary format for object files
Derives from 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)
- core files
  - generated, for example, when a program receives SIGABRT
  - no sections, has segments (PT_LOAD/PT_NOTE)

Generic name: ELF binaries
Better support for shared libraries than old a.out formats.

ELF Object File Format

ELF header
- Magic bytes (0x7fELF), 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

Section header table (required for relocatables)

ELF Object File Format (cont)

.text section
- Symbol table
- Procedure and static variable names
- Section names and locations

.data section
- Relocation info for .text section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

.bss section
- Relocation info for .data section
- Addresses of pointer data that will need to be modified in the merged executable

.text section
- Symbol table
- Procedure and static variable names
- Section names and locations

.data section
- Relocation info for .text section
- Addresses of instructions that will need to be modified in the executable

.bss section
- Relocation info for .data section
- Addresses of pointer data that will need to be modified in the merged executable

.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

.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)
Example C Program

m.c
int e=7;
int main() {
    int r = a();
    exit(0);
}

a.c
extern int e;
int *ep=&e;
int x=15;
int y;
int a() {
    return *ep+x+y;
}

Merging Relocatable Object Files into an Executable Object File

Relocatable Object Files

Executable Object File

Disassembly of section .text:
00000000 <main>: 00000000 <main>:
0:   55    pushl %ebp
1:   89 e5  movl %esp,%ebp
3:   e8 fc ff ff ff  call 4 <main+0x04>
4: R_386_PC32 a
8:   6a 00  pushl %0x0
a:   e8 fc ff ff ff  call b <main+0x0b>
f:   90    nop

Disassembly of section .data:
00000000 <e>:
0:   07 00 00 00

source: objdump

Relocating Symbols and Resolving External References

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.

Def of local symbol e
Ref to external symbol exit (defined in libc.so)
Ref to external symbol a
Def of local symbol a
Ref to external symbol ep
Def of local symbol ep
Defs of local symbols x and y
Refs of local symbols ep, x, y
```c
extern int e;
int *ep=&e;
int x=15;
int y;
int a() {
    return *ep+x+y;
}
```

**Executable After Relocation and External Reference Resolution (.text)**

```
08048530 <main>:
  55  pushl %ebp
  89 e5  movl %esp,%ebp
  e8 80 00 00 00  call  8048540 <a>
  6a 00  pushl $0x0
  e8 35 ff ff ff  call  8048474 <_init+0x94>
  90  nop
  08048540 <a>:
  55  pushl %ebp
  8b 15 1c a0 04  movl 0x804a01c,%edx
  08048546:  08  
  8b 0a 04 04 08  movl 0x804a020,%eax
  8b 0a 0a 0a 0a  movl %esp,%ebp
  0804854e:  03 02  addl (%edx),%eax
  08048550:  89 ec  movl %ebp,%esp
  08048552:  03 05 d0 a3 04  addl 0x804a3d0,%eax
  08048557:  08  
  8b 0a 04 04 08  movl 0x804a020,%eax
  8b 0a 0a 0a 0a  movl %esp,%ebp
  0804855e:  03 02  addl (%edx),%eax
  08048560:  89 ec  movl %ebp,%esp
  08048562:  03 05 d0 a3 04  addl 0x804a3d0,%eax
  08048567:  08  
  8b 0a 04 04 08  movl 0x804a020,%eax
  8b 0a 0a 0a 0a  movl %esp,%ebp
  08048575:  00  
  8b 0a 04 04 08  movl 0x804a020,%eax
  8b 0a 0a 0a 0a  movl %esp,%ebp
  0804857d:  00  
  8b 0a 04 04 08  movl 0x804a020,%eax
  8b 0a 0a 0a 0a  movl %esp,%ebp
  08048585:  00  
  8b 0a 04 04 08  movl 0x804a020,%eax
  8b 0a 0a 0a 0a  movl %esp,%ebp
  0804858f:  00  
  8b 0a 04 04 08  movl 0x804a020,%eax
  8b 0a 0a 0a 0a  movl %esp,%ebp
  08048599:  c3  ret
```

**Executable After Relocation and External Reference Resolution (.data)**

```
int e=7;
int main() {
    int r = a();
    exit(0);
}
```
**Strong and Weak Symbols**

Program symbols are either strong or weak

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

```plaintext
p1.c
int foo=5;

p1() {
}

p2.c
int foo;

wea

p2() {
}
```

**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**

- Link time error: two strong symbols (p1)
  ```plaintext
  int x;
p1() {}
  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!
  ```plaintext
  int x;
  int y;
p1() {} 
  double x;
p2() {} 
  Writes to x in p2 will overwrite y!
  ```

- References to x will refer to the same initialized variable.
  ```plaintext
  int x=7;
  int y=5;
p1() {}  
  double x;
p2() {}  
  References to x will refer to the same initialized variable.
  ```

**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 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.
Static Libraries (archives)

- Translator
  - p1.c
  - p2.c
  - p1.o  p2.o
  - lib.c

Translator

Linker (ld)

- executable object file (only contains code and data for lib functions that are called from p1.c and p2.c)

Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (libc), math library (libm)]

Linker selectively only the .o files in the archive that are actually needed by the program.

Creating Static Libraries

- Translator
  - atoi.c
  - printf.c
  - random.c
  - atoi.o
  - printf.o
  - random.o
  - ar rs lib.a
  - ar rs lib.a

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

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 in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:
- Command line order matters!

```bash
$ gcc -L libtest.o -lmine
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```
Loading Executable Binaries

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

Process image

Virtual addr

0x080483e0

init and shared lib segments

0x08048494

.text segment (r/o)

0x0804a010

.data segment (initialized r/w)

0x0804a3b0

.bss segment (uninitialized r/w)

Loader

When you run a program, e.g., "/a.out", this is what happens

- The shell invokes the loader function (execve), which copies the code and data in the executable file a.out into memory, and then transfers control to the beginning of the program.
- Example for gcc and linux: [http://linuxgazette.net/84/hawk.html](http://linuxgazette.net/84/hawk.html)
- The default name “a.out” stands for Assembler OUTput

Linker vs Loader:

- The linker generates the ELF executable and stores it on disk (performing symbol resolution and address relocation)
- The loader copies the program image from disk to main memory (and may also allocate storage and map virtual addresses to disk pages)
- Either can do relocation

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

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 executable is first loaded and run.
    - Common case for Linux, handled automatically by `ld-linux.so`
    - Dynamic linking can also occur after 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

![Diagram of dynamic linking process]
The Complete Picture

Translator

m.o

m.c

Translator

a.o

libwhatever.a

Static Linker (ld)

p

libc.so libm.so

Loader/Dynamic Linker
(ld-linux.so)

p'