Linking

Week 3
CS:APP2e Chapter 7

Topics
- Static linking
- Object files
- Static libraries
- Loading
- Dynamic linking of shared libraries

Linker Puzzles

```plaintext
int x:
p1() {} p1() {}
int x:
p1() {} p2() {}
int x:
double x:
p1() {}
double x:
p2() {}
int x=7:
l = 5:
p1() {} p1() {}
```

A Simplistic Program Translation Scheme

```
```

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

```
```

Compiler driver

- What you usually think of as the compiler (e.g., gcc) is actually a compiler driver, which invokes several other tools:

  ```
gcc -g -o test test.c square1.c
  ```

  ```
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:
l -o test [sys. objs] /tmp/test.o /tmp/square1.o
  ```
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

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?

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.

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
    - code: a(); /* reference to symbol a */
    - data: int *p=x; /* reference to symbol x */

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
- "Better Save Space"
- Has section header but occupies no space

**.bss section**
- Uninitialized (static) data
- "Block Started by Symbol"

**Section header table** (required for relocatables)

**.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)

**ELF Object File Format (cont)**

**Program header table** (required for executables)
- .text section
- .data section
- .bss section
- .symtab
- .rel.text
- .rel.data
- .debug

**Section header table** (required for relocatables)
Example C Program

```
Example C Program

``` Int e = 7;
    int main() {
        int x = a();
        exit(0);
    }
```

```
Merging Relocatable Object Files into an Executable Object File

```

```
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*.

```

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

```
08048530 <main>: pushl %ebp
08048531: 89 e5 movl %esp,%ebp
08048533: 89 08 00 00 00 call 08048540 <a>
08048538: 6a 00 pushl $0x0
0804853a: e8 35 ff ff ff call 0804874 <__init+0x94>
0804853f: 90 nop
```

**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

```
strong

int foo=5;

strong

p1() {
    ...
}

weak

int foo;

strong

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**

```
int x, p1() {}

Link time error: two strong symbols (p1)

int x, p1() {}

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

int x, int y, p1() {}

daule x:

double x, p2() {}

Writes to x in p2 might overwrite y! Evil!

double x, p2() {}

Writes to x in p2 will overwrite y! Nasty!

int x=7, p1() {}

References to x will refer to the same initialized variable.

int x=7, p1() {}

int x, p2() {}

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.
```

**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

\[ p1.c \quad p2.c \]

\[ \text{Linker (ld)} \]

\[ \text{libc.a} \]

static library (archive) of relocatable object files concatenated into one file.

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

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

Creating Static Libraries

Translator

\[ \text{atoi.c} \quad \text{printf.c} \quad \text{random.c} \]

\[ \text{Linker (ld)} \]

\[ \text{libc.a} \]

Archiver (ar)

\[ \text{ars rs libc.a \ Considering .o files and .a files in the command line order.} \]

\[ \text{Scan .o files and .a files in the command line order.} \]

\[ \text{During the scan, keep a list of the current unresolved references.} \]

\[ \text{As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.} \]

\[ \text{If any entries in the unresolved list at end of scan, then error.} \]

Problem:

\[ \text{Command line order matters!} \]

Using Static Libraries

Linker’s algorithm for resolving external references:

\[ \text{Scan .o files and .a files in the command line order.} \]

\[ \text{During the scan, keep a list of the current unresolved references.} \]

\[ \text{As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.} \]

\[ \text{If any entries in the unresolved list at end of scan, then error.} \]

Problem:

\[ \text{Command line order matters!} \]

Using Static Libraries

Linker’s algorithm for resolving external references:

\[ \text{Scan .o files and .a files in the command line order.} \]

\[ \text{During the scan, keep a list of the current unresolved references.} \]

\[ \text{As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.} \]

\[ \text{If any entries in the unresolved list at end of scan, then error.} \]

Problem:

\[ \text{Command line order matters!} \]

Loading Executable Binaries

Executable object file

\[ \text{Program header table (required for relocatables)} \]

\[ \text{.text section} \]

\[ \text{.data section} \]

\[ \text{.bss section} \]

\[ \text{.symtab} \]

\[ \text{.rel.text} \]

\[ \text{.rel.data} \]

\[ \text{debug} \]

\[ \text{Section header table (required for relocatables)} \]

\[ \text{ELF header} \]

Virtual addr

0x0080493e0

0x008048494

0x00804ae10

0x00804ae00

0x00804a3b0

Loader

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

\[ \text{The shell invokes the loader function (execve), which copies the code and data in the executable file .a.out into memory, and then transfers control to the beginning of the program.} \]

\[ \text{Example for gcc and linux: } \text{http://linuxgazette.net/84/hawk.html} \]

\[ \text{The default name "a.out" stands for Assembler OUTput} \]

Linker vs Loader:

\[ \text{The linker generates the ELF executable and stores it on disk (performing symbol resolution and address relocation)} \]

\[ \text{The loader copies the program image from disk to main memory (and may also allocate storage and map virtual addresses to disk pages)} \]

\[ \text{Either can do relocation} \]
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

The Complete Picture