CS412/413

Introduction to Compilers
Radu Rugina

Lecture 36: Linking and Loading
01 May 06

Outline

• Static linking
  – Object files
  – Symbol resolution
  – Relocation

• Libraries
  – Shared libraries
  – Dynamically linked libraries

• Book: “Linkers and Loaders”, by J. Levine

Big Picture

• Output of compiler is a set of assembly/object files
  – Not executable
  – May refer to external symbols (variables, functions, etc.)
  – Each object file has its own address space

• Linker joins together object files into one executable file
• Loader brings program in memory and executes it

Main Issues

• Symbol resolution
  – May have in one module references to external symbols from another module
  – Linker fixes such references when combining modules

• Relocation
  – Symbols may have different addresses in the final executable (they have been relocated)
  – Linker must fix references to relocated symbols
  – Loader may also relocate symbols

• Program loading
  – Bring the program from disk to memory
  – May require setting up virtual memory

Symbol Resolution

source code

extern int foo(int x);
extern int n;
...
foo(n);

assembly code

push _n
call _foo
add $4, $gp

object code

ff 35 00 00 00 00
a8 00 00 00 00
83 c4 04

to be filled in by linker

Executable Code

00401044: _bar:
401044: 55
401045: 89 a5
401046: ff 35 00 00 00
401047: push ldx(0x0)
401048: 8e 1e 00 00 00
401049: call 010170: _foo
40104b: 83 c4 04
40104d: add $bnr, $gp
40104f: mov $gp, $sp
401051: 5d
401052: pop $sp
401053: c3
401054: ret

00401070: _foo:
401070: 55
401071: 89 a5
401072: mov $sp, $gp
...

00402008: _nr:
402008: 64 00 00 00
Relocation Problem

- Object files have separate address spaces
- Need to combine them into an executable with a single (linear) address space
- Relocation = compute new addresses in the new address space (add a relocation constant)

Example:

```
file1

extern int foo(int x);
int n = 100;
int foo(int x) {
    return n + i;
}
```

file2

```
extern int n;
... 
```

Relocation Example

- Object file:

```
file: 
...
3: 00 40 08
6: 00 40 00 00 00 00
b: 00 41 01
file: 
...
4: 00 40 00 00
```

- Executable file:

```
file: 
...
40107b: 00 40 08
40107c: 00 40 00 00 00 00
40108c: 00 41 01
file: 
...
40109e: 00 40 00 00
```

Unresolved Refs vs. Relocation

- Similar problems: have to compute new address in the resulting executable file
- Several differences
- External (unresolved) symbols:
  - Space for symbols allocated in other files
  - Don’t have any address before linking
- Relocated symbols:
  - Space for symbols allocated in current file
  - Have a local address for the symbol
  - Don’t have absolute addresses
  - Don’t need relocation if we use relative addresses

Object File Structure

- Object file contains various sections
- Text section contains the compiled code with some patching needed
- Initialized data: need to know initial values
- Uninitialized data: only need to know total size of data segment
- Points to places in text and data section that need fix-up

Action of Linker

```
f1.o

object files

f2.o

f3.o

executable file
```

Two-pass Linking

- Usually need two passes to resolve external references and to perform relocation

  - Pass 1: read all modules and construct:
    - Table with modules names and lengths
    - Global symbol table: all unresolved references (symbols used, but not defined by a module) and entry points (symbols defined by a module)

  - Pass 2: combine modules:
    - Compute relocation constants
    - Perform relocation
    - Resolve external references
1. • Same as object file, but code is ready to be executed as-is
• Pages of code and data brought in lazily from text and data section as needed: rapid start-up
• Symbols allow debugging
• Text section shared across processes

Executable File Structure

File Formats

• Unix:
  – a.out format
  – COFF: Common Object File Format
  – ELF: Executable and Linking Format
  – All support both executable and object files

• Windows:
  – COM, EXE: executable formats
  – PE: Microsoft Portable Executable format
    • For Windows NT, adapted from COFF

Libraries

• Library = collection of object files
• Linker adds all object files necessary to resolve undefined references in explicitly named files
• Object files, libraries searched in user-specified order for external references

Unix linker: id
  ld main.o foo.o /usr/lib/lib11.a /usr/lib/libc.a
Microsoft linker: link
  link main.obj foo.obj kernel32.lib user32.lib ...
• Index over all object files in library for rapid searching

Unix: ranlib
  ranlib mylib.a

Shared Libraries

• Problem: libraries take up a lot of memory when linked into many running applications
• Solution: shared libraries
**Intra-Library Calls**

- **Problem:** shared libraries may depend on external symbols (even symbols within the shared library); different applications may have different linkage:
  
  ```
  ld ~o prog1 main.o /usr/lib/libc.a
  ld ~o prog2 main.o mymalloc.o /usr/lib/libc.a
  ```

- If routine in libc.a calls malloc(), for prog1 should get standard version; for prog2, version in mymalloc.o
- Calls to external symbols are made through global tables unique to each program

**Malloc Example**

**Dynamic Linking**

- **Idea:** link shared libraries when loading the program or at run-time:
  - Easier to create
  - Easier to update
  - Programs can load and unload routines at run-time
- **Drawback:** loading-time or run-time overhead

**Dynamic Shared Objects**

- **Unix systems:** Code is typically compiled as a dynamic shared object (DSO), a relocatable shared library
  
  - Shared libraries in UNIX use the ELF format, which supports Position-Independent Code (PIC)
    - Program can determine its current address
    - Add constant offset to access local data
    - If data located in a different library, use indirect through a Global Offset Table (GOT).
  - Address of GOT usually computed and stored in a register at the beginning of each procedure

**Dynamic Shared Objects**

- For calls to methods in shared libraries, uses procedure linkage tables (PLT) – same as GOT, but with entries for functions.
  - Entries represent pointers to functions from the shared library that may be invoked
  - The dynamic linker fills the PLT lazily: it fills in the entry for a function the first time that function is invoked
  - Subsequent calls just use the function pointer stored in the PLT

**Cost of DSOs**

- Assume `%ebx` contains PLT/GOT address
- Call to function:
  ```
  call *f_offset(%ebx)
  ```
- Global variable accesses:
  ```
  mov v_offset(%ebx), %eax
  mov (%eax), %eax
  ```
- Calling global functions = calling methods
- Accessing global variables is more expensive than accessing local variables
- Most computer benchmarks run w/o DSOs!
Dynamic Linking

- DSOs can be linked dynamically into a running program
- Implicit dynamic linking: when setting up global tables, shared libraries are automatically loaded if necessary (even lazily). Symbols looked up & global tables created.
- Explicit dynamic linking: application can choose how to extend its own functionality
  - Unix example:
    - `h = dlopen(filename)` loads an object file into the memory, if not already loaded;
    - `p = dlSYM(h, name)` queries for symbols in the library