CS412/413
Introduction to Compilers
Radu Rugina
Lecture 35: Linking and Loading
26 Apr 04

Outline
- Static linking
  - Object files
  - Libraries
  - Shared libraries
  - Relocatable code
- 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 need to relocate symbols
- Program loading
  - Bring the program from disk to memory
  - May require setting up virtual memory

Symbol Resolution

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

assembly code

```
ff 35 00 00 00 00
```

object code

```
e8 00 00 00 00
83 c4 04
```

to be filled in by linker

Executable Code

```
00000000 <__bar>:
00000004:  55
00000005:  89 e5
00000006:  89 55 08 20 40 00
00000010:  e8 80 00 00 00
00000014:  00 00 00 00
00000018:  call 401057 <__foo>
0000001c:  83 c4 04
0000001e:  89 ac
00000020:  00 00 00 00
00000024:  00 00 00 00

00000000 <__foo>:
00000004:  55
00000005:  89 e5
00000006:  89 55 08 20 40 00
00000010:  e8 80 00 00 00
00000014:  add 0x04, %esp
00000018:  mov %ebp, %esp
0000001c:  push %ebp
```

reset
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:

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

Relocation Example

- Object file:

```
00000000 <_foo>: 3: 8b 45 08  mov 0xb (ebp), %eax
4: 8b 0d 04 00 00 00  mov 0x4, %eax
d: 8d 14 01  lea  (%eax,%eax,1), %edx
```

- Executable file:

```
00000000 <_foo>: 3: 8b 45 08  mov 0xb (ebp), %eax
40107b: 8b 45 08  mov 0xb (ebp), %eax
40107e: 8b 0d 04 00 00 00  mov 0x4, %eax
401084: 8d 14 01  lea  (%eax,%eax,1), %edx
```

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

```
\[ \text{f1.o} \rightarrow \text{f2.o} \rightarrow \text{f3.o} \rightarrow \text{a.exe} \]
```

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
**Executable File Structure**

- 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

**Executing Programs**

- Multiple copies of program share code (text), have own data
- Data appears at same virtual address in every process

**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

**Shared Libraries**

- Problem: libraries take up a lot of memory when linked into many running applications
- Solution: shared libraries

- Executable file refers to, does not contain library code; library code brought in the address space when the program is loaded
- Library compiled at fixed address, far away from the application (e.g. Linux: hex 60000000, BSD a0000000)
- Link program against stub library (no code, data)
- Shared library uses a jump table: client code jumps to jump table and follows indirection (useful for library updates)
Intra-Library Calls

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

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 indirection through a Global Offset Table (GOT).
  
  - Address of GOT usually computed and stored in a register at the beginning of each procedure

Cost of DSOs

- Assume ebx contains PLT/GOT address
- Call to function f:
  
  ```
  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: \texttt{h = dlopen(filename)} loads an object file into some free memory (if necessary), allows query of globals: \texttt{p = dlsym(h, name)}
  - Windows: \texttt{h = LoadLibrary(filename)},
    \texttt{p = GetProcAddress(h, name)}