CS412/413

Introduction to Compilers

Radu Rugina

Lecture 33: Linking and Loading

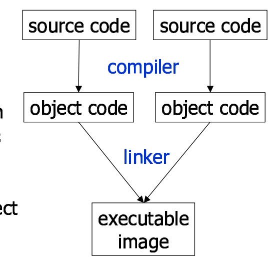
22 Apr 02

Outline

- Static linking
  - Object files
  - Libraries
  - Shared libraries
  - Relocatable code
- Dynamic linking
  - Explicit vs. implicit linking
  - Dynamically linked libraries
  - Dynamic shared objects
- Book: "Linkers and Loaders", by J. Levine

Object Files

- Output of compiler is a set of object files
  - Not executable
  - May refer to external symbols (variables, functions, etc.) whose definition is not known
  - Each object file has its own address space
- Linker joins together object files, resolves external references, relocates



Unresolved References

source code

```
extern int abs( int x );
…
y = y + abs(x);
```

assembly code

```
push %ecx
call _abs
add %eax, %ebx
```

object code

51				
e8	00	00	00	00
01	c3			

} to be filled in by linker

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:

```
char c;
c = 'A';
```

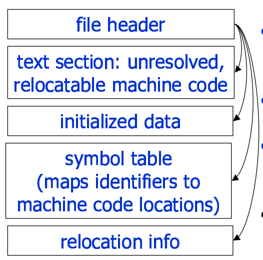
```
movb $65, _c
```

c6	05	00	00	00	00	41

Unresolved Refs vs. Relocation

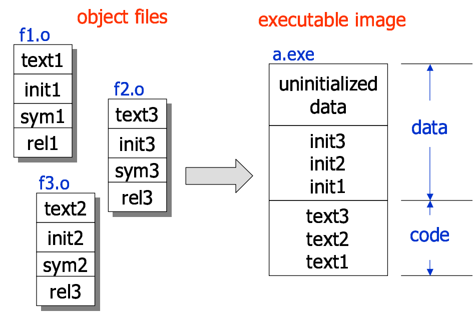
- Similar problems: have to compute new address in the resulting executable file
- Several differences
- External (unresolved) symbols:
  - Space for symbols allocated in other files
  - Don't have any address before linking
- Relocated symbols:
  - Space for symbols allocated in current file
  - Have 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

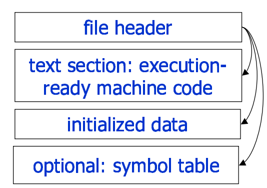


Two-Pass Linking

- 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

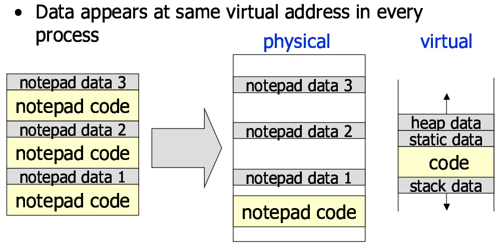
Executable File Structure

- 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



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: ld

  ld main.o foo.o /usr/lib/X11.a /usr/lib/libc.a

  Microsoft linker:link

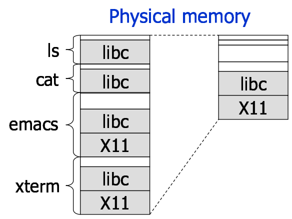
  link main.obj foo.obj kernel32.lib user32.lib ...
- Index over all object files in library for rapid searching

  Unix: ranlib

  ranlib mylib.a

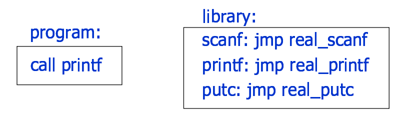
Shared Libraries

- Problem: libraries take up a lot of memory when linked into many running applications
- Solution: shared libraries (e.g. DLLs)



Step 1: Jump Tables

- Executable file refers to, does not contain library code; library code loaded dynamically
- Library code found in separate shared library file (similar to DLL); linking done against import library that does not contain code
- Library compiled at fixed address, starts with jump table to allow new versions; client code jumps to jump table (indirection).



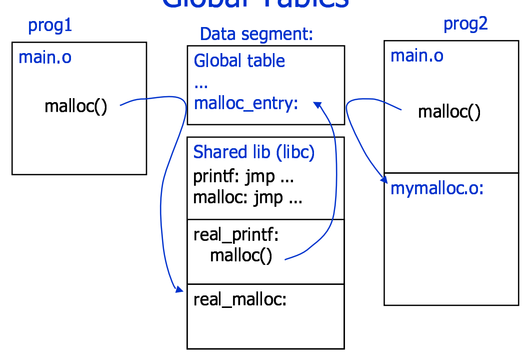
Global Tables

- 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

Global Tables



Using Global Tables

- Global table contains entries for all external references

malloc(n) ⇒ push n
mov (malloc_entry), %eax
call %eax # indirect jump!

- Non-shared application code unaffected
- Same-object references can still be used directly
- Global table entries (malloc_entry) placed in non-shared memory locations so each program has different linkage
- Initialized by dynamic loader when program begins: reads symbol tables, relocation info

Relocation

- After combining object files into executable image, program has a linear address space
- With virtual memory, all programs could start at same address, could contain fixed addresses
- Problem with shared libraries (e.g., DLLs): if allocated at fixed addresses, can collide in virtual memory (code, data, global tables, …)
  - Collision ⇒ code copied and explicitly relocated

Dynamic Shared Objects

- Unix systems: Code is typically compiled as a dynamic shared object (DSO): relocatable shared library
- Shared libraries can be mapped to any address in virtual memory—no copying!
- Questions:
  - how to make code completely relocatable?
  - what is the performance impact?

Relocation Difficulties

- Can't use absolute addresses (directly named memory locations) anywhere:
  - Not in calls to external functions
  - Not for global variables in data segment
  - Not even for global table entries

```
push n
mov (malloc_entry), %eax   # Oops!
call %eax
```

- Not a problem: branch instructions, local calls (when using relative addressing)

Global Tables

- Can put address of all globals into global table
- But...can't put the global table at a fixed address: not relocatable!
- Solutions:
  1. Pass global table address as an extra argument (possibly in a register)
  2. Use address arithmetic on current program counter (eip register) to find global table. Use link-time constant offset between eip and global table.
  3. Stick global table entries into the current object's dispatch vector : DV is the global table (only works for methods, but otherwise the best)

Cost of DSOs

- Assume esi contains global table pointer (set-up code at beginning of function)
- Call to function f:

```
call f_offset(%esi)
```

- Global variable accesses:

```
mov v_offset(%esi), %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!

Link-time Optimization

- When linking object files, linker provides flags to allow optimization of inter-module references
- Unix: –non_shared (or –static) link option means application to get its own copy of library code
  - calls and global variables performed directly
- Allows performance/functionality trade-off

call malloc_addr(%esi) ⟹ call malloc

Dynamic Linking

- Shared libraries (DLLs) and 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: h = dlopen(filename) loads an object file into some free memory (if necessary), allows query of globals: p = dlsym(h, name)
  - Windows: h = LoadLibrary(filename),
    p = GetProcAddress(h, name)

Conclusions

- Shared libraries and DSOs allow efficient memory use on a machine running many different programs that share code
- Improves cache, TLB performance overall
- Hurts individual program performance by adding indirections through global tables, bloating code with extra instructions
- Important new functionality: dynamic extension of program
- Peephole linker optimization can restore performance, but with loss of functionality