CS 412
Introduction to Compilers

Andrew Myers
Cornell University

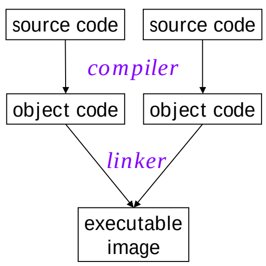
Lecture 34: Linking and Loading
25 Apr 01

Outline

- Static linking
  - Object files
  - Libraries
  - Shared libraries
  - Relocatable code
- Dynamic linking
  - explicit vs. implicit linking
  - dynamically linked libraries/dynamic shared objects

Object files

- Output of compiler is an *object file*
  - not executable
  - may refer to external symbols (variables, functions, etc.) whose definition is not known.
- Linker joins together object files, resolves external references



Unresolved references

source code

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

assembly code

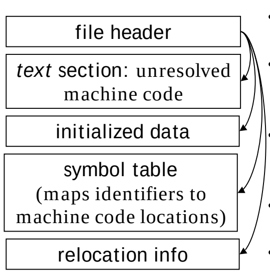
```
PUSH ecx
CALL _abs
ADD ebx, eax
```

object code

51				
9A	*00*	*00*	*00*	*00*
03	D8			

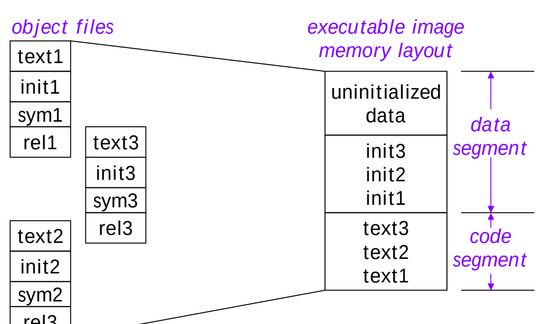
to be filled in by linker

Object file structure



- Object file contains various ***sections***
- ***text*** section contains the compiled code with some patching needed
- For uninitialized data, only need to know total size of data segment
- Describes structure of text and data sections
- Points to places in text and data section that need fix-up

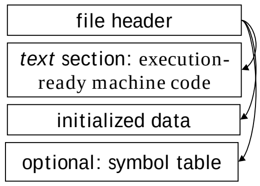
Action of Linker



+peephole optimizations

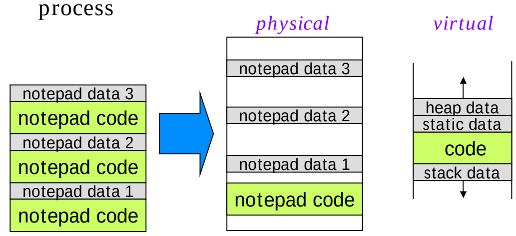
Executable file structure

- 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
- Text section shared across processes
- Symbols allow debugging



Executing programs

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

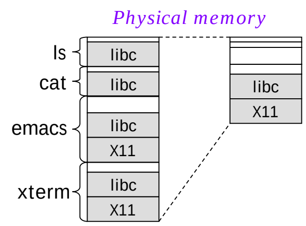


Libraries

- *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:** ld main.o foo.o /usr/lib/X11.a /usr/lib/libc.a
  - **NT:** link main.obj foo.obj kernel32.lib user32.lib ...
- Index over all object files in library for rapid searching

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).

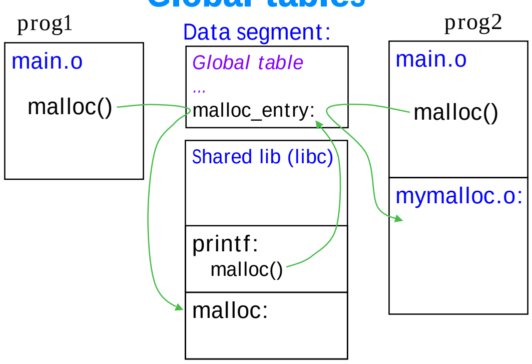
program: *library:*

```
                scanf: jmp real_scanf
call printf     printf: jmp real_printf
                putc: jmp real_putc
```


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 [ebp + n]
            mov eax, [malloc_entry]
            call eax                ; indirect jump!
```

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

Relocation

- Before widespread support for virtual memory, code had to be *relocatable* (could not contain fixed memory addresses)
- 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
- Back to relocatable code!

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 [ebp + n]
mov eax, [malloc_entry]    ; Oops!
call eax
```

- Not a problem: branch instructions, local calls. Use *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!
- Three approaches:
  1. Pass global table address as an extra argument (possibly in a register) : affects first-class functions (next global table address stored in current GT)
  2. Use address arithmetic on current program counter (eip register) to find global table. Offset between eip and global table is a link-time constant
  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 [esi + f_offset]
- Global variable accesses:

  mov eax, [esi + v_offset]
  mov ebx, [eax]
- Calling global functions ≈ calling methods
- Accessing global variables is *more* expensive than accessing local variables
- Most computer benchmarks run w/o DSOs!

Module values return

- Let *M* be an external module, *f* a fcn in *M*
- When accessing *M.f*, go through global table:

  mov eax, [si + f_offset]
- Looks just like the code to access a field *f* of a record located at si...
- si refers to a module value!
- Dynamic loader creates module values as program starts (actually creates multiple copies for various using modules; si points to concatenated records for all modules used by the current code's module)

Link-time optimization

- When linking object files, linker provides flags to allow peephole optimization of inter-module references
- Unix: –non_shared link option means application to get its own copy of library code
  - calls and global variables performed directly (peephole opt.)

    call [esi + malloc_addr] ⇒ call malloc
- Allows performance/functionality trade-off

Dynamic linking

- Shared libraries (DLLs) and DSOs can be linked dynamically into a running program
- Normal case: implicit 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