CS 3411 Systems Programming

Department of Computer Science
Michigan Technological University

Compilation and Linking

Today's Topics

- Compilation and Linking
- Libraries

Compilation and Linking

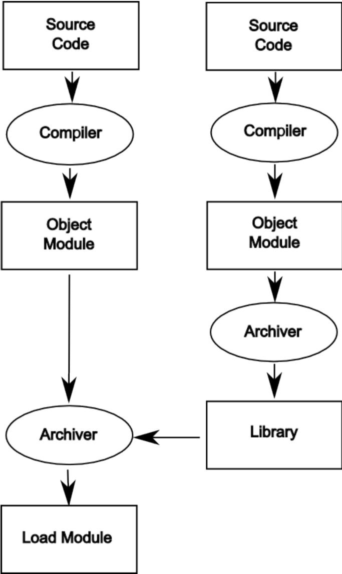
```c
#include <stdio.h>

int x;
int z;
float arr[100];

main() {
  x = 0; z = 0;
  int res = f(3);
  printf("f(3)=%d x=%d z=%d\n", res, x, z);
}
```

- Code for int f(int) not available yet (nor printf)
- x and z available to other object modules
- Compiled module must reflect these facts

Compilation and Linking



Compilation and Linking

- Compiler: Converts program from source file to machine language, produces and object module
- Linker: Produces a load module which is ready to be executed
- Operating system creates a process from the load module

Compilation and Linking

```
static int z;
int f(a)
int a;
{
  extern int x;
  x = 14; z = 1;
  return a;
}
```

- Let's try checking out what the compiled code looks like!

Object Files

- Object file may contain unresolved global symbols
- Defined: Variables, functions defined within object file, can be referenced within other object files
- Undefined: variables, functions used within this object file, defined elsewhere
- Linker combines object files and resolves symbols while creating executable
  - Object file contains symbol table
  - Symbol table will contain information needed to resolve symbols
  - Linker uses information from the symbol table
- Executable will contain no unresolved symbols

Object Modules

- Has many different formats (ELF, COFF)
- Header section - Sizes required to parse object module and create program
- Machine code - Generated machine code (the text section!)
- Initialized Data - Initialized global and static data (doesn't go on stack)
- Symbol Table - External Symbols
  - Undefined - Used in this module, defined elsewhere
  - Defined - Defined in this module, may be undefined in another module
- Relocation Information - Record of places where symbols must be relocated

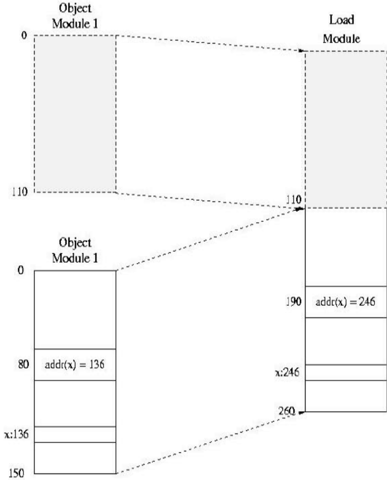
Tools for Examining Object Files

- file
- nm
- objdump
- readelf

Linking

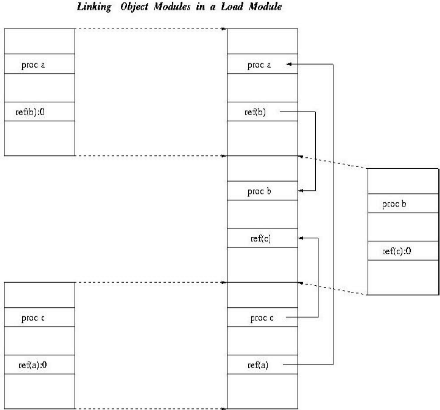
- Object module will (usually) assume starting address is zero
- Linker combines several object modules
  - Text sections combined, data sections combined, ...
- Combined modules cannot all start at zero!
- Cannot have unresolved references in load module
- Two tasks then:
  - Relocate modules (account for starting address that results from combining modules)
  - Link modules (resolve undefined external references)

Relocation



[1] Figure taken from *Operating Systems: A Design-Oriented Approach*, Charles Crowley, Irwin, 1997

Linking



[2] Figure taken from *Operating Systems: A Design-Oriented Approach*, Charles Crowley, Irwin, 1997

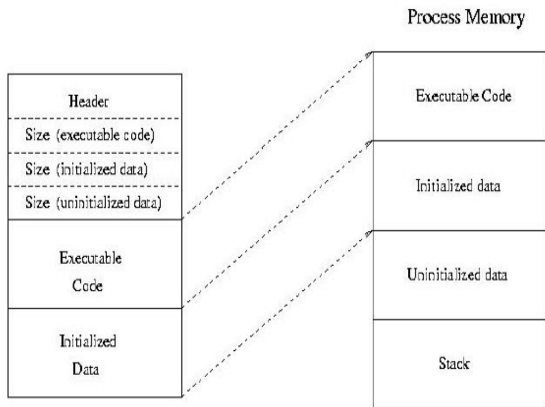
Load Module Creation

1. Create load module and global symbol table
2. Get next object module or library name
3. Object module
   3.1 Insert code and data, remember where
   3.2 Relocate object module and all symbols in module's symbol table
   3.3 Undefined external references
       - Already defined in global symbol table, write value in just loaded object module
       - Not yet defined, note that links must be fixed when symbol defined
   3.4 Defined external references:
       - Fix up all previous references (to this symbol) noted in global symbol table

Load Module Creation

4. Library
   - Find each undefined external reference in global symbol table
   - See if symbol defined in library
   - If so, load it per step (3)
5. Back to step 2

- Load module need not contain reloaction (in most cases) or symbol table sections
- Symbol table information may be used by debugger or stripped to reduce binary size



[3] Figure taken from *Operating Systems: A Design-Oriented Approach*, Charles Crowley, Irwin, 1997

Static Linking

- Library routines combined into binary program image
- Creates large load modules
- Same library may be contained in multiple images throughout file system
- Once load module is created, it is impervious to changes in referenced library
  - Cannot incorporate new versions without recompilation
  - Does not depend on existence of (specific version of) library on system
- gcc -static ...

Dynamic Linking

- Stub included in binary program image for each library-routine reference
- Stub is code to locate memory-resident routine or load it if library routine not present
- Stub replaces itself with address of routine and executes routine
- Will use most recent version of library routine
- Higher overhead during use; faster startup than statically linked
- Allows same code to be shared by multiple processes
- All processes using a language library execute single copy of library code (shared library)
- Generally requires help from OS (code mapped into multiple address spaces)
- More efficient use of memory