CS354: Machine Organization and Programming

Lecture 37
Monday the November 30th 2015

Section 2
Instructor: Leo Arulraj

Class Announcements

1. Final exam will be for 1.5 hours total duration.
2. Date: 12/18 Time: From 10:05 to 12:05 at ENGR Hall Room Number 1800 (for Section 2)
3. Final exam is not cumulative.
4. To give you a general Idea about the overall letter grade assignment: (might change)
   - A/AB 90+
   - B/BC 80+
   - C 70+
   - D/F <60

Lecture Overview

1. Recap of Compilation Process
2. Types of object files
3. Relocatable object files
4. Symbols and Symbol tables
5. Static Linking: Symbol resolution

Why learn about Linkers?

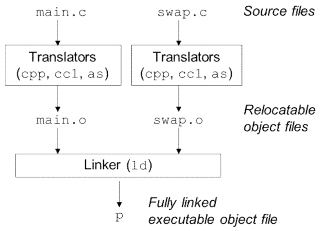
Will help you:

1. Build large programs
2. Avoid dangerous programming bugs
3. Understand how language scoping rules are implemented
4. Understand other important systems concepts
5. Exploit shared libraries

Compilation Process

1. C Preprocessor (cpp) translates main.c into main.i intermediate file
2. C Compiler (cc1) translates main.i to main.s assembly file
3. Assembler (as) translates main.s into main.o relocatable object file
4. Finally, Linker (ld) creates the executable object file.

Static Linking



Static Linkers generate a fully linked executable object file from a collection of relocatable object files.

Static Linking

Two main tasks involved in static linking:

Symbol resolution: Associate every symbol reference in object files with exactly one symbol definition.

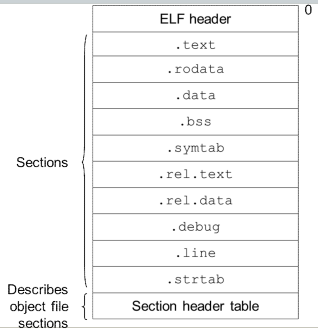
Relocation: Relocate the code and data sections of different object files that all start at address 0 and also make sure all symbol references are to their relocated address.

Object Files

Three types of object files:

1. Relocatable object file: Contains binary code and data in a form that can be combined with other relocatable object file.
2. Executable object file: Contains binary code and data that can be copied directly into memory and executed.
3. Shared object file: A type of relocatable object file that can be loaded into memory and linked dynamically at either load time or run time.

An ELF Relocatable Object File



ELF Relocatable Object File

.symtab: functions and global variables defined and referenced in the program.

.rel.text: A list of locations in the .text section that will need to be modified when the linker combines this object file with others.

.rel.data: Relocation information for any global variables that are referenced or defined by the module.

.debug: A debugging symbol table with entries for local variables, typedefs, global variables and the original C source file. Requires –g option.

.line: A mapping between line numbers in the original C source program and machine code instructions in the .text section. Requires –g option.

.strtab: A string table for the symbol tables in the .symtab and .debug sections and for the section names in the section headers.

Symbols

Three types of Symbols (from the linker's perspective):

Global symbols that are defined by module *m* and that can be referenced by other modules.

Global symbols that are referenced by module *m* but defined by some other module.

Local symbols that are defined and referenced exclusively by the module *m*. E.g. defined with "*static*" attribute. Important: Local linker symbols are not the same as local program variables.

Local Symbols defined with "static"

```
int f(){
    static int x=0;
    return x;
}

int g(){
    static int x=1;
    return x;
}
```

Local Variables named "x" are not managed on the stack.

Compiler allocates space in .data or .bss for each definition and creates a local linker symbol in the symbol table with unique name.

Eg. x.1 for definition in function f and x.2 for definition in function g.

Local Symbol Table Entry

```
typedef struct {
int name;          /*String table offset*/
int value;         /*Section offset or VM address*/
int size;          /*Object size in bytes*/
char type:4,       /*Data, func, sec or src file name*/
     binding:4;    /*Local or global*/
char reserved;     /*Unused*/
char section;      /*Pseudo section header index (ABS,
                     UNDEF or COMMON)*/
}Elf_Symbol;
```

Symbol Resolution

Local Symbol Resolution: is straightforward because compiler makes sure there is only one definition of each local symbol per module.

Global Symbol Resolution: Trickier!
At compile time, compiler exports each global symbol as either *strong or weak*.

Strong: Functions and Initialized Global Variables get strong symbols.
Weak: Uninitialized global variables get weak symbols.

Global Symbol Resolution

Rules for dealing with multiply defined global symbols:

1. Multiple strong symbols are not allowed.
2. Given a strong symbol and multiple weak symbols, choose the strong symbol.
3. Given multiple weak symbols, choose any of the weak symbols.

CS354: Machine Organization and Programming

Lecture 38
Wednesday the December 2nd 2015

Section 2
Instructor: Leo Arulraj

Lecture Overview

1. Static Linking: Relocation
2. Executable object files
3. Dynamic Linking: Shared Libraries
4. Dynamic Linking from application: Example Code
5. Position Independent Code
6. Tools for Manipulating Object files

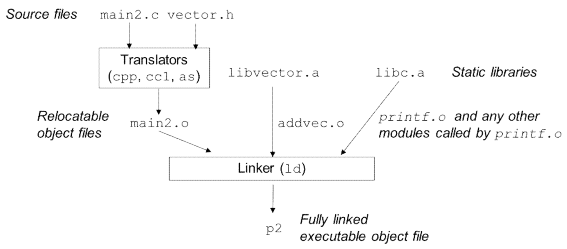
Linking with Static Libraries

Related library functions can be compiled into separate object modules and then packaged in a single static library file.

unix> gcc main.c /usr/lib/libm.a /usr/lib/libc.a

Contrast this approach with a separate library file for each library function or a single library file for all library functions.

Example: Static Library



Resolving references using Static Libraries

Linker scans relocatable object files and archives left to right as specified in the command line.

Linker maintains:

Set E: relocatable object files
Set U: unresolved symbols
Set D: defined symbols so far

Initially sets E, U , D are empty.

Resolving references using Static Libraries

> gcc main.c f1 f2 f3 …

- Each input object file f is added to E and the sets U, D are updated to reflect the symbol definitions and references in f.
- Each input archive file's member *m* is added to E if it resolves a reference in U. Sets U, D are updated to reflect symbol definitions and references in *m*.
- If U is non-empty when linker finishes, it prints an error. Otherwise, it merges and relocates object files in E to build output executable file.

Relocation

Relocating sections and symbol definitions: Linker merges all sections of the same type into a new aggregate section of the same type.

Relocating symbol references within sections: Linker modifies every symbol reference in the bodies of the code and data sections so that they point to the correct run-time addresses.

Relocation Entries

```
typedef struct {
  int offset; /*offset of the reference to relocate*/
  int symbol:24, /*Symb the ref. should point to*/
        type: 8; /*Relocation type*/
} Elf32_Rel;
```

Relocation Algorithm

```
foreach section s{
        foreach relocation entry r{
            refptr = s + r.offset; /*ptr to reference to be
                                    relocated */
            if(r.type is PC relative){
                    refaddr = ADDR(s) + r.offset;
                        *refptr = (unsigned)
                    (ADDR(r.symbol) + *refptr – refaddr);
            }
            if(r.type is Absolute){
                    *refptr = (unsigned)
                    (ADDR(r.symbol) + *refptr);
            }
        }
}
```

Relocation Algorithm

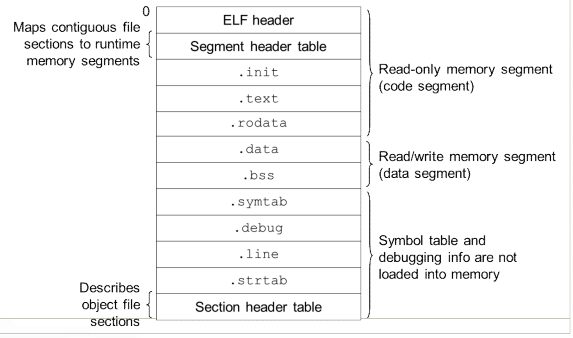
```
foreach section s{
  foreach relocation entry r{
    refptr = s+r.offset; /*ptr to reference to be relocated*/
    if(r.type is PC relative){
      refaddr = ADDR(s) + r.offset;
      *refptr = (unsigned) (ADDR(r.symbol) + *refptr -
                refaddr);
    }
    if(r.type is absolute){
      *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
    }
}
```

Go over example from CSAPP Textbook

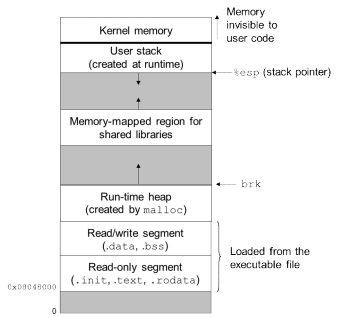
Relocating PC-Relative References

Relocating Absolute References

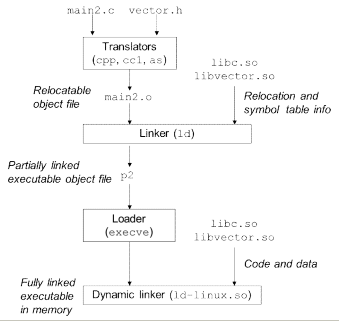
ELF Executable Object File



Loading Executable Object Files



Dynamic Linking with Shared Libraries



Loading and Linking Shared Libraries from Applications

Example program from CSAPP textbook that dynamically loads and links a shared library.

Position Independent Code (PIC)

How do multiple processes share single copy of a program?

An approach to compile library code so that it can be loaded and executed at any address without being modified by the linker.

GCC option –fPIC

Compiler creates a table called "Global Offset Table (GOT)" at the beginning of the data segment.

PIC Data References

```
      call L1
L1:  popl   %ebx       /*ebx contains current PC*/
     addl   $VAROFF, %ebx  /*ebx->GOT entry
                             for var*/
     movl  (%ebx), %eax      /* reference indirect
     movl  (%eax), %eax        through GOT */
```

Performance Disadvantage: Each global memory reference now requires five instructions instead of one.

PIC Function Calls

```
      call L1
L1:  popl   %ebx        /*ebx contains the
                         current PC*/
     addl   $PROCOFF, %ebx     /*ebx ->GOT
                                entry for proc*/
     call    *(%ebx)          /* call indirect through
                               GOT*/
```

Performance Disadvantage: Each procedure call requires three additional instructions.

PIC Function Calls: Optimization

Lazy binding of PIC function calls using a Procedure Linkage Table.

After first call, each subsequent call needs only one instruction and one memory reference.

Go over example in CSAPP textbook.

Tools for Manipulating Object Files

AR: Creates static libraries
STRINGS: lists all printable strings contained in object file.
STRIP: Deletes symbol table info from object file.
NM: Lists symbols defined in symbol table of an object file.
SIZE: Lists the names and sizes of the sections in an object file.
READELF: Displays the complete structure of an object file.
OBJDUMP: Displays information in an object file.
LDD: Lists shared libraries needed by an executable.