CS 354 - Machine Organization & Programming
Tuesday May 2, and Thursday May 4, 2023

Course Evals

https://aefis.wisc.edu Course: CS354 Instructor: DEPPELER

Final Exam -Wednesday May 10th, 2:45 PM - 4:45 PM

See Exams Page for more information -- cummulative with focus on material since E2.
Exam Room information sent via email -- Bring to lecture and fill out scantron correctly
Arrive early if possible with UW ID and #2 pencils. See exam info on course web site.

Homework hw8: DUE on Monday May 1st

Homework hw9: DUE on Wednesday May 3rd

Project p6: Due on last day of classes (NO LATE PERIOD or OOPS). If you plan on getting help in labs, be sure to bring your own laptop in case there is no workstation available.

Last Week

Meet Signals Three Phases of Signaling Processes IDs and Groups Sending Signals Receiving Signals	Issues with Multiple Signals Forward Declaration Multifile Coding Multifile Compilation Makefiles

This Week

Makefiles (from last week) Relocatable Object Files Static Linking Linker Symbols Linker Symbol Table Symbol Resolution	Resolving Globals Symbol Relocation Executable Object File Loader What's next? take OS cs537 as soon as possible and Compilers cs536, too!
Next Week: FINAL EXAM	

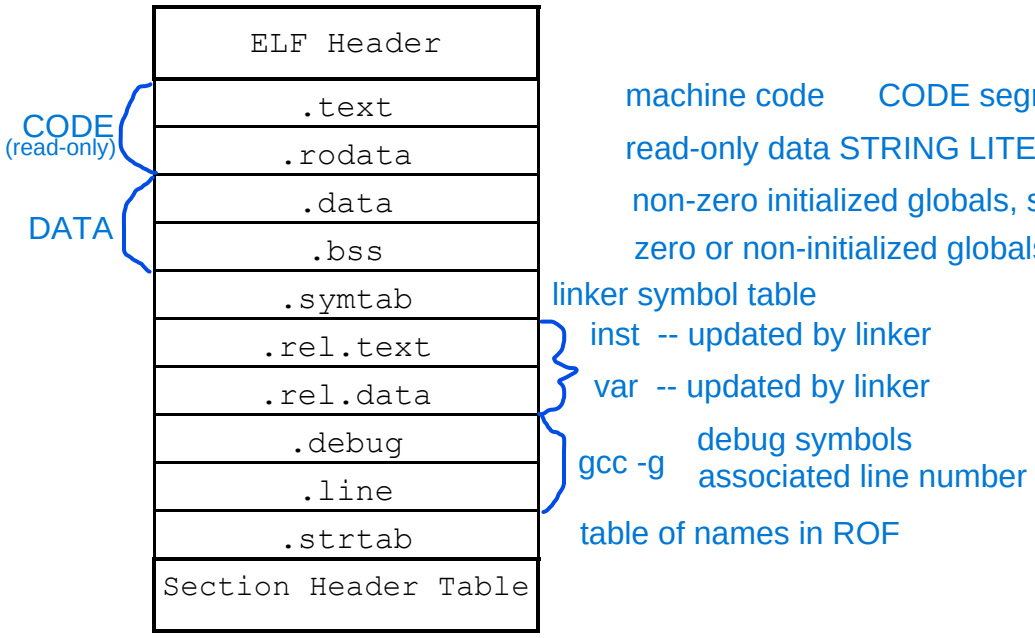
Relocatable Object Files (ROFs)

What? A *relocatable object* file is

- an object file (binary CODE & DATA)
- in Executable and Linkable Format = easy for linker

Executable and Linkable Format (ELF) - Linux

**static linking will be assumed to be only kind of linking on exam



ELF Header general info

ELF header size, object file type (ROF SOF EOF)
offset SHT, size SHT and number of entries
also! word-size, byte-ordering "little endian"

Section Header Table (SHT) location and size of each section

Static Linking

What? *Static linking*

generate a complete EOF without var or function identifiers

	static	vs.	dynamic
executable size:	larger		smaller
library code:	smaller		not included, dynamically linked

How?

*ALL language translation has been done
Need only to combine ROF and SOF into EOF

→ What issues arise from combining ROFs?

1. variable and function identifiers need to be checked for O.D.R. (one definition rule)
   (can't have vars, fcns of same name in linked obj files)
   symbol RESOLUTION

2. variable and function identifiers need to be replaced with their "new" address, offset
   symbol RELOCATION

Making Things Private

→ Are functions and global variables only in a source file actually private if they're not in the correspondng header file? NO
   can still be declared in other files and accessed

→ How do you make them truly private? Yes, declare them static

Linker Symbols

What?

Symbols identifiers for vars & fcns in source files

Linker Symbols managed for and by Linker

→ Which kinds of variables need linker symbols?
those allocated in DATA segment (.data and .bss)

NO 1. local variables

YES 2. `static` local variables DATA

NO 3. parameter variables

YES 4. global variables DATA

YES 5. `static` global variables DATA

YES 6. `extern` global variables DATA
defined elsewhere (in different file)

→ Which kinds of functions need linker symbols?

ALL functions need linker symbols for relocation

1. extern needed for RESOLUTION
must connect with defn

2. non-static resolution needed and need O.D.R
and must relocate and update references

3. static (decl/defn)
compiler can resolve as private fcn
must relocate

Linker Symbol Table

What? The linker ***symbol table*** is

- built by assembler, using symbols exported by compiler
- represented as array of ELF symbols

`ELF_Symbol` Data Members and their Use

`int name` byte offset to name in .strtab

`int value` symbols addr

if ROF offset from begin of section

if EOF virtual address

`int size` # of bytes for mem alloc.

`char type:4` data (OBJECT), (FUNC), (NO TYPE)

`binding:4` GLOBAL or LOCAL

`char section` (Ndx) index into Section Header Table

pseudo sections: 1 .text 2 .rodata 3 .data 4 .bss

ABS:olute must not relocate

UND:efined extern (referenced here defn wlsewhere)

COM:mon .bss symbols, loader must make space for

These change meaning

`value` alignment req

`size` min # bytes

Example

%readelf -s <ROF or EOF filename>

```
Num:  Value Size  Type     Bind     Ot Ndx   Name
1 - 7 not shown
8:    0     4     OBJECT   GLOBAL   0  3     bufp0
9:    0     0     NOTYPE   GLOBAL   0  UND   buf
10:   0     39    FUNC     GLOBAL   0  1     swap
11:   4     4     OBJECT   GLOBAL   0  COM   bufp1
```

.bss

→ Is `bufp0` initialized? Yes .data section

→ Was `buf` defined in the source file or declared `extern`?

→ What is the function's name? swap

→ What is the alignment and size of `bufp1`?

multiple of 4 min 4 bytes

Symbol Resolution

What? *Symbol resolution*

- check O.D.R. (one definition rule)

- work done by compiler and finished by linker

Compiler's Resolution Work

resolves local var symbols

- locals check O.D.R.

  a.x b.x

  `static` locals in DATA segment, can ensure unique name

- globals leave for linker

  `static` globals can check O.D.R. in file, since private

✵ *If a global symbol is only declared in this source file*
the compiler assumes in another file

Linker's Resolution Work

resolves global symbols across multiple files

- `static` locals linker does not resolve

- globals check O.D.R. across all object files

✵ *If a global symbol is not defined or is multiply defined* LINKER ERROR

Resolving Globals

Globals - ODR=One Definition Rule

**static and extern can both be used to resolve error, use static if you want different value for variable than other variables of same name

main.c

```
extern int m;
       int n = 11;
static short o;

       extern int x;
       int y;
       static int z = 66;

       //code continues...
```

funit1.c

```
       int m = 22;
static int n;
static int o;

       int x;
       static int y = 33;
       static int z = 77;

       //code continues...
```

funit2.c

```
extern int m;                  linker error
       extern int n;           linker error
static char o;                 linker error

       static int x = 33;      No linker error
       static int y;           No linker error
       int z;                  No linker error

       //code continues...
```

- *What happens if multiple definitions of a variable identifier?* LINKER ERROR
- *Use* `extern` *to indicate when* global var is decl only
- *Use* `static` *to indicate when* global var is private

TEXTBOOK and OLD NOTES may describe old rules for resolving globals

~~**Strong and Weak Symbols**~~

~~strong: function definitions and initialized global variables~~

~~weak: function declarations and uninitialized global variables~~

→ ~~Which code statements above correspond to strong symbols?~~

~~**Rules for Resolving Globals**~~

→ ~~Which code statements above correspond to definitions?~~
~~Recall: `extern` is only a declaration~~

~~1. Multiple strong symbols~~
~~linker errorRecall: `static` makes a global private, i.e., only visible within its source file)~~

~~2. Given one strong symbol and one or more weak symbols,~~

~~3.Given only weak symbols,~~
~~dangerous with different types,to avoid use gcc -fno-common~~

Symbol Relocation

What? *Symbol relocation* combine ROF into EOF

How?

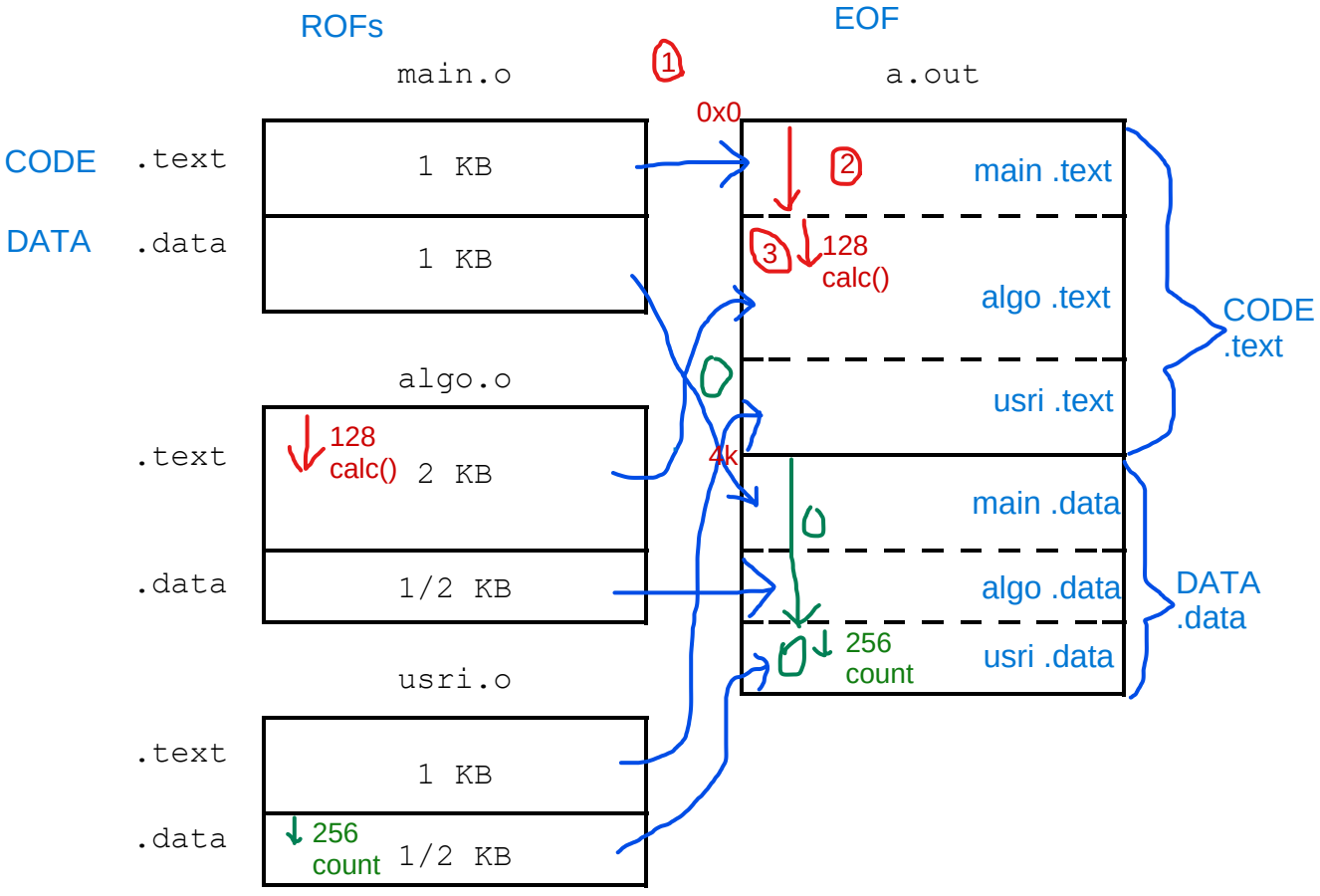
1. Merges the same sections of ROFs / SOFs into aggregate section of type

2. Assigns virtual addresses to each aggregate section and each global definition

3. Updates symbol references as listed in .rel.text .rel.deata

Example

Consider the .text and .data sections of 3 object files below combined into an executable:



address = 1 start addr of section + 2 offset to subsection 3 offset with subesction

calc() = 0 + 1024 + 128 = 1152

count = 4096 + 1024 + 512 + 256 = 5888

Excutable Object File (EOF)

What? An EOF, like an ROF, is file containing object code in ELF

prod by linker, and loaded in memory by loader

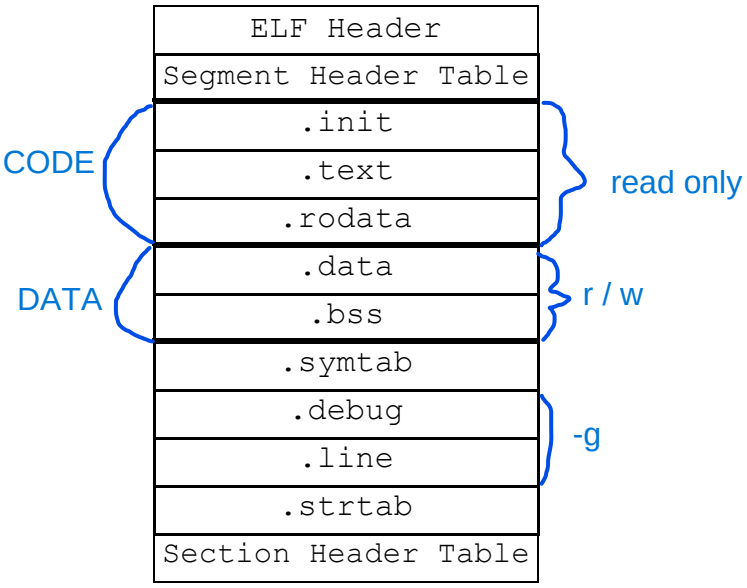
Executable and Linkable Format almost same as ROF

ELF Header

+Entry Point = 1st instruction

+ Segment Header Table info for each segment

offset to section
alignment
page size
. . .



→ Why aren't there relocation sections (.rel.text or .rel.data) in EOF?

all symbols replaced with virtual addresses

➢ Why is the data segment's size in memory larger than its size in the EOF?

because LOADER allocates space for symbols in .bss
when LOADED into memory

Loader

What? The *loader*

- kernel code to start program
- can be invoked by any linux process

Loading

1. copy CODE & DATA segment from EOF into mem
2. Start prog running by jumping to entry point

Execution - the final story

1. shell creates child process, fork()
2. child process invokes LOADER, execve
3. loader create a new runtime image
   a. delete current segments
   b. create current segments
   c. heap and stack
   d. EOF's CODE & DATA are mapped to that space
4. loader

```
setup     call __libc_init_first
          call _init      .init
          call atexit     .text
program   call main
teardown  call _exit
```