CS 354 - Machine Organization & Programming
Tuesday December 12th, 2023

Course Evals

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

Final Exam - Saturday Dec 16th, 7:25 PM - 9:25 PM

Your final exam room has been sent to you via email.
You must attend the exam room as assigned in the email you receive.
Arrive early if possible with UW ID and #2 pencils. See additional exam info on course web site.

All office hours, TA consulting, and Peer Mentoring end on Wed December 13th

Homework hw9: DUE on or before Wednesday Dec 13 (NO LATE DAY)

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

Learning Objectives

- understand and describe how compiler resolves symbols across multiple source files
- understand and describe why relocation is necessary and how it occurs
- understand and describe what the Loader is and does

This Week

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 Watch your email for your exam room assignment. All students must take the final exam in their assigned final exam room. **Students with accomodations should receive email with exam date/time/venue by 12/6.**	

Resolving Globals

if trying to use only w/in file use static

Confusing Globals

```
        main.c                     fun1.c                     fun2.c
       def                   def & init                 def                           Linker Error
extern  int m;                     int m = 22;          extern int m;                     Yes
        int n = 11;         extern int n;                      extern int n;              Yes
static  short o;            static int o;               static char o;                    Yes

                                   def                  priv  global
        extern int x; · · · · ·>  int x;                      static int x = 33;          No
        int y;                     static int y = 33;          static int y;              No
        static int z = 66;         static int z = 77;          int z;                     No

        //code continues...        //code continues...         //code continues...
```

- *What happens if multiple definitions of an identifier exist?* Linker error
- *Use* `extern` *to clearly indicate when* global var is decl only
- *Use* `static` *to clearly indicate when* global var is private (file specific)

TEXTBOOK and OLD NOTES describe old rules for resolving global variables.

~~Strong and Weak Symbols (no such thing any more)~~

~~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? type & name and NO EXTERN
Recall: `extern` is only a declaration
Note: extern vars must be defined in another file, otherwise undefined symbol linker error

1. Multiple symbol defs in public gobal scope are not allowed
   linker error -> mult defined symbol
   Recall: `static` makes a global private, i.e., only visible within its source file)

~~2. Given one strong symbol and one or more weak symbols,~~
   ~~declare weak symbols with extern choose the strong.~~

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

Symbol Relocation

What? *Symbol relocation* combine ROFs into EOF

How?

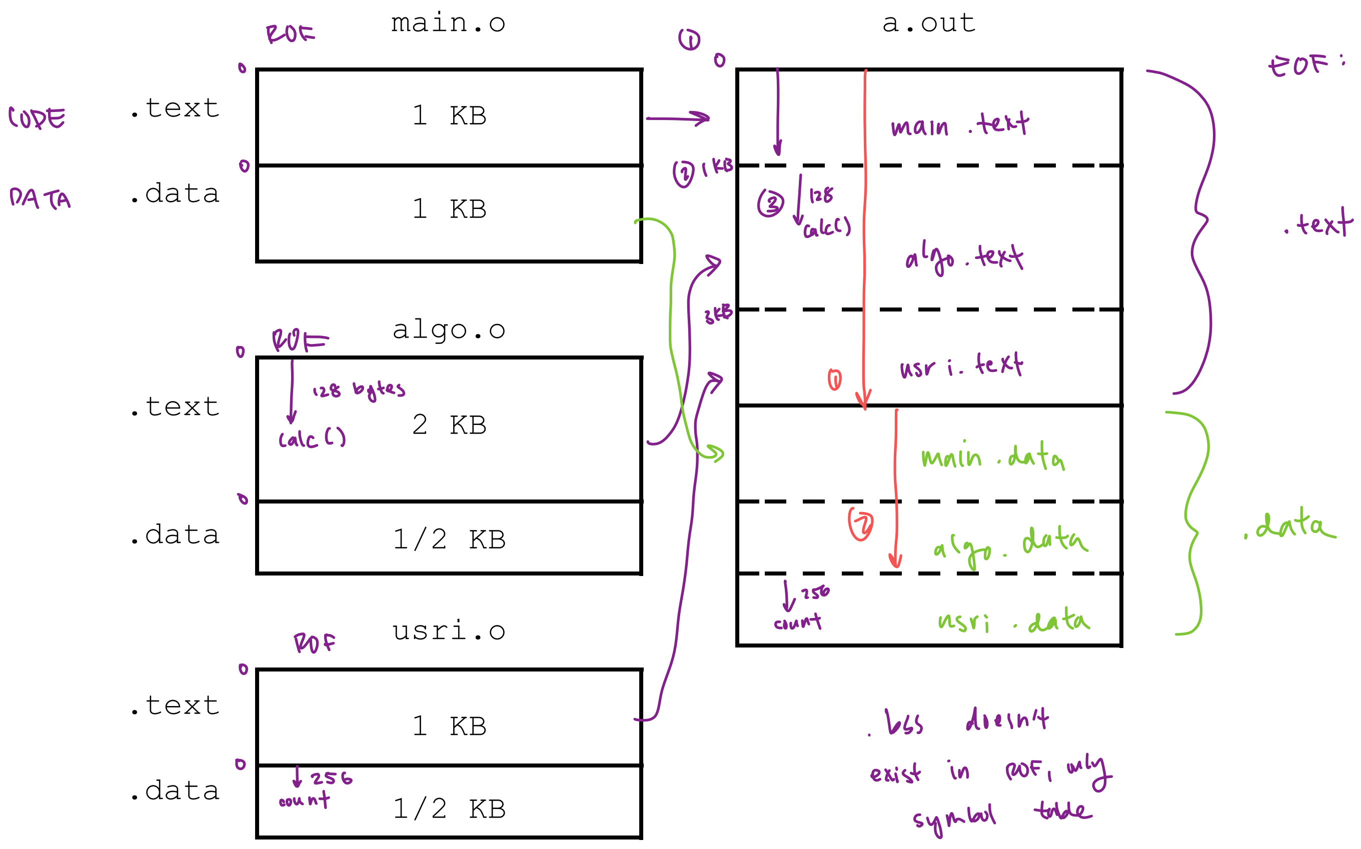
1. Merges the same sections of each ROF into aggregate section of EOF

2. Assigns virtual addresses to each aggregate sections and global defn.

3. Updates symbol references as listed in .rel.text & .rel.data

Example

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



.bss doesn't exist in ROF, only symbol table

address = ① start addr of section + ② offset to subsection + ③ offset w/in subsection

calc() = 0 + 1024 + 128 = 1152

count = 4096 + (1024 + 512) + 256 = 5888

— similar to ROF

Excutable Object File (EOF)

What? An EOF, like an ROF, is produced by the linker, contains binary code + data

✱ that can be loaded into memory and run

Executable and Linkable Format

same format as ROF w/ additions needed by LOADER

ELF Header

+ Entry Point "start addr" addr of 1st inst

+ Segment Header Table info for each segment loaded into mem when run

offset in file for each section

alignment (~4K for page)

size in file and size in mem

run-time permission (r w x)

ELF Header	+ entry point
Segment Header Table	
.init	prog initialization { CLA's STACK HEAP
.text	CODE segment read, execute (r x)
.rodata	strings literals
.data	DATA - globals, static locals read, write (r w)
.bss	
.symtab	- desc of symbols (.bss)
.debug	only if -g
.line	
.strtab	- names of things in sym tab
Section Header Table	

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

all symbols have been replaced w/ their virtual addr

b/c we are using static linking

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

b/c .bss is place holder in EOF

but loader must alloc mem at load time

for all .bss symbols based on symbol table description

Loader

What? The *loader*

- the kernel code that starts a program executing
- can be invoked by any linux program, execve() syscall

Loading

1. copies CODE & DATA - segments into memory from EOF

   expands .bss

2. starts program executing : set up + jump to entry point

Execution - the final story

1. shell creates child process w/ fork()
2. child process invokes loader w/ execve()
3. loader creates a new run-time image
   a. deletes curr segments CODE, DATA, HEAP, STACK
   b. creates new segments CODE, DATA
   c. Heap & stack are created
   d. EOF's CODE & DATA are mapped in page table into pag size chunks
4. loader jumps to start addr

```
→ call __libc_init_first
  call _init
→ call atexit
→ call main
  call _exit
```

loader setup

"AT EXIT"

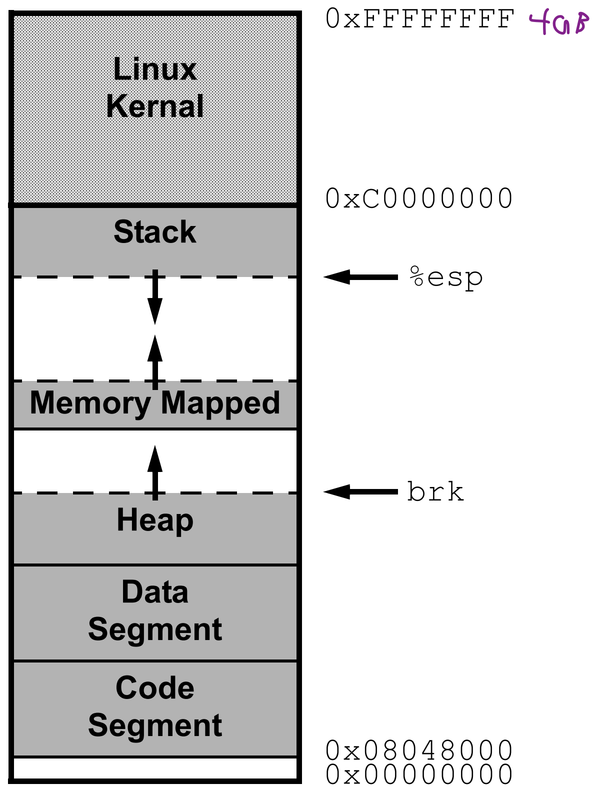
our program

exit(0);

Entry Point

1st addr of our code

ends w/ return 0



0xFFFFFFFF
4GB
Linux Kernal
0xC0000000
Stack
%esp
Memory Mapped
brk
Heap
Data Segment
Code Segment
0x08048000
0x00000000