CS 354 - Machine Organization & Programming
Tuesday Dec 5th, and Thursday Dec 7, 2023

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

Homework hw8: DUE on or before Monday December 4

Homework hw9: DUE on or before Wednesday December 13

Project p6: Due on last day of classes. ***NOTE: There is no LATE day or OOPS point available for p6. All work must be submitted before 11:59 pm Dec 13th. Please complete p6 this week as all support is very busy last week of classes.***

Learning Objectives

- able to describe how multiple signals are received "handled"
- able to describe purpose and how to use forward declarations
- able to explain difference between declaration and definition in resolving symbols
- know how to declare variable without defining and when and why; reserved word "extern"
- be able to code and compile a project across multiple source files, use make and Makefile
- able to create, read, and interpret a Relocatable Object File, ROFs
- name and describe sections of object files
- understand and describe static linking of multiple files into a single Executable Object File
- understand and describe how compiler resolves symbols across multiple source files

This Week

Issues with Multiple Signals NOT IN FINAL Forward Declaration Multifile Coding Multifile Compilation Makefiles	Relocatable Object Files Static Linking Linker Symbols Linker Symbol Table Symbol Resolution
Next Week: Resolving Globals Symbol Relocation Executable Object File Loader What's next? take OS cs537 as soon as possible and Compilers cs536, too!	**Read:** B&O 7.1 Compiler Drivers 7.2 Static Linking 7.3 Object Files 7.4 Relocatable Object Files 7.5 Symbols and Symbols Tables 7.6 Symbol Resolution 7.7 Relocation

Issues with Multiple Signals

What? Multiple signals of the same type as well as those of different types

can be sent during same period

Some Issues

→ Can a signal handler be interrupted by other signals? yes, but

Linux: signals of same type do not interrupt, they become pending

❋ *Block any signals* that you don't want to interrupt

sig empty set (& sa, sa_mask) // blocks all

sig fill set

sig add set sig del set sig is member

→ Can a system call be interrupted by a signal? yes, for *slow system calls* which can take a long time

- such sys calls ret imm. w/ value EINTR, not ready yet
- sa.sa_flags = SA_RESTART; // restart sys call
- sleep() cannot be restart

→ Does the system queue multiple standard signals of the same type for a process? NO

the bit vector can't keep count of duplicates of the same type

❋ *Your signal handler shouldn't assume* that signal was sent only once

Real-time Signals

Linux has 33 additional application defined signals

- they include an int or ptr in their message
- Multiple signals of same type are queued in order delivered
- Multiple signals of different types are recieved from low to high signal #

Forward Declaration

What? Forward declaration tells compiler about certain attributes of an identifier before it is fully defined

❋ *Recall, C requires that an identifier* be declared before it is used

Why?

- one pass compiler (gcc) can ensure identifier exists and used correctly
- large programs can be divided into separate func units that can be indept. compiled (p3)
- mutual recursion a → b → c → a

Declaration vs. Definition

declaring tells compiler about

variables: name, type

functions: return type, name, parameter types

defining provides full detail

variables: where in memory allocated (DATA, STACK

functions: function's body "block of code"

❋ *Variable declarations* usually declare and define

```
void f(){
  int i = 11;          // STACK   declare, defined, initialize
  static int j;        //         declare, defined, not init
```

❋ *A variable is proceeded with* extern is NOT defined

```
extern char * title;   // tells compiler it will exist
```

Multifile Coding

What? Multifile coding divides program into functional units

each coded w/ its own header (.h) and source (.c)

Header File (filename.h) - "public" interface

contains things you intend to share

mainly func. declarations

but also defns of types, consts, macros

recall **heapAlloc.h** from project p3:

```
#ifndef __heapAlloc_h__
#define __heapAlloc_h__

int   initHeap(int sizeOfRegion);
void* allocHeap(int size);
int   freeHeap(void *ptr);
void  dumpMem();

#endif // __heapAlloc_h__
```

public func decl

what funcs are avail for their heap allocer

- *An identifier* can only be defined once in global scope

ODR: One Def Rule

#include guard: prevents mult. inclusion of same header file

prevents ODR from mult inclusion of same identifiers

Source File (filename.c) - "private" implementation

- must include defn of identifiers in header file
- also includes things you don't intend to share

recall **heapAlloc.c** from project p3:

```
#include <unistd.h>
. . .
#include "heapAlloc.h"

typedef struct blockHeader {
   int size_status;
} blockHeader;

blockHeader *heapStart = NULL;

void* allocHeap(int size) { . . . }
int   freeHeap(void *ptr) { . . . }
int   initHeap(int sizeOfRegion) { . . . }
void  dumpMem() {. . . }
```

src file includes its header

struct

global

func defn

"private"

Multifile Compilation

gcc Compiler Driver directs all tools needed to build executable

p1 !

main.c → preprocessor — cpp — removes comments does pp directives

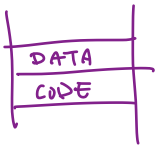
main.i → compiler — cc — translates C → asm

main.s → assembler — as — translates asm → mc — ROF, SOF

main.o → linker — ld — combines all .o files into EOF

main — EOF

Object Files contain binary code and binary data



relocatable object file (ROF) prod by assembler

can be combined by linker w/ other ROF, SOF

executable object file (EOF) prod by linker

can be loaded into memory by the loader and run

shared object file (SOF) prod by assembler

can be loaded into memory and linked dynamically at loading or execution

Compiling All at Once

```
gcc align.c heapAlloc.c  -o align
```

EOF align

Compiling Separately

```
gcc -c align.c
gcc -c heapAlloc.c
gcc align.o heapAlloc.o -o align
```

→ cpp → cc → as — produces align.o ROF

→ cpp → cc → as — produces heapAlloc.o ROF

→ ld — produces align EOF

link to create EOF

- *Compiling separately is* more efficient and easier to manage

Makefiles

cmake - generates make files

What? Makefiles are

- text files named Make file
- used w/ "make" command

Why?

- convenience - specifies how to build program
- efficiency - only build what's necessary

Rules have form!

<target> : files target depends on
<tab> <comands to make target>

Example

```
#simplified p3 Makefile
align: align.o heapAlloc.o
	gcc align.o heapAlloc.o -o align
align.o: align.c
	gcc -c align.c
heapAlloc.o: heapAlloc.c heapAlloc.h
	gcc -c heapAlloc.c
clean:
	rm *.o
	rm align
```

ld to align EOF

cpp → cc → as to create align.o

Using

```
$ls
align.c  Makefile  heapAlloc.c  heapAlloc.h
$make
gcc -c align.c
gcc -c heapAlloc.c
gcc align.o heapAlloc.o -o align
$ls
align  align.c  align.o  Makefile  heapAlloc.c  heapAlloc.h  heapAlloc.o
$rm heapAlloc.o
rm: remove regular file 'heapAlloc.o'? y
$make
gcc -c heapAlloc.c
gcc align.o heapAlloc.o -o align
$make heapAlloc.o
make: 'heapAlloc.o' is up to date.
$make clean
rm *.o
rm align
$ls
align.c Makefile heapAlloc.c heapAlloc.h
```

EOF src ROF src ROF

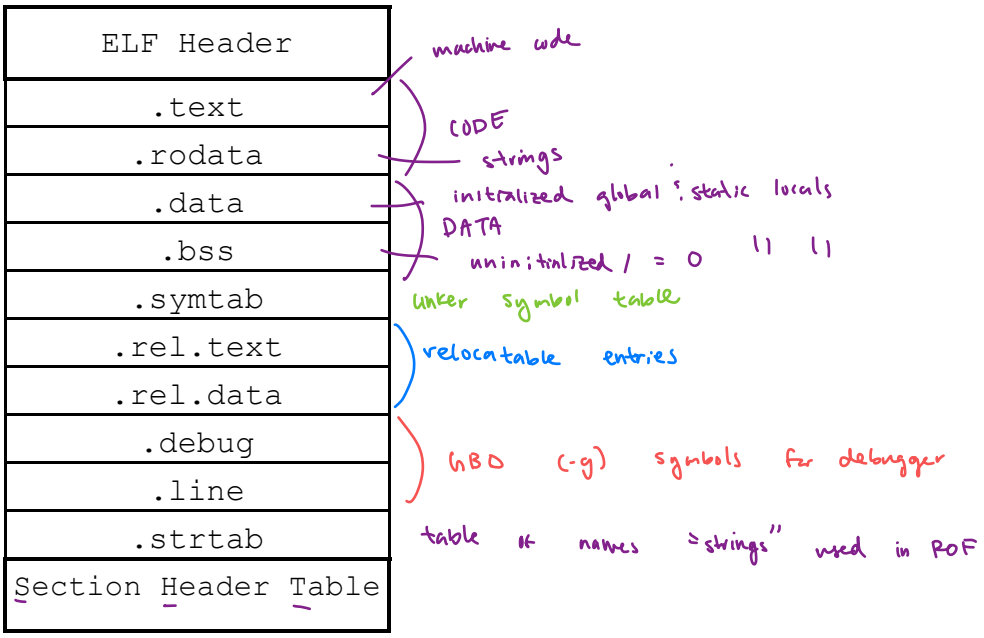
heap Alloc.o

Relocatable Object Files (ROFs)

What? A *relocatable object* file is

- an object file .o contains binary code & data
- in Executable and Linkable Format (easy for linker)

Executable and Linkable Format (ELF)



ELF Header — general info

ELF header size, file type ROF, SOF, EOF

offset to SHT, # entries in SHT

arch info: word size, byte ordering

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

TOC for sections

Static Linking

What? *Static linking* generates a complete EOF w/ no vars or func identifiers

	static	vs.	dynamic
executable size:	larger		smaller
library code:	smaller		not included - dynamically linked at LOAD or RUNTIME

How?

All language translation has been done (cpp, cc, as)

only need to combine ROF / SOF → EOF

→ What issues arise from combining ROFs?

1. variable and function identifiers need to be checked for O.D.R.

2. variable and function identifiers need to be replaced w/ ther address

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! they can still be accessed by other src files

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

Linker Symbols

What?

Symbols are identifiers used for vars and funcs in src code

Linker Symbols are symbols managed by linker

→ Which kinds of variables need linker symbols? those alloc in DATA
.data .bss

NO 1. local variables local scope STACK

YES 2. `static` local variables local scope - in DATA seg - reloc in this file only

NO 3. parameter variables STACK in caller's SF

YES 4. global variables global scope - reloc and update code from all ROF

YES 5. `static` global variables "private" global - reloc w/in this file only

YES 6. `extern` global variables global scope - must update in this file

→ Which kinds of functions need linker symbols?

All funcs for relocation, likely also for resolution

1. extern funcs (decl only) - linker must find defn and replace w/ location

2. non-static funcs: (need decl & defn)
   - linker may need to connect to other ROF

3. static funcs: (decl & defn)
   - possibly no resolution req.d
   - reloc update to code in this file

Linker Symbol Table

What? The linker *symbol table* is

- built by assembler using symbols exported by compiler
- represented as an array of ELF_Symbol structures

`ELF_Symbol` Data Members and their Use

code/link/elfstructs.c

```
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, section, or src file name (4 bits) */
         binding:4;/* Local or global (4 bits) */
    char reserved; /* Unused */
    char section;  /* Section header index, ABS, UNDEF, */
                   /* Or COMMON  */
} Elf_Symbol;
```

string name · EOF · ROF · "mem alloc size" · object · NOTYPE · extern symbol · COM · .bss · value = align · size = min size · data reloc

code/link/elfstructs.c

readelf -s < ROF or EOF file name >

NDX	section
1	.text
2	.rodata
3	.data
4	.bss

Example

section index

```
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
```

extern

→ Is `bufp0` initialized? Yes, .data is initialized

→ Was `buf` defined in the source file or declared `extern`? extern, b/c section is UND

→ What is the function's name? swap

→ What is the alignment and size of `bufp1`? 4 and 4

value → alignment = 4 bytes

size → min size = 4 bytes

① Symbol Resolution

What? *Symbol resolution*

- checks O.D.R for each var and func symbol
- work is divided by compiler & linker

Compiler's Resolution Work

(local)
resolve symbols in one source file at a time

- locals check ODR

  `static` locals also ensure that each has unique name for linker
  src1.f src2.f

- globals leaves for linker to resolve

  `static` globals check ODR since private to this src file

❋ *If a global symbol is only declared in this source file* → "extern" the compiler assumes it is defined in another file

Linker's Resolution Work

resolves global symbols across mult object files

- `static` locals - linker does NOT resolve, but will relocate w/ .data
- globals

❋ *If a global symbol is not defined or is multiply defined*