CS 354 - Machine Organization & Programming
Tuesday Dec 3rd, and Thursday Dec 5, 2024

Course Evaluations: https://aefis.wisc.edu Course: CS354 Instructor: DEPPELER|MAHMOOD

Homework hw8: DUE on or before Monday December 9

Homework hw9: DUE on or before Wednesday December 11

Project p6: Due on or before last day of classes Wednesday December 11. ***NOTE: There is no LATE day or OOPS point available for p6. All work must be submitted before 11:59 pm Dec 11th. 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 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 that other signals are sent
and even while a signal handler is running.

Some Issues

→ Can a signal handler be interrupted by other signals?YES, but...
Linux signals of the same type as running signal handler don't interrupt.
Instead, they become pending.

❋ *Block any signals you don't want to interrupt your handler:*

```
sigemptyset(&sa.sa_mask); //blocks all signals
      sigfillset(&sa.sa_mask); //enables all signals
      sigaddset/sigdelset/sigismember(&sa.sa_mask, signum)
```

→ Can a system call be interrupted by a signal?YES for

slow system calls potentially take a long time, e.g., read()-scanf / write() - printf

Such system calls return immediately with an error condition

```
sa.sa_flags = SA_RESTART;
```

NOTE: sleep() CANNOT be restarted

→ Does the system queue multiple standard signals of the same type for a process?NO
Bit vector can't keep a count of duplicates.
Instead they're ignored.

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

Real-time Signals

Linux has 33 additional application defined signals.

- They can include an integer or pointer in their message.
- Multiple signals of same type are queued in order delivered.
- Multiple signals of different types are received from low to high signal number

Forward Declaration

What? Forward declaration

tells the compiler about certain attributes of an identifer before it is fully defined

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

Why?

- one pass compiler (gcc) can then ensure the identifier exists and is correctly used
- large programs can be divided into separate functional units

that can be independently compiled

- mutual recursion is possible

Declaration vs. Definition

declaring tells the compiler about

variables:

functions:

defining provides the full details

variables:

functions:

- *Variable declarations usually both declare and define.*

```
void f(){
   int i = 11;
   static int j;
```

- *A variable proceeded with*

Multifile Coding

What?

divide programs into functional units, each coded with its own header file and source file

Header File (filename.h) - “public” interface

contains things you intend to sharemainly function declarations

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__
```

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

#include guard: prevents multiple inclusion of same header file

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

Must include definitions of things declared in its header file.

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() {. . . }
```

Multifile Compilation

gcc Compiler Driver

directs all the tools needed to BUILD an executable from source code

Filename	-> build step	tool name	description of work or result
main	preprocessor		removes comments, does pp directives
main	compiler		translates C to ASM
main	assembler		translates ASM to MC (ROF)
main	linker		combines R/SOF's into EOF
main			the EOF

Object Files

contain binary code and binary data in ELF

relocatable object file (ROF) produced by

executable object file (EOF) produced by

shared object file (SOF) produced by

Compiling All at Once (gcc does it all to create EOF)

```
gcc align.c heapAlloc.c  -o align      produces EOF named align
```

Compile Separately (gcc builds individual ROF)

```
gcc -c align.cproduces align.oROF
gcc -c heapAlloc.cproduces heapAlloc.o ROF
gcc align.o heapAlloc.o -o alignproduces align EOF
```

❋ *Compiling separately is more efficient and easier to manage.*

Makefiles

What? Makefiles are

- text files named m/Makefile that have rules
- used with “make” command

Why?

- convenience - specifies how to build a program
- efficiency - only builds what’s necessary using rules and file dates

Rules have target (name), dependencies (files), instructions (commands) in this form:

<target>: <files the target depends on>
<tab><command(s) for making target>

Example

```
#simplified p3 Makefile
align: align.o heapAlloc.oRule 1: how to make align EOF
     gcc align.o heapAlloc.o -o align
align.o: align.cRule 2: how to make align.o ROF
     gcc -c align.c
heapAlloc.o: heapAlloc.c heapAlloc.hRule 3: how to make heapAlloc.o ROF
     gcc -c heapAlloc.c
clean:Rule 4: delete OFs to allow build EOF from scrtch
     rm *.o
     rm align
```

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
```

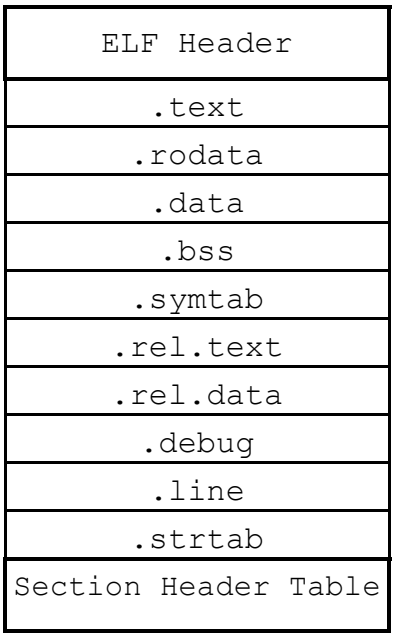
Relocatable Object Files (ROFs)

What? A *relocatable object* file is (aka object file)

- a file with ".o" extension containing object code - the binary instructions and data
- in a format that the linker can easily combined with other ROFs

Executable and Linkable Format (ELF)

Object file format used by Linux [layouts vary by OS]



.text/.rodata/.data/.bss all start at address 0x0

machine code
Read Only data = string literals, switch jump tables
init to non-0 global and static local vars
uninit global and static local vars ONLY A PLACE HOLDEF
linker symbol table - global vars & extern funcs

if gcc -g then debug sym tab + locals & typedefs
maps src with machine code
table of names used in ROF

ELF Header
contains general info:
ELF header size, object file type (ROF,SOF,EOF),
offset to SHT, size of SHT and num entriesin SHT

also contains arch info:
word size, byte ordering, machine type

Section Header Table (SHT)
contains location and size of each section in the object file

Static Linking

What? *Static linking*

generates a complete EOF with no var or func identifiers remaining in the OF

static vs. dynamic

executable size:

library code:

How?

Note: All language translation has already been done (cc and as).

Need only to combine R/SOFs into an EOF.

→ What issues arise from combining ROFs?

1. variable and function identifiers need to be checked for having exactly one definition ODR

2. variable and function identifiers need to be replaced with their addresses

Making Things Private

→ Are functions and global variables only in a source file actually private if they're not in the correspondng header file?

→ How do you make them truly private?

Linker Symbols

What?

Symbols are identifiers used for variables and functions in a source code

Linker Symbols are symbols managed by the linker

→ Which kinds of variables need linker symbols?

those that are allocated in the data segment

1. local variables
2. `static` local variables
3. parameter variables
4. global variables
5. `static` global variables
6. `extern` global variables

→ Which kinds of functions need linker symbols?

ALL functions for relocation, likely all for resolution

1. `extern` functions
2. non-`static` functions
3. `static` functions

Linker Symbol Table

What? The linker *symbol table* is

- built by assembler using symbols exported by compiler
- represented as an array are `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;
```

code/link/elfstructs.c

Example

```
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
```

→ Is `bufp0` initialized?Yes - section is Ndx 3 its data OBJECT in .data

→ Was `buf` defined in the source file or declared `extern`?extern - section is UND

→ What is the function's name? `swap` - type is FUNC, section is Ndx 1

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

Symbol Resolution

What? *Symbol resolution*

- checks ODR
- work is divided between the compiler and the linker

Compiler's Resolution Work

resolves local symbols in one source file at a time

- locals checks ODR

  `static` locals also ensures each has a unique name for the linker

- globals leaves for linker to resolve

  `static` globals can check ODR since private to this source file

✵ *If a global symbol is only declared in this source file the compiler assumes it's defined in another source file.*

Linker's Resolution Work

resolves global symbols across multiple OFs (source files)

- `static` locals - linker doesn't resolve
- globals - checks ODR - only one definition in all R/SOFs

✵ *If a global symbol is not defined or is multiply defined it is a Linker error*