outlineL27-w14TR-student

Wednesday, December 14, 2022 4:01 PM

outlineL27-w14TR-st...

CS 354 - Machine Organization & Programming
Tuesday Dec 6th, and Thursday Dec 8, 2022

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

Homework hw7: DUE on or before Monday December 7th

Homework hw8: DUE on or before Monday December 12

Homework hw9: DUE on or before Wednesday December 14

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 14th. Please complete p6 this week as labs are very busy last week of classes. If you do plan on getting help during last week of classes, be sure to bring your own laptop in case there is no workstation available.***

Last Week

Kinds of Exceptions Transferring Control via Exception Table Exceptions/System Calls in IA-32 & Linux Processes and Context User/Kernel Modes Context Switch Context Switch Example	Meet Signals Three Phases of Signaling Processes IDs and Groups Sending Signals Receiving Signals

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 the same time

Some Issues

→ Can a signal handler be interrupted by other signals?

Yes, but linux sigs of the same type as running becomes pending

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

Sig emptyset(&sa.sa_mask); //block ALL
Sig fillset (&sa.sa_mask); //enables ALL
Sigaddset / sigdelset / sigismember(&sa.sa_mask, signum)

→ Can a system call be interrupted by a signal? Yes, for…

slow system calls Eg. Read-scanf, write-printf

Such syscalls return immediately with EINTR
Sa.sa_flags = SA_RESTART

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

Bit vector cannot count multi sig. they are ignored

✱ *Your signal handler shouldn't assume* That a signal was sent only once

Real-time Signals

Linux has 33 additional APP. Defined signals

- They can include integers or a ptr in their messages
- Multiple signals of same type are queued in order delivered.
- Multiple signals of different types Are received from low to high signed numbers

Forward Declaration

What? Forward declaration

Tells the compiler certain attributes of the identifiers, before they're fully defined

- *Recall, C requires that an identifier* Be declared before it's used

Why?

- One-pass compiler (gcc) can ensure an identifier exists and is used correctly
- Large programs can be divided into separate functional units
  These units can be independently compiled
- Mutual recursion is possible (where one function calls another, and it calls the previous function back)

Declaration vs. Definition

declaring Tells compiler about

variables: Name and type (extern)

functions: Return type, name, parameter list of types

defining Provides full details to the compiler

variables: Where in memory it's allocated

functions: Function's body that defines it

- *Variable declarations* Usually both declare and define

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

//declaration + definition and initialization here

//declaration + definition -> goes to DATA segment

- *A variable is proceeded with* Extern is not defined

Extern char * title; //declared, not defined, not initialized

Multifile Coding

What? Multifile coding — Is dividing program into functional units. Each have own header + source file

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

Contains things you intend to share, mainly function declarations, also definitions for types, CONSTANTS, macro

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 declaration

- *An identifier* — Can be defined only once in global scope (one definition rule) (ODR)

#include guard: Prevents multiple inclusion of the same header file

Which prevents linker errors later

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

Must include definitions of things declared in header file
Includes additional things that 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() {. . . }
```

← system

← src file includes it's header

"private" block header

"private" definition of function

Multifile Compilation

gcc Compiler Driver

src → gcc → EOF

Directs all tools needed to build an executable from source code

main.c → preprocessor cpp removes comments, does pre-processor directives
main.i → compiler cc responsible for translating C to assembly language
main.s → assembler as translates assembly to machine code (ROF)
main.o → linker ld Combines all relocatable object files into EOF — relocatable object file

Object Files Contains binary code and DATA

relocatable object file (ROF) Produced by the assembler, in specific ELF format

Can be combined by linker with other ROFs and SOFs to create a single EOF

executable object file (EOF) produced by the linker

Can be loaded by the LOADER into memory, and run

shared object file (SOF) Produced by assembler

Loaded into memory, and linked dynamically at load time or runtime

Compiling All at Once

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

EOF

Compiling Separately

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

cpp → cc → as produce align.o (ROF)
cpp → cc → as produce heapAlloc.o (ROF)
ld produce align (EOF)

EOF

- *Compiling separately is*
  more efficient and easier to manage, as you don't have to recompile code
  At all times

Makefiles

What? Makefiles are

- Text files named "makefile" that have certain rules (called recipes)
- They are used with the make command

Why?

- They are convenient- specifies exactly how to build a program
- They are efficient - only builds what is necessary using rules and file data

Rules Have required form
<target>: <files target depends on>
[tab]<command(s) for making program>

Example

```
#simplified p3 Makefile  — comment
align: align.o heapAlloc.o
     gcc align.o heapAlloc.o -o align   EOF
align.o: align.c
     gcc -c align.c   (ROF)
heapAlloc.o: heapAlloc.c heapAlloc.h
     gcc -c heapAlloc.c   (ROF)
clean:
     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
```

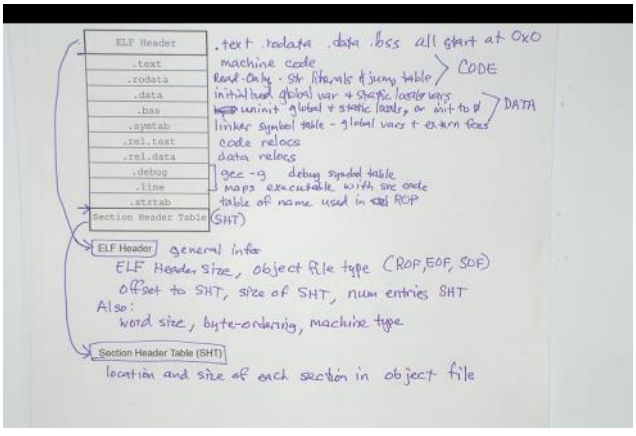

Relocatable Object Files (ROFs)

What? A *relocatable object* file is

- An .o file containing object code - the binary instructions + data
- In a format that linker can use to create EOF

Executable and Linkable Format (ELF)

Object file format used by Linux



Section Header Table (SHT)

Section Header Table (SHT)

Location and size of each section in the object file

Static Linking

What? *Static linking* Generate a complete EOF without vars or function identifiers

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

How?

All language translations have been done (cpp, cc, asm)
Need only to combine R/SOFs into an EOF

→ What issues arise from combining ROFs?

1. variable and function identifiers For one definition rule (ODR)

   RESOLUTION

2. variable and function identifiers Need to be replaced with their addresses where located in EOF

   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! not private - still can be accessed by source files that declare them

→ How do you make them truly private? Make it static

Linker Symbols

What?

Symbols Are identifiers used for variables and functions in a src code

Linker Symbols Are symbols managed by the linker

→ Which kinds of variables need linker symbols? Those allocated in the DATA segment

1. local variables On stack, no
2. `static` local variables Local scope - yes
3. parameter variables On stack - no
4. global variables Global scope - yes
5. `static` global variables Global scope - "private" to src file RELOC, yes
6. `extern` global variables Global scope- defined elsewhere, yes

→ Which kinds of functions need linker symbols?

All functions for relocation, likely all for resolution

1. Extern functions
   Linker must connect funcs in this ROF with def'n in another ROF
2. Non-static functions - resolution
   Linker may need to connect function call in ROF with definition in another ROF
3. Static functions "private"
   possibly no resolution required, since private
   but still need to relocate in EOF

Linker Symbol Table

What? The linker *symbol table* is

- Is built by the assembler using symbols exported by the compiler
- Is represented as an array of ELF symbol "structs"

`ELF_Symbol` Data Members and their Use

`int name` Byte offset into .strtab containing null terminated strings

`int value` Symbols address
- if ROF: Offset from start of it's section
- if EOF: Its virtual address or absolute address

`int size` Number of bytes for this symbols memory allocation

`char type:4` Data (OBJECT), fcn(FUNC), extern (NOTYPE)

`binding:4` Global/local

`char section` Index into section header table (SHT)

pseudo sections:
- ABS:olute: Should not relocate symbol
- UND:efined: Extern - sym reference. Declared here, defined elsewhere
- COM:mon: Symbols that are uninitialized or zero
  - `value`: Now means alignment
  - `size`: Now means min size

Example

section

.text 1
.rodata 2
.data 3

```
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
```

data

→ Is `bufp0` initialized? Yes, because section ndx = 3 which equals data

→ 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`? Align is multiple of 4
Minimum size is 4 bytes

Symbol Resolution ODR

What? *Symbol resolution* Within a single ROF compiler

- Checks ODR across multiple ROFs

- Work is divided btw. Compiler + linker ld

Compiler's Resolution Work

Resolves local symbols in one src file

- locals

  `static` locals Ensure each has a unique name for the linker

  X.add X in add()
  X.swap X in swap()

- globals Leave for linker

  `static` globals Compiler can check ODR, since private to file

❋ *If a global symbol is only declared in this source file*

Compiler assumes it's defined in another

Linker's Resolution Work

Resolve global symbols across multiple O.F.

- `static` locals Linker does not resolve, but it does relocate

- globals
  - Checks O.D.R.
  - - can only be one definition for each symbol across All ROFs

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

Linker error unless -z multi defined