C Introduction

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 8 released today, due next Monday on Gradescope

Switch

It looks like a bunch of labels. And it's going to work exactly like we've seen labels. Where I jumped to a label and then I start executing code.

Not like a function: I go there, run things, then return.

Not like if statement: I do a bunch of conditional jumps.

One thing to note. we have to switch on an integer. That integer is going to be the index into our array of labels. (Indexing into an array of labels and picking which one we want to jump to).

```
    .globl  describe
describe:
    pushq   %rax
    movl    %edi, %eax
    addl    $-1, %eax
    cmpl    $4, %eax
    ja  Label5
    leaq    Text0, %rdi
    leaq    JumpTable, %rcx
    movslq  (%rcx,%rax,4), %rax
    addq    %rcx, %rax
    jmpq    *%rax
Label2:
    leaq    Text1, %rdi
    jmp Label6
Label5:
    leaq    Text4, %rdi
    jmp Label6
Label3:
    leaq    Text2, %rdi
    xorl    %eax, %eax
    callq   puts
Label4:
    leaq    Text3, %rdi
```

→ first parameter (age).

→ The index of an array starts from 0, so we want to map 1~5 to 0~4

→ if index > 4, then jump to Label 5

→ It's preparing the argument for a later call like puts (Text0).

→ Load the address of the jump table into %rcx.

→ Loads a 32-bit offset from the jump table entry indexed by %rax and sign-extends it to 64 bits.
Each entry is 4 bytes long (hence the *4).

→ Adds the base address of the jump table to that offset.

→ indirect jump to the address in %rax.

→ load Text1 and jump to Label 6.

→ load Text4 and jump to Label 6

Switch

```
Label3:
    leaq    Text2, %rdi
    xorl    %eax, %eax
    callq   puts
Label4:
    leaq    Text3, %rdi
Label6:
    xorl    %eax, %eax
    callq   puts
    popq    %rax
    retq
```

empty out eax and outputs.

Switch

in assembly a long type is 32 bites.

```
    .section    .rodata
JumpTable:
    .long   Label6-JumpTable
    .long   Label2-JumpTable
    .long   Label5-JumpTable
    .long   Label3-JumpTable
    .long   Label4-JumpTable

Text0:
    .asciz  "You're one."
Text1:
    .asciz  "You're two."
Text2:
    .asciz  "You're four."
Text3:
    .asciz  "You're four or five."
Text4:
    .asciz  "You're not 1, 2, 4 or 5!"
```

jump table. Why they choose this order ?
I don't know but it doesn't matter.

read only section

Calling Functions

The C code

```
long a = f(23, "yes", 34uL);
```

compiles to

```
movl $23, %edi
leaq label_of_yes_string, %rsi
movq $34, %rdx
callq f
# %rax is "long a" here
```

without respect to how `f` was defined. It is the calling convention, not the type declaration of `f`, that controls this.

Calling Functions

But, if the C code has access to the type declaration of `f`, then it might perform some implicit casting first; for example, if we declared

```
long f(double a, const char *b, double c);

long a = f(23, "yes", 34uL);
```

then the call would be interpreted by C as having implicit casts in it:

```
long a = f((double)23, "yes", (double)34uL);
```

Calling Functions

and the arguments would be passed in floating-point registers, like so:

```
movl $23, %eax
cvtsi2sd %eax, %xmm0           # first floating-point argument

leaq label_of_yes_string, %rdi # first integer/pointer argument

movl $34, %eax
cvtsi2sd %eax, %xmm1           # second floating-point argument

callq f
# %rax is "long a" here
```

Functions Declaration

```
int f(int x);
```

- Declaration of the function
- Function header
- Function signature
- Function prototype

We want this in every file that invokes `f()`

Functions Definition

```
int f(int x) {
    return 2130 * x;
}
```

- Definition of the function

We only want this in **one** `.c` file

- Do not want 2 definitions
- Which one should the linker choose?

Header Files

C header files: `.h` files

- Written in C, so look like C
- Only put header information in them
  - Function headers
  - Macros
  - `typedef`s
  - `struct` definitions
- Essentially: information for the **type checker** that does not produce any actual binary
- `#include` the header files in our `.c` files

Big Picture

Header files

- Things that tell the type checker how to work
- Do not generate any actual binary

C files

- Function definitions and implementation
- Include the header files

Including Headers

```
#include "myfile.h"
```

- Quotes: look for a file where I'm writing code
- Our header files

```
#include <string.h>
```

- Angle brackets: look in the standard place for includes
- Code that came with the compiler
- Likely in **/usr/include**

ls /usr/include you'll see a lot of header files

vim /usr/include/string.h

Macros

```
#define NAME something else
```

- Object-like macro
- Replaces `NAME` in source with `something else`

It does this at the text level.

```
#define NAME(a,b) something b and a
```

whatever is after the comma until the second parentheses is the second thing.

Whatever is the first thing after the opening parentheses until the comma.

- Function-like macro
- Replaces `NAME(X,Y)` with `something Y and X`

text level

Lexical replacement, *not* semantic

Check the example on portal: include-me.h and use-include.c

run cpp use-include.c | less

Interesting Example

```
#define TIMES2(x)   x * 2          /* bad practice */
#define TIMES2b(x) ((x) * 2)       /* good practice */

int x = ! TIMES2(2 + 3);
```

x = !2+3*2; ⇒ x = 0+6 ⇒ x = 6.

(0 and 6)

```
int y = ! TIMES2b(2 + 3);
```

x = !((2+3)*2);

Be very explicit when you're using the macro define. Wrap them in parentheses and enforce an order of operations.

Examples and More

- header example
- `string.h`
- variadic functions

#ifndef } If ... is already defined, it won't do
#endif } anything between the ifndef and endif

Why? We want to include a header file in all of our C files. If I include the same file multiple times, it's correct but unred.