CSO-Recitation 08

CSCI-UA 0201-007

R08: Assessment 06 & Assembly

Today’s Topics

- Assessment 06
- Assembly
- Some exercises
  - give some senses about lab3

Assessment 06

Q1 Static and extern

Below are 4 C source files and their contents.

Q1.1 foo1.c

Which of the following statements are true?

A. The command gcc foo1.c creates a binary executable file called a.out

B. The command gcc -c foo1.c creates a non-executable object file called foo1.o

C. The command gcc foo1.c results in an error.

D. After executing line 3, variable g has value 1.

E. After executing line 3, variable g could have any value.

foo1.c

```
1: int g = 0;
2: int main() {
3:   g++;
4: }
```

bar1.c

```
int g = 1;
```

foo2.c

```
1: extern int g;
2: int main() {
3:   g++;
4:}
```

bar2.c

```
static int g = 1;
```

Q1 Static and extern

Q1.2 foo2.c

Which of the following statements are true?

A. The command gcc foo2.c creates a binary executable file called a.out.

B. The command gcc -c foo2.c creates a non-executable object file called foo2.o.

C. The command gcc foo2.c results in an error.

D. The command gcc foo2.c bar1.c creates a binary executable file called a.out.

E. The command gcc foo2.c bar1.c results in an error.

F. The command gcc foo2.c bar2.c creates a binary executable file called a.out.

G. The command gcc foo2.c bar2.c results in an error.

foo1.c

```
1: int g = 0;
2: int main() {
3:   g++;
4: }
```

bar1.c

```
int g = 1;
```

foo2.c

```
1: extern int g;
2: int main() {
3:   g++;
4:}
```

bar2.c

```
static int g = 1;
```

Q1 Static and extern

Q1.3 Value of g

If I want to have variable g to have value 2 right before returning from main function, what command should be used to generate the executable file?

A. gcc foo1.c 1

B. gcc foo2.c error

C. gcc foo1.c bar1.c error

D. gcc foo1.c bar2.c 1

E. gcc foo2.c bar1.c 2

F. gcc foo2.c bar2.c error

G. gcc foo1.c foo2.c error

H. gcc foo1.c bar1.c bar2.c error

I. gcc foo2.c bar1.c bar2.c 2

J. gcc foo1.c foo2.c bar1.c bar2.c error

foo1.c

```
1: int g = 0;
2: int main() {
3:   g++;
4: }
```

bar1.c

```
int g = 1;
```

foo2.c

```
1: extern int g;
2: int main() {
3:   g++;
4:}
```

bar2.c

```
static int g = 1;
```

Q2 Static for local variable

```
void my_func(int v)
{
    static int c1 = 0;
    int c2 = 0;
    c1 += v;
    c2 += v;
}
```

The following shows the code for function my_func

Q2.1 basic

A. Local variable c1 is allocated upon each invocation of my_func and de-allocated upon its return.

B. Local variable c2 is allocated upon each invocation of my_func and de-allocated upon its return.

C. Local variable c1 and c2 always have the same value right before the return of my_func.

D. Local variable c1 has scope within function my_func and cannot be referred to from outside of this function.

E. Local variable c2 has scope within function my_func and cannot be referred to from outside of this function.

When "static" prefix local variables:

- Initialized once, never deallocated
- like a global variable, except with local scope

Q2 Static for local variable

```
void my_func(int v)
{
    static int c1 = 0;
    int c2 = 0;
    c1 += v;
    c2 += v;
}
```

Suppose one executes the following code snippet:

```
my_func(10);
my_func(20);
```

Which of the following statements are true?

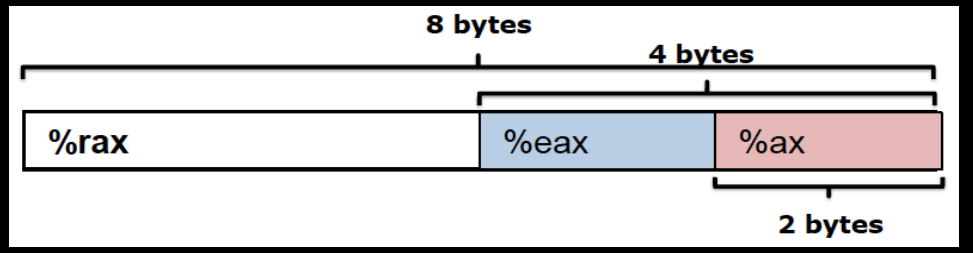
A. Right before returning from my_func(20), variable c1 has value 20.

B. Right before returning from my_func(20), variable c1 has value 30.

C. Right before returning from my_func(20), variable c2 has value 20.

D. Right before returning from my_func(20), variable c2 has value 30.

Q3 register



After x86 CPU executes instruction movq $0x12345678, %rax, which of the following is true?

A. The higher order 4-byte of register %rax are all zeros.
B. The higher order 4-bytes of register %rax remain the same as before the movq instruction is executed.
C. Register %eax has value 0x00000000
D. Register %eax has value 0x12345678
E. Register %eax is not changed by the movq instruction.
F. Register %ax has value 0x1234
G. Register %ax has value 0x5678

Q4 illegal instructions

Which of the following instructions are **not** legitimate x86-64 instructions?

A. movq %rax, %rbx

B. movq (%rax), %rbx

C. movq (%rax), (%rbx)

D. movq %rax, %rip

E. movq $1, %rax

- Moving data
  - movq src dst
- Operand types
  - Immediate: constant integer data
    - immediate can only be *source*
  - Register: one of the general purpose register
  - Memory
    - no memory-memory mov

Q5 mov

Suppose register %rax stores C variable long *x. Which of the following instruction corresponds to the C statement *x = 10;

A. movq $10, %rax
B. movq $10, (%rax)
C. movq (%rax), $10
D. movq %rax, $10

- long *x
  - x is a pointer to long
- *x=10;
  - de-referencing x, assign the value 10
- x is an address stored in %rax, use (%rax) to deference it.

Q6 GDB -- Segmentation fault

Inside GDB, what operations can help us debug a segmentation fault?

A. run code and hit control-c
B. use *backtrace* to see the call stack trace
C. use *up* or *frame* to go to where your code was last running
D. use *p* to print some value or *info locals* to check if local variables are bad

Q7 String manipulation

Q7.1 What does this function return for the following arguments?

- ignore_case = true
- buf1 = "Develop and use skills and knowledge."
- buf2 = "Develop and use Skills and Knowledge."

A. It returns 0

B. It returns a non-zero integer

```
int str_cmp(char *buf1, char *buf2, bool ignore_case) {
    int diff = 0;
    if(ignore_case)
        diff = strcasecmp(buf1,buf2);
    else
        diff = strcmp(buf1,buf2);
    return diff;
}
```

strcasecmp is a case *insensitive* comparison of 2 char arrays
strcmp is a case *sensitive* comparison of 2 char arrays

Q7 String manipulation

Q7.2 What does this function return for the following arguments?

- ignore_case = false
- buf1 = "Develop and use skills and knowledge."
- buf2 = "Develop and use Skills and Knowledge."

A. It returns 0

B. It returns a non-zero integer

diff = ‘s’-’S’;

```
int str_cmp(char *buf1, char *buf2, bool ignore_case) {
    int diff = 0;
    if(ignore_case)
        diff = strcasecmp(buf1,buf2);
    else
        diff = strcmp(buf1,buf2);
    return diff;
}
```

strcasecmp is a case *insensitive* comparison of 2 char arrays
strcmp is a case *sensitive* comparison of 2 char arrays

Q7 String manipulation

Q7.3 What does this function return for the following arguments?

- ignore_case = false
- buf1 = "Develop and use skills and knowledge."
- buf2 = "Develop and use skills and knowledge."

A. It returns 0

B. It returns a non-zero integer

```
int str_cmp(char *buf1, char *buf2, bool ignore_case) {
    int diff = 0;
    if(ignore_case)
        diff = strcasecmp(buf1,buf2);
    else
        diff = strcmp(buf1,buf2);
    return diff;
}
```

strcasecmp is a case *insensitive* comparison of 2 char arrays
strcmp is a case *sensitive* comparison of 2 char arrays

Assembly

Assembly programming

Important Instructions

Instruction	What it does
mov src, dest	dest = src
add src, dest	dest = dest + src
sub src, dest	dest = dest - src
imul src, dest	dest = dest * src
inc dest	dest = dest + 1

Moving data - Instruction operands

- src and dest can be one of three things
  1. An **immediate**
     - A constant value, prefaced with $
     - Eg. $0, or $0xabcdabcd
     - dest cannot be an immediate
  2. A **register**
     - One of the 16 general purpose registers
     - Eg. %eax, %rax, %rsi..
  3. A location in **memory**
     - (Register): Consider registers as pointers, and get the value at an address in memory with various “addressing modes”
     - Eg. (%rax), 10(%rax), 10(%rax, rbx, 4)
     - You cannot perform a mov from memory into memory
     - How big is what you are getting from memory, in bytes?

Instruction Suffixes

Suffix	Name	Size (Bytes)
b	byte	1
w	word	2
l	long	4
q	quadword	8

Memory Addressing Modes

- Direct addressing
  - Given a register, use the value located at the memory address contained in the register
  - Register name in parens
  - Eg. mov (%rax), %rbx
- With displacement
  - Use the value in memory located at the register value plus a constant displacement
  - Have the constant appear before the parens
  - Eg. mov 10(%rax), %rbx

Memory Addressing Modes

- Complete
  - We have a constant displacement, a starting point, an offset, and constant to scale the offset by…
  - **D(Rb, Ri, S)**
    - The address at Rb + Ri*S + D, where S and D are constant and Rb and Ri are registers
  - Eg. movq 10(%rax, %rbx, 4), %rcx
  - If the displacement is 0 or the scale is 1, you may leave them out

Lea src, dest

- Lea: Load Effective Address
- Take the address expression from src, and save it to dest
- Do not access memory, just compute the address from the offsets, index, base, and scale, and then save the computed address in dest
- Can also be used to quickly add registers and store the result in a third register

RFLAGS

- A special purpose register that stores some status about the executed instructions
- Different bits tell us different things
- Instructions may set those bits depending on what has happened
  - These include arithmetic instructions like add or sub, as well as instructions like cmp

RFLAGS

Flag	Meaning
ZF	Result was 0
SF	The most significant bit of the result
CF	Set if the result borrowed from or carried out of the most significant bit
OF	Overflow for signed arithmetic

- The CPU doesn’t know if operands are signed or unsigned
- So, it calculates both the signed overflow (OF) and the unsigned overflow (CF) for each instruction
  - That is, OF is set assuming both are signed
  - CF is set assuming both are unsigned

How to decide whether there is overflow?

- Unsigned int:
  - you can look at whether there is a carry/borrow of the MSB
- Signed int:
  - for machine:
    - if there is carry-in but no carry-out of MSB
    - or, there is no carry-in but there's carry out of MSB
  - for human:
    - if two positive numbers add to a negative number
    - or, two negative numbers add to a positive number

Instructions set/read RFLAGS

- Instructions that set RFLAGS
  - Regular arithmetic instructions
    - add, sub, imul, inc
  - Special flag-setting instructions
    - cmp, test
- Instructions that read RFLAGS
  - Instructions that read RFLAGS to set register values
    - Set
  - Instructions that read RFLAGs to set %rip
    - jmp

cmp

- Same as sub (*dest-src*), except it doesn't store the result in dest
- It does, however, still change the RFLAGS I just mentioned
- This makes it useful for comparisons and conditions

jmp

- jmp label
  - Continues executing from the label, unconditionally
  - label is where to jump to
  - It acts like goto in C

Conditional Jumps

- je label
  - Jump if ZF is set
- jne label
  - Jump if ZF is not set
- jg label
  - Jump if ZF is not set and SF and OF are the same
- jl label
  - Jump if SF and OF are not the same
- ja label
  - Jump if CF and ZF are both not set

Exercises

Lab3 -- Uncover the mystery

- Very much like a puzzle game
- In this lab, we give you 5 object files, ex1_sol.o, ex2_sol.o, ..., ex5-sol.o, and withhold their corresponding C sources
- Each object file implements a particular mystery function (e.g. ex1_sol.o implements the function ex1)
- We ask you to deduce what these mystery functions do based on their x86-64 assembly code
- Write the corresponding C function that accomplishes the same thing

Exercise

- Try to figure out what the assembly code does, and write C code that does the same thing in `main.c`.

```
mystery:
    movl    $0, %edx
    movl    $0, %eax
    movl    $1, %ecx
    jmp .L2
.L3:
    leal    (%rcx,%rax), %esi
    addl    $1, %edx
    movl    %ecx, %eax
    movl    %esi, %ecx
.L2:
    cmpl    %edi, %edx
    jl  .L3
    ret
```