CSO-Recitation 09

CSCI-UA 0201-007

R09: Assessment 07 & More Assembly

Today's Topics

- Assessment 07
- More Assembly
  - Procedure calls & data segment
- Breakout exercises

Assessment 07

Q1 %eax

Suppose register %eax corresponds to the C variable x of some integer type. If the value of %eax is 0xffffffff, what potentially could be the type and value of x?

[✓ A. type: int, value: -1]

B. type: int, value: -2^{31}

C. type: long, value: -1

D. type: long, value: -2^{63}

[✓ E. type: unsigned int, value: 2^{32}-1]

F. type: unsigned int, value 2^{32}

G. type: unsigned long, value 2^{32}-1

H. type: unsigned long, value 2^{32}

Q2 movq

Suppose register %rdi and %rsi corresponds to C variable x and y, respectively. Given machine instruction movq (%rdi, %rsi, 8), %rax, what can you infer to be the most likely type of x and y, respectively?

A. long and long

[✓ B. long * and long]

C. long * and long *

D. int * and long   movl (%rdi, %rsi, 4), %rax

E. int and int

F. int * and int

G. int * and int *

- movq (%rdi, %rsi, 8), %rax
- (%rdi, %rsi, 8)
- *(x+8y)
- x+8y is a pointer
- y is an integer type, since %rsi, y should be long type
- x is also 8 bytes, here, more likely to be long * type

Q3 Deference pointers

Suppose %rsi corresponds to C variable y of some pointer type. Which of the following instructions dereference the pointer y?

A. leaq (%rsi), %rax

[✓ B. movq (%rsi), %rax]

C. movq %rsi, %rax

[✓ D. subq %rax, (%rsi)]

E. subq %rax, %rsi

F. none of the above

- derefence the pointer y stored in register %rsi:
- (%rsi)
- lea: no memory access!

Q4 Basic machine execution

Which of the following statements are true?

A. Accessing data stored in memory is as fast as accessing data stored in CPU registers.

[✓ B. Accessing data stored in memory is much slower than accessing data in CPU registers.]

[✓ C. A C program is compiled into x86 instructions which are directly executed by the CPU.]

D. A Java program is compiled into x86 instructions which are directly executed by the CPU.

E. One can use %rip as an operand for the mov instruction

only 16 general purpose registers

Q5 mov vs. lea

Let a be an array of int elements. Suppose %rdi stores the address of a[0], and %rsi stores index i of type long. Which of the following instruction or sequence of instructions result in %eax storing a[i]?

A. leal (%rdi, %rsi, 4), %eax

[✓ B. movl (%rdi, %rsi, 4), %eax]

C. movl (%rsi, %rdi, 4), %eax

D. leal (%rdi, %rsi, 8), %eax

E. movl (%rdi, %rsi, 8), %eax

F. movl (%rsi, %rdi, 8), %eax

[✓ G. salq $2, %rsi]
addq %rdi, %rsi
movl (%rsi), %eax

[✓ H. salq $2, %rsi]
movl (%rsi, %rdi), %eax

- a is an array of int
- a[i] == *(a+i)
- (%rdi, %rsi, 4)

- salq src, dest => dest=dest << src
  - arithmetic left shift
- salq $2, %rsi
  - == 4 * %rsi
  - now, %rsi -> 4i
  - then, %rsi=%rsi+%rdi=4i+&a[0]
  - then, derefence it to get the value of a[i]

Q6 Lab3 with gdb

For the next series of questions, you need to use gdb to run Lab3's tester_sol which is the executable tester linked with ex_sol{1-5}.o.

Q6.1 ex1

Stop execution in the **first** invocation of function ex1 (use breakpoints).

- Examine ex1's machine instructions. What is the value of register %rsi prior to executing the first instruction of ex1? (%rsi contains the second function argument).
- (Please write the value as a decimal number)
- 100

Q6 Lab3 with gdb

Q6.2 ex1

- During tester_sol's **first** invocation of function ex1, what is the value of register %eax prior to the function's return? (Write the value as a decimal number)
- 1

Q6.3 ex2

- During tester_sol's **first** invocation of function ex2, what is the value of register %rsi prior to executing the first instruction of ex2? (%rsi contains the second function argument).
- (Please write the value as a decimal number)
- 4

Q6 Lab3 with gdb

Q6.4 ex2 (%rdi)

- During tester_sol's first invocation of function ex2, what is the value of register %rdi prior to executing the first instruction of ex2? (%rdi contains the first function argument).
- Please write %rdi's value as a decimal number.
  - 140737488347056
  - 140737488347024

Q6.5 ex2 (%rdi)

- This question is the same as Q6.4, except that please write %rdi's value as a hex number (your answer should include the prefix 0x)

Q6 Lab3 with gdb

Q6.6 ex2 (%rdi)

By looking at your answers for Q6.4 and Q6.5, guess the most likely data type for the variable stored in %rdi (which is the first argument of function ex2)?

A. unsigned long

B. long

C. int

D. unsigned int

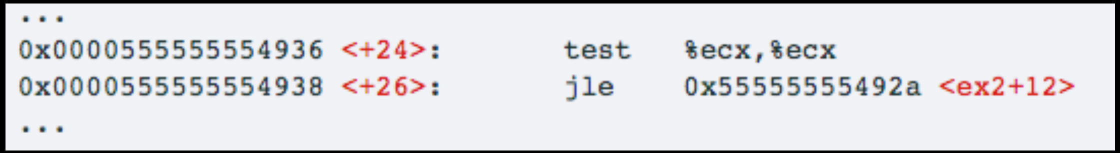
[✓ E. some pointer type]

F. none of the above

Q6 Lab3 with gdb

Q6.7 ex2

The machine instructions for ex2 contain the following instruction



```
...
0x0000555555554936 <+24>:       test   %ecx,%ecx
0x0000555555554938 <+26>:       jle    0x55555555492a <ex2+12>
...
```

For which values of %ecx would the jump to instruction at address 0x55555555492a occur?

[✓ A. zero]

B. any positive value

[✓ C. any negative value]

D. 1

E. None of the above

- testq src dst: like andq src, dst except dst is unchanged
  - set ZF, SF appropriately
- jle label: less or euqal (signed)
  - (SF^OF) | ZF
- when ZF?
  - val(%ecx)=0
- when SF^OF ?
  - OF -> 0
  - so SF should be set (SF-> 1). When?
    - val(%ecx) is negative

Procedure calls

Calling functions

How do you call functions?

- How do you actually start executing the code of a function?
  - Well, we know about jmp, does that help us? Why not?
- Do you need to do something before calling a function?
  - What?

How do you call functions?

```
mystrlen:
movl $0, %eax
jmp .condition
.loop:
addl $1, %eax
.condition:
movb (%rdi,%rax), %bl
cmp $0, %bl
jne .loop
```

```
main:
jmp mystrlen
```

How do you call functions?

```
mystrlen:                          main:
movl $0                                   mystrlen
jmp .
.loo
ad          eax
.co
mov          %rax), %bl
cmp $
jne .loop
```

How do you call functions?

```
mystrlen:
movl $0, %eax
jmp .condition
.loop:
addl $1, %eax
.condition:
movb (%rdi,%rax), %bl
cmp $0, %bl
jne .loop
// How do we get back?
```

```
main:
//Where are the arguments?
jmp mystrlen
```

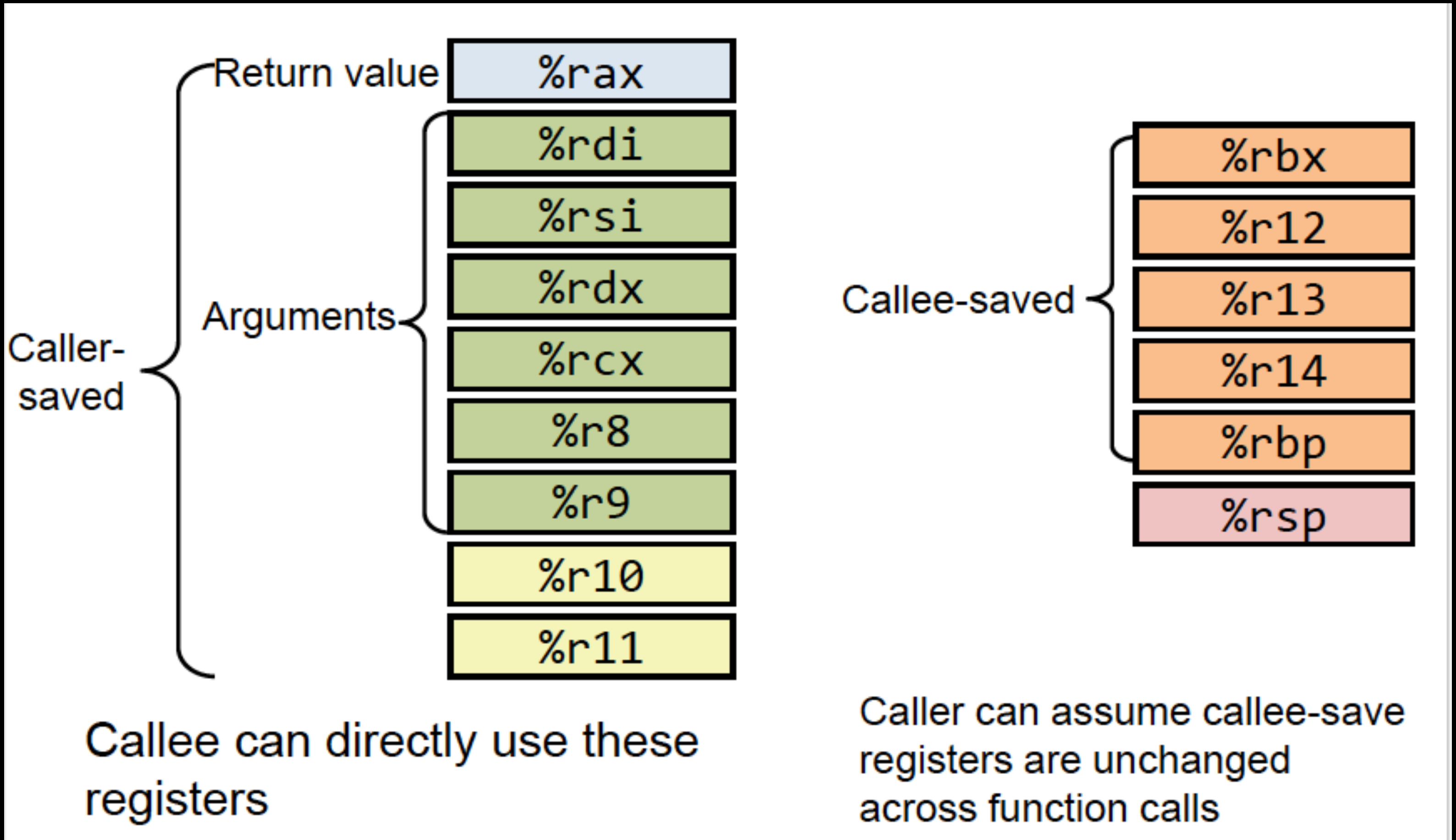
Remember where we came from

- A function that calls another (a caller) knows what it is calling
- A function that is called (a callee) does not know who its caller is
    - But it needs to know where to restore execution when it is done
    - It is the responsibility of the caller to tell the callee where to restore execution
    - We want to restore execution on the instruction after we called the function
    - We store this return address on the stack
        - callq handles this for us

Set up registers

- The first six arguments are stored in this order:
  - %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - So when calling a function, you must set those registers to the correct value for that argument
- The return value is stored in %rax
- Functions may feel free to use the argument registers and the return value register, as well as %r10, and %r11
- If the caller was using the argument registers for something, it must save them first, as the callee may use those registers for any purpose
  - It can save them to the stack
  - This is also true of the registers %r10, %r11, and %rax
- The callee must save certain registers if it plans on using them
  - They are %rbx, %r12, %13, %r14, %rbp, and %rsp

Set up registers



Callee can directly use these registers

Caller can assume callee-save registers are unchanged across function calls

The stack

- The register %rsp points to the top of the stack
- The stack grows downwards
- We use it to store return addresses as well as registers whose values we don’t want to lose
- We use it to store the 7th, 8th, 9th etc. function arguments
- We also use it to store local variables
- You can use pushq and popq to add and remove things from the stack

The Stack

- pushq src
- Takes one operand
- DECREASES %rsp by 8
- THEN stores the operand at the memory location given by the new %rsp

The Stack

- popq dst
- Takes one operand
- Takes the value in memory located at %rsp and stores it in the operand
- THEN INCREASES %rsp by 8

The Stack

- callq label
- Takes one operand
- DECREASES %rsp by 8
- THEN stores the return address at the memory location given by the new %rsp
- THEN jumps to the operand

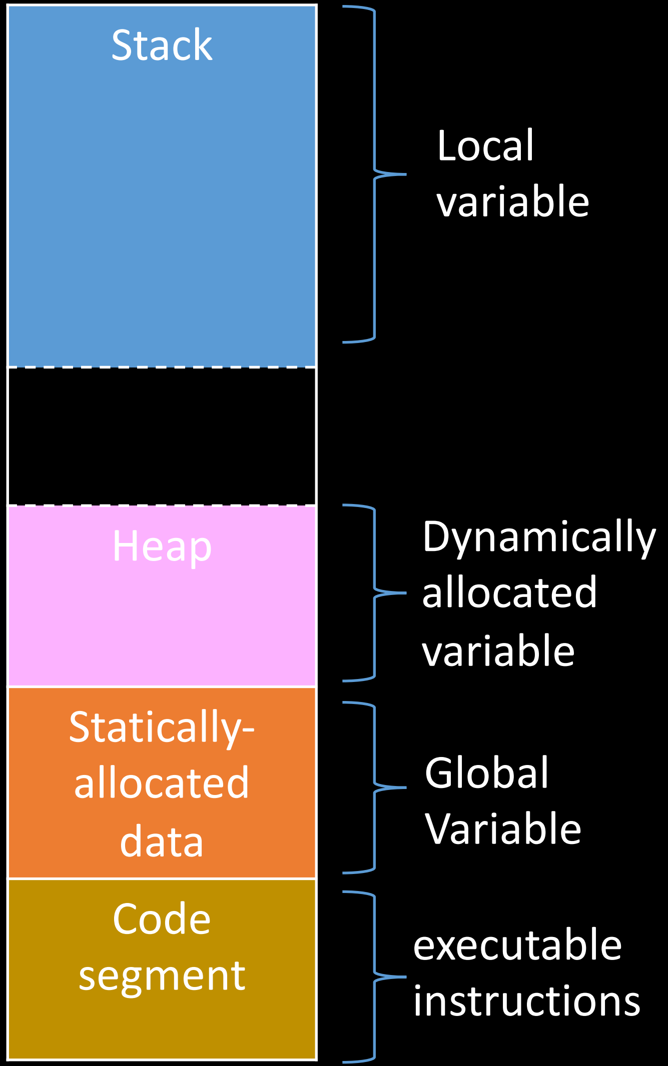
Push return address on stack

The Stack

- retq
- Takes no operands
- Jumps to the location given by the value in memory located at %rsp
- THEN INCREASES %rsp by 8

Data segment

- Local variables
  - Stack
    - C’s primitive data type and pointer – registers whenever possible
    - Array, struct
- Global variables
  - global variable / static global variable
- Dynamic allocated variables
  - e.g. malloc
  - Heap



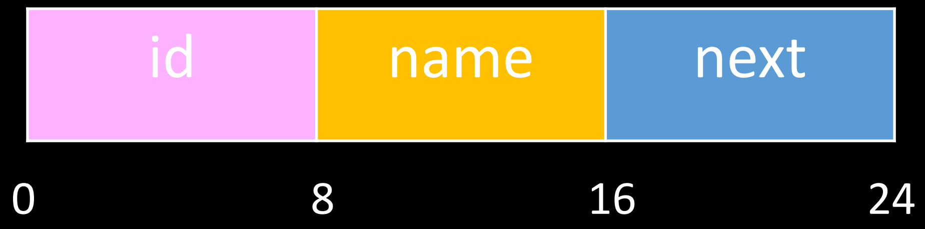
Example of Array/Struct accessing

- Array Accessing Example
  - int getnum(int *arr, long i) { return arr[i];}
  - Suppose %rdi contains arr; %rsi contains i; %eax is to contain arr[i]
  - movl (%rdi, %rsi, 4), %eax

  - char* getpointer(char **arr, long i) { return arr[i];}
  - Suppose %rdi contains arr; %rsi contains I; %rax is to contain arr[i]
  - movq (%rdi, %rsi, 8), %rax

Example of Array/Struct accessing

```
typedef struct node {
        long id;
        char *name;
        struct node *next;
}node;
```



```
void init_node(node*n, long id, char *name){
        n->id=id;
        n->name=name;
        n->next=NULL;
}
```

```
movq %rsi, (%rdi)
movq %rdx, 8(%rdi)
movq $0, 16(%rdi)
```

Exercise