

CSO-Recitation 07

CSCI-UA 0201-007

R07: Assessment 05 & Assembly & lab2

Today's Topics

- Assessment 06
- Assembly
- Using GDB to debug assembly

Assessment 06

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 *

- `(%rdi, %rsi, 8) => val(%rdi)+val(%rsi)*8` is a pointer
- Pointer arithmetic
 - `=> %rdi (x)` is a pointer
 - `=> %rsi (y)` is used for offset



- Typically this is indexing an array. Since `+8y` here, more likely the element is 8-byte.

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 *

- Typically this is indexing an array. Since “+8y” here, more likely the element is 8-byte.

```
long x[10] = {0};
```

```
long y = 1;
```

```
x[y] <= the address is val(x)+8*y
```

```
Q: x[y] == *(x+8*y)?
```

```
No! Pointer arithmetic:
```

```
The code x+8*y evaluates to val(x)+8*y*sizeof(long) =  
val(x)+64*y
```

```
x[y] == *(x+y)
```

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

- dereference the pointer y stored in register %rsi:
- `(%rsi)`
- `leaq`: 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

`%rip`: store the address of the instructions
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+the address of a[0]=address of a[i]`
 - then, dereference it to get the value of `a[i]`

Debugging with GDB without C
source files

Recap: Some common gdb commands

- help
 - Gdb provides online documentation. Just typing *help* will give you a list of topics. Or just type *help command* and get information about any other command.

Short Name	Long Name	What do it do?
r	run	Begins executing the program – you can specify arguments after the word run
s	step	Execute the current source line and stop before the next source line, going inside functions and running their code too
n	next	
p	print	Prints the value of an expression or variable
l	list	Prints out source code
q	quit	Exit gdb

step through the program one line at a time

Recap: Some more advanced gdb commands

Set the breakpoint at the beginning of the function

Short Name	Long Name	What do it do?
b	break	Sets a breakpoint at a specified location (either a <i>function</i> name or <i>line number</i>)
c	continue	Continues executing after being stopped by a breakpoint
bt	backtrace	Prints out information on the call stack, i.e. where in the program's execution it is being stopped at
f	frame	Prints information on the current frame / allows you to change frames
i	info	Prints out helpful information (e.g. info args and info locals)

Segmentation fault
(core dumped)

Debugging without C source code

- ~~Use *Next/step* to run one line of code~~
 - No C source code.
 - Instead, use *ni/si* to run one line of assembly
- ~~Use *list* to see the source code that ran~~
 - Instead: use *disas* to see the assembly that ran
- ~~Use *print var* to examine the variable named var~~
- ~~Check the locals (*info locals*) and args (*info args*) to see if they are bad~~
 - Variable names generally disappear; Instead, examine registers/memory contents that stores the variables
 - Use *print \$reg* to examine the register name reg
 - E.g. *print \$rsi*
 - Alternatively, use *info registers* to see all the registers
 - Use *x address* to examine the content stored in 'address'

Other useful command and options

- Delete specified breakpoint id.

- *info breakpoints*
- *d <id>*

d	decimal
x	hex
t	binary
f	floating point
i	instruction
c	character

- *print* accepts *format*

- *p[rint][/*format*] *expr**, where *format* can be
- E.g. *p/x \$rip* prints the register `%rip` in hex format

- To examine the assembly code of a function

- Use *disas [func_name]*
- E.g., *disas ex1* prints for function `ex1`
- Alternatively, use *x/[count][format] [func_name]*

- A more descriptive source: <https://web.cecs.pdx.edu/~apt/cs322/gdb.pdf>

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.

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)
- Seem problematic
 - The result is a bit random. One possible result for Q6.4 is 140737488347056
 - Let's skip for now

Q6 Lab3 with gdb

Q6.6 ex2 (%rdi)

By looking at your answers for Q6.4 and Q6.5, guess for the variable stored in %rdi (which is the first argument)

A. unsigned long

B. long

C. int

D. unsigned int

E. some pointer type

F. none of the above

A few clues:

- The value starts with 0x7fff..xxx
- More precisely, this is likely a pointer to stack memory

Linux Memory Layout

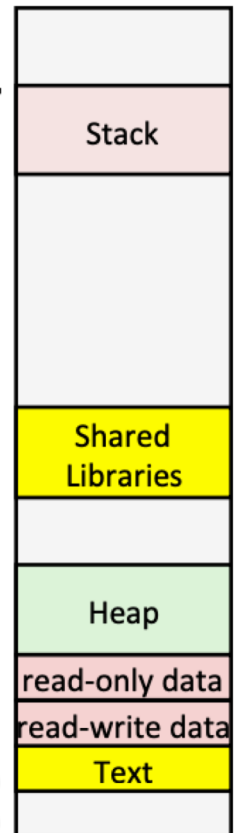
memory regions

program:

Shared Libraries

Instructions

0000000000400000
0000000000000000



8MB
default
limit

Assembly

C is for people

Why Assembly

- In the real world, computers don't "understand" code
- They only "understand" a set of instructions
- To run code
 - 1. The CPU fetches an instruction from the memory at the PC(program counter)
 - 2. The CPU decodes that instruction
 - 3. If needed, the CPU fetches data from memory
 - 4. The CPU performs computations
 - 5. If needed, the CPU writes data to memory
 - 6. The CPU increments the PC to the next instruction

Why Assembly

- Computers don't "understand" assembly either, but assembly maps much more closely to machine instructions than C code
- Assembly code involves instruction "mnemonics"
 - For x86_64, These are things like `addq`, `movq`, `imul`

X86 general purpose registers

- Accessing memory is very, very slow compared to the rest of what a CPU can do
- Registers are fast temporary storage
- X86-64 ISA: 16 8-byte general purpose registers
- 8 of them were evolved from 16-bit ISA, `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%rbp`, `%rsp`
 - Lower 32-bit – replace `r` with `e`, eg `%eax`, `%esp`
 - Lower 16-bit – remove `r`, eg `%ax`, `%sp`
- With 64 bits came 8 more registers, `%r8` to `%r15`
 - Lower 32-bit - add a `d`, eg `%r8d`
 - Lower 16-bit – add a `w`, eg `%r8w`
 - Lower 8-bit – add a `b`, eg `%r8b`
- `%ax`, `%bx`, `%cx`, and `%dx`, allow you to access their upper 8 bits (replace “`x`” with “`h`”)

As a table:

64-bit	32-bit	16-bit	8-bit (low)
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

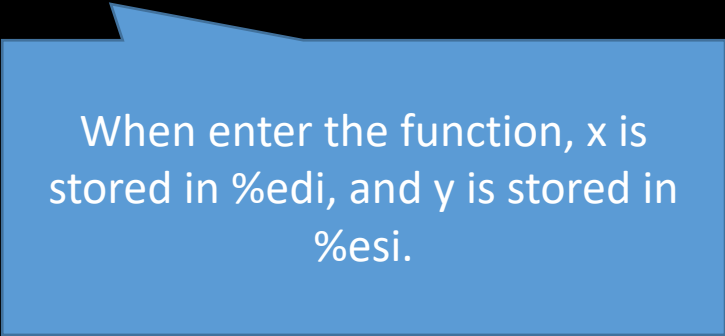
Some of the 16-bit subregisters are also special: the original 8086 allowed the **high** byte of AX, BX, CX, and DX to be accessed independently, so x86-64 preserves this for some encodings:

16-bit	8-bit (high)
AX	AH
BX	BH
CX	CH
DX	DH

Usage of registers

- rdi, rsi, rdx, rcx, r8, r9: used for pass the parameters (follow the sequence)

```
void function(int x, int y)
{
    ....
}
```



When enter the function, x is stored in %edi, and y is stored in %esi.