Assembly: ASCII Strings, Heterogeneous Data Structure Example, and Writing to Standard Output

In this lab, we will:

1. Learn how to represent and work with strings in assembly code
2. Have our first exposure to working with more complex “data objects”
3. Learn how to send data to standard output

Our focus will be on understanding and working with the provided example code.

Setup

Follow the post on piazza to create your github classroom lab repository.

Log into your UNIX environment and clone the repository and change directories into the repository working copy.

All example code provided in this writeup are also available in this Disucssion’s repository.

Reference Sheet

It’s a good idea to have a copy of the assembly and gdb reference sheet with out when working on this lab.

The pdf is included in the text book and can be found here: https://cs-210-fall-2023.github.io/UndertheCovers/textbook/images/INTELAssemblyAndGDBReferenceSheet.pdf

A Heterogeneous Data Structure Example: `flowers.s`

Writing assembly code is a little like "painting" our ideas into memory. We can create data structures/objects as chunks of memory, then write code that interprets certain pieces of that memory however we want. To get an idea, let's look at an example.

Everything is from scratch

In the end there really is not that much to know with respect to how a computers works – fetch, decode, execute ;-) The trick is to creatively organize memory and write code that correctly works with that memory. The CPU and Memory have no notion of things like structures and objects. Rather, it is up to us to write code that creates these ideas. So while we use terms like object and fields/members, remember these are just ideas that we will then need to write code to implement.

In this example, an assembly programmer has written some code to work with a little data base of "flowers". The data base will be an array of 13 "flower objects". The idea is that each flower has three "fields/members": 1) A name 2) count 3) number of petals. In our case, the programmer is going to map this idea as follows:

1. Each flower will be represented as a 24 byte chunk of memory, broken down as follows
   - The first 8 bytes will be used to store an address that "points" to an ASCII 0-terminating string which holds the name of the flower
   - The second 8 bytes will be treated as an integer that will store the count/quantity of that type of flower
   - The third 8 bytes will be treated as an integer that will store the number of petals that type of flower has
2. The array will be an area of memory whose location in the code will be at the label: "flowers"
   - Given that each element in the array is a flower and each one is 24 bytes in size. With 13 flowers, the array will consume 24×13 bytes
   - Each flower object will be initialized appropriately
3. All string names of flowers will be defined separately

How might you draw a generic box diagram for a flower?

- three 8 bytes boxes
  - the first is a pointer to the name of the flower : we can think of this as the "name" field of the object
  - the second is the count field
  - the third is the number of petals field

Here is the assembly code our programmer has written to encode our data base:

```
       .section .data
flowers:
        .quad lilystr
        .quad 5
        .quad 3
        .quad irisstr
        .quad 7
        .quad 3
        .quad buttercupstr
        .quad 4
        .quad 5
        .quad wildrosestr
        .quad 2
        .quad 5
        .quad larkspurstr
        .quad 9
        .quad 5
        .quad columbinestr
        .quad 2
        .quad 5
        .quad delphiniumsstr
        .quad 3
        .quad 8
        .quad ragwortstr
        .quad 4
        .quad 13
        .quad cinerariastr
        .quad 1
        .quad 13
        .quad asterstr
        .quad 2
        .quad 21
        .quad chicorystr
        .quad 1
        .quad 21
        .quad plantainstr
        .quad 8
        .quad 34
        .quad pytethrumstr
        .quad 10
        .quad 34
```

And the following is how the programmer coded the string names of the followers:

```
names:
lilystr:        .asciz "lily"
irisstr:        .asciz "iris"
buttercupstr:   .asciz "buttercup"
wildrosestr:    .asciz "wild rose"
larkspurstr:    .asciz "larkspur"
columbinestr:   .asciz "columbine"
delphiniumsstr: .asciz "delphiniums"
ragwortstr:     .asciz "ragwort"
cinerariastr:   .asciz "cineraria"
asterstr:       .asciz "aster"
chicorystr:     .asciz "chicory"
plantainstr:    .asciz "plantain"
pytethrumstr:   .asciz "pytethrum"
```

Can you visualize the memory layout this code creates? Can you see how the name field of each flower object “points” to its name string?

Now let’s look at a complete program that does some calculations on the data base:

1. Use gdb and explore this code
   - where is the flower array located : `p /x & flowers`
   - explore a single flower – display all three fields : `x /3xg <address of flower>`
   - “follow” the pointer in a flower to its name string : `x /1s <address of name string>`
2. Now single step this code and see if you can figure out what it is doing

```
        .intel_syntax noprefix

        .section .text
        .global _start
_start:
        mov rax, OFFSET flowers
        mov rbx, 13
        xor rdi, rdi
        xor rcx, rcx

loop_head:
        mov rdx, QWORD PTR [rax]
        cmp BYTE PTR [rdx], 'c
        jne loop_next
        mov rsi, QWORD PTR [rax + 8]
        imul rsi, QWORD PTR [rax + 16]
        add rcx, rsi

loop_next:
        add rax, 24
        inc rdi
        cmp rdi, rbx
        jl loop_head

        mov rax, 60
        mov rdi, rcx
        syscall

        .section .data
flowers:
        .quad lilystr
        .quad 5
        .quad 3
        .quad irisstr
        .quad 7
        .quad 3
        .quad buttercupstr
        .quad 4
        .quad 5
        .quad wildrosestr
        .quad 2
        .quad 5
        .quad larkspurstr
        .quad 9
        .quad 5
        .quad columbinestr
        .quad 2
        .quad 5
        .quad delphiniumsstr
        .quad 3
        .quad 8
        .quad ragwortstr
        .quad 4
        .quad 13
        .quad cinerariastr
        .quad 1
        .quad 13
        .quad asterstr
        .quad 2
        .quad 21
        .quad chicorystr
        .quad 1
        .quad 21
        .quad plantainstr
        .quad 8
        .quad 34
        .quad pytethrumstr
        .quad 10
        .quad 34

names:
lilystr:        .asciz "lily"
irisstr:        .asciz "iris"
buttercupstr:   .asciz "buttercup"
wildrosestr:    .asciz "wild rose"
larkspurstr:    .asciz "larkspur"
columbinestr:   .asciz "columbine"
delphiniumsstr: .asciz "delphiniums"
ragwortstr:     .asciz "ragwort"
cinerariastr:   .asciz "cineraria"
asterstr:       .asciz "aster"
chicorystr:     .asciz "chicory"
plantainstr:    .asciz "plantain"
pytethrumstr:   .asciz "pytethrum"
```

Writing to Standard Out

So far we have limited our use of the operating system to “calling” the “Exit” system call. Well it’s time we use the OS kernel for something a little more interesting. Remember that when we studied UNIX, we said that the OS provided a whole range of “system calls” that a process could invoke. Two of the most fundamental UNIX kernel system calls are “write” and “read” to and from a file descriptor. Recall that when we launch a new process, by default the new process inherits standard input, standard output and standard error streams from the parent. These streams are identified by file descriptor numbers 0, 1 and 2 respectively.

The following link will take you to an online table of system calls. Here you can learn what a specific system call’s number is and what registers to use to pass arguments to them.

https://filippo.io/linux-syscall-table/

Notice that the first two entries are for “read” and “write”. Each system call is identified by a unique “system call number”. When writing code for an INTEL computer running Linux, you place the number of the system call into the `rax` register prior to executing the `syscall` instruction.

The kernel will use the value of `rax` to locate the right code within itself, then execute it on behalf of the process. When done, the OS will resume execution of the calling process at the instruction right after the `syscall`, replacing the value in `rax` with a return code. The first column of the online table tells us what value we should load into `rax` for the particular system call. Double clicking on a row in the table will reveal a concise description of any additional arguments that you need to pass to the system call and what registers you should place them in.

Warning

Please be aware that the Intel `syscall` instruction will overwrite the `rcx` and `R11` registers. If your code is using either `rcx` or `R11`, then be sure to save their values prior to using the `syscall` instruction and restore/reload it after.

Note

The Linux documentation states that a system call may potentially return a second value in the `rdx` register. If a system call were to do this, it would have to be documented for that system call. To our knowledge, we have never seen a Linux system call that does this.

Write

Examining the entry for write (row two of the table), we see that:

1. It’s system call number is `1` and we should place this value in `rax`
2. The first argument should be placed in rdi and specifies the file descriptor number that we want to write to should be in `rdi` – thus to write to standard output we would put the value 1 into `rdi`
3. The second argument should be placed in `rsi` – at this stage without knowing ‘c’ syntax, it will be hard to tell what the argument description means – here is the explanation
    - the second argument to the write system call should be an address in our process that points to the bytes we want to send to the stream identified by the file descriptor we place in `rdi` – in our case `1` and thus standard output
4. The final argument to write is the number of bytes starting at the address specified in `rsi` that we want the kernel to copy from our process to the stream.

So to write bytes to standard output is pretty easy:

```
    mov rax, 1                          # rax = system call number of write
    mov rdi, 1                          # rdi = file descriptor number of standard
output
    mov rsi, address of data            # a memory address of where our data is (NOTE
IT MUST BE A MEMORY ADDRESS)
    mov rdx, length of data in bytes  # the number of bytes starting at the
specified address to be written
```

Some things to note:

1. We must use these registers to write. So if you currently have other values in them, you must either reorganize your code to free up these registers (aka not use them for other purposes or use the stack to save their value prior to making the call and restore them after)
2. The data must be at a memory location. So if you want to write a value that is currently in a register, you will first need to put the value somewhere in memory, and then pass that memory address to write

So let's put this knowledge to use with three examples.

Writing an arbitrary 8 byte value to standard out : `stdout1.s`

```
        .intel_syntax noprefix

        .section .data
x:      .quad 0xdeadbeeffeedface

        .section .text
        .global _start
_start:
        mov rax, 1
        mov rdi, 1
        mov rsi, OFFSET x
        mov rdx, 8

        # syscall instruction over writes, clobbers, rcx
        # (see intel manuals for details)
        syscall

        mov rax, 60
        mov rdi, 0
        syscall
```

Running a binary from this source will write the 8 byte value located at the label x to the standard output.

```
$ ./stdout1
����▯�$
```

Of course the value is just raw 8 byte binary number not ASCII data. So of course the terminal displays gibberish. Using our UNIX knowledge however we can do several things to make the output usable:

1. we could direct it to a file `./stdout1 > out.bin` or
2. we could pipe it to a program that translates binary data into other notations such as hex. As an example, let's use `od` (see `man od`)

```
$ ./stdout1 | od -t x1
0000000 ce fa ed fe ef be ad de
0000010
$ ./stdout1 | od -t x8
0000000 deadbeeffeedface
0000010
```

The first use of `od` translates the input into single bytes ascii hex notation of the underlying byte values and the second translates them into a single `8 byte` litte-endian value.

Writing an ASCII string to standard output : `stdout2.s`

Of course one of the useful things we can now do is send ASCII output directly from our program to standard output.

```
        .intel_syntax noprefix

        .section .data
mystr:  .asciz "Hello World\n"

        .section .text
        .global _start
_start:
        mov rax, 1
        mov rdi, 1
        mov rsi, OFFSET mystr
        # note length does not include the 0 to
        # ensure we only send valid ascii data
        mov rdx, 12
        # syscall instruction over writes, clobbers, rcx
        # (see intel manuals for details)
        syscall

        mov rax, 60
        mov rdi, 0
        syscall
```

Now running this and letting it’s output go directly to the terminal makes more sense:

```
$ ./stdout2
Hello World
$
```

Writing to stdout in the wild: `stdoutflowers.s`

Finally, let’s take a look at how we might integrate writing to standard output in code that is a little more complicated. Let’s update our “flowers” example so that it prints out the name of the flowers that it identifies.

Read this code carefully, comparing it to the original. You should strive to understand what each new line is doing and why.

If you are confused, draw a diagram of the registers used in this code along with what values get pushed and popped off of the stack. As you trace the code, update your diagram and think about what values are in the registers at any given time.

```
        .intel_syntax noprefix

        .section .text
        .global _start
_start:
        mov rax, OFFSET flowers
        mov rbx, 13
        xor rdi, rdi
        xor rcx, rcx

loop_head:
        mov rdx, QWORD PTR [rax]
        cmp BYTE PTR [rdx], 'c
        jne loop_next
        mov rsi, QWORD PTR [rax + 8]
        imul rsi, QWORD PTR [rax + 16]
        push rsi     # mywritestr expects pointer to string in rsi
        mov rsi, rdx
        call mywritestr
        pop rsi
        add rcx, rsi

loop_next:
        add rax, 24
        inc rdi
        cmp rdi, rbx
        jl loop_head

        mov rax, 60
        mov rdi, rcx
        syscall

        # To use this code correctly:
        #  call with pointer to zero temrinate string in rsi
        # inputs: rsi = & string
        # outputs: none
        # all registers used are restored to their orginal values
mywritestr:
        push rax
        push rdi
        push rdx
        push rcx     # clobbered by system call instruction
        push r11     # clobbered by system call instruction

        xor rdx,rdx
loop_strlen:
        cmp BYTE PTR [rsi + rdx],0
        je loop_strlen_done
        inc rdx
        jmp loop_strlen
loop_strlen_done:

        mov rax, 1
        mov rdi, 1
        # syscall instruction over writes, clobbers, rcx
        # (see intel manuals for details)
        syscall

        mov rax, 1
        mov rdi, 1
        push rsi
        mov rsi, OFFSET newline
        mov rdx, 1
        # syscall instruction over writes, clobbers, rcx
        # (see intel manuals for details)
        syscall

        pop rsi
        pop r11
        pop rcx
        pop rdx
        pop rdi
        pop rax

        ret

        .section .data
newline:        .byte '\n

flowers:
        .quad lilystr
        .quad 5
        .quad 3
        .quad irisstr
        .quad 7
        .quad 3
        .quad buttercupstr
        .quad 4
        .quad 5
        .quad wildrosestr
        .quad 2
        .quad 5
        .quad larkspurstr
        .quad 9
        .quad 5
        .quad columbinestr
        .quad 2
        .quad 5
        .quad delphiniumsstr
        .quad 3
        .quad 8
        .quad ragwortstr
        .quad 4
        .quad 13
        .quad cinerariastr
        .quad 1
        .quad 13
        .quad asterstr
        .quad 2
        .quad 21
        .quad chicorystr
        .quad 1
        .quad 21
        .quad plantainstr
        .quad 8
        .quad 34
        .quad pytethrumstr
        .quad 10
        .quad 34

names:
lilystr:        .asciz "lily"
irisstr:        .asciz "iris"
buttercupstr:   .asciz "buttercup"
wildrosestr:    .asciz "wild rose"
larkspurstr:    .asciz "larkspur"
columbinestr:   .asciz "columbine"
delphiniumsstr: .asciz "delphiniums"
ragwortstr:     .asciz "ragwort"
cinerariastr:   .asciz "cineraria"
asterstr:       .asciz "aster"
chicorystr:     .asciz "chicory"
plantainstr:    .asciz "plantain"
pytethrumstr:   .asciz "pytethrum"
```