Assembly Programming

X86_64 101

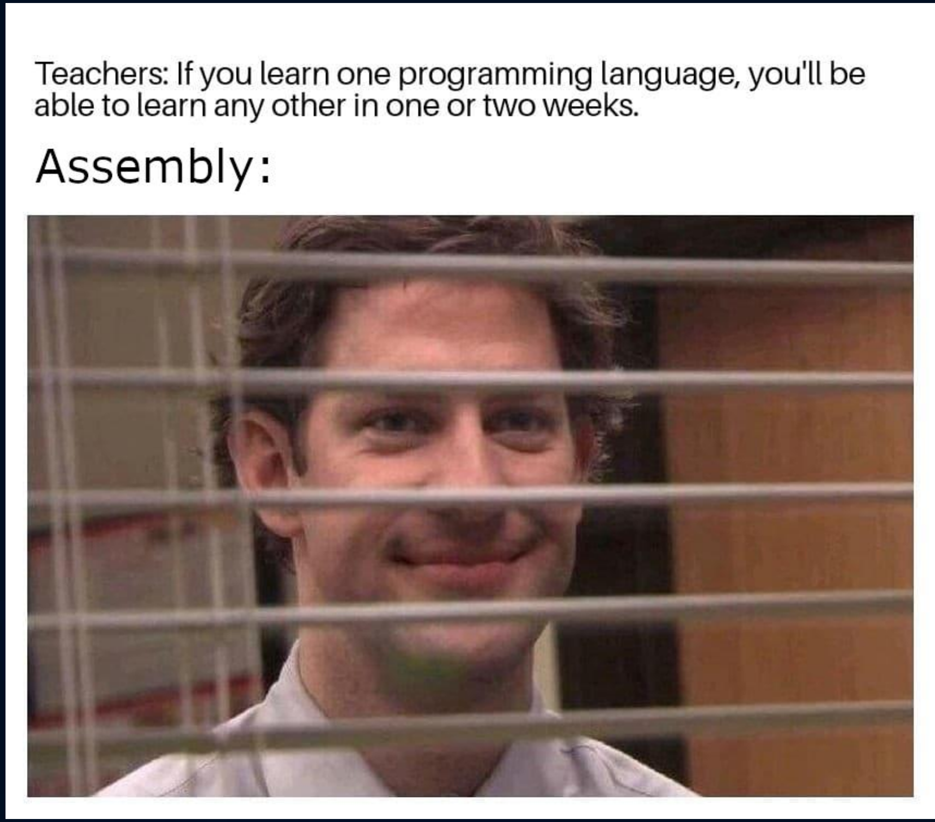
By @sleepunderflow

What is assembly

- Assembly – Low level, architecture (and platform) specific programming language that translates (almost) 1:1 to machine code, we will use x86_64 assembly under GNU Linux
- Assembler – Program used to assemble (convert) assembly language into a machine code. There is many different assemblers depending on what feature set you want, what architecture and platform etc.) (we will use yasm)
- Machine code – the code directly executed by a CPU (essentially numbers)
- X86_64 – 64-bit version of Intel's x86 platform (also known as AMD64)

Sample code (Hello world)

```
global _start:
_start:
   xor rax, rax
   mov qword r10, -1
   add r10, 3
   sub r10
   xchg rax, r10
   xor rdi, rdi
   inc rdi
   xor ebx, ebx
   push ebx
   push 0x21646c726f57206f
   push 0x6c6c6548
   mov rsi, rsp
   mov rdx, 13
   syscall
```



Hello world (serious)

```
global _start
_start:
  mov qword rax, 1
  mov qword rdi, 1
  mov rsi, text
  mov rdx, text_length
  syscall
text:    db "Hello world", 0x0a, 0x00
text_length:    equ $ - text
```

Assembly formats – Intel vs AT&T

- There are two competing assembly formats – Intel and AT&T
- Intel – used by nasm/yasm, radare and in Intel's SDM, this is the one and only true format and we will use this one
  - MOV DWORD [rbp + 4*rdx -8], rax
- AT&T – used by GCC, G++, GDB, objdump (although in most cases can be switched to use Intel format instead)
  - movl -8(%rbp, %rdx, 4), %rax

Linux specific assembly

- Syscalls
  - Used to call kernel exposed functions
  - Required to interact with files (that includes devices, terminal, shared memory, networking, everything)
  - Required to allocate and free memory on the lowest level
  - Deals with timers and executing other processes
  - A lot more
  - Two different formats, two different sets of values, two different speeds!

SYSCALLS - continued

- 32-bit:
  - Uses interrupt 0x80 (int 0x80)
  - Slower
  - Only supports addresses up to 32 bits
  - It is also present in 64-bit system (restricted to 32 bit values!) and is not traced by default by strace in this case
  - Uses eax for syscall number and ebx, ecx, edx, esi, edi and ebp for parameters
  - Return value in eax

SYSCALLS - continued

- 64-bit
  - Uses a SYSCALL instruction
  - Faster
  - Supports full 64-bit range
  - Has a different syscall table (in many cases simplified)
  - Uses rax for syscall code and rdi, rsi, rdx, r10, r8 and r9 for arguments
  - Return value in rax

CPU rings

- On x86_64 platform, in long (64-bit) mode, there are 4 rings defined by only 2 are really used (both by Unix and Windows)
- Ring 0 – kernel mode
- Ring 3 – user mode
- Ring 3 (our code) does not have access to ring 0 code and to many instructions, direct access to hardware, some registers, etc. (hence SYSCALLS and interrupts)

CPU modes

- Real mode – 16 bit, that's what the CPU starts at
- Protected (legacy) mode – 32 bit
- Long mode – 64 bit ← that's what we care about today

Addressing

- 32-bit:
  - 32-bit registers
  - 32-bit addresses
- 64-bit:
  - 64-bit registers
  - ~~64~~-bit addresses
  - 48-bit addresses
    - 0x0000000000000000 – 0x00007fffffffffff
    - 0xffff800000000000 – 0xffffffffffffffff

Registers

- General purpose registers:
  - rax (eax, ax, al, ah)
  - bax (ebx, bx, bl, bh)
  - rcx, rdx, rdi, rsi, r8-r15
- General purpose registers with additional functions:
  - rsp – stack pointer (top of the stack)
  - rbp – base pointer (bottom of the stack)
- Control registers:
  - rip – instruction pointer – points to the next instruction to be executed
  - rflags – flags register
  - cs, ds, es, ss, fs, gs – segment registers (out of scope)
  - xmm0-15, mmx0-7 – floating point and vector registers (out of scope)
  - MSRs – model specific registers – not accessed directly and out of scope

Little endianness

- X86 and x86_64 are little-endian, that means that they use so called reverse byte order
- Big endian:
  - 01 23 45 67 89 ab cd ef in memory translates to 0x0123456789abcdef
- Little endian:
  - 01 23 45 67 89 ab cd ef in memory translates to 0xefcdab8967452301

Main addressing modes

- Direct – go to this address and take a value for there
- Indirect – (uses ModR/M+SIB) – calculate the address where to go to, take a value from there and use it as an address from which to take the final value
- RIP-relative – calculate the final address to take the value from as value from RIP (next instruction address) + immediate value
- Immediate value – take the value from the operand as a final value, doesn't hit the memory (apart from the instruction and operand fetch ofc)

Address and operand sizes

- 4 value sizes defined:
  - BYTE – 1 byte
  - WORD – 2 bytes (different from how Windows understands WORD which is 4 bytes)
  - DWORD – double word (4 bytes)
  - QWORD – quadruple word (8 bytes)
- By default, for most instructions in 64-bit mode (unless the exploicitly operate on other sizes) the sizes are:
  - 64 bit (8 bytes) for addresses
  - 32 bit (4 bytes) for operands
- Default sizes can be modified by prefixes!

Linux file format

- The main executable format we will be using is ELF (Executable and Linkable Format)
- Universal format for executables, libraries, intermediate object files etc. under Linux and Unix (similar to Windows using PE (Portable Executable)
- Extension doesn't matter (as it's Linux) but it is recommended to use:
  - No extension for executable files (exe equivalent for Windows)
  - .so for shared libraries (like DLLs)
  - .o for intermediate object files that still need linking

Signals

- Types of interrupts sent to a process, predefined list of hardcoded numbers
- Have default handlers but can be changed
- Most important ones for us:
  - SIGILL – illegal instruction
  - SIGTRAP – debug breakpoint
  - SIGFPE – floating-point exception
  - SIGKILL – kill – can’t be caught or ignored
  - SIGTERM – please commit suicide

How to assemble

- Create a .asm file (and write your code)
- Use yasm (or nasm) to compile using the following command:
  - yasm -f elf64 source.asm -o intermediate.o
- Use LD (linker) to compile the final executable (alternatively, if you want to use libc use gcc instead but that is out of scope for this session)
  - ld intermediate.o -o executable
- Run
  - ./executable
- ???
- Profit

INSTRUCTIONS

THE BASIC ONES

MOV

- Copies data:
  - Register to register
  - Register to memory
  - Memory to register
  - Immediate value to register
  - Immediate value to memory
- Default operand size: 32 bits

ADD, SUB, INC, DEC

- ADD and SUB add and subtract values from the memory or register
- INC and DEC increment and decrement register/memory by 1
- Default operand sizes – 32 bit
- INC and DEC don’t accept second operand, only what to increment/decrement

POP and PUSH

- Put the value from the register/memory onto the stack or take the value from the stack and put it to a register/memory
- Usually 64-bit operands, can sometimes be 16 or 32 bit
- Top of the stack SHOULD be 16-byte aligned
- Pushing and popping modify the RSP register!

INT / SYSCALL

- INT – call a specific software interrupt (take the address from the interrupt vector table)
  - On Unix interrupt 0x80 (128) is used for system calls
- SYSCALL – the 64-bit equivalent of INT 0x80
  - Fast system call

JMP, Jcc

- Jump to a given address
- JMP – unconditional jump
- Jcc – jump if condition (cc) is met
  - JZ/JE – jump if tested value (difference) is 0
  - JNZ/JNE – jump if not zero
  - JG – jump if greater than, unsigned
  - JA – jump if above - signed

TEST, CMP

- Compares two values without modifying participating registers, only changes RFLAGS register which is later used by Jcc and MOVcc instructions
- SUB has similar effect on flags (TEST subtracts values as well) just also modifies target register

CALL, RET

- CALL – jump to a given address but before that PUSH to the top of the stack a return address (current value of RIP, address of next instruction)
- RET – POP the value from the stack into the RIP register (jump back to the original code)
  - Hint: if the version of RET instructions is used that accepts an operand, it will also add a given value to the stack pointer (clean up allocated space on the stack)

OR, XOR, AND

- That should be obvious

More notable instructions

- DTABWC - Do That Again But Without Crashing
- PMOVLAY - Program Memory Overlay Register Update
- SUBX4 - Subtract With Shift By 2
- MDFM - Modulus Floating Point Multi Reg
- JTNJ - Jump To Next JTNJ
- JBP - Jump If Branch Predicted
- MOVBE - Move, Convert to Big Endian
- CZSAPLSWS - Copy Null Terminated UTF8 String And Pad Left Side With Spaces
- MPNB - Mispredict Next Branch

Useful syscalls

- 0, 1, 2, 3 – read, write, open, close
- 9, 10, 11 – mmap, mprotect, munmap – allocate/deallocate memory and modify permissions
- 33 – dup2 – duplicate handle to a file
- 41, 42, 43, 49, 50 – socket, connect, accept, bind, listen
- 57, 59 – fork, execve
- 60 – exit
- 104-124 – various forms of setuid, getuid, setguid etc.

- Use man pages (2) for details and strace to see what syscalls executable makes

Resources

- Intel's software developer manual – description of (almost) all instructions and entire architecture: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4
- 64 bit Linux syscall table: https://filippo.io/linux-syscall-table/
- 32 bit Linux syscall table: https://syscalls.kernelgrok.com/
- ELF specification: https://uclibc.org/docs/elf-64-gen.pdf and https://refspecs.linuxbase.org/elf/elf.pdf

Today’s task

- Create:
  - Hello World
  - Program that spawns /bin/bash on execution

Homework

- Create:
  - Program that accepts password from the user and if it’s correct accepts text from user and writes it to a file