DLDCA Lab 2: Assembly
Week 1

August 26, 2025

Introduction

In this lab we are going to start with a simple set of problems through which we will understand the basics of x86 assembly syntax and semantics.

Necessary tools

We shall be using `nasm` in order to compile x86 assembly code.

Before starting the lab, ensure that the tool is installed in your system. This can be verified by running the following:

```
$ nasm
nasm: fatal: no input file specified
Type nasm -h for help.
```

Structure of submission/templates

```
your-roll-no/
|- task1/
    |- add-nums.asm (TODO)
    |- compile.sh
|- task2/
    |- run-gdb.asm
    |- compile.sh
    |- mod-regs.txt (TODO)
```

The folder structure of the expected submission for this lab is given above. We expect this folder to be compressed into a tarball with the naming convention of your-roll-no.tar.gz. The command given below can be used to do the same

```
$ tar -cvzf your-roll-no.tar.gz your-roll-no/
```

Tasks

An overview of tasks can be seen in `TODO.md`. **Please go through this file before starting the lab**.

Task 1: Add two numbers

The goal of this task is to create a simple assembly code which loads two numbers from memory into **registers**, adds them and stores their result in a particular register (`rax`).

We have also provided in the template code (`task1/add-nums.asm`), some code to print the value stored in the `rax` register to stdout, you should try to understand what is happening (the sample code has comments :)) and maybe mess around with the given sample code to better grasp it.

To add two numbers you should use the `add` instruction, syntax for it has been provided in the template code itself. To load from memory, use the `mov` instruction, again, the syntax for this instruction has been provided.

To run your code, you may run:

```
$ bash compile.sh
$ ./add-nums
```

OR you can do it manually using,

```
nasm -f elf64 add-nums.asm -o add-nums.o
ld add-nums.o -o add-nums
./add-nums
```

The output of this program should just be:

```
$ ./add-nums
12
```

Task 2: Understanding x86 asm using gdb

The goal of this task is to understand the concept of the **system call**, `caller-saved` and `callee-saved` registers.

What is a system call?

A system call (or `syscall`) can be thought of as a function that the OS (Operating System) provides. That is, the user provides a set of parameters in various registers (`rdi`, `rsi`, `rdx`...) and a specific number, called the **syscall number** in the `rax` register.

Depending upon the syscall number, the OS executes a different function on the parameters.

Types of registers

To execute syscalls, the OS will modify the value of certain registers and leave some other registers' values unchanged. Furthermore, in assembly, you can externally import functions such as C library functions, these functions too may modify certain register values!

Thus, there is a convention as to which registers are to have their values modified. The registers that may have their values modified over a function or system call are called `caller-saved` (or *volatile*) registers, and the ones that will not have their values modified are called `callee-saved` (or *non-volatile*) registers.

The task itself

You are given a piece of code that prints "Hello" to the standard output. You have to run this code through gdb, instructions for doing so have been given in `TODO.md`. You have to report the list of registers modified across the `write` syscall in `mod-regs.txt`.

1 Assembly instructions cheatsheet (non-exhaustive)

Instruction	Operands	Effect
mov	reg, reg/∗mem/imm	reg ← reg / ∗mem / imm
mov	∗mem, reg	∗mem ← reg
add	reg, reg/∗mem/imm	reg ← reg + reg/∗mem/imm
add	∗mem, reg/imm	∗mem ← ∗mem + reg/imm
syscall		Performs a syscall with syscall number stored in rax Arguments passed in other registers starting from rdi

Table 1: Common x86-64 Instructions and Their Effects