TD 3: x86-32 and x86-64 assembly (Part 1)

The goal of these exercises is to get you acquainted with x86-32 and x86-64 assembly language. Either to understand small programs or to write even smaller programs. We will go through the usage of tools (compilers, debuggers, disassemblers) and, then, write small assembler programs.

1 Get your Feet Wet!

1. Compile the following assembler code (`asm.s`) with :

   ```
   gcc -Wall -Wextra -m64 -no-pie -g -o asm asm.s
   ```

   ```
   .globl main

   main:
           movq $0, %rax
           movq $8, %rbx
           cmpq %rax, %rbx
           jle L0
           incq %rbx
           ret
   L0:
           decq %rbx
           ret
   ```

2. Use `gdb` to follow the execution of the program. First type `control-x` then `2`, to switch the display layout. Repeat that until the registers panel shows up. Then use the `gdb` commands (or their shortcuts in parentheses) :
   — 'break main' (b main),
   — 'run' (r),
   — 'nexti' (ni),
   — 'stepi' (si),
   — 'disassemble' (disas),
   — 'info registers' (i r),
   — 'print $rax' (p $rax).
3. Try to force the program to go to `L0` by setting the value of the register `rax` to `9` : `set $rax = 9`
4. Use the command 'objdump -d ./asm' to see the whole disassembled program and its opcodes.
5. Write a small piece of code `loop.s` in x86-64 assembly, that loops ten times and then, terminates.
6. Rewrite and compile the previous programs (asm.s and loop.s) in 32bits by changing the gcc option from `-m64` to `-m32`
7. Try to see what is the effect of the option `-nostdlib` on the executable when you add it to the compilation line. Solve the compilation errors generated by this change and get a working example (even if it segfault at the end, do not worry about that, we'd have to make an exit system call, we'll see that later). In the following we'll stay with a `main` and not use `-nostdlib`.
8. Use the gcc builtin `__asm__()` to insert assembler instructions in C code :

```c
#include <stdlib.h>
int main () {
        int foo = 1;
        __asm__ ("movl %%eax, %%ebx\n"
        "movl $56, %%esi\n"
        "addl %%ecx, %%ebx\n"
        "addb %%ah, %%bh"::"a" (foo):"ebx", "esi");
        return EXIT_SUCCESS;
}
```

This is called *inline assembly* and is very complex to use in practice, so we won't use it this semester, but sometimes we have to use it.

2 Warming up!

1. Write a program that computes the factorial of 8 in `eax`.
2. Write a program that returns the 25th term of the Fibonacci sequence in eax (remember that the Fibonacci sequence is given by $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$).
3. Write a program that compares two given strings with the `repe cmpsb` instruction and stops at the addresses where they differ.
4. Write a program that computes the area of a disk of radius 10 (use the FPU to compute $\pi.r^2$ and store the result in `st(0)`. You can use `.data foo: .long 0` to create a 32-bit variable called `foo`).

3 Programming with x86 System Calls

If the following, to make sure how you eventually called system calls, use `strace ./yourprogram`

1. Write an "Hello World!" program in assembler using write and exit systems calls. First make a 32bit version that uses the '`int $0x80`' instruction and, then, make a 64bit version that uses the '`syscall`' instruction (and the different system call number and parameter convention)
2. Write a program that takes one chararacter from `stdin` (i.e. in C `read(0, &c, 1)`) and writes it to `stdout` (i.e. in C `write(1, &c, 1)`)
3. Write an assembler program that starts '`/bin/ls`' (`execve`). See `man execve` for the details on the parameters : yes, you have to pass a pointer to `"/bin/ls"` both as first argument *and* as first element of the argv array parameter. For envp, you can pass an array that just contains NULL (i.e. pointer value 0).
4. Extend your program to call `/bin/ls /tmp` instead of the shell
5. Extend your program to first call `fork`, in the child (the value returned by `fork`, `eax`, is zero), execute `nc -l -p 1234` which will represent a backdoor that runs in the background. To check what happens in both processes, use `strace -f ./yourprogram` and you can connect to `nc` with `telnet localhost 1234`