Writing, compiling and linking assembly programs

DAT103

October 8, 2021

Taking care of prerequisites

In order to actually write, compile, link and run some assembly programs we will need to set up our environment. Let us assume we are starting with a fresh virtualbox machine with the previously provided ubuntu image. The following assumes you are in your virtualbox Ubuntu instance.

Installing nasm, binutils and ddd

Let us open up a terminal and install nasm. For this we will use the package manager apt available in Ubuntu. A handy tool to query available packages is `apt-cache`. For example try running the command `apt-cache search 'netwide assembler'`. It should output some lines one of which should be:

```
nasm - General-purpose x86 assembler
```

We need this so install it by using the package manager command `apt-get`, run the below command

```
sudo apt-get install nasm binutils ddd xterm
```

This will install `nasm`; the assembler we will be using, `binutils`; a couple of useful programs among them the linker we will need, `ddd` a useful program for debugging our assembly program, and `xterm` which will enable additional functionality inside `ddd` (run in execution window).

We should now be good to go on to writing, compiling, linking and running a simple assembly program!

Writing a simple assembly program

Use an editor such as `vi` and enter the following program; i.e. `vi hello.asm`.

```
; Hello Word in nasm

; Constants
  cr equ 13                          ; define a constant cr (carriage-return) equal to 13
  lf equ 10                          ; define a constant lf (line-feed) equal to 10

section .data                        ; start writing the .data segment
  message db 'Hello World!',cr,lf
  length equ $ - message

section .text                        ; start writing the .text segment
global _start                        ; declare _start as a global symbol
_start:                              ; create the label _start
  mov edx,length
  mov ecx,message
  mov ebx,1
  mov eax,4                          ; system call 4 in x86 Linux kernel is sys_write
  int 80h
  mov ebx,0
  mov eax,1                          ; system call 1 in x86 Linux kernel is sys_exit
  int 80h
```

Compiling and linking the program

Having entered the program in a file `hello.asm` it is now time to compile it, in your terminal type in the command shown below.

```
nasm -f elf -F dwarf -g hello.asm
```

If you're wondering what all that means a good way to find out is to either consult the manpage for `nasm`, i.e. running `man nasm`, or by running nasm with the help option; `nasm --help`. This will tell you that the option we set with `-f elf` sets the assembled output file format to the Executable and Linkable Format (ELF). Furthermore the option we set with `-F dwarf` set the debug information format to `dwarf` and the option `-g` enabled the generation of debug information.

We have now assembled an object file `hello.o`, which cannot be executed. In order to get an executable file we must first perform linking with our linker `ld`. Now link the program as shown below.

```
ld -m elf_i386 -o hello hello.o
```

It can be useful to know of tools like `objdump` to explore executables, for instance try out the command `objdump -d hello` and see if you can recognize the output.

Now execute the program and see that it does the expected; prints "Hello World!".

Exploring this program with a graphical debugger

Now you will step through the program with `ddd` (the Data Display Debugger) to get some experience with a simple to use debugger.

```
ddd hello
```

Perform these steps

- Click at the line containing the system call `int 80h`
- Click the Break icon (now you've set a breakpoint)
- Click run and observe that the program has halted waiting to perform this instruction

- Click status and select registers, observe that `eax` is set to 4 as expected
- Click step and observe that when a step is taken, or in other words an instruction more is performed, a "Hello World!" is show as output.

This can be a useful tool for you to check what state your machine is in at certain points.

Bonus: Exploring this program with gdb

It may be useful, but slightly more confusing to explore program with the gnu debugger (gdb). To do that perform the command `gdb hello` and perform the following steps.

- Enter `list`, this will show you 10 lines of the program. Entering it again will advance to the next 10 lines and so on. Advance until you see `_start` somewhere.
- Let's look at the program from the label `_start`, if it's on line n then enter `list n,`. This should show the program from line n and onwards.
- To understand the command you can read some documentation through entering `help list`
- Now let's set a breakpoint the same place as we did before. Find the line where the system call is, if it's on line m then enter `break m`.
- We can show all breakpoints with `info break`.
- To make the program start running we enter the command `run`, and we will observe that it hits the breakpoint.
- Now we want to examine register `eax`, so type in `info registers eax` and observe that it is set to the value 4.
- Now to step one instruction further we simply type in `step` and observe that it shows the expected output as well as the instruction we end up in (at line m+1).
- end the debugging session by typing `quit`

Most commands given to gdb have non-verbose versions you can look up!

Things to ponder

How do we figure out the system call tables?

You can dig up the system call table in the Linux kernel sources, recall we are using 32bit x86 Linux kernel and these can be found documented in the kernel source here or you can view the nicely formatted table here be sure to remember to look only at the tables x86 (32 bit) relevant parts.

Let's try to understand what happens in the code, by looking at the table entry for write. You will find the entry

NR	syscall name	references	eax	arg0 (ebx)	arg1 (ecx)	arg2 (edx)	...
				⋮			
4	write	man/ cs/	0x04	unsigned int fd	const char *buf	size_t count	...
				⋮			

Which tells you (if you look at the manual page as well) the calling convention for making a write system call

- eax should have value 0x04
- ebx contains the 0th arg; should have the file descriptor (1 is standard out)
- ecx contains the 1th arg; should contain the address of the buffer where our message (to be written out) resides
- edx contains the 2th arg; should contain the count in bytes of our message

Now if you look at the manual you will see that write has a return value. Where is it placed? The relevant parts of the table found in the last source shows this information, reproduced below.

arch	syscall NR	return	arg0	arg1	arg2	arg3	arg4	arg5
x86	eax	eax	ebx	ecx	edx	esi	edi	ebp

Now this is rather arcane and useless information, except for being able to understand why exactly the sequence of assembly instructions are as they are (for sys_write) in the shown example in this exercise.

Now look up the relevant information in the table for sys_exit.