
```
[ Writing A Useful Program With NASM ]

        by Jonathan Leto
        Version 1.0 -  Sun Dec 17 17:46:38 EST 2000

[ Intro ]

Much fun can be had with assembly programming, it gives you a much deeper
understanding about the inner workings of your processor and kernel. This
article is geared towards the beginning assembly programmer who can't seem to
justify why he is doing something as masochistic as writing an entire program in
assembly language. If you don't already know one or more other programming
languages, you really have no business reading this. Many constructs will also
be explained in terms of C. You should also be familiar with the command line
options of NASM, no sense going over them again here.

[ Getting Started ]

So you want to write a program that actually DOES something. "Hello, world"
isn't cutting it anymore. First, an overview of the various parts of an
assembly program: (For terse documentation, the NASM manual is the place to go.)

[ The .data section ]

This section is for defining constants, such as filenames or buffer sizes,
this data does not change at runtime. The NASM documentation has a good
description of how to use the db,dd,etc instructions that are used in this
section.

[ The .bss section ]

This section is where you declare your variables.
They look something like this:

        filename:       resb    255     ; REServe 255 Bytes
        number:         resb    1       ; REServe 1 Byte
        bignum:         resw    1       ; REServe 1 Word (1 Word = 2 Bytes )
        longnum:        resd    1       ; REServe 1 Double Word
        pi:             resq    1       ; REServe 1 double precision float
        morepi:         rest    1       ; REServe 1 extended precision float

[ The .text section ]

This is where the actual assembly code is written. The term "self modifying code"
means a program which modifies this section while being executed.


[ In The Beginning ... ]

The next thing you probably noticed while looking at the source to various
assembly programs, there always seems to be "global _start" or something similar
at the beginning of the .text section. This is the assembly program's way of
telling the kernel where the program execution begins. It is exactly, to my
knowledge, like the main function in C, other than that it is not a function,
just a starting point.

[ The Stack and Stuff ]

Also like in C, the kernel sets up the environment with all of the environment
variables, and sets up **argv and argc. Just in case you forgot, **argv is an
array of strings that are all of the arguments given to the program, and argc
is the count of how many there are.  These are all put on the stack. If you
have taken Computer Science 101, or read any type of introductory computer
science book, you should know what a stack is. It is a way of storing data so
that the last thing you put in is the first that comes out. This is fine and
dandy, but most people don't seem to grasp how this has anything to do with
their computer. "The stack" as it is ominously referred too, is just your RAM.
That's it.  It is your RAM organized in such a way, so that when you "push"
something onto "The stack", all you are doing is saving something in RAM. And
when you "pop" something off of "The stack", you are retrieving the last thing
you put in, which is on the top.

Ok, now let's look at some code that you are likely to see.
```

```
        section .text           ; declaring our .text segment
                global  _start  ; telling where program execution should start

        _start:                 ; this is where code starts getting exec'ed
                pop     ebx     ; get first thing off of stack and put into ebx
                dec     ebx     ; decrement the value of ebx by one
                pop     ebp     ; get next 2 things off stack and put into ebx
                pop     ebp
```

What does this code do? It simply puts the first actual argument into the ebx register. Let's say we ran the program on the command line as so:

```
        $ ./program 42 A
```

When we are are on the _start line, the stack looked something like this:

```
        -----------
        | 3       |     The number of arguments, including argv[0],
        |         |     which is the program name
        -----------
        |"program"|     argv[0]
        -----------
        | "42"    |     argv[1] NOTE: This is the character "4" and "2",
        |         |     not the number 42
        -----------
        | "A"     |     argv[2]
        -----------
```

So, the first instruction, "pop ebx", took the 3, and put it into ebx. Then we decrement it by one, because the program name isn't really an argument.

Depending on if you need to later use the argument count later on, you will see other arguments put into either the same register or a different one.

Now, "pop ebp" puts the program name into ebp, and then the next "pop ebp" overwrites it, and puts "42" into ebp. The last value of ebp is not preserved, and since you have popped it off of the stack, it is gone forever.

[Doing more interesting things]

Moving on, how exactly do you interact with the rest of the system? You know how to manipulate the stack, but how to you get the current time, or make a directory, or fork a process, or any other wonderful thing a Unix box can do? I am pleased to introduce you to the "system call". A system call is the translator that lets user-land programs (which is what you are writing), talk to the kernel, who is in kernel-land, of course. Each syscall has a unique number, so that you can put it into the eax register, and tell the kernel "Yo, wake up and do this", and it hopefully will. If the syscall takes arguments, which most do, these go into ebx,ecx,edx,esi,edi,ebp , in that order.

Some example code always helps:

```
        mov     eax,1           ; the exit syscall number
        mov     ebx,0           ; have an exit code of 0
        int     80h             ; interrupt 80h, the thing that pokes the kernel
                                ; and says, "do this"
```

The preceding code is equivalent to having a "return 0" at the end of your main function. Ok, ok, still not very useful, but we are getting there.

A more useful example:

```
        pop     ebx             ; argc
        pop     ebx             ; argv[0]
        pop     ebx             ; the first real arg, a filename


        mov     eax,5           ; the syscall number for open()
                                ; we already have the filename in ebx

        mov     ecx,0           ; O_RDONLY, defined in fcntl.h

        int     80h             ; call the kernel

                                ; now we have a file descriptor in eax
```

```
        test    eax,eax         ; lets make sure it is valid
        jns     file_function   ; if the file descriptor does not have the
                                ; sign flag ( which means it is less than 0 )
                                ; jump to file_function

        mov     ebx,eax         ; there was an error, save the errno in ebx
        mov     eax,1           ; put the exit syscall number in eax
        int     80h             ; bail out
```

Now we are starting to get somewhere. You should be starting to realize that there is no black magic or voodoo in assembly programming, just a very strict set of rules. If you know how the rules work, you can do just about everything. Though I haven't tried it, I have seen network coding in assembly, console graphics (intros!), and yes, even X windows code in assembly.

So where do find out all of the semantics for all of the various system calls? Well first, the numbers are listed in asm/unistd.h in Linux, and sys/syscall.h in the *BSD's. To find out information about each one, such as what arguments they take and what values they return, look no further that your man pages! I will hold your hand in finding out about the next syscall we are going to use, read() .

"man read" didn't give you exactly what you wanted did it? That is because program manuals and shell manuals are shown before the programming manuals are. If you are using bash, you probably are looking at the BASH_BUILTINS(1) man page. To get to what you really want, try "man 2 read". Now you should be looking at sections like SYNOPSIS, DESCRIPTION, DESCRIPTION, ERRORS and a few others. These are the most important. Take a look at synopsis, it should look like:

```
        ssize_t read(int fd, void *buf, size_t count);
```

NOTE: ssize_t and size_t are just integers .

The first argument is the file descriptor, followed by the buffer, and then how many bytes to read in, which should be however long the buffer is. For the best performance, use 8192, which is 8k, as your count. Make your buffer a multiple of this, 8192 is fine. Now you know what to put in your registers. Reading the RETURN VALUE section, you should see how read() returns the number of bytes it read, 0 for EOF, and -1 for errors.

```
file_function:
        mov     ebx,eax         ; sys_open returned file descriptor into eax
        mov     eax,3           ; sys_read
                                ; ebx is already setup
        mov     ecx,buf         ; we are putting the ADDRESS of buf in ecx
        mov     edx,bufsize     ; we are putting the ADDRESS of bufsize in edx

        int     80h             ; call the kernel

        test    eax,eax         ; see what got returned
        jz      nextfile        ; got an EOF, go to read the next file
        js      error           ; got an error, bail out

                                ; if we are here, then we actually read some bytes
```

Now we have a chunk of the file read (up to 8192 bytes), and sitting in what you would call an array in C. What can you do now? Well, the first thing that comes to mind is print it out. Wait a sec, there is no man page for printf in section 2. What's the deal? Well, printf is a library function, implemented by good ol' libc. You are going to have to dig a little deeper, and use write(). So now you looking at the man page. write() writes to a file descriptor. What the hell good does that do me? I want to print it out! Well, remember, everything in Unix is a file, so all you have to do is write to STDOUT. From /usr/include/unistd.h, it is defined as 1 . So the next chunk of code looks like:

```
        mov     edx,eax         ; save the count of bytes for the write syscall
        mov     eax,4           ; system call for write
        mov     ebx,1           ; STDOUT file descriptor
                                ; ecx is already set up
        int     80h             ; call kernel

        ; for the program to properly exit instead of segfaulting right here
        ; ( it doesn't seem to like to fall off the end of a program ), call
        ; a sys_exit
```

```
        mov     eax,1
        mov     ebx,0
        int     80h
```

What you have now just written is basically "cat", except it only prints the first 8192 bytes.

[Portability]

In the preceding section, you saw how the call the kernel in Linux with NASM. This is fine if you are never ever going to use another operating system, and you enjoy looking up the system kernel numbers, but is not very practical, and extremely unportable. What to do? There is a great little package called asmutils started by Konstantin Boldyshev, who runs linuxassembly.org . If you haven't read all of the good documentation on that site, that should be your next step. Asmutils provides an easy to use and portable interface to doing system calls in whichever Unix variant you use (and even has support for BeOS.) Even if you aren't interesting in using these Unix utilities that are rewritten in assembly, if you want to write portable NASM code, you are better off using it's header files than rolling your own. With asmutils, your code will look like this:

```
        %include "system.inc"   ; all the magic happens here

        CODESEG                 ; .text section

        START:                  ; always starts here

        sys_write STDOUT,[somestring],[strlen]

        END                     ; code ends here
```

This is much more readable then doing everything by system call number, and it will be portable across Linux,FreeBSD,OpenBSD,NetBSD,BeOS and a few other lesser known OS's. You can now use system calls by name, and use standard constants like STDOUT or O_RDONLY, just like in C. The "%include" statement works precisely as it does in C, sourcing the contents of that file.

To learn more about how to use asmutils, read the Asmutils-HOWTO, which is in the doc/ directory of the source. Also, to get the latest source, use the following commands:

```
export CVS_RSH=ssh
cvs -d:pserver:anonymous@cvs.linuxassembly.org:/cvsroot/asm login
cvs -z3 -d:pserver:anonymous@cvs.linuxassembly.org:/cvsroot/asm co asmutils
```

This will download the newest, bleeding edge source into a subdirectory called "asmutils" of your current directory. Take a look at some of the simpler programs, such as cat,sleep,ln,head or mount, you will see that there isn't anything horrendously difficult about them. head was my first assembly program, I made extra comments on purpose, so that would be a good place to start.

[Debugging]

Strace will definitely by your friend. It is the easiest tool to use to debug your problem. Most of the time when writing in assembly, other that syntax errors, you will just get a segmentation fault. This provides you with a ZERO useful information. With strace, at least you will see after which system call your program is choking. Example:

```
        $ strace ./cal2
        execve("./cal2", ["./cal2"], [/* 46 vars */]) = 0
        read(1, "", 0)                          = 0
        --- SIGSEGV (Segmentation fault) ---
        +++ killed by SIGSEGV +++
```

Now you know to look after your first read system call. But it starts getting tricky when you have lots of pure assembly, which strace cannot show. That's when gdb comes into play. There is some very good information about using gdb and enabling debugging information in NASM in the Asmutils-HOWTO, so I won't reproduce it here. For a quick and dirty solution, you could do something like this:

```
        %define notdeadyet      sys_write STDOUT,0,__LINE__
```

Now you can litter the source with notdeadyet's, and hopefully see where things are going astray with the help of strace. Obviously this is not practical for complex bugs or voluminous source, but works great for finding careless mistakes when you are starting out. Example:

```
$ strace ./cal2
execve("./cal2", ["./cal2"], [/* 46 vars */]) = 0
write(1, NULL, 16)                      = 16
write(1, NULL, 26)                      = 26
write(1, NULL, 41)                      = 41
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

Now we know that we are still going on line 41, and the problem is after that.

[Next ?]

Now it is your turn to explore the insides of your operating system, and take pride in understanding what's really going on under the covers.

[Reference]

Places to get more information:

```
Linux Assembly - http://www.linuxassembly.org
NASM Manual ( available in doc/html directory of source )
Assembly Programming Journal - http://asmjournal.freeservers.com/
Mammon_'s textbase - http://www.eccentrica.org/Mammon/sprawl/textbase.html
Art Of Assembly - http://webster.cs.ucr.edu/Page_asm/ArtOfAsm.html
Sandpile - http://www.sandpile.org
comp.lang.asm.x86
NASM - http://www.cryogen.com/Nasm
Asmutils-HOWTO - doc/ directory of asmutils
```

[Feedback]

Feedback is welcome, hopefully this was of some use to budding Unix assembly programmers.

[Availability]

The most current version of this document should be available at
http://www.leto.net/writing/nasm.php

[Appendix : Jumps]

When I first began looking at assembly source code, I saw all these crazy instructions like "jnz" and the like. It looked like I was going to have to remember the names of a whole slew of inanely named instructions. But after a while it finally clicked what they all were. They are basically just "if statements" that you know and love, that work off of the EFLAGS register. What is the EFLAGS register? Just a register with lots of different bits that are set to zero or one, depending on the previous comparison that the code made.

Some code to set the stage:

```
mov     eax,82
mov     ebx,69

test    eax,ebx
jle     some_function
```

What on earth is "jle"? Why it's "Jump if Less than or Equal." If eax was less than or equal to ebx, code execution will jump to "some_function", if not, it keeps chugging along. Here is a list which will hopefully shed some light on this part of assembly that was mysterious to me when I began. Some of these are logically the same, but are provided because is some situations one will be more intuitive than the other.

Jump	Meaning	Signedness (S or U)
ja	Jump if above	U
jae	Jump if above or Equal	U
jb	Jump if below	U
jbe	Jump if below or Equal	U

```
jc      | Jump if Carry                 |
jcxz    | Jump if CX is Zero            |
je      | Jump if Equal                 |
jecxz   | Jump if ECX is Zero           |
jz      | Jump if Zero                  |
jg      | Jump if greater               |       S
jge     | Jump if greater or Equal      |       S
jl      | Jump if less                  |       S
jle     | Jump if less or Equal         |       S
jmp     | Unconditional jump            |
jna     | Jump Not above                |       U
jnae    | Jump Not above or Equal       |       U
jnc     | Jump if Not Carry             |
jncxz   | Jump if CX Not Zero           |
jne     | Jump if Not Equal             |
jng     | Jump if Not greater           |       S
jnge    | Jump if Not greater or Equal  |       S
jnl     | Jump if Not less              |       S
jnle    | Jump if Not less or Equal     |       S
jno     | Jump if Not Overflow          |
jnp     | Jump if Not Parity            |
jns     | Jump if Not signed            |
jnz     | Jump if Not Zero              |
jo      | Jump if Overflow              |
jp      | Jump if Parity                |
jpe     | Jump if Parity Even           |
jpo     | Jump if Parity Odd            |
js      | Jump if signed                |
jz      | Jump if Zero                  |
-------------------------------------------------------
```

Jonathan Leto - jonathan at leto dot net

Generated Saturday 15th of March 2014 10:29:18 PM