NASM Assembly Language Tutorials - asmtutor.com

LESSONS

Learn Assembly Language

This project was put together to teach myself NASM x86 assembly language on linux.

Github Project » (https://github.com/DGivney/assemblytutorials)

Lesson 1

Hello, world!

_First, some background:

Assembly language is bare-bones. The only interface a programmer has above the actual hardware is the kernel itself. In order to build useful programs in assembly we need to use the linux system calls provided by the kernel. These system calls are a library built into the operating system to provide functions such as reading input from a keyboard and writing output to the screen.

When you invoke a system call the kernel will immediately suspend execution of your program. It will then contact the necessary drivers needed to perform the task you requested on the hardware and then return control back to your program.

Note: Drivers are called *drivers* because the kernel literally uses them to drive the hardware.

We can accomplish this all in assembly by loading EAX with the function number (operation code OPCODE) we want to execute and filling the remaining registers with the arguments we want to pass to the system call. A software interrupt is requested with the INT instruction and the kernel takes over and calls the function from the library with our arguments. Simple.

For example requesting an interrupt when EAX=1 will call *sys_exit* and requesting an interrupt when EAX=4 will call *sys_write* instead. EBX, ECX & EDX will be passed as arguments if the function requires them. Click here to view an example of a Linux System Call Table and its corresponding OPCODES. (https://chromium.googlesource.com/chromiumos/docs/+/HEAD/constants/syscalls.md#x86-32_bit)

_Writing our program:

Firstly we create a variable 'msg' in our .data section and assign it the string we want to output in this case 'Hello, world!'. In our .text section we tell the kernel where to begin execution by providing it with a global label _start: to denote the programs entry point.

We will be using the system call *sys_write* to output our message to the console window. This function is assigned OPCODE 4 in the Linux System Call Table. The function also takes 3 arguments which are sequentially loaded into EDX, ECX and EBX before requesting a software interrupt which will perform the task.

The arguments passed are as follows:

- EDX will be loaded with the length (in bytes) of the string.
- ECX will be loaded with the address of our variable created in the .data section.
- EBX will be loaded with the file we want to write to – in this case STDOUT.

The datatype and meaning of the arguments passed can be found in the function's definition.

We compile, link and run the program using the commands below.

helloworld.asm

```
; Hello World Program - asmtutor.com
; Compile with: nasm -f elf helloworld.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld.o -o helloworld
; Run with: ./helloworld

SECTION .data
msg     db      'Hello World!', 0Ah     ; assign msg variable with your message string

SECTION .text
global  _start

_start:

    mov     edx, 13     ; number of bytes to write - one for each letter plus 0Ah (line feed charact
    mov     ecx, msg    ; move the memory address of our message string into ecx
    mov     ebx, 1      ; write to the STDOUT file
    mov     eax, 4      ; invoke SYS_WRITE (kernel opcode 4)
    int     80h
```

```
~$ nasm -f elf helloworld.asm
~$ ld -m elf_i386 helloworld.o -o helloworld
~$ ./helloworld
Hello World!
Segmentation fault
```

Error: Segmentation fault

Lesson 2

Proper program exit

_Some more background:

After successfully learning how to execute a system call in Lesson 1 we now need to learn about one of the most important system calls in the kernel, *sys_exit.*

Notice how after our 'Hello, world!' program ran we got a Segmentation fault? Well, computer programs can be thought of as a long strip of instructions that are loaded into memory and divided up into sections (or segments). This general pool of memory is shared between all programs and can be used to store variables, instructions, other programs or anything really. Each segment is given an address so that information stored in that section can be found later.

To execute a program that is loaded in memory, we use the global label _start: to tell the operating system where in memory our program can be found and executed. Memory is then accessed sequentially following the program logic which determines the next address to be accessed. The kernel jumps to that address in memory and executes it.

It's important to tell the operating system exactly where it should begin execution and where it should stop. In Lesson 1 we didn't tell the kernel where to stop execution. So, after we called *sys_write* the program continued sequentially executing the next address in memory, which could have been anything. We don't know what the kernel tried to execute but it caused it to choke and terminate the process for us instead - leaving us the error message of 'Segmentation fault'. Calling *sys_exit* at the end of all our programs will mean the kernel knows exactly when to terminate the process and return memory back to the general pool thus avoiding an error.

_Writing our program:

sys_exit has a simple function definition. In the Linux System Call Table it is allocated OPCODE 1 and is passed a single argument through EBX.

In order to execute this function all we need to do is:

- Load EBX with 0 to pass zero to the function meaning 'zero errors'.
- Load EAX with 1 to call *sys_exit.*
- Then request an interrupt on libc using INT 80h.

We then compile, link and run it again.

Note: Only new code added in each lesson will be commented.

helloworld.asm

```
; Hello World Program - asmtutor.com
; Compile with: nasm -f elf helloworld.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld.o -o helloworld
; Run with: ./helloworld

SECTION .data
msg     db      'Hello World!', 0Ah

SECTION .text
global  _start

_start:

    mov     edx, 13
    mov     ecx, msg
    mov     ebx, 1
    mov     eax, 4
    int     80h

    mov     ebx, 0      ; return 0 status on exit - 'No Errors'
    mov     eax, 1      ; invoke SYS_EXIT (kernel opcode 1)
    int     80h
```

```
~$ nasm -f elf helloworld.asm
```

```
~$ ld -m elf_i386 helloworld.o -o helloworld
~$ ./helloworld
Hello World!
```

Lesson 3

Calculate string length

_Firstly, some background:

Why do we need to calculate the length of a string?

Well *sys_write* requires that we pass it a pointer to the string we want to output in memory and the length in bytes we want to print out. If we were to modify our message string we would have to update the length in bytes that we pass to *sys_write* as well, otherwise it will not print correctly.

You can see what I mean using the program in Lesson 2. Modify the message string to say 'Hello, brave new world!' then compile, link and run the new program. The output will be 'Hello, brave ' (the first 13 characters) because we are still only passing 13 bytes to *sys_write* as its length. It will be particularly necessary when we want to print out user input. As we won't know the length of the data when we compile our program, we will need a way to calculate the length at runtime in order to successfully print it out.

_Writing our program:

To calculate the length of the string we will use a technique called pointer arithmetic. Two registers are initialised pointing to the same address in memory. One register (in this case EAX) will be incremented forward one byte for each character in the output string until we reach the end of the string. The original pointer will then be subtracted from EAX. This is effectively like subtraction between two arrays and the result yields the number of elements between the two addresses. This result is then passed to *sys_write* replacing our hard coded count.

The CMP instruction compares the left hand side against the right hand side and sets a number of flags that are used for program flow. The flag we're checking is the ZF or Zero Flag. When the byte that EAX points to is equal to zero the ZF flag is set. We then use the JZ instruction to jump, if the ZF flag is set, to the point in our program labeled 'finished'. This is to break out of the nextchar loop and continue executing the rest of the program.

helloworld-len.asm

```
; Hello World Program (Calculating string length)
; Compile with: nasm -f elf helloworld-len.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-len.o -o helloworld-
; Run with: ./helloworld-len

SECTION .data
msg     db      'Hello, brave new world!', 0Ah ; we can modify this now without having to update any

SECTION .text
global  _start

_start:

    mov     ebx, msg        ; move the address of our message string into EBX
    mov     eax, ebx        ; move the address in EBX into EAX as well (Both now point to the same s

nextchar:
    cmp     byte [eax], 0   ; compare the byte pointed to by EAX at this address against zero (Zero
    jz      finished        ; jump (if the zero flagged has been set) to the point in the code label
    inc     eax             ; increment the address in EAX by one byte (if the zero flagged has NOT
    jmp     nextchar        ; jump to the point in the code labeled 'nextchar'

finished:
    sub     eax, ebx        ; subtract the address in EBX from the address in EAX
                            ; remember both registers started pointing to the same address (see line
                            ; but EAX has been incremented one byte for each character in the messag
                            ; when you subtract one memory address from another of the same type
                            ; the result is number of segments between them - in this case the numbe

    mov     edx, eax        ; EAX now equals the number of bytes in our string
    mov     ecx, msg        ; the rest of the code should be familiar now
    mov     ebx, 1
    mov     eax, 4
    int     80h

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

```
~$ nasm -f elf helloworld-len.asm
~$ ld -m elf_i386 helloworld-len.o -o helloworld-len
~$ ./helloworld-len
Hello, brave new world!
```

Lesson 4

Subroutines

_Introduction to subroutines:

Subroutines are functions. They are reusable pieces of code that can be called by your program to perform various repeatable tasks. Subroutines are declared using labels just like we've used before (eg. _start:) however we don't use the JMP instruction to get to them - instead we use a new instruction CALL. We also don't use the JMP instruction to return to our program after we have run the function. To return to our program from a subroutine we use the instruction RET instead.

_Why don't we JMP to subroutines?:

The great thing about writing a subroutine is that we can reuse it. If we want to be able to use the subroutine from anywhere in the code we would have to write some logic to determine where in the code we had jumped from and where we should jump back to. This would litter our code with unwanted labels. If we use CALL and RET however, assembly handles this problem for us using something called the stack.

_Introduction to the stack:

The stack is a special type of memory. It's the same type of memory that we've used before however it's special in how it is used by our program. The stack is what is call **Last In First Out** memory (LIFO). You can think of the stack like a stack of plates in your kitchen. The last plate you put on the stack is also the first plate you will take off the stack next time you use a plate.

The stack in assembly is not storing plates though, its storing values. You can store a lot of things on the stack such as variables, addresses or other programs. We need to use the stack when we call subroutines to temporarily store values that will be restored later.

Any register that your function needs to use should have it's current value put on the stack for safe keeping using the PUSH instruction. Then after the function has finished it's logic, these registers can have their original values restored using the POP instruction. This means that any values in the registers will be the same before and after you've called your function. If we take care of this in our subroutine we can call functions without worrying about what changes they're making to our registers.

The CALL and RET instructions also use the stack. When you CALL a subroutine, the address you called it from in your program is pushed onto the stack. This address is then popped off the stack by RET and the program jumps back to that place in your code. This is why you should always JMP to labels but you should CALL functions.

helloworld-len.asm

```
; Hello World Program (Subroutines)
; Compile with: nasm -f elf helloworld-len.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-len.o -o helloworld-
; Run with: ./helloworld-len

SECTION .data
msg     db      'Hello, brave new world!', 0Ah

SECTION .text
global  _start

_start:

    mov     eax, msg        ; move the address of our message string into EAX
    call    strlen          ; call our function to calculate the length of the string

    mov     edx, eax        ; our function leaves the result in EAX
    mov     ecx, msg        ; this is all the same as before
    mov     ebx, 1
    mov     eax, 4
    int     80h

    mov     ebx, 0
    mov     eax, 1
    int     80h

strlen:                     ; this is our first function declaration
    push    ebx             ; push the value in EBX onto the stack to preserve it while we use EBX i
    mov     ebx, eax        ; move the address in EAX into EBX (Both point to the same segment in me

nextchar:                   ; this is the same as lesson3
    cmp     byte [eax], 0
    jz      finished
    inc     eax
    jmp     nextchar

finished:
    sub     eax, ebx
    pop     ebx             ; pop the value on the stack back into EBX
    ret                     ; return to where the function was called
```

```
~$ nasm -f elf helloworld-len.asm
~$ ld -m elf_i386 helloworld-len.o -o helloworld-len
~$ ./helloworld-len
Hello, brave new world!
```

Lesson 5

External include files

External include files allow us to move code from our program and put it into separate files. This technique is useful for writing clean, easy to maintain programs. Reusable bits of code can be written as subroutines and stored in separate files called libraries. When you need a piece of logic you can include the file in your program and use it as if they are part of the same file.

In this lesson we will move our string length calculating subroutine into an external file. We fill also make our string printing logic and program exit logic a subroutine and we will move them into this external file. Once it's completed our actual program will be clean and easier to read.

We can then declare another message variable and call our print function twice in order to demonstrate how we can reuse code.

Note: I won't be showing the code in functions.asm after this lesson unless it changes. It will just be included if needed.

functions.asm

```
;-----------------------------------------
; int slen(String message)
; String length calculation function
slen:
    push    ebx
    mov     ebx, eax

nextchar:
    cmp     byte [eax], 0
    jz      finished
    inc     eax
    jmp     nextchar

finished:
    sub     eax, ebx
    pop     ebx
    ret


;-----------------------------------------
; void sprint(String message)
; String printing function
sprint:
    push    edx
    push    ecx
    push    ebx
    push    eax
    call    slen

    mov     edx, eax
    pop     eax

    mov     ecx, eax
    mov     ebx, 1
    mov     eax, 4
    int     80h

    pop     ebx
    pop     ecx
    pop     edx
    ret


;-----------------------------------------
; void exit()
; Exit program and restore resources
quit:
    mov     ebx, 0
    mov     eax, 1
    int     80h
    ret
```

helloworld-inc.asm

```
; Hello World Program (External file include)
; Compile with: nasm -f elf helloworld-inc.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-inc.o -o helloworld-
; Run with: ./helloworld-inc

%include        'functions.asm'                             ; include our external file

SECTION .data
msg1    db      'Hello, brave new world!', 0Ah              ; our first message string
msg2    db      'This is how we recycle in NASM.', 0Ah      ; our second message string

SECTION .text
global  _start

_start:

    mov     eax, msg1       ; move the address of our first message string into EAX
    call    sprint          ; call our string printing function

    mov     eax, msg2       ; move the address of our second message string into EAX
    call    sprint          ; call our string printing function

    call    quit            ; call our quit function
```

```
~$ nasm -f elf helloworld-inc.asm
~$ ld -m elf_i386 helloworld-inc.o -o helloworld-inc
~$ ./helloworld-inc
Hello, brave new world!
This is how we recycle in NASM.
This is how we recycle in NASM.
```

Error: Our second message is outputted twice. This is fixed in the next lesson.

Lesson 6

NULL terminating bytes

Ok so why did our second message print twice when we only called our sprint function on msg2 once? Well actually it did only print once. You can see what I mean if you comment out our second call to sprint. The output will be both of our message strings.

But how is this possible?

What is happening is we weren't properly terminating our strings. In assembly, variables are stored one after another in memory so the last byte of our msg1 variable is right next to the first byte of our msg2 variable. We know our string length calculation is looking for a zero byte so unless our msg2 variable starts with a zero byte it keeps counting as if it's the same string (and as far as assembly is concerned it is the same string). So we need to put a zero byte or 0h after our strings to let assembly know where to stop counting.

Note: In programming 0h denotes a null byte and a null byte after a string tells assembly where it ends in memory.

helloworld-inc.asm

```
; Hello World Program (NULL terminating bytes)
; Compile with: nasm -f elf helloworld-inc.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-inc.o -o helloworld-
; Run with: ./helloworld-inc

%include        'functions.asm'

SECTION .data
msg1    db      'Hello, brave new world!', 0Ah, 0h          ; NOTE the null terminating byte
msg2    db      'This is how we recycle in NASM.', 0Ah, 0h  ; NOTE the null terminating byte

SECTION .text
global  _start

_start:

    mov     eax, msg1
    call    sprint

    mov     eax, msg2
    call    sprint

    call    quit
```

```
~$ nasm -f elf helloworld-inc.asm
~$ ld -m elf_i386 helloworld-inc.o -o helloworld-inc
~$ ./helloworld-inc
Hello, brave new world!
This is how we recycle in NASM.
```

Lesson 7

Linefeeds

Linefeeds are essential to console programs like our 'hello world' program. They become even more important once we start building programs that require user input. But linefeeds can be a pain to maintain. Sometimes you will want to include them in your strings and sometimes you will want to remove them. If we continue to hard code them in our variables by adding 0Ah after our declared message text, it will become a problem. If there's a place in the code that we don't want to print out the linefeed for that variable we will need to write some extra logic remove it from the string at runtime.

It would be better for the maintainability of our program if we write a subroutine that will print out our message and then print a linefeed afterwards. That way we can just call this subroutine when we need the linefeed and call our current sprint subroutine when we don't.

A call to *sys_write* requires we pass a pointer to an address in memory of the string we want to print so we can't just pass a linefeed character (0Ah) to our print function. We also don't want to create another variable just to hold a linefeed character so we will instead use the stack.

The way it works is by moving a linefeed character into EAX. We then PUSH EAX onto the stack and get the address pointed to by the Extended Stack Pointer. ESP is another register. When you PUSH items onto the stack, ESP is decremented to point to the address in memory of the last item and so it can be used to access that item directly from the stack. Since ESP points to an address in memory of a character, *sys_write* will be able to use it to print.

Note: I've highlighted the new code in functions.asm below.

functions.asm

```
;-----------------------------------------
; int slen(String message)
; String length calculation function
slen:
    push    ebx
    mov     ebx, eax

nextchar:
    cmp     byte [eax], 0
    jz      finished
    inc     eax
    jmp     nextchar

finished:
    sub     eax, ebx
    pop     ebx
    ret


;-----------------------------------------
; void sprint(String message)
; String printing function
sprint:
    push    edx
    push    ecx
    push    ebx
    push    eax
    call    slen

    mov     edx, eax
    pop     eax

    mov     ecx, eax
    mov     ebx, 1
    mov     eax, 4
    int     80h

    pop     ebx
    pop     ecx
    pop     edx
    ret


;-----------------------------------------
; void sprintLF(String message)
; String printing with line feed function
sprintLF:
    call    sprint

    push    eax         ; push eax onto the stack to preserve it while we use the eax register in th
    mov     eax, 0Ah    ; move 0Ah into eax - 0Ah is the ascii character for a linefeed
    push    eax         ; push the linefeed onto the stack so we can get the address
    mov     eax, esp    ; move the address of the current stack pointer into eax for sprint
    call    sprint      ; call our sprint function
    pop     eax         ; remove our linefeed character from the stack
    pop     eax         ; restore the original value of eax before our function was called
    ret                 ; return to our program


;-----------------------------------------
; void exit()
; Exit program and restore resources
quit:
    mov     ebx, 0
    mov     eax, 1
    int     80h
    ret
```

helloworld-lf.asm

```
; Hello World Program (Print with line feed)
; Compile with: nasm -f elf helloworld-lf.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-lf.o -o helloworld-l
; Run with: ./helloworld-lf

%include        'functions.asm'

SECTION .data
msg1    db      'Hello, brave new world!', 0h          ; NOTE we have removed the line feed characte
msg2    db      'This is how we recycle in NASM.', 0h  ; NOTE we have removed the line feed characte

SECTION .text
global  _start

_start:

    mov     eax, msg1
    call    sprintLF    ; NOTE we are calling our new print with linefeed function

    mov     eax, msg2
    call    sprintLF    ; NOTE we are calling our new print with linefeed function

    call    quit
```

```
~$ nasm -f elf helloworld-lf.asm
~$ ld -m elf_i386 helloworld-lf.o -o helloworld-lf
~$ ./helloworld-lf
Hello, brave new world!
This is how we recycle in NASM.
```

Lesson 8

Passing arguments

Passing arguments to your program from the command line is as easy as popping them off the stack in NASM. When we run our program, any passed arguments are loaded onto the stack in reverse order. The name of the program is then loaded onto the stack and lastly the total number of arguments is loaded onto the stack. The last two stack items for a NASM compiled program are always the name of the program and the number of passed arguments.

So all we have to do to use them is POP the number of arguments off the stack first, then iterate once for each argument and perform our logic. In our program that means calling our print function.

Note: We are using the ECX register as our counter for the loop. Although it's a general-purpose register it's original intention was to be used as a counter.

helloworld-args.asm

```
; Hello World Program (Passing arguments from the command line)
; Compile with: nasm -f elf helloworld-args.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-args.o -o helloworld
; Run with: ./helloworld-args

%include        'functions.asm'

SECTION .text
global  _start

_start:

    pop     ecx             ; first value on the stack is the number of arguments

nextArg:
    cmp     ecx, 0h         ; check to see if we have any arguments left
    jz      noMoreArgs      ; if zero flag is set jump to noMoreArgs label (jumping over the end of
    pop     eax             ; pop the next argument off the stack
    call    sprintLF        ; call our print with linefeed function
    dec     ecx             ; decrease ecx (number of arguments left) by 1
    jmp     nextArg         ; jump to nextArg label

noMoreArgs:
    call    quit
```

```
~$ nasm -f elf helloworld-args.asm
~$ ld -m elf_i386 helloworld-args.o -o helloworld-args
~$ ./helloworld-args "This is one argument" "This is another" 101
./helloworld-args
This is one argument
This is another
101
```

Lesson 9

User input

_Introduction to the .bss section:

So far we've used the .text and .data section so now it's time to introduce the .bss section. BSS stands for BLOCK Started by Symbol. It is an area in our program that is used to reserve space in memory for uninitialised variables. We will use it to reserve some space in memory to hold our user input since we don't know how many bytes we'll need to store.

The syntax to declare variables is as follows:

.bss section example

```
SECTION .bss
variableName1:      RESB    1       ; reserve space for 1 byte
variableName2:      RESW    1       ; reserve space for 1 word
variableName3:      RESD    1       ; reserve space for 1 double word
variableName4:      RESQ    1       ; reserve space for 1 double precision float (quad word)
variableName5:      REST    1       ; reserve space for 1 extended precision float
```

_Writing our program:

We will be using the system call *sys_read* to receive and process input from the user. This function is assigned OPCODE 3 in the Linux System Call Table. Just like *sys_write* this function also takes 3 arguments which will be loaded into EDX, ECX and EBX before requesting a software interrupt that will call the function.

The arguments passed are as follows:

- EDX will be loaded with the maximum length (in bytes) of the space in memory.
- ECX will be loaded with the address of our variable created in the .bss section.
- EBX will be loaded with the file we want to write to – in this case STDIN.

As always the datatype and meaning of the arguments passed can be found in the function's definition.

When *sys_read* detects a linefeed, control returns to the program and the users input is located at the memory address you passed in ECX.

helloworld-input.asm

```
; Hello World Program (Getting input)
; Compile with: nasm -f elf helloworld-input.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-input.o -o helloworl
; Run with: ./helloworld-input

%include        'functions.asm'

SECTION .data
msg1        db      'Please enter your name: ', 0h      ; message string asking user for input
msg2        db      'Hello, ', 0h                       ; message string to use after user has enter

SECTION .bss
sinput:     resb    255                                 ; reserve a 255 byte space in memory for the

SECTION .text
global  _start

_start:

    mov     eax, msg1
    call    sprint

    mov     edx, 255        ; number of bytes to read
    mov     ecx, sinput     ; reserved space to store our input (known as a buffer)
    mov     ebx, 0          ; write to the STDIN file
    mov     eax, 3          ; invoke SYS_READ (kernel opcode 3)
    int     80h

    mov     eax, msg2
    call    sprint

    mov     eax, sinput     ; move our buffer into eax (Note: input contains a linefeed)
    call    sprint          ; call our print function

    call    quit
```

```
~$ nasm -f elf helloworld-input.asm
~$ ld -m elf_i386 helloworld-input.o -o helloworld-input
~$ ./helloworld-input
Please enter your name: Daniel Givney
Hello, Daniel Givney
```

Lesson 10

Count to 10

_Firstly, some background:

Counting by numbers is not as straight forward as you would think in assembly. Firstly we need to pass *sys_write* an address in memory so we can't just load our register with a number and call our print function. Secondly, numbers and strings are very different things in assembly. Strings are represented by what are called ASCII values. ASCII stands for **American Standard Code for Information Interchange**. A good reference for ASCII can be found here (http://www.asciitable.com/). ASCII was created as a way to standardise the representation of strings across all computers.

Remember, we can't print a number - we have to print a string. In order to count to 10 we will need to convert our numbers from standard integers to their ASCII string representations. Have a look at the ASCII values table and notice that the string representation for the number '1' is actually '49' in ASCII. In fact, adding 48 to our numbers is all we have to do to convert them from integers to their ASCII string representations.

_Writing our program:

What we will do with our program is count from 1 to 10 using the ECX register. We will then ADD 48 to our counter to convert it from a number to it's ASCII string representation. We will then PUSH this value to the stack and call our print function passing ESP as the memory address to print from. Once we have finished counting to 10 we will exit our counting loop and call our quit function.

helloworld-10.asm

```
; Hello World Program (Count to 10)
; Compile with: nasm -f elf helloworld-10.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-10.o -o helloworld-1
; Run with: ./helloworld-10

%include        'functions.asm'

SECTION .text
global  _start

_start:

    mov     ecx, 0          ; ecx is initalised to zero.

nextNumber:
    inc     ecx             ; increment ecx

    mov     eax, ecx        ; move the address of our integer into eax
    add     eax, 48         ; add 48 to our number to convert from integer to ascii for printing
    push    eax             ; push eax to the stack
    mov     eax, esp        ; get the address of the character on the stack
    call    sprintLF        ; call our print function

    pop     eax             ; clean up the stack so we don't have unneeded bytes taking up space
    cmp     ecx, 10         ; have we reached 10 yet? compare our counter with decimal 10
    jne     nextNumber      ; jump if not equal and keep counting

    call    quit
```

```
~$ nasm -f elf helloworld-10.asm
~$ ld -m elf_i386 helloworld-10.o -o helloworld-10
```

```
~$ ./helloworld-10
1
2
3
4
5
6
7
8
9
:
```

Error: Our number 10 prints a colon (:) character instead. What's going on?

Lesson 11

Count to 10 (itoa)

So why did our program in Lesson 10 print out a colon character instead of the number 10?. Well lets have a look at our ASCII table. We can see that the colon character has a ASCII value of 58. We were adding 48 to our integers to convert them to their ASCII string representations so instead of passing *sys_write* the value '58' to print ten we actually need to pass the ASCII value for the number 1 followed by the ASCII value for the number 0. Passing *sys_write* '4948' is the correct string representation for the number '10'. So we can't just simply ADD 48 to our numbers to convert them, we first have to divide them by 10 because each place value needs to be converted individually.

We will write 2 new subroutines in this lesson 'iprint' and 'iprintLF'. These functions will be used when we want to print ASCII string representations of numbers. We achieve this by passing the number in EAX. We then initialise a counter in ECX. We will repeatedly divide the number by 10 and each time convert the remainder to a string by adding 48. We will then PUSH this onto the stack for later use. Once we can no longer divide the number by 10 we will enter our second loop. In this print loop we will print the now converted string representations from the stack and POP them off. Popping them off the stack moves ESP forward to the next item on the stack. Each time we print a value we will decrease our counter ECX. Once all numbers have been converted and printed we will return to our program.

_How does the divide instruction work?:

The DIV and IDIV instructions work by dividing whatever is in EAX by the value passed to the instruction. The quotient part of the value is left in EAX and the remainder part is put into EDX (Originally called the data register).

For example.

IDIV instruction example

```
1  mov     eax, 10         ; move 10 into eax
2  mov     esi, 10         ; move 10 into esi
3  idiv    esi             ; divide eax by esi (eax will equal 1 and edx will equal 0)
4  idiv    esi             ; divide eax by esi again (eax will equal 0 and edx will equal 1)
```

_If we are only storing the remainder won't we have problems?:

No, because these are integers, when you divide a number by an even bigger number the quotient in EAX is 0 and the remainder is the number itself. This is because the number divides zero times leaving the original value as the remainder in EDX. How good is that?

Note: Only the new functions iprint and iprintLF have comments.

functions.asm

```
;-----------------------------------------
; void iprint(Integer number)
; Integer printing function (itoa)
iprint:
    push    eax             ; preserve eax on the stack to be restored after function runs
    push    ecx             ; preserve ecx on the stack to be restored after function runs
    push    edx             ; preserve edx on the stack to be restored after function runs
    push    esi             ; preserve esi on the stack to be restored after function runs
    mov     ecx, 0          ; counter of how many bytes we need to print in the end

divideLoop:
    inc     ecx             ; count each byte to print - number of characters
    mov     edx, 0          ; empty edx
    mov     esi, 10         ; mov 10 into esi
    idiv    esi             ; divide eax by esi
    add     edx, 48         ; convert edx to it's ascii representation - edx holds the remainder af
    push    edx             ; push edx (string representation of an intger) onto the stack
    cmp     eax, 0          ; can the integer be divided anymore?
    jnz     divideLoop      ; jump if not zero to the label divideLoop

printLoop:
    dec     ecx             ; count down each byte that we put on the stack
    mov     eax, esp        ; mov the stack pointer into eax for printing
    call    sprint          ; call our string print function
    pop     eax             ; remove last character from the stack to move esp forward
    cmp     ecx, 0          ; have we printed all bytes we pushed onto the stack?
    jnz     printLoop       ; jump is not zero to the label printLoop

    pop     esi             ; restore esi from the value we pushed onto the stack at the start
    pop     edx             ; restore edx from the value we pushed onto the stack at the start
    pop     ecx             ; restore ecx from the value we pushed onto the stack at the start
    pop     eax             ; restore eax from the value we pushed onto the stack at the start
    ret


;-----------------------------------------
; void iprintLF(Integer number)
; Integer printing function with linefeed (itoa)
iprintLF:
    call    iprint          ; call our integer printing function

    push    eax             ; push eax onto the stack to preserve it while we use the eax register
    mov     eax, 0Ah        ; move 0Ah into eax - 0Ah is the ascii character for a linefeed
    push    eax             ; push the linefeed onto the stack so we can get the address
    mov     eax, esp        ; move the address of the current stack pointer into eax for sprint
    call    sprint          ; call our sprint function
    pop     eax             ; remove our linefeed character from the stack
    pop     eax             ; restore the original value of eax before our function was called
    ret


;-----------------------------------------
; int slen(String message)
; String length calculation function
slen:
    push    ebx
    mov     ebx, eax

nextchar:
    cmp     byte [eax], 0
    jz      finished
    inc     eax
    jmp     nextchar
```

```
64
65  finished:
66      sub     eax, ebx
67      pop     ebx
68      ret
69
70
71  ;------------------------------------------
72  ; void sprint(String message)
73  ; String printing function
74  sprint:
75      push    edx
76      push    ecx
77      push    ebx
78      push    eax
79      call    slen
80
81      mov     edx, eax
82      pop     eax
83
84      mov     ecx, eax
85      mov     ebx, 1
86      mov     eax, 4
87      int     80h
88
89      pop     ebx
90      pop     ecx
91      pop     edx
92      ret
93
94
95  ;------------------------------------------
96  ; void sprintLF(String message)
97  ; String printing with line feed function
98  sprintLF:
99      call    sprint
100
101     push    eax
102     mov     eax, 0AH
103     push    eax
104     mov     eax, esp
105     call    sprint
106     pop     eax
107     pop     eax
108     ret
109
110
111 ;------------------------------------------
112 ; void exit()
113 ; Exit program and restore resources
114 quit:
115     mov     ebx, 0
116     mov     eax, 1
117     int     80h
118     ret
```

helloworld-itoa.asm

```
; Hello World Program (Count to 10 itoa)
; Compile with: nasm -f elf helloworld-itoa.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 helloworld-itoa.o -o helloworld
; Run with: ./helloworld-itoa

%include        'functions.asm'

SECTION .text
global  _start

_start:

    mov     ecx, 0

nextNumber:
    inc     ecx
    mov     eax, ecx
    call    iprintLF        ; NOTE call our new integer printing function (itoa)
    cmp     ecx, 10
    jne     nextNumber

    call    quit
```

```
~$ nasm -f elf helloworld-itoa.asm
~$ ld -m elf_i386 helloworld-itoa.o -o helloworld-itoa
~$ ./helloworld-itoa
1
2
3
4
5
6
7
8
9
10
```

Lesson 12

Calculator - addition

In this program we will be adding the registers EAX and EBX together and we'll leave our answer in EAX. Firstly we use the MOV instruction to load EAX with an integer (in this case 90). We then MOV an integer into EBX (in this case 9). Now all we need to do is use the ADD instruction to perform our addition. EBX & EAX will be added together leaving our answer in the left most register in this instruction (in our case EAX). Then all we need to do is call our integer printing function to complete the program.

calculator-addition.asm

```
; Calculator (Addition)
; Compile with: nasm -f elf calculator-addition.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 calculator-addition.o -o calcul
; Run with: ./calculator-addition

%include        'functions.asm'

SECTION .text
global  _start

_start:

    mov     eax, 90     ; move our first number into eax
    mov     ebx, 9      ; move our second number into ebx
    add     eax, ebx    ; add ebx to eax
    call    iprintLF    ; call our integer print with linefeed function

    call    quit
```

```
~$ nasm -f elf calculator-addition.asm
~$ ld -m elf_i386 calculator-addition.o -o calculator-addition
~$ ./calculator-addition
99
```

Lesson 13

Calculator - subtraction

In this program we will be subtracting the value in the register EBX from the value in the register EAX. Firstly we load EAX and EBX with integers in the same way as Lesson 12. The only difference is we will be using the SUB instruction to perform our subtraction logic, leaving our answer in the left most register of this instruction (in our case EAX). Then all we need to do is call our integer printing function to complete the program.

calculator-subtraction.asm

```
; Calculator (Subtraction)
; Compile with: nasm -f elf calculator-subtraction.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 calculator-subtraction.o -o cal
; Run with: ./calculator-subtraction

%include        'functions.asm'

SECTION .text
global  _start

_start:

    mov     eax, 90     ; move our first number into eax
    mov     ebx, 9      ; move our second number into ebx
    sub     eax, ebx    ; subtract ebx from eax
    call    iprintLF    ; call our integer print with linefeed function

    call    quit
```

```
~$ nasm -f elf calculator-subtraction.asm
~$ ld -m elf_i386 calculator-subtraction.o -o calculator-subtraction
```

```
~$ ./calculator-subtraction
81
```

Lesson 14

Calculator - multiplication

In this program we will be multiplying the value in EBX by the value present in EAX. Firstly we load EAX and EBX with integers in the same way as Lesson 12. This time though we will be calling the MUL instruction to perform our multiplication logic. The MUL instruction is different from many instructions in NASM, in that it only accepts one further argument. The MUL instruction always multiples EAX by whatever value is passed after it. The answer is left in EAX.

calculator-multiplication.asm

```
; Calculator (Multiplication)
; Compile with: nasm -f elf calculator-multiplication.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 calculator-multiplication.o -o
; Run with: ./calculator-multiplication

%include        'functions.asm'

SECTION .text
global  _start

_start:

    mov     eax, 90     ; move our first number into eax
    mov     ebx, 9      ; move our second number into ebx
    mul     ebx         ; multiply eax by ebx
    call    iprintLF    ; call our integer print with linefeed function

    call    quit
```

```
~$ nasm -f elf calculator-multiplication.asm
~$ ld -m elf_i386 calculator-multiplication.o -o calculator-multiplication
~$ ./calculator-multiplication
810
```

Lesson 15

Calculator - division

In this program we will be dividing the value in EBX by the value present in EAX. We've used division before in our integer print subroutine. Our program requires a few extra strings in order to print out the correct answer but otherwise there's nothing complicated going on.

Firstly we load EAX and EBX with integers in the same way as Lesson 12. Division logic is performed using the DIV instruction. The DIV instruction always divides EAX by the value passed after it. It will leave the quotient part of the answer in EAX and put the remainder part in EDX (the original data register). We then MOV and call our strings and integers to print out the correct answer.

calculator-division.asm

```
; Calculator (Division)
; Compile with: nasm -f elf calculator-division.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 calculator-division.o -o calcul
; Run with: ./calculator-division

%include        'functions.asm'

SECTION .data
msg1        db      ' remainder '      ; a message string to correctly output result

SECTION .text
global  _start

_start:

    mov     eax, 90     ; move our first number into eax
    mov     ebx, 9      ; move our second number into ebx
    div     ebx         ; divide eax by ebx
    call    iprint      ; call our integer print function on the quotient
    mov     eax, msg1   ; move our message string into eax
    call    sprint      ; call our string print function
    mov     eax, edx    ; move our remainder into eax
    call    iprintLF    ; call our integer printing with linefeed function

    call    quit
```

```
~$ nasm -f elf calculator-division.asm
~$ ld -m elf_i386 calculator-division.o -o calculator-division
~$ ./calculator-division
10 remainder 0
```

Lesson 16

Calculator (atoi)

Our program will take several command line arguments and ADD them together printing out the result in the terminal.

_Writing our program:

Our program begins by using the POP instruction to get the number of passed arguments off the stack. This value is stored in ECX (originally known as the counter register). It will then POP the next value off the stack containing the program name and remove it from the number of arguments stored in ECX. It will then loop through the rest of the arguments popping each one off the stack and performing our addition logic. As we know, arguments passed via the command line are received by our program as strings. Before we can ADD the arguments together we will need to convert them to integers otherwise our result will not be correct. We do this by calling our Ascii to Integer function (atoi). This function will convert the ascii value into an integer and place the result in EAX. We can then ADD this value to

EDX (originally known as the data register) where we will store the result of our additions. If the value passed to atoi is not an ascii representation of an integer our function will return zero instead. When all arguments have been converted and added together we will print out the result and call our quit function.

_How does the atoi function work:

Converting an ascii string into an integer value is not a trivial task. We know how to convert an integer to an ascii string so the process should essentially work in reverse. Firstly we take the address of the string and move it into ESI (originally known as the source register). We will then move along the string byte by byte (think of each byte as being a single digit or decimal placeholder). For each digit we will check if it's value is between 48-57 (ascii values for the digits 0-9).

Once we have performed this check and determined that the byte can be converted to an integer we will perform the following logic. We will subtract 48 from the value – converting the ascii value to it's decimal equivalent. We will then ADD this value to EAX (the general purpose register that will store our result). We will then multiple EAX by 10 as each byte represents a decimal placeholder and continue the loop.

When all bytes have been converted we need to do one last thing before we return the result. The last digit of any number represents a single unit (not a multiple of 10) so we have multiplied our result one too many times. We simple divide it by 10 once to correct this and then return. If no integer arguments were pass however, we skip this divide instruction.

_What is the BL register:

You may have noticed that the atoi function references the BL register. So far in these tutorials we have been exclusively using 32bit registers. These 32bit general purpose registers contain segments of memory that can also be referenced. These segments are available in 16bits and 8bits. We wanted a single byte (8bits) because a byte is the size of memory that is required to store a single ascii character. If we used a larger memory size we would have copied 8bits of data into 32bits of space leaving us with 'rubbish' bits - because only the first 8bits would be meaningful for our calculation.

The EBX register is 32bits. EBX'S 16 bit segment is referenced as BX. BX contains the 8bit segments BL and BH (Lower and Higher bits). We wanted the first 8bits (lower bits) of EBX and so we referenced that storage area using BL.

Almost every assembly language tutorial begins with a history of the registers, their names and their sizes. These tutorials however were written to provide a foundation in NASM by first writing code and then understanding the theory. The full story about the size of registers, their history and importance are beyond the scope of this tutorial but we will return to that story in later tutorials.

Note: Only the new function in this file 'atoi' is shown below.

functions.asm

```
;-----------------------------------------
; int atoi(Integer number)
; Ascii to integer function (atoi)
atoi:
    push    ebx             ; preserve ebx on the stack to be restored after function runs
    push    ecx             ; preserve ecx on the stack to be restored after function runs
    push    edx             ; preserve edx on the stack to be restored after function runs
    push    esi             ; preserve esi on the stack to be restored after function runs
    mov     esi, eax        ; move pointer in eax into esi (our number to convert)
    mov     eax, 0          ; initialise eax with decimal value 0
    mov     ecx, 0          ; initialise ecx with decimal value 0

.multiplyLoop:
    xor     ebx, ebx        ; resets both lower and uppper bytes of ebx to be 0
    mov     bl, [esi+ecx]   ; move a single byte into ebx register's lower half
    cmp     bl, 48          ; compare ebx register's lower half value against ascii value 48 (char v
    jl      .finished       ; jump if less than to label finished
    cmp     bl, 57          ; compare ebx register's lower half value against ascii value 57 (char v
    jg      .finished       ; jump if greater than to label finished

    sub     bl, 48          ; convert ebx register's lower half to decimal representation of ascii v
    add     eax, ebx        ; add ebx to our interger value in eax
    mov     ebx, 10         ; move decimal value 10 into ebx
    mul     ebx             ; multiply eax by ebx to get place value
    inc     ecx             ; increment ecx (our counter register)
    jmp     .multiplyLoop   ; continue multiply loop

.finished:
    cmp     ecx, 0          ; compare ecx register's value against decimal 0 (our counter register)
    je      .restore        ; jump if equal to 0 (no integer arguments were passed to atoi)
    mov     ebx, 10         ; move decimal value 10 into ebx
    div     ebx             ; divide eax by value in ebx (in this case 10)

.restore:
    pop     esi             ; restore esi from the value we pushed onto the stack at the start
    pop     edx             ; restore edx from the value we pushed onto the stack at the start
    pop     ecx             ; restore ecx from the value we pushed onto the stack at the start
    pop     ebx             ; restore ebx from the value we pushed onto the stack at the start
    ret
```

calculator-atoi.asm

```
; Calculator (ATOI)
; Compile with: nasm -f elf calculator-atoi.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 calculator-atoi.o -o calculator
; Run with: ./calculator-atoi 20 1000 317

%include        'functions.asm'

SECTION .text
global  _start

_start:

    pop     ecx             ; first value on the stack is the number of arguments
    pop     edx             ; second value on the stack is the program name (discarded when we initi
    sub     ecx, 1          ; decrease ecx by 1 (number of arguments without program name)
    mov     edx, 0          ; initialise our data register to store additions

nextArg:
    cmp     ecx, 0h         ; check to see if we have any arguments left
    jz      noMoreArgs      ; if zero flag is set jump to noMoreArgs label (jumping over the end of
    pop     eax             ; pop the next argument off the stack
    call    atoi            ; convert our ascii string to decimal integer
    add     edx, eax        ; perform our addition logic
    dec     ecx             ; decrease ecx (number of arguments left) by 1
    jmp     nextArg         ; jump to nextArg label

noMoreArgs:
    mov     eax, edx        ; move our data result into eax for printing
    call    iprintLF        ; call our integer printing with linefeed function
    call    quit            ; call our quit function
```

```
~$ nasm -f elf calculator-atoi.asm
~$ ld -m elf_i386 calculator-atoi.o -o calculator-atoi
~$ ./calculator-atoi 20 1000 317
1337
```

Lesson 17

Namespace

Namespace is a necessary construct in any software project that involves a codebase that is larger than a few simple functions. Namespace provides scope to your identifiers and allows you to reuse naming conventions to make your code more readable and maintainable. In assembly language where subroutines are identified by global labels, namespace can be achieved by using local labels.

Up until the last few tutorials we have been using global labels exclusively. This means that BLOCKS of logic that essentially perform the same task needed a label with a unique identifier. A good example would be our "finished" labels. These were global in scope meaning when we needed to break out of a loop in one function we could jump to a "finished" label. But if we needed to break out of a loop in a different function we would need to name this same task something else maybe calling it "done" or "continue". Being able to reuse the label "finished" would mean that someone reading the code would know that these BLOCKS of logic perform almost the same task.

Local labels are prepended with a "." at the beginning of their name for example ".finished". You may have noticed them appearing as our code base in functions.asm grew. A local label is given the namespace of the first global label above it. You can jump to a local label by using the JMP instruction and the compiler will calculate which local label you are referencing by determining in what scope (based on the above global labels) the instruction was called.

Note: The file functions.asm (https://github.com/DGivney/assemblytutorials/blob/master/code/lesson17/functions.asm) was modified adding namespaces in all the subroutines. This is particularly important in the "slen" subroutine which contains a "finished" global label.

namespace.asm

```
1  ; Namespace
2  ; Compile with: nasm -f elf namespace.asm
3  ; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 namespace.o -o namespace
4  ; Run with: ./namespace
5
6  %include        'functions.asm'
7
8  SECTION .data
9  msg1        db      'Jumping to finished label.', 0h        ; a message string
10 msg2        db      'Inside subroutine number: ', 0h        ; a message string
11 msg3        db      'Inside subroutine "finished".', 0h     ; a message string
12
13 SECTION .text
14 global  _start
15
16 _start:
17
18 subrountineOne:
19     mov     eax, msg1       ; move the address of msg1 into eax
20     call    sprintLF        ; call our string printing with linefeed function
21     jmp     .finished       ; jump to the local label under the subrountineOne scope
22
23 .finished:
24     mov     eax, msg2       ; move the address of msg2 into eax
25     call    sprint          ; call our string printing function
26     mov     eax, 1          ; move the value one into eax (for subroutine number one)
27     call    iprintLF        ; call our integer printing function with linefeed function
28
29 subrountineTwo:
30     mov     eax, msg1       ; move the address of msg1 into eax
31     call    sprintLF        ; call our string print with linefeed function
32     jmp     .finished       ; jump to the local label under the subrountineTwo scope
33
34 .finished:
35     mov     eax, msg2       ; move the address of msg2 into eax
36     call    sprint          ; call our string printing function
37     mov     eax, 2          ; move the value two into eax (for subroutine number two)
38     call    iprintLF        ; call our integer printing function with linefeed function
39
40     mov     eax, msg1       ; move the address of msg1 into eax
41     call    sprintLF        ; call our string printing with linefeed function
42     jmp     finished        ; jump to the global label finished
43
44 finished:
45     mov     eax, msg3       ; move the address of msg3 into eax
46     call    sprintLF        ; call our string printing with linefeed function
47     call    quit            ; call our quit function
```

```
~$ nasm -f elf namespace.asm
~$ ld -m elf_i386 namespace.o -o namespace
~$ ./namespace
Jumping to finished label.
Inside subroutine number: 1
Jumping to finished label.
Inside subroutine number: 2
Jumping to finished label.
Inside subroutine "finished".
```

Lesson 18

Fizz Buzz

_Firstly, some background:

FizzBuzz is group word game played in schools to teach children division. Players take turns to count aloud integers from 1 to 100 replacing any number divisible by 3 with the word "fizz" and any number divisible by 5 with the word "buzz". Numbers that are both divisible by 3 and 5 are replaced by the word "fizzbuzz". This children's game has also become a defacto interview screening question for computer programming jobs as it's thought to easily discover candidates that can't construct a simple logic gate.

_Writing our program:

There are a number of code solutions to this simple game and some languages offer very trivial and elegant solutions. Depending on how you choose to solve it, the solution almost always involves an if statement and possibly an else statement depending whether you choose to exploit the mathematical property that anything divisible by 5 & 3 would also be divisible by 3 * 5. Being that this is an assembly language tutorial we will provide a solution that involves a structure of two cascading if statements to print the words "fizz" and/or "buzz" and an else statement in case these fail, to print the integer as an ascii value. Each iteration of our loop will then print a line feed. Once we reach 100 we call our program exit function.

fizzbuzz.asm

```
; Fizzbuzz
; Compile with: nasm -f elf fizzbuzz.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 fizzbuzz.o -o fizzbuzz
; Run with: ./fizzbuzz

%include        'functions.asm'

SECTION .data
fizz        db      'Fizz', 0h      ; a message string
buzz        db      'Buzz', 0h      ; a message string

SECTION .text
global  _start

_start:

    mov     esi, 0          ; initialise our checkFizz boolean variable
    mov     edi, 0          ; initialise our checkBuzz boolean variable
    mov     ecx, 0          ; initialise our counter variable

nextNumber:
    inc     ecx             ; increment our counter variable

.checkFizz:
    mov     edx, 0          ; clear the edx register - this will hold our remainder after division
    mov     eax, ecx        ; move the value of our counter into eax for division
    mov     ebx, 3          ; move our number to divide by into ebx (in this case the value is 3)
    div     ebx             ; divide eax by ebx
    mov     edi, edx        ; move our remainder into edi (our checkFizz boolean variable)
    cmp     edi, 0          ; compare if the remainder is zero (meaning the counter divides by 3)
    jne     .checkBuzz      ; if the remainder is not equal to zero jump to local label checkBuzz
    mov     eax, fizz       ; else move the address of our fizz string into eax for printing
    call    sprint          ; call our string printing function

.checkBuzz:
    mov     edx, 0          ; clear the edx register - this will hold our remainder after division
    mov     eax, ecx        ; move the value of our counter into eax for division
    mov     ebx, 5          ; move our number to divide by into ebx (in this case the value is 5)
    div     ebx             ; divide eax by ebx
    mov     esi, edx        ; move our remainder into edi (our checkBuzz boolean variable)
    cmp     esi, 0          ; compare if the remainder is zero (meaning the counter divides by 5)
    jne     .checkInt       ; if the remainder is not equal to zero jump to local label checkInt
    mov     eax, buzz       ; else move the address of our buzz string into eax for printing
    call    sprint          ; call our string printing function

.checkInt:
    cmp     edi, 0          ; edi contains the remainder after the division in checkFizz
    je     .continue        ; if equal (counter divides by 3) skip printing the integer
    cmp     esi, 0          ; esi contains the remainder after the division in checkBuzz
    je     .continue        ; if equal (counter divides by 5) skip printing the integer
    mov     eax, ecx        ; else move the value in ecx (our counter) into eax for printing
    call    iprint          ; call our integer printing function

.continue:
    mov     eax, 0Ah        ; move an ascii linefeed character into eax
    push    eax             ; push the address of eax onto the stack for printing
    mov     eax, esp        ; get the stack pointer (address on the stack of our linefeed char)
    call    sprint          ; call our string printing function to print a line feed
    pop     eax             ; pop the stack so we don't waste resources
    cmp     ecx, 100        ; compare if our counter is equal to 100
    jne     nextNumber      ; if not equal jump to the start of the loop

    call    quit            ; else call our quit function
```

```
~$ nasm -f elf fizzbuzz.asm
~$ ld -m elf_i386 fizzbuzz.o -o fizzbuzz
~$ ./fizzbuzz
1
2
Fizz
```

```
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
...
```

Lesson 19

Execute Command

_Firstly, some background:

The EXEC family of functions replace the currently running process with a new process, that executes the command you specified when calling it. We will be using the *sys_execve* function in this lesson to replace our program's running process with a new process that will execute the linux program /bin/echo to print out “Hello World!”.

_Naming convention:

The naming convention used for this family of functions is **exec** (execute) followed by one or more of the following letters.

- E - An array of pointers to environment variables is explicitly passed to the new process image.
- L - Command-line arguments are passed individually to the function.
- P - Uses the PATH environment variable to find the file named in the path argument to be executed.
- V - Command-line arguments are passed to the function as an array of pointers.

_Writing our program:

The V & E at the end of our function name means we will need to pass our arguments in the following format: The first argument is a string containing the command to execute, then an array of arguments to pass to that command and then another array of environment variables that the new process will use. As we are calling a simple command we won't pass any special environment variables to the new process and instead will pass 0h (null).

Both the command arguments and the environment arguments need to be passed as an array of pointers (addresses to memory). That's why we define our strings first and then define a simple null-terminated struct (array) of the variables names. This is then passed to *sys_execve.* We call the function and the process is replaced by our command and output is returned to the terminal.

execute.asm

```
; Execute
; Compile with: nasm -f elf execute.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 execute.o -o execute
; Run with: ./execute

%include        'functions.asm'

SECTION .data
command         db      '/bin/echo', 0h     ; command to execute
arg1            db      'Hello World!', 0h
arguments       dd      command
                dd      arg1                ; arguments to pass to commandline (in this case just on
                dd      0h                  ; end the struct
environment     dd      0h                  ; arguments to pass as environment variables (inthis cas

SECTION .text
global  _start

_start:

    mov     edx, environment    ; address of environment variables
    mov     ecx, arguments      ; address of the arguments to pass to the commandline
    mov     ebx, command        ; address of the file to execute
    mov     eax, 11             ; invoke SYS_EXECVE (kernel opcode 11)
    int     80h

    call    quit                ; call our quit function
```

```
~$ nasm -f elf execute.asm
~$ ld -m elf_i386 execute.o -o execute
~$ ./execute
Hello World!
```

Note: Here are a couple other commands to try.

execute.asm

```
SECTION .data
command         db      '/bin/ls', 0h       ; command to execute
arg1            db      '-l', 0h
```

execute.asm

```
SECTION .data
command         db      '/bin/sleep', 0h    ; command to execute
arg1            db      '5', 0h
```

Lesson 20

Process Forking

_Firstly, some background:

In this lesson we will use *sys_fork* to create a new process that duplicates our current process. *sys_fork* takes no arguments - you just call fork and the new process is created. Both processes run concurrently. We can test the return value (in EAX) to test whether we are currently in the parent or child process. The parent process returns a non-negative, non-zero integer. In the child process EAX is zero. This can be used to branch your logic between the parent and child.

In our program we exploit this fact to print out different messages in each process.

Note: Each process is responsible for safely exiting.

fork.asm

```
; Fork
; Compile with: nasm -f elf fork.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 fork.o -o fork
; Run with: ./fork

%include        'functions.asm'

SECTION .data
childMsg        db      'This is the child process', 0h     ; a message string
parentMsg       db      'This is the parent process', 0h    ; a message string

SECTION .text
global  _start

_start:

    mov     eax, 2              ; invoke SYS_FORK (kernel opcode 2)
    int     80h

    cmp     eax, 0              ; if eax is zero we are in the child process
    jz      child               ; jump if eax is zero to child label

parent:
    mov     eax, parentMsg      ; inside our parent process move parentMsg into eax
    call    sprintLF            ; call our string printing with linefeed function

    call    quit                ; quit the parent process

child:
    mov     eax, childMsg       ; inside our child process move childMsg into eax
    call    sprintLF            ; call our string printing with linefeed function

    call    quit                ; quit the child process
```

```
~$ nasm -f elf fork.asm
~$ ld -m elf_i386 fork.o -o fork
~$ ./fork
This is the parent process
This is the child process
```

Lesson 21

Telling the time

Generating a unix timestamp in NASM is easy with the *sys_time* function of the linux kernel. Simply pass OPCODE 13 to the kernel with no arguments and you are returned the Unix Epoch (https://en.wikipedia.org/wiki/Unix_epoch) in the EAX register.

That is the number of seconds that have elapsed since January 1st 1970 UTC.

time.asm

```
; Time
; Compile with: nasm -f elf time.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 time.o -o time
; Run with: ./time

%include        'functions.asm'

SECTION .data
msg         db      'Seconds since Jan 01 1970: ', 0h     ; a message string

SECTION .text
global  _start

_start:

    mov     eax, msg        ; move our message string into eax for printing
    call    sprint          ; call our string printing function

    mov     eax, 13         ; invoke SYS_TIME (kernel opcode 13)
    int     80h             ; call the kernel

    call    iprintLF        ; call our integer printing function with linefeed
    call    quit            ; call our quit function
```

```
~$ nasm -f elf time.asm
~$ ld -m elf_i386 time.o -o time
~$ ./time
Seconds since Jan 01 1970: 1374995660
```

Lesson 22

File Handling - Create

_Firstly, some background:

File Handling in Linux is achieved through a small number of system calls related to creating, updating and deleting files. These functions require a file descriptor (https://en.wikipedia.org/wiki/File_descriptor) which is a unique, non-negative integer that identifies the file on the system.

_Writing our program:

We begin the tutorial by creating a file using *sys_creat.* We will then build upon our program in each of the following file handling lessons, adding code as we go. Eventually we will have a full program that can create, update, open, close and delete files.

sys_creat expects 2 arguments - the file permissions in ECX and the filename in EBX. The *sys_creat* opcode is then loaded into EAX and the kernel is called to create the file. The file descriptor of the created file is returned in EAX. This file descriptor can then be used for all other file handling functions.

create.asm

```
; Create
; Compile with: nasm -f elf create.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 create.o -o create
; Run with: ./create

%include    'functions.asm'

SECTION .data
filename db 'readme.txt', 0h    ; the filename to create

SECTION .text
global  _start

_start:

    mov     ecx, 0777o          ; set all permissions to read, write, execute
    mov     ebx, filename       ; filename we will create
    mov     eax, 8              ; invoke SYS_CREAT (kernel opcode 8)
    int     80h                 ; call the kernel

    call    quit                ; call our quit function
```

```
~$ nasm -f elf create.asm
~$ ld -m elf_i386 create.o -o create
~$ ./create
```

Note: The file 'readme.txt' will now have been created in the folder.

Lesson 23

File Handling - Write

Building upon the previous lesson we will now use *sys_write* to write content to a newly created file.

sys_write expects 3 arguments - the number of bytes to write in EDX, the contents string to write in ECX and the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) in EBX. The *sys_write* opcode is then loaded into EAX and the kernel is called to write the content to the file. In this lesson we will first call *sys_creat* to get a file descriptor which we will then load into EBX.

write.asm

```
; Write
; Compile with: nasm -f elf write.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 write.o -o write
; Run with: ./write

%include    'functions.asm'

SECTION .data
filename db 'readme.txt', 0h    ; the filename to create
contents db 'Hello world!', 0h  ; the contents to write

SECTION .text
global  _start

_start:

    mov     ecx, 0777o          ; code continues from lesson 22
    mov     ebx, filename
    mov     eax, 8
    int     80h

    mov     edx, 12             ; number of bytes to write - one for each letter of our contents str
    mov     ecx, contents       ; move the memory address of our contents string into ecx
    mov     ebx, eax            ; move the file descriptor of the file we created into ebx
    mov     eax, 4              ; invoke SYS_WRITE (kernel opcode 4)
    int     80h                 ; call the kernel

    call    quit                ; call our quit function
```

```
~$ nasm -f elf write.asm
~$ ld -m elf_i386 write.o -o write
~$ ./write
```

Note: Open the newly created file 'readme.txt' in this folder and you will see the content 'Hello world!'.

Lesson 24

File Handling - Open

Building upon the previous lesson we will now use *sys_open* to obtain the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) of the newly created file. This file descriptor can then be used for all other file handling functions.

sys_open expects 2 arguments - the access mode (table below) in ECX and the filename in EBX. The *sys_open* opcode is then loaded into EAX and the kernel is called to open the file and return the file descriptor.

sys_open additionally accepts zero or more file creation flags and file status flags in EDX. Click here for more information about the access mode, file creation flags and file status flags (http://man7.org/linux/man-pages/man2/open.2.html).

	Description	Value
O_RDONLY	open file in read only mode	0

	Description	Value
O_WRONLY	open file in write only mode	1
O_RDWR	open file in read and write mode	2

Note: *sys_open* returns the file descriptor in EAX. On linux this will be a unique, non-negative integer which we will print using our integer printing function.

open.asm

```
; Open
; Compile with: nasm -f elf open.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 open.o -o open
; Run with: ./open

%include    'functions.asm'

SECTION .data
filename db 'readme.txt', 0h    ; the filename to create
contents db 'Hello world!', 0h  ; the contents to write

SECTION .text
global  _start

_start:

    mov     ecx, 0777o          ; Create file from lesson 22
    mov     ebx, filename
    mov     eax, 8
    int     80h

    mov     edx, 12             ; Write contents to file from lesson 23
    mov     ecx, contents
    mov     ebx, eax
    mov     eax, 4
    int     80h

    mov     ecx, 0              ; flag for readonly access mode (O_RDONLY)
    mov     ebx, filename       ; filename we created above
    mov     eax, 5              ; invoke SYS_OPEN (kernel opcode 5)
    int     80h                 ; call the kernel

    call    iprintLF            ; call our integer printing function
    call    quit                ; call our quit function
```

```
~$ nasm -f elf open.asm
~$ ld -m elf_i386 open.o -o open
~$ ./open
4
```

Lesson 25

File Handling - Read

Building upon the previous lesson we will now use *sys_read* to read the content of a newly created and opened file. We will store this string in a variable.

sys_read expects 3 arguments - the number of bytes to read in EDX, the memory address of our variable in ECX and the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) in EBX. We will use the previous lessons *sys_open* code to obtain the file descriptor which we will then load into EBX. The *sys_read* opcode is then loaded into EAX and the kernel is called to read the file contents into our variable and is then printed to the screen.

Note: We will reserve 255 bytes in the .bss section to store the contents of the file. See Lesson 9 for more information on the .bss section.

read.asm

```
; Read
; Compile with: nasm -f elf read.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 read.o -o read
; Run with: ./read

%include    'functions.asm'

SECTION .data
filename db 'readme.txt', 0h    ; the filename to create
contents db 'Hello world!', 0h  ; the contents to write

SECTION .bss
fileContents resb 255,          ; variable to store file contents

SECTION .text
global  _start

_start:

    mov     ecx, 0777o          ; Create file from lesson 22
    mov     ebx, filename
    mov     eax, 8
    int     80h

    mov     edx, 12             ; Write contents to file from lesson 23
    mov     ecx, contents
    mov     ebx, eax
    mov     eax, 4
    int     80h

    mov     ecx, 0              ; Open file from lesson 24
    mov     ebx, filename
    mov     eax, 5
    int     80h

    mov     edx, 12             ; number of bytes to read - one for each letter of the file contents
    mov     ecx, fileContents   ; move the memory address of our file contents variable into ecx
    mov     ebx, eax            ; move the opened file descriptor into EBX
    mov     eax, 3              ; invoke SYS_READ (kernel opcode 3)
    int     80h                 ; call the kernel

    mov     eax, fileContents   ; move the memory address of our file contents variable into eax for
    call    sprintLF            ; call our string printing function

    call    quit                ; call our quit function
```

```
~$ nasm -f elf read.asm
~$ ld -m elf_i386 read.o -o read
~$ ./read
Hello world!
```

Lesson 26

File Handling - Close

Building upon the previous lesson we will now use *sys_close* to properly close an open file.

sys_close expects 1 argument - the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) in EBX. We will use the previous lessons code to obtain the file descriptor which we will then load into EBX. The *sys_close* opcode is then loaded into EAX and the kernel is called to close the file and remove the active file descriptor.

close.asm

```
; Close
; Compile with: nasm -f elf close.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 close.o -o close
; Run with: ./close

%include    'functions.asm'

SECTION .data
filename db 'readme.txt', 0h    ; the filename to create
contents db 'Hello world!', 0h  ; the contents to write

SECTION .bss
fileContents resb 255,          ; variable to store file contents

SECTION .text
global  _start

_start:

    mov     ecx, 0777o          ; Create file from lesson 22
    mov     ebx, filename
    mov     eax, 8
    int     80h

    mov     edx, 12             ; Write contents to file from lesson 23
    mov     ecx, contents
    mov     ebx, eax
    mov     eax, 4
    int     80h

    mov     ecx, 0              ; Open file from lesson 24
    mov     ebx, filename
    mov     eax, 5
    int     80h

    mov     edx, 12             ; Read file from lesson 25
    mov     ecx, fileContents
    mov     ebx, eax
    mov     eax, 3
    int     80h

    mov     eax, fileContents
    call    sprintLF

    mov     ebx, ebx            ; not needed but used to demonstrate that SYS_CLOSE takes a file des
    mov     eax, 6              ; invoke SYS_CLOSE (kernel opcode 6)
    int     80h                 ; call the kernel

    call    quit                ; call our quit function
```

```
~$ nasm -f elf close.asm
~$ ld -m elf_i386 close.o -o close
~$ ./close
Hello world!
```

Note: We have properly closed the file and removed the active file descriptor.

Lesson 27

File Handling - Seek

In this lesson we will open a file and update the file contents at the end of the file using *sys_lseek.*

Using *sys_lseek* you can move the cursor within the file by an offset in bytes. The below example will move the cursor to the end of the file, then pass 0 bytes as the offset (so we append to the end of the file and not beyond) before writing a string in that position. Try different values in ECX and EDX to write the content to different positions within the opened file.

sys_lseek expects 3 arguments - the whence argument (table below) in EDX, the offset in bytes in ECX, and the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) in EBX. The *sys_lseek* opcode is then loaded into EAX and we call the kernel to move the file pointer to the correct offset. We then use *sys_write* to update the content at that position.

	Description	Value
SEEK_SET	beginning of the file	0
SEEK_CUR	current file offset	1
SEEK_END	end of the file	2

Note: A file 'readme.txt' has been included in the code folder for this lesson. This file will be updated after running the program.

seek.asm

```
; Seek
; Compile with: nasm -f elf seek.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 seek.o -o seek
; Run with: ./seek

%include    'functions.asm'

SECTION .data
filename db 'readme.txt', 0h    ; the filename to create
contents  db '-updated-', 0h     ; the contents to write at the start of the file

SECTION .text
global  _start

_start:

    mov     ecx, 1              ; flag for writeonly access mode (O_WRONLY)
    mov     ebx, filename       ; filename of the file to open
    mov     eax, 5              ; invoke SYS_OPEN (kernel opcode 5)
    int     80h                 ; call the kernel

    mov     edx, 2              ; whence argument (SEEK_END)
    mov     ecx, 0              ; move the cursor 0 bytes
    mov     ebx, eax            ; move the opened file descriptor into EBX
    mov     eax, 19             ; invoke SYS_LSEEK (kernel opcode 19)
    int     80h                 ; call the kernel

    mov     edx, 9              ; number of bytes to write - one for each letter of our contents str
    mov     ecx, contents       ; move the memory address of our contents string into ecx
    mov     ebx, ebx            ; move the opened file descriptor into EBX (not required as EBX alre
    mov     eax, 4              ; invoke SYS_WRITE (kernel opcode 4)
    int     80h                 ; call the kernel

    call    quit                ; call our quit function
```

```
~$ nasm -f elf seek.asm
~$ ld -m elf_i386 seek.o -o seek
~$ ./seek
```

Lesson 28

File Handling - Delete

Deleting a file on linux is achieved by calling *sys_unlink.*

sys_unlink expects 1 argument - the filename in EBX. The *sys_unlink* opcode is then loaded into EAX and the kernel is called to delete the file.

Note: A file 'readme.txt' has been included in the code folder for this lesson. This file will be deleted after running the program.

unlink.asm

```
; Unlink
; Compile with: nasm -f elf unlink.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 unlink.o -o unlink
; Run with: ./unlink

%include    'functions.asm'

SECTION .data
filename db 'readme.txt', 0h    ; the filename to delete

SECTION .text
global  _start

_start:

    mov     ebx, filename       ; filename we will delete
    mov     eax, 10             ; invoke SYS_UNLINK (kernel opcode 10)
    int     80h                 ; call the kernel

    call    quit                ; call our quit function
```

```
~$ nasm -f elf unlink.asm
~$ ld -m elf_i386 unlink.o -o unlink
~$ ./unlink
```

Lesson 29

Sockets - Create

_Firstly, some background:

Socket Programming in Linux is achieved through the use of the *sys_socketcall* kernel function. The *sys_socketcall* function is somewhat unique in that it encapsulates a number of different subroutines, all related to socket operations, within the one function. By passing different integer values in EBX we can change the behaviour of this function to create, listen, send, receive, close and more. Click here (https://gist.github.com/DGivney/7196bd7a9f21a12c9397bdcf9ae040d2) to view the full commented source code of the completed program.

_Writing our program:

We begin the tutorial by first initalizing some of our registers which we will use later to store important values. We will then create a socket using *sys_socketcall's* first subroutine which is called 'socket'. We will then build upon our program in each of the following socket programming lessons, adding code as we go. Eventually we will have a full program that can create, bind, listen, accept, read, write and close sockets.

sys_socketcall's subroutine 'socket' expects 2 arguments - a pointer to an array of arguments in ECX and the integer value 1 in EBX. The *sys_socketcall* opcode is then loaded into EAX and the kernel is called to create the socket. Because everything in linux is a file, we recieve back the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) of the created socket in EAX. This file descriptor can then be used for performing other socket programming functions.

Note: XORing a register by itself is an efficent way of ensuring the register is initalised with the integer value zero and doesn't contain an unexpected value that could corrupt your program.

socket.asm

```
; Socket
; Compile with: nasm -f elf socket.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 socket.o -o socket
; Run with: ./socket

%include    'functions.asm'

SECTION .text
global  _start

_start:

    xor     eax, eax            ; init eax 0
    xor     ebx, ebx            ; init ebx 0
    xor     edi, edi            ; init edi 0
    xor     esi, esi            ; init esi 0

_socket:

    push    byte 6              ; push 6 onto the stack (IPPROTO_TCP)
    push    byte 1              ; push 1 onto the stack (SOCK_STREAM)
    push    byte 2              ; push 2 onto the stack (PF_INET)
    mov     ecx, esp            ; move address of arguments into ecx
    mov     ebx, 1              ; invoke subroutine SOCKET (1)
    mov     eax, 102            ; invoke SYS_SOCKETCALL (kernel opcode 102)
    int     80h                 ; call the kernel

    call    iprintLF            ; call our integer printing function (print the file descriptor in E

_exit:

    call    quit                ; call our quit function
```

```
~$ nasm -f elf socket.asm
~$ ld -m elf_i386 socket.o -o socket
~$ ./socket
3
```

Lesson 30

Sockets - Bind

Building on the previous lesson we will now associate the created socket with a local IP address and port which will allow us to connect to it. We do this by calling the second subroutine of *sys_socketcall* which is called 'bind'.

We begin by storing the file descriptor we recieved in lesson 29 into EDI. EDI was originally called the Destination Index and is traditionally used in copy routines to store the location of a target file.

sys_socketcall's subroutine 'bind' expects 2 arguments - a pointer to an array of arguments in ECX and the integer value 2 in EBX. The *sys_socketcall* opcode is then loaded into EAX and the kernel is called to bind the socket.

socket.asm

```
; Socket
; Compile with: nasm -f elf socket.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 socket.o -o socket
; Run with: ./socket

%include    'functions.asm'

SECTION .text
global  _start

_start:

    xor     eax, eax            ; initialize some registers
    xor     ebx, ebx
    xor     edi, edi
    xor     esi, esi

_socket:

    push    byte 6              ; create socket from lesson 29
    push    byte 1
    push    byte 2
    mov     ecx, esp
    mov     ebx, 1
    mov     eax, 102
    int     80h

_bind:

    mov     edi, eax            ; move return value of SYS_SOCKETCALL into edi (file descriptor for
    push    dword 0x00000000    ; push 0 dec onto the stack IP ADDRESS (0.0.0.0)
    push    word 0x2923         ; push 9001 dec onto stack PORT (reverse byte order)
    push    word 2              ; push 2 dec onto stack AF_INET
    mov     ecx, esp            ; move address of stack pointer into ecx
    push    byte 16             ; push 16 dec onto stack (arguments length)
    push    ecx                 ; push the address of arguments onto stack
    push    edi                 ; push the file descriptor onto stack
    mov     ecx, esp            ; move address of arguments into ecx
    mov     ebx, 2              ; invoke subroutine BIND (2)
    mov     eax, 102            ; invoke SYS_SOCKETCALL (kernel opcode 102)
    int     80h                 ; call the kernel

_exit:

    call    quit                ; call our quit function
```

```
~$ nasm -f elf socket.asm
~$ ld -m elf_i386 socket.o -o socket
~$ ./socket
```

Lesson 31

Sockets - Listen

In the previous lessons we created a socket and used the 'bind' subroutine to associate it with a local IP address and port. In this lesson we will use the 'listen' subroutine of *sys_socketcall* to tell our socket to listen for incoming TCP requests. This will allow us to read and write to anyone who connects to our socket.

sys_socketcall's subroutine 'listen' expects 2 arguments - a pointer to an array of arguments in ECX and the integer value 4 in EBX. The *sys_socketcall* opcode is then loaded into EAX and the kernel is called. If succesful the socket will begin listening for incoming requests.

socket.asm

```
; Socket
; Compile with: nasm -f elf socket.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 socket.o -o socket
; Run with: ./socket

%include    'functions.asm'

SECTION .text
global  _start

_start:

    xor     eax, eax            ; initialize some registers
    xor     ebx, ebx
    xor     edi, edi
    xor     esi, esi

_socket:

    push    byte 6              ; create socket from lesson 29
    push    byte 1
    push    byte 2
    mov     ecx, esp
    mov     ebx, 1
    mov     eax, 102
    int     80h

_bind:

    mov     edi, eax            ; bind socket from lesson 30
    push    dword 0x00000000
    push    word 0x2923
    push    word 2
    mov     ecx, esp
    push    byte 16
    push    ecx
    push    edi
    mov     ecx, esp
    mov     ebx, 2
    mov     eax, 102
    int     80h

_listen:

    push    byte 1              ; move 1 onto stack (max queue length argument)
    push    edi                 ; push the file descriptor onto stack
    mov     ecx, esp            ; move address of arguments into ecx
    mov     ebx, 4              ; invoke subroutine LISTEN (4)
    mov     eax, 102            ; invoke SYS_SOCKETCALL (kernel opcode 102)
    int     80h                 ; call the kernel

_exit:

    call    quit                ; call our quit function
```

```
~$ nasm -f elf socket.asm
~$ ld -m elf_i386 socket.o -o socket
~$ ./socket
```

Lesson 32

Sockets - Accept

In the previous lessons we created a socket and used the 'bind' subroutine to associate it with a local IP address and port. We then used the 'listen' subroutine of *sys_socketcall* to tell our socket to listen for incoming TCP requests. Now we will use the 'accept' subroutine of *sys_socketcall* to tell our socket to accept those incoming requests. Our socket will then be ready to read and write to remote connections.

sys_socketcall's subroutine 'accept' expects 2 arguments - a pointer to an array of arguments in ECX and the integer value 4 in EBX. The *sys_socketcall* opcode is then loaded into EAX and the kernel is called. The 'accept' subroutine will create another file descriptor, this time identifying the incoming socket connection. We will use this file descriptor to read and write to the incoming connection in later lessons.

Note: Run the program and use the command `sudo netstat -plnt` in another terminal to view the socket listening on port 9001.

socket.asm

```
; Socket
; Compile with: nasm -f elf socket.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 socket.o -o socket
; Run with: ./socket

%include    'functions.asm'

SECTION .text
global  _start

_start:

    xor     eax, eax            ; initialize some registers
    xor     ebx, ebx
    xor     edi, edi
    xor     esi, esi

_socket:

    push    byte 6              ; create socket from lesson 29
    push    byte 1
    push    byte 2
    mov     ecx, esp
    mov     ebx, 1
    mov     eax, 102
    int     80h

_bind:

    mov     edi, eax            ; bind socket from lesson 30
    push    dword 0x00000000
    push    word 0x2923
    push    word 2
    mov     ecx, esp
    push    byte 16
    push    ecx
    push    edi
    mov     ecx, esp
    mov     ebx, 2
    mov     eax, 102
    int     80h

_listen:

    push    byte 1              ; listen socket from lesson 31
    push    edi
    mov     ecx, esp
    mov     ebx, 4
    mov     eax, 102
    int     80h

_accept:

    push    byte 0              ; push 0 dec onto stack (address length argument)
    push    byte 0              ; push 0 dec onto stack (address argument)
    push    edi                 ; push the file descriptor onto stack
    mov     ecx, esp            ; move address of arguments into ecx
    mov     ebx, 5              ; invoke subroutine ACCEPT (5)
    mov     eax, 102            ; invoke SYS_SOCKETCALL (kernel opcode 102)
    int     80h                 ; call the kernel

_exit:

    call    quit                ; call our quit function
```

```
~$ nasm -f elf socket.asm
~$ ld -m elf_i386 socket.o -o socket
~$ ./socket
```

Lesson 33

Sockets - Read

When an incoming connection is accepted by our socket, a new file descriptor identifying the incoming socket connection is returned in EAX. In this lesson we will use this file descriptor to read the incoming request headers from the connection.

We begin by storing the file descriptor we recieved in lesson 32 into ESI. ESI was originally called the Source Index and is traditionally used in copy routines to store the location of a target file.

We will use the kernel function *sys_read* to read from the incoming socket connection. As we have done in previous lessons, we will create a variable to store the contents being read from the file descriptor. Our socket will be using the HTTP protocol to communicate. Parsing HTTP request headers to determine the length of the incoming message and accepted response formats is beyond the scope of this tutorial. We will instead just read up to the first 255 bytes and print that to standardout.

Once the incoming connection has been accepted, it is very common for webservers to spawn a child process to manage the read/write communication. The parent process is then free to return to the listening/accept state and accept any new incoming requests in parallel. We will implement this design pattern below using *sys_fork* and the JMP instruction prior to reading the request headers in the child process.

To generate valid request headers we will use the commandline tool `curl` to connect to our listening socket. But you can also use a standard web browser to connect in the same way.

sys_read expects 3 arguments - the number of bytes to read in EDX, the memory address of our variable in ECX and the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) in EBX. The *sys_read* opcode is then loaded into EAX and the kernel is called to read the contents into our variable which is then printed to the screen.

Note: We will reserve 255 bytes in the .bss section to store the contents being read from the file descriptor. See Lesson 9 for more information on the .bss section.

Note: Run the program and use the command `curl http://localhost:9001` in another terminal to view the request headers being read by our program.

socket.asm

```
1  ; Socket
2  ; Compile with: nasm -f elf socket.asm
3  ; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 socket.o -o socket
4  ; Run with: ./socket
5
6  %include    'functions.asm'
7
8  SECTION .bss
9  buffer resb 255,                ; variable to store request headers
10
11 SECTION .text
12 global  _start
13
14 _start:
15
16     xor     eax, eax            ; initialize some registers
17     xor     ebx, ebx
18     xor     edi, edi
19     xor     esi, esi
20
21 _socket:
```

```
22
23      push    byte 6              ; create socket from lesson 29
24      push    byte 1
25      push    byte 2
26      mov     ecx, esp
27      mov     ebx, 1
28      mov     eax, 102
29      int     80h
30
31  _bind:
32
33      mov     edi, eax            ; bind socket from lesson 30
34      push    dword 0x00000000
35      push    word 0x2923
36      push    word 2
37      mov     ecx, esp
38      push    byte 16
39      push    ecx
40      push    edi
41      mov     ecx, esp
42      mov     ebx, 2
43      mov     eax, 102
44      int     80h
45
46  _listen:
47
48      push    byte 1              ; listen socket from lesson 31
49      push    edi
50      mov     ecx, esp
51      mov     ebx, 4
52      mov     eax, 102
53      int     80h
54
55  _accept:
56
57      push    byte 0              ; accept socket from lesson 32
58      push    byte 0
59      push    edi
60      mov     ecx, esp
61      mov     ebx, 5
62      mov     eax, 102
63      int     80h
64
65  _fork:
66
67      mov     esi, eax            ; move return value of SYS_SOCKETCALL into esi (file descriptor for
68      mov     eax, 2              ; invoke SYS_FORK (kernel opcode 2)
69      int     80h                 ; call the kernel
70
71      cmp     eax, 0              ; if return value of SYS_FORK in eax is zero we are in the child pro
72      jz      _read               ; jmp in child process to _read
73
74      jmp     _accept             ; jmp in parent process to _accept
75
76  _read:
77
78      mov     edx, 255            ; number of bytes to read (we will only read the first 255 bytes for
79      mov     ecx, buffer         ; move the memory address of our buffer variable into ecx
80      mov     ebx, esi            ; move esi into ebx (accepted socket file descriptor)
81      mov     eax, 3              ; invoke SYS_READ (kernel opcode 3)
82      int     80h                 ; call the kernel
83
84      mov     eax, buffer         ; move the memory address of our buffer variable into eax for printi
85      call    sprintLF            ; call our string printing function
86
87  _exit:
88
89      call    quit                ; call our quit function
```

```
~$ nasm -f elf socket.asm
~$ ld -m elf_i386 socket.o -o socket
```

```
~$ ./socket
GET / HTTP/1.1
Host: localhost:9001
User-Agent: curl/x.xx.x
Accept: */*
```

Lesson 34

Sockets - Write

When an incoming connection is accepted by our socket, a new file descriptor identifying the incoming socket connection is returned in EAX. In this lesson we will use this file descriptor to send our response to the connection.

We will use the kernel function *sys_write* to write to the incoming socket connection. As our socket will be communicating using the HTTP protocol, we will need to send some compulsory headers in order to allow HTTP speaking clients to connect. We will send these following the formatting rules set out in the RFC Standard (https://tools.ietf.org/html/rfc2616?spm=5176.doc32013.2.3.Aimyd7#section-4.2).

sys_write expects 3 arguments - the number of bytes to write in EDX, the response string to write in ECX and the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) in EBX. The *sys_write* opcode is then loaded into EAX and the kernel is called to send our response back through our socket to the incoming connection.

Note: We will create a variable in the .data section to store the response we will write to the file descriptor. See Lesson 1 for more information on the .data section.

Note: Run the program and use the command `curl http://localhost:9001` in another terminal to view the response sent via our socket. Or connect to the same address using any standard web browser.

socket.asm

```
1  ; Socket
2  ; Compile with: nasm -f elf socket.asm
3  ; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 socket.o -o socket
4  ; Run with: ./socket
5
6  %include    'functions.asm'
7
8  SECTION .data
9  ; our response string
10 response db 'HTTP/1.1 200 OK', 0Dh, 0Ah, 'Content-Type: text/html', 0Dh, 0Ah, 'Content-Length: 14',
11
12 SECTION .bss
13 buffer resb 255,                ; variable to store request headers
14
15 SECTION .text
16 global  _start
17
18 _start:
19
20     xor     eax, eax            ; initialize some registers
21     xor     ebx, ebx
22     xor     edi, edi
23     xor     esi, esi
24
25 _socket:
26
27     push    byte 6              ; create socket from lesson 29
28     push    byte 1
```

```
    push    byte 2
    mov     ecx, esp
    mov     ebx, 1
    mov     eax, 102
    int     80h

_bind:

    mov     edi, eax            ; bind socket from lesson 30
    push    dword 0x00000000
    push    word 0x2923
    push    word 2
    mov     ecx, esp
    push    byte 16
    push    ecx
    push    edi
    mov     ecx, esp
    mov     ebx, 2
    mov     eax, 102
    int     80h

_listen:

    push    byte 1              ; listen socket from lesson 31
    push    edi
    mov     ecx, esp
    mov     ebx, 4
    mov     eax, 102
    int     80h

_accept:

    push    byte 0              ; accept socket from lesson 32
    push    byte 0
    push    edi
    mov     ecx, esp
    mov     ebx, 5
    mov     eax, 102
    int     80h

_fork:

    mov     esi, eax            ; fork socket from lesson 33
    mov     eax, 2
    int     80h

    cmp     eax, 0
    jz      _read

    jmp     _accept

_read:

    mov     edx, 255            ; read socket from lesson 33
    mov     ecx, buffer
    mov     ebx, esi
    mov     eax, 3
    int     80h

    mov     eax, buffer
    call    sprintLF

_write:

    mov     edx, 78             ; move 78 dec into edx (length in bytes to write)
    mov     ecx, response       ; move address of our response variable into ecx
    mov     ebx, esi            ; move file descriptor into ebx (accepted socket id)
    mov     eax, 4              ; invoke SYS_WRITE (kernel opcode 4)
    int     80h                 ; call the kernel

_exit:

    call    quit                ; call our quit function
```

```
~$ nasm -f elf socket.asm
~$ ld -m elf_i386 socket.o -o socket
~$ ./socket
```

New terminal window

```
~$ curl http://localhost:9001
Hello World!
```

Lesson 35

Sockets - Close

In this lesson we will use *sys_close* to properly close the active socket connection in the child process after our response has been sent. This will free up some resources that can be used to accept new incoming connections.

sys_close expects 1 argument - the file descriptor (https://en.wikipedia.org/wiki/File_descriptor) in EBX. The *sys_close* opcode is then loaded into EAX and the kernel is called to close the socket and remove the active file descriptor.

Note: Run the program and use the command `curl http://localhost:9001` in another terminal or connect to the same address using any standard web browser.

socket.asm

```
; Socket
; Compile with: nasm -f elf socket.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 socket.o -o socket
; Run with: ./socket

%include    'functions.asm'

SECTION .data
; our response string
response db 'HTTP/1.1 200 OK', 0Dh, 0Ah, 'Content-Type: text/html', 0Dh, 0Ah, 'Content-Length: 14',

SECTION .bss
buffer resb 255,                ; variable to store request headers

SECTION .text
global  _start

_start:

    xor     eax, eax            ; initialize some registers
    xor     ebx, ebx
    xor     edi, edi
    xor     esi, esi

_socket:

    push    byte 6              ; create socket from lesson 29
    push    byte 1
    push    byte 2
    mov     ecx, esp
    mov     ebx, 1
    mov     eax, 102
    int     80h

```

```
35  _bind:
36
37      mov     edi, eax            ; bind socket from lesson 30
38      push    dword 0x00000000
39      push    word 0x2923
40      push    word 2
41      mov     ecx, esp
42      push    byte 16
43      push    ecx
44      push    edi
45      mov     ecx, esp
46      mov     ebx, 2
47      mov     eax, 102
48      int     80h
49
50  _listen:
51
52      push    byte 1              ; listen socket from lesson 31
53      push    edi
54      mov     ecx, esp
55      mov     ebx, 4
56      mov     eax, 102
57      int     80h
58
59  _accept:
60
61      push    byte 0              ; accept socket from lesson 32
62      push    byte 0
63      push    edi
64      mov     ecx, esp
65      mov     ebx, 5
66      mov     eax, 102
67      int     80h
68
69  _fork:
70
71      mov     esi, eax            ; fork socket from lesson 33
72      mov     eax, 2
73      int     80h
74
75      cmp     eax, 0
76      jz      _read
77
78      jmp     _accept
79
80  _read:
81
82      mov     edx, 255            ; read socket from lesson 33
83      mov     ecx, buffer
84      mov     ebx, esi
85      mov     eax, 3
86      int     80h
87
88      mov     eax, buffer
89      call    sprintLF
90
91  _write:
92
93      mov     edx, 78             ; write socket from lesson 34
94      mov     ecx, response
95      mov     ebx, esi
96      mov     eax, 4
97      int     80h
98
99  _close:
100
101     mov     ebx, esi            ; move esi into ebx (accepted socket file descriptor)
102     mov     eax, 6              ; invoke SYS_CLOSE (kernel opcode 6)
103     int     80h                 ; call the kernel
104
105 _exit:
106
107     call    quit                ; call our quit function
```

```
~$ nasm -f elf socket.asm
~$ ld -m elf_i386 socket.o -o socket
~$ ./socket
```

New terminal window

```
~$ curl http://localhost:9001
Hello World!
```

Note: We have properly closed the socket connections and removed their active file descriptors.

Lesson 36

Download a Webpage

In the previous lessons we have been learning how to use the many subroutines of the *sys_socketcall* kernel function to create, manage and transfer data through Linux sockets. We will continue that theme in this lesson by using the 'connect' subroutine of *sys_socketcall* to connect to a remote webserver and download a webpage.

These are the steps we need to follow to connect a socket to a remote server:

- Call *sys_socketcall's* subroutine 'socket' to create an active socket that we will use to send outbound requests.
- Call *sys_socketcall's* subroutine 'connect' to connect our socket with a socket on the remote webserver.
- Use *sys_write* to send a HTTP formatted request through our socket to the remote webserver.
- Use *sys_read* to recieve the HTTP formatted response from the webserver.

We will then use our string printing function to print the response to our terminal.

_What is a HTTP Request:

The HTTP specification has evolved through a number of standard versions including 1.0 in RFC1945 (https://tools.ietf.org/html/rfc1945), 1.1 in RFC2068 (https://tools.ietf.org/html/rfc2068) and 2.0 in RFC7540 (https://tools.ietf.org/html/rfc7540). Version 1.1 is still the most common today.

A HTTP/1.1 request is comprised of 3 sections:

1. A line containing the *request method*, *request url*, and *http version*
2. An optional section of *request headers (https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers)*
3. An *empty line* that tells the remote server you have finished sending the request and you will begin waiting for the response.

A typical HTTP request for the root document on this server would look like this:

```
1  GET / HTTP/1.1                ; A line containing the request method, url and version
2  Host: asmtutor.com            ; A section of request headers
3                                ; A required empty line
```

_Writing our program:

This tutorial starts out like the previous ones by calling *sys_socketcall's* subroutine 'socket' to initially create our socket. However, instead of calling 'bind' on this socket we will call 'connect' with an IP Address and Port Number to connect our socket to a remote webserver. We will then use the *sys_write* and *sys_read* kernel methods to transfer data between the two sockets by sending a HTTP request and reading the HTTP response.

sys_socketcall's subroutine 'connect' expects 2 arguments - a pointer to an array of arguments in ECX and the integer value 3 in EBX. The *sys_socketcall* opcode is then loaded into EAX and the kernel is called to connect to the socket.

Note: In Linux we can use the following command `./crawler > index.html` to save the output of our program to a file instead.

crawler.asm

```
; Crawler
; Compile with: nasm -f elf crawler.asm
; Link with (64 bit systems require elf_i386 option): ld -m elf_i386 crawler.o -o crawler
; Run with: ./crawler

%include    'functions.asm'

SECTION .data
; our request string
request db 'GET / HTTP/1.1', 0Dh, 0Ah, 'Host: 139.162.39.66:80', 0Dh, 0Ah, 0Dh, 0Ah, 0h

SECTION .bss
buffer resb 1,                  ; variable to store response

SECTION .text
global  _start

_start:

    xor     eax, eax            ; init eax 0
    xor     ebx, ebx            ; init ebx 0
    xor     edi, edi            ; init edi 0

_socket:

    push    byte 6              ; push 6 onto the stack (IPPROTO_TCP)
    push    byte 1              ; push 1 onto the stack (SOCK_STREAM)
    push    byte 2              ; push 2 onto the stack (PF_INET)
    mov     ecx, esp            ; move address of arguments into ecx
    mov     ebx, 1              ; invoke subroutine SOCKET (1)
    mov     eax, 102            ; invoke SYS_SOCKETCALL (kernel opcode 102)
    int     80h                 ; call the kernel

_connect:

    mov     edi, eax            ; move return value of SYS_SOCKETCALL into edi (file descriptor for
    push    dword 0x4227a28b    ; push 139.162.39.66 onto the stack IP ADDRESS (reverse byte order)
    push    word 0x5000         ; push 80 onto stack PORT (reverse byte order)
    push    word 2              ; push 2 dec onto stack AF_INET
    mov     ecx, esp            ; move address of stack pointer into ecx
    push    byte 16             ; push 16 dec onto stack (arguments length)
    push    ecx                 ; push the address of arguments onto stack
    push    edi                 ; push the file descriptor onto stack
    mov     ecx, esp            ; move address of arguments into ecx
    mov     ebx, 3              ; invoke subroutine CONNECT (3)
    mov     eax, 102            ; invoke SYS_SOCKETCALL (kernel opcode 102)
    int     80h                 ; call the kernel

_write:

    mov     edx, 43             ; move 43 dec into edx (length in bytes to write)
    mov     ecx, request        ; move address of our request variable into ecx
    mov     ebx, edi            ; move file descriptor into ebx (created socket file descriptor)
    mov     eax, 4              ; invoke SYS_WRITE (kernel opcode 4)
    int     80h                 ; call the kernel

```

```
_read:

    mov     edx, 1              ; number of bytes to read (we will read 1 byte at a time)
    mov     ecx, buffer         ; move the memory address of our buffer variable into ecx
    mov     ebx, edi            ; move edi into ebx (created socket file descriptor)
    mov     eax, 3              ; invoke SYS_READ (kernel opcode 3)
    int     80h                 ; call the kernel

    cmp     eax, 0              ; if return value of SYS_READ in eax is zero, we have reached the en
    jz      _close              ; jmp to _close if we have reached the end of the file (zero flag se

    mov     eax, buffer         ; move the memory address of our buffer variable into eax for printi
    call    sprint              ; call our string printing function
    jmp     _read               ; jmp to _read

_close:

    mov     ebx, edi            ; move edi into ebx (connected socket file descriptor)
    mov     eax, 6              ; invoke SYS_CLOSE (kernel opcode 6)
    int     80h                 ; call the kernel

_exit:

    call    quit                ; call our quit function
```

```
~$ nasm -f elf crawler.asm
~$ ld -m elf_i386 crawler.o -o crawler
~$ ./crawler
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html lang="en">
...
</html>
```
