ASM slides

Please look at the asm document that goes along side this

Table of contents (based on the google slides)

Before you start

You can go to the fasm website and find example code and the manuals at https://flatassembler.net/docs.php

For an in depth explanation of a certain section you can go to the manual in **Programmer's manual** section. If you want an even more advanced manual you can look into the **Understanding the flat assembler** section.

You can learn more windows functions in the **Windows Programming** section.

I am comfortable using fasm in windows however I am not comfortable using it in linux. There is a lot of overlap between the two operating systems but a main difference is how the heap is used and how syscalls/winapi is used.

Before you start

When looking at fasm you can look at example projects at https://flatassembler.net/examples.php which has projects for both windows and linux. You can copy the code to easily get the syntax for something like the ‘idata’ section or do something else with it.

In your download of fasm you can go to the examples folder for additional examples.

Finally there is a macro engine for fasm called fasm g. It is useful to know about it and you can start learning about it in the **Introduction and Overview, User Manual and the Design principles** section of the manual.

How you write assembly is based on an instruction set architecture or ISA. ISAs are the connection of the underlying hardware to us, the assembly user. Within individual ISAs there are often different assemblers. What this means is that one computer hardware set there are basically multiple “assembly programming languages”. These assemblers take assembly and turn it into machine code.

Machine code looks like the writing on the left and middle of the image. Assembly is on the right.

Types of ISA

The different types of ISAs are basically the different types of assembly paradigms

CISC (Complex) vs RISC (Reduced)

X86 (CISC)

ARM (RISC)

MIPS (RISC)

VLIW (RISC)

PowerPc (RISC)

X86 assemblers

FASM (Flat assembly. The one we will be using)

NASM (Netwide assembly)

TASM (Turbo assembly)

MASM (Microsoft assembly)

GAS (Gnu Assembly)

AT&T vs Intel syntax

FASM uses intel syntax. There are other assemblers which are AT&T syntax such GAS (Gnu assembler). (Explain the difference between AT&T and intel syntax.)

For our purposes you don’t need to know about at&t syntax.

Refer to accompanying document for extra information regarding at&t syntax and its difference to intel syntax.

Format

The first line in a fasm file is always the format. This specifies how the executable is formatted and is usually OS specific. If you are using DOS (Disk Operating System) you use MZ. If you use windows uses PE (portable executable). ELF executables are used in Linux. COFF used to be used by Linux but is still used by executables are still used in some windows scenarios and embedded systems.

In windows you can use sub formats such as GUI/console for windows apps and native for drivers.

For more information on formats go to the Programmer’s Manual section 2.4.

X86 registers

Al - lowest 8 bits of ax

Ah - highest 8 bits of ax

Ax - 16 bits of rax

Eax - 32 bits

Rax - 64 bits

Other examples are eax, ebx, ecx, edx, rsi, rdi, and r8 through r15.

To see all of the registers go to the fasm Programmer’s Manual section 2.1.19

Sections

Sections are regions in memory that have a specific purpose. The data section and bss sections are for variables and there are others that you should also know.

For more information on which sections can be used in which operating system go to Programmer’s Manual section 2.4.

ASM slides

Essential assembly sections in fasm (These are sections in an executable)

Data section

```
section '.data' data readable writeable
```

Where you write variables

Idata section

```
section '.idata' import data readable writeable
```

Code or text section

```
section '.code' data readable executable
```

Some Nonessential assembly sections

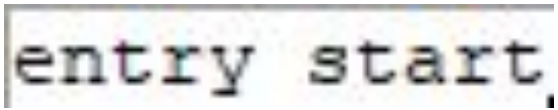
Relocation section/.rloc (only in windows)

Bss section (uninitialized data) (Not necessary in fasm)

Entry

This is usually a label such as entry <label name>. It is good practice to always have an entry point so that you don’t get any unexpected execution.

Entry is windows exclusive.



Simple ASM instructions

Mov

Lea

Cmp

Jmp

Making variables.

Variables are made in the the .data section. In the data section they are initialized by writing the code in the image.

X and Y are initialized as 32. This is

Similar to doing int x = 32; int y = 32;

```
section '.data' data readable writeable
        x db 32
        y db 32
```

Notice the data section is readable and writable. It is impossible to make a section not readable however it is possible to not make the section writeable.

Define/Reserve table

Db - define byte

Dw - define word

Dd - define dword

Dq - define qword

Rb - reserve byte

Rw - reserve word

Rd - reserve dword

Rq - reserve qword

Making variables ext.

There are many ways to make variables however you should know that the .bss section is essentially pointless in fasm.

You can also reserve data with rb <value>. The value is the number of bits to be reserved. It is essentially the same as db <value>.

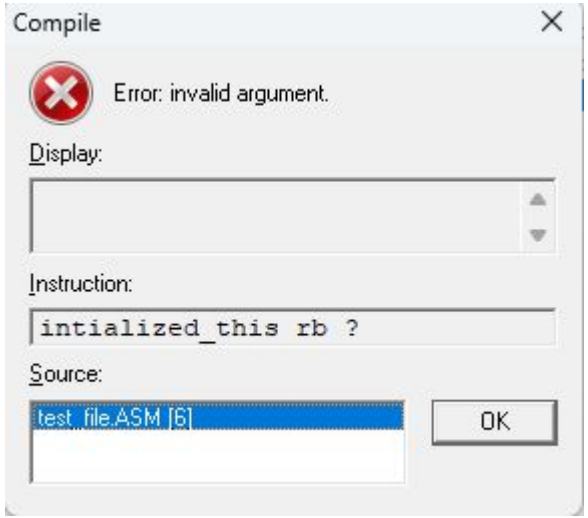
You can also assign <variable_name> db ? and it will evaluate to 0. You cannot assign a rb to a ?. It will error.

```
variable_name dd ? ; This evaluates to zero.

variable_name rb 32 ; 32 bits are saved as zero.

intialized_this rb ? ; This causes an error.
```

Invalid argument error message

This correlates to the error on the previous slide. Invalid argument means that you assigned the variable to a value that assembler will not accept. Under the source tab you can see test_file.ASM [6]. 6 is the line number and test_file.ASM is the name of the file that was compiled.

```
intialized_this rb ? ; This causes an error.
```



Assigning variables with mov

In fasm syntax you use mov <dest>, <src>. What that means is that you copy the contents of the rightmost parameter(<src>) to the leftmost parameter(<dest>).

You can move the memory location (also called a pointer) of a value into register with mov rdx, value. Where value is the name of your variable.

You can move the value at a memory location into register with mov rdx, [value]. If value was equal to 32 then rdx would be 32.

```
mov [variable_name], 100 ;Assigning the variable variable_name 100

mov edx, variable_name ; This sets edx to the memory value of variable_name

mov variable_name, edx ; edx is currently zero. This gives an error.

mov 100, edx; This errors out.
```

Jump

You switch the place where code is read from and align it to a label. Labels are written as their name and then colon Ex. <label_name>: Keep them on their own line. This sets the instruction pointer to that instruction. These can be used to make loops and other similar things. Jumps can be conditional but they also can be regular. The jmp instruction always jumps to start.

```
start: ; This is a label.
       mov rax, 0
       mov rbx, 0
       mov rcx, 0
       jmp example_label ; This will always jump to example label.
       mov rdx, 0 ; This code is skipped.
example_label:
       ; This code begins executing.
       mov rsi, 0
```

Cmp

Cmp is similar to an if statement. The syntax for a cmp is <arg1>, <arg 2>. You compare both of the left and right sided and these set the eflags. eflags are boolean values that can inform certain actions. A cmp works by simulating subtracting arg1 - arg2 in order to set the flags (the arguments stay the same as when they were from before they were compared). A common use case is to use a cmp statement to set the eflags and then use a conditional jump statement. That means that the jmp has a j at the start and a condition at the end. Ex. je, jc, ja, jb, etc.

```
start:
        cmp rax, variable1 ; These are all valid compares
        cmp rax, [number] ; Right now nothing is done with them
        mov rcx, [number]
        cmp rax, rcx
```

Eflags(32 bit)/ Rflags(64 bit)

There are two ways to use eflags. One is to put them after a conditional. The other is to use them after an arithmetic operation. Using eflags with conditionals is dependent on whether you are comparing signed (either positive or negative) or unsigned (guaranteed to be positive) arguments.

There are many types of eflags but status flags are the ones used most commonly.

There is also a flag used for setting the direction of strings.

Flags can be used together such as having a jump evaluated based on the if two numbers are equal based on the carry and zero flags.

Eflags/conditional jumps

When comparing two numbers you can check if the two numbers are less or greater with jl and je when arg1 is less or greater than arg2 when cmp <arg1>, <arg2>.

```
        mov rax, 2
        mov rbx, 1
        cmp rax, rbx ; 2 is greater 1.
        jg label_1 ; This does execute.
label_2:
        mov rax, 2 ; This is not executed.
label_1:
        mov rax, 1 ; instruction after the jump is this one.
```

For more information you can look at section 7.3.8.2 of the intel ia-32 manual vol1 or https://www.tutorialspoint.com/assembly_programming/assembly_conditions.htm

eflags/conditional jumps ext.

```
        mov rax, 2
        mov rbx, 1
        cmp rax, rbx ; 2 is not less than 1.
        jl label_1 ; This doesn't execute.
label_2:
        mov rax, 2 ; instruction executed after the jump.
label_1:
        mov rax, 1 ; This isn't executed after the jump.
```

When you want to compare two arguments to see if they are equal you use je.

```
        mov rax, 1
        mov rbx, 1
        cmp rax, rbx ; 1 is equal to 1.
        je label_1 ; This does execute.
label_2:
        mov rax, 2 ; This is not executed.
label_1:
        mov rax, 1 ; instruction after the jump is this one.
```

eflags/ conditional jumps ext.

```
        mov al, 128 ;
        cmp bl, al ; 0 (bl) - 128 (al) = -128 causes an overflow
        jo label_1 ; This
label_2:
        mov rbx, 1
label_1 : ; This executes
        mov rax, 1
```

The above goes over jump overflow. For more information go to the document associated with these slides.

```
        mov rax, 0
        or rax, rax ; This evaluates that rax is zero.
        jz label_1 ; The jump is taken.
label_2:
        mov rbx, 1
label_1 : ; This executes
        mov rax, 1
```

Jump zero (jz) and jump equal (je) are the same (go to the document to learn how).

Notable tips (Loops)

Labels can be used to make loops (both for and while loops). You do this by making a condition that breaks out of the loop and then add loop contents.

```
start: ; This acts the same way as a loop does.
       ; In c code you would write this as
       ; while(i != 32){
       ;         i++;
       ; }
         mov eax, [x]
         cmp eax, 32
         je finish
         add [x], 1
         jmp start
finish:
         mov al, 15
```

Lea

Lea or load effective address is not used very often. Lea is legacy and has specific use cases that you can learn from a professional. Moving into a register without the brackets has the same utility as lea.

For more information see Programmer’s Manual section 2.1.11.

Bitwise/logical operations

Add

Sub

Mul

Div

Xor

And

Nor

And the list goes on…

Adding, subtracting and multiplying

Adding is done with add <dest>, <src>. Subtracting is the same however multiplication is different.

```
add eax, 1
sub eax, 1
```

Mul is used for unsigned multiplication (positive number multiplication). Mul is done with mul <place that is multiplied>. Based on the size of the place that is multiplied it is multiplied by the same size version of rax.

```
mul bx ; This is multiplied by ax. It is basically ax * bx and put in bx.
mul ecx ; This is multiplied by eax. It is basically ecx * eax and put in ecx.
mul rdx; This is multiplied by rax. It is basically rdx * rax and put in rdx.
```

imul

Imul is the signed version of mul (negative/positive multiplication). If you treat imul as mul (with only having one thing besides the imul) then it is the same mul.

```
imul bx ; This is multiplied by ax. It is basically ax * bx and put in bx.
imul ecx ; This is multiplied by eax. It is basically ecx * eax and put in ecx.
imul rdx; This is multiplied by rax. It is basically rdx * rax and put in rdx.
```

Otherwise you can use two elements after the imul.

```
imul bx, cx ; Same as bx = bx * cx.
imul cx, 3 ; Same as cx = cx * 3.
imul ecx, [x] ; Same as ecx = ecx * [x]
imul [x], ecx ; This causes an error._
```

You can also use three elements after the imul.

```
imul ax,bx,10   ; register by immediate value to register
imul ax,[si],10 ; memory by immediate value to register
```

For more information go to Programmer's Manual section 2.1.3.

Division

Dividing with unsigned value uses div while signed values use idiv. The syntaxes are div <value> or idiv <value> depending on which one you wish to use. The two options (you can change the registers and variables) for div/idiv are:

```
div rbx ; dividend / divisor = quotient. rbx is the divisor.
div qword [mem_value] ; You don't have to make it a qword
idiv rbx
idiv qword [mem_value]; You don't have to make it a qword
```

When dividing the <value> in div/idiv <value> are the divisor. The dividend is ax if the divisor is a byte or the appropriate size of eax (double the memory size of the divisor e.g. if the divisor is a word then the dividend is a dword) acting as the lower half and the appropriate size of edx as the upper half. The quotient is then stored in the appropriate part of eax and the remainder is stored in the appropriate size of edx.

Division ext.

When working with numbers it is very important to use registers that are not too big for the numbers you are using. For example if you want to divide by 2 (the divisor would be 2) you should use a byte register.

```
mov cl, 2 ; This is the divisor
mov ax, 12 ; This is the dividend.
div cl ; dividend/divisor = quotient._al is now 6 and cl is still 2.
```

The max size of the divisor is a dword. When this happens the top portion of edx is added to eax. When there is no remainder edx is set to 0 and eax is set to the answer (in this case 6 because 360000/60000).

```
mov ecx, 60000 ; This is the divisor in decimal(60000)
mov rax, 0111111001000000b ; This is binary.
mov rdx, 101b ; You have to add rax and rdx. They make 360000
; 360000 is 19 bit number. The bottom 16 are supplied by rax and the top 3 are supplied by rdx
div ecx ; dividend/divisor = quotient. rax is now 6.
```

For more information go to Programmer's Manual section 2.1.4.

Logical operations

Logical operations are operations that are like putting your input through a logic gate. Note that every value is taken as binary when using a logical operation.

Not is a very basic example. If you have a 1 bit input it will be reversed so 0 will be 1 and 1 will be 0.

And is a two input gate. What that means is that you give two inputs such as 0 and 0 and then have your input based on them. If you have the inputs 0 and 0 then your value would be 0. However if you have your option as 1 and 1 then the output will be 1.

For more information go to Programmer’s Manual section 2.1.5.

Logical operations ext.

You look at a truth table to tell what is going to be output when you give an input. The truth table for a not operation is below. The value on the left is the input and the value on the right is the output.

0 -> 1 **Look in the document for the truth tables.**

1 -> 0

When looking at logical operations you look at every individual bit of a binary value as its own logical operation and then combine them together.

A not of 1111 would turn into 0000.

Strings

Rsi, the 64 bit version, and esi, the 32 bit version is the source string register. You mov a pointer into rsi/esi and then use lodsb or one of its permutations, which stores it in al (or any of its appropriate permutations). Do note that lodsb loads a byte value into al and increments the value of the pointer so that you can use it again immediately. lodsw, lodsd, and lodsdq also exist and load word, dword and qword value into ax, eax and rax respectively.

```
; The reason esi is used is because a is a dword and
; esi is recieving the value of a.
mov esi, [a] ; This causes an "error" when running.
mov esi, a ; This works the same as esi besides the obvious differences.
mov rsi, a ; rsi recieves the memory location of value a
lodsb ; It loops itself to stop a buffer overflow.
; It does this because there is only one value. If a was pointing to a character
; array with a size greater than 1 lodsb would run effectively.
```

Strings continued

Rdi, the 64 bit version, and edi, the 32 bit version is the destination string register. You mov the pointer into rdi the same way you you mov the rsi/esi. You can use stosb to place the value of al into rdi and then increment the value of rdi so that you can put in another value without overriding it. The stosb works similar to lodsb.

```
mov rsi, stri
mov rdi, variable_name ; If you don't set rdi to a value
; EXCEPTION_ACESS_VIOLATION occurs at runtime.
lodsb ; loads the first byte (in this case s) into al.
stosb ; loads the value of al into the variable at a.
```

Stack

When you are in 64 bit mode you have to push qwords. You push register or regular numbers and can pop them later. The stack is meant to store a copy of values and acts as a memory unit.

```
push rax ; This pushes the value to the stack. The value of rax remains the same.
push rdx
pop rdx ; You pop off the stack in reverse order to the way you pushed onto the stack.
pop rax
push 0
pop rax ; The value of rax is 0
push stri
pop rax ; The pop value releases the latest value of the stack onto whatever is requested
push a ; a is a dword. This runs fine.
push [a] ; This gives an error.
push rdx
pop [a] ; This causes an error
push rdx
pop 0 ; This causes an error
push edx ; This causes an error.
```

Calls

Calls are the assembly version of functions. The syntax for a call is call <call_name> where call_name is a label that you want to jump to. A call is run until you manually request for the call to end with the ret keyword. When a call is running it is common practice to push all registers that are used during the call onto the stack so that the values from before the call was made are saved.

```
        mov rax, 0
        call func
        mov rax, 1
func: ; The call label
      push rax ; saves rax
      mov rax, 2 ; modifies rax in the call.
      pop rax ; restores the value of rax after the call is done.
      ret ; The thing that stops the call.
```

Floats

The assembler uses a floating point register stack for handling floating point numbers (numbers with decimals). It starts at st0 with the top of the stack and goes to st7 at the bottom. These registers can be sized 32, 64, or 80 bits.

You add a variable to the stack with fld <variable>. Whenever you add a variable after your first initialization the first variable moves to st1 and the new variable moves to st0. The variables loop up to the beginning when you add more than 8 variables. Meaning st7 moves to st0.

```
fld [x]
```

You can transfer variables from the top of the stack(st0) to specific stack variables with fst <variable>.

```
fst st3
```

Floats ext.

You add two float registers together with fadd <dest>, <src>.

```
fadd st1, st2 ; This is how you add registers together.
```

You do the same for subtracting, multiplying and dividing.

```
fsub st2, st0
fmul st2, st0
fdiv st2, st0
```

For imul with floats you use and it has to be multiplied with st0.

```
fimul [x]
```

For more information look at section 2.1.3 of the fasm Programmer's Manual.

Winapi

Windows uses an api instead of syscalls. You can set the stack before hand or you can use invoke and list off your parameters. You have to include the library that holds whichever winapi you are calling. You also have to set the import table.

Please note that the winapi is notoriously difficult to use and was most of the reason this tutorial was made.

```
format PE64 ; Portable Executable(PE) 64 bit
entry start
include 'win64a.inc'
section '.idata' import data readable writeable
; Notice that import data is after '.idata'
; instead of jsut data
library kernel32, 'kernel32', \
        user32, 'user32' _
```

Winapi continued

.inc files are similar to .asm files. .asm files are your traditional assembly file and are written as the tutorial has suggested. .inc files however are written as library files (according to the way they are used in fasm) and therefore are expected to be added with .asm files. The library command should be used by default. kernel32 is an example of a library being referenced and ‘kernel32’ is the reference to the library file.

It is also called a dll and ‘kernel32’

can be replaced with ‘kernel32.dll’.

```
format PE64 ; Portable Executable(PE) 64 bit
entry start
include 'win64a.inc'
section '.idata' import data readable writeable
; Notice that import data is after '.idata'
; instead of jsut data
library kernel32, 'kernel32', \
        user32, 'user32' _
```

Winapi ext.

The other possible library arguments are kernel32, user32, gdi32, comctl32, comdlg32, and shell32. After library arguments are made then you have to add the linking table. You can add every single possible linking argument with include ‘api/<whatever file correlates to the library you chose>’ or you could import linking arguments individually with the import that align with your library and then the individual arguments.

```
import kernel32,\
       ExitProcess,'ExitProcess'
```

Reminder of linking table/linking - Linking is the process in which a compiler/assembler brings in other files. A basic definition of a link table is a list of functions that the compiler/assembler links.

Winapi ext.

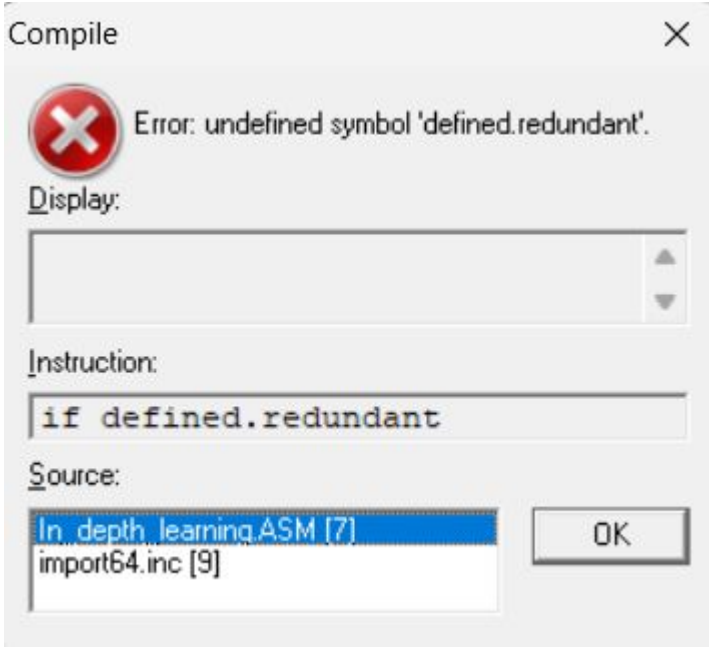
Note that in the .idata section commas are used to separate text. It is expected that if you put a comma that text will follow it.

Notice the instruction says if defined.redundant.

This is fasm code. You may ask how because

We required as cmp statement before.

```
format PE64 ; Portable Executable(PE) 64 bit
entry start
include 'win64a.inc'
section '.idata' import data readable writeable
; Notice that import data is after '.idata'
; instead of jsut data
library kernel32, 'kernel32', \
        user32, 'user32', ; This causes an error.
```



Winapi ext.

The fasm macro system is used in order to make the process of writing assembly faster. The error seen before referenced an

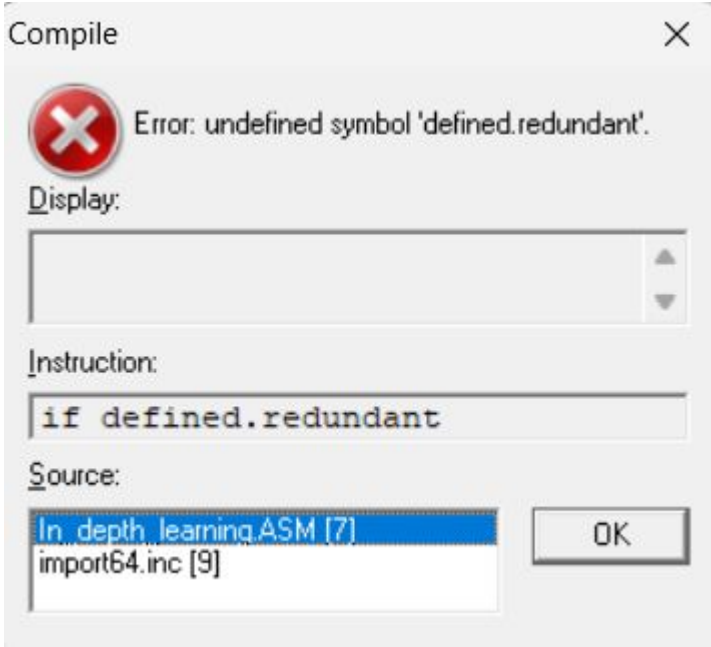
if macro.

Also notice the source tab. Import64.inc is a file

That is the file that is run when fasm is compiling

And the error occurred. You can click on import64

And see the source code in the fasm ide.



Notable tips

When learning how to use the winapi you can look up the utility of the function at the microsoft website or by looking up <argument name> winapi. Ex. MessageBox winapi. You can also look in the examples folder in the fasm folder.

Windows heap

You call GetProcessHeap to get the default process heap which is made with every process. All winapi calls have their return value assigned to the rax register. You should check the return value with an or statement. or rax, rax is the value used most commonly as winapi usually sets zero as the error code. When using GetProcessHeap requires kernel32.dll to be imported.

```
invoke GetProcessHeap ; This is the winapi for getting the default process heap.
or rax, rax ; This ensures rax doesn't change.
jz error ; In this case rax being zero means that there is an error.
mov [hStdHeap], rax ; rax gives the handle (memory location) of the heap.
```

Windows creating heap

You can allocate parts of the heap specifically for a purpose with HeapAlloc. You can do a similar thing with HeapRealloc to make the size change.

```
invoke HeapAlloc, [hStdHeap], HEAP_ZERO_MEMORY, 100
; This allocates 100 bytes which are all initially zero.
or rax, rax
jz error
mov [hSpecificHeap], rax
```

You can allocate an entirely new heap with HeapCreate. HeapCreate has a lot more distinct uses.

```
invoke HeapCreate, HEAP_GENERATE_EXCEPTIONS, 100, 100
; This creates an entirely new heap that is 100 bytes and
; cannot grow past 100 bytes.
or rax, rax
jz error
mov [hCreatedHeap], rax
```

Windows heap ext.

You can use DestoryHeap to delete a created heap.

```
invoke HeapDestroy, [hCreatedHeap] ; Destroys the created heap.
or rax, rax
jz error
```

In order to add something to the heap you do basically the same thing as you would for a string.

```
mov rsi, str_value ; A value to put in the heap.
mov rdi, [hStdHeap] ; The memory location of the heap.
lodsb ; Take a byte from str_value and store it in al
stosb ; Store al at a memory location of the heap. _
```

Segment Registers

There are 6 types of segment registers in x86. They are 16 bit and build on the fact of operating systems that the size of their memory is bigger than the size that can be represented by 64 bit registers.

This is mainly due to operating system limitations and can be explained further by looking into the virtualization part of OSTEP in the OS part of the extra learning document.

For more information go to 1.2.1 in the Programmer’s Manual.

Suggested additions

Syscalls are used in linux. You set the stack before hand and then call the syscall you want.

Vector registers are ways of adding multiple things at once. They are complicated and useful.

I don’t have enough experience with linux fasm or x86 vector registers. If you do and want to contribute to this section then look at the individual projects readme for adding to existing projects.