

# Introduction to Assembly Programming

```
instructor      db "Aaron Bray", 0x0a, 0x00
email   db "aaron.m.bray@gmail.com", 0x0a, 0x00
```

## Course Roadmap

```
content:
    jmp .introduction

.introduction:
    mov rax, HowToAsm
    jmp .basic_operations

.basic_operations:
    call .arithmetic
    call .bit_operations
    mov rcx, DataTypes
    jmp .control_flow

.control_flow:
    jmp .hardware_essentials

.hardware_essentials:
    mov rax, Memory
    mov rcx, Interrupts
    call FloatingPoint
    call Simd
    call .systems_programming

.systems_programming:
    ret
```

## Assembly: An Introduction

### Objectives

- Understand the relationship between assembly language and opcodes
- Understand byte ordering, as it pertains to Assembly Programming
- Identify x86(\_64) General Purpose Registers
- Perform basic memory access operations
- Begin debugging with the GNU Source-Level Debugger (GDB)
- Understand basic data sizes and types with regard to x86(\_64)

## Understanding Assembly

- What is Assembly?
  - Provides "instructions" (human-friendly) that map to "opcodes" (processor-friendly)
  - Typically very hardware-specific
- Why use assembly?
  - Performance
  - Utilize otherwise unexposed hardware features
  - Some operations can't easily be expressed in higher level languages (such as C)

Performance is now much less often a reason to use assembly than it was in the past (as compilers have steadily improved, and more features exposed via intrinsics).

# Assembly Instructions

- Typically consist of an instruction of some kind, and some operands
- Operands can consist of several things, to include:
  - Registers
  - Memory Addresses
  - Immediate (literal) Values
- Other data types and some prefixes (which modify what the instruction does) also exist

## Opcodes

- One or more bytes that the processor decodes (and executes)
- Typically direct translations from assembly language instructions
- x86 and x86\_64 instructions are variable length
- Syntax is (slightly) complicated

# Assembly Instructions

This set of instructions:

```
mov eax, 0x01  
ret
```

Becomes...

Thus, this set of instructions becomes the follow set of opcodes.

## Opcodes

...This set of opcodes

```
0xb8 0x01 0x00 0x00 0x00  
0xc3
```

Perform opcodes demo

# Assemblers and Syntax

- A number of different options exist for assemblers
  - GAS - the GNU Assembler
  - nasm/yasm - The Netwide Assembler/Yet another Assembler (a rewrite of NASM)
  - masm - the Microsoft assembler
- Most have special quirks and slight differences in how syntax is handled (though they are similar)
- This course will focus on NASM, which uses Intel syntax

## Syntax Differences - Some Examples

- Intel Syntax: Used by NASM/YASM and others

```
mov eax, 0x01
```

- AT&T Syntax: Used by GAS and others

```
movl $0x01, %eax
```

- Other flavors also exist

This is one very blatant example, but many other assemblers (MASM, etc) will have small quirks to how you type in your assembly instructions. All equate to (about) the same thing, however.

## Byte Ordering

- Determines the order bytes appear in memory
- Big Endian stores the most significant (or biggest) value first
  - the memory address: 0x10203040 would appear as: 0x10 0x20 0x30 0x40
- Little Endian puts the least significant (or little) value first
  - the memory address: 0x10203040 would appear as: 0x40 0x30 0x20 0x10

## Byte Ordering

- x86(\_64) is little Endian
- Again, least significant byte (not bit) appears first

In memory, this address:

```
0xdeadbeef
```

## Byte Ordering

Becomes:

```
0xefbeadde
```

## Byte Ordering

Initial: 0xde0xad0xbe0xef

Memory: 0xef0xbe0xad0xde

## Memory: The 10,000 Foot View

- Various Memory Components take differing amounts of time to access
- Most higher level languages (such as C) abstract this away, the developer is not really exposed to it
- Assembly gives you a bit more control (though some things are still hidden on most modern platforms)

## The Memory Hierarchy

From Fastest Access to Slowest:

- Registers
- Cache (L1/L2/L3)
- System Memory
- Disk

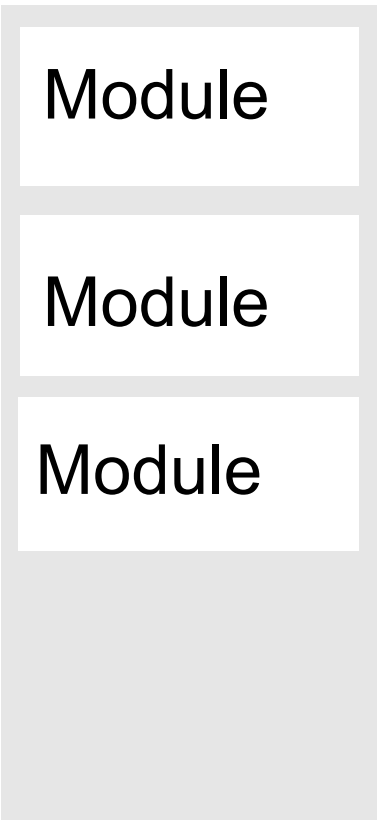
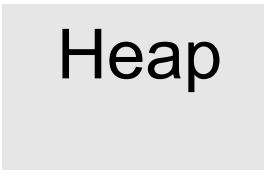
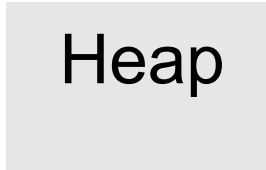
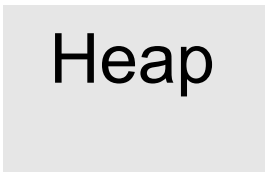
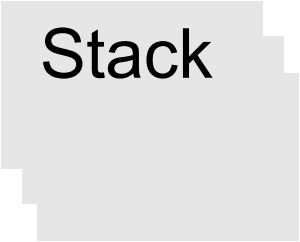
# Virtual Memory

- Hardware allows abstraction of memory addressing
- Most addressing deals with virtual addresses, which are translated (via lookup table) to physical addresses
- More than one "view" of a physical memory segment can exist (in different processes)
- Each user mode process appears to have a full range of addressable memory and resources
- Most modern OSes support paging, allowing us to pretend we have a much greater amount of physical memory than actually exists

We will discuss virtual memory and memory mappings in general later on, when we discuss specific hardware features

## Process Memory Layout

A Very High level view:



# Process Memory Layout

- Stack Segments typically grown from high to low memory addresses
- Modules in the previous diagram indicate executable files loaded into the process space; some examples include:
  - glibc (more specifically, the .so containing the libc code)
  - kernel32.dll
  - Currently running executable
- Heap sections and Anonymous Mappings
- Kernel memory
- Other Items

## Registers

- Assembly programming gives us total control over access to these
- Special hardware structures on the processor
- Some are general purpose (e.g., can store any type of data)
- Others are specialized, and may contain status codes, flags, etc., or be associated with specific hardware
- Limited in number

## General Purpose Registers

- Shared registers have addressable subregisters
- 64 bit/32 bit/16 bit/8 bit
- x86\_64 contains many more general purpose registers than x86 (though they don't all have subregisters)

## x86 and x64 Registers

64 bit	32 bit	16 bit	8 bit - h/l
rax	eax	ax	ah/al
rcx	ecx	cx	ch/cl
rdx	edx	dx	dh/dl
rdi	edi	N/A	N/A
rsi	esi	N/A	N/A

- rbp/ebp - Base Pointer
- rsp/esp - Stack Pointer

Being general purpose, most of the registers may be used to store arbitrary values, though some may have defined uses with certain instructions (which we'll discuss later on). The registers listed at the bottom generally have some special uses however, where RIP/EIP points at the current place in memory we are executing, and RSP/ESP typically points to the top of the stack (which will also be discussed in greater detail later on).

## Registers (cont'd)

- rip/eip - Instruction Pointer (Program Counter)
- Additional x86\_64 Registers: r8 - r15

# Register Data and Pointers

- General Purpose Registers can contain up to pointer-sized amounts of data (4 bytes on 32 bit, 8 on 64)
- They can also contain memory addresses (pointers) to blocks of data residing elsewhere in the process.
- Addresses can be manipulated via addition, subtraction, multiplication, etc
- Square brackets dereference (access the stuff stored AT the memory address)

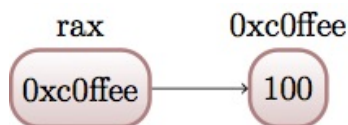
Example:

```
; A register - we will be operating on whatever  
; happens to be stored in it  
rax  
; We are attempting to access the stuff stored  
; at the address in rax (dereference)  
[rax]
```

## Register Data and Pointers - Example

**First, we'll store a pointer (memory address) in rax, and then store some stuff there:**

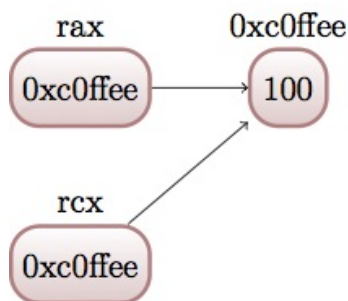
```
mov rax, 0xc0ffee ; a memory address (hopefully valid!)  
mov [rax], 100 ; now we store some data there!
```



## Register Data and Pointers - Example (Part 2)

**Now, we'll copy that address into rcx:**

```
mov rcx, rax ; now we copy the pointer!
```

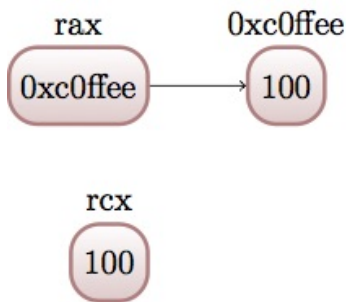


- Now both point to the same place!

## Register Data and Pointers - Example (Part 3)

**Now, we'll access the data stored at the address, and copy it into rcx**

```
mov rcx, [rcx] ; copy the data from addr 0xc0ffee into rcx
```



Please note that this replaces the old value in `rcx`, which was just the address we're accessing.

## Now, for our first instruction...

### NOP

- Does nothing! (Sort of)
- Used for padding/alignment/timing reasons
- Idempotent instruction (doesn't affect anything else in the system)
- One-byte NOP translates to the opcode: `0x90`

## Debugging Assembly

- We will be using the GNU Project Debugger (GDB)
- Command Line Debugger, provides a large set of features
  - Natively supports Python scripting
  - Supports a large number of architectures (and even quite a few languages)
  - Provides a Text User Interface (TUI) mode

## Debugging Assembly (cont'd)

- Setting breakpoints programmatically may be difficult at times
- A good strategy may include applying breakpoints directly in your code for debugging purposes
- Fortunately, an assembly instruction exists for doing just this:

```
int3 ; NOTE: no space between int and 3
```

Which translates to the following opcode:

```
0xcc
```

Tips for debugging assembly: keep an eye on registers, use breakpoints liberally!

## Debugging With GDB

Preconfiguration:

- `.gdbinit` provides a way to run a number of setup commands on launch
- Simply copy the config file to your home directory:

```
~/Desktop/handouts $ cp sample-gdbinit ~/.gdbinit
```

Launching a program with GDB:



```
~/Desktop/Lab1 $ gdb lab1
(gdb) run
...
(gdb) quit
```

## GDB

- Basic Use: Generally useful commands
  - info - Displays information (in general, or about specific commands)
  - help - Can provide context-specific help; e.g., listing available commands/options
- refresh: will redraw the console window

## GDB

- Single Stepping (step/s)
  - Can also use stepi
- Stepping Over (next/n)
  - Can also use nexti

## GDB

- Breakpoints (break)
  - Allows us to programmatically set breakpoints without modifying application source code
- info break - shows us information about all currently set breakpoints
- Removing breakpoints (clear and delete)

Example:

```
(gdb) break myfunc
Breakpoint 1 at 0x4004a4
(gdb) info break
Num      Type           Disp Enb Address
1        breakpoint      keep y   0x00000000004004a4
(gdb) delete 1
(gdb) info break
No breakpoints or watchpoints
```

Demo stepping and using GDB with a sample init file and our opcodes demo

## Memory Access Instructions

- We'll begin looking at instructions to copy and access data from various locations in memory
- Additionally, we will begin examining address calculation

## Memory Access - mov

### Description

Moves a small block of memory from a source (the right-hand operand) to destination (the left operand). An amount of data may be specified (more on this later).

## Basic Use

```
mov rax, 0x01      ; immediate - rax is now 1
mov rax, rcx        ; register - rax now has a copy of ecx
mov rax, [rbx]      ; memory - rbx is treated as a pointer
mov rax, qword [rbx + 8] ; copying a quad word (8 bytes)
```

The mov instruction simply copies data from source (the operand on the right), to destination (the operand on the left).

## Memory Access - lea

### Description

Calculates an address, but does not actually attempt to access it.

### Basic Use

```
; calculate an address by taking the address
; of what RDX points at,
; and adding 8 bytes to it (perhaps indexing
; into an array). Note that we are just calc-
; ulating the address, NOT accessing memory.
lea rax, [rdx + 8]
mov rax, [rax]      ; actually accessing the memory
```

## Memory Access - xchg

### Description

Exchanges the values provided atomically (more on this later).

### Basic Use

```
xchg rax, rcx      ; exchange two register values
; exchange a register value with a value stored in memory
xchg rax, [rcx]
```

## Lab 1

### Memory Access

- Copy the Lab1 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab1 $ cmake . && cmake --build .
~/Desktop/Lab1 $ ./lab1
```

## Assembly and Data Types

- Slightly different concept than in higher level languages
  - Typically just bytes in a buffer
  - Data 'type' is really just interpretation
  - Generally differentiated by sizes, alignment, and certain bits being set
- Some operations will preserve special properties in a given data set (such as sign, e.g., +/-)

- Other operations may expect different alignments in the data they work on, or may have issues with certain values (such as floating point)

## x86(\_64) general data sizes

- byte - "smallest" addressable unit
- word - two bytes
- dword - double word (4 bytes - pointer width on x86)
- qword - quad word (8 bytes - pointer width on x64)

## GDB: Examining Memory

- We can use GDB to examine various places in memory with "x" (for "eXamine")
- x has several options:
  - x/nfu - where n is the Number of things to examine, f is the Format, and u is the Unit size
  - x addr
  - x \$<register> - examines the memory address pointed to by the register

## GDB Formatting

- The "f" in x/nfu
- Format options include:
  - s - For a NULL-terminated string
  - i - For a machine instruction
  - x - For hexadecimal (the default, which changes when x is used)
- Example: Disassembling at RIP

```
(gdb) x/i $rip
```

## GDB Unit Sizes

- The "u" in x/nfu
- Unit size options are a bit confusing in the context of x86(\_64) assembly, and include:
  - b - bytes
  - h - Halfwords (equivalent to "word" in x86(\_64) asm; e.g., 2 bytes)
  - w - Words (4 bytes, equivalent to dwords)
  - g - Giant words (8 bytes, equivalent to qwords)

Demo - Dumping memory via GDB

# Sub Registers

64 bit	32 bit	16 bit	8 bit - h/l
rax	eax	ax	ah/al
rcx	ecx	cx	ch/cl
rdx	edx	dx	dh/dl
rdi	edi	N/A	N/A
rsi	esi	N/A	N/A

- Subregisters are still part of the bigger "parent" register
- Unless special instructions (not yet mentioned) are used, will NOT modify data in the other portions of the register.

## Memory/Register Access - mov

- When accessing memory, amount of data to copy can be specified

```
mov al, byte [rsi] ; copy a single byte
mov eax, dword [rcx] ; copy a dword (4 bytes)
mov rax, qword [rsi] ; copy a qword (8 bytes)
```

- Also, data can be copied from subregister to subregister

```
mov al, cl ; copy from cl to al
xchg al, ah ; exchange the low and high bytes in ax
```

## Register Access - movzx

### Description

Move with zero extend. When moving data that is smaller than the destination size, zero out the remaining bits.

### Basic Use

```
movzx rax, cl ; everything above al is now set to 0
movzx rax, byte [rsi + 5]
```

## Lab 2

Using subregisters, accessing smaller values, and zero extending.

- Copy the Lab2 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab2 $ cmake . && cmake --build .
~/Desktop/Lab2 $ ./lab2
```

## Structures

- NASM provides a data structure concept for convenience in handling complex data types

- More of a macro than something truly representative of C-style structs
- Very useful for keeping track of local variables or parameters (among other things)

## Structures

```

struct MyStruct
    .field1      resd 1  ; field1's size is 1 dword
    .field2      resd 1  ; field2's size is 1 dword
    .field3      resq 1  ; field3's size is 1 qword
endstruct

; ...
; This will be equivalent to: mov rax, [rdi+8]
mov rax, [rdi + MyStruct.field3]

```

## Section Review

- Byte Ordering
- Process Layout
- Registers
  - Stack Pointer
  - Base Pointer
  - Instruction Pointer
- Dereferencing Pointers via Register
- Sub registers

<b>Space</b>	Forward
<b>Right, Down, Page Down</b>	Next slide
<b>Left, Up, Page Up</b>	Previous slide
<b>P</b>	Open presenter console
<b>H</b>	Toggle this help

## Basic Operations

### Objectives

- Utilize basic arithmetic and bit operations
- Understand the difference between signed and unsigned values, from an assembly perspective
- Understand the Two's Complement representation of signed numbers
- Understand the Stack as it pertains to assembly programming, and implement functions that utilize it to load and store data

## Arithmetic Operations

Basic Math Operations

### The add and sub instructions

#### Description

Adds and subtracts arbitrary values. The destination (where the result is stored) is the first value provided.

## Basic Use

We can use a combination of registers and immediates as operands:

```
mov rax, 1
add rax, 2 ; rax now contains 3
sub rax, 1 ; rax now contains 2
mov rcx, 2
add rax, rcx ; as above, rax now contains 4
sub rax, rcx ; rax is now back to 2
```

## The mul instruction

### Description

Allow multiplication of arbitrary values. Takes a single argument, multiplies by rax/eax/ax (depending on operand size).

### Basic Use

```
mov eax, 10
mov ecx, 10
mul ecx      ; rax now contains 100

mov rax, 5
mov rcx, 7
mul rcx      ; rax now contains 35
```

Results are mostly stored in the source operand (ax/eax/rax), but may be stored in dx/edx/rdx as well if overflow occurs. The table on the next slide illustrates this.

## The mul instruction: storing results

Results are stored in the source (possible), or in a combination of registers in the configuration below:

Operand Size	First Source	Destination
byte	al	ax
word	ax	dx:ax
dword	eax	edx:eax
qword	rax	rdx:rax

## The div instruction

### Description

As with mul, div takes a single argument, and divides the value stored in the dividend register(s) by it. This is typically ax/eax/rax (and the \*dx equivalents), but may vary a bit depending on the size (chart provided on the next slide).

### Basic Use

```
; clearing the register where the
; high bits would be stored, we're only using what's in rax!
```

```

mov rdx, 0
mov rax, 10
mov rcx, 2
div rcx    ; rax now contains 5 this is rax/rcx

```

## div: operation results

Where to retrieve the results of a div from depends on the size of the arguments. The table below illustrates this relationship:

Maximum	Dividend	Quotient	Remainder
byte/word	ax	al	ah
word/dword	dx:ax	ax	dx
dword/qword	edx:eax	eax	edx
dqword/qword	rdx:rax	rax	rdx

## inc and dec

### Description

Adds or subtracts one from the provided register, storing the result in place.

### Basic Use

```

mov rax, 1    ; rax now contains 1
inc rax      ; rax now contains 2
inc rax      ; rax now contains 3
dec rax      ; rax now contains 2

```

## Lab 3

### Arithmetic Operations

- Copy the Lab3 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```

~/Desktop/Lab3 $ cmake . && cmake --build .
~/Desktop/Lab3 $ ./lab3

```

## The Stack

- Grows from high memory to low memory
- Current function typically exists within a stack "frame" (but not always!)

## Stack Frames

- RSP (or ESP) points to the top of the stack
- RBP (or EBP) points to the "base" of the stack frame

# Stack Frame Layout

0x0000	RSP	
0x0008	0x0000	
0x0010	0x0000	
0x0018	RBP	

## Expanding the Stack Frame

- Can modify the value of RSP directly to allocate more stack space:

```
sub rsp, 16
```

- But you must always ensure you clean up before the function returns:

```
add rsp, 16
```

## Stack Alignment

- x86\_64 expects 16 byte stack alignment
- Allocating odd amounts of space can cause things to break
- ALWAYS make sure you clean up your stack before returning

## GDB - Stack Frames

- Examining the Call Stack (backtrace / bt)
- Frames and information
  - frame || f - Get information about the current frame
  - info args - Get information about function arguments
  - info locals - Information about local variables

May be appropriate to demo GDB and stack frames

## New Instructions: push and pop

### Description

Push will subtract a pointer-width amount of space from RSP, and place the argument in the newly-allocated location. Pop performs the opposite action, storing the value just below RSP in the register provided, and adding a pointer-width amount to RSP. For every push, you will need to pop!

### Basic Use

```
mov rax, 1      ; 1 is now stored in rax.
push rax        ; 1 is now stored at the top of the stack
pop rcx         ; rcx now contains 1
```



# Growing the Stack

After a push operation:

0x0000	RSP	
0x0008	Old RSP	
0x0010	0x0000	
0x0018	0x0000	
0x0020	0x0000	
0x0028	RBP	

# Restoring the Stack

After a pop operation:

0x0000	Old RSP	
0x0008	RSP	
0x0010	0x0000	
0x0018	0x0000	
0x0020	0x0000	
0x0028	RBP	

## Lab 4

Stack Operations

- Copy the Lab4 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab4 $ cmake . && cmake --build .  
~/Desktop/Lab4 $ ./lab4
```

# Negative Numbers

## Two's Complement



- Negative numbers on the x86(\_64) platform are represented via Two's Complement

On understanding Two's Complement: think of what happens when a mechanical counter (like the one pictured on the slide) counts down to zero, and rolls over. You might see it flip all the numbers over: e.g., 9999

## Two's Complement

- Invert the bits of the number (in binary), and add one!

Decimal	Positive (bin)	Negative (binary)
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100

## Two's Complement (cont'd)

- Simplifies addition operations
- Unified Add/Sub

Example: Adding 2 and -1

```
Carry Row:  11
             1111
            + 0010
            ----
             0001
```

## Sub Registers and Sign extending

- When copying smaller data into a register, sign extending may be used (rather than zero extending)
- Sign extending preserves the "signed" attributes of the data being copied.
- The movsx instruction (just like movzx) handles this.

# The movsx Instruction

## Description

Much like movzx, movsx can be used to move data into a portion of a larger register, while preserving its sign.

## Basic Use

```
mov cl, -1
movsx rax, cl ; rax now contains -1.
```

## Bitwise Operations

### Bit shifting

- Two unsigned shift operations exist: shl (shift left) and shr (shift right)
- Shifting moves the bits in the register over the direction (left or right) and number of bits specified
- Bits that fall off the end (and overflow) will disappear, except for the last one, which ends up in the carry flag (which we'll discuss later)
- The extra space created gets padded with 0's

### Left Shift Diagram

The following snippet of assembly:

```
mov rax, 1
shl rax, 1
shl rax, 3
```

Can be modelled by the following table:

Decimal	Binary	State
1	00000001	Initial
2	00000010	shl rax, 1
16	00010000	shl rax, 3

### Right Shift Diagram

Similarly, the following snippet of assembly:

```
mov rax, 32
shr rax, 1
shr rax, 4
```

Can be modelled by the following table:

Decimal	Binary	State
32	00100000	Initial
16	00010000	shr rax, 1
1	00000001	shr rax, 4

## Binary and/or

- and can be used to determine whether or not one or more bits are set in
- or will tell you if the bit is set in at least one place
- Both take two operands, one of which will hold the result after the operation completes

### Use:

```
mov rax, 1          ; rax contains 00000001
mov rcx, 5          ; rcx contains 00000101
```

```
and rax, rcx        ; rax contains 00000001
or rax, rcx          ; rax contains 00000101
```

Another way to think about this (if familiar with sets and set theory): AND gives us the intersection between the two sets of bits, OR gives us their union.

## And Table

Set	Binary
First	01010011
Second	01000010
Result	01000010

## Or Table

Set	Binary
First	01010011
Second	01001010
Result	01011010

## Binary not

- Inverts the bits in a given register

Example:

```
mov rax, 0          ; rax now contains 00000000
not rax              ; rax is now all 1's (or 0xffffffff)
```

Similarly:

```
mov rcx, 1      ; rcx now contains 1
not rcx         ; rcx now contains:
                ; 0xffffffff (all 1's minus the first bit)
```

## Properties of eXclusive Or

- XOR yields 1 only if the bit is set in either the source or destination, but NOT both
- Any value XOR'd with itself is 0.
- 0 XOR'd with any value is that value
- For numbers A, B, and C, if  $A \oplus B = C$ , then  $C \oplus A = B$  and  $C \oplus B = A$ .

## XOR table

Assembly	First Value	Second Val	Result
xor rax, rax	01010011	01010011	00000000
xor rax, rcx	01000010	01001010	00001000
xor rcx, rax	01001010	00001000	01000010

## Rotating Bits

- The values in the register are rotated the indicated number of places to the right or left
- Bits that are rotated off the end of the register and moved back to the beginning

Instruction:

```
mov rax, 1      ; rax contains 1 (00000001)
rol rax, 1      ; rax now contains 2 (00000010)
ror rax, 1      ; rax now contains 1 (00000001)
ror rax, 1      ; rax now looks like: (10000000)
```

## Signed Bit Operations

- Shift operations that are sign aware exist (SAR for right and SAL for left)
- Work in the same fashion as shr/shl, except for how bits shifted off the end are treated (bits still disappear, but the sign of the resulting value is retained)

## Lab 5

Bit operations

- Copy the Lab5 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab5 $ cmake . && cmake --build .
~/Desktop/Lab5 $ ./lab5
```

# Section Review

- Arithmetic Operations
- The Stack
  - Stack Frames
  - Stack Alignment
- Signed Values and Two's Complement
- Bit Operations

<b>Space</b>	Forward
<b>Right, Down, Page Down</b>	Next slide
<b>Left, Up, Page Up</b>	Previous slide
<b>P</b>	Open presenter console
<b>H</b>	Toggle this help

# Control Flow

## Objectives

- Understand and utilize status flags and conditional control flow
- Understand and utilize x86(\_64) string instructions and corresponding instruction prefixes
- Understand and implement methods utilizing a variety of calling conventions (both x86 and x86\_64)

# FLAGS

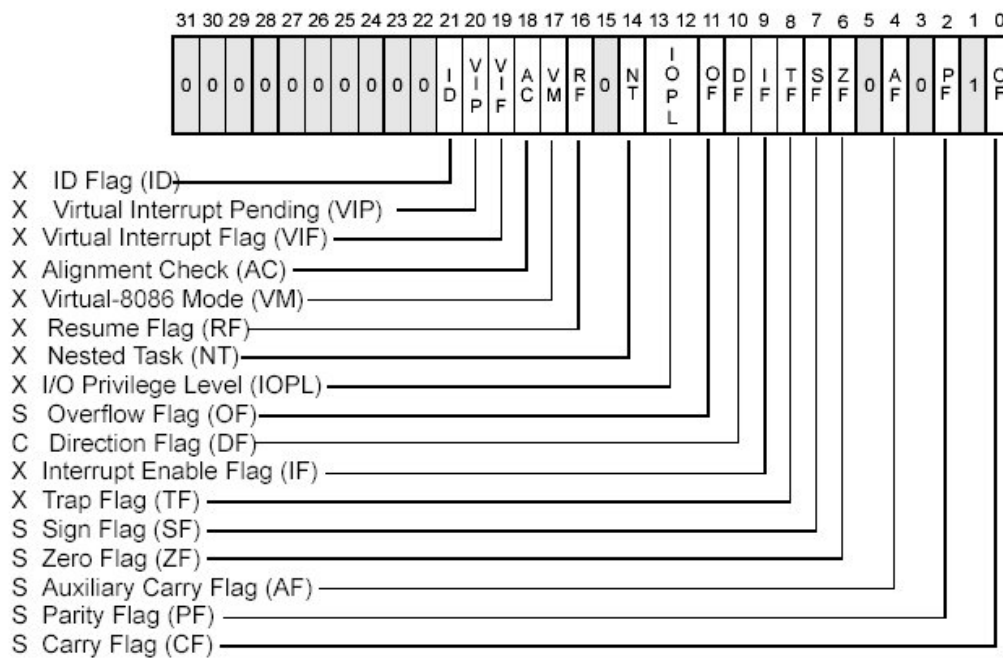
- A register that contains a variety of bits representing state and status information
- Varies in size, but many portions (in newer processors) aren't used

FLAGS 16 bits


EFLAGS 32 bits

RFLAGS 64 bits

## Flag Layout



S Indicates a Status Flag  
C Indicates a Control Flag  
X Indicates a System Flag

 Reserved bit positions. DO NOT USE.  
Always set to values previously read.

## EFLAGS Register

Credit: the Intel Manual

## Flags we care about now

- Zero Flag (ZF) - set when an operation that sets the zero flag produces a zero - includes arithmetic and bitshift operations
- Carry Flag (CF) - set when an arithmetic borrow or carry occurs during add/sub - e.g., the result of an add would have set bit 33 (in x86), or bit 65 (in x86\_64)
  - also set with some bitshift operations (such as when a bit falls off the end in a shl/shr)

## Flags we care about now (cont'd)

- **Overflow Flag (OF)** - Indicates that sign bit of the result of an operation is different than the sign bits of the operands
  - Ex.: Adding two large positive numbers ends up producing a negative result (due to overflow)
- **Sign Flag (SF)** - Set to indicate the result of an operation is negative

## Accessing the Flags

- Can be set and checked manually

- Some have special instructions for set and clear (which we'll talk about later)
- Flag register can be accessed and set via pushf(d|q)/popf(d|q)

## pushf and popf

### Description

Pushes the flags register (or the first 16 bits... eflags (32 bits) or rflags (64 bits) if pushfd or pushfq) onto the stack, and pops the value on top of the stack into the flags register (or eflags/rflags)

### Basic Use

```
pushf    ; flags have been pushed to the stack
; ... do stuff
popf     ; flags have been restored!
```

## Lab 6

Flag manipulation...

- Copy the Lab6 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab6 $ cmake . && cmake --build .
~/Desktop/Lab6 $ ./lab6
```

## Control Flow

### Line Labels

- Global and local

```
global _label:
    ; stuff

.local _label:
    ; more stuff
```

## Everybody jmp .around

- jmp provides an unconditional branch, or transfer of execution to the target

```
.label1:
    xor rax, rax
    inc rax
    mov rcx, rax
    jmp .label2
    mov rsp, rax    ; never gets executed
.label2:
    shl rcx, 3    ; execution continues here...
    xchg rcx, rax
    ret
```



## call and ret

- Similar to jmp, but with a few key differences
- Functionally equivalent to: push rip followed by a jmp X
- Typically indicates a function call

```
mov rax, 1
call label1 ; push RIP, jump to label1
jmp label2
label1:
    ror rax, 1
    ret      ; returns control returns to "jmp label2"
label2:
    ; ...
```

## More on ret

- Pops the return pointer off the stack and jumps to it
- Used to return to the last point of execution (as shown on the previous slide)

The stack, during function execution:


0x0000	Parameters	
0x0008	Old RIP/return pointer	
0x0010	Old RBP	
0x0018	...	

Once we get to the end, and we're ready to return:

```
; ...
pop rbp
ret
```

Our stack frame does something like this:

0x0000	Parameters	
0x0008	Old RIP/return pointer	
0x0010	...	
0x0018	...	



popping off the old RBP, then popping the return pointer, and jumping to it (effectively "pop rip")

## A Side Note About Functions

- Typically store the stack pointer ((E)R)SP at the top of the function
- If stored, must be (re)stored before returning
  - If we don't, our stack location will be off
  - If left at the top of the stack, we will return ONTO the stack!
- This is not always done, as in FPO (Frame Pointer Optimization/Omission)
- Functions will be covered in more depth later

```
myfunc:
    mov rbp, rsp
    push rbp
    ; ...
    pop rbp
    ret
```

## Conditional Control Flow: Comparisons

### cmp

- Compares two values by subtraction (e.g., SUB op1, op2)
- Sets flags to indicate whether the values were equal, or if one was larger
- Flags set by this instruction: CF, OF, SF, ZF, AF, and PF

Example:

```
xor rax, rax
cmp rax, 0 ; they're equal! the ZF is now set
```

### test

- Compares two values by doing a bitwise AND
- The SF, PF, and ZF get set by this operation
- Often used to test whether or not a register is 0

Example:

```
mov rax, 1
test rax, rax ; the ZF is set to 0, as the result isn't 0

; ...

xor rax, rax
test rax, rax ; the ZF is now 1
```

## Jcc

- A large set of conditional branch instructions
- Most execute based on the value of one or more flags
- Some common conditional jumps:
  - je or jz - Jump if Equal (or Jump if Zero)
  - jne/jnz - Jump if Not Equal (or Not Zero)
  - ja - Jump if Above (if the operand compared previously is greater)
  - jb/jc - Jump if Below (or Jump if Carry)
- Many others - Refer to the intel manual for a comprehensive list

A large number of the Jcc instructions actually evaluate to the same thing (e.g., JE vs JZ)

## Jcc Cont'd

A simple check to see if the result of an operation is 0:

```
xor rax, rax
test rax, rax
; Because the zero flag is set here, we jump to the end
jz .end
```

```

mov rsi, rax ; not executed
; ...
.end:
ret

```

## Jcc Cont'd

A simple loop:

```

mov rcx, 10 ; set our loop count to 10
xor rax, rax ; set rax to 0
; This evaluates to: 10 + 9 + 8 + ... + 1 + 0
.continue:
    add rax, rcx ; add the current value of rcx to rax
    dec rcx ; subtract 1 from rcx
    test rcx, rcx ; check to see if rcx is 0
    jnz .continue ; jump back to .continue, if rcx isn't 0

ret

```

## loop

- A simple macro for `dec rcx/test rcx,rcx/jnz <target>`
- Expects ECX/RCX to be populated with a counter variable

The loop from the previous slide could be re-written

```

mov rcx, 10
xor rax, rax
.continue:
    add rax, rcx
    loop .continue
ret

```

## Lab 7

Execution control flow...

- Copy the Lab7 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```

~/Desktop/Lab7 $ cmake . && cmake --build .
~/Desktop/Lab7 $ ./lab7

```

## String Instructions

- What a "string" means to x86(\_64)
  - Really just a string of bytes
  - No particular qualms about terminators (e.g., '0')
- Several prefixes and a flag that will modify behavior (more on those later)
- All of them have the unit to move/copy/initialize/scan appended to the end (e.g., `scasb` vs `scasw` vs `scasd`, etc)

## String Instructions - Cont'd

- Common features:
  - RSI (or ESI, in x86) is treated as a pointer to the beginning of the "source"
  - RDI (or EDI, in x86) is treated as a pointer to the beginning of the "destination"

- RCX (or ECX, in x86) is assumed to hold the count, if needed
- RAX (or EAX, in x86) is assumed to hold the value to evaluate, if needed (e.g., store, compare against, etc)
- Typically increments source and/or destination register pointers by the amount of data operated on (e.g., movsb would add 1 to both RSI and RDI, where movsd would add 4)

## Some Common Instructions

- Scan String - scas(b/w/d/q) - Scans a string located at RDI for the value found in RAX/EAX/AX/AL (depending on size used), and increments the pointer
- Store String - stos(b/w/d/q) - Initializes the string located at RDI to the value pointer at by RAX/EAX/AX/AL (depending on size used) and increments the pointer
- Load String - lods(b/w/d/q) - Copies the value from RSI into RAX/EAX/AX/AL, and increments the pointer
- Move String - movs(b/w/d/q) - Copies data from RSI into RDI, and increments both pointers.
- Compare String - cmps(b/w/d/q) - Compares the values stored at RSI and RDI, and increments the pointer, updating the RFLAGS (or EFLAGS) register with the result.

## Prefixes

- Several instruction prefixes available to modify behavior - looping the instruction over a section of memory
- All of them tend to use RCX/ECX/etc as a termination condition - decrementing each execution
- Some prefixes available:
  - REP - continue performing the action RCX times.
  - REPNE - continue performing the action RCX times, or until the FLAGS register indicates the operands were equal.
  - REPE - Continue perform the action RCX times, or until the FLAGS register indicates the operands were not equal.
- Often used by compilers to essentially inline C string functions (such as strlen, memset, memcpy, etc)

## Prefix Examples

- Unconditional:

```
xor rax, rax    ; rax is now 0
mov rcx, 20     ; rcx now contains 20
mov rdi, _my_string_buf
rep stosb       ; Continue to store 0 till rcx
                ; is 0
```

- Conditional:

```
xor rax, rax
mov rcx, 20
; assume the buffer below contains a string
mov rdi, _my_populated_buf
repne scasb     ; continue until we hit a NULL byte
; RCX now contains 20 - <the number of bytes we checked>
; ...
```

## The Direction Flag

- Controls the direction buffers are traversed when using the REP\* prefixes
- If set during execution/an operation, ALWAYS clear after (or crashes will likely occur)

```
std             ; the direction flag has been set
; do stuff here
cld             ; clear the direction flag, continue operations
```

# Lab 8

## String Operations

- Copy the Lab8 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab8 $ cmake . && cmake --build .  
~/Desktop/Lab8 $ ./lab8
```

## Functions

### Calling Conventions: x86

- Microsoft - many calling conventions exist for x86
  - Different implications for how arguments get passed
  - Different implications for stack cleanup after function returns
  - Name mangling is often used to differentiate
- System V x86 Calling Convention
  - Most POSIX-compliant (and POSIX-like) platforms abide by this
    - Such as Linux, Solaris, BSD, OSX, etc
    - Also called cdecl
- Other Calling Conventions

Many others exist (such as safecall or pascal) on Windows alone Only a few will be covered here (outside of passing mention)

### Microsoft Conventions: stdcall

- Indicated to compiler (from C) by \_\_stdcall prefix
- Arguments pushed on the stack (in order from right to left)
- The function being called (the "callee") cleans up the space allocated
- Name gets decorated with an appended "@X", where X is the number of bytes to allocate (num args \* 4)

### stdcall - cont'd

Standard call in action - Stack Cleanup:

```
; Equiv: void __stdcall myfunc(int a, int b);  
_myfunc@8:  
    ; do stuff  
    ret 8    ; we've cleaned up 8 bytes
```

Optionally, we can clean up like this:

```
_myfunc@4:  
    ; do stuff  
    add esp, 4  
    ret
```

### stdcall - cont'd

Standard call in action - Accessing Parameters:

- If EBP hasn't been pushed to the stack:

```
_myfunc@8:
    mov eax, [esp + 4] ; parameter 1-above the return pointer
    mov ecx, [esp + 8] ; parameter 2-above param 1
    ; do stuff
    ret 8
```

- Otherwise:

```
_myfunc@8:
    push ebp
    mov ebp, esp
    mov eax, [ebp + 8] ; above both the ret ptr and old ebp
    mov ecx, [ebp + 12]
    pop ebp
    ret 8
```

## Microsoft Conventions: cdecl

- This is also the System V calling convention (e.g., what most non-microsoft platforms use)
- Parameters passed in the same fashion as in stdcall
- Stack cleanup is different, the calling function (e.g., caller) is responsible for cleanup
- No real name mangling, aside from a leading underscore "\_"

## cdecl - cont'd

Cdecl in action: Stack cleanup

```
; callee
_myfunc:
    push ebp
    mov ebp, esp
    ; do stuff
    pop ebp
    ret

_caller:
    ; ...
    push 2 ; arg 2
    push 1 ; arg 1
    call _myfunc
    add esp, 8 ; clean up
    ; ...
```

## Microsoft Conventions: fastcall

- First two arguments (from left to right) passed via registers (ECX and EDX)
- Remaining arguments pushed on the stack (right to left, as with cdecl and stdcall)
- Cleanup is performed by callee (as with stdcall)
- Name mangling is similar to stdcall, but an additional "@" is prepended (e.g., "\_@myfunc@8")

## Conventions: thiscall

- "Special" convention used for C++ non-static member functions
- Defines a method of passing the "this" pointer (which allows those functions access to a specific instance of a class)
- Slightly different between Microsoft and System V
- Microsoft: The "this" pointer is passed in ECX, other parameters work like stdcall
- System V: Works like cdecl, but the "this" pointer is the first argument to the function
- C++ name mangling is a more complex topic (and somewhat compiler dependent)

# x64 Calling Conventions

- Only one convention for each (Mostly... there are still some oddballs like vectorcall, but we aren't going to dive into those)
- thiscall on x64 (both conventions) passes the "this" pointer as an implicit first argument (as it does for System V x86)
- Both conventions work similarly to \_\_fastcall, passing arguments in registers (though the registers differ between platforms)

## Microsoft x64 Calling Convention

- Uses 4 registers to pass the first 4 parameters (RCX, RDX, R8, and R9)
- Floating point values are passed via SIMD registers (XMM0-3... we'll talk more about this later)
- Remaining values are added to the stack
- Caller's responsibility to clean up (as with \_\_cdecl)

## Shadow Space

- x64 Calling Conventions require stack allocation for passed variables
- Intent is to allow function being called to immediately spill registers (if desired)
- Windows ABI requires space to be allocated for 4 registers (regardless of function parameter count)
- Additional arguments (beyond 4) are added via the stack
  - BUT in the location they would normally occur at if all parameters were passed that way
  - Example: param 5 would begin at [rsp + 0x20]

## Microsoft x64 Calling Convention

No parameters:

```
callee:
    ; ...
    ret

caller:
    sub rsp, 0x20    ; 8 * 4 - for register spillage
    call callee
    add rsp, 0x20    ; cleanup
```

## Microsoft x64 Calling Convention

5 Or More Parameters

```
sub rsp, 0x28        ; space to store 5 params
mov rcx, 0x41         ; param 1 = A
mov rdx, 0x42         ; param 2 = B
mov r8, 0x43          ; param 3 = C
mov r9, 0x44          ; param 4 = D
mov [rsp + 0x20], 0x45 ; param 5 = E
call myfunc
add rsp, 0x28         ; cleanup
```

## Microsoft x64 Calling Convention

Some additional reading on Microsoft's x64 calling convention:

- <https://blogs.msdn.microsoft.com/oldnewthing/20040114-00/?p=41053/>

# System V x64 Calling Convention

- Similar to the Microsoft calling convention, but more values are passed via registers
- The first 6 arguments are passed via register (RDI, RSI, RCX, RDX, R8, and R9)
- Floating point arguments go in SIMD registers (XMM0-7)
- Additional arguments are pushed onto the stack
- Shadow space is not required, but stack must remain 16-byte aligned
- Red zone optimization provides free stack space for leaf functions

## Red Zone

- Allows use of the next 128 bytes below RSP without modifying stack pointer
- Further function calls WILL clobber space
  - Because of this, Red Zone use is most suitable for leaf functions
  - Safe from interrupt handlers, etc.

## System V x64 Example

Calling strlen

```
extern strlen

; ensure NULL termination!
mystring db "this is a string", 0x00

call_strlen:
    mov rdi, mystring
    call strlen
    ret
```

## Return Values

Typically, the value returned at the end of the function call will be stored in RAX (for x64), or EAX (for x86)

## Register Preservation - x86

- Volatile: EAX, ECX, and EDX don't need to be saved during a function call
- All others must be preserved.

## Register Preservation - x64

- Windows: Volatile Registers (don't need to be preserved by callee)
  - RAX, RCX, RDX, R8, R9, R10, and R11
  - XMM0-3 and 5
  - All others need to be preserved
- System V
  - Most registers are volatile (need to be preserved by caller if the values are to be retained)
  - Exception: RBP, RBX, and R12-15 are non-volatile (must be preserved)

## Additional Links

More information on both x64 calling conventions:



- <http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/>

## Lab 9

### Functions

- Copy the Lab9 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab9 $ cmake . && cmake --build .  
~/Desktop/Lab9 $ ./lab9
```

## Windows Functions Lab

### Functions - Calling Conventions (x86)

- Copy the WinFunctions folder to your Windows system
- Copy the nasm binary to WinFunctions\ASM\nasm.exe
- Edit WinLab.nasm under WinFunctions\ASM\ASM\WinLab.nasm
- build via VS2015 (as normal), or via msbuild using the following command:

```
C:\..\WinFunctions\ASM> "%VS140COMNTOOLS%vsvars32.bat"  
C:\..\WinFunctions\ASM> msbuild ASM.sln
```

## Section Review

- Flags
- Jumps
- Call and ret
- string instructions
  - prefix
- Functions and calling conventions

Space	Forward
Right, Down, Page Down	Next slide
Left, Up, Page Up	Previous slide
P	Open presenter console
H	Toggle this help

## Hardware Overview

## Objectives

- Understand the different privilege modes of operation, and some of their implications
- Understand basic memory segmentation and some descriptor tables
- Understand, at a basic level, virtual memory
- Understand basic processor features, control registers, and how they fit together
- Understand and implement Model Specific Registers (MSRs)
- Utilize x86(\_64) instructions to identify the current processor, and understand how to programmatically query its capabilities

# Processor Modes

## User and Kernel modes

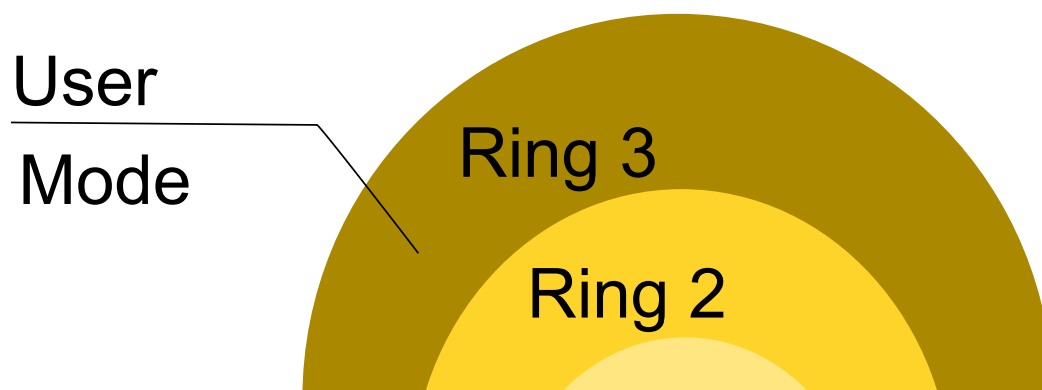
- x86(\_64) defines several modes of operation (or "rings") the processor can work in
- Each mode has various instructions (and portions of memory) it is allowed to perform
- User mode (which is least privileged) is Ring 3 (which is where most of your applications reside)
- Kernel mode (most privileged) is Ring 0, which is where the core (or kernel) of the OS resides
- Rings 1 and 2 are (largely) unused by most operating systems

## User Mode

- Mode of least privilege (Ring 3)
- Cannot touch more privileged memory sections
- Cannot execute "privileged" instructions
- Typically must task some facility in the kernel in order to get resources

## Kernel Mode

- Mode of most privilege (Ring 0)
- Can access any portion of memory (and change protection)
- Can perform privileged instructions
- Device drivers often run here (in addition to the core portions of the OS)



# Interrupts and Memory Segmentation

## Tables and Memory Segmentation

- Intel Specifies a number of tables, populated by the OS, which map functionality to the processor.
  - The Interrupt Descriptor Table, or IDT
  - The Global Descriptor Table, or GDT
  - The Local Descriptor Table, or LDT
- Virtual memory also relies on a set of page tables

## Interrupts and The Interrupt Descriptor Table

### Interrupts

- What are interrupts?
  - Interrupts provide a special mechanism to alert the kernel of an event
  - Some (though not all) can be temporarily disabled
  - Specified via the IDT
- Interrupts can be generated many ways:
  - Via hardware events (e.g., a keypress on a keyboard)
  - Page or segmentation faults
  - Software interrupts also exist
  - Many others

We won't spend too much time on this particular topic (as it only loosely relates to segmentation), but it is important to understand how interrupts work (at a low level), and this will give you the foundational knowledge needed to understand what's happening under the hood later on.

### Interrupts (Cont'd)

- Interrupt Service Routines (ISRs)
  - Functions that respond to interrupts
  - Set via Interrupt Gates in the IDT (See below)
- Interrupt Gates
  - Essentially the entry number (in the IDT) of the ISR you want to call
  - The 'int' assembly instruction will call the corresponding ISR
  - The 'iret' instruction is provided (on the kernel side) to return back to user mode

### Interrupts - Example:

The following code will perform an `exit(0)` on Linux (x86):

```
mov eax, 0x01    ; the system call number
mov ebx, 0x00    ; first parameter
int 0x80         ; interrupt
```

### Segment Registers

- Segment registers are a special type of register not covered yet, which come in a variety of flavors

- Each of them can be mapped to provide a special "view" of a section of memory
- Most modern operating systems use a "flat" memory model, forgoing segmentation (almost) entirely
- Still have some real world applications, particularly in Windows

## Segment Registers

- The Basics:
  - CS - Code Segment
  - DS - Data Segment
  - SS - Stack Segment
  - FS - Far Segment
  - GS - Global Segment
- Example: Getting a value from some offset into a segment

```
; Retrieves the value stored at offset
; 0x33 into the Global Segment
mov rax, [gs:0x33]
```

## Segmentation - Segments

- What is a segment?
  - Describes a logical section of memory
  - Specifies who can access it (e.g., what privilege level you need)
  - Indicates the range (start address and length)
- Why are they important?
  - Part of the segmentation model, used to map a flat section of memory to the segment registers

## Segmentation

The Global Descriptor Table (GDT)

- Initialized by the operating system
- Contains various segment descriptors in its entries
- The GDT Register (GDTR) indicates where it is located
- Contains information about how the memory in your system gets mapped
- Also (partially) defines how the transition from user to kernel mode occurs
- Intended to be a global structure

## Segmentation

The Local Descriptor Table (LDT)

- Similar to the GDT
- Intended to have smaller scope: e.g., a per-process construct

## Segmentation - Real World examples

- Microsoft uses segmentation to provide fast access to key data structures
- The Thread Environment Block (TEB) in user mode
  - Hangs off of the FS register in x86/GS in x64
  - Provides lots of important per-thread information

- The Processor Control Block (or KPRCB) in kernel mode
  - Hangs off of the FS register in x86/GS in x64
  - Provides lots of important per-processor information

## Segmentation

Further Reading (if interested):

- The OSDev Wiki describes GDT initialization - [http://wiki.osdev.org/GDT\\_Tutorial](http://wiki.osdev.org/GDT_Tutorial)
- The Segment Descriptor Cache - Mr. Robert Collins (from a Dr. Dobbs article) - <http://www.rcollins.org/ddj/Aug98/Aug98.html>

## Other Processor Features

## Processor Security Features

- DEP/NX
- SMEP/SMAP
- Page Protection
- Write Protection

## Control Registers

- Control CPU enforcement of a variety of features
- Most security features are enabled in this fashion
- Requires privileged execution (Ring0) to access
- Other features (such as hardware virtualization) also enabled in this fashion
- Feature mappings detailed in the Intel manuals

## Virtual Memory

- Allows a virtual abstraction of hardware addresses
- Paging enabled via CR1
- Page Table location stored in CR3
- Tables and Directories provide fast lookup of address translations

# Page Tables and Directories

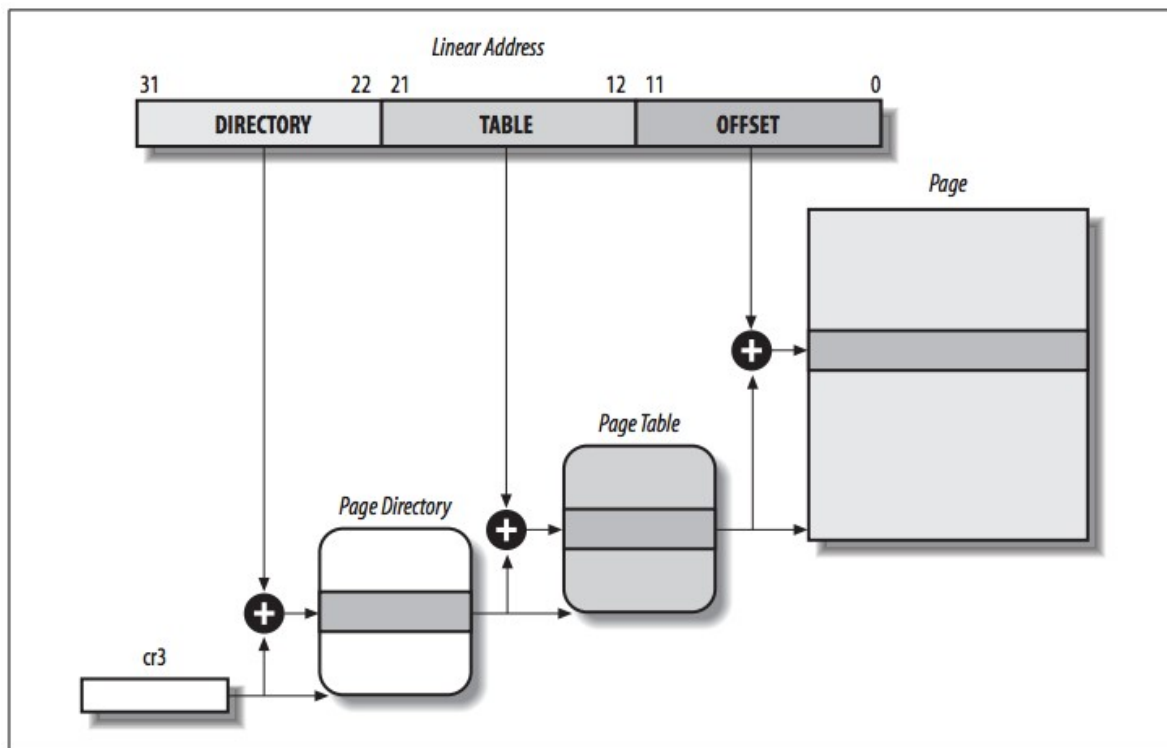


Image Credit: <https://notes.shichao.io/utlk/ch2/>

## Model Specific Registers (MSRs)

- Many of these exist for x86(\_64)
- Most only accessible in privileged mode
- used (sometimes) via RDMSR and WRMSR
- Others have special instructions
- Mainly documented by vendor (e.g., Intel manual)

## Time Stamp Counter

- Can read from user mode (via rdtsc)
- Can only modify from kernel mode
- Low bits of result are stored into EAX/high bits in EDX
  - This is the same on both x86 and x86\_x64
  - x64 - rdtsc will clear the high bits of storage registers
  - Results can be combined on x64 to full width with a left-shift and a bitwise or

## Feature Support

- The CPUID instruction can provide information about the current CPU
  - Vender string
  - Model number
  - Size of internal cache
  - Various features supported

- The instruction behaves similarly on BOTH x86 and x86\_64

## Feature Support (Cont'd)

- The value in EAX at the time of the CUID call determines what information comes back
  - 0 -> Vender ID String - stored in EBX/EDX/ECX
  - 1 -> Returns a bitfield containing supported features
  - ...

## Lab 10

### MSRs and CUID

- Copy the Lab10 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab10 $ cmake . && cmake --build .
~/Desktop/Lab10 $ ./lab10
```

## Section Review

- Ring 0/3
- Interrupts
- Memory Segmentation
  - GDT
  - LDT
- MSRs

<b>Space</b>	Forward
<b>Right, Down, Page Down</b>	Next slide
<b>Left, Up, Page Up</b>	Previous slide
<b>P</b>	Open presenter console
<b>H</b>	Toggle this help

## Floating Point and SIMD

### Objectives

- Understand at a basic level how floating point numbers are represented
- Perform basic floating point operations
- Understand and utilize the Single Instruction Multiple Data hardware to perform basic operations

# x87 - The Floating Point Architecture

Register Name(s)	Size	Register Type
R0-R7	80 bits	Data
Control	16 bits	Special
Status	16 bits	Special
Tag	16 bits	Special
Last Instruction Pointer	pointer-width (sorta)	Special
Last Data Pointer	pointer-width (sorta)	Special
Last Instruction Opcode	10 bits	Special

## Data Registers

- Can contain any 32 or 64 bit data
- Can't transfer directly from standard registers (e.g., EAX/RAX)

Sign	Exponent	Significand
bit 79	bits 64-79	bits 0-64

## GDB

Display floating point registers:

```
(gdb) tui reg float
```

## Floating Point Exceptions

- Floating point exceptions are separate from typical interrupts
- They are the only interrupts you can disable from an unprivileged operating context
- The previous instruction pointer (and other contextual information) are stored via special registers

## Floating Point - Additional Information

- Floating point hardware also includes its own FLAGS register
- As mentioned in the previous slide, contextual information is stored in special registers under certain circumstances



Register Name(s)	Size	Register Type
R0-R7	80 bits	Data
Control	16 bits	Special
Status	16 bits	Special
Tag	16 bits	Special
Last Instruction Pointer	pointer-width (sorta)	Special
Last Data Pointer	pointer-width (sorta)	Special
Last Instruction Opcode	10 bits	Special

## Floating Point Encoding

- Data encoding is a great deal more complicated for floating point than other types
- Floating point numbers are represented via scientific notation (sort of)
- We can store floats in one of three ways:
  - Single Precision -> which is 32 bits
  - Double Precision -> which is 64 bits
  - Quad Precision -> which is 128 bits

## Floating Point Encoding (cont'd)

Four parts to the equation:

- Significand (also called the mantissa) - This is the decimal representation of our number. A non-zero value will always be in the left-most position
- Radix - The base to multiply by (e.g., 10)
- Exponent - The power to raise the radix to

## Floating Point Data Encoding

- x87 Registers Show the split between different parts (under the hood)

So if we have a number like 1378.5, and our hardware looks like this:

Sign	Exponent	Significand
bit 79	bits 64-79	bits 0-64

We have:

$$\begin{array}{rcl}
 \textit{Significand} & & \textit{Radix} \\
 1.3785 & \times & 10^3 \longleftarrow \textit{Exponent}
 \end{array}$$

# Binary Representation

1. We take the base 2 representation of the number
2. We transform the number, such that it can be represented via scientific notation
3. The exponent is encoded using a "biased" value, which expedites compare operations
4. Since we know that the left-most position of the significand will always be 1, it is dropped for single precision numbers (though it is preserved in doubles).

## Exponent Values

Bias value added to exponent

Parameter	Single	Double	Extended
Total Size	32	64	80
Significand Size	23	53	63
Exponent Size	8	11	15
Exponent Bias	127	1023	16383

e.g., an exponent of 3 (111 in binary) would get added with 127 (in the case of a single precision float), or 1111111, to get 10000110.

## Special Exponent Values

- Some values are preserved for special cases in exponent representation
  - 00000000 (all 0s) - This value is used to encode +/- infinity
  - 11111111 (all 1s) - This value is used to indicate NaN (Not a Number)
- NaN simply indicates that the floating point encoded value is not valid.

## Working with Floating Point

- The floating point registers are treated like a stack
- Values get pushed on (via "load" instructions) and popped off/copied (via "store" instructions)

## FPU Instruction Set

Basic Operations

- Loads (push)
  - fld - Loads a floating point value from the indicated location, onto the stack
  - flid - Loads an integer value, encoding it as a double, onto the stack
- Stores
  - fst - Stores the value on the top of the stack at the specified location (either memory or elsewhere in the floating point stack)
  - fstp - Performs the same operation as above, but also pops the value off the stack.
  - fist - Converts the value at the top of the stack to an integer, and stores it at the destination.
- Exchange: fxch - Swaps the contents of the given floating point registers

# FPU Conditional Move

fcmovcc - Move if: Copies the contents from the requested register in the stack to the top if the condition is satisfied

- b (e.g., fcmovb) - Move if below (if CF is set to 1)
- nb - Move if not below (e.g., CF is 0)
- e - Move if equal (e.g., ZF is 1)
- ne - Move if not equal (ZF is 0)
- be - Move if below or equal (CF is 1 or ZF is 1)
- nbe - Move if not below or equal (CF is 0 and ZF is 0)

# FPU Arithmetic Operations

- fadd/fsub/fmul - Performs the requested operation on the source and destination operands, storing the results in the destination
- fiadd/fisub/fimul - Similar to above, but performs the requested operation on a floating point and integer value

## Lab 11

Floating Point Operations

- Copy the Lab11 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab11 $ cmake . && cmake --build .  
~/Desktop/Lab11 $ ./lab11
```

# Single Instruction, Multiple Data (SIMD)

## What is SIMD?

- Set of specialized hardware and instructions
- SSE (Streaming SIMD Extensions) is part of this (among others)
- Provide a mechanism for operating on "vectors" of data at a time
- Can perform a variety of operations 128 bits at a time

## SIMD Hardware

- x86: 8 SSE registers available, from XMM0 - XMM7
- x64: provides 16 SSE registers, from XMM0 - XMM15

# SIMD Operations - Moving Data

- Operations typically operate on vector or scalar values
  - Scalar - 32 bits (single element)
  - Vector - All elements of the SIMD register (128 bits)
- Moving data in/out:
  - movups - Move 128 bits of data between memory and/or SIMD register(s), unaligned
  - movaps - Move 128 bits of data between memory and/or SIMD register(s), aligned
  - movhps - Move 64 bits into the high part of a SIMD register
  - movlps - Move 64 bits into the low part of a SIMD register

- movss - Move a 32 bit value between memory and/or SIMD register(s)

## SIMD: Moving Around

```
movups xmm0, [rdi]      ; moving 128 bits of data into xmm0
movss  xmm1, [rsi]      ; moving 32 bits of data into xmm1
mov    [rdx], eax
movss  xmm2, [rdx]
```

## SIMD Arithmetic

The table below lists arithmetic instructions, both the scalar and vector variations:

Vector	Scalar	Description
addps	addss	Adds operands
subps	subss	Subtracts operands
mulps	mulss	Multiplies operands
divps	divss	Divides operands

## SIMD Arithmetic

Example use:

```
movups xmm0, [rdi]      ; load first vector
movups xmm1, [rsi]      ; load second vector
addps  xmm0, xmm1       ; add the two
movups [rdx], xmm0       ; store the result
```

## SIMD Comparisons

Comparison Operations

- cmp\* operation is a bit strange at first
  - Compares a combination of registers/memory
  - Stores the result in the first operand
  - Third param indicates the type of compare to do (next slide)
  - Result stored as 0 if the condition is false, -1 (all 1's) if true
- Min and max preserve the values that are greater (or smaller) between operands

## SIMD Comparisons

Cmp's third parameter:

Number	Operation	C Equiv
0	cmp	
1	cmpl	
2	cmpr	
3	cmpl	
4	cmpr	
5	cmpl	
6	cmpr	
7	cmpl	
8	cmpr	
9	cmpl	
10	cmpr	
11	cmpl	
12	cmpr	
13	cmpl	
14	cmpr	
15	cmpl	
16	cmpr	
17	cmpl	
18	cmpr	
19	cmpl	
20	cmpr	
21	cmpl	
22	cmpr	
23	cmpl	
24	cmpr	
25	cmpl	
26	cmpr	
27	cmpl	
28	cmpr	
29	cmpl	
30	cmpr	
31	cmpl	
32	cmpr	
33	cmpl	
34	cmpr	
35	cmpl	
36	cmpr	
37	cmpl	
38	cmpr	
39	cmpl	
40	cmpr	
41	cmpl	
42	cmpr	
43	cmpl	
44	cmpr	
45	cmpl	
46	cmpr	
47	cmpl	
48	cmpr	
49	cmpl	
50	cmpr	
51	cmpl	
52	cmpr	
53	cmpl	
54	cmpr	
55	cmpl	
56	cmpr	
57	cmpl	
58	cmpr	
59	cmpl	
60	cmpr	
61	cmpl	
62	cmpr	
63	cmpl	
64	cmpr	
65	cmpl	
66	cmpr	
67	cmpl	
68	cmpr	
69	cmpl	
70	cmpr	
71	cmpl	
72	cmpr	
73	cmpl	
74	cmpr	
75	cmpl	
76	cmpr	
77	cmpl	
78	cmpr	
79	cmpl	
80	cmpr	
81	cmpl	
82	cmpr	
83	cmpl	
84	cmpr	
85	cmpl	
86	cmpr	
87	cmpl	
88	cmpr	
89	cmpl	
90	cmpr	
91	cmpl	
92	cmpr	
93	cmpl	
94	cmpr	
95	cmpl	
96	cmpr	
97	cmpl	
98	cmpr	
99	cmpl	
100	cmpr	

0	Equal	==
1	Less Than	<
2	Less or equal	<=
3	Unordered	n/a
4	Not Equal	!=
5	Not Less than	!(x < y)
6	Not less or equal	!(x <= y)
7	Ordered	n/a

## SIMD Comparisons

Operations

VectorScalarDescription

maxps maxss Obtains maximum of operands

minps minss Obtains minimum of operands

cmpps cmpss Compares operands, all 1's or 0's returned.

## SIMD Comparisons

Example:

```
movups xmm0, [rax]
movups xmm1, [rcx]
cmpps xmm0, xmm1, 4 ; find the values that are not equal
```

## SIMD Bitwise Operations

VectorDescription

andps Bitwise and of operands

orps Bitwise or of operands

xorps Bitwise xor of operands

## GDB

Display SIMD registers:

```
(gdb) tui reg vector
```

## Lab 12

SIMD

- Copy the Lab12 folder (and its contents)
- Modify the \*.nasm file (Each function should have a comment block - lines starting with ';' containing instructions)
- Build and run using the following commands:

```
~/Desktop/Lab12 $ cmake . && cmake --build .  
~/Desktop/Lab12 $ ./lab12
```

## Section Review

- Floating Point Architecture
  - Interrupts
  - Register Stack
  - Flags
- SIMD
  - Registers
  - Vectors vs Scalars

<b>Space</b>	Forward
<b>Right, Down, Page Down</b>	Next slide
<b>Left, Up, Page Up</b>	Previous slide
<b>P</b>	Open presenter console
<b>H</b>	Toggle this help

## Section 2

Practical Application

## Utility Methods

## Objectives

- Implement a number of basic standard library functions in assembly
- Implement some essential data structures in assembly
- (OPTIONAL) Complete provided bonus labs

Assembly: A Practical Application

This section of the course will be mostly hands-on, with relatively short sections of lecture, followed by larger blocks of labs. The labs have (mostly) been staged so that there are a set of easier exercises, followed by a section of bonus material. If you

complete any of the labs early, the bonus material is provided for you to work on. Most of the bonus sections have a C implementation provided for reference.

## Utility Functions

- Copy and search functions (Some methods from previous labs, such as the string instruction lab, may be helpful here)
  - strlen
  - memcpy
  - memset
  - memchr
  - memcmp
  - strchr
  - strcmp
  - strcpy
  - strstr
- Conversion
  - atoi

atoi

```
int atoi(char* c)
{
    long len = 0;
    int step = 1;
    int accum = 0;

    if(NULL == c)
        return 0;

    // We will start from the 1's place
    len = strlen(c);
    --len;
    while(len >= 0) {
        // subtract 0x30 (the difference
        // between the number and its
        // ordinal value as an ASCII character),
        // multiply by the step,
        // and add.
        accum += (c[len] - 0x30) * step;
        --len;
        // Move up by a factor of 10.
        // First: 1's place, then 10's, etc.
        step *= 10;
    }

    return accum;
}
```

## Sorting (Bonus Labs)

- Insertion Sort
- Quicksort

## Lab 13 - Utility Functions

## System Calls - an Introduction

# Objectives

- Understand the basic function and implementation of system calls (including legacy methods)
- Understand the basic functionality provided by a C Runtime
- Implement a system call wrapper in assembly
- Begin work on a C Runtime

# System Calls

- What they are
- How they work

# Legacy System Call Method

- `int 0x80/0x2e`

# Modern Alternative

- x86: `sysenter / sysexit`
- x64: `syscall`

# My First Syscall

Wrapping system calls

- x86
- x64

# Getting Information

- Man pages often have a comprehensive list of required flags (even if definitions are buried in header files)
- May be more than 1 section to a man page (if the page overlaps with a utility page "2" generally has dev docs)

```
~$ man mmap
~$ man 2 open
```

# C Runtimes: A good \_start

- What is a runtime?
- `int main()` vs your program's real entry point
- Building without a standard library
- stuff your crt usually does

# Compiling with no CRT

- All functions will need to be implemented/provided
- Initial effort: Wrap system functionality
  - `sys_exit`
  - `write`



# sys\_exit

RAXRDI

60 status (int)

## Implementing sys\_exit

```
    mov rax, 60 ; the syscall number (in this case exit)
    xor rdi, rdi ; argument 1, the exit code
syscall
    ret
```

## Some Setup

- STDOUT - A special kind of file descriptor (1)
- sys\_write

RAXRDIRSI RDX

1 fd buffer ptrCount

## Lab 14

Finally, time for "Hello, World!"

## Allocation

## Objectives

- Understand the basic roles and responsibilities of a simple allocator
- Understand the function of the mmap syscall
- Implement a simple allocator

## Allocating Memory

- The Heap - no longer just a call to malloc
- How do we add memory to our process?

## About Allocators

- Many different strategies for heap management
  - Lots of special cases to consider
  - Multithreading adds more concerns (we'll talk more about this later)
- Our strategy here will try to remain simple

# Our Allocator Strategy

- Getting new memory from the kernel every time we need to allocate is very inefficient
- We'll want to build a list of unused (or "free") chunks to hand out when allocations are requested
- When a chunk is requested, we can check the free list first (if initialized), to see if we have something that will work
- If not, we'll need to allocate memory

## Asking for More

- We can actually ask for memory from the kernel in two ways:
  - mmap - This is the more "modern" approach; we can ask the kernel for more memory by requesting an anonymous page mapping (we'll be discussing mmap in much greater detail over the next few sections)
  - brk - We won't really touch this too much; it lets you extend or shrink the end of the memory mappings in your program
- Some additional initialization logic can also be added to `_start`, if needed

## mmap

- Lets us create a memory mapping
- May be backed by a file, or anonymous
- This will be the base for our allocator

RAX RDI                      RSI   RDX              R10 R8                                      R9

9    addr (or NULL)lengthProtectionFlagsDescriptor (or NULL)offset

## Arguments

- Protection (from `mman-linux.h`)

```
%define PROT_READ      0x1      ; Page can be read.
%define PROT_WRITE     0x2      ; Page can be written.
%define PROT_EXEC      0x4      ; Page can be executed.
%define PROT_NONE      0x0      ; Page can not be accessed.
```

- Flags (need to be OR'd together)

```
%define MAP_ANONYMOUS  0x20    ; Don't use a file.
; ...
%define MAP_PRIVATE    0x02    ; Changes are private.
```

## Creating a Heap

- Beginning the Process: malloc and free
- Steps to success
  - Initialization: Handled in `_start`
  - Making Requests: Define a "block" size
  - Keeping a list: Maintain a list of "free" chunks

# munmap

RAXRDIRSI

11 addrlenlength

## Problem Description

Some pseudo-C to describe our malloc strategy:

Some initial structure information:

```
/**
 * Our free list node structure definition.
 * In this case, just a simple linked list.
 */
struct heap_node;
typedef struct heap_node {
    struct heap_node* next;
    unsigned long size;
    unsigned char data[1];
} heap_node;

typedef struct free_list {
    heap_node* head;
} free_list;

/* Add structure overhead to our alloc size */
#define HEAP_ALLOC_SIZE(x) (sizeof(heap_node) + x)
/* The beginning of our free list; head starts out NULL */
free_list free_start;
```

## Problem Description

```
/* NOTE: n is the allocation size requested */
heap_node* check_free_list(unsigned long n)
{
    heap_node* prev = NULL;
    heap_node* current = NULL;

    /*
     * If our list is empty, we need to
     * allocate.
     */
    if(NULL == free_start->head)
        return NULL;

    current = free_start->head;
    prev = current;
    if (current->size >= n) {
        /* Remove from the list */
        free_start->head = (current->next != NULL) ?
            &current->next : NULL;
        return current;
    } else if (current->next == NULL) {
        return NULL;
    }

    do {
        /* Walk the list, find and remove a
         * chunk of at least size n
         */
    } while(current->next);
```

```

        return NULL;
    }

```

## Problem Description

```

void* allocate(unsigned long n)
{
    heap_node* tmp = NULL;
    unsigned long alloc_size = 0;

    /*
     * We'll check the free list first,
     * and see if there is a suitable chunk
     */
    if(NULL != (tmp = check_free_list(n))) {
        /* We found a match! */
        return (void*)tmp->data;
    }

    /*
     * Since we need to allocate, we have
     * to add enough to our header to
     * account for our heap_node struct!
     */
    alloc_size = HEAP_ALLOC_SIZE(n);
    /* We'll allocate some space */
    if(NULL == (tmp = mmap_anon_page(alloc_size))) {
        return NULL;
    }

    tmp->size = n;
    tmp->next = NULL;
    /* Return the beginning of the data buffer */
    return (void*)tmp->data;
}

```

## Problem Description

```

void deallocate(void* p)
{
    heap_node* node = NULL;

    if(NULL == p)
        return;

    /*
     * Subtract the size of the bookkeeping struct to get
     * back to the top of the heap_node structure in
     * memory
     */
    node = (heap_node*)((char*)p - sizeof(heap_node));
    /* Zero the user provided data */
    memset(node->data, 0, node->size);
    /* Add the node to the free list */
    node->next = &free_start->head;
    free_start->head = node;
}

```

## Additional Steps to Consider

- Keep track of the number of items on the free list; release some if it becomes too large
- Keep multiple free lists based on chunk size

## Lab - Creating an Allocator

Space	Forward
Right, Down, Page Down	Next slide
Left, Up, Page Up	Previous slide
P	Open presenter console
H	Toggle this help

## I/O

## Objectives

- Understand and Identify basic facts about Linux files and file descriptors
- Implement wrappers for several file I/O system calls
- Understand and utilize file-based process bookkeeping mechanisms (via /proc)

## Files and Operations

- UNIX Model - Everything is a file!
- File Descriptors
  - A bookkeeping mechanism to represent your access to a resource
  - Some typically reserved numbers: 0/1/2 (for std in/out/err)

## File Operations

- Read and Write
- Open and Close (for existing files)
- Unlink (for deleting)
- Syncing changes - msync and fsync

## Process Information and Virtual Memory

### mmap - A different use

- Can be used to map a file into memory
- Essentially (part of) how executables are loaded
- Can be more efficient for I/O

### mmap - Some new flags

- Required to be set to Shared for changes to appear in base file
- Changes may not show up until either munmap or a call to msync

```
%define MAP_SHARED      0x01 ; Share changes.
```

## Process Information and Virtual Memory

- /proc - a special type of directory
- /proc/self
- Getting to process parameters - /proc/self/cmdline

# Syscall Info - pt1

Syscall	RAXRDI	RSI	RDX	R10	R8	R9
mmap	9	address	length	ProtectionFlags	Descriptor	offset
munmap	11	address	length			
read	0	Descriptor	buffer	ptrCount		
write	1	Descriptor	buffer	ptrCount		
open	2	filename (char*)	flags	mode		
close	3	Descriptor				

# Syscall Info - pt2

Syscall	RAXRDI	RSI	RDX	R10	R8	R9
unlink	87	Path (char*)				
msync	26	address	startlength	flags		
fsync	74	Descriptor				

# Flags and Modes

## Msync options

```
; Flags to `msync'.
#define MS_ASYNC      1 ; Sync memory asynchronously.
#define MS_SYNC       4 ; Synchronous memory sync.
```

## Open options:

- One of the following options must be chosen:

```
#define O_RDONLY      00
#define O_WRONLY      01
#define O_RDWR        02
```

- Zero or more of the following may be chosen:

```
#define O_CREAT        0100 ; Create the file
#define O_TRUNC        01000 ; Truncate (if exists)
#define O_APPEND       02000 ; Append
```

# Mode

- If file is being created, specifies permissions to set on it
- Can be one of the following values (follow UNIX-style permission rules) specified on the next slide

S\_IRWXU 000700 user (file owner) has read, write and execute permission

S\_IRUSR 000400 user has read permission

S\_IWUSR 000200 user has write permission

S\_IXUSR 000100 user has execute permission

S\_IRWXG 000700 group has read, write and execute permission

S\_IRGRP 000400 group has read permission

S\_IWGRP 000200 group has write permission

S\_IXGRP 000100 group has execute permission

S\_IRWXO 000700 others have read, write and execute permission

# Lseek

- Lets you move to an offset within a file
- Returns the distance (in bytes) your current offset is from the file's beginning

# Lseek

```
syscall RAX RDI RSI RDX R10 R8 R9
```

```
lseek 8 int (fd) long offset or origin
```

# Lseek

Values for origin (indicating where to move from):

```
%define SEEK_SET      0    ; Seek from beginning of file
%define SEEK_CUR      1    ; Seek from current position
%define SEEK_END      2    ; Seek from the end of the file
```

# Lab - File I/O

# Directories

- Several syscalls exist to read directories
- Focus will be on getdents

# Getdents

Syscall RAX RDI RSI RDX R10 R8 R9

getdents78 int (fd) struct dirent\* buff size

# Misc Syscalls

- Execve - Execute a program
- First arg is the binary path to run
- Second is argv[]
- Third is environment strings

Syscall RAX RDI RSI RDX R10 R8 R9

execve59 char\* char\* argv[] char\* envp[]

# Threading

## Objectives

- Understand at a basic level the Linux threading model
- Understand some of the pitfalls of working with multithreaded applications
- Understand and implement some of the basic synchronization tools provided by the x86\_64 instruction set
- Implement a simple threading library

## What is a Thread?

- Each thread is essentially a separate stream of execution
  - The register values for each thread are different
  - This is referred to as "context"
  - Transitions from one thread to another is referred to as "switching context"
- Multiple threads may be running at the same time
- It is difficult (if not impossible!) to predict how scheduling will occur

## Synchronization

- Access to data needs to be synchronized (meaning: we need to make sure only one thread at a time can modify it)
- Race conditions happen if multiple threads are trying to update the same data at once



# Safe memory access

- Think in terms of "transactions"
- The lock prefix
- Some instructions, such as xchg, implicitly lock

## The clone Syscall

- This syscall creates a new process, but allows you to specify some amount of sharing with the parent process
- Threads and processes in Linux are synonymous, but the amount of resources they share may differ

## Clone

Syscall RAX RDI      RSI      RDX      R10

Clone 56   Clone flags Stack Pointer parent tid child tid

## Flags

Some flags we'll want for our thread library:

```
%define CLONE_VM      0x00000100 ; VM shared between procs.
%define CLONE_FS      0x00000200 ; fs info shared
%define CLONE_FILES    0x00000400 ; open files shared
%define CLONE_SIGHAND  0x00000800 ; signal handlers shared.
%define CLONE_THREAD   0x00010000 ; add to same thread group.
```

## Basic Steps to Success

1. Allocate Stack Space
2. Call Clone
3. Transfer Control to Intended function
4. Block main thread until children are done

## Allocating Stack Space

- The stack grows down, and thus we need to give the high part of the new stack segment to clone
- mmap is the best choice to do this, as it has flags that let us specify that we wish to use the allocated memory as a thread stack:

```
%define MAP_GROWSDOWN 0x0100
```

## Calling Clone

- As seen above, clone has several arguments
- For our purposes, only two are really useful: the flags argument (RDI), and a pointer to our new stack
- We'll need to get to the end of the new stack:

```
; assuming rsi contains a pointer to
; our newly allocated stack segment
lea rsi, [rsi + STACK_SIZE]
```

## Calling Clone (cont'd)

- After clone, both the parent and child continue executing in the same place (right after the syscall)
- The child (our newly created thread) will have the same initial register values as the parent, with two exceptions:
  - RAX - this will be set to 0
  - RSP - this will now point to our new stack
- The parent will now have (in RAX) the PID of the thread

## Running the Thread Function

- A number of options exist to transfer control to the new function
  - Pass via non-volatile register
  - Pass via stack
- If any thread-specific setup is to be done, just need to:

```
test rax, rax
jz .child
jmp .parent
```

- easiest method of control transfer is probably passing via the new stack

## Running the Thread Function (cont'd)

- Recall from our section on control flow that the ret instruction essentially performs "pop rip" (or pop X + jmp X)
- Thus, we can now set our stack up so that the new thread function will be our return point (we'll just change the way our stack looks before the call to clone):

Now, instead of:

```
lea rsi, [rsi + STACK_SIZE]
; ...
syscall
```

we will:

```
lea rsi, [rsi + STACK_SIZE - 8]
mov [rsi], rdi ; our function pointer
; ...
syscall
; ...
ret
```

## Running the Thread Function (cont'd)

- With our function set to be at the top of the new stack, we can now simply return
- On return, our new thread will begin executing inside of the thread function
- This works wonderfully, BUT
  - What happens when the thread function returns?

## Exit

- We need to ensure that we call exit after execution completes
- Since we are at the top of the stack to begin with, there is no place to go
- The easy solution:

```

    lea rsi, [rsi + STACK_SIZE - 8]
    mov [rsi], rdi ; our function pointer
    ; ...
    syscall
    test rax, rax ; check to see if we are parent/child
    jnz .parent ; jump to end if we are the parent
    pop rax ; pop the function pointer (top of stack)
    call rax ; call our thread function!
    ; ...
    call exit ; call exit (no place to return)
.parent:
    ; ... ; parent: cleanup/return
    ret

```

## A Better Exit Strategy

- We can take the previous code a step further, and add another return address to the stack
- If we put exit first, we will still be able to transfer control in the same fashion, but don't need to wrap out child function with additional calls

```

lea rsi, [rsi + STACK_SIZE - 8]
mov [rsi], exit ; our exit function
sub rsi, 8 ; go back just a bit
mov [rsi], rdi ; now our function pointer
; ...
syscall
; ...
ret

```

## Waiting till done

- Since all of the threads are part of the same thread group, can't wait() for them
- Alternate strategy is to "pause"
  - Will block us until a signal of some sort happens (such as all child threads exiting)
  - Syscall takes no arguments

## Pause

Syscall RAX RDI RSI RDX R10

Pause 34

## Race Conditions

- Can occur when multiple threads access data at once, where the data is being modified
- Can be rather difficult to spot at first
- Multiple strategies exist to mitigate
  - Locks
  - Atomic Instructions
- Can be difficult to get absolutely correct

Think in terms of a busy intersection with no stop sign.

# What do Race Conditions look like?

```
mov rax, [rdi] ; we load our data
; but by the time we reach here,
; any number of things could have
; happened to the value in the pointer
test rax, rax
jz .bad_stuff
```

## More Problems

- Deadlocks
- Starvation
- Recursive locking
- And much, much, more!

## Making Atomic adds and Comparisons

```
lock xadd          ; exchange and add
lock bts           ; bit test and set
lock btr           ; bit test and clear
lock cmpxchg       ; compare and swap
xchg               ; implicitly locks
```

## XADD

### Description

Exchanges the values in its two operands, adds them together, and stores the result into the first operand. Can be used with the lock prefix.

```
xadd rax, rdx
; rdi contains: a pointer to "20", rax contains: 10
lock xadd [rdi], rax ; rdi now contains a pointer to "30"
```

## Bit Test and Set, Bit Test and Clear

### Description

Tests and sets the selected bit in memory, sets the carry bit to indicate the previous value. Bit test and clear resets the bit to 0, setting the carry bit to indicate the previous value.

```
lock bts dword [rdi], 0
jc .was_set
; ...
lock btr dword [rdi], 0
jc .was_set
```

## Compare and Swap

### Description

Compares the first operand to the value in RAX/EAX/AX/AL, if they are equal, copies the second operand into the destination, and sets the zero flag (ZF). Otherwise, it leaves the destination alone, and clears the zero flag.

```
mov rax, [rdi]
lock cmpxchg [rdi], 1
jnz .not_replaced
```

## Creating a Simple Spinlock

```
lock_func:
    ; ...
    lock bts [rdi], 0
    jc .done
    jmp lock_func
    ; ...

.done:
    ret

unlock_func:
    ; ...
    lock btr [rdi], 0
    ; ...
```

## Lab and Demo - Threading and Synchronization

### Review

Space	Forward
Right, Down, Page Down	Next slide
Left, Up, Page Up	Previous slide
P	Open presenter console
H	Toggle this help