First Precept!

COS 318: Operating Systems

Precept Objectives:

In precepts we will usually cover two components:

- Outline of assignment, and what is necessary to know/do
- Understanding why the assignment is necessary & useful for Operating Systems

Today we will also be going over some of the high-level details of the course, and covering x86 asm, which is necessary for the 1st assignment.

Icebreaker

Because this may be your first in-person precepts in a while, it would be good to meet the other people interested in Operating Systems!

High Level Objectives of OS

Below, sorted by amount of time to learn & difficulty to achieve:

1. Be able to understand the various functions and structure of an OS
    a. Better understand the interface w/ the system in user programs
2. Can read & understand existing OS source, given enough time
3. Can modify existing OSs (Linux Patches!)
4. Can develop new components/structure of an OS

- Google has Fuchsia & Android, Microsoft has Windows, Apple has MacOS/iOS
- Plenty of others, like VMWare which have OS adjacent services

Today's Material

- Timeline for Project 1
  - Also covering what is expected in the design review
- Review of x86 assembly (or introduction for those who used ARM) which should be sufficient for doing the 1st assignment

Project 1 Schedule

- Design Review: Mon(9/13) & Wed(9/15)
  - Sign up for 10-min slot from Mon (8:30pm-10:30pm) or Wed (3pm-7pm)
  - Complete set up and answer posted questions
- Project 1 Precept: Mon (9/13) & Tue (9/14), 7:30pm - 8:20pm. **Due: Sun (9/26), 11:55pm**

Design Review

- USBs will be given at the Design Review
- Write print_char and print_string assembly functions
- Be ready to describe:
  - How to move the kernel from disk to memory
  - How to create disk image
  - (More specific guidelines are provided on the project page)

x86 Assembly Tutorial

- x86/IA-32/i386 Assembly Overview

Components:

  - Registers, Flags, Memory Addressing, Instructions, Stack / Calling Convention, Directives, Segments

- BIOS
- GDB

Registers

General Purpose Registers: 8,16,32 bits

Size (in Bits)

32	16	8
EAX	AX	AH/AL
EBX	BX	BH/BL
ECX	CX	CH/CL
EDX	DX	DH/DL
EDI	DI	DIL
ESI	SI	SIL
EBP	BP	BPL
ESP	SP	SPL

Segment Registers: 16 bits

CS
DS
SS
ES
FS
GS

Instruction Pointer (EIP): 32 bits

Flags (EFLAGS): 32 bits

Flags

Flags are single bits in the EFLAGS register, where each bit has a different purpose such as:

- Controlling the behavior of CPU
- Saving the status of the last instruction
- More details at: https://en.wikipedia.org/wiki/FLAGS_register

Flags

- Status:
  - CF: Carry flag
  - ZF: Zero flag (op = 0)
  - SF: Sign flag (op < 0)
- Control Flags:
  - IF: Interrupt flag (sti, cli), En/Disable interrupts

AT&T Syntax

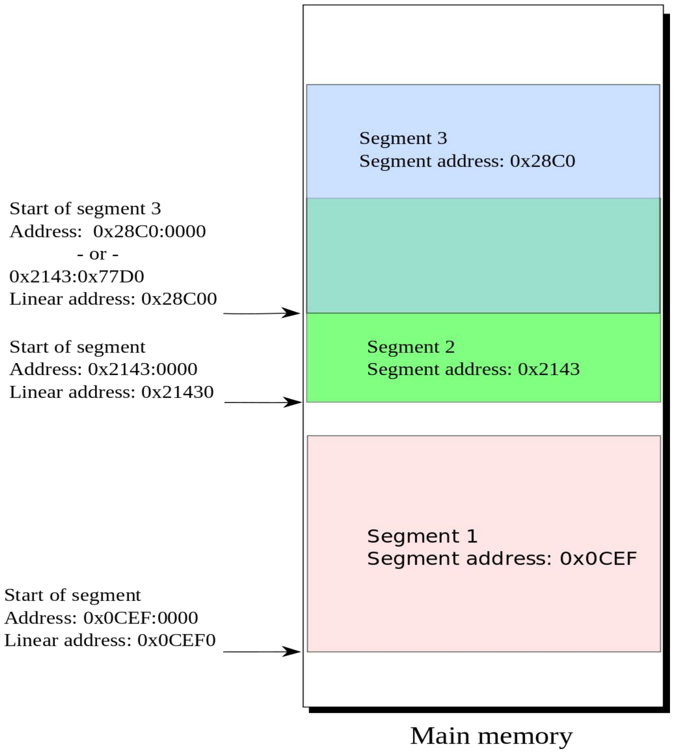
- Instruction format: **`instr src, dest`**
  - `movw %ax,%bx`
- Prefix register names with %: %ax
- Prefix constants (immediate values) with $
  - `movw $1,%ax`
- Suffix instructions with size of data
  - b for byte (8 bit), w for word (16 bit), l for long (32 bit)
  - i.e. movb, movw, movl

Memory Addressing in Real Mode

- 1MB memory
  - Valid address range: 0x00000-0xFFFFF= 20 bits = 1MB
- 16-bit segments and 16-bit offsets into each segment: (segment << 4) + offset

Memory Addressing (Real Mode)

- Format (AT&T syntax):
  - **segment:displacement(base,index,scale)**
- Offset = Base + Index * Scale + Displacement
- Address = (Segment << 4) + Offset
- Displacement/Scale must be a constant w/o $
- register that could be base: i.e. %bx, %bp
- Register that could be index: i.e. %si, %di
- Segment registers: %cs, %ds, %ss, %es, %fs, %gs



Instructions: Arithmetic & Logic

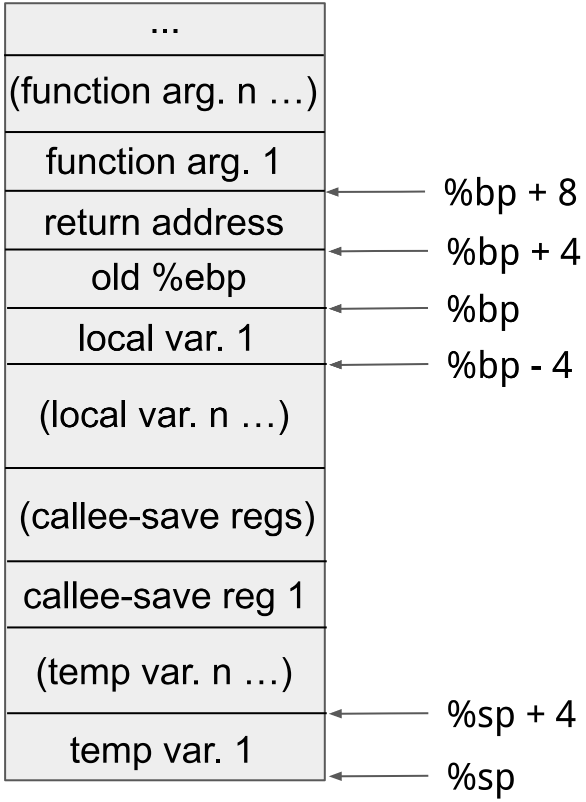
- **add/sub{l,w,b} source,dest (dest - source)**
- **inc/dec/neg{l,w,b} dest**
- **cmp{l,w,b} source,dest (dest > source)**
- **and/or/xor{l,w,b} source,dest ...**
- Restrictions
    - No more than one memory operand, other must be register

Instructions: Data Transfer

- **mov{l,w,b} source, dest**
- **xchg{l,w,b} source, dest**
- movsb/movsw
    - %es:(%di) $\leftarrow$ %ds:(%si), must be these registers
    - Often used with %cx to move a number of bytes
        - movw $0x10,%cx
        - rep movsw
- Segment registers can only address mem w/ registers

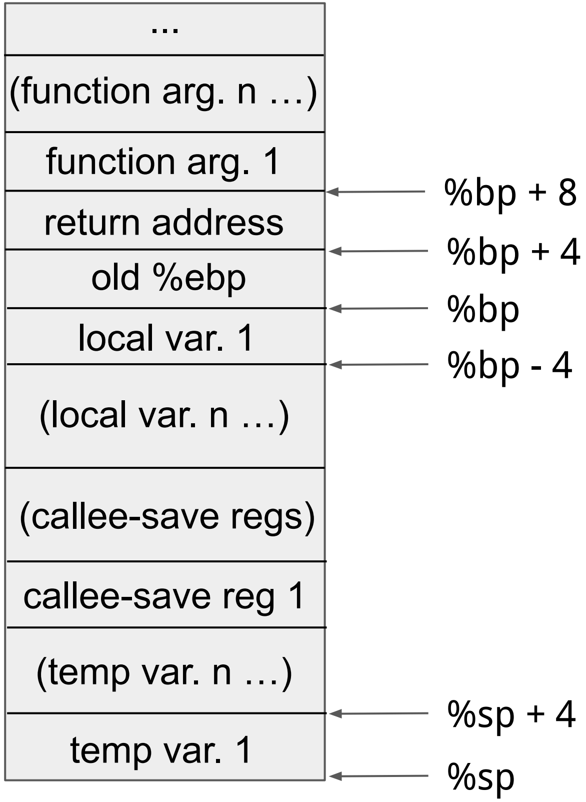
Stack Layout

- Grows from high to low
  - Lowest value address = “top” of stack
- %sp points to top of the stack
  - Used to reference temporary variables
- %bp points to bottom of stack frame
  - Used for local vars + function args.



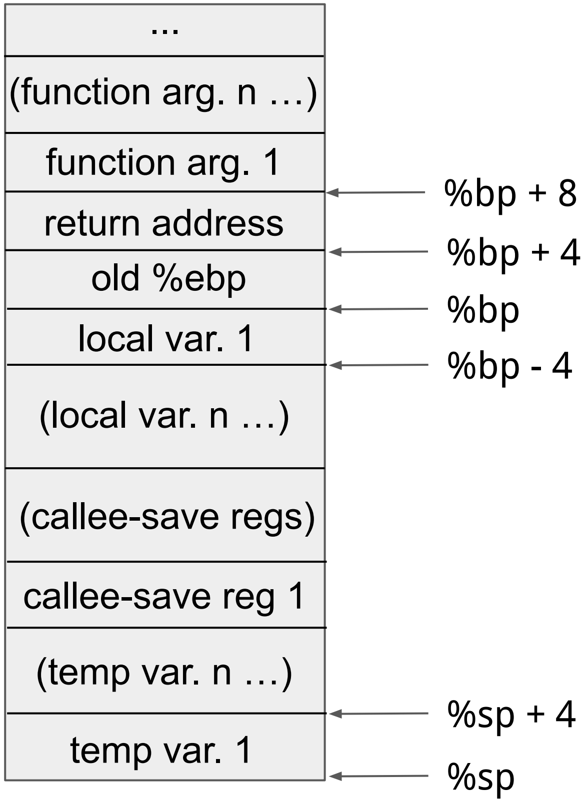
Calling Convention

- When calling a function:
  - 1. Push caller-save regs onto stack
  - 2. Push function args onto stack
  - 3. Push return address then jump
- In function after call:
  - 1. Push old %bp + set %bp = %sp
  - 2. Allocate space for local variables
  - 3. Push callee-save regs if necessary



Instructions: Stack Access

- **pushl source**
  - %sp ← %sp - 4
  - %ss:(%sp) ← source
- **popl dest**
  - dest ← %ss:(%sp)
  - %sp ← %sp + 4



Instructions: Control Flow

- **jmp label**
  - %eip ← label
- **ljmp NEW_CS, offset**
  - %cs ← NEW_CS
  - %eip ← offset
- **call label**
  - push %eip
  - %eip ← label
- **ret**
  - pop %eip

Instructions: Conditional Jump

- Relies on %eflags bits
  - Most arithmetic operations change %eflags
- **j* label**
  - Jump to label if * flag is 1, can be z(ero), e(qual), etc
- **jn* label**
  - Jump to label if * flag is 0

Assembler Directives

- Commands that “direct” the assembler
  - Are not instructions
- Examples:
  - .globl - defines a list of symbols as global
  - .equ - defines a constant (like #define)
  - .bytes, .word, .asciz - reserve space in RO memory

https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html

Assembler Segments

- Organize memory by data properties
  - .text - holds executable instructions
  - .bss - holds zero-initialized data (e.g. static int i;)
  - .data - holds initialized data (e.g. char c = ‘a’;)
  - .rodata - holds read-only data
- Stack - Initialized by linker & loader
- Heap - Start defined by compiler, contents by programmer (usually thru an included library)

BIOS = Basic Input/Output System

- Firmware (compiled with the device itself)
- Configures buses/connections to and from various hardware devices
- Setups RAM for usage, puts CPU in 20-bit Real Mode
- Determines which device (CD, USB, etc) has a loadable boot sequence.
  - Loads boot sequence into RAM, then transfers control to top of boot sequence
- Afterwards, BIOS provides abstractions over low-level interfaces, such as reading/writing from external disks

- Source:
  - https://wiki.osdev.org/System_Initialization_(x86)
  - https://wiki.osdev.org/BIOS

BIOS Services

- Use BIOS services through `int` instruction
  - Store parameters in specified registers, such as AH/AX
  - Software Interrupt: triggers a func in firmware
- **asm: `int SERVICE_NUM`**
  - i.e. int $0x10: Video services, int $0x13: Disk services

Useful GDB Commands

- r - show register values
- sreg - show segment registers
- s - step into instruction
- n - next instruction
- c - continue
- u <start> <stop> - disassembles C code into assembly
- b - set a breakpoint
- d <n> - delete a breakpoint
- bpd / bpe <n> - disable / enable a breakpoint
- x/Nx addr - display hex dump of N words, starting at addr
- x/Ni addr - display N instructions, starting at addr

Assembly ***can be useful***

In order to understand **efficiency** of implementation, sometimes looking at the assembly of generated code can be useful:

https://godbolt.org/z/5rxdz8s8K