

# Introduction to Operating Systems

Mythili Vutukuru  
CSE, IIT Bombay

# What is a computer system?

- Software + hardware to run user applications and programs, to accomplish some tasks
- Components:
  - **Hardware**: CPU, memory, I/O devices, ...
  - **System software**: Operating System (OS), ...
  - **User software**: user applications (browser, email client, games, ...)
- Real-life computer systems (e.g., an e-commerce website) have multiple interconnected computers, each running one or more applications that communicate with each other

# Real life computer systems are complex

- Real life systems are **complex**
  - Multiple interacting components and sub-systems
  - Each component independently developed, but have to work together for a common purpose
  - Prone to failures, bugs, crashes
- But still, we expect:
  - The system always does what it is supposed to do ... (**functional correctness**)
  - Quickly, efficiently, for a large number of users, lots of data ... (**performance**)
  - Even when it is overloaded or when failures occur ... (**reliability**)

# Why study operating systems?

- Knowledge of hardware (architecture) + system software (OS), and how user programs interact with these lower layers, is essential to writing “good” (high performance, reliable) user programs
  - What exactly happens when you run a user program?
  - How to make your program run faster and more efficiently?
  - Why do programs crash and how to avoid it?
  - How to make your programs more secure?
- OS expertise is one of the most important building blocks when building high performance, robust, complex real life systems

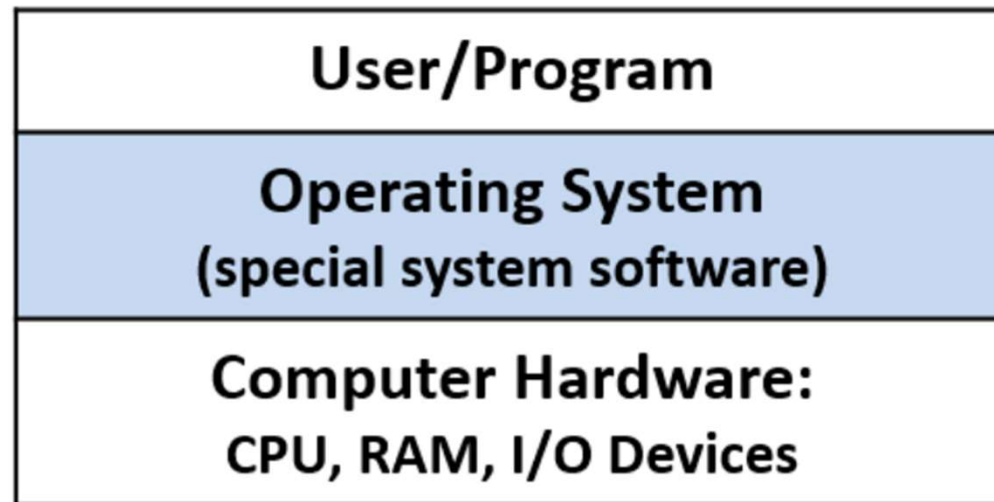
# Beyond OS to real systems

- Architecture + OS: Basic foundation to understand how a user program runs on a single machine
- Networking: How programs talk to each other across machines
- Databases and data storage: How applications store data efficiently and reliably in one or more machines
- Performance engineering: how to make programs run faster
- Distributed systems: How multiple applications across multiple machines work together to perform a useful task reliably
- Virtualization, cloud computing, security, ...

# What is an operating system?

- Middleware between user programs and system hardware
  - Not user application software but **system software**
  - Example: Linux, Windows, MacOS
- Manages computer hardware: CPU, main memory, I/O devices (hard disk, network card, mouse, keyboard etc.)
  - User applications do not have to worry about low-level hardware details
- Operating system has **kernel** + other extra useful software
  - Kernel = the core functionality of the OS
  - Other useful programs = shell, commands on shell, other utilities that help users interact with the OS

# What is an operating system?



*Figure 1. The OS is special system software between the user and the hardware. It manages the computer's hardware and implements abstractions to make the hardware easier to use.*

# History of operating systems

- Started out as a library to provide common functionality to access hardware, invoked via function calls from user program
  - **Convenient** to use OS instead of each user writing code to manage hardware
  - Centralized management of hardware resources is more **efficient**
- Later, computers evolved from running a single program to multiple processes concurrently
  - Multiple untrusted users must share same hardware
- So OS evolved to become trusted system software providing **isolation** between users, and protecting hardware
  - Multiple users are isolated and protected from each other
  - System hardware and software is protected from unauthorized access by users



# Hardware organization

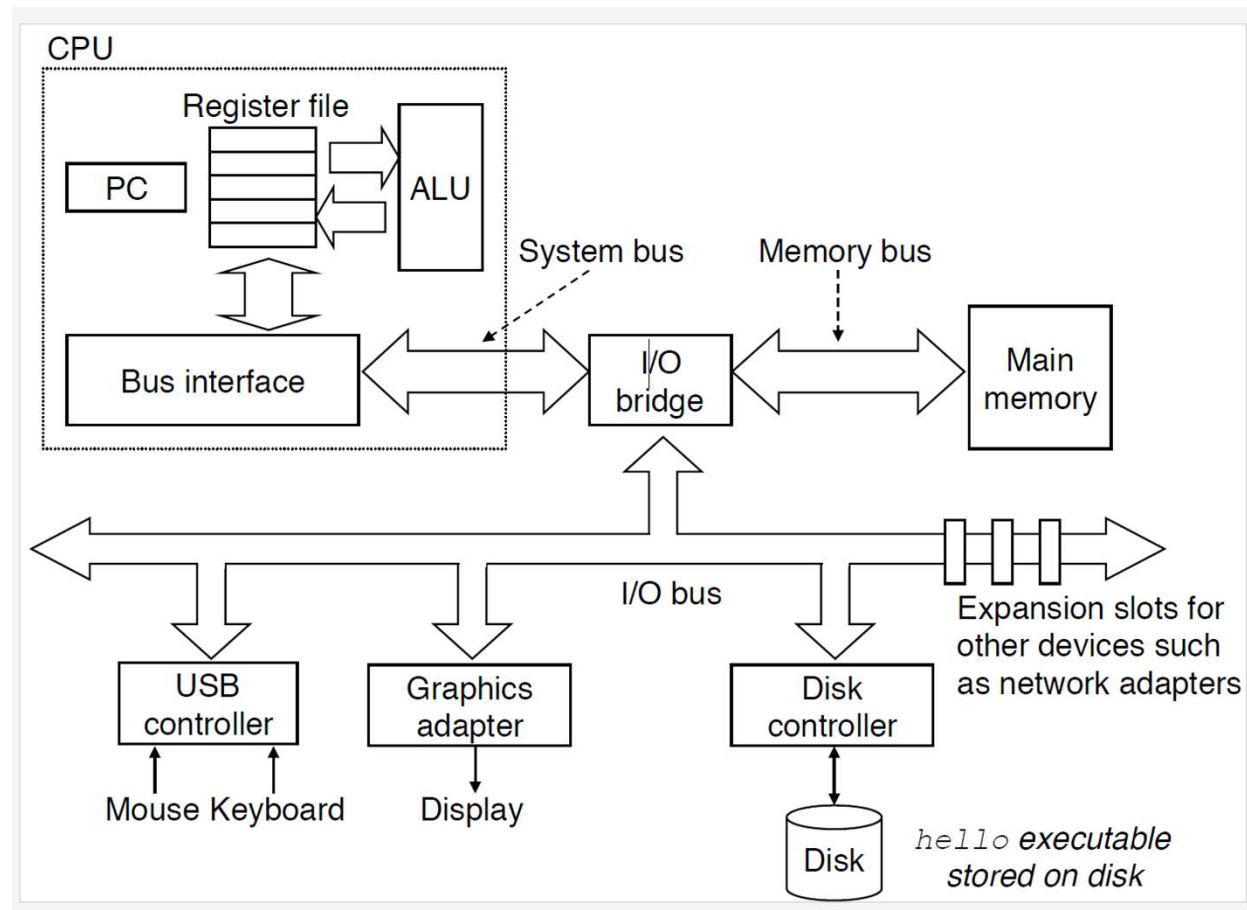


Image credit: CSAPP

# What is a program?

- User **program** = code (instructions for CPU) + data
- Stored program concept
  - User programs stored in main memory or **Random Access Memory (RAM)**
  - Instructions/data occupy multiple contiguous bytes in memory
  - Memory is **byte-addressable**: data accessed via memory address / location / byte#
  - CPU fetches code/data from RAM using memory address, and executes instructions
- CPU runs **processes = running programs**
- Modern CPUs have multiple **cores** for parallel execution
  - Each core runs one process at a time each
  - Modern CPUs have hyper-threading, where each core can run more than one process also (OS treats hyper-threading cores also as multiple CPU cores)

# CPU ISA

- Every CPU has
  - A set of **instructions** that the hardware can execute
  - A set of **registers** for temporary storage of data within the CPU
- High level language (C code) translated into CPU instructions by compiler
  - Can directly write machine code, but cumbersome
- Instructions and registers defined by **ISA** = Instruction Set Architecture
  - Specific to CPU manufacturer (e.g., Intel CPUs follow x86 ISA)
- Registers: special registers (specific purpose) or general purpose
  - **Program counter (PC)** is special register, has memory address of the next instruction to execute on the CPU
  - General purpose registers can be used for anything, e.g., operands in instructions
- Size of registers defined by architecture (32 bit / 64 bit)

# CPU instructions

- Some common examples of CPU instructions
  - **Load**: copy content from memory location → register
  - **Store**: copy content from register → memory location
  - **Arithmetic** and **logical** operations like add:  $\text{reg1} + \text{reg2} \rightarrow \text{reg3}$ , compare, ..
  - **Jump**: change value of PC
- Simple model of CPU
  - Each clock cycle, **fetch** instruction at PC, **decode**, access required data, **execute**, update PC, repeat
  - PC increments to next instruction, or jumps to some other value
- Many optimizations to this simple model
  - **Pipelining**: run multiple instructions concurrently in a pipeline
  - Many more in modern CPUs to optimize #instructions executed per clock cycle

# Memory hierarchy

- Hierarchy of storage elements which store instructions and data
  - CPU registers (small number, accessed in <1 nanosec)
  - Multiple levels of CPU caches (few MB, 1-10 nanosec)
  - Main memory or RAM (few GB, ~100 nanosec)
  - Hard disk (few TB, ~1 millisec)
- Hard disk is non-volatile storage, rest are volatile
  - Hard disk stores files and other data persistently
- As you go down the hierarchy, memory access technology becomes cheaper, slower, less expensive
- CPU caches **transparent** to OS, managed by hardware
  - Software only accesses memory, doesn't know if served from cache or DRAM

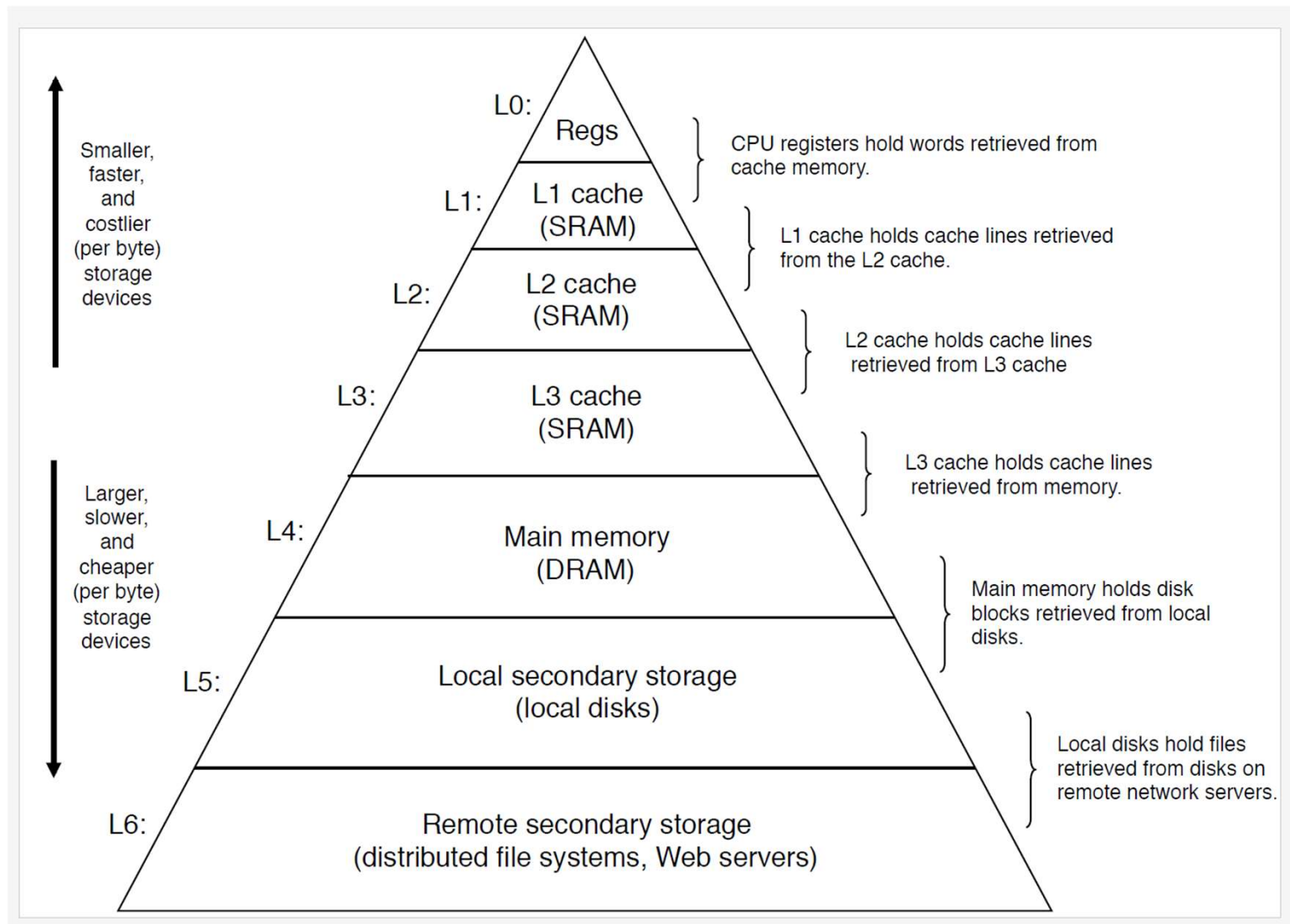
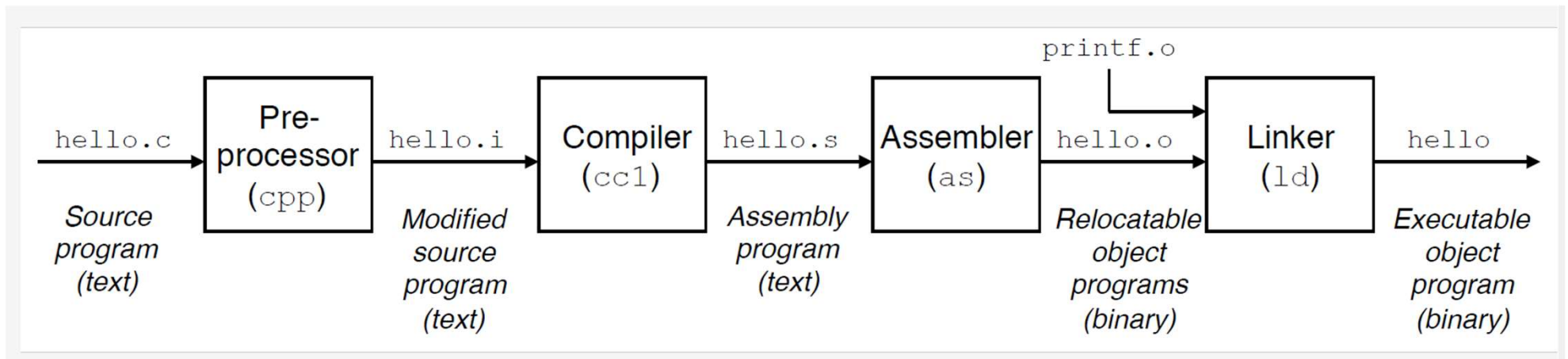


Image credit: CSAPP

# Running a program

- What happens when you run a C program?
  - C code translated into executable by compiler



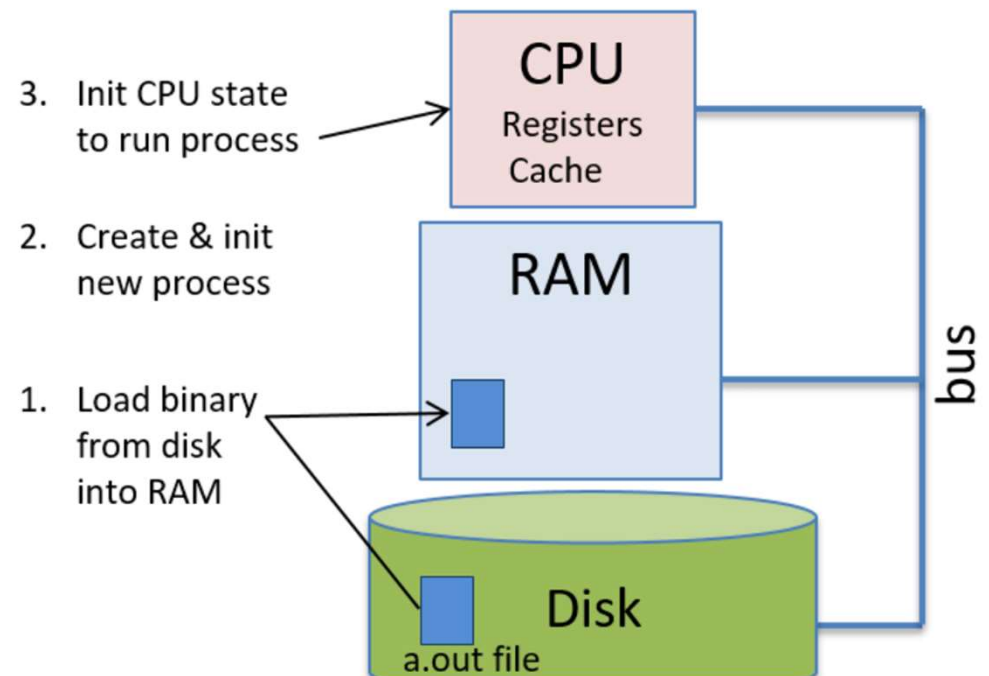
# Running a program

- What happens when you run a C program?
  - C code translated into executable by compiler
  - Executable file stored on hard disk (say, “a.out”)
  - When executable is run, a new **process** is created
  - Process allocated space in RAM to store code and data
  - CPU starts executing the instructions of the program
- When CPU is running a process, CPU registers contain the execution **context** of the process
  - PC points to instruction in the program, general purpose registers store data in the program, and so on



# Role of OS in running a process

- Allocates memory for new process in RAM
  - Loads code, data from disk
- Initializes CPU context
  - PC points to first instruction
- Process starts to run
  - OS steps in as needed



# xv6: a simple teaching OS

- xv6 is a simple OS for easy teaching of OS concepts
  - Two versions, one for x86 hardware and one for RISC-V hardware
  - This series of lectures based on x86 version
  - <https://github.com/mit-pdos/xv6-public>
- Easy to read code, simple exercises to write OS code
- Much simpler than real OS like Linux, but basic concepts remain the same
- Runs inside QEMU emulator

# OS code in C or assembly?

- OS is also like any other program run by CPU, but it is the most important program that manages other programs
  - OS code mostly written in a high-level language like C, compiled into executable, loaded at boot time
- But some parts of OS are written directly in assembly language or CPU instructions that the hardware can understand?
  - Why not write everything in C? Not possible to express certain low level actions performed by OS in high level language
- Basic understanding of x86 assembly code required for understanding xv6 OS code in this course

# Learn how to use xv6

- xv6 source code is available online
  - xv6 kernel code
  - User programs to test OS functionality, e.g., simple shell programs like ls
- Instructions have been provided for you to learn how to:
  - Set up QEMU and other software needed to run xv6 (on your personal laptops; the lab machines should have all of this)
  - Compile and run xv6, for example, execute simple shell commands like “ls” in the xv6 shell
  - Add your own simple “command” to the xv6 code, make necessary changes to the code and Makefile, and rerun xv6 again

## Reference: x86 registers

- General purpose registers: store data during computations (eax, ebx, ecx, edx, esi, edi)
- Pointers to stack locations: base of stack (ebp) and top of stack (esp)
- Program counter or instruction pointer (eip): next instruction to execute
- Control registers: hold control information or metadata of a process (e.g., cr3 has information related to memory of process)
- Segment registers (cs, ds, es, fs, gs, ss): information about segments (related to memory of process)

## Reference: x86 instructions

- Load/store: *mov src, dst*
  - *mov %eax, %ebx* (copy contents of eax to ebx)
  - *mov (%eax), %ebx* (copy contents at the address in eax into ebx)
  - *mov 4(%eax), %ebx* (copy contents stored at offset of 4 bytes from address stored at eax into ebx)
- Push/pop on stack: changes esp
  - *push %eax* (push contents of eax onto stack, update esp)
  - *pop %eax* (pop top of stack onto eax, update esp)
- *jmp* sets eip to specified address
- *call* to invoke a function, *ret* to return from a function
- Variants of above (*movw, pushl*) for different register sizes

# Reference: Mechanics of function call

- Local variables, arguments, return address stored on stack for duration of function call
- What happens in a function call?
  - Push function arguments on stack
  - *call fn* (instruction pushes return address on stack, jumps to function)
  - Allocate local variables on stack
  - Run function code
  - *ret* (instruction pops return address, eip goes back to old value)
- All of this is automatically done by the C compiler for you, and is part of the C calling convention.

## Reference: Caller and callee save registers

- What about values in registers that existed before function call?  
Registers can get clobbered during a function call, so how can computation resume?
  - Some registers saved on stack by caller before invoking the function (caller save registers). Function code (callee) can freely change them, caller restores them later on.
  - Some registers saved by callee function and restored after function ends (callee save registers). **Caller expects them to have same value on return.**
  - Return value stored in eax register by callee (one of caller save registers)
- All of this is automatically done by C compiler (C calling convention)



# Reference: More details of function call

- Anatomy of a function call
  - Push caller save registers (eax, ecx, edx)
  - Push arguments in reverse order
  - Return address (old eip) pushed on stack by the call instruction
  - Push old ebp on stack
  - Set ebp to current top of stack (base of new “stack frame” of the function)
  - Push local variables and callee save registers (ebx, esi, edi)
  - Execute function code
  - Pop stack frame and restore old ebp
  - Return address popped and eip restored by the ret instruction
- Stack pointers: ebp stores address of base of current stack frame and esp stores address of current top of stack
  - Function arguments are accessible from looking under the stack base pointer