Introduction to Operating Systems (Background)

Mythili Vutukuru

CSE, IIT Bombay

What is a computer system?

- Software
  - User programs (instructions and data) to accomplish some tasks
  - System software like operating systems
- Hardware
  - CPU (registers, ALU, caches, …)
  - Main memory (DRAM)
  - I/O devices, secondary storage…
- Software written in high-level languages is compiled into binary files (executables) containing instructions that the CPU hardware can execute
- Operating systems written in high-level language like C
  - Please be comfortable with C before proceeding further in this course

Running a program

- What happens when you run a C program?
  - C code translated into executable by compiler in multiple steps (see below)
  - Executable loaded from disk to main memory when program starts
  - CPU fetches program instructions from main memory and executes them

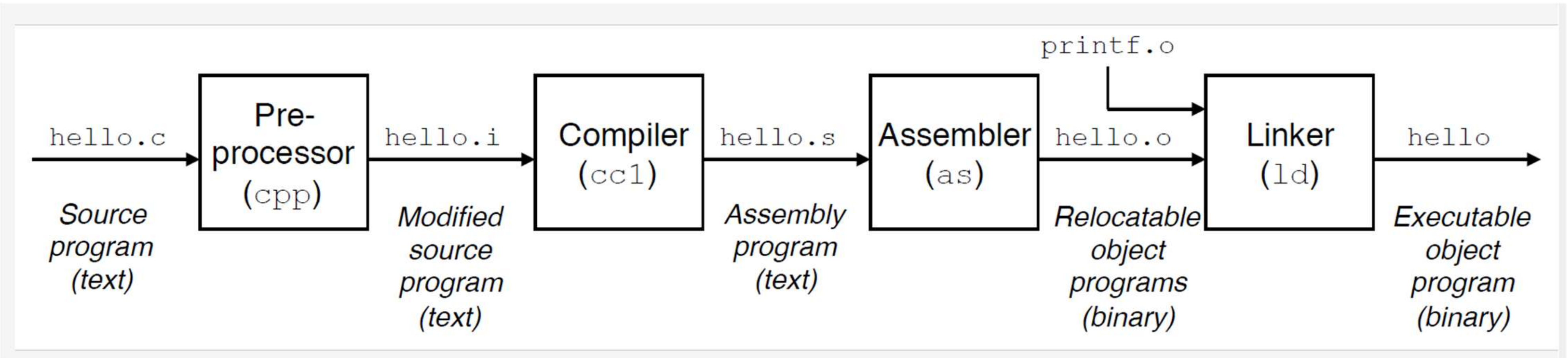
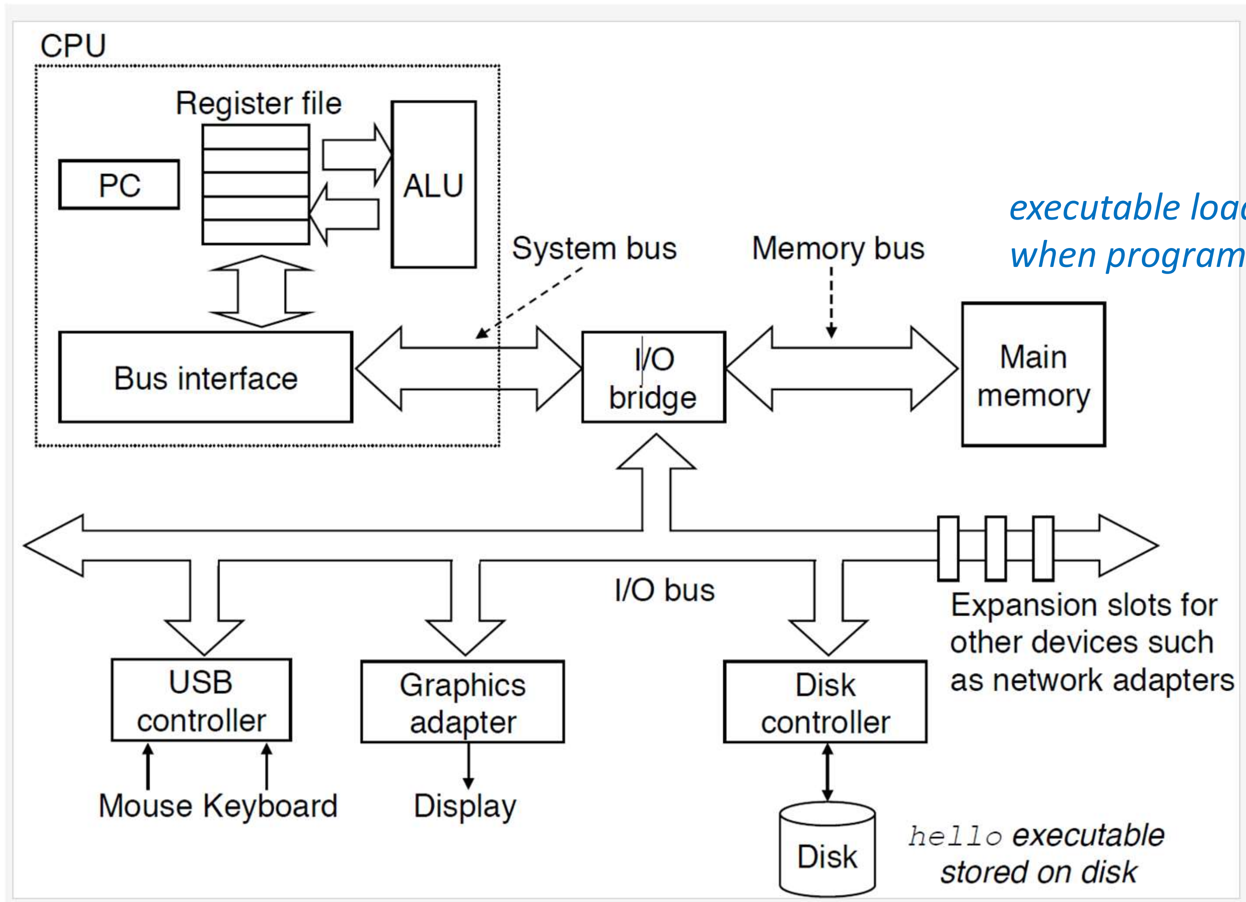


Image credit: CSAPP

Hardware Organization



executable loaded in memory when program runs

Image credit: CSAPP

CPU ISA

- Every CPU has a well-defined set of
  - Instructions that the hardware can execute
  - Registers for temporary storage of data within the CPU
- Instructions and registers specified 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: reg1 + reg2 → reg3, compare, ..
  - Jump: change value of PC
  - Call: invoke a function
- 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/storage hierarchy

- Program executable loaded from secondary storage to main memory
- When CPU runs program, recently accessed instructions and data stored in CPU caches (faster access than DRAM)
- Registers in CPU provide temporary storage, e.g., hold operands

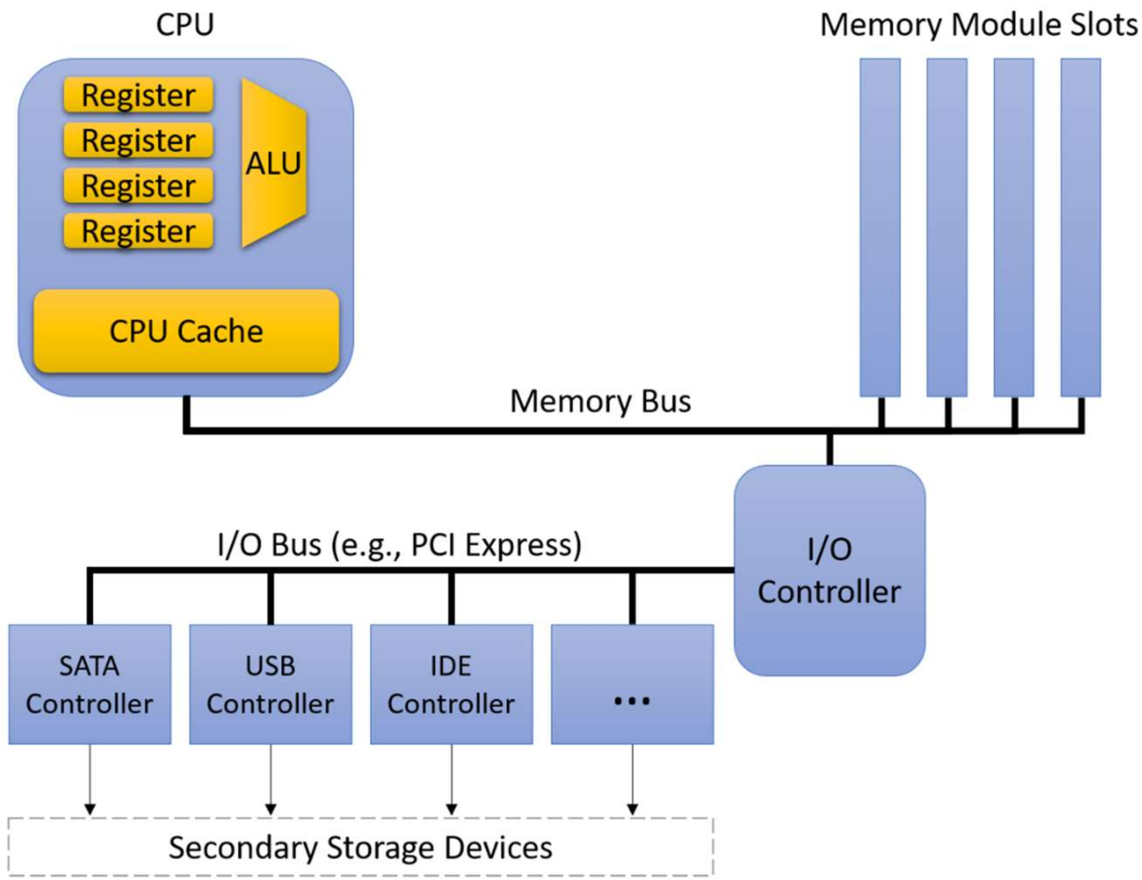


Image credit: Dive Into Systems

Memory/storage 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 software, managed by hardware
  - Software only accesses memory, doesn’t know if served from cache or DRAM

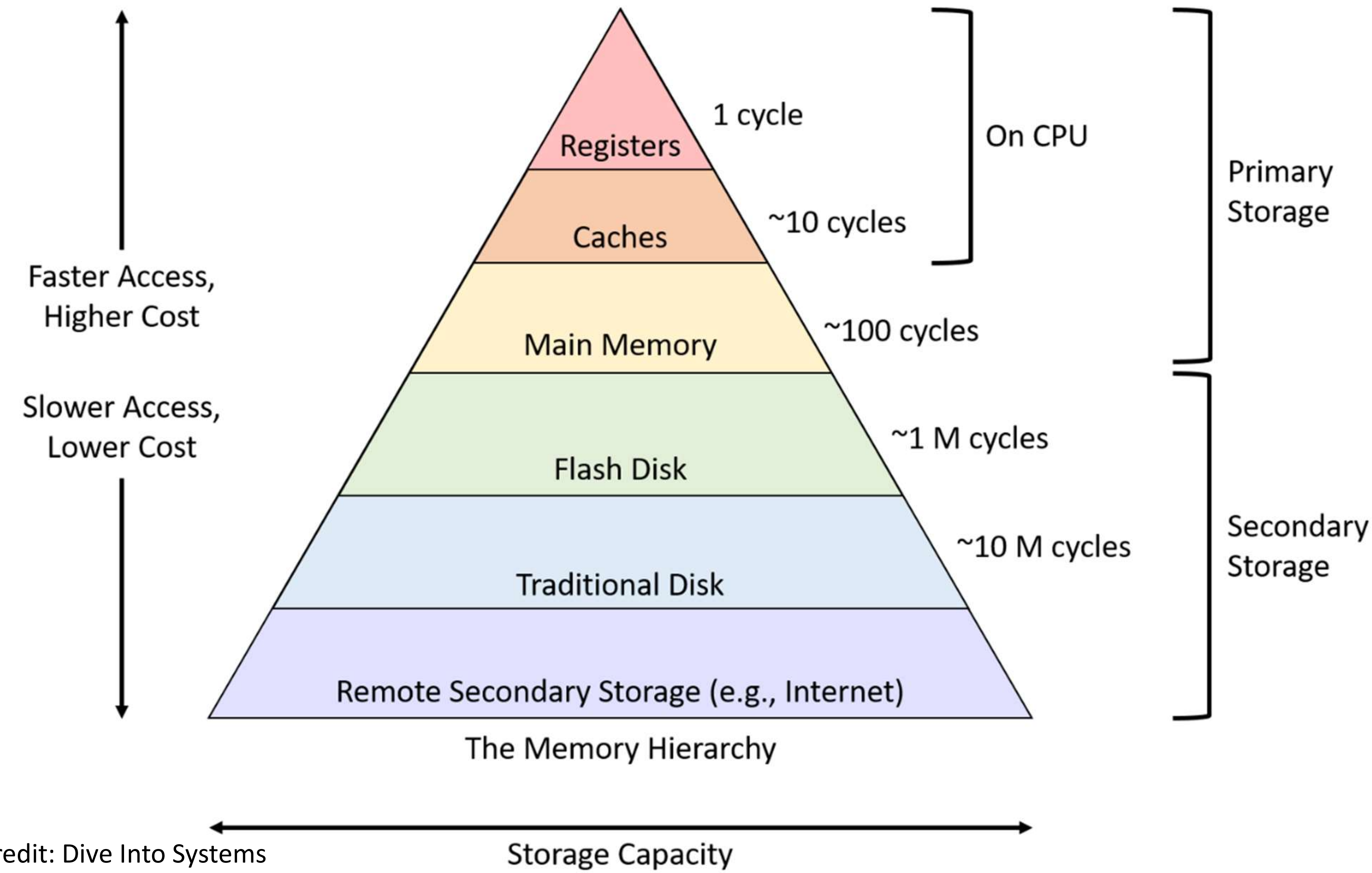


Image credit: Dive Into Systems

Parts of program memory

- The memory of a running program in DRAM has the following components
  - Compiled code (instructions)
  - Compile-time data (global/static variables)
  - Runtime data on stack (function arguments, local variables, …)
  - Runtime data on heap (dynamically allocated memory via malloc, …)
- All instructions and data are assigned memory addresses, based on their location in memory
- Main memory contains user programs + code/data of OS

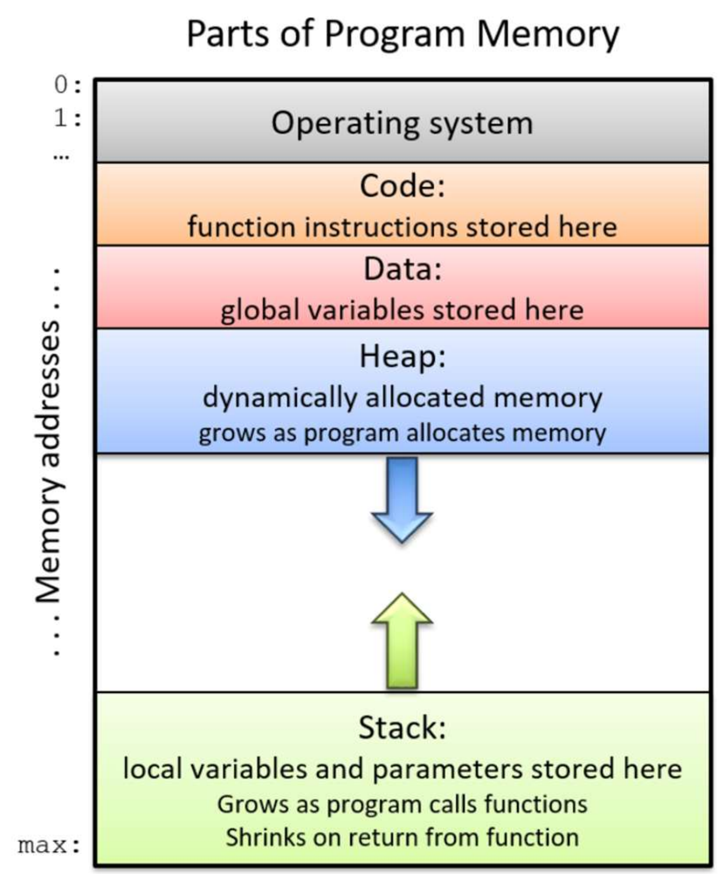


Image credit: Dive Into Systems

Example: memory allocation

- When is memory allocated for the various parts of this program?
- Memory for global variable "g" allocated when executable loaded into memory at start of execution
- Memory for function arguments and local variables (a, b, x, y, z, …) allocated ("pushed") on stack when the corresponding function is called
  - Why not allocate memory at start of program? Because we do not know if/how many times the function will be called at runtime
  - Function variables "popped" from stack when function returns
- Memory requested dynamically via malloc is allocated on the heap at runtime, when malloc is invoked

```
int g;

int increment(int a) {
    int b;
    b = a+1;
    return b;
}

main() {
    int x, y;
    x = 1;
    y = increment(x);

    int *z = malloc(40);

}
```

Pointers and addresses

- A pointer variable contains the memory address of another variable
- Note that these addresses are only logical addresses, and not the actual physical addresses in DRAM (why? more later)
- Pointer variables contain space to only store the address, and the variable being pointed to must be declared/allocated separately
- Ensure pointer contains valid address before accessing it

```
/* Assuming an integer named x has already been declared, this code sets the
   value of x to 8. */

ptr = &x;   /* initialize ptr to the address of x (ptr points to variable x) */
*ptr = 8;   /* the memory location ptr points to is assigned 8 */
```

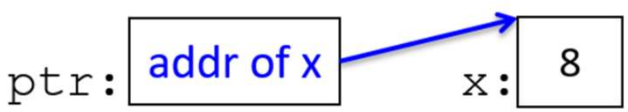


Image credit: Dive Into Systems

Stack vs. heap

- Functions like malloc allocate memory on heap and return start address of allocated chunk
- This heap address is stored in a pointer variable, which may be a local variable in a function, and hence located on the stack
- Dynamically allocated memory on heap must be explicitly freed up (in languages like C), else memory leak
    - Stack memory automatically popped when function returns

```c
int *arr;
char *c_arr;

// allocate an array of 20 ints on the heap:
arr = malloc(sizeof(int) * 20);

// allocate an array of 10 chars on the heap:
c_arr = malloc(sizeof(char) * 10);
```

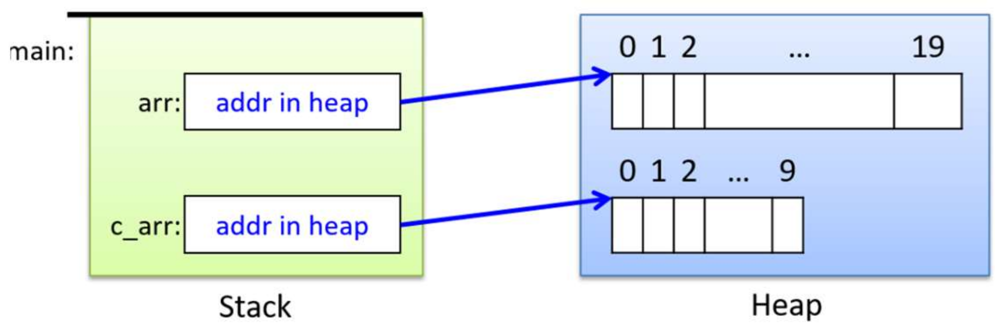


Image credit: Dive Into Systems

What happens on a function call?

- Function arguments allocated on stack (in reverse order, by convention)
- Old PC (return addr) pushed on stack, PC jumps to function code
- Local variables allocated on stack
- Some register context saved too (more later)
- Now, new stack frame is ready on stack
- Function code runs using data on stack
- When function returns, all of the function memory is popped off the stack