# A Tour of Computer Systems

CS-2011: Introduction to Computer Systems (Fall 2022)
Lecture 2

# Outline

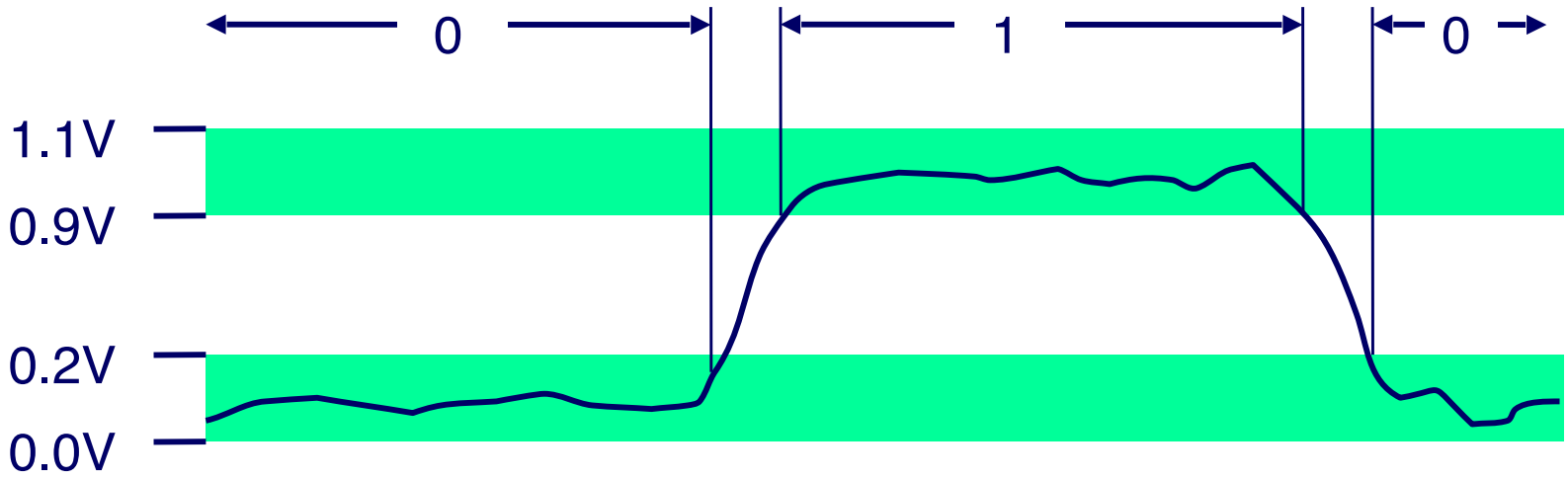
- Information is Bits!
- C Compilation Process
- Hardware Organization
- Running a hello world
- Cache Memory and Memory Hierarchy
- Operating System Role

# Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers (integer or floating-point), sets, strings, etc…
- **Why bits?  Electronic Implementation**

# Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers (integer or floating-point), sets, strings, etc…
- **Why bits? Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



# How is hello.c source file encoded

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

- hello.c source program (a **text file**) is a **sequence of bits**
- Each **8 bits** make **1 Byte** each represents a text **character**
- Most computers represent characters using **ASCII** standard
  - Each byte has an integer value that corresponds to some character
    - e.g., 35 —> #, 105 —> i, etc…
    - Full ASCII table: https://www.asciitable.com/

# How is hello.c source file encoded

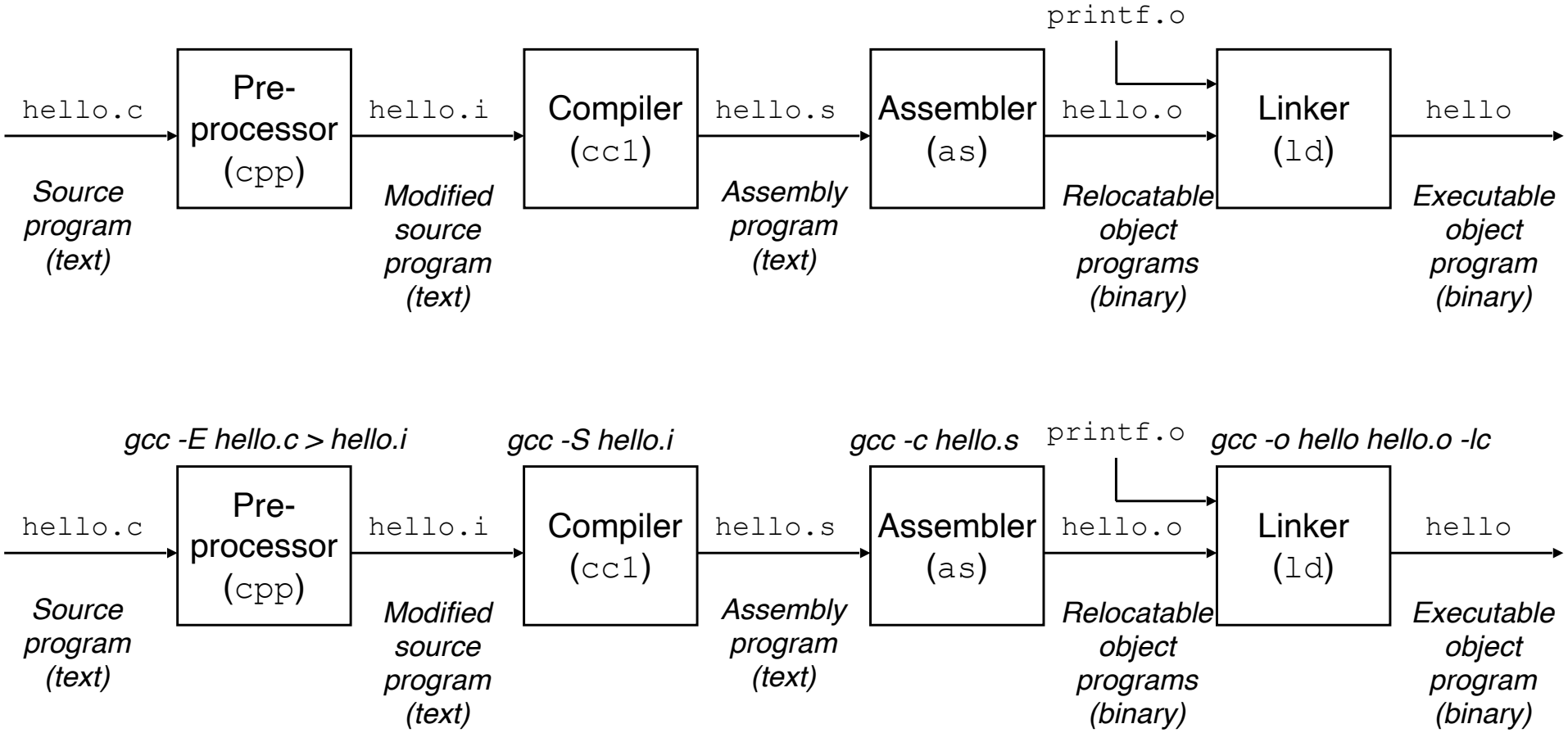
```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

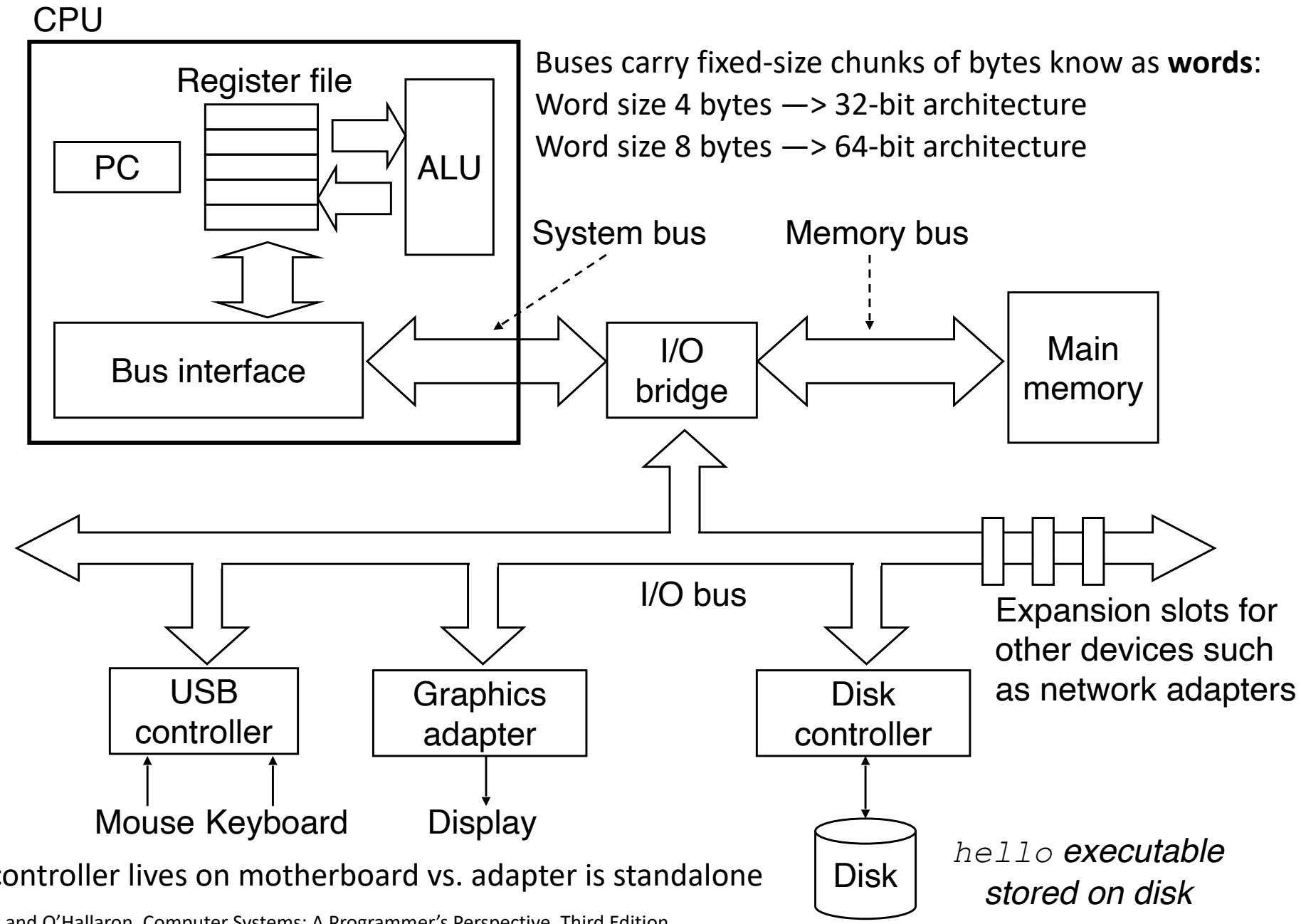
$more hello.c | od -An -vtu1

```
 35 105 110  99 108 117 100 101  32  60 115 116 100 105 111  46
104  62  10  10 105 110 116  32 109  97 105 110  40  41  10 123
 10   9 112 114 105 110 116 102  40  34 104 101 108 108 111  44
 32 119 111 114 108 100  92 110  34  41  59  10   9 114 101 116
117 114 110  32  48  59  10 125  10
```

# C Compilation Process



# Hardware Organization



# Hardware Organization (Memory)

**1 Byte = 8 bits**
**e.g., 11111100**

**Each Byte is addressed using a 32-bit value**

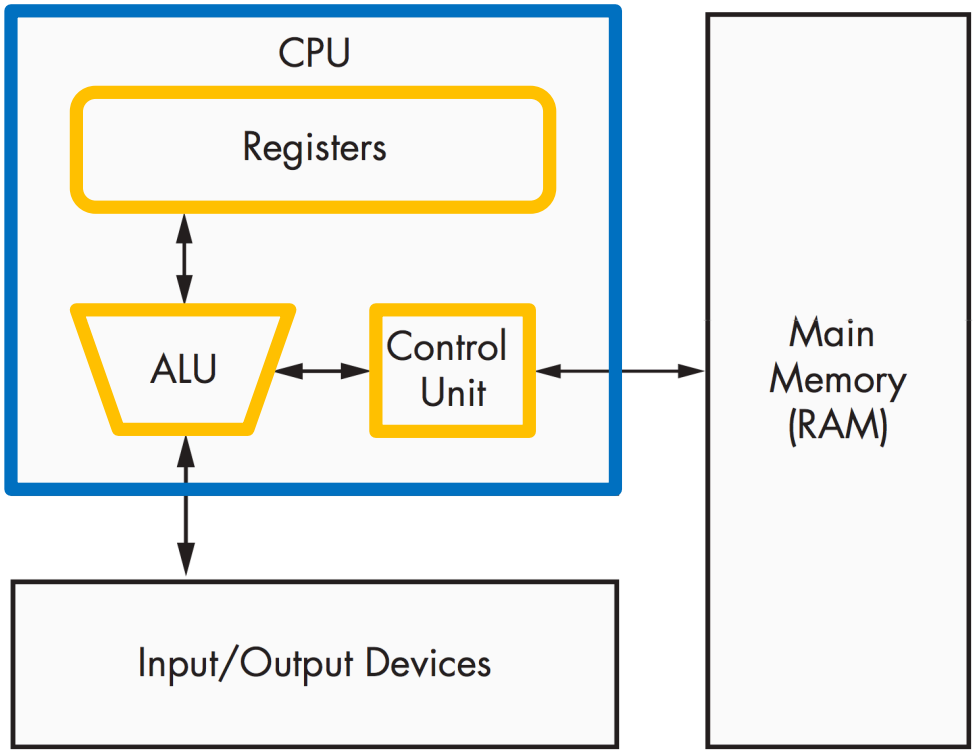
High Memory Address

| | | | | Address |
|---|---|---|---|---|
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F050 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F04C |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F048 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F044 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F040 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F03C |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F038 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F034 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F030 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F02C |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F028 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F024 |
| 1 Byte | 1 Byte | 1 Byte | 1 Byte | 0x0012F020 |

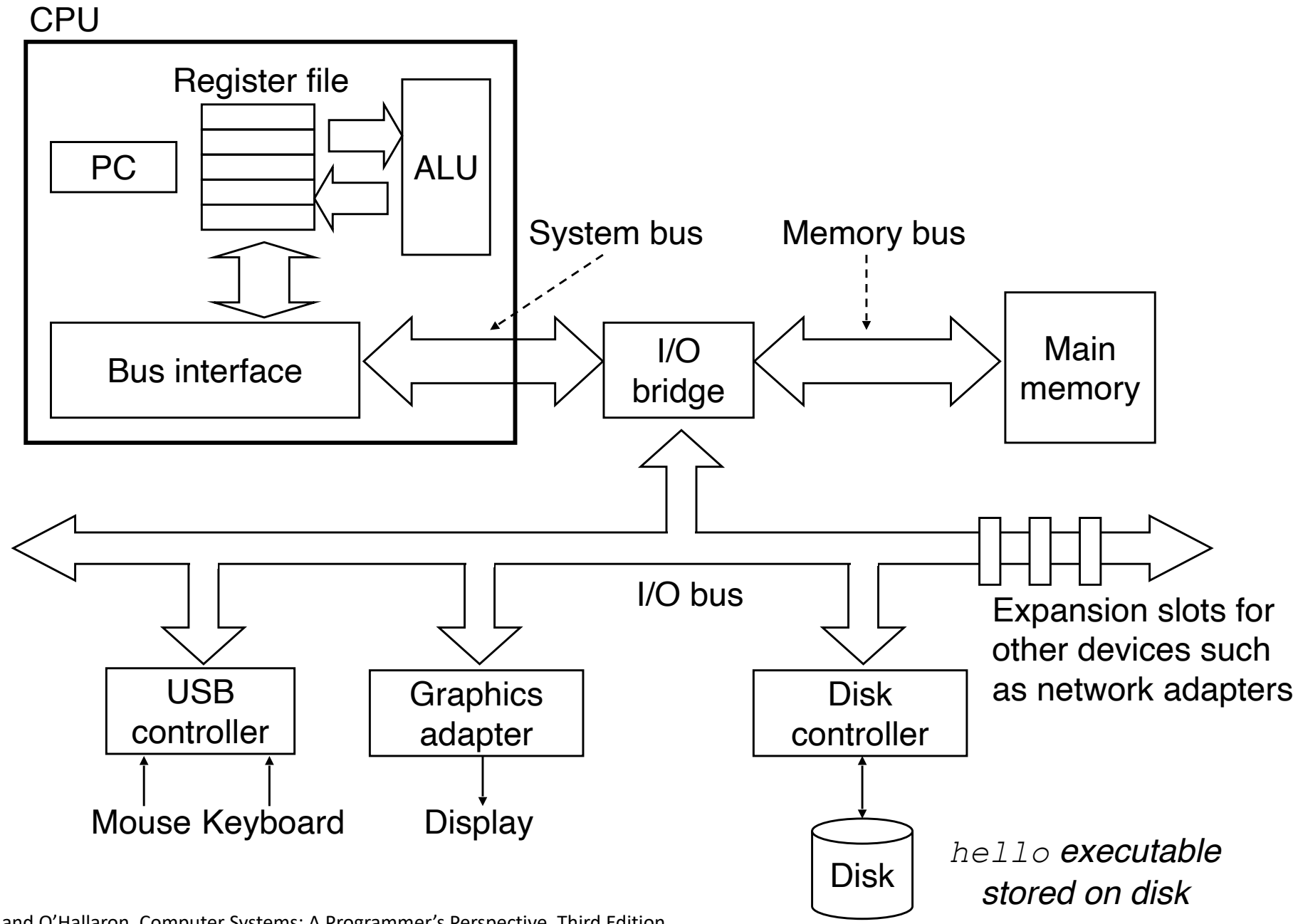
Low Memory Address

# Hardware Organization (Processor)

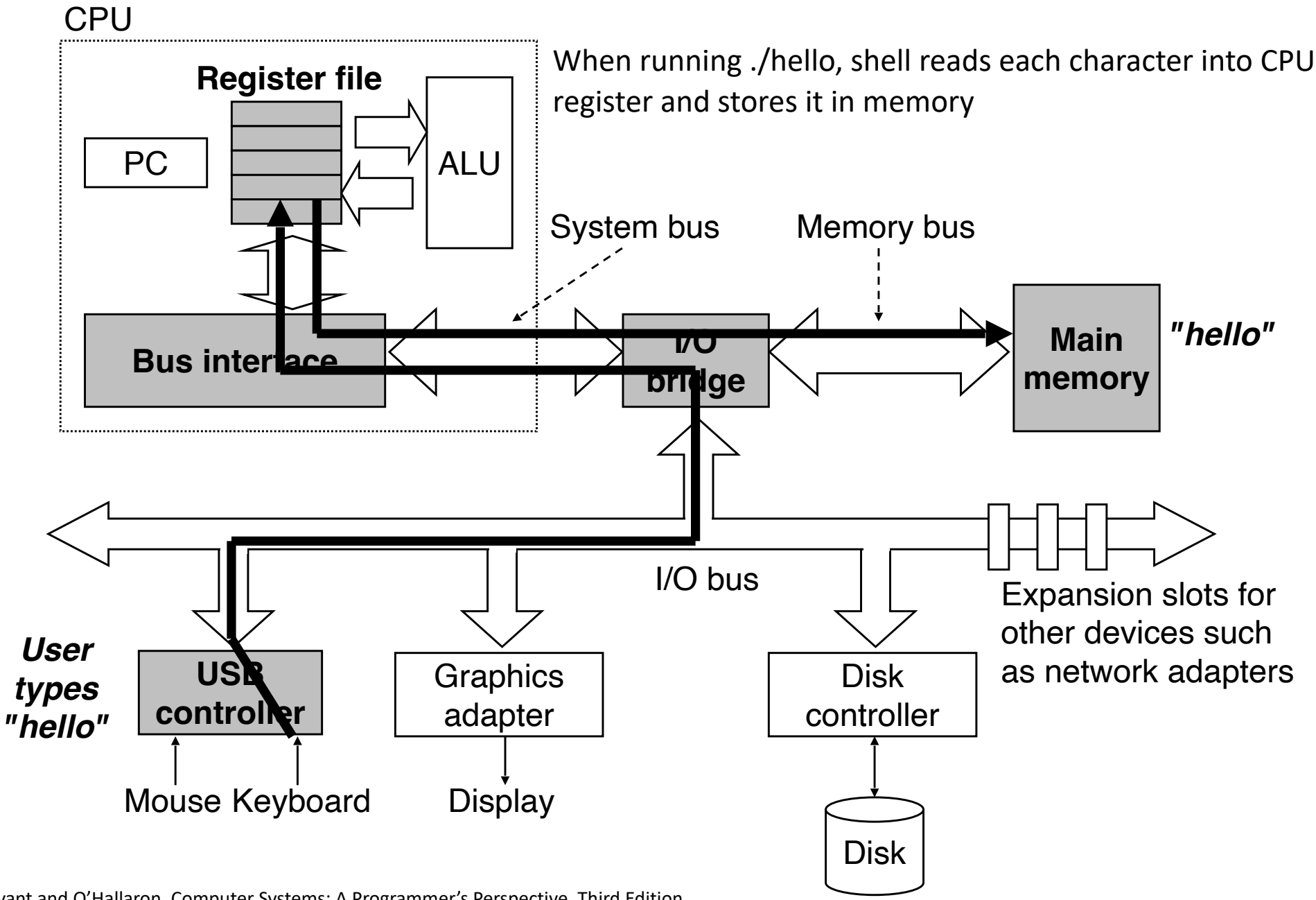
- Control Unit
  - Gets instructions from RAM
    - Uses **instruction pointer** to store address of next instruction
- Registers
  - **Basic data storage** units
  - Used to **avoid RAM access**
    - Saves time
- ALU (**A**rithmetic **L**ogic **U**nit)
  - **Executes** instruction fetched from RAM
  - Places results in registers/RAM
- PC (Program Counter) - Word-size storage device
  - Always runs current instruction and updates pointer to point to next instruction



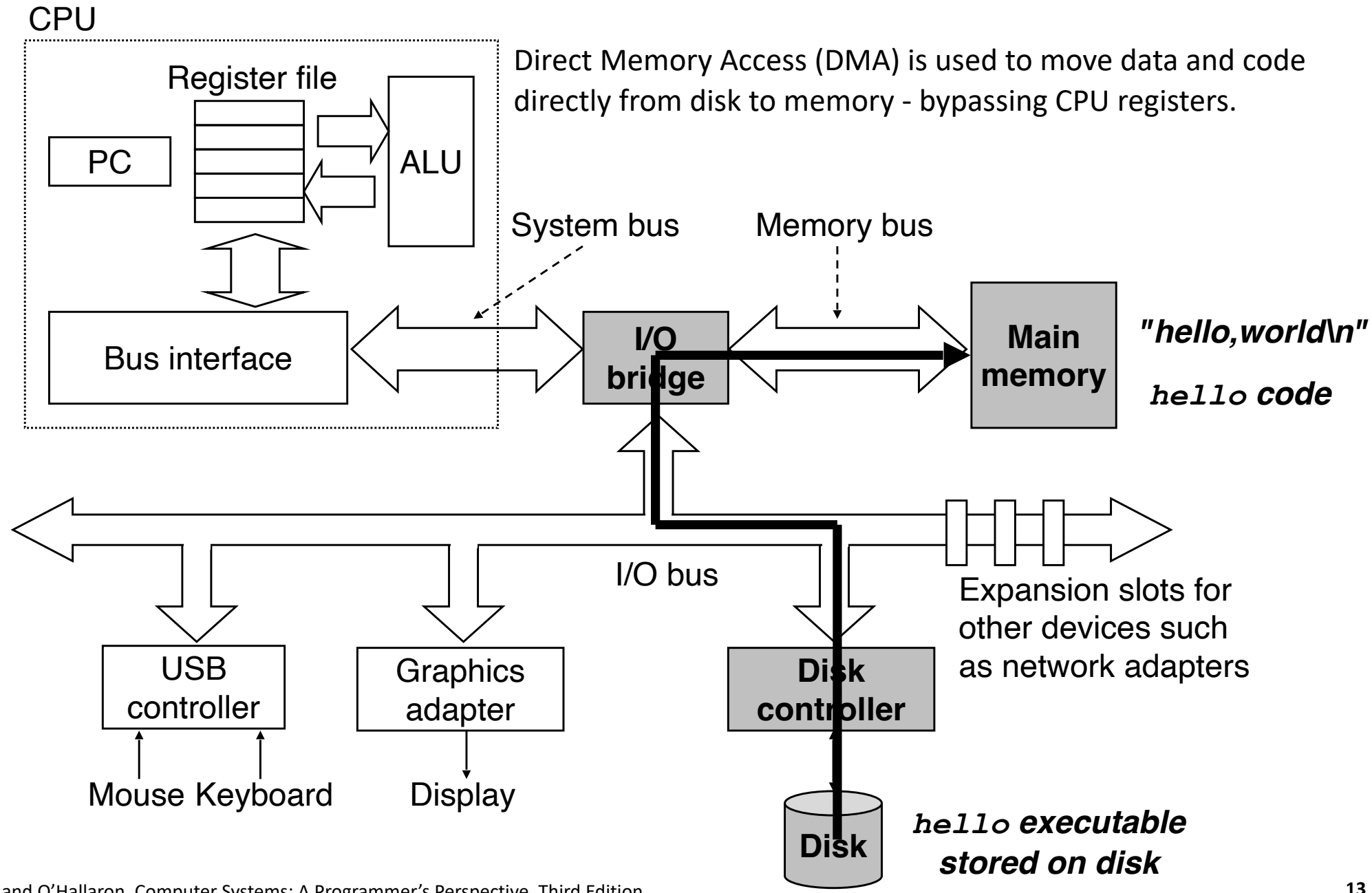
# Running the hello Program



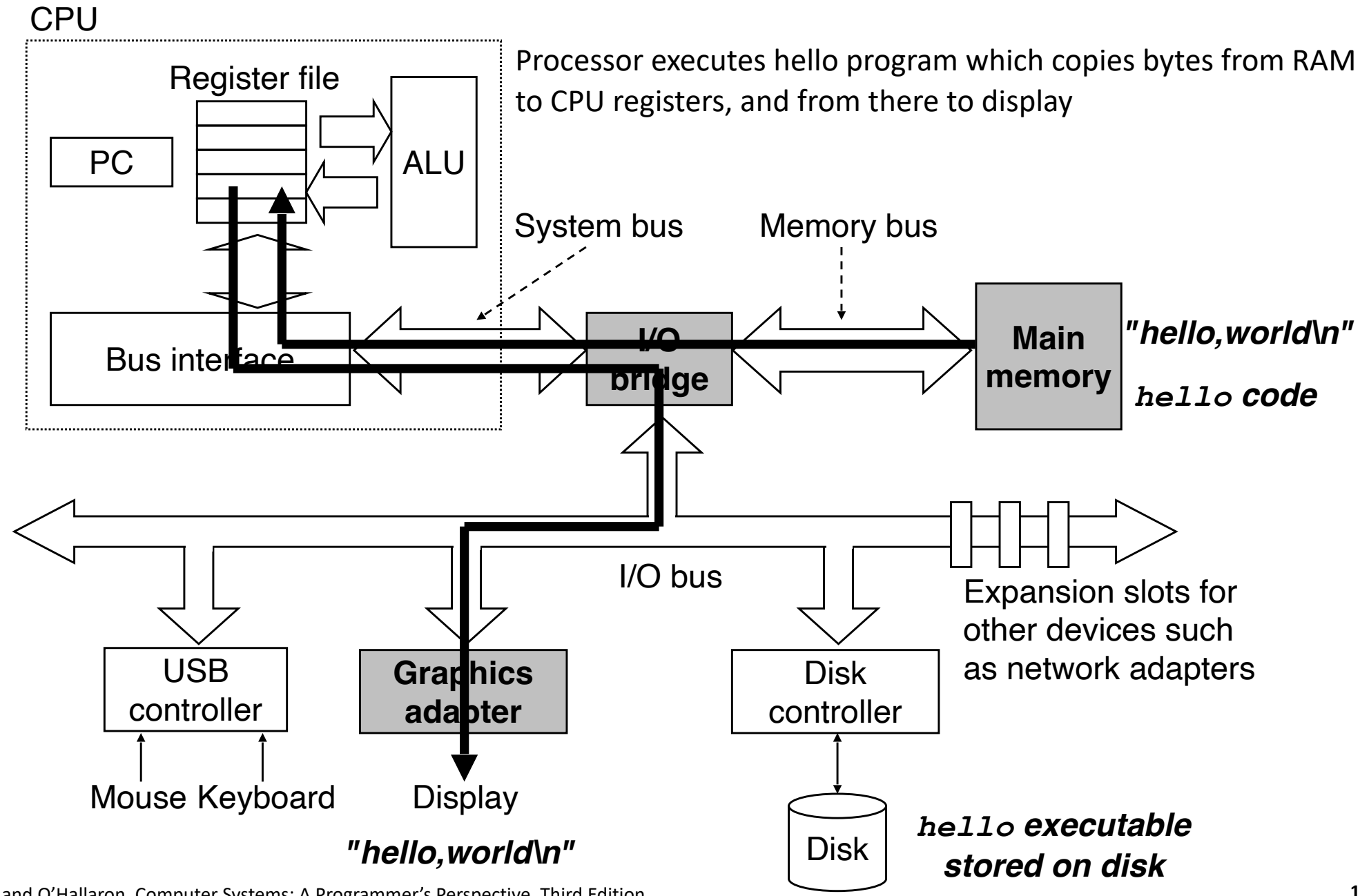
# Running the hello Program



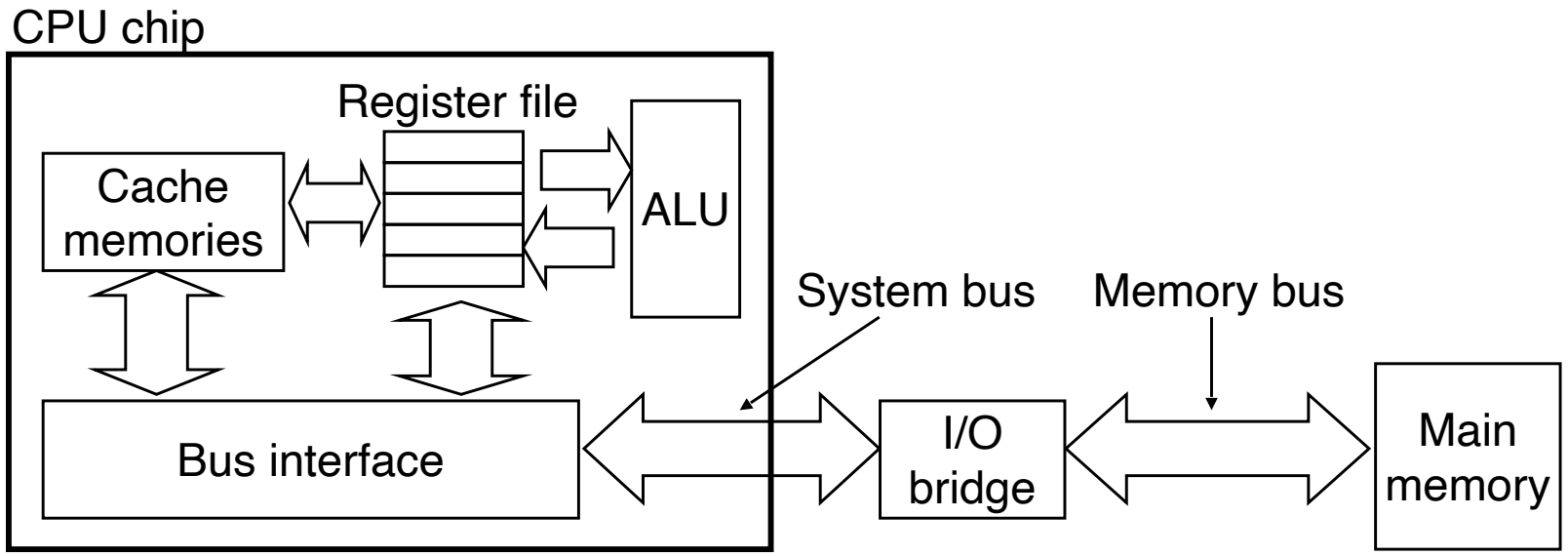
# Loading Executable from Disk into Memory

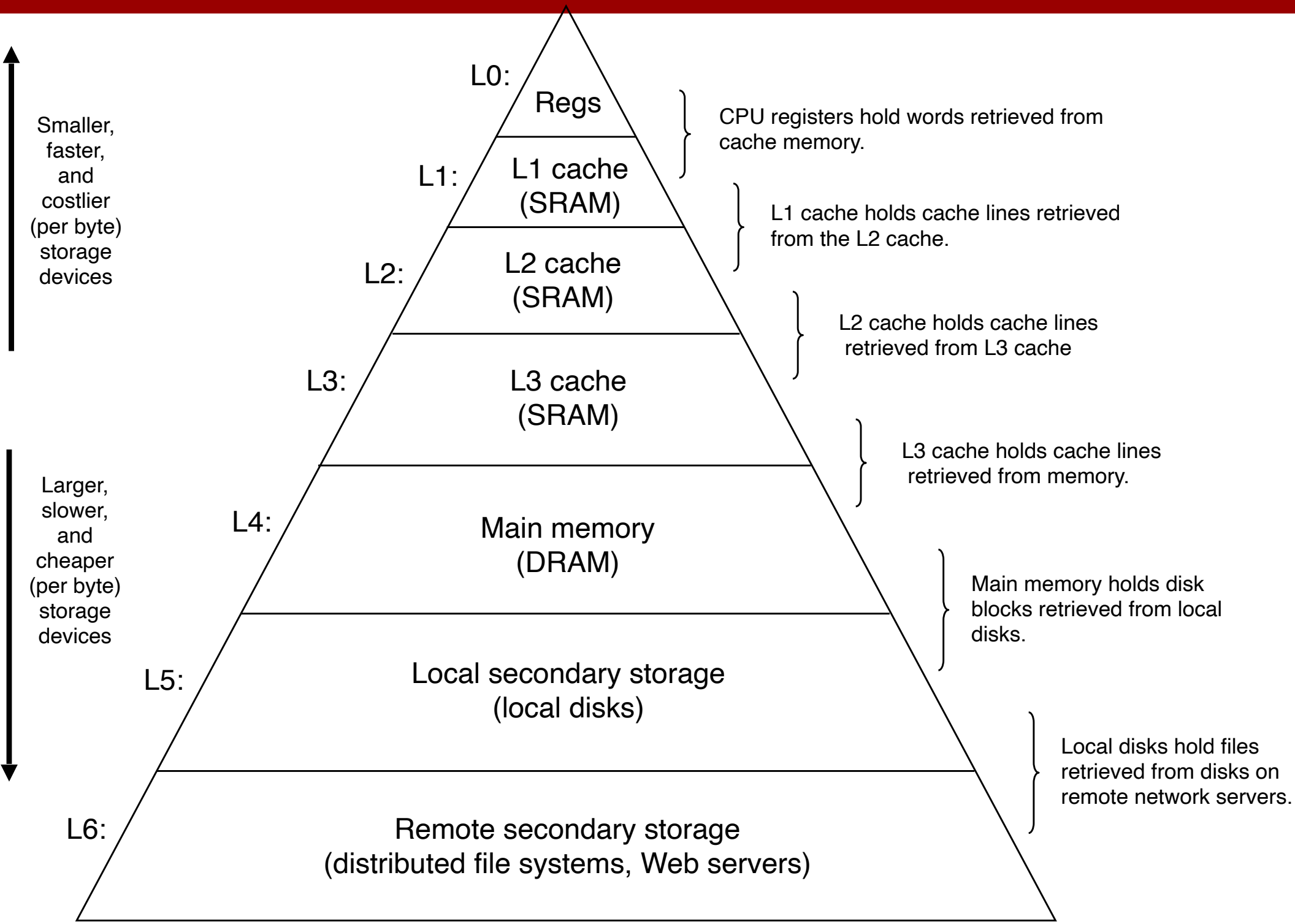


# Writing Output from Memory to Display

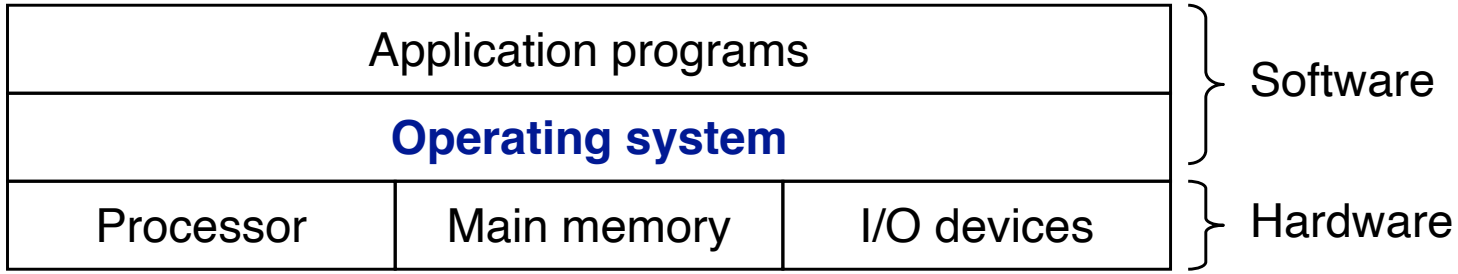


# Cache Memory & Memory Hierarchy

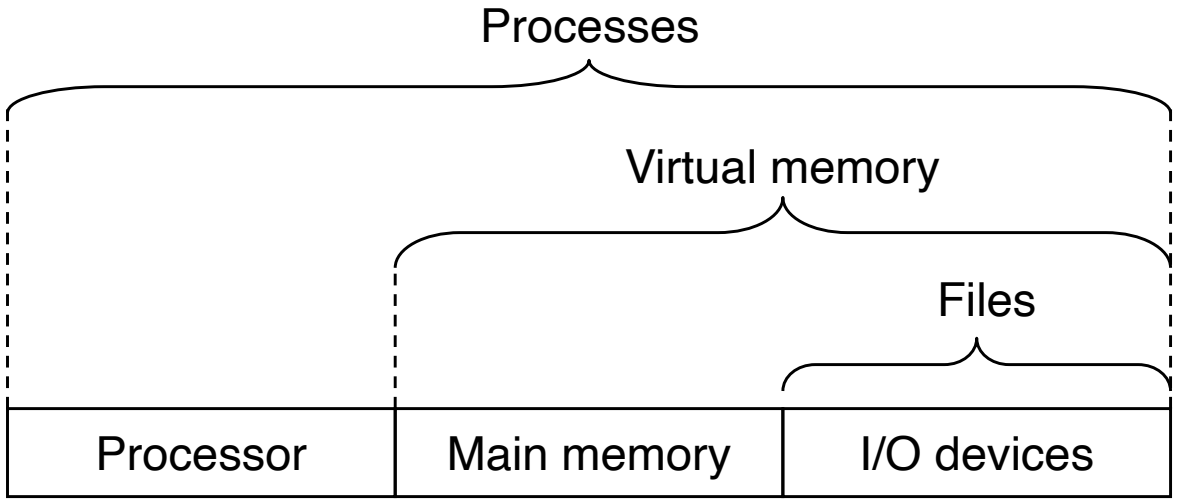



Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

# Operating System Role

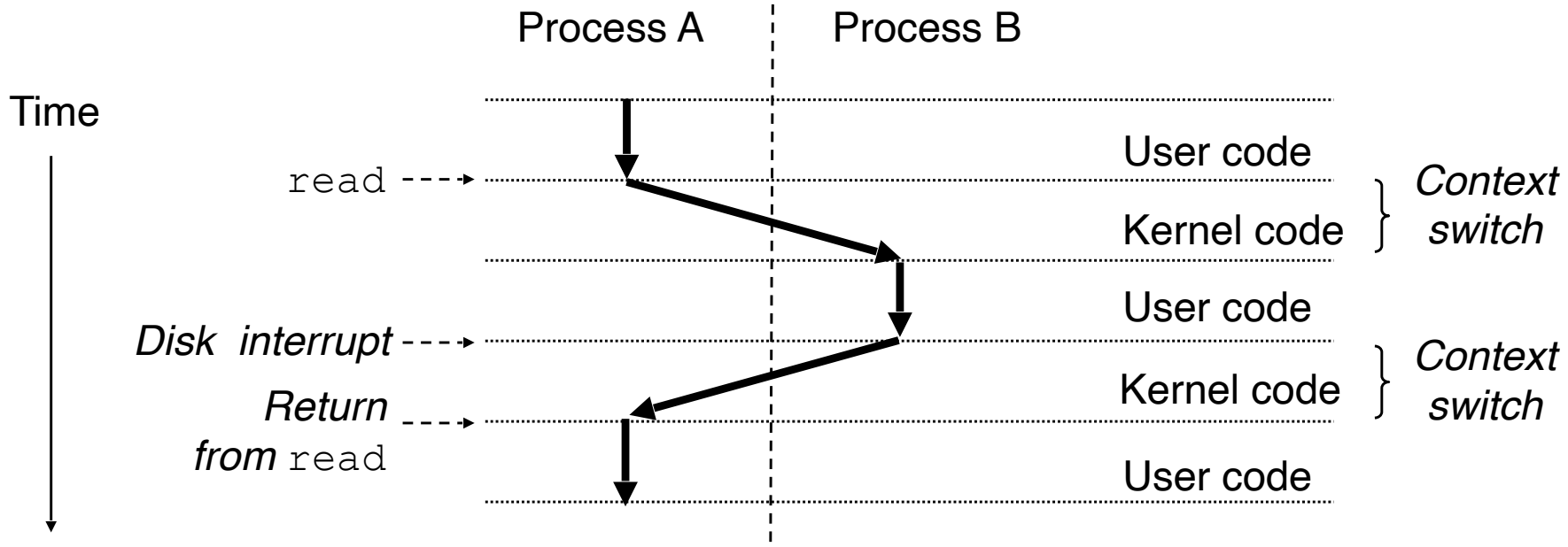



Operating system (OS) manages the hardware




Abstraction Provided by OS

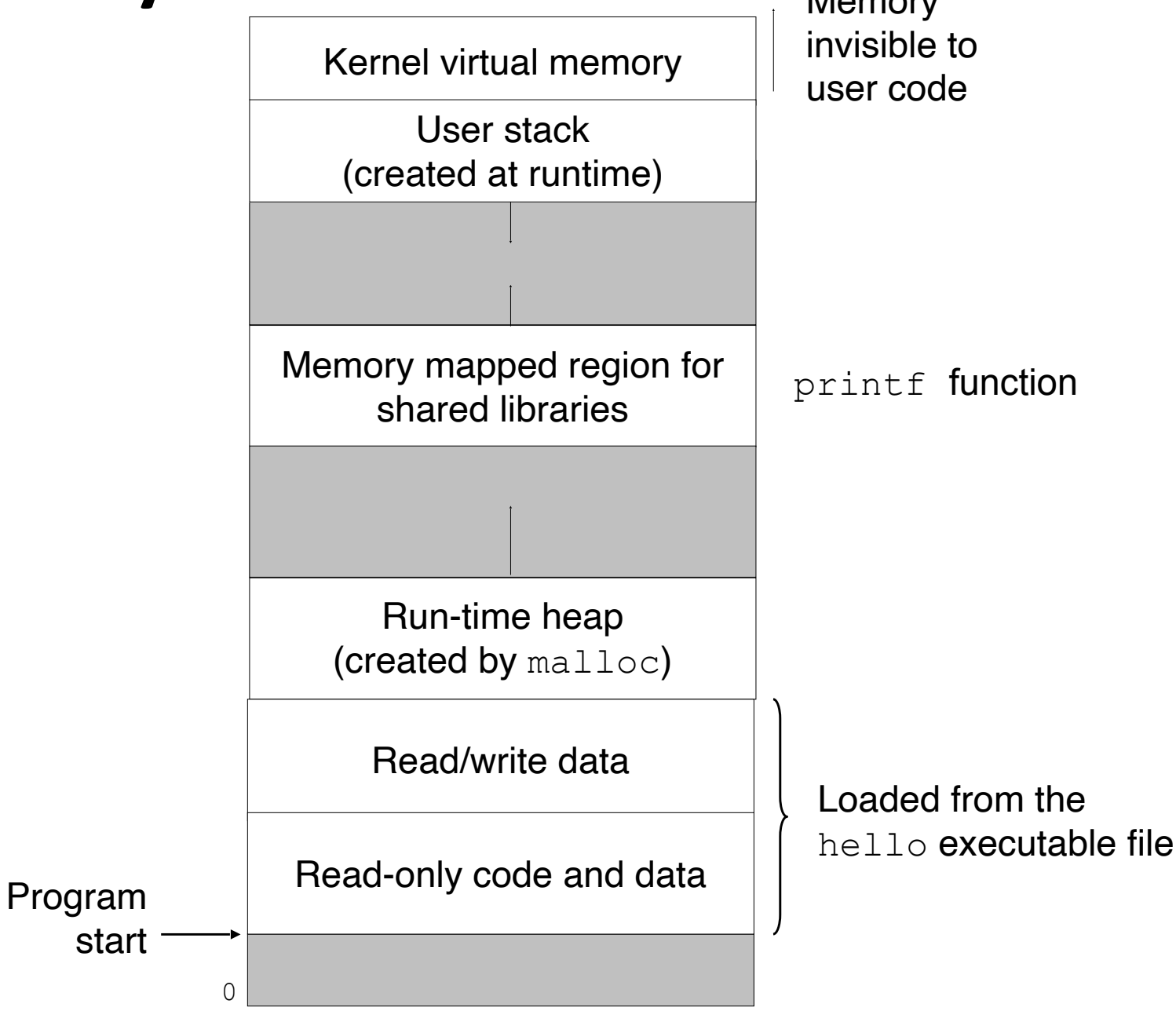
# Processes and Context Switching




E.g., Process A: shell process, Process B: hello program

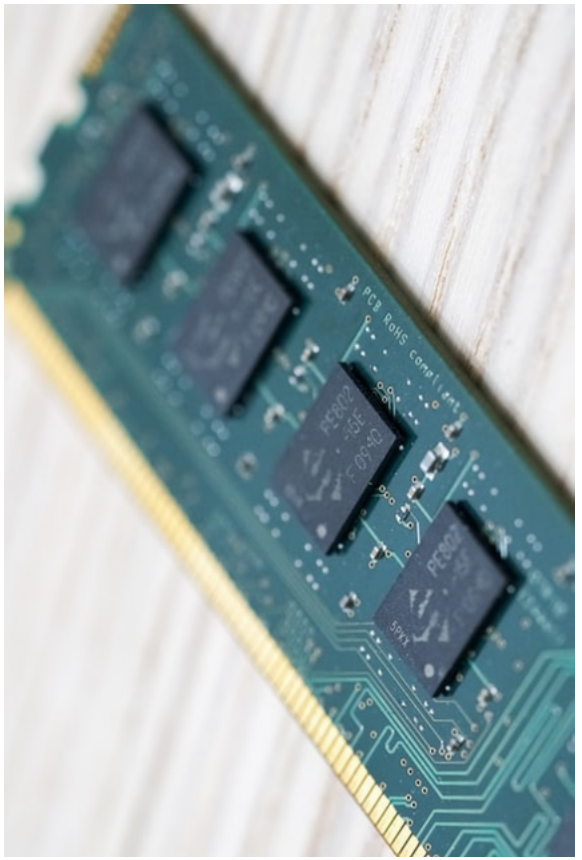
- shell program makes a **system call** into kernel to run hello program, passing control to OS
- OS saves the shell's program context
- OS creates a new hello process and its context, and then passes control to hello process
- After hello process terminates, OS restores context and pass control back to shell program

# Virtual Memory



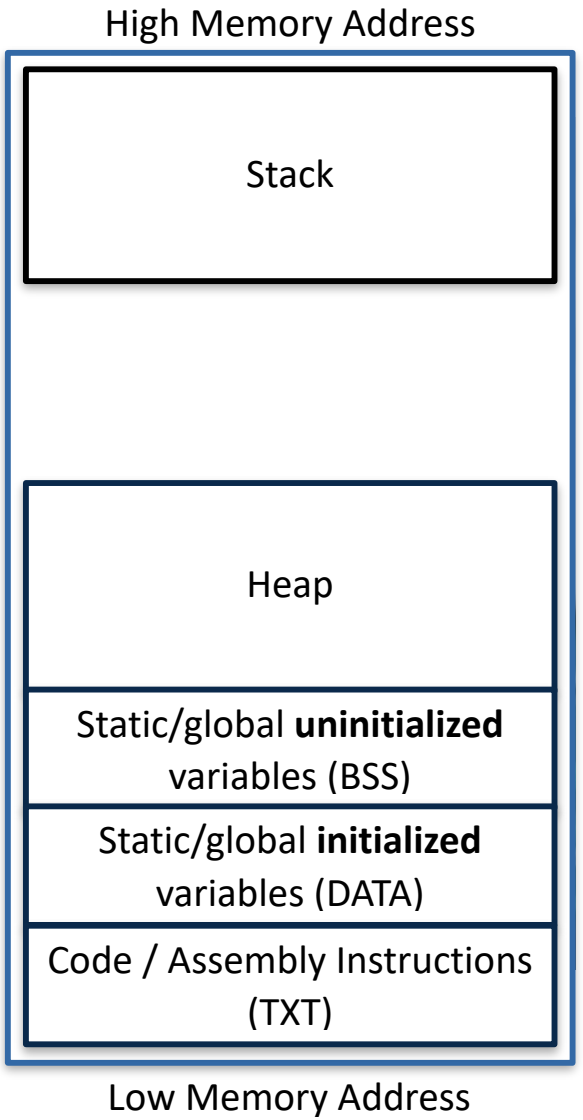
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```



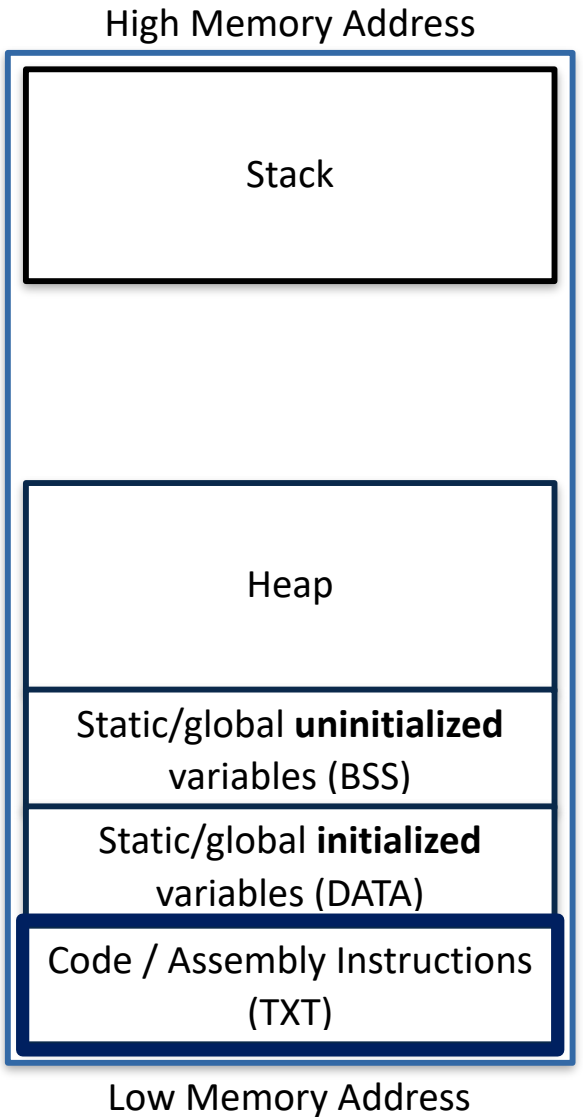
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```



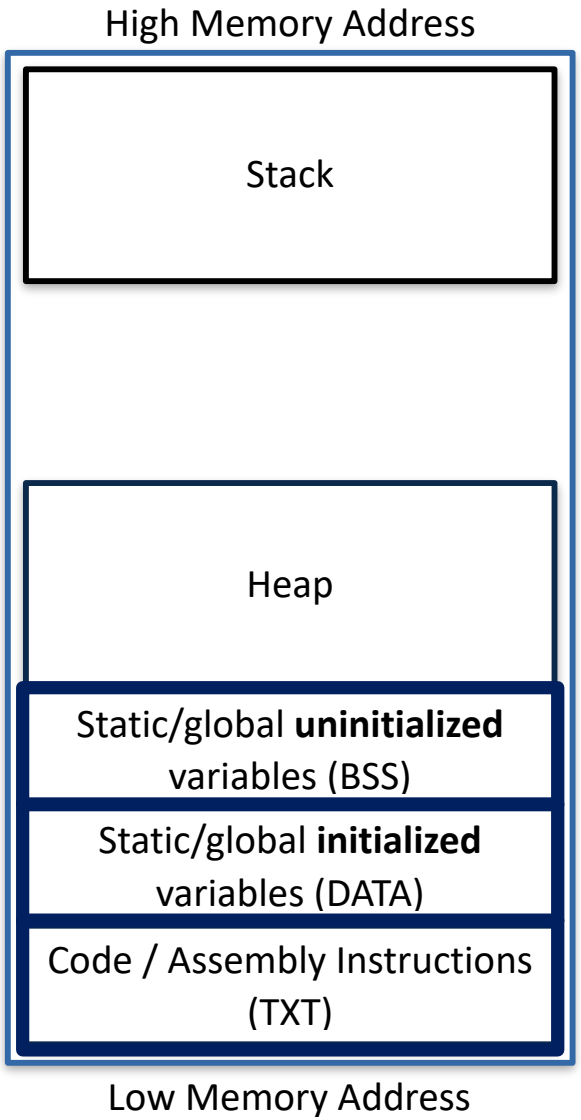
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```



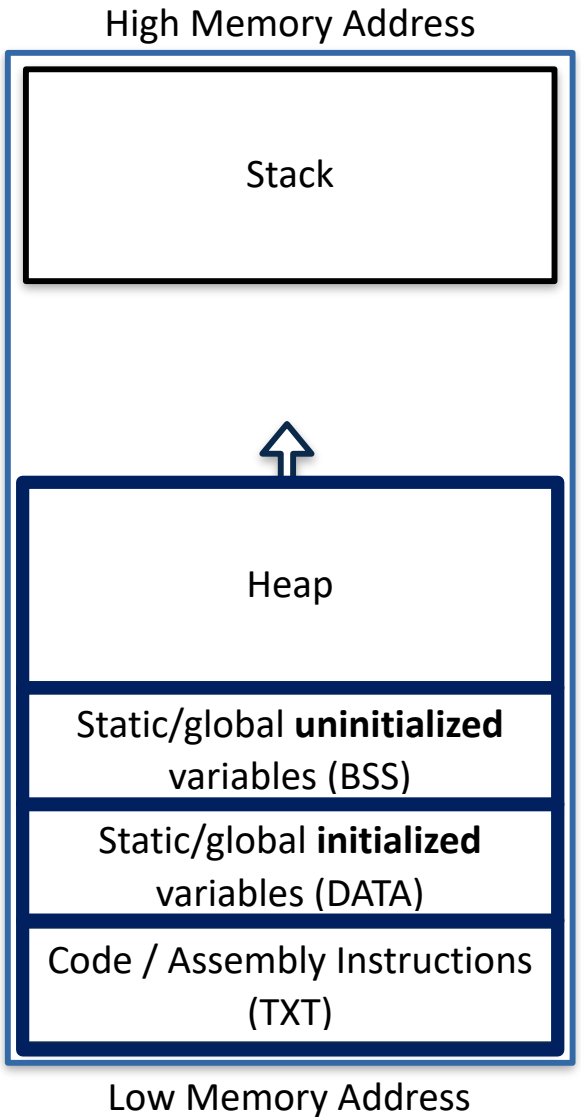
# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```



# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```



# Program Memory Structure

```
const int globalInt = 100;
char* globalString;
foo(arg1, arg2, …., argN) {
        int localVariable1, 2, ….,N;
        return 0;
}
void bar() {
        foo(1, 5, 10, 20, …., 100);
}
void main() {
        bar();
        int *ptr;
        ptr = malloc(15 * sizeof(*ptr));
}
```