System Programming

Ch00. Introduction to HW/SW
Ch01. A Tour of Computer Systems

2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

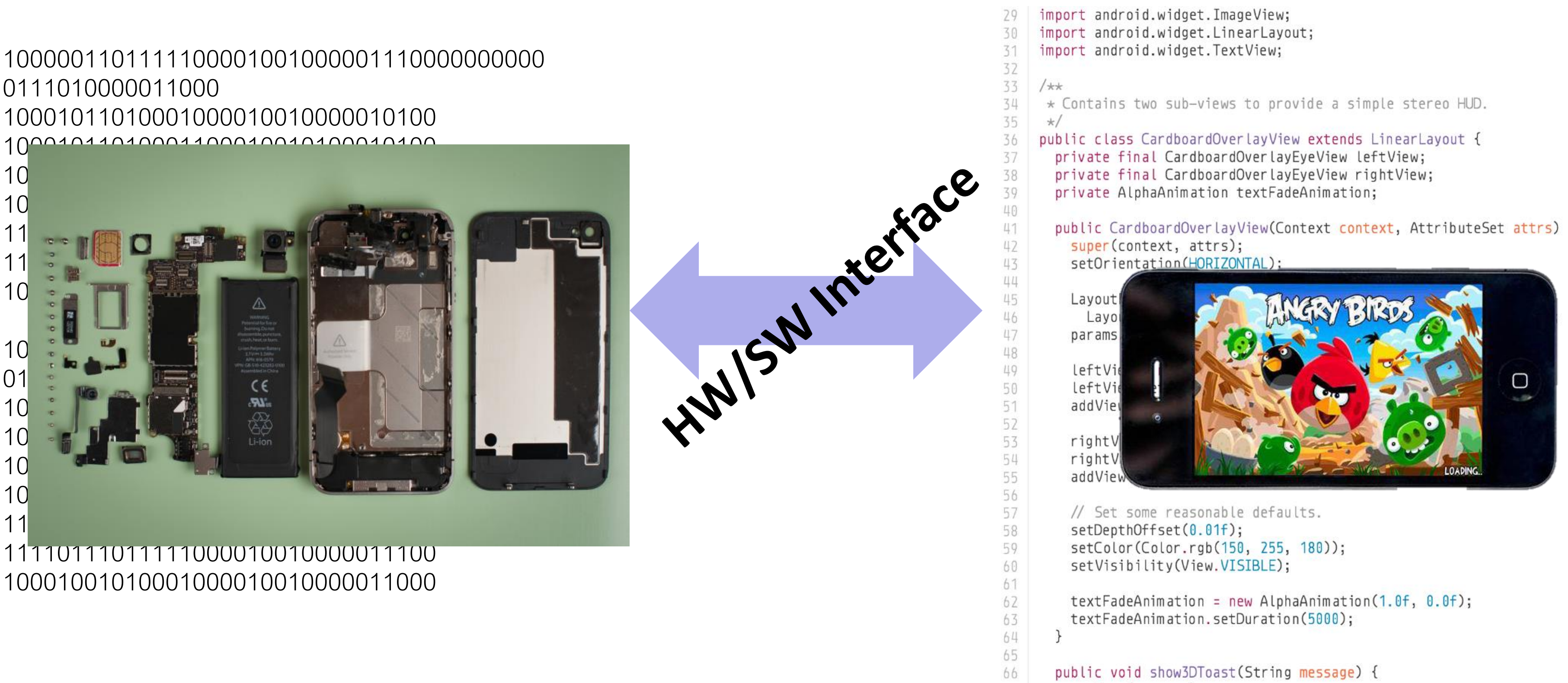
Data Science Lab @ PNU

data science laboratory

These slides are based on slides prepared by Dan Grossman.

The Hardware/Software Interface

- What do we mean by hardware? software?
- What is an interface?
- Why do we need a hardware/software interface?
- Why do we need to understand both sides of this interface?



C/Java, assembly, and machine code

```
if (x != 0) y = (y+z)/x;
```

High Level Language (e.g. C, Java)

```
        cmpl    $0, -4(%ebp)
        je      .L2
        movl    -12(%ebp), %eax
        movl    -8(%ebp), %edx
        leal    (%edx, %eax), %eax
        movl    %eax, %edx
        sarl    $31, %edx
        idivl   -4(%ebp)
        movl    %eax, -8(%ebp)
.L2:
```

Assembly Language

```
1000001101111100001001000001110000000000
0111010000011000
10001011010001000010010000010100
10001011010001100010010100010100
100011010000010000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011100
10001001010001000010010000011000
```

Machine Code

C/Java, assembly, and machine code

```
if (x != 0) y = (y+z)/x;
```

High Level Language (e.g. C, Java)

Compiler

```
        cmpl        $0, -4(%ebp)
        je  m       .L2
        ovl         -12(%ebp), %eax
        movl        -8(%ebp), %edx  (%
        leal        edx, %eax), %eax
        movl        %eax, %edx
        sarl        $31, %edx
        idivl       -4(%ebp)
        movl        %eax, -8(%ebp)
.L2:
```

Assembly Language

Assembler

```
1000001101111100001001000001110000000000
0111010000011000
10001011010001000010010000010100
10001011010001100010010100010100
100011010000010000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011100
10001001010001000010010000011000
```

Machine Code

C/Java, assembly, and machine code

```
if (x != 0) y = (y+z)/x;
```

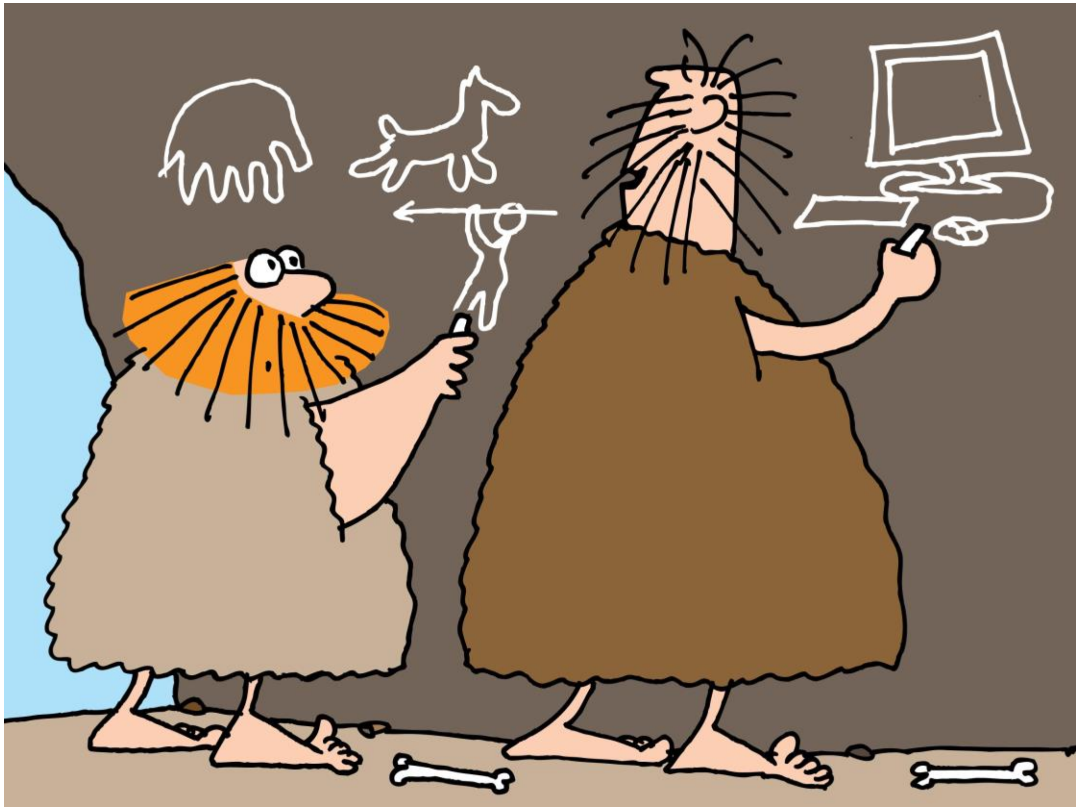
```
        cmpl    $0, -4(%ebp)
        je      .L2
        movl    -12(%ebp), %eax
        movl    -8(%ebp), %edx
        leal    (%edx, %eax), %eax
        movl    %eax, %edx
        sarl    $31, %edx
        idivl   -4(%ebp)
        movl    %eax, -8(%ebp)
.L2:
```

```
1000001101111100001001000001110000000000
0111010000011000
10001011010001000010010000010100
10001011010001100010010100010100
100011010000010000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011100
10001001010001000010010000011000
```

- The three program fragments are equivalent
- You'd rather write C! - a more human-friendly language
- The hardware likes bit strings! - everything is voltages
  - Everything is voltages
  - The machine instructions are actually much shorter than the number of bits we would need to represent the characters in the assembly language

HW/SW Interface: Historical Perspective

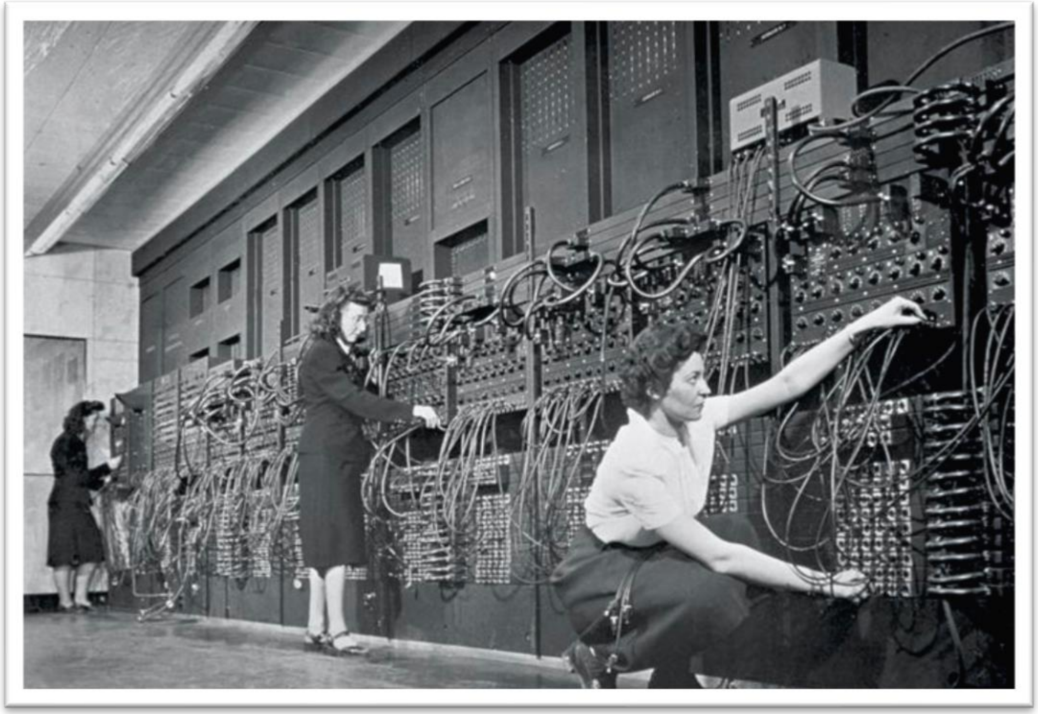
- Hardware started out quite primitive



HW/SW Interface: Historical Perspective

- Hardware started out quite primitive

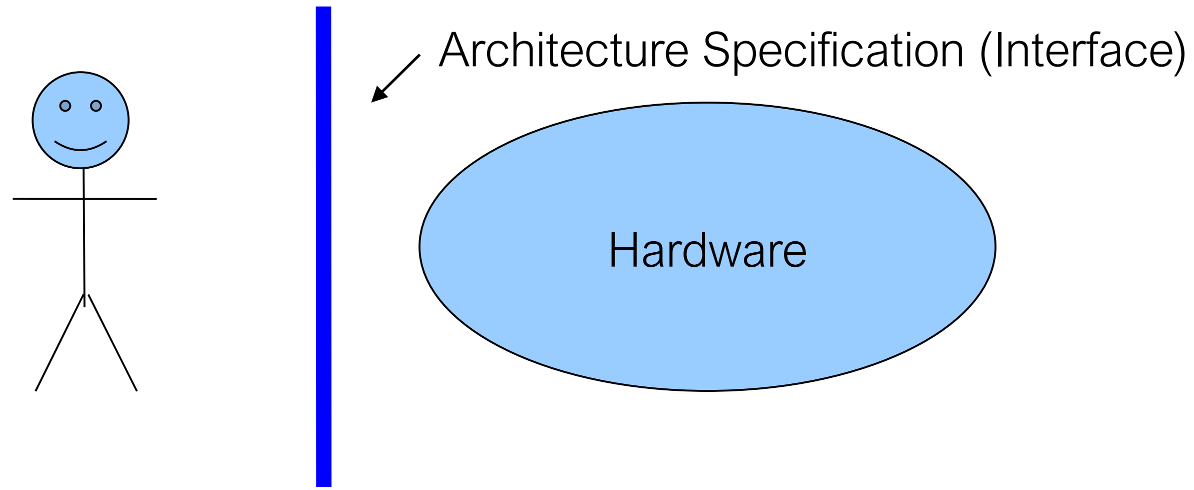
http://hightechhistory.com/2011/10/24/the-late-dennis-macalistair-ritchie-innovator-of-the-%E2%80%9Cc%E2%80%9D-programming-language-and-the-unix-operating-system-an-appreciation/



Jean Jennings (left), Marlyn Wescoff (center), and Ruth Lichterman program ENIAC at the University of Pennsylvania, circa 1946. Photo: Corbis

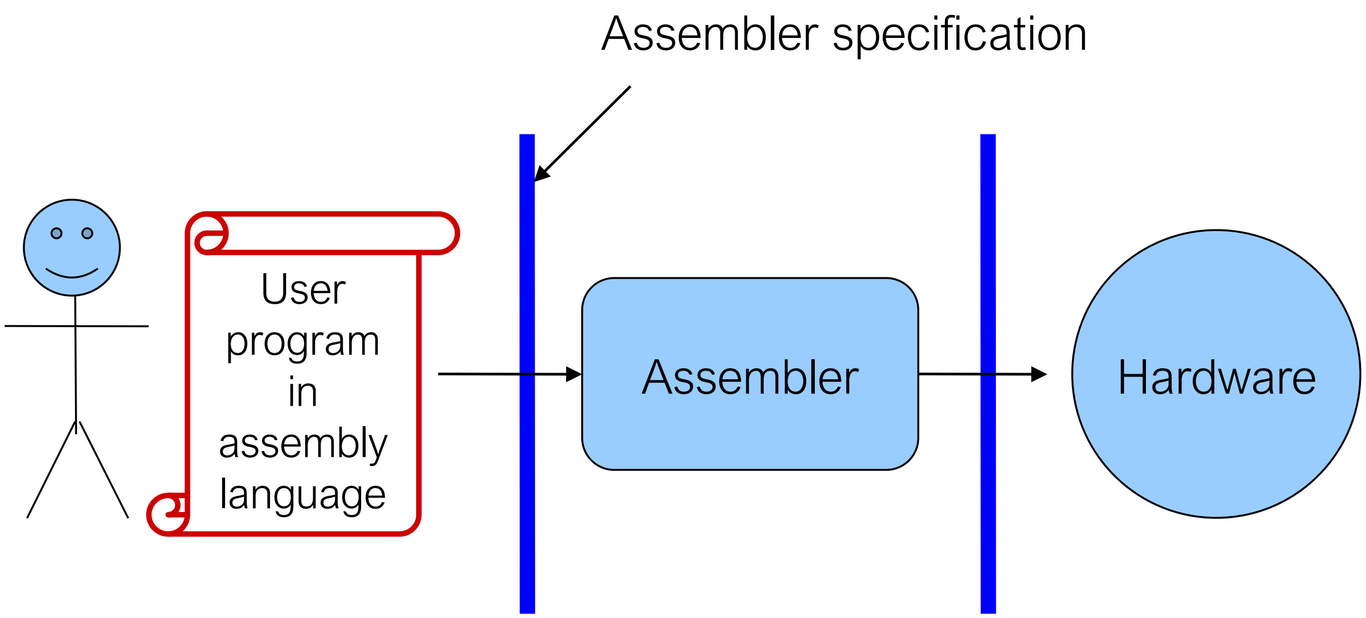
HW/SW Interface: The Historical Perspective

- Hardware started out quite primitive
  - Programmed with very basic instructions.
  - E.g. a single instruction for adding two integers
- Software was also very basic
  - Closely reflected the actual hardware it was running on
  - Specify each step manually



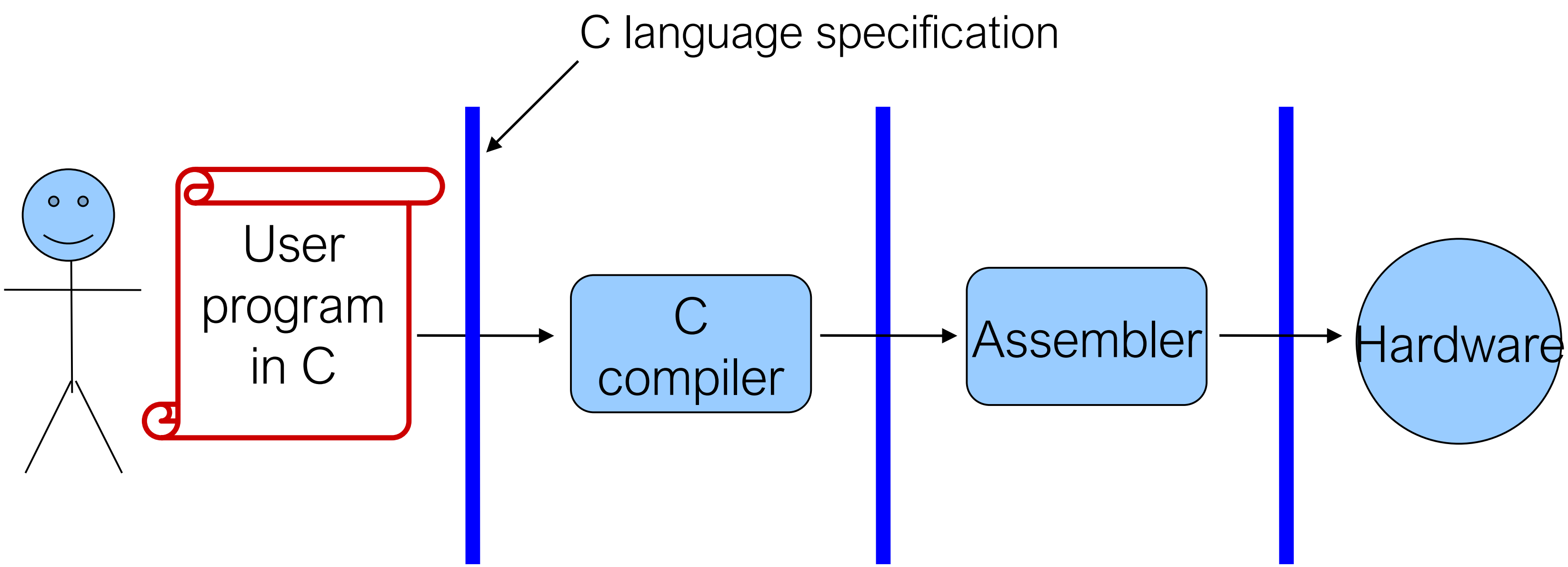
HW/SW Interface: Assemblers

- Life was made a lot better by assemblers
  - 1 assembly instruction = 1 machine instruction, but...
  - different syntax: assembly instructions are character strings, not bit strings, a lot easier to read/write by humans
  - can use symbolic names

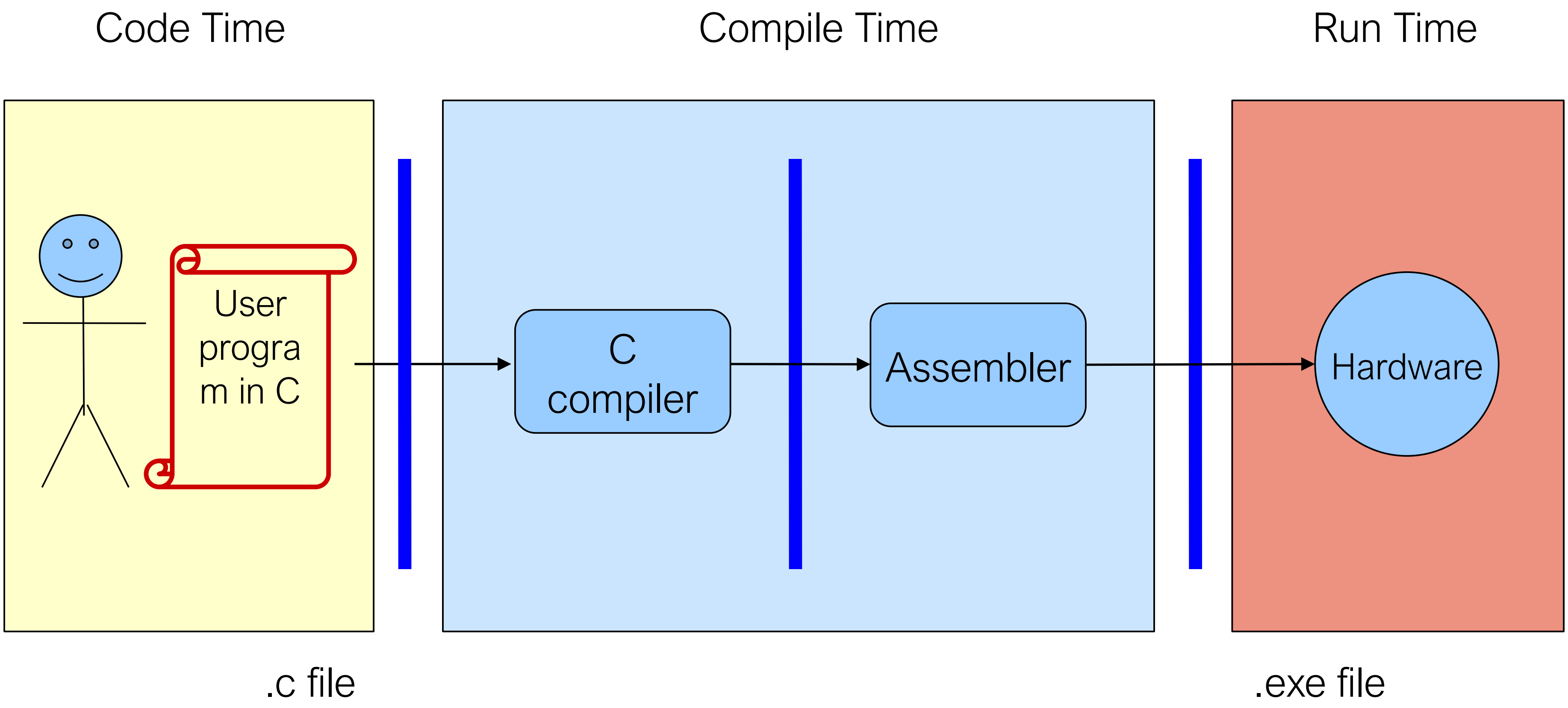


HW/SW Interface: Higher-Level Languages

- Higher level of abstraction:
  - 1 line of a high-level language is compiled into many (sometimes very many) lines of assembly language



HW/SW Interface: Code / Compile / Run Times



Note: The compiler and assembler are just programs, developed using this same process.

The Big Theme: Abstractions and Interfaces

- Computing is about abstractions
  - (but we can’t forget reality)
- What are the abstractions that we use?

- What do YOU need to know about them?
  - When do they break down and you have to peek under the hood?
  - What bugs can they cause and how do you find them?

- How does the hardware (0s and 1s, processor executing instructions) relate to the software (C/Java programs)?
  - Become a better programmer and begin to understand the important concepts that have evolved in building ever more complex computer systems

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8

Mac

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Little Theme 1: Representation

- All digital systems represent everything as 0s and 1s
  - The 0 and 1 are really two different voltage ranges in the wires
  - Or magnetic positions on a disc, or hole depths on a dvd, or…
- “Everything” includes:
  - Numbers – integers and floating point
  - Characters – the building blocks of strings
  - Instructions – the directives to the CPU that make up a program
  - Pointers – addresses of data objects stored away in memory
- These encodings are stored throughout a computer system
  - In registers, caches, memories, disks, etc.
- They all need addresses
  - A way to find them
  - Find a new place to put a new item
  - Reclaim the place in memory when data no longer needed

Little Theme 2: Translation

- There is a big gap between how we think about programs and data and the 0s and 1s of computers

- Need languages to describe what we mean
- Languages need to be translated one step at a time

- We know C/Java as a programming language
  - Have to work our way down to the 0s and 1s of computers
  - Try not to lose anything in translation!
  - We'll encounter Java byte-codes, C language, assembly language, and machine code (for the X86 family of CPU architectures)
    - Not in that order, but will all connect by the last lecture!!!

Little Theme 3: Control Flow

- How do computers orchestrate the many things they are doing?
- In one program:
  - How do we implement if/else, loops, switches?
  - What do we have to keep track of when we call a procedure, and then another, and then another, and so on?
  - How do we know what to do upon “return”?
- Across programs and operating systems:
  - Multiple user programs
  - Operating system has to orchestrate them all
    - Each gets a share of computing cycles
    - They may need to share system resources (memory, I/O, disks)
  - Yielding and taking control of the processor
    - Voluntary or “by force”?

Writing Assembly Code??? In 2019??? (1/2)

- Chances are, you'll never write a program in assembly code
  - Compilers are much better and more patient than you are

Writing Assembly Code??? In 2019??? (2/2)

- But: understanding assembly is the key to the machine-level execution model
  - Behavior of programs in presence of bugs
    - High-level language model breaks down
  - Tuning program performance
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Operating systems must manage process state
  - Fighting malicious software
  - Using special units (timers, I/O co-processors, etc.) inside processor!

Course Perspective

- This course will make you a better programmer
  - Purpose is to show how software really works
  - Understanding the underlying system makes you more effective
  - Better debugging
  - Better basis for evaluating performance
  - How multiple activities work in concert (e.g., OS and user programs)
- Not just a course for hardware enthusiasts!
  - What every CSE major needs to know (plus many more details)
  - Job interviewers love to ask questions from this course!
- Like other courses, “stuff everybody learns and uses and forgets not knowing”

HTTP://XKCD.COM/676/

Ch01. A Tour of Computer Systems

The hello Program

```
#include <stdio.h>


int main() {
  printf(“hello, world\n”);
}
```

Information is Bits

- Source File
  - The hello program begins life as a source program
  - Most programs consist of multiple source files
- Representation
  - A sequence of bits with a value of 0 or 1
  - Organized into bytes of 8 bits each
  - Each byte represents a character in the program

Character Interpretation

- Bytes and Characters
  - Characters are encoded in bytes
  - Bytes are 8 bits
  - A bit is a 1 or a 0
- Bytes and Numbers
  - A byte can represent a number in base--2: $00000110_2 = 6_{10}$, $00100000_2 = 64_{10}$, $10000111_2 = 135_{10}$, $111111112 = 255_{10}$
  - Numbers can be used to represent characters

Representing Characters

- ASCII Standard (ASCII = American Standard Code for Information Interchange)
  - Represents each character with a unique byte--sized integer value

The ASCII text representation of **hello** source file

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

Unicode

- ASCII has limitations (only 128 characters).
- Unicode is an extension of ASCII.
- Unicode characters can be stored in 32 bits, but there are representations of them that use fewer bits.
- Java uses Unicode, though Linux does not.

Program Translation

- Program Source Files
  - Beginning of life for a C program
  - Represented as ASCII character text
  - “Easy” for humans to understand
  - Not understood by machines
- Executable Object File
  - Low--level primitive machine operations
  - Understood by the machine
- Program Translation
  - Translates source file into object (machine code) file!
  - Also known as **compilation**

Compilation System

`unix> gcc -o hello hello.c`

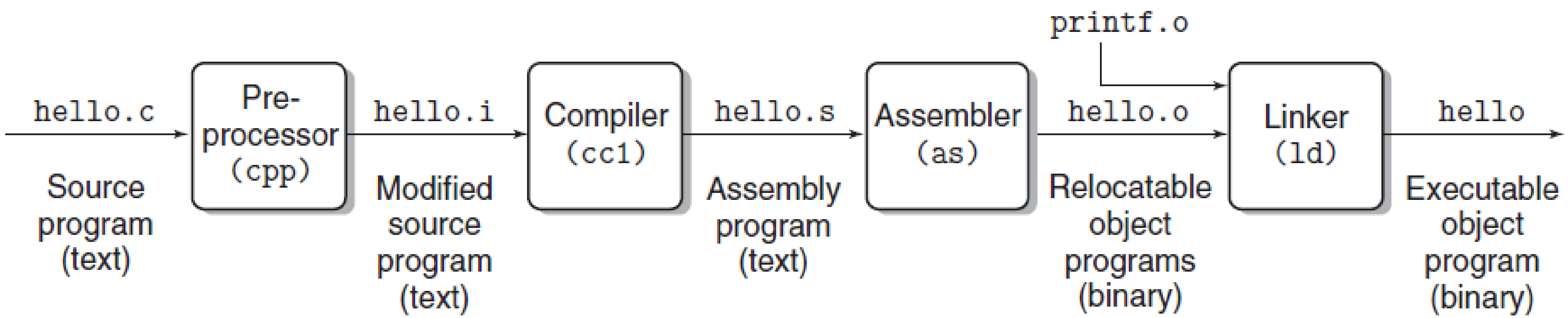


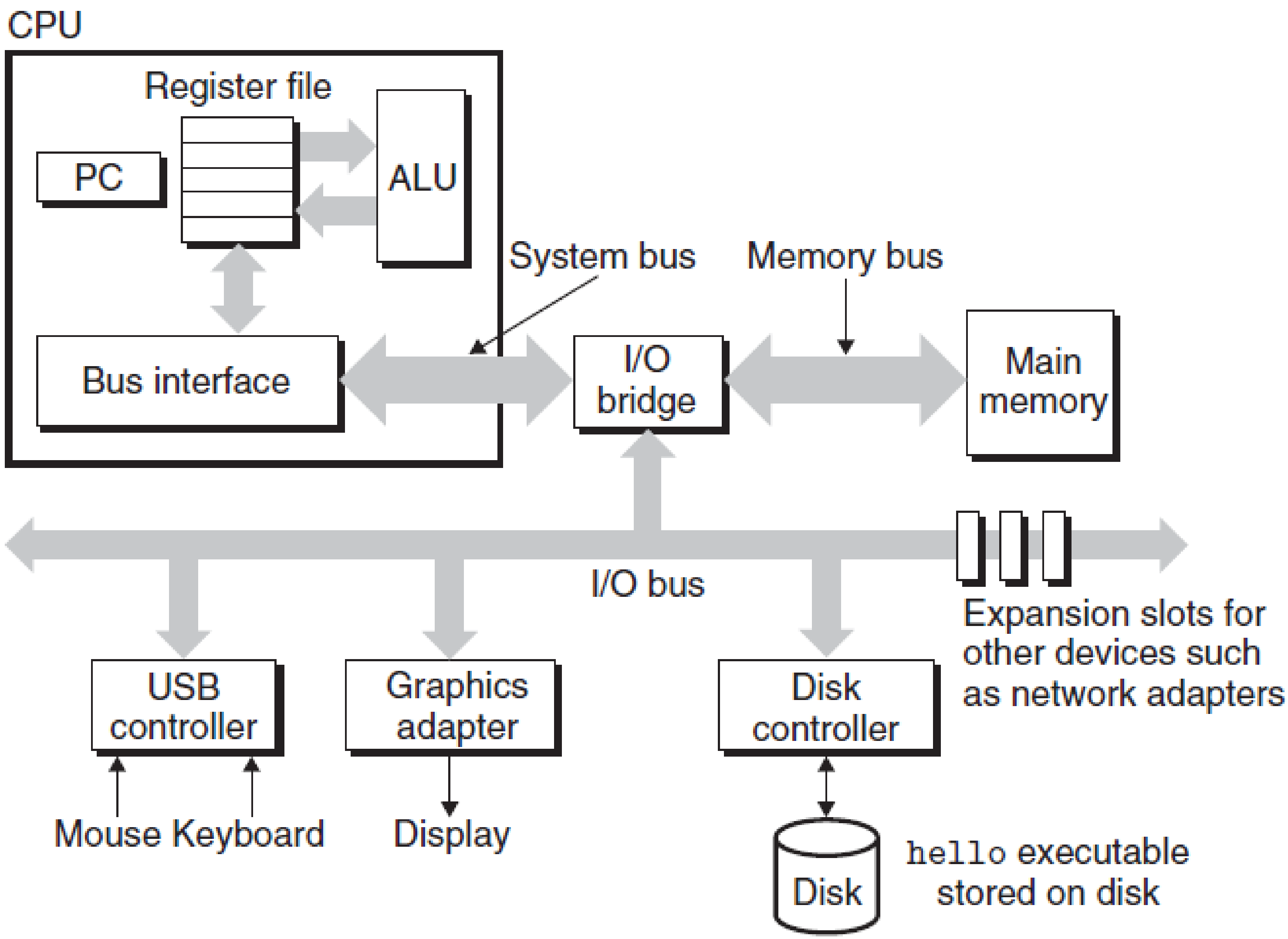
Figure 1.3 The compilation system.

Processors

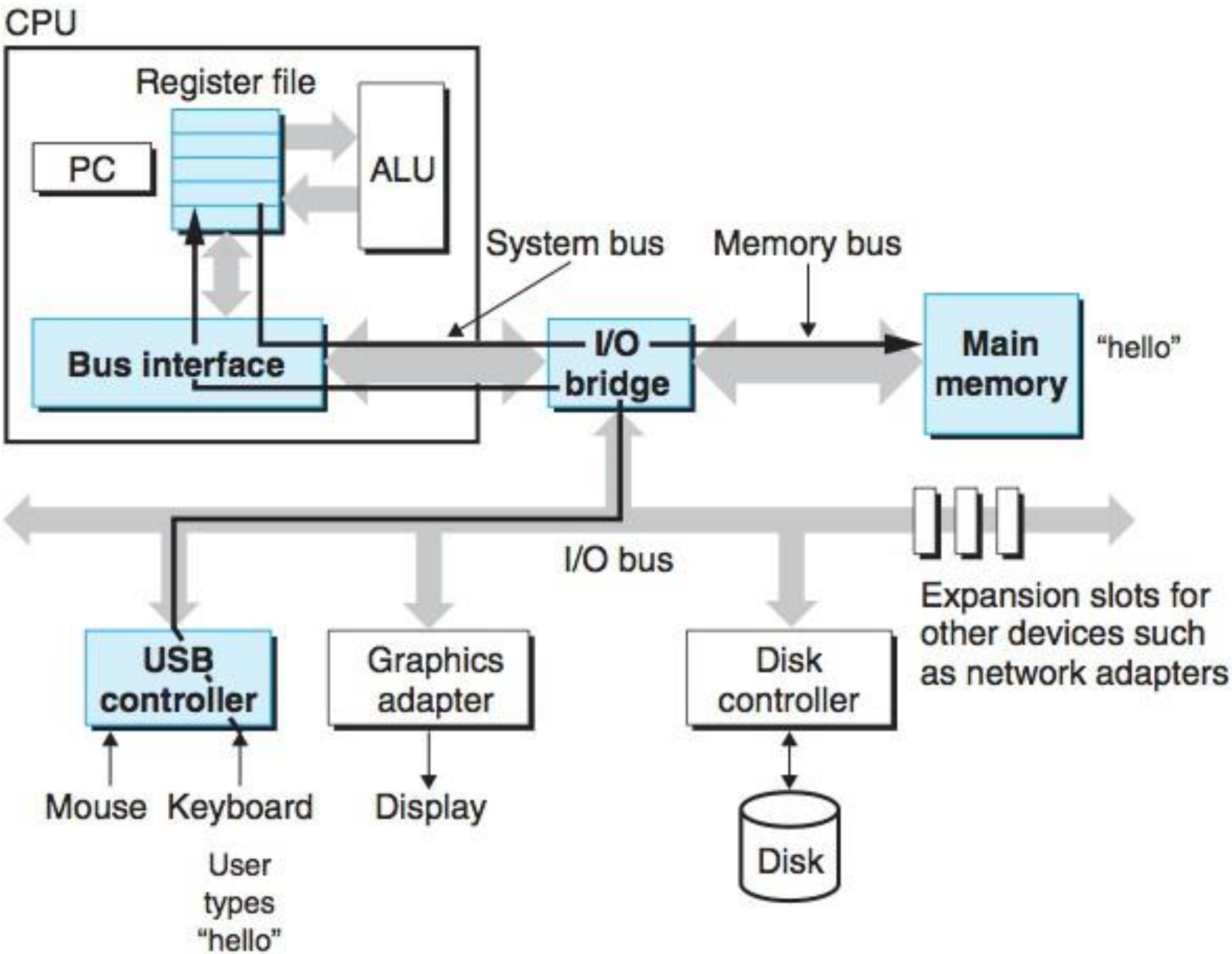
- Machine Code Instructions
  - Programs at the only level a machine can understand
  - Stored in memory

- Processors
  - Read instructions from memory
  - Interpret those instructions (do what the instructions say to do)
  - Implemented in hardware

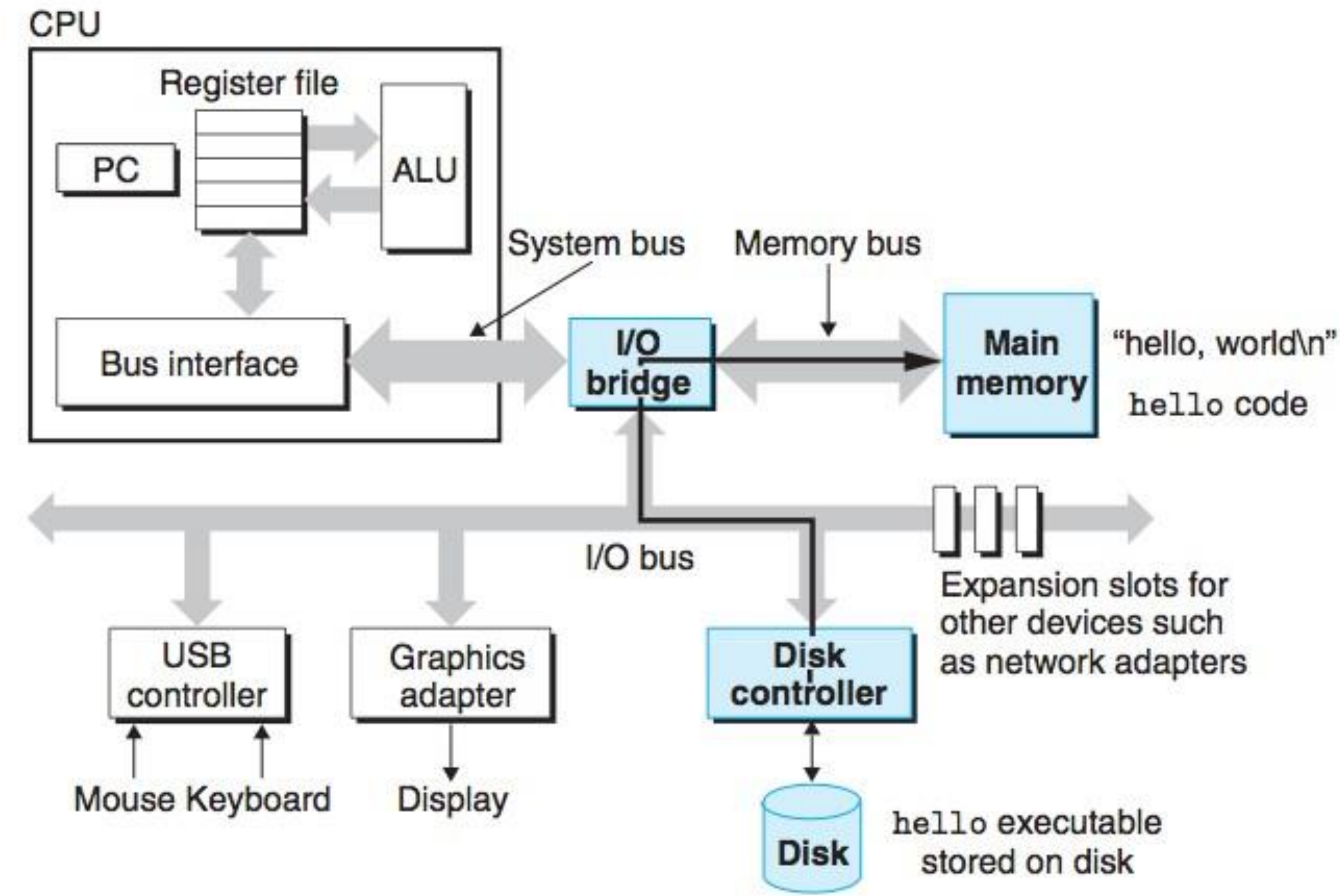
Hardware Organization



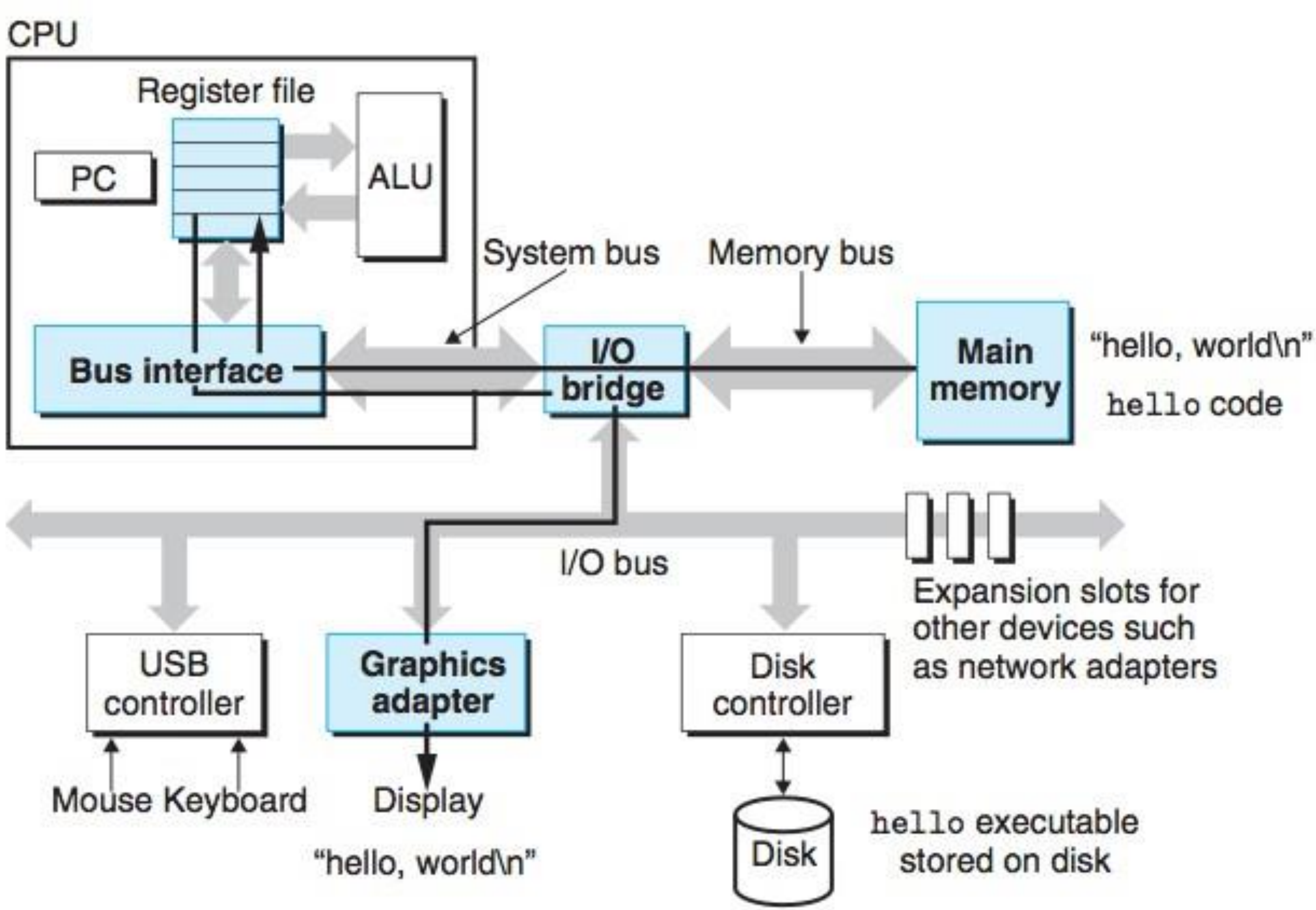
Reading hello command from keyboard



Loading the executable from disk into main memory



Writing the output string from memory to the display



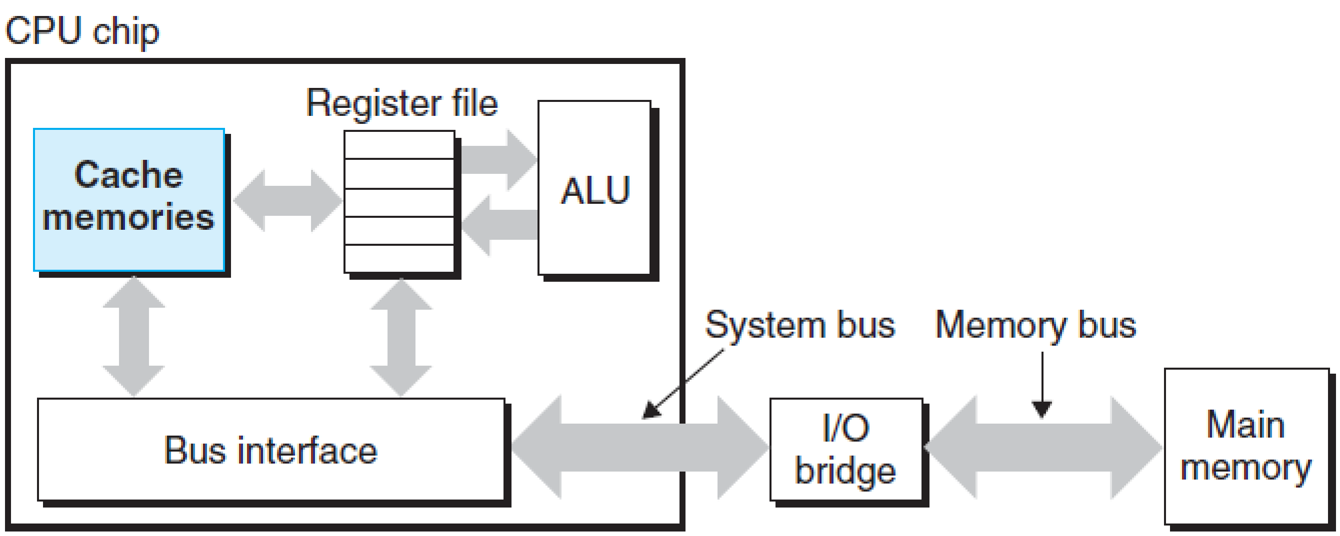
Memory this, Memory that

- Memory is Important
  - Stores program code
  - Stores program data
  - Accesses required for execution

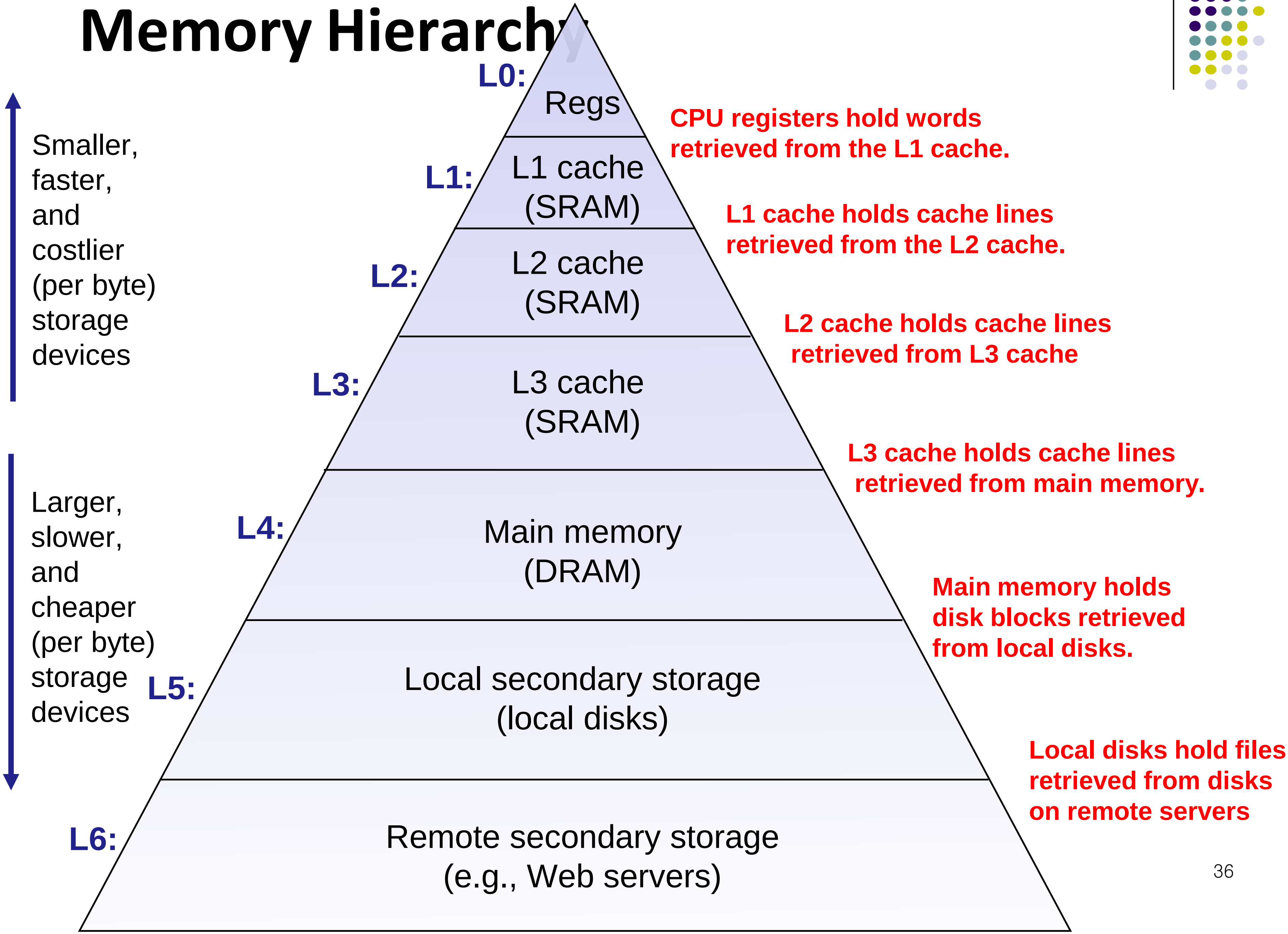
- Memory is Slow
  - Yep, it takes a long time to access memory
  - Need a mechanism to reduce memory latency

Cache

- Smaller Memories
  - Resides on CPU chip
  - Larger than register file
  - Smaller than RAM
- Locality
  - Access to program code and data tends to exhibit a high degree of locality, on both space and time
  - Caches exploit this!

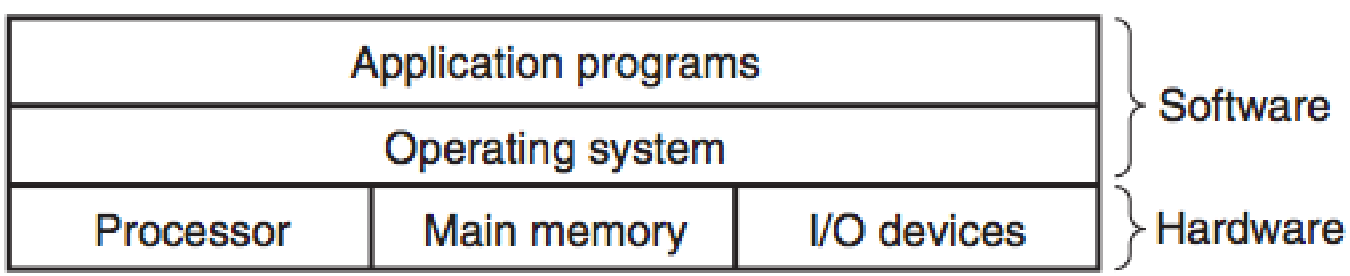


Memory Hierarchy



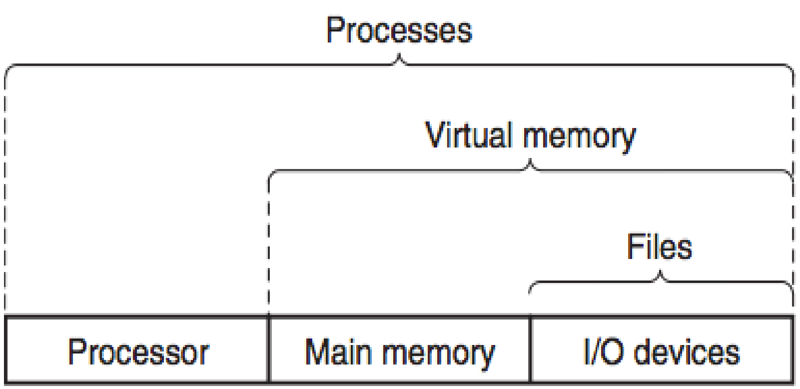
Operating System

- Two Primary Purposes
  - to protect the hardware (and other programs and files) from misuse by runaway applications
  - provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low--level hardware devices



OS Abstractions

- How does the OS do this?
  - Three fundamental abstractions



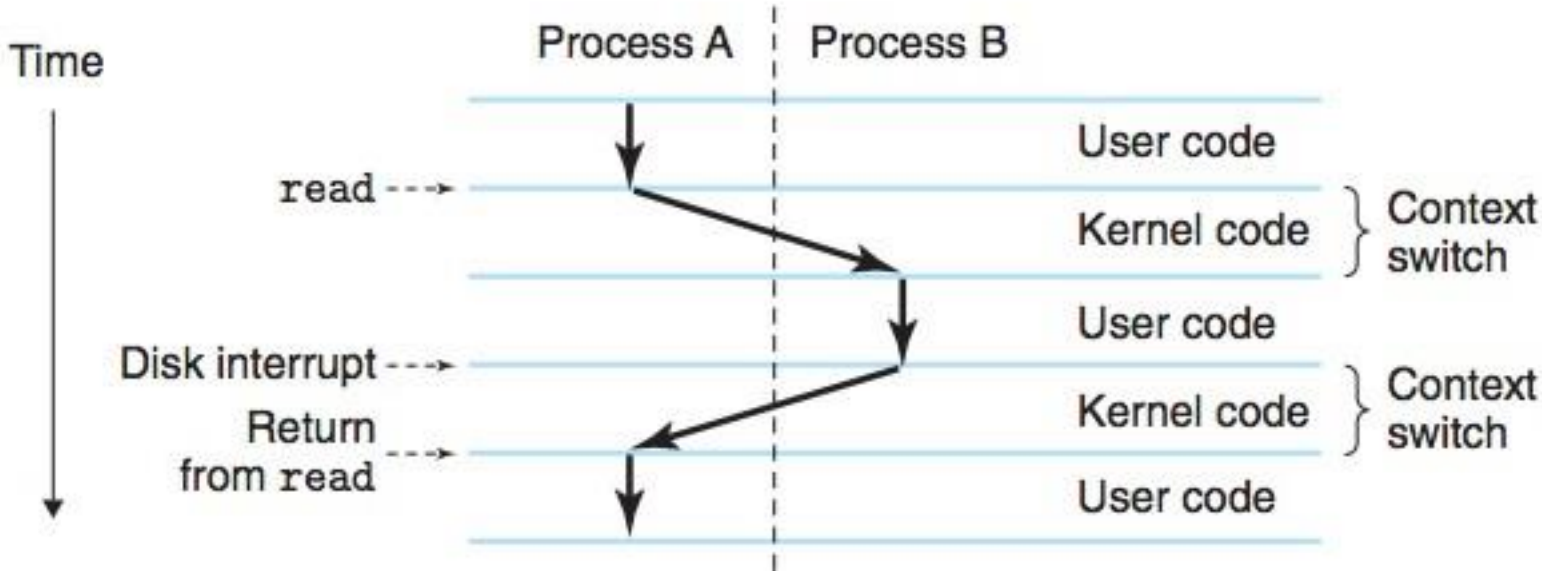
- 1: Files
  - Abstraction for I/O devices
- 2: Virtual Memory
  - Abstraction for main memory
  - Abstraction for I/O devices
- 3: Processes
  - Abstraction for the processor
  - Abstraction for main memory
  - Abstraction for I/O devices

Processes

- What are they?
  - An abstraction for a running program
- How many?
  - Lots of them
  - Multiple processes can run concurrently
- What do they give us?
  - Illusion that each program has exclusive access to the processor and memory

What does “concurrently” mean?

- The machine code instructions of one process are **interleaved** with the machine code instructions of another process.



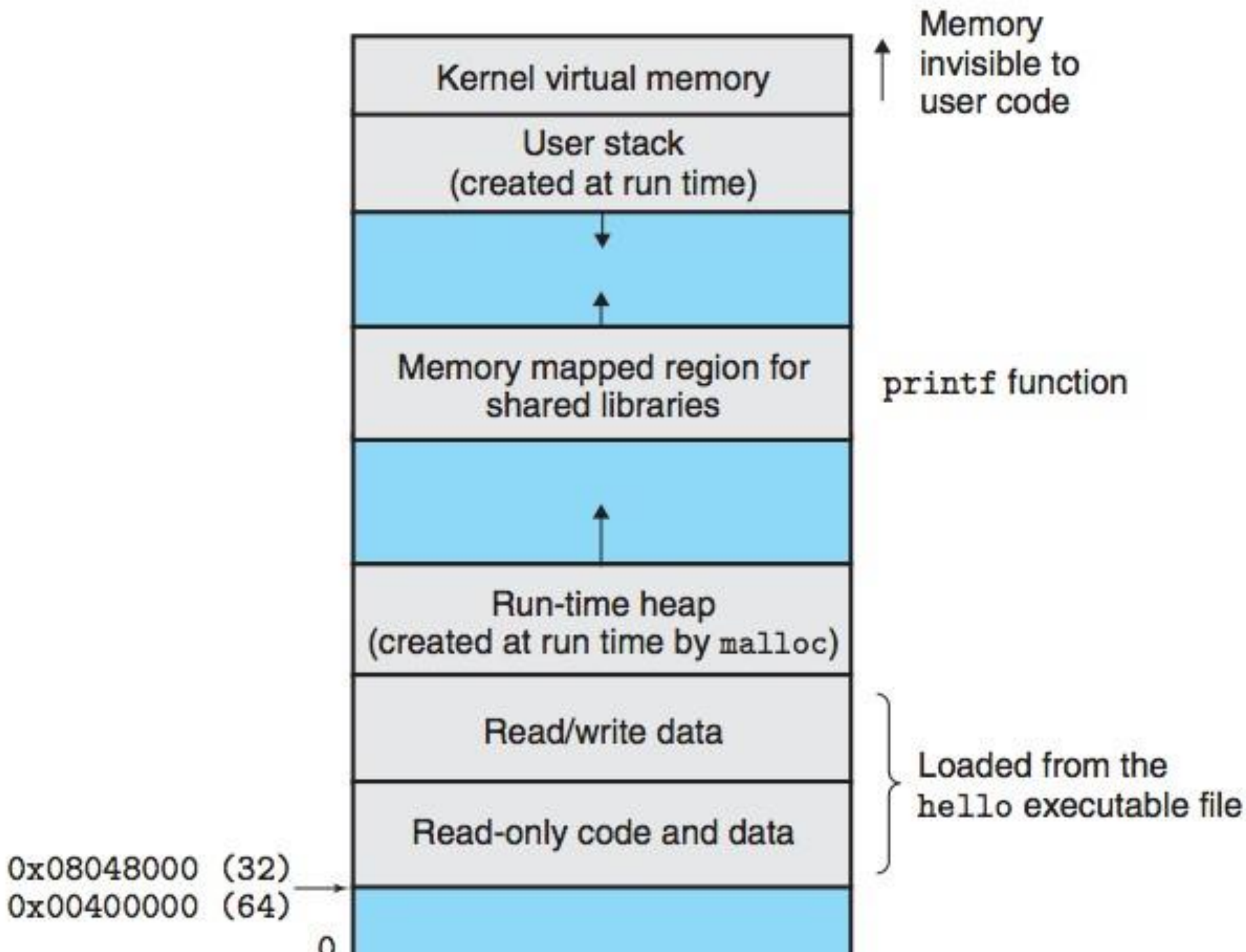
Aside: What about multiple “cores”?

- Each “core” in a multi--core CPU
  - is effectively a separate CPU
  - can context--switch independently
- If enough processes are ready to run, two or more cores can be running programs at the same time
- Can be thought of as multiple computers on the same chip, but managed by the same OS and sharing the same memory and I/O devices

Processes and Threads

- Processes
  - The illusion is great, but what if I want to share my memory with another process?
  - You can't!
- Threads
  - Associated with each process
  - Can be lots of them
  - Can share memory between them

Virtual Memory

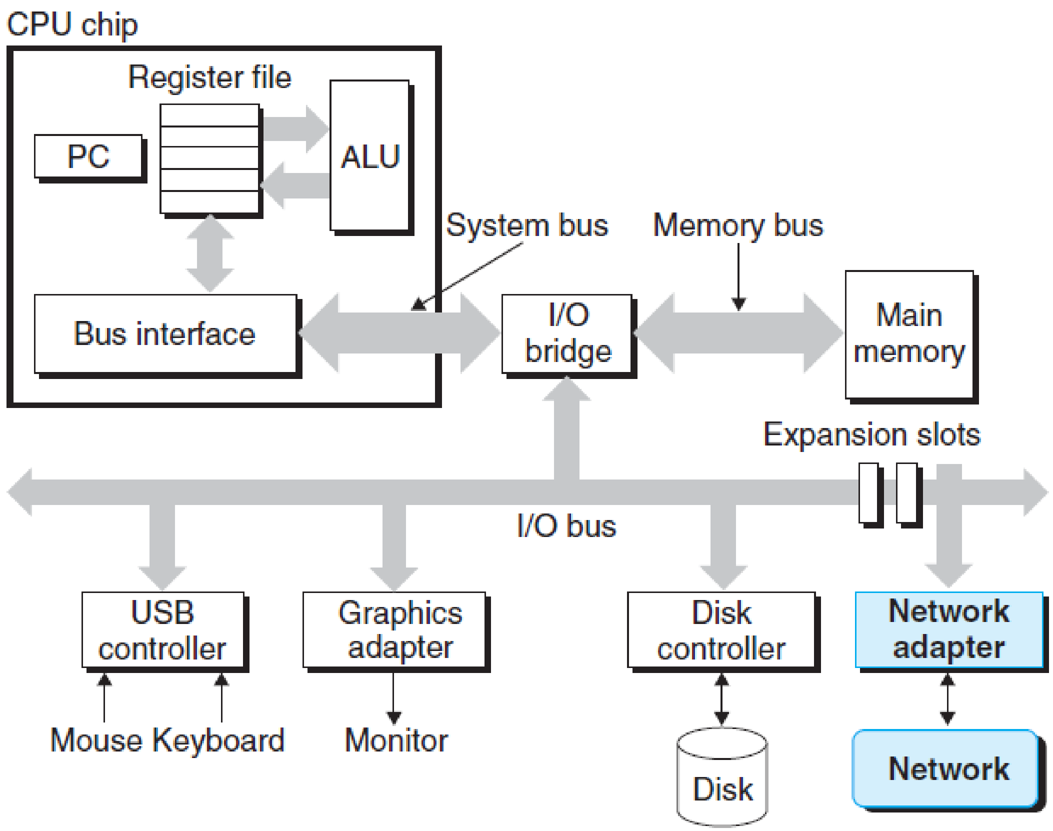


Files

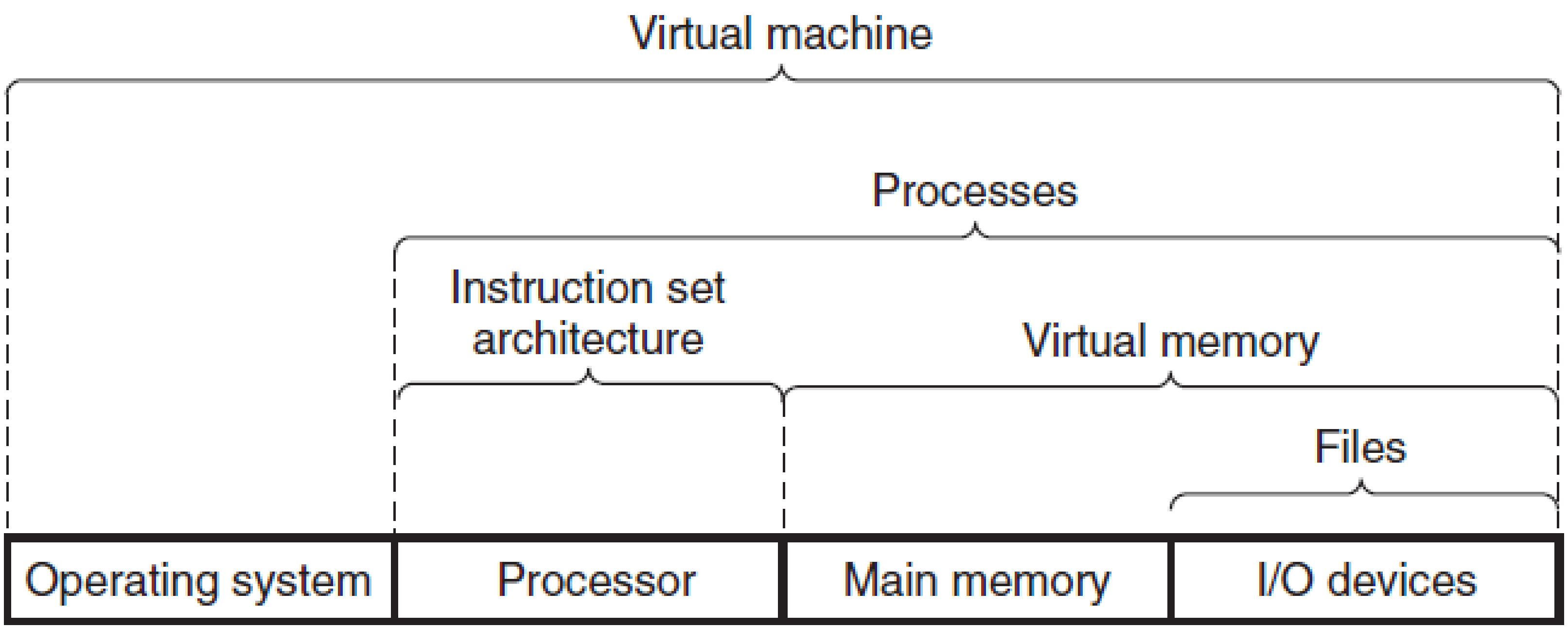
- **Sequence of bytes… nothing more, nothing less**
- **File System** resides on secondary storage (disks)

Network Communication

- **Processes like to talk to other processes**



Abstractions in Computer Systems



Q&A