

```

29 struct {
30     int..
31     ScePspFVector3
32     ScePspFVector3
33     int..
34     float.
35 } BALLOONDAT;
36
37 static BALLOONDAT
38 static ScePspFVector3
39 static ScePspFVector3
40
41 extern void DrawSphere(ScePspFVector3 *array, float r)
42 extern void DrawPole(ScePspFVector3 *array, float r)
43
44 void init_balloon(void)
45 {
46     int.. 1;
47
48     balloon.mode=MODE
49     balloon.pos.x= 0.
50     balloon.pos.y=-8.
51     balloon.pos.z= 0.
52     balloon.t=0.0f;
53     balloon.scnt=2;
54
55     for (i=0; i<3; i++)
56     {
57         balloon.sbuf[i].x=RANGERRAND(0.9f, 0.91f, 2000)
58         balloon.sbuf[i].y=RANGERRAND(0.0f, 0.91f, 2000)
59         balloon.sbuf[i].z=RANGERRAND(0.0f, 0.91f, 2000)
60     }
61
62 void draw_balloon(void)
63 {
64     ScePspFVector3 vec;
65     vec.x=balloon.pos.x;
66     vec.y=balloon.pos.y;
67     vec.z=balloon.pos.z;
68     DrawSphere(&vec, 1000);
69     DrawPole(&vec, 1000);
70 }

```

# Operating Systems and C

## Fall 2022

### 1. Computer Systems

# Why this course?

Why teach about operating systems?

Why teach about C?

Why teach about operating systems AND C?

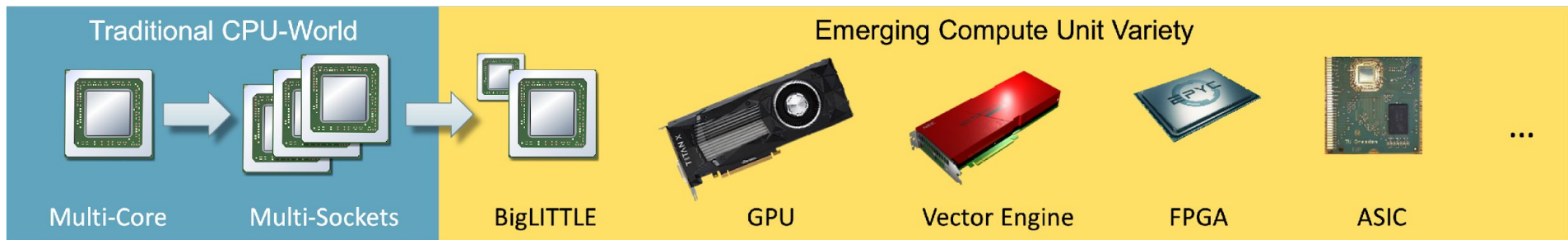
# Why this course?

“An operating system (OS) is a program that manages computer hardware. And although today's commercial-off-the-shelf desktop operating systems appear to be an integral part of PCs and workstation to many users, a fundamental understanding of the algorithms, principles, heuristics, and optimizations used is crucial for creating efficient application software. Furthermore, many of the principles in OS courses are relevant to large system applications like databases and web servers.”

A. Polze (U.Potsdam)

OS gives upper layers **abstraction** over available HW.  
learning OS is learning principles of *how app is structured*.  
organization of OS not just relevant for OS, but other large applications.

# Computer Hardware



General Purpose	Compute Optimised	Memory Optimised	Accelerated Computing	Storage Optimised
 ARM based core and custom silicon	 Compute - CPU intensive apps and DB's	 RAM - Memory intensive apps and DB's	 Processing optimised - Machine Learning	 High Disk Throughput - Big data clusters
 Tiny - Web servers and small DB's		 Xtreme RAM - For SAP/Spark	 Graphics Intensive - Video and streaming	 IOPS - NoSQL DB's
 Main - App servers and general purpose		 High Compute and High Memory - Gaming	 Field Programmable - Hardware acceleration	 Dense Storage - Data Warehousing

Name	vCPUs	Memory (GiB)	Network Bandwidth (Gbps)	EBS Throughput (Gbps)
m6i.large	2	8	Up to 12.5	Up to 10
m6i.xlarge	4	16	Up to 12.5	Up to 10
m6i.2xlarge	8	32	Up to 12.5	Up to 10
m6i.4xlarge	16	64	Up to 12.5	Up to 10
m6i.8xlarge	32	128	12.5	10
m6i.12xlarge	48	192	18.75	15
m6i.16xlarge	64	256	25	20
m6i.24xlarge	96	384	37.5	30
m6i.32xlarge	128	512	50	40

AWS EC2 instance types

# Why this course?

“C has the power of assembly language and the convenience of ... assembly language.”

D. Ritchie

“Learn at least one programming language every year.”

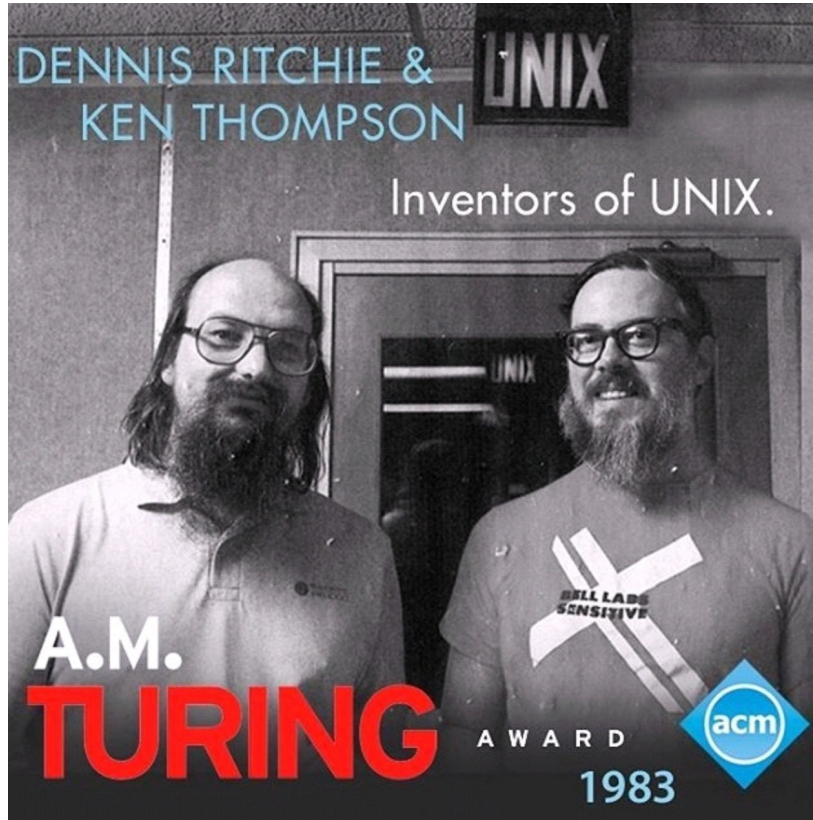
A.Hunt and D.Thomas, The Pragmatic Programmer.

C is a mess. syntactic sugar on top of assembly (Linus Torvalds quote)  
why learn C: to understand how computers work. (C and Linux)  
care about security, performance, resource utilization? C gives control.  
high-level PLs abstract away many issues.  
learn a PL each semester. this semester: C.

# Why this course?

C has a rich history. UNIX - C

## ACM citation:



- The success of the UNIX system stems from its **tasteful selection** of a **few key ideas** and their **elegant implementation**. The model of the Unix system has led a generation of software designers to new ways of thinking about programming. The genius of the Unix system is its framework, which enables programmers to stand on the work of others.
- **Ken Thompson also created an interpretive language called B**, based on BCPL, which he used to re-implement the non-kernel parts of Unix. **Ritchie added types** to the B language, and later created a compiler for **the C language**. Thompson and Ritchie rewrote most of Unix in C in 1973, which made further development and porting to other platforms much easier.

# Why this course?



<https://github.com/torvalds/linux>



<https://gcc.gnu.org/>

# What is in it for you?

You want to become  
a software  
engineer?

You want to become  
a programmer?

You want to become  
a data engineer?

You want to get your  
Bachelor?



# What is in it for you?

- Deep understanding of how computer systems impact software design
- Way to learn a new programming language
- Proficiency in shell, Linux, vim
- (First) experience with system programming

## **Security | Performance**

- General knowledge: history, (geo-)politics, business

linux written in C. extremely successful OS.  
compiled using gcc. open-source movement.

# Outline

## Part I: Overview

1. Why this course?
2. What is in it for you?

## Part II: What is this class about?

1. **Computer Systems**
2. Operating Systems
3. C Programming Language
4. Take-away

## Part III: Logistics

# Model of Computation

Computers implement a model of computation ("mechanized arithmetic").

Many models of computation exist.

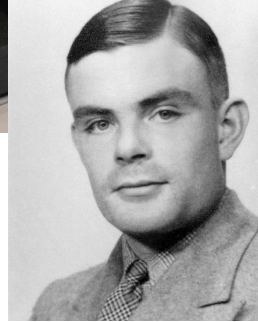
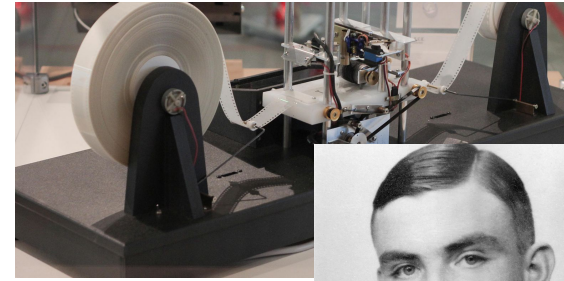
- Turing Machine, Counter Machines, ...

Why current computation model?  
(CISC, RAM, Von-Neumann Arch., ...)  
(choice seems **arbitrary!**)

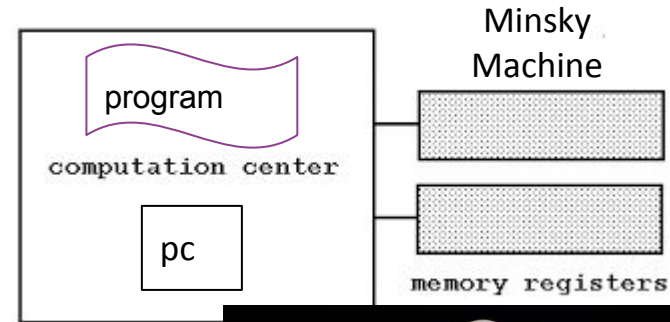
- performance
- cost, convenience

systems research:  
manage **trade-offs!**

Turing Machine



Alan Turing,  
1937

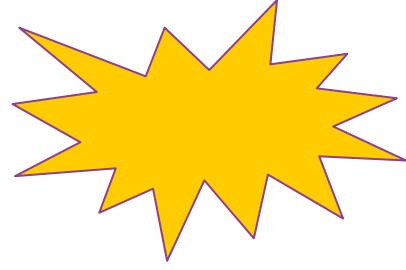


Marvin Minsky,  
1967

A system is a set of interconnected components with a well-defined behavior at the interface with its environment.

Coping with system complexity:

- Modularity, Abstraction, Layering, Hierarchy

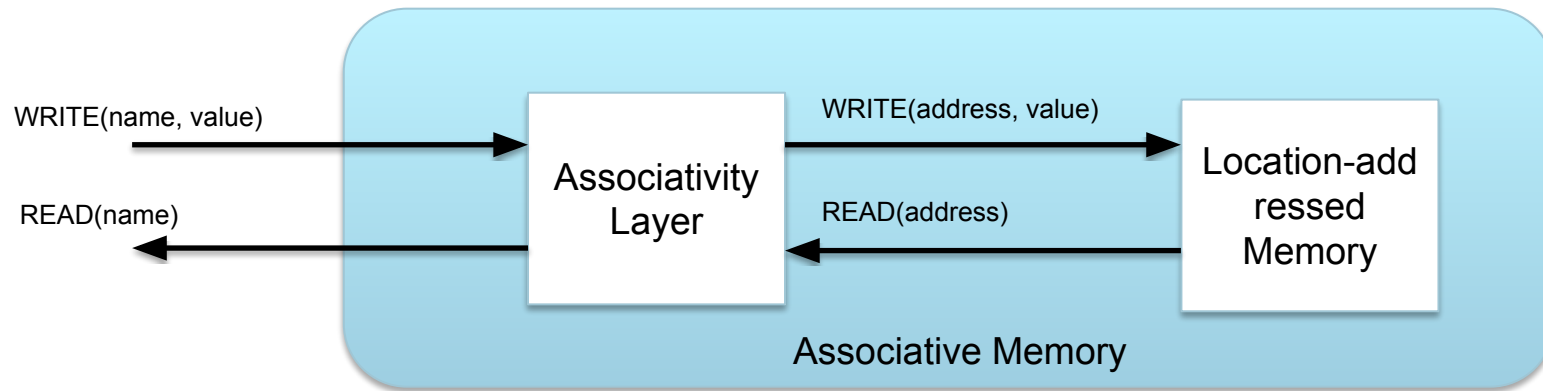


3 fundamental abstractions for computer systems:

- Interpreter
- Memory
- Communication

# Memory Abstraction

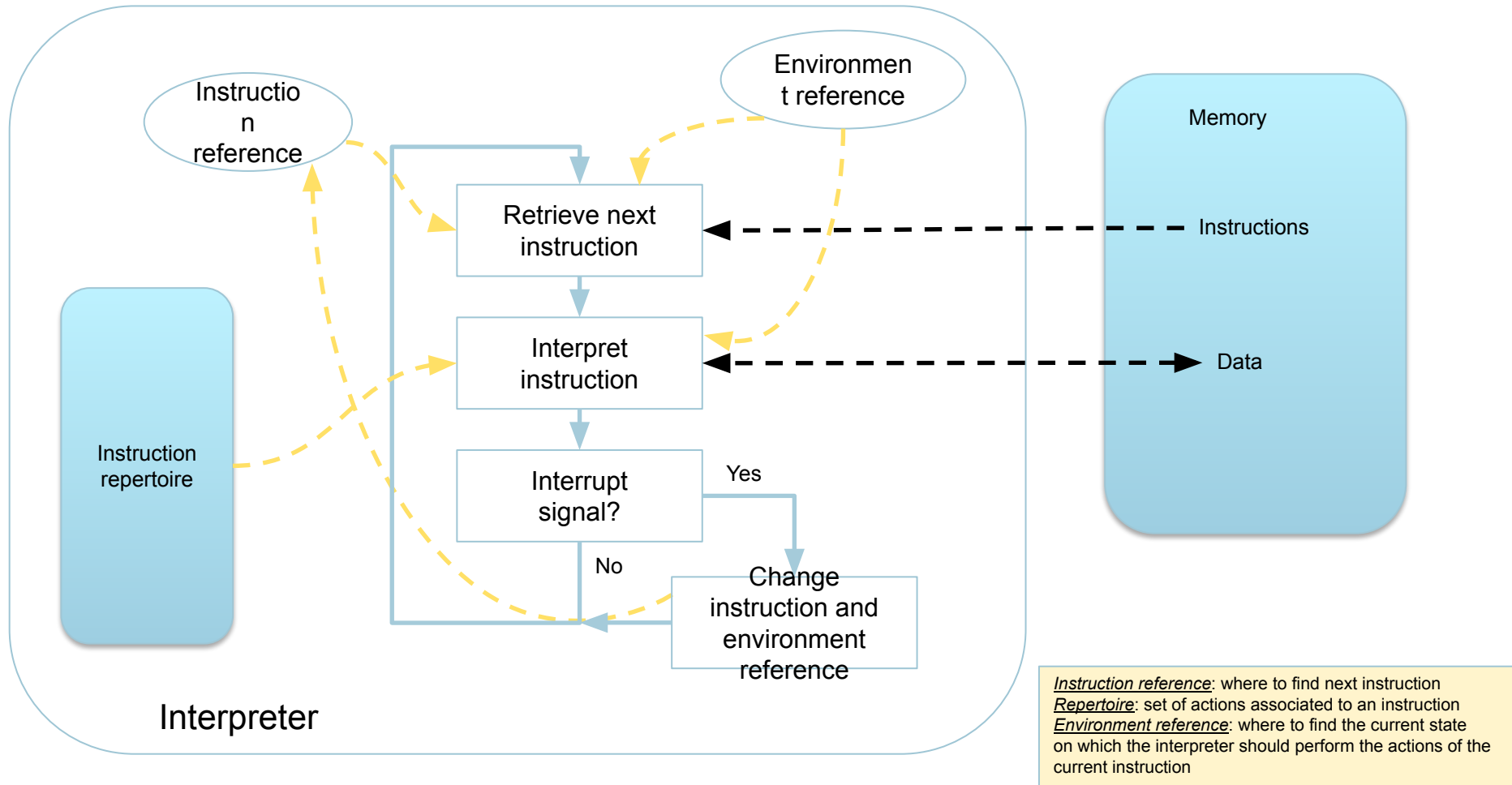
Source: Saltzer and Kaashoek

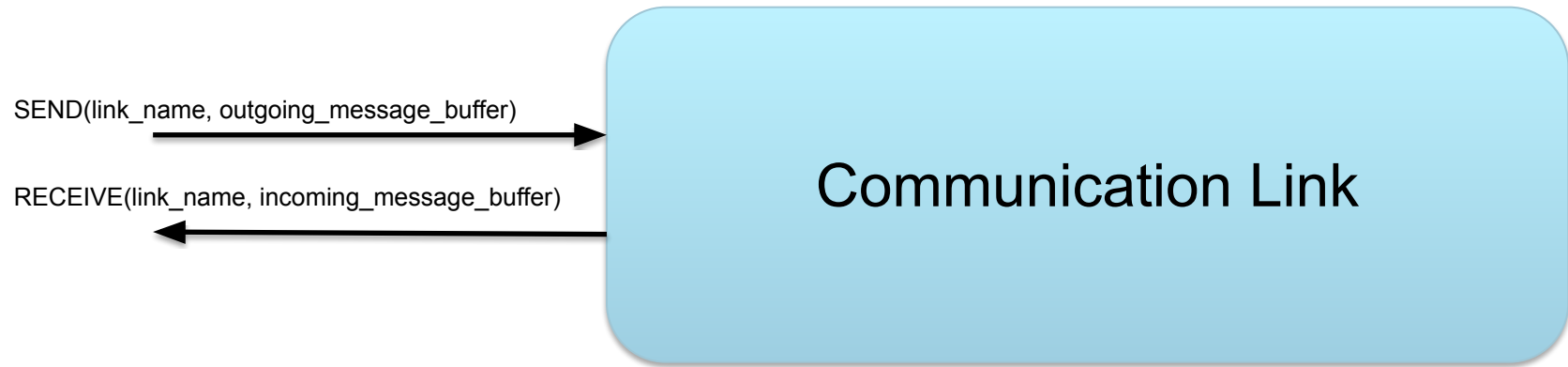


yes,  
memory  
is an abstraction

# Interpreter Abstraction

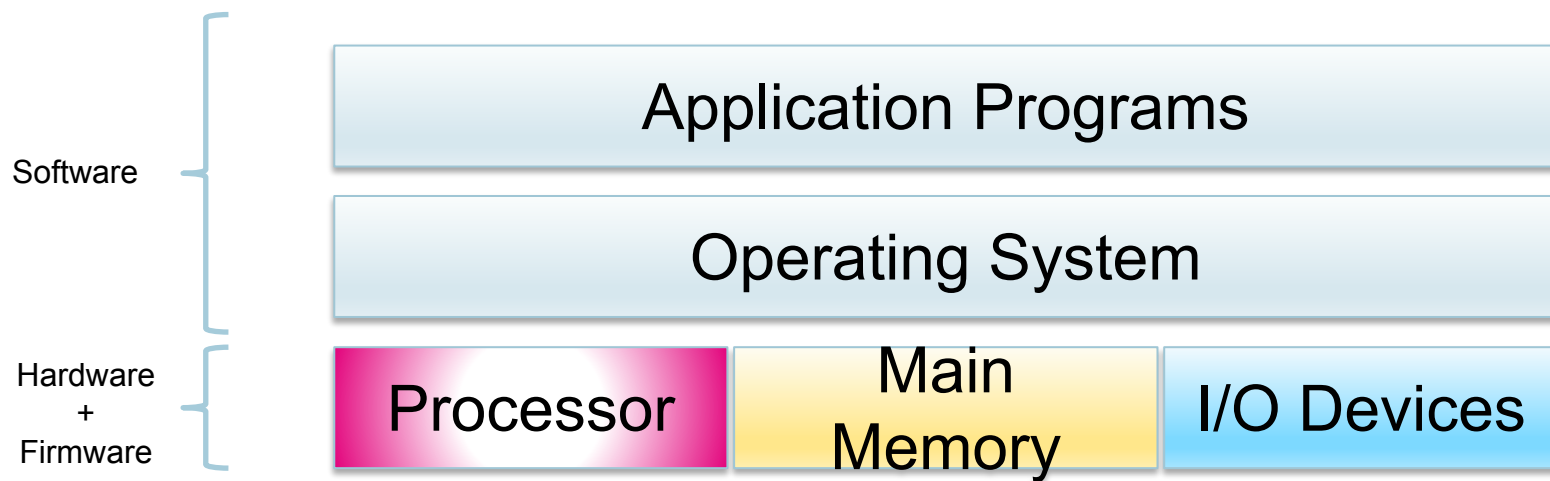
Source: Saltzer and Kaashoek





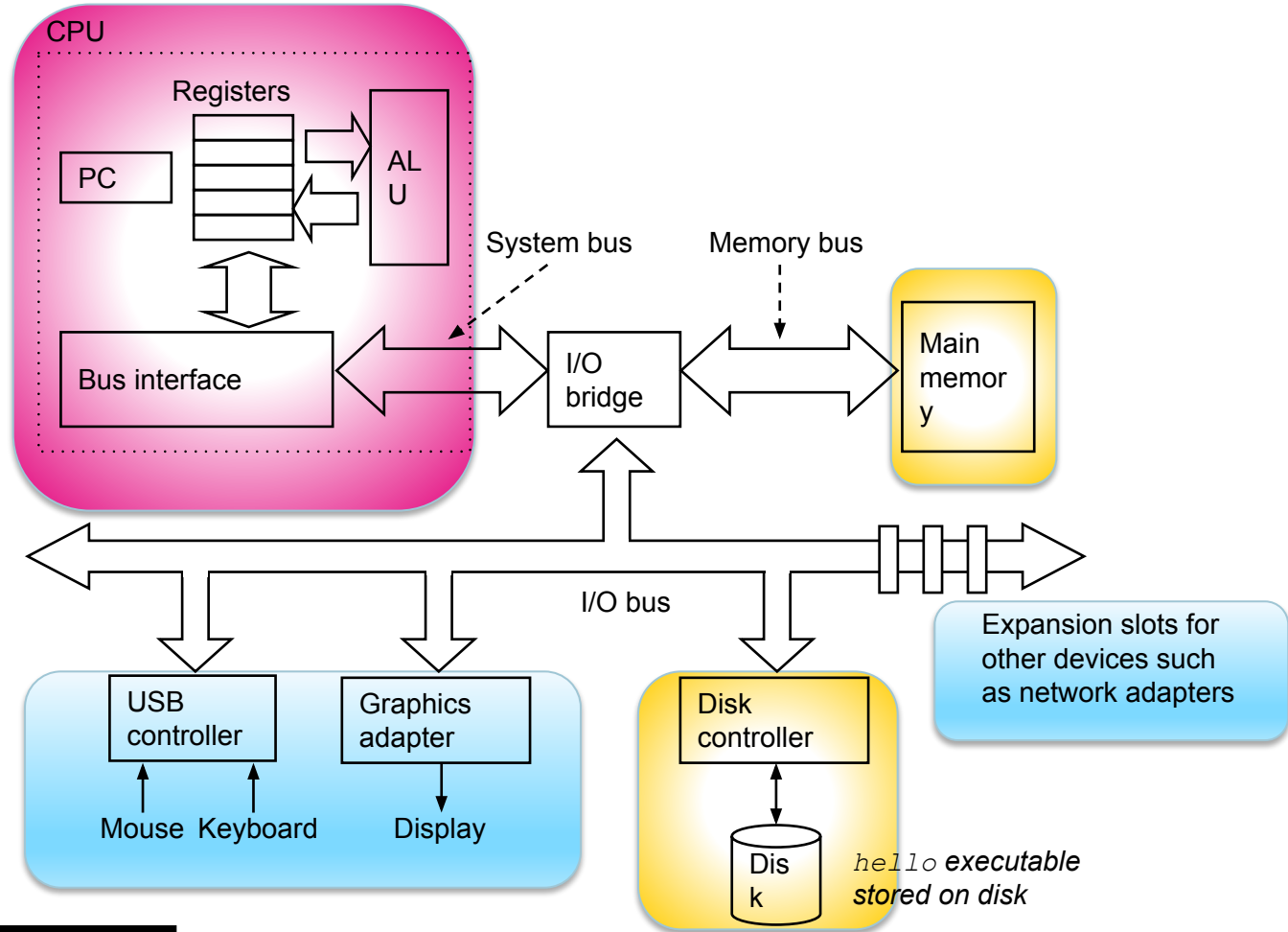


# Layered view of a Computer System

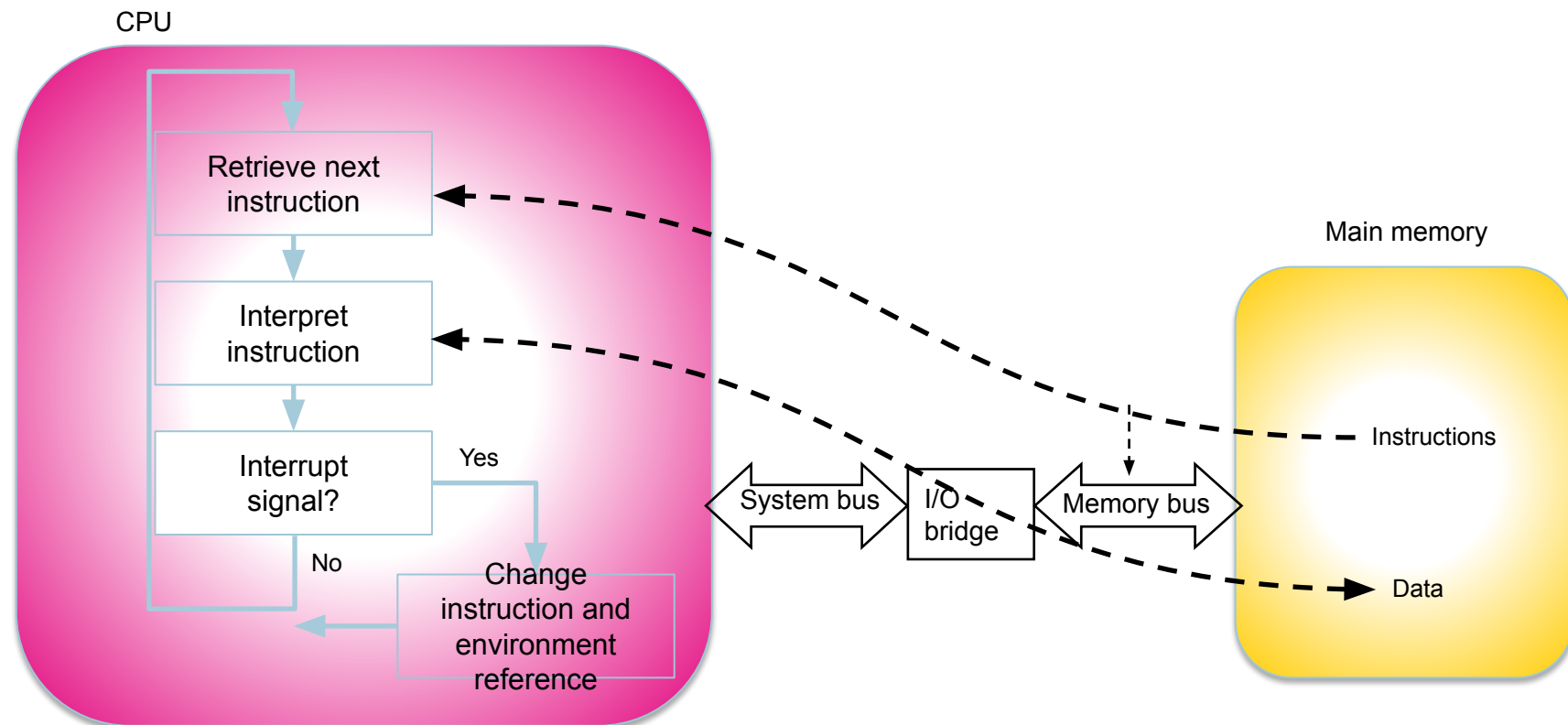


OS is a layer on top of hardware.  
OS manages HW, provides abstractions to apps

# Computer Hardware



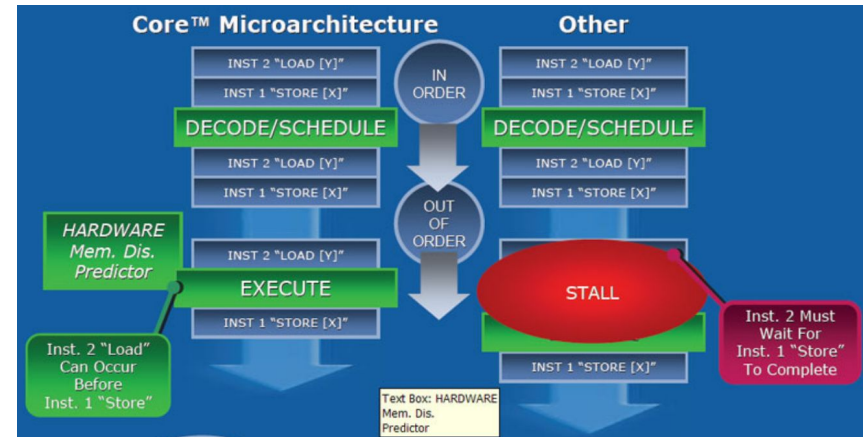
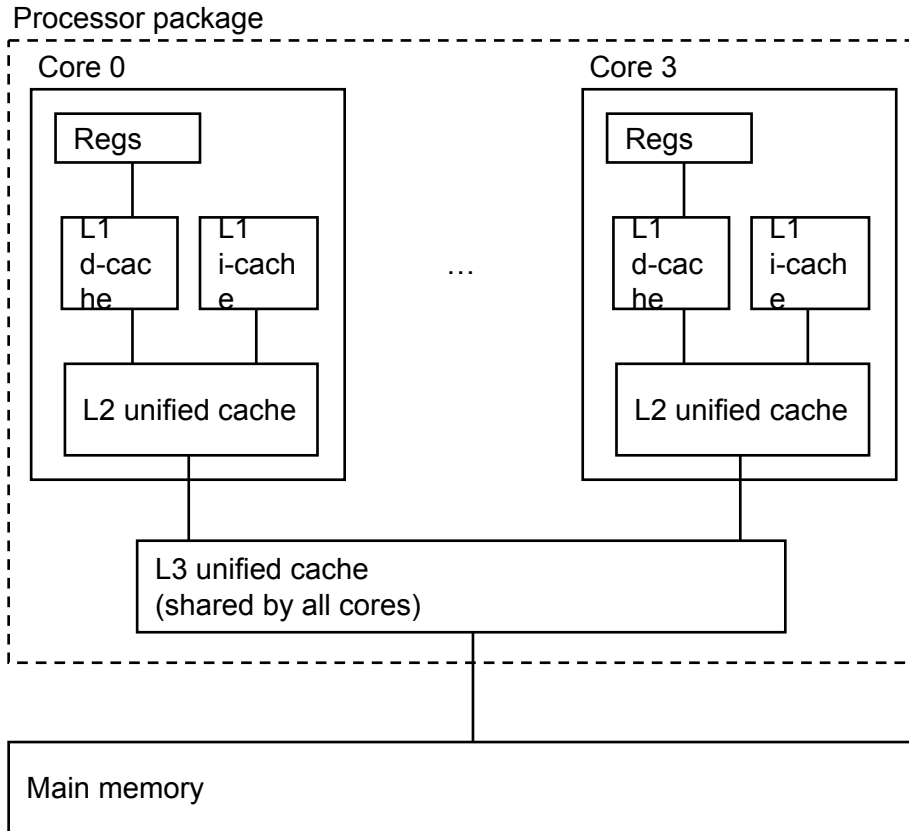
# How does a CPU work?



Instruction repertoire:  
CISC / RISC

A single core CPU can be seen as one interpreter

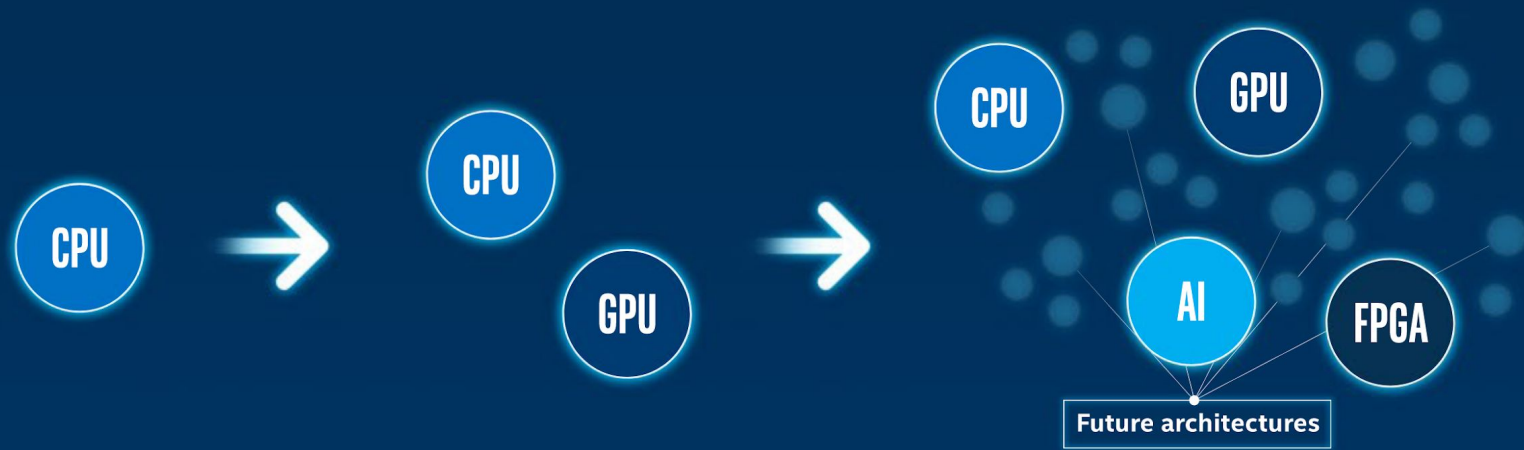
# How does a CPU work?



# The situation is getting more complex

<https://newsroom.intel.com/wp-content/uploads/sites/11/2019/11/intel-oneapi-info.pdf>

As the world's data-centric workloads become more specialized,  
so do the architectures that best process that data.



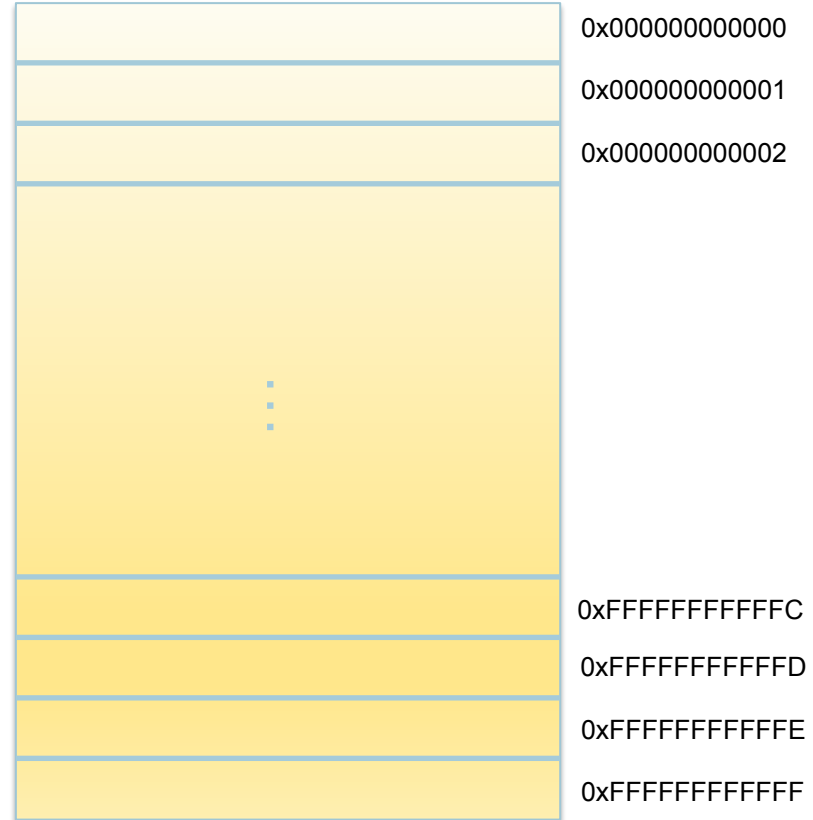
**DIVERSE ARCHITECTURES WILL CONTINUE TO EMERGE AND EVOLVE**

# How does main memory work?

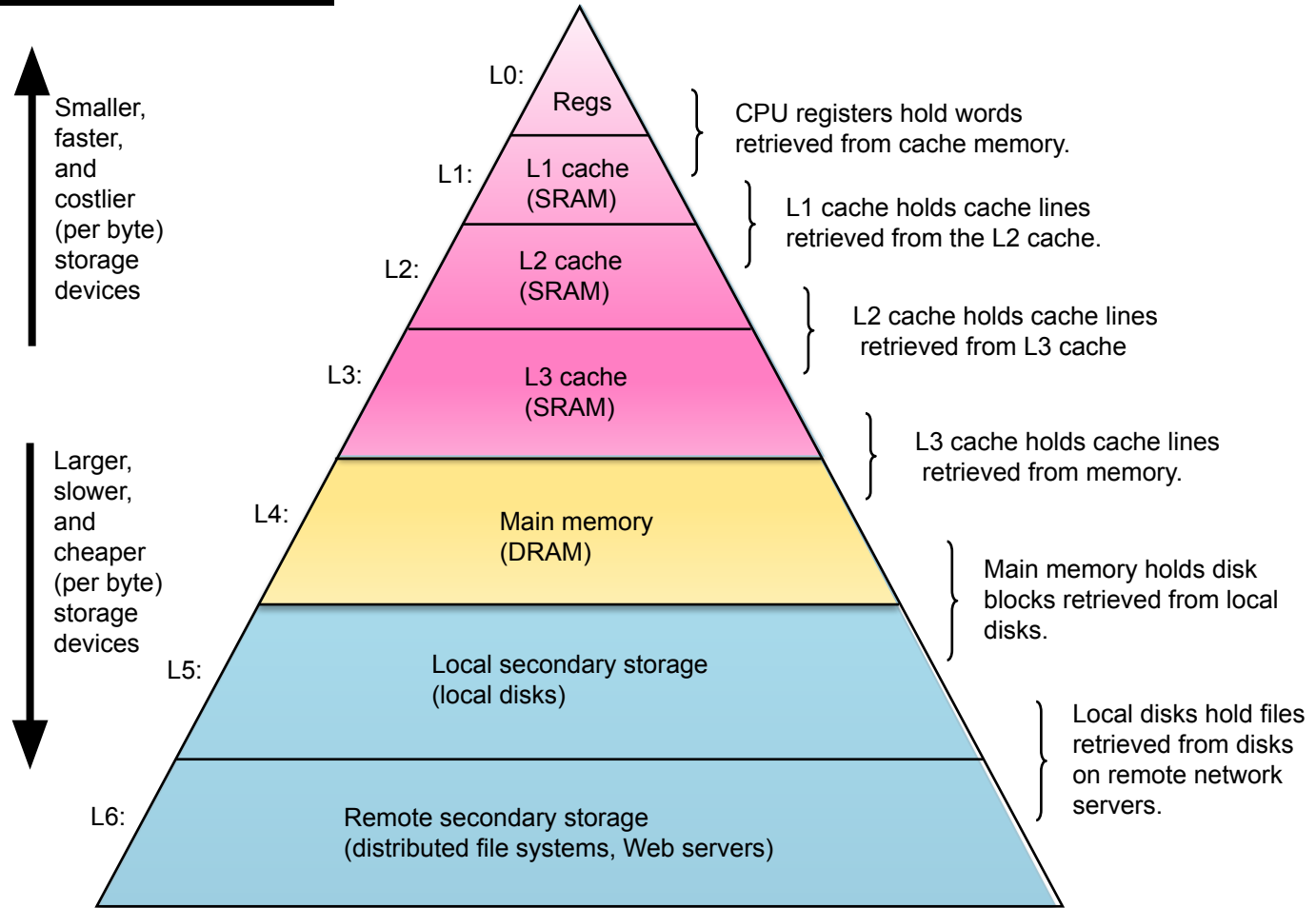
- Main memory is an array of bytes.
- Each byte has a unique address.
- Address space is linear.

## Technology:

- DRAM, SRAM: transient
- 3D Xpoint: persistent



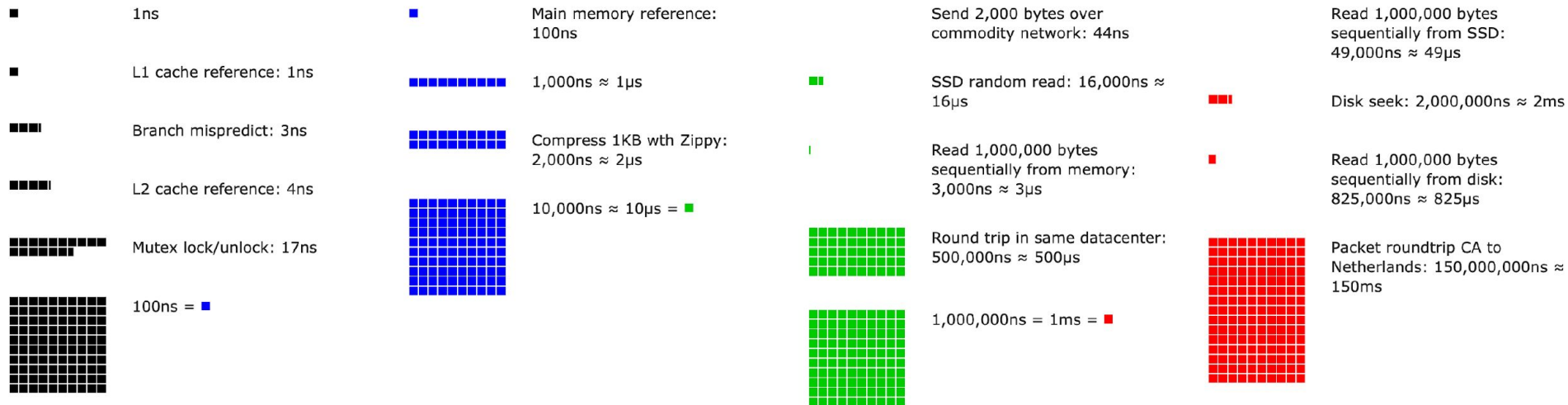
# Memory Hierarchy



# Latency Numbers Every Programmer Should Know

## Latency Numbers Every Programmer Should Know

2020



[https://colin-scott.github.io/personal\\_website/research/interactive\\_latency.html](https://colin-scott.github.io/personal_website/research/interactive_latency.html)

in the last years:  
only SSD have sped up significantly.



# Outline


## Part I: Overview

1. Why this course?
2. What is in it for you?

## Part II: What is this class about?

1. Computer Systems
- 2. Operating Systems**
3. C Programming Language
4. Take-away

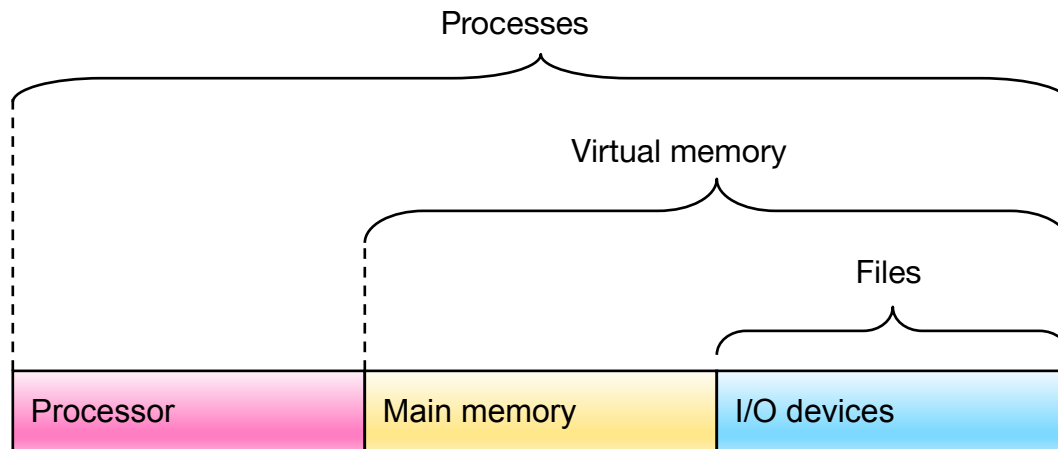
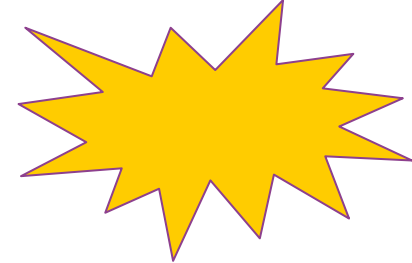
## Part III: Logistics



for now, remember  
fundamental abstractions:

- **interpreter,**
- **memory,**
- **communication**

**An operating system (OS) is a program that manages computer hardware.**



**process**

**virtual memory**

**file**

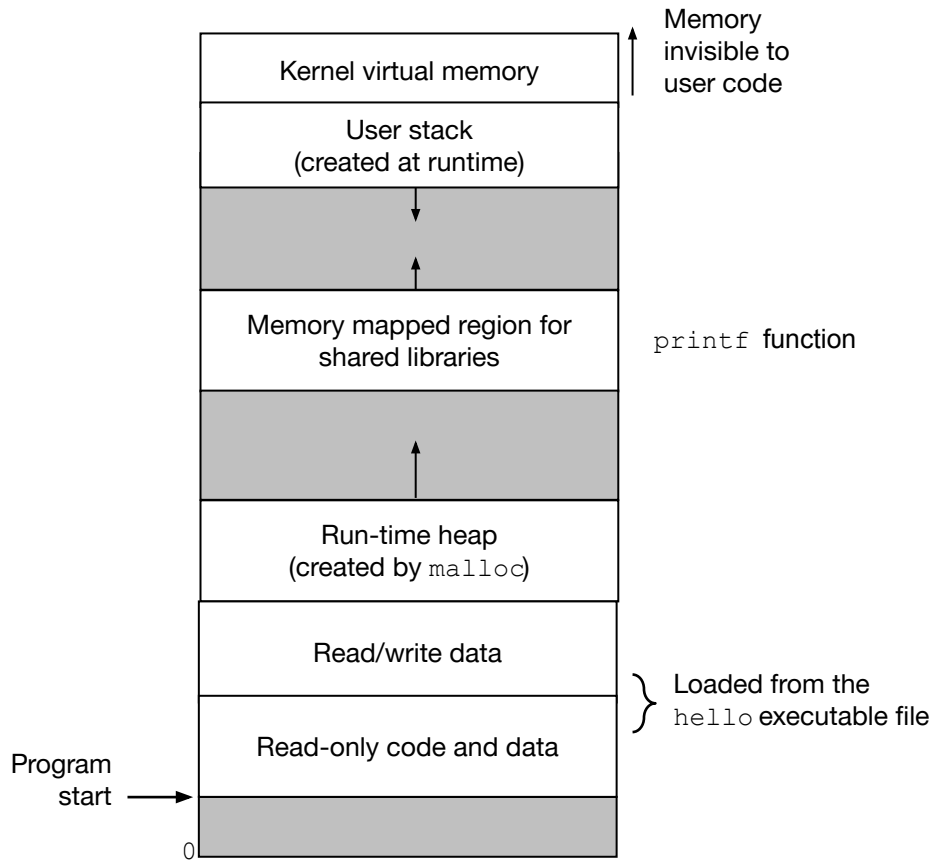
represents processor in HW,

represents main memory in HW,

represents IO devices

- A process:
  - OS Abstraction of a running program
  - An interpreter
- On multi-core CPUs:
  - Multiple processes run simultaneously
- On each core:
  - Multiple processes can execute concurrently.  
They share the same physical core
  - Need to switch from one interpreter to another.

# Virtual Memory



In Linux, files are a universal abstraction for all I/O devices.

A file is an array of bytes.

A file has a unique name (file descriptor).

Basic operations on files are create/delete, open/close, read/write

# Outline

## Part I: Overview

1. Why this course?
2. What is in it for you?

## Part II: What is this class about?

1. Computer Systems
2. Operating Systems
3. **C Programming Language**
4. Take-away

## Part III: Logistics

for now, remember  
Operating System abstractions:

- **process,**
- **virtual memory,**
- **file**

How to write programs that manage computer hardware?

- OS kernel
- Embedded systems
- Infrastructure software that must tightly control its use of hardware resources:
  - Compilers, Database systems, Version control,



# C for system programming

More *portable* than assembly.

*Efficient* enough to give programmers full **control/responsibility** over processes, virtual memory and file abstractions

Alternatives: [Rust](#) (Mozzila), C++

Extensions: OpenCL, OneAPI



# C as a Programming Language

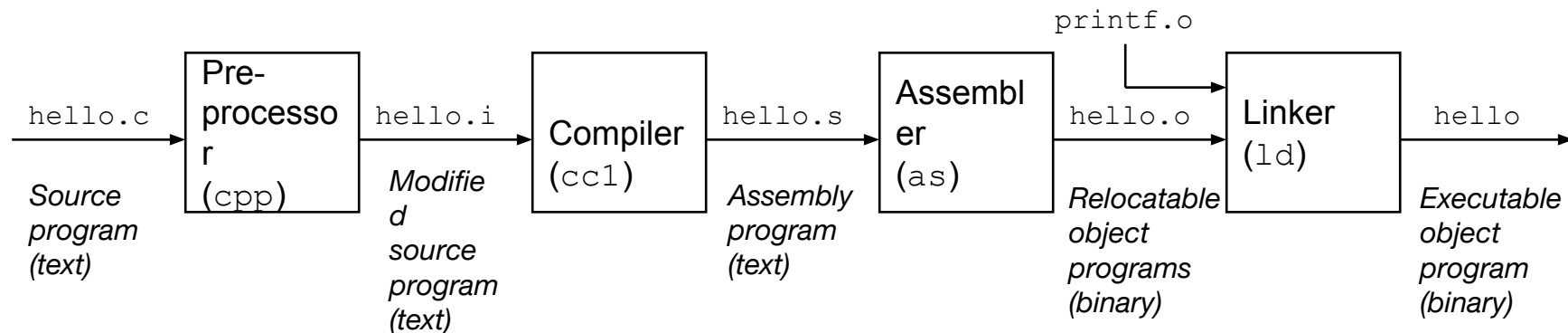
Chapter 7 (specially section 7.5 in Programming Languages Concepts)

C is an imperative programming language.

C is a permissive statically typed language.

# Compilation phases

<https://github.com/gcc-mirror/gcc>



`$ gcc -save-temps hello.c`

**DEMO**

# The C Standard Library

“The standard library provides a variety of functions, a few of which stand out as especially useful.” K&R

“By the way, printf is not part of the C language; there is no input or output defined in C itself. There is nothing magic about printf ; it is just a useful function which is part of the standard library of routines that are normally accessible to C programs.”  
K&R

<http://www.gnu.org/software/libc/manual/pdf/libc.pdf>

<http://ws3.ntcu.edu.tw/ACS099133/cheatsheet/c-libraries-cheatsheet.pdf>

C language itself very minimal. even printing is part of stdio library.  
when learning C, you must be acquainted w/ C library.

# The C Standard Library

Name	From	Description
<assert.h>		Contains the <code>assert</code> macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
<complex.h>	C99	A <a href="#">set of functions</a> for manipulating <a href="#">complex numbers</a> .
<ctype.h>		Defines <a href="#">set of functions</a> used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used <a href="#">character set</a> (typically <a href="#">ASCII</a> or one of its extensions, although implementations utilizing <a href="#">EBCDIC</a> are also known).
<errno.h>		For testing error codes reported by library functions.
<fenv.h>	C99	Defines a <a href="#">set of functions</a> for controlling <a href="#">floating-point</a> environment.
<float.h>		Defines <a href="#">macro constants</a> specifying the implementation-specific properties of the <a href="#">floating-point</a> library.
<inttypes.h>	C99	Defines <a href="#">exact width integer types</a> .
<iso646.h>	NA1	Defines <a href="#">several macros</a> that implement alternative ways to express several standard tokens. For programming in <a href="#">ISO 646</a> variant character sets.
<limits.h>		Defines <a href="#">macro constants</a> specifying the implementation-specific properties of the integer types.
<locale.h>		Defines <a href="#">localization functions</a> .
<math.h>		Defines <a href="#">common mathematical functions</a> .
<setjmp.h>		Declares the macros <code>setjmp</code> and <code>longjmp</code> , which are used for non-local exits.
<signal.h>		Defines <a href="#">signal handling functions</a> .
<stdalign.h>	C11	For querying and specifying the <a href="#">alignment</a> of objects.
<stdarg.h>		For accessing a varying number of arguments passed to functions.
<stdatomic.h>	C11	For <a href="#">atomic operations</a> on data shared between threads.
<stdbool.h>	C99	Defines a boolean data type.
<stddef.h>		Defines <a href="#">several useful types and macros</a> .
<stdint.h>	C99	Defines <a href="#">exact width integer types</a> .
<stdio.h>		Defines <a href="#">core input and output functions</a>
<stdlib.h>		Defines <a href="#">numeric conversion functions</a> , <a href="#">pseudo-random numbers generation functions</a> , <a href="#">memory allocation</a> , <a href="#">process control functions</a>
<stdnoreturn.h>	C11	For specifying non-returning functions.
<string.h>		Defines <a href="#">string handling functions</a> .
<tgmath.h>	C99	Defines <a href="#">type-generic mathematical functions</a> .
<threads.h>	C11	Defines functions for managing multiple <a href="#">Threads</a> as well as <a href="#">mutexes</a> and <a href="#">condition variables</a> .
<time.h>		Defines <a href="#">date and time handling functions</a>
<uchar.h>	C11	Types and functions for manipulating <a href="#">Unicode characters</a> .
<wchar.h>	NA1	Defines <a href="#">wide string handling functions</a> .
<wctype.h>	NA1	Defines <a href="#">set of functions</a> used to classify wide characters by their types or to convert between upper and lower case

“C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments. “

language **trusts** you to **do the right thing** (what needs to be done).  
**trade-off** between **fast** and **reliable/definite/portable**.  
if not careful, you might write programs w/ unintended consequences

- (a) Trust the programmer.*
- (b) Don't prevent the programmer from doing what needs to be done.*
- (c) Keep the language small and simple.*
- (d) Provide only one way to do an operation.*
- (e) Make it fast, even if it is not guaranteed to be portable.*
- (f) Make support for safety and security demonstrable*

“Coding style is all about readability and maintainability using commonly available tools.” L. Torvald

- 1) Indentation
- 2) Breaking long lines
- 3) Placing Braces and Spaces
- 4) Naming
- 5) Typedefs
- 6) Functions
- 7) Centralized exiting of functions [goto considered helpful]
- 8) Commenting
- 9) Function return values and names



# Key Features

Imperative language

Static (but permissive) type checking

Minimal run-time support:

- Explicit memory management
- Explicit threads programming
- Efficient mapping to assembly code

Current standard: **C11**

Unicode support, threads.h, stdatomic.h,  
type generic expressions

Past standards: **C99**, C95, C90, C89

Removed features from K&R C (such as implicit int  
or partial function prototypes). Introduced long,  
variable length arrays, and many library headers.

Future standard: C2X ([charter](#)), planned for 2023 (C23)

Latest version of gcc released July 2020: gcc 11.2

<https://gcc.gnu.org/gcc-11/>

**Q:** what if the standard does not define a behavior?

**A:** then it's up to the compiler writer. (anything could happen. **ex:**)

# Undefined Behavior

“In a *safe* programming language, errors are trapped as they happen. Java, for example, is largely safe via its exception system. In an *unsafe* programming language, errors are not trapped. (...)

[In C], anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.”

John Regehr

<https://blog.regehr.org/archives/213>

<http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>

# Undefined Behavior

```
#include <limits.h>
#include <stdio.h>

int main (void)
{
    printf ("%d\n", (INT_MAX+1) < 0);
    return 0;
}
```

What happens if we add 1 to the largest integer?  
This is undefined behavior.

<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html>

^-- core undefined behavior

# Outline

## Part I: Overview

1. Why this course?
2. What is in it for you?

## Part II: What is this class about?

1. Computer Systems
2. Operating Systems
3. C Programming Language
4. Take-away

## Part III: Logistics

# Take-aways

You will learn how the hardware infrastructure impacts software with a focus on either **performance** or **security**.

We will cover in details programming issues related to the **three fundamental abstractions** provided by operating systems:

- Processes are interpreters
- Memory is an array of bytes
- I/O devices are seen as files

# Take-aways

4 compilation phases: preprocessing, compiler, assembler, linker

The C standard library contains collections of useful functions

The C standard creates undefined behaviours. Beware!

# Outline

## Part I: Overview

1. Why this course?
2. What is in it for you?

## Part II: What is this class about?

1. Computer Systems
2. Operating Systems
3. C Programming Language
4. Take-away

## Part III: Logistics



1. (learnit;) github.itu.dk; slack [show]
2. textbooks: CS:APP, LCTHW
3. lectures (2hr), exercises (2hr)
4. assignments (next slide)
5. exam (take-home, based on assignments)

## SWU, SD\*

- 3 assignments:
  - datalab
  - perflab | attacklab
  - malloclab
- Exam: 4 questions – 25% each (datalab, perflab | attacklab, malloclab, topics from the class)

two tracks!

hardest so far

\*: **SD** (a master program) has a higher passing criteria on the assignments.

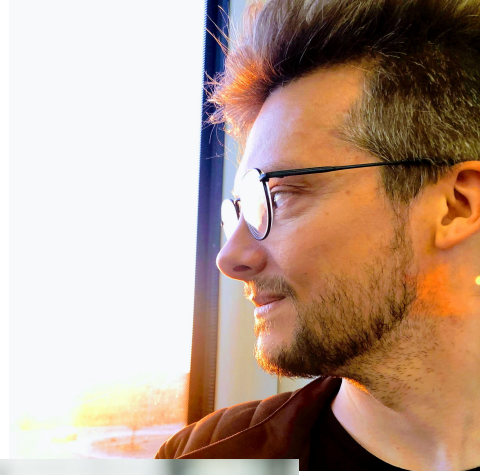
## DS

- 2 assignments:
  - datalab
  - perflab
- Exam: 3 questions – 33% each (datalab, perflab, topics from class)

don't underestimate the labs!

# Logistics - We

- [Willard Rafnsson](#): course responsible
- [Niclas Hedam](#): head-TA (PhD, SWU)
- [Alexander Berg](#): TA (CS, SWU)
- [Mikkel Lippert](#): TA (SWU)
- [Noah Brunken Syrkis](#): TA (DS)
- [Viktor Bello Thomsen](#): TA (DS)



communication policy:  
**no e-mails**

