Duke Systems

CPS 310 / ECE 353

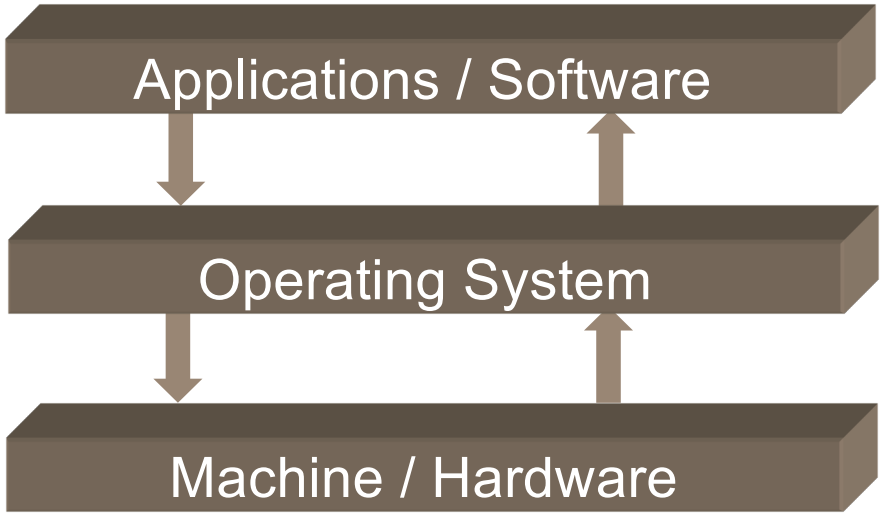
Welcome to the Machine

Jeff Chase

Duke University

Hardware and software

- OS is “all the code you don’t have to write” to create and run useful software programs on a computer.

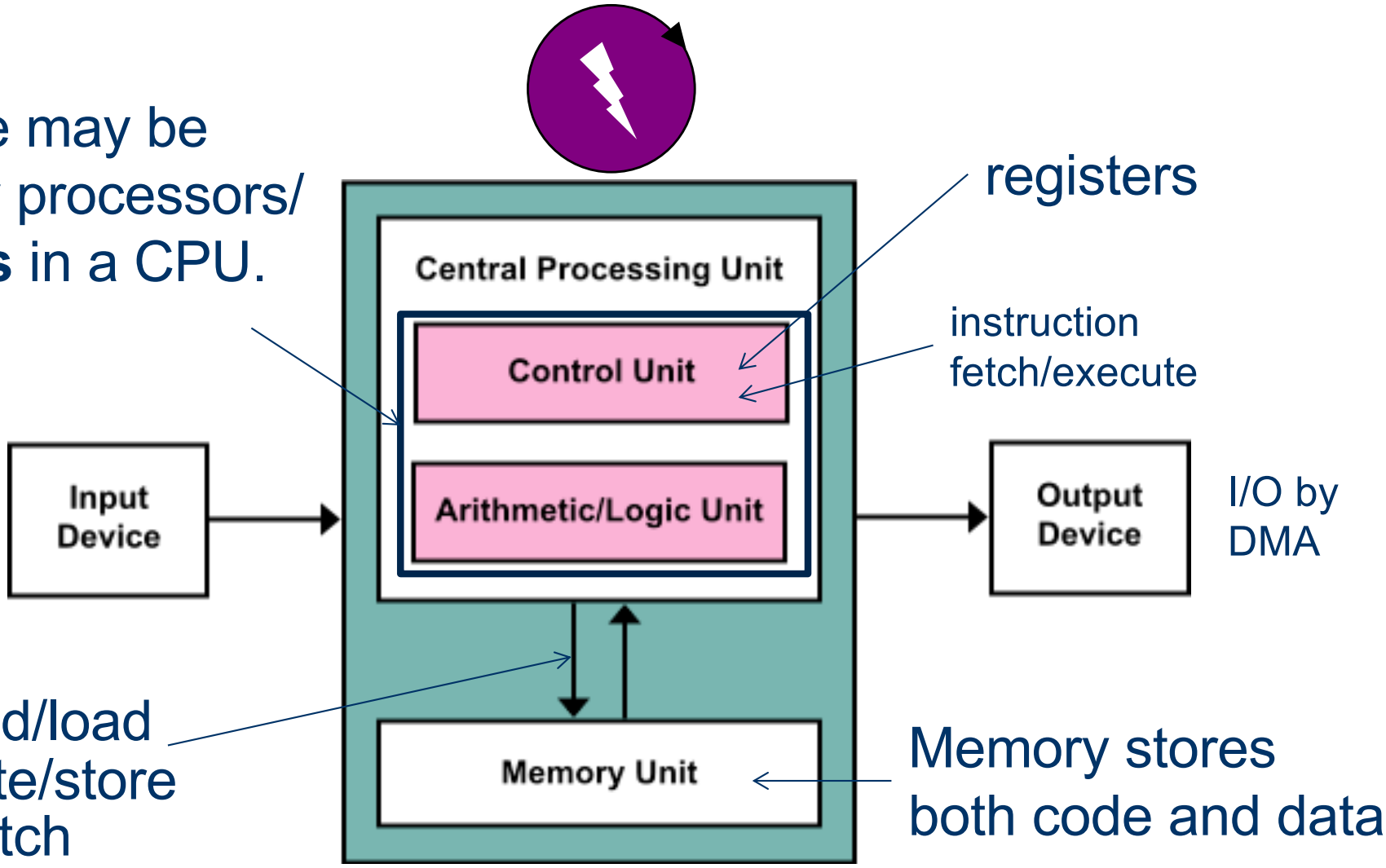


- Software is code.
- Code runs on hardware.
- Hardware executes code.

- OS mediates access to the machine and protects programs and users from one another.

Von Neumann Architecture

There may be many processors/ **cores** in a CPU.



https://commons.wikimedia.org/wiki/File:Von_Neumann_Architecture.svg


78014
78024
78034
78044
78054
78064
78015
78025
78035
78045
78055
78065
78016
78026
78036
78046
78056
78066
78076
78017

Memory/storage is an abstraction

- Storage: an array of locations.
- Locations are numbered: each has an address.
- Each location stores a data value.

WRITE(*addr*, *value*) $\rightarrow \varnothing$ Store *value* in the storage cell identified by *addr*.
READ(*addr*) $\rightarrow$ *value* Return the *value* argument to the most recent WRITE call referencing *addr*.

Figure 5: The memory abstraction

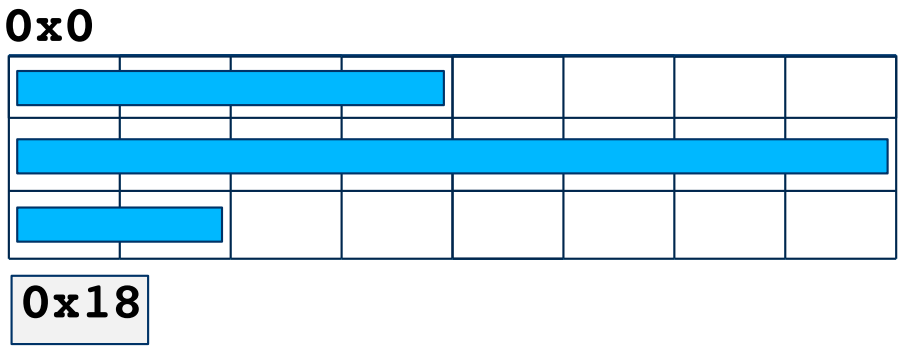
Graphic from **Intel SGX Explained** (Costan and Devadas, IACR Cryptology, 2016)

Data types

Basic data items often span multiple contiguous locations/cells.
On a **typical** 64-bit system: a char is one byte (8 bits), an address (pointer) is 64 bits, an integer is 32 bits, and “long” integer is 64 bits.
Details vary.

```
struct stuff {
  int i;
  long j;
  char c[2];
};
```

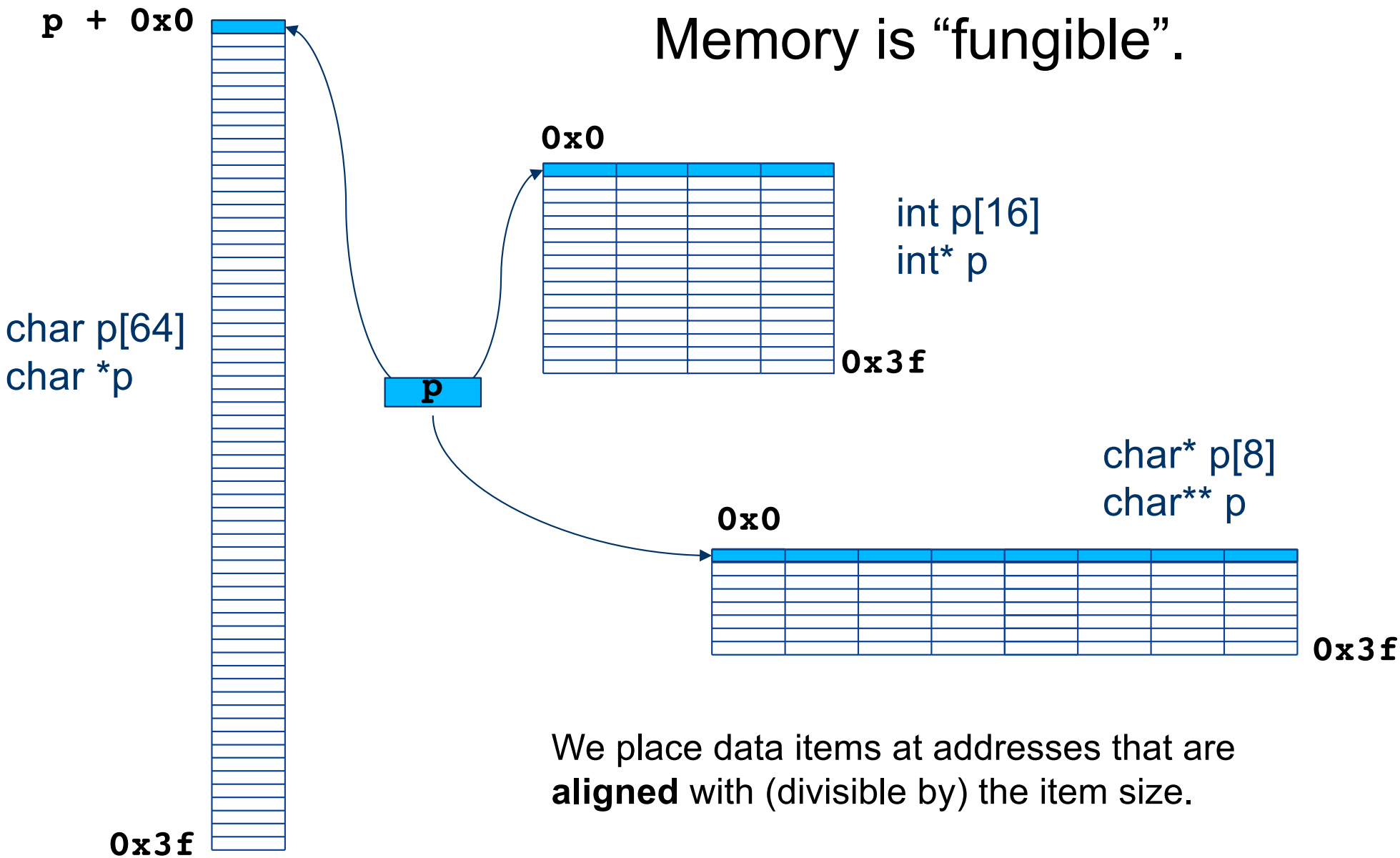
24 bytes (=0x18)



Data model	short (integer)	int	long (integer)	long long	pointers/ size_t	Sample operating systems
LLP64/ IL32P64	16	32	32	64	64	Microsoft Windows (x86-64 and IA-64)
LP64/ I32LP64	16	32	64	64	64	Most Unix and Unix-like systems, e.g. Solaris, Linux, BSD, and OS X; z/OS

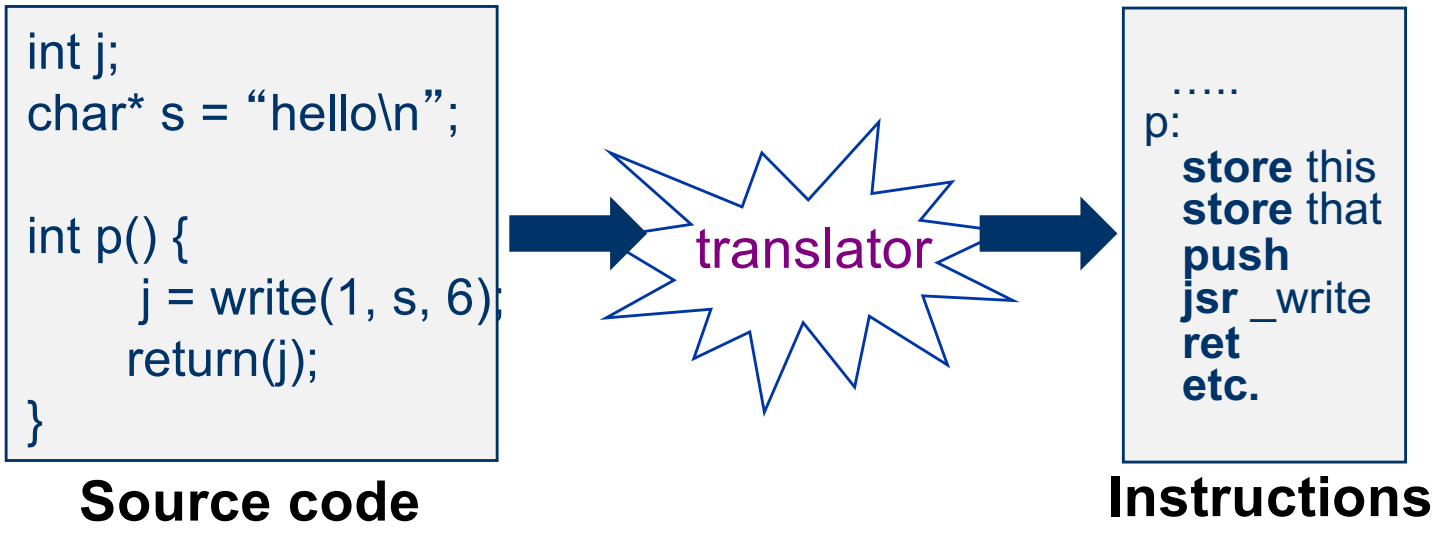
64 bytes: 3 ways

Memory is “fungible”.



We place data items at addresses that are **aligned** with (divisible by) the item size.

Code

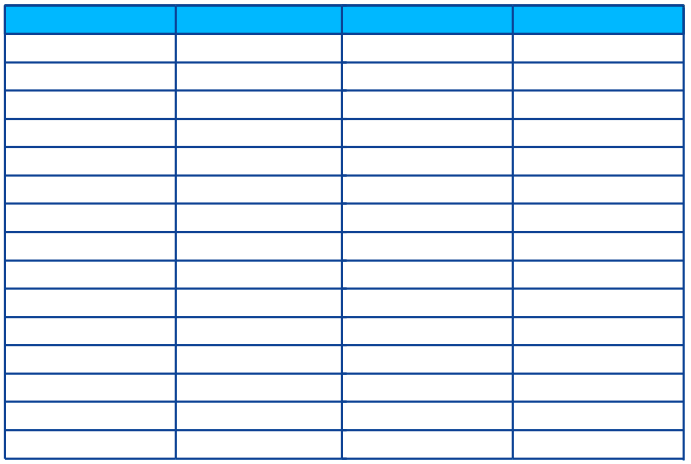


- Code has many representations.
- Many tools/programs transform from one to another.
- Each language has tools and steps to translate and combine subprograms and prepare them for execution.
- Compilers, interpreters, assemblers, linkers, etc.

Code: instructions in memory

```
_procedure1:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $1, %eax
        movq    %rdi, -8(%rbp)
        popq    %rbp
        ret
```

Code is data!

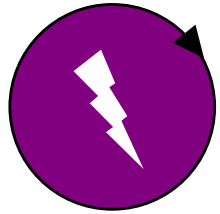


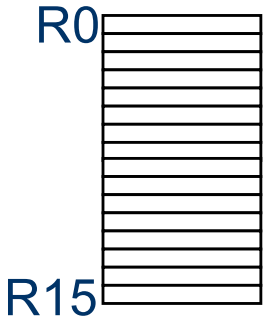
Here is a sequence of x86-64 machine instructions in a human-readable (**assembly**) representation. Details depend on machine's instruction set architecture (ISA).

Each instruction has an **opcode** and zero or more **operands**.
The CPU fetches the instruction from memory at some address.

Code and data locations correspond to **symbols** in the source program (e.g., procedure1).

Registers



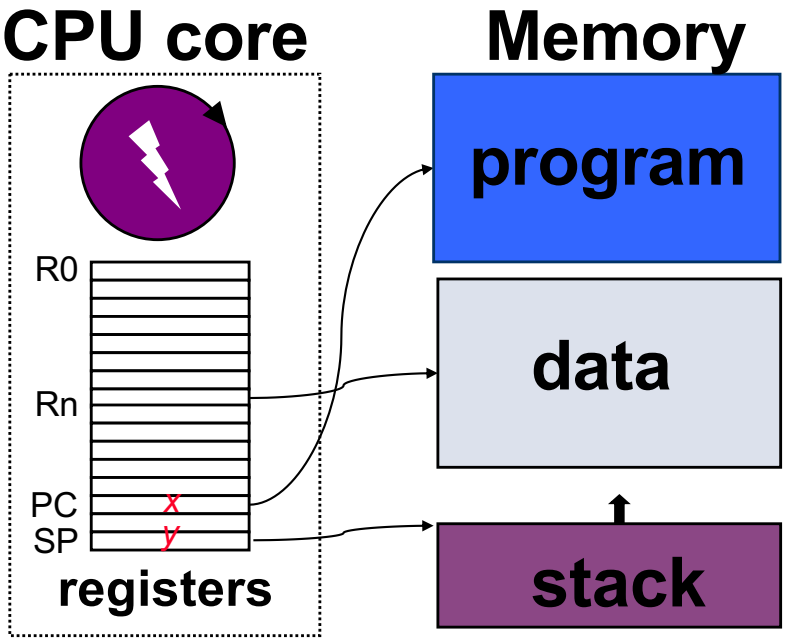


- Registers are fast memory inside the CPU/core.
- Instructions may operate on registers by name.
- How fast? Access at CPU clock speed.
- Arithmetic instructions have register operands.
- Load/store/branch instructions have address operands.
- Register naming/layout/size is processor-dependent.

```
load    x, R2          ; load global variable x
add     R2, 1, R2      ; increment: x = x + 1
store   R2, x          ; store global variable x
```

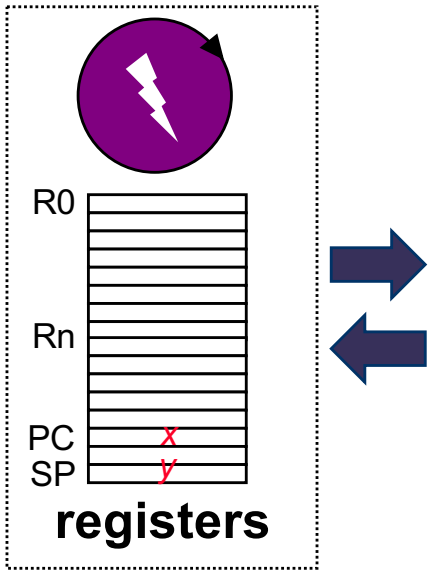
Code stream

- CPU executes a sequence or **stream** of instructions.
  - PC (program counter) or IP (instruction pointer) register →address of the current/next instruction to execute.
  - CPU repeats: fetch; execute; increment PC; loop
  - Branch instructions change PC.
  - E.g., call/return: control flow.
- A stream runs with a **stack**.
  - RAM region for its scratch use
  - SP (stack pointer) register →address of stack top

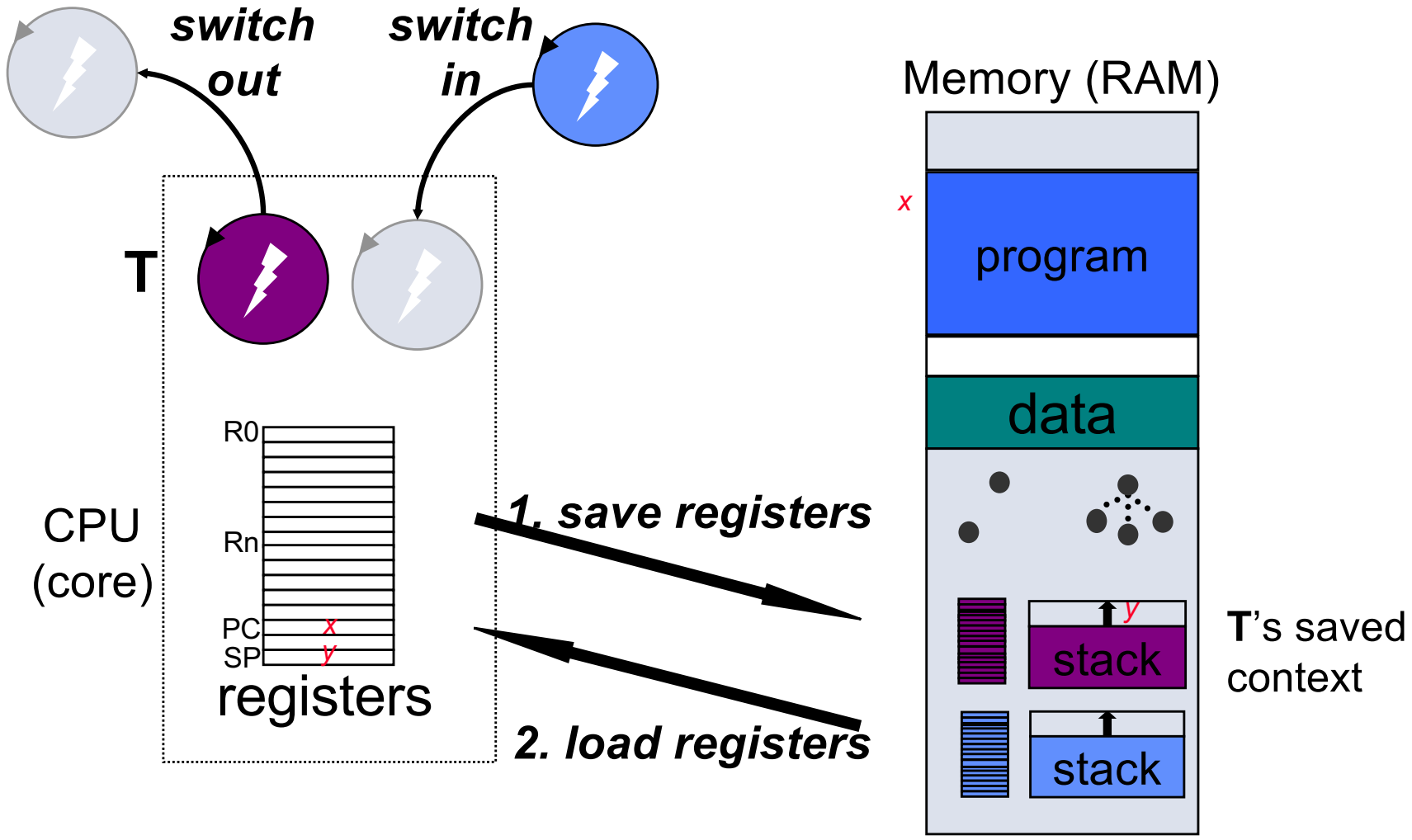


Threads and contexts

- Register values contain the **context** of a code stream.
  - Including any pointers to memory it uses (e.g., its stack).
- **Contexts are data objects.**
- **Context switch**:
  - Stream executes instructions to:
  - Take a snapshot of the register values.
  - Save snapshot (context) in memory.
  - Load registers with another context.
  - Switch to another stream!
  - OS may manipulate saved contexts.
- We call them **threads**.

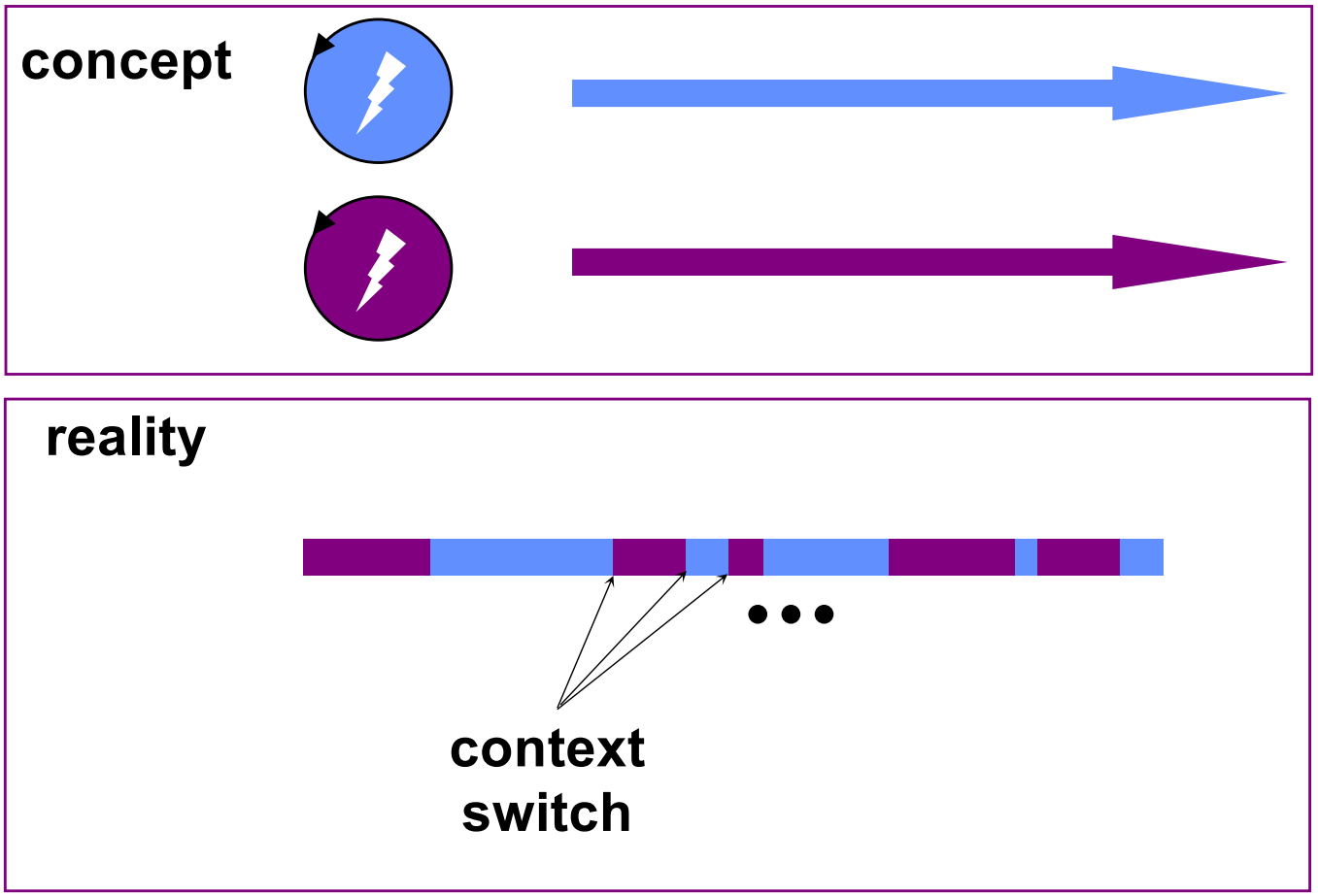


Thread context switch



Running code can suspend current thread **T** by storing its register values in memory. Load them back to resume **T** at any time.

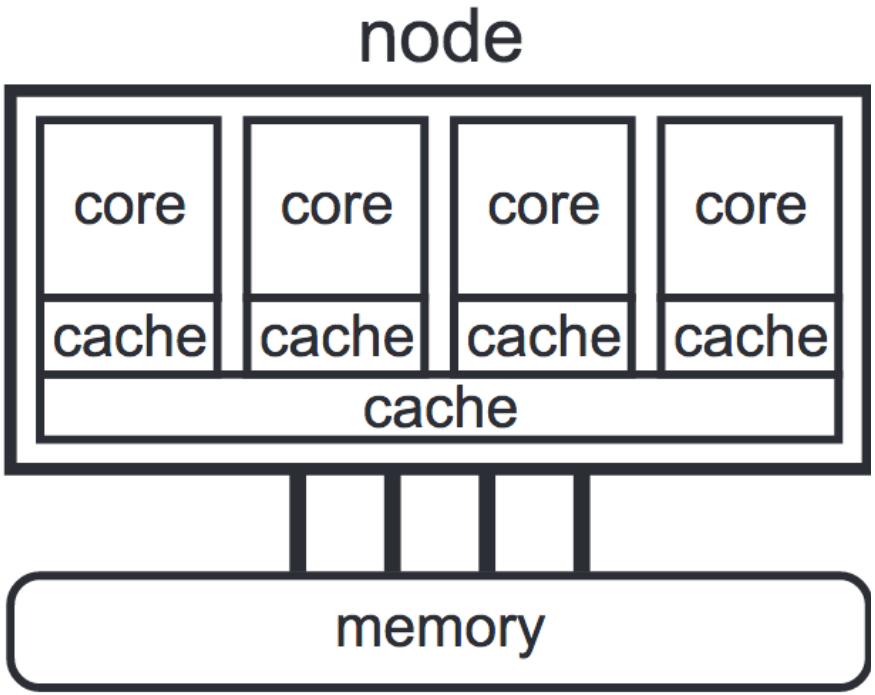
Two threads sharing a CPU/core

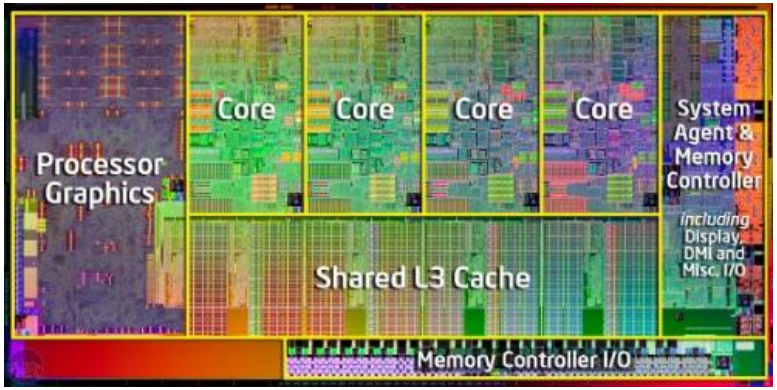


OS dispatches a thread to run on each core…

…and interleaves threads on the same core.

Modern computers are multi-core



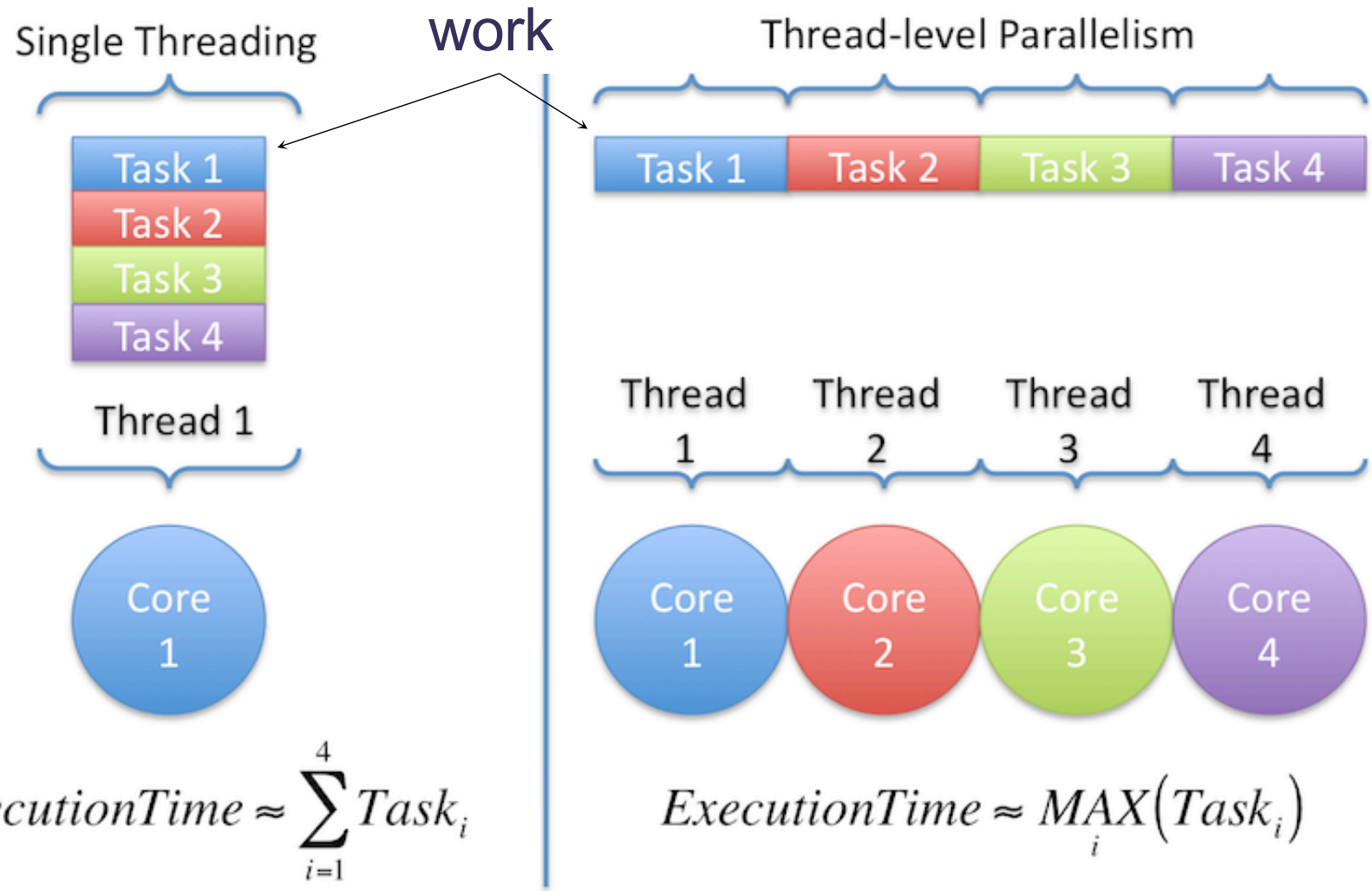


Intel Xeon Platinum "Cooper Lake" (2020) has 28 cores @ two slots each.

- A core with R register sets can run R threads in parallel.
- Core has R **slots —** logical processors or hardware threads.
- For simplicity, we suppose R=1: treat each slot as a separate core.

Left graphic from: **How to implement any concurrent data structure for modern servers**.
By Calciu, Sen, Balakrishnan, Aguilera. CACM 2020.

Multithreading for parallelism



"Because it's faster."

http://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore

Reading Between the Lines of C

```
load    x, R2          ; load global variable x
add     R2, 1, R2      ; increment: x = x + 1
store   R2, x          ; store global variable x
```

Two threads execute this code section. **x** is a shared variable.

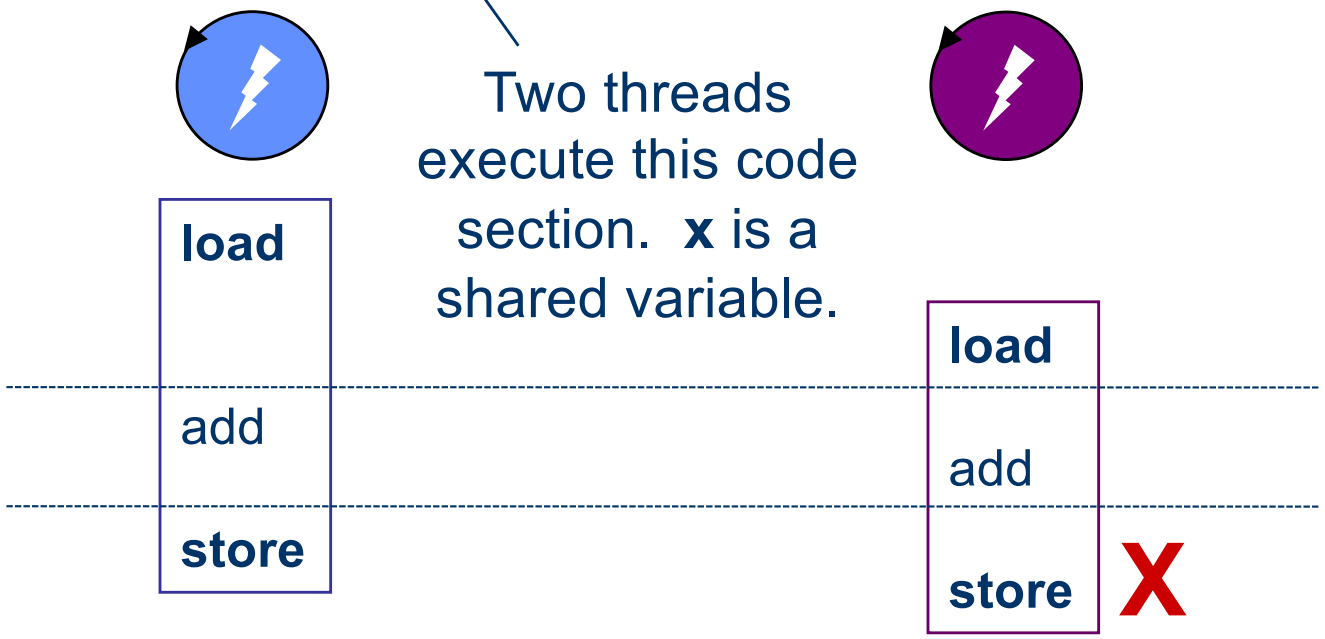
```
load
add
store
```

```
load
add
store
```

Two executions of this code, so:
x is incremented by two.

Concurrency
Interleaving matters

```
load    x, R2          ; load global variable x
add     R2, 1, R2      ; increment: x = x + 1
store   R2, x          ; store global variable x
```



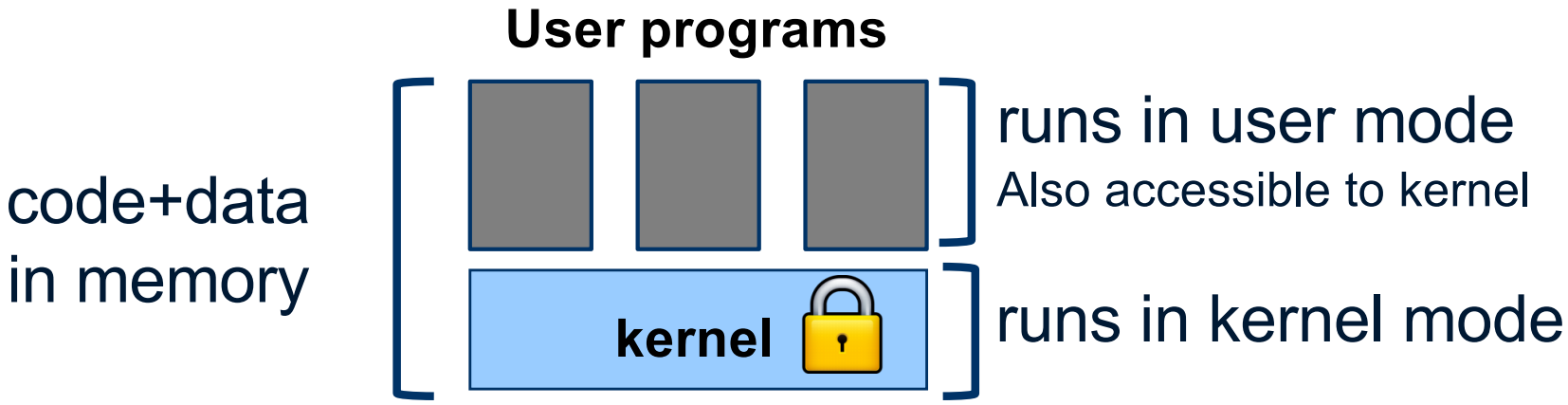
In this schedule, **x** is incremented only once: last writer wins.
The program breaks under this schedule. This bug is a **race**.

Protection: kernel mode

There is a switch in the processor (core) that gives it special powers.

- We call it **mode**.
- When a core is in **kernel mode** it can execute privileged instructions.
- Else the core is in **user mode:** these functions are disabled.
- **How to turn it on?**

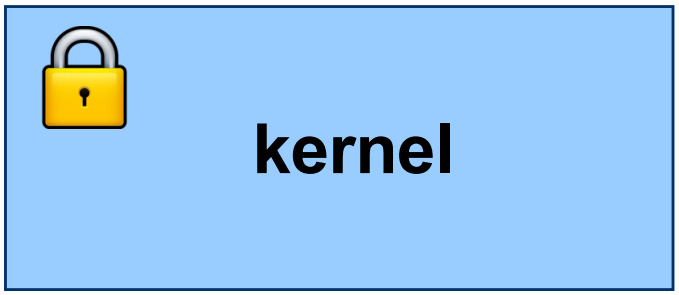
Trusted OS code runs in kernel mode



- User programs reside in memory along with the OS.
- But their code runs at different privilege levels.
- Kernel mode enables OS code to control the machine.
- User programs “cannot” access OS kernel memory.

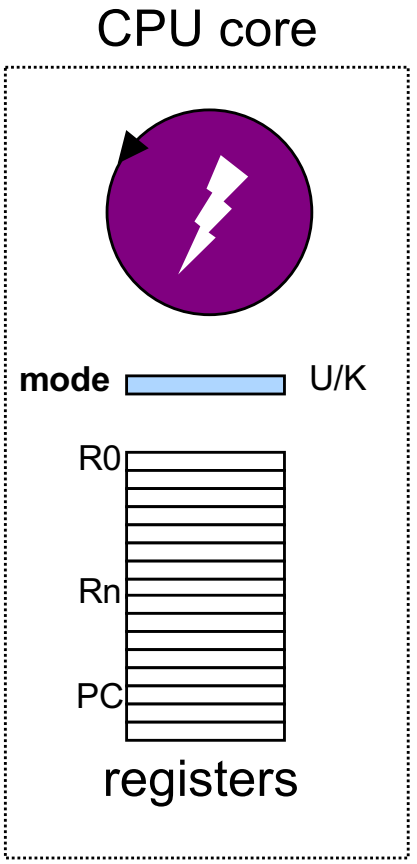
The kernel

- The OS **kernel** is a program that resides in a file.
- On power-on/reset, the machine starts a bootloader program.
- That loads the kernel into memory and transfers control to it (**boot**).
- Once running, the kernel initializes certain machine functions.
- Then the kernel can launch a user program: set up a context and execute a special instruction to switch into the context in user mode.
- Control transfers back to the kernel to handle CPU **exceptions**.
- Once booted, the kernel acts as one big event handler.



Mode: CPU privilege level

The current **mode** of a CPU core is given by a field (e.g., one bit) in a protected control register.



If the core is in **kernel mode** then it can:

- access memory reserved to kernel;
- access any machine memory;
- access certain **control registers;**
- command I/O devices;
- execute certain special instructions;
- **control the entire machine.**

If code running in **user mode** attempts such a privileged operation:

- core raises a **fault**: a CPU exception;
- transitions to **handler** code in kernel mode.

Kernel mode

What turns it on? **Exceptions**

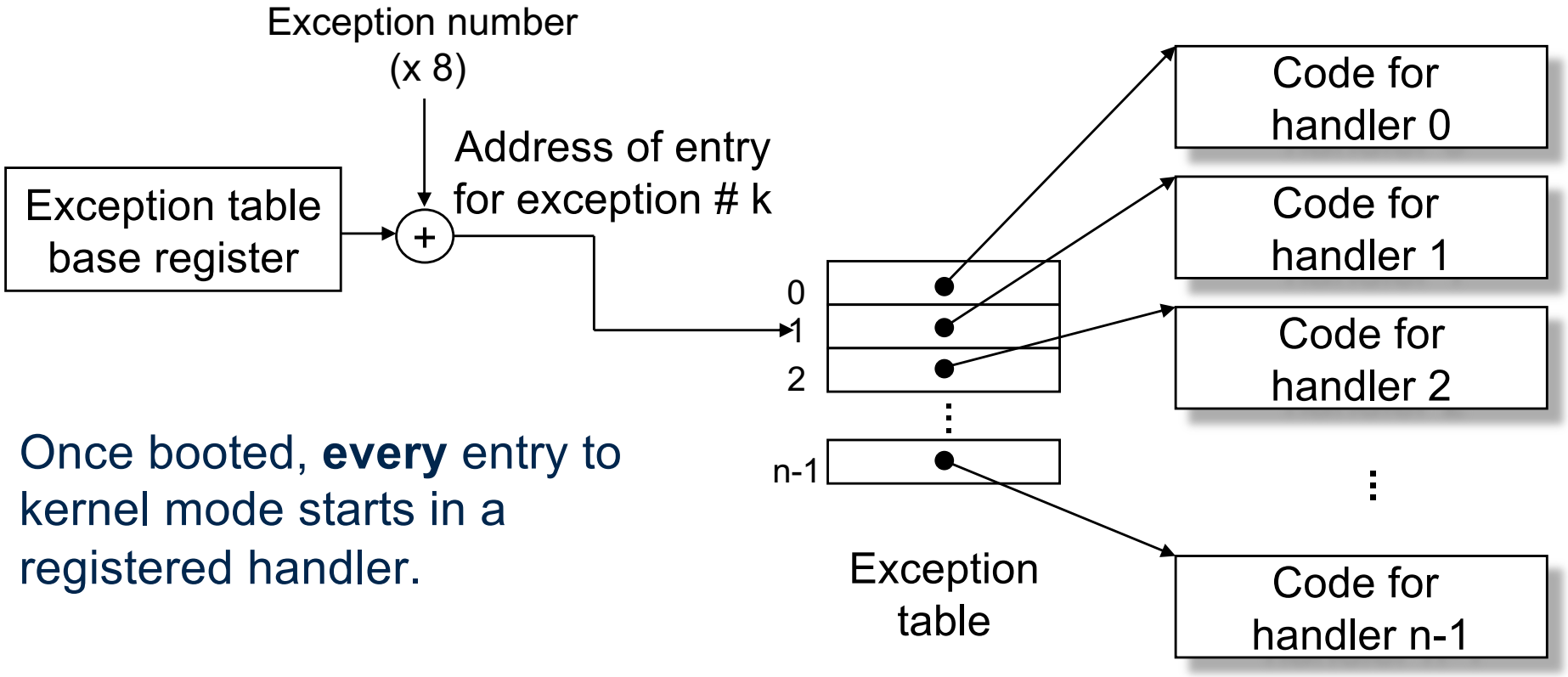
- **Trap**: system call from user code
- **Fault**: CPU needs kernel help
- **Interrupt**: device needs attention

What turns it off?

- Kernel code executes special instruction to enter user mode.
- Kernel decides where to enter.
- E.g., return to saved user context.

Protecting entry to the kernel

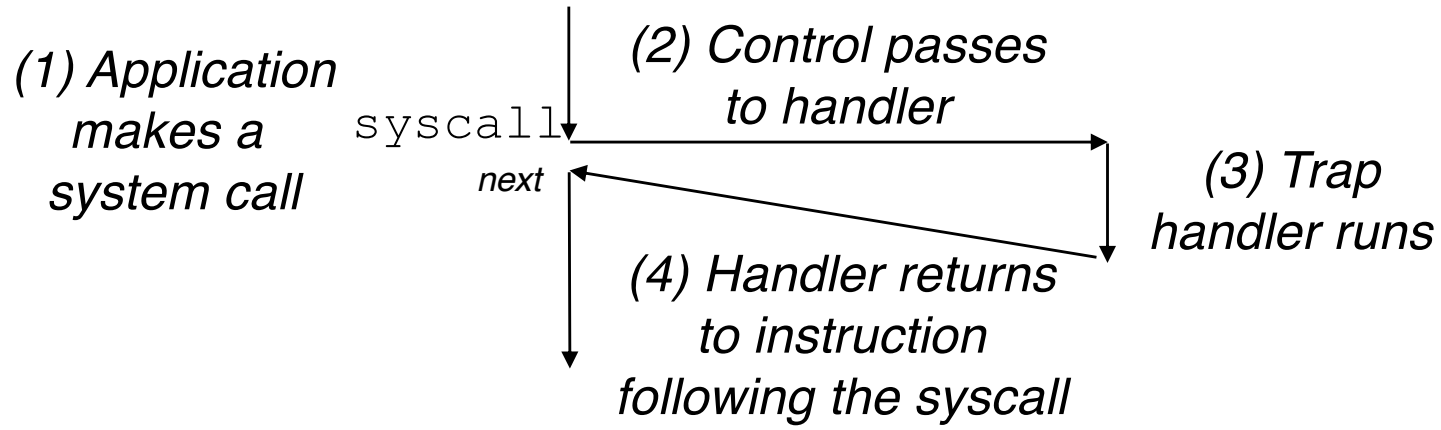
- Kernel sets up a **vector table** in memory at boot time.
- It writes the table address into a special protected control register.
- Table entries point to handlers for each type of exception/interrupt.



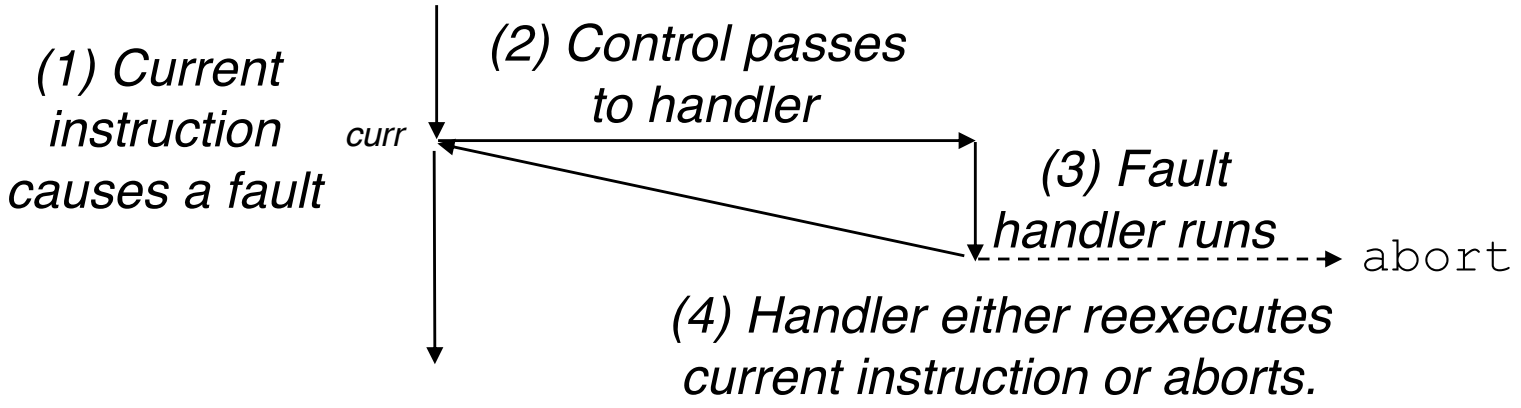
Once booted, **every** entry to kernel mode starts in a registered handler.

[from CS:APP3e F8:2-3 combined/edited]

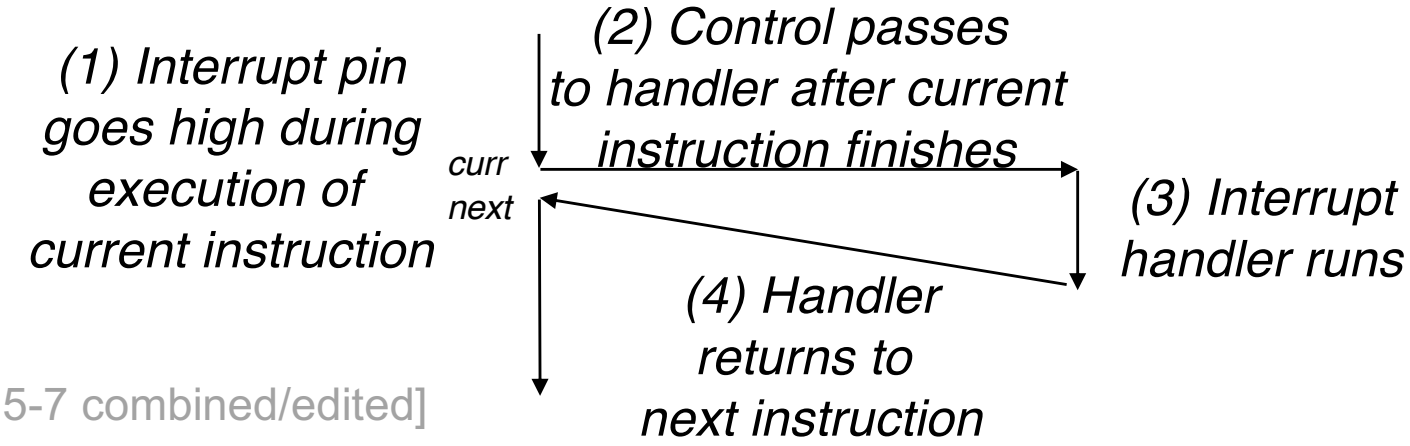
Trap



Fault



Interrupt



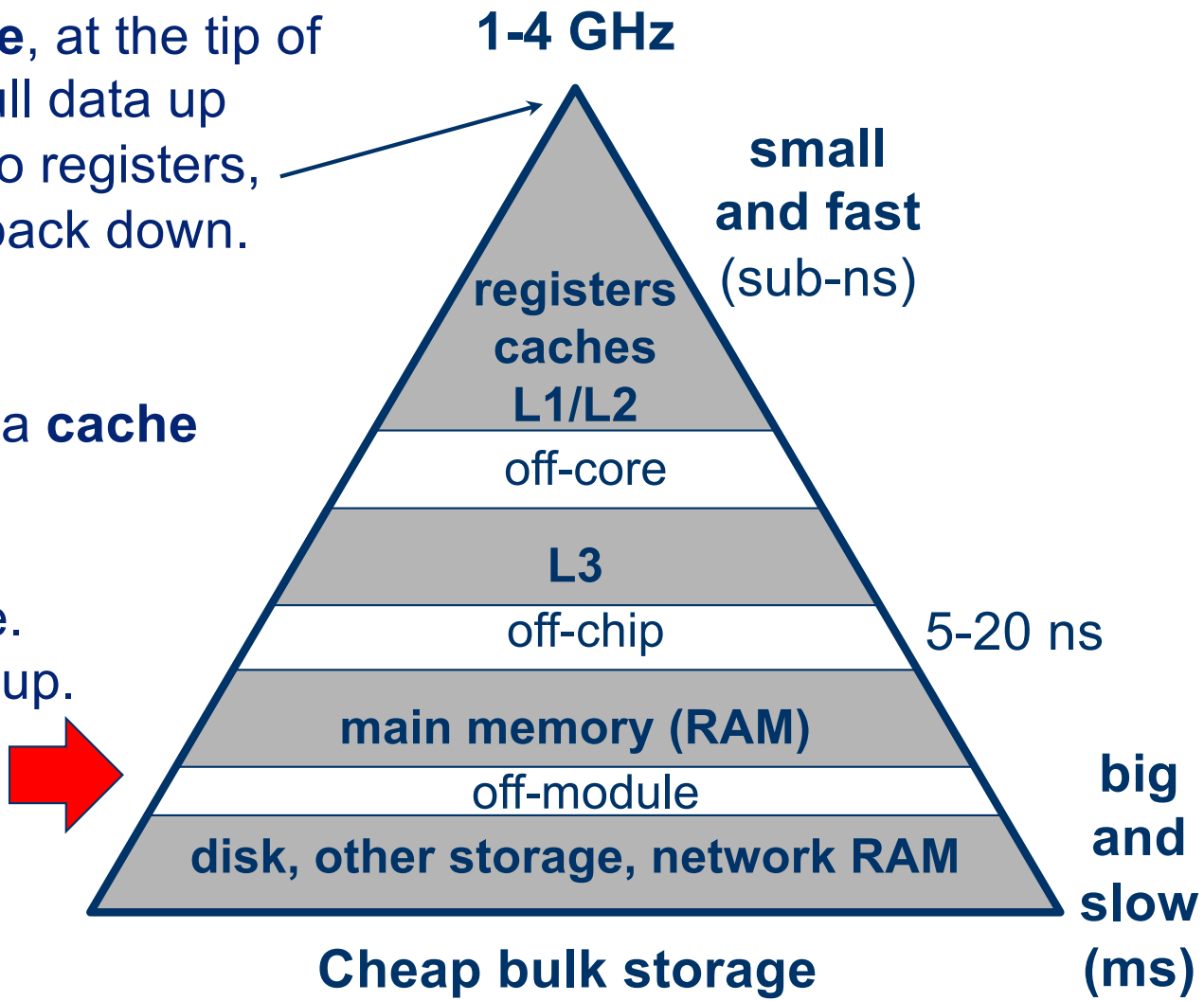
[from CS:APP3e F8:5-7 combined/edited]

Memory/storage hierarchy

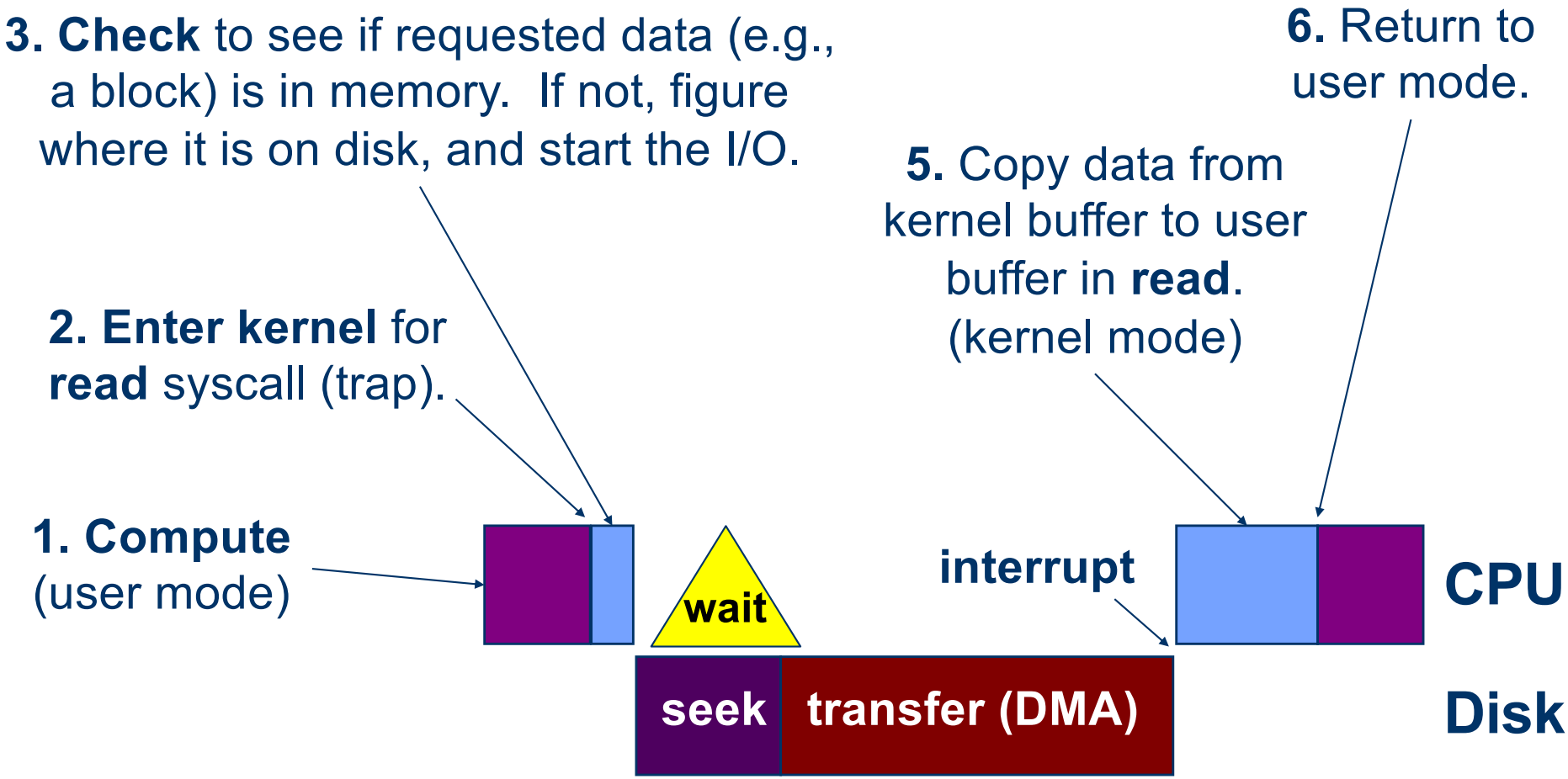
Computing happens **here**, at the tip of the spear. The cores pull data up through the hierarchy into registers, and then push updates back down.

In general, each layer is a **cache** over the layer below.

Input/Output (I/O) is here. Requires OS help to set up.



Anatomy of a disk read



Summary

- Software is code (**instructions**) and typed data.
- Hardware runs code on CPU and stores data in RAM.
- Code executes as a sequential **stream** with its **context** in registers, and in the memory it uses (e.g., its **stack**).
- CPU hardware can run one stream per register set (hardware thread). Suppose each CPU **core** has one.
- **Concurrency**: streams execute “at the same time”.
- **Protection**: each core executes in some privilege **mode** at any given time. OS runs in trusted **kernel** mode.
- CPU **exception** (trap, fault, interrupt) switches the core to kernel mode to run a **handler** in the OS.