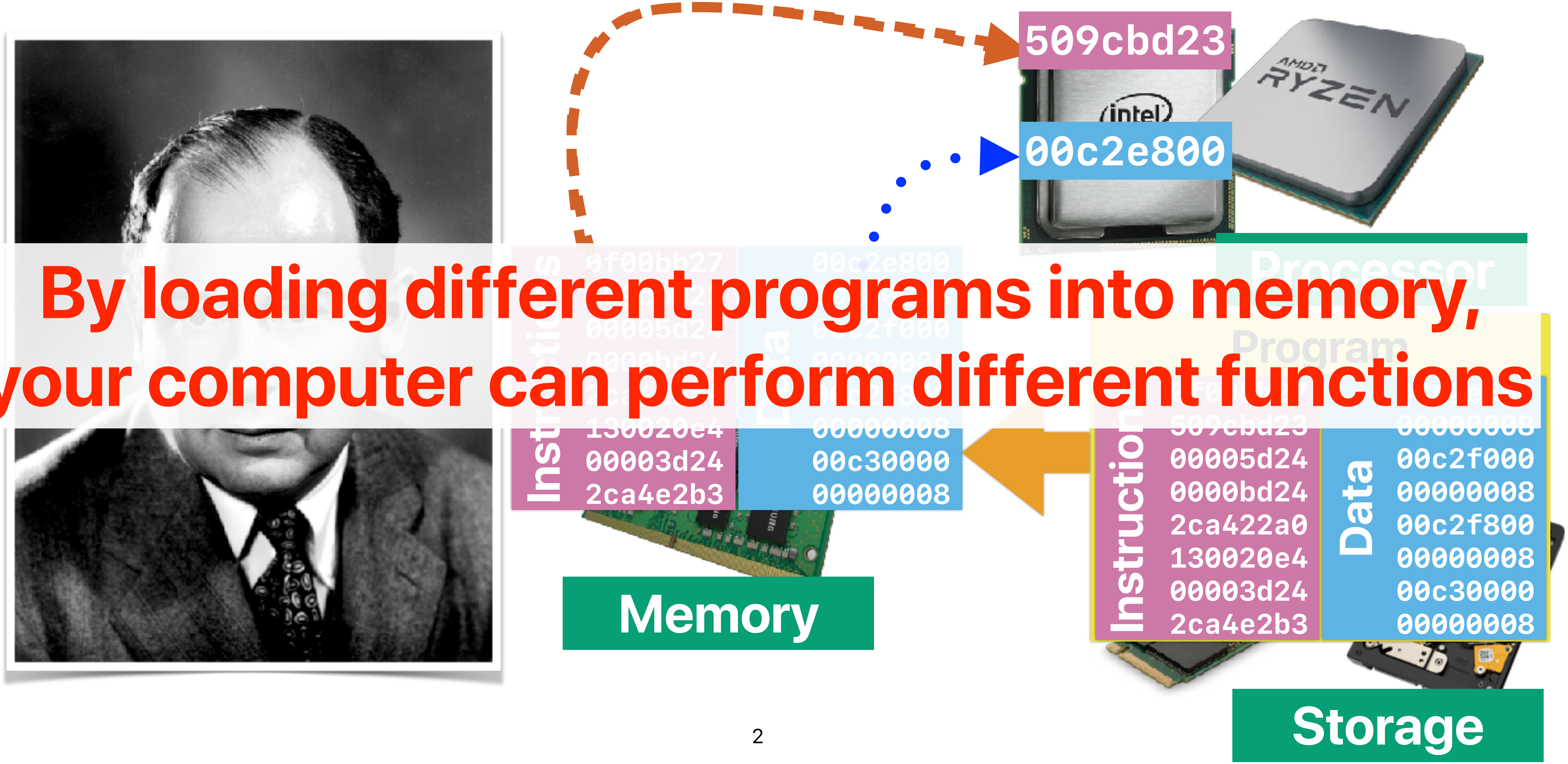
The Fundamentals of Operating Systems

Hung-Wei Tseng

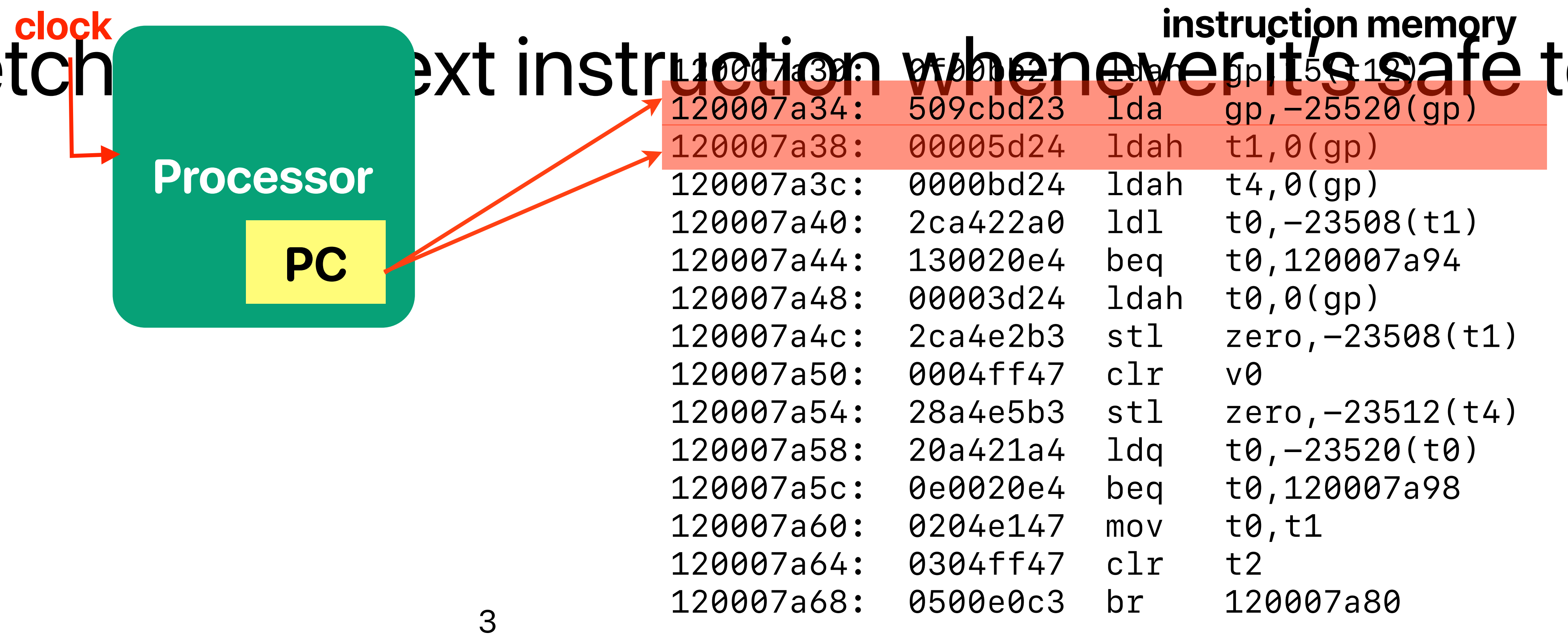
Recap: von Neumman Architecture



By loading different programs into memory, your computer can perform different functions

Recap: How processor executes a program

- The program counter (PC) tells where the upcoming instruction is in the memory
- Processor fetches the instruction, decode the instruction, execute the instruction, present the instruction results according to clock signals
- The processor fetch ext instruction whenever it's safe to do so

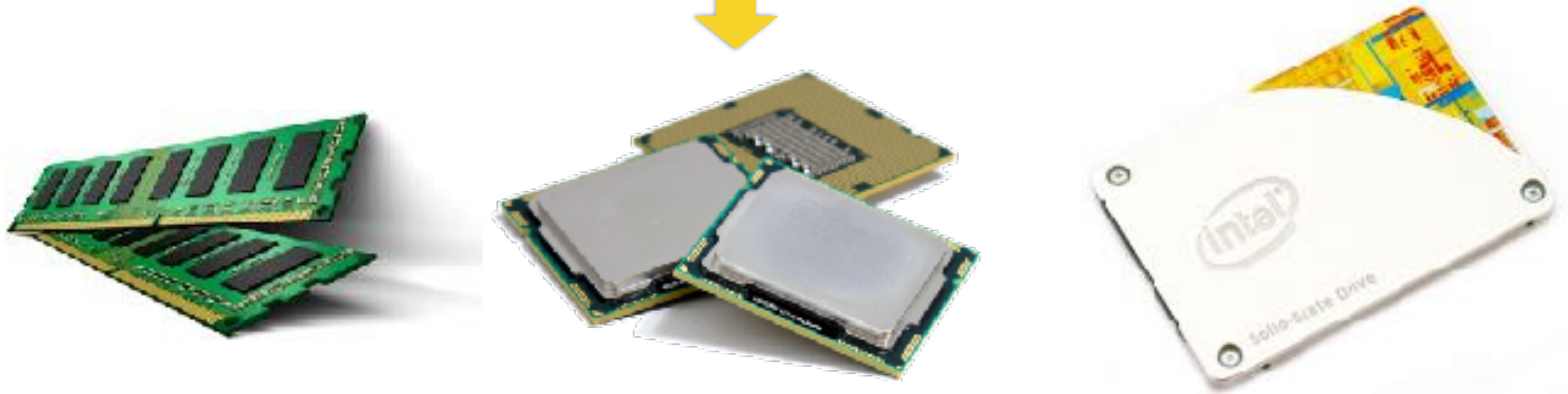


Without OSs...

https://az-eandt-live-legacy.azureedge.net/news/2013/apr/images/640_edsac-web.jpg

Without an OS: Direct Execution

Only one at a time, no way to interrupt a running task, and etc...



The goal of an OS

Recap: What modern operating systems support?

- **Virtualize** hardware/architectural resources
  - Easy for programs to interact with hardware resources
  - Share hardware resource among programs
  - Protect programs from each other (security)
- Execute multithreaded programs **concurrently**
  - Support multithreaded programming model
  - Execute multithreaded programs efficiently
- Store data **persistently**
  - Store data safely
  - Secure

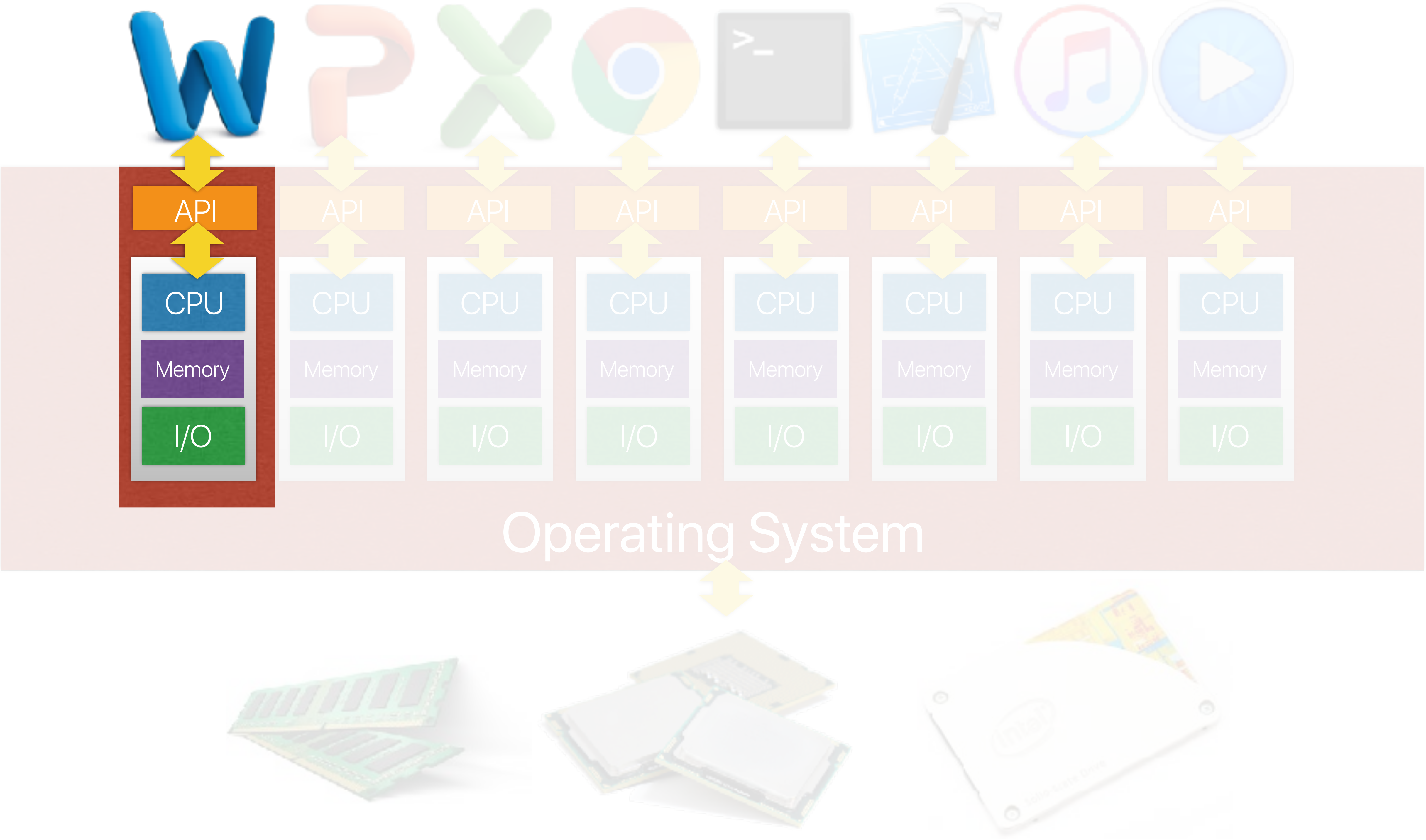
Outline

- Operating systems: virtualizing computers
- Process: the most important abstraction in modern OSs
- Restricted operations: kernel and user modes

Operating systems: virtualizing computers

The goal of an OS

The idea of an OS: virtualization



The idea: virtualization

- The operating system presents an illusion of a virtual machine to each running program and maintains architectural states of a von Neumann machine
  - Processor
  - Memory
  - I/O
- Each virtualized environment accesses architectural facilities through some sort of application programming interface (API)
- Dynamically map those virtualized resources into physical resources

Demo: Virtualization

```
double a;

int main(int argc, char *argv[])
{
    int cpu, status, i;
    int *address_from_malloc;
    cpu_set_t my_set;        // Define your cpu_set bit mask.
    CPU_ZERO(&my_set);       // Initialize it all to 0, i.e. no CPUs selected.
    CPU_SET(4, &my_set);     // set the bit that represents core 7.
    sched_setaffinity(0, sizeof(cpu_set_t), &my_set); // Set affinity of this process to the defined mask, i.e. only 7.
    status = syscall(SYS_getcpu, &cpu, NULL, NULL);
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s process_nickname\n",argv[0]);
        exit(1);
    }

    srand((int)time(NULL)+(int)getpid());
    a = rand();

    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(1);

    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(3);

    return 0;
}
```

getcpu system call to retrieve the executing CPU ID

create a random number

print the value of a and address of a

print the value of a and address of a again after sleep

```
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
```

The same processor!

Different values

Different values are preserved

The same memory address!

Demo: Virtualization

- Some processes may use the same processor
- Each process has the same address for variable a, but different values.
- You may see the content of a compiled program using objdump

Why virtualization

- How many of the following statement is true about why operating systems virtualize running programs?
  ① Virtualization can help improve the utilization and the throughput of the underlying hardware
  ② Virtualization may allow the system to execute more programs than the number of physical processors installed in the machine
  Make programs less machine-dependent
  ③ Virtualization may allow a running program or running programs to use more than install physical memory
  ④ Virtualization can improve the latency of executing each program
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Latency v.s. Throughput

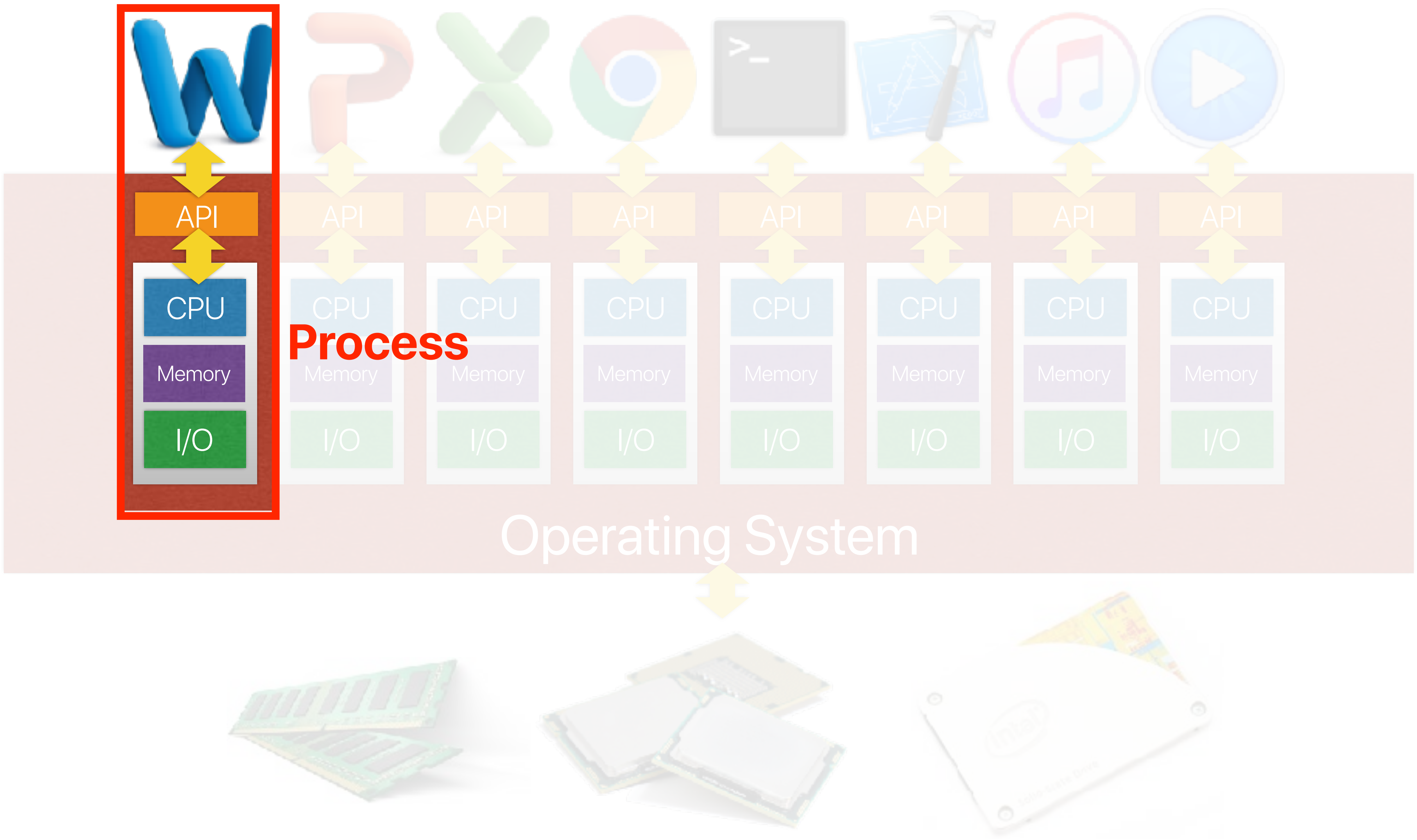
- A 4K movie clip using H.265 coding takes **70GB** in storage
- If you want to transfer a total of 2 Peta-Byte video clips (roughly 29959 movies) from UCSD
  - 100 miles from UCR
  - Assume that you have a **100Gbps** ethernet
    - Throughput: 100 Gbits per second
    - 2 Peta-byte (16 Peta-bits) over 167772 seconds = 1.94 Days
    - Latency: first 70GB (first movie) in 6 seconds

Or ...

	Toyota Prius 100 miles from UCSD 75 MPH on highway! 50 MPG Max load: 374 kg = 2,770 hard drives (2TB per drive) = 5.6 PB	**10Gb Ethernet** 
Throughput/ bandwidth	**450GB/sec**	100 Gb/s or 12.5GB/sec
latency	3.5 hours	2 Peta-byte over 167772 seconds = 1.94 Days
response time	You see **nothing** in the first 3.5 hours	You can start watching the first movie as soon as you get a frame!

Process: the most important abstraction in modern operating systems

The idea of an OS: virtualization



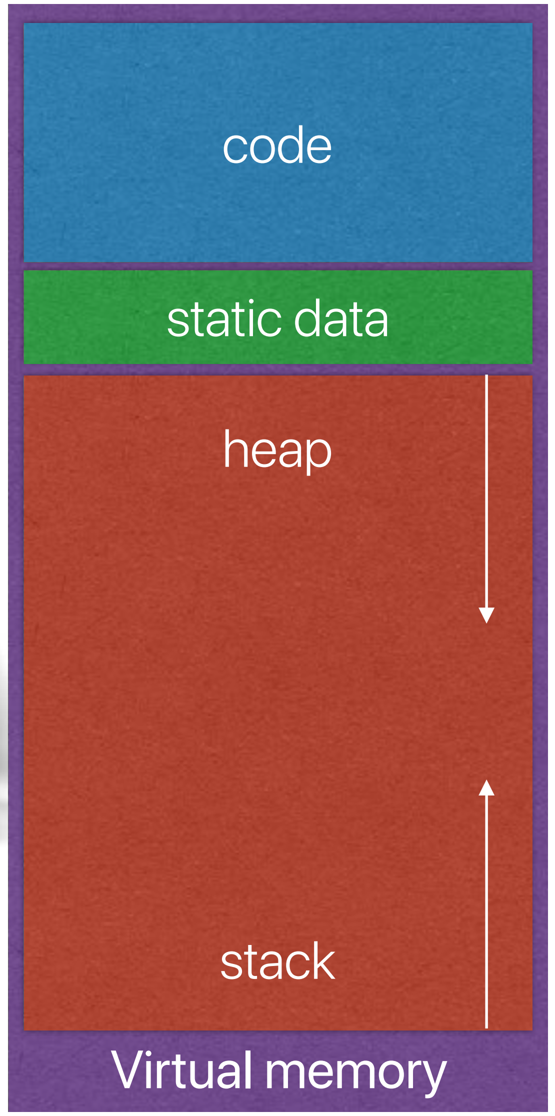
Processes

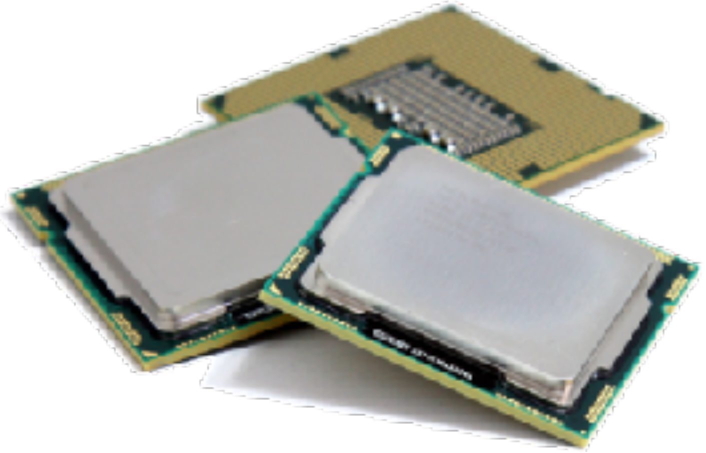
- The **most important abstraction** in modern operating systems.
- A process abstracts the underlying computer.
- A process is a **running program —** a dynamic entity of a program.
  - Program is a static file/combination of instructions
  - Process = program + states
  - The states evolves over time
- A process may be dynamically switched out/back during the execution

Virtualization

- The operating system presents an **illusion** of a **virtual machine** to each running program — **process**
  - Each virtual machine contains architectural states of a von Neumann machine
    - Processor
    - Memory
    - I/O
- Each virtualized environment accesses architectural facilities through some sort of application programming interface (API) — **system calls**
- Dynamically map those virtualized resources into physical resources — **policies, mechanisms**

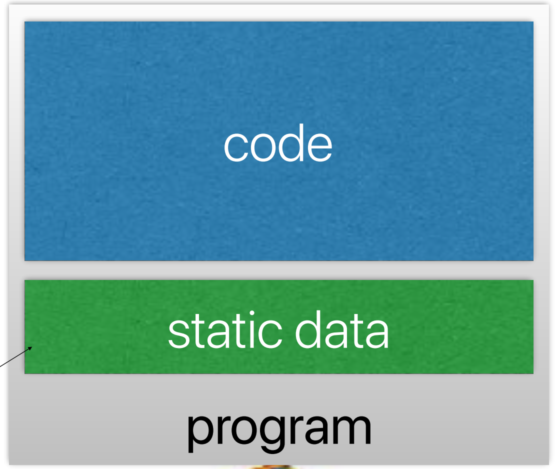
What happens when creating a process





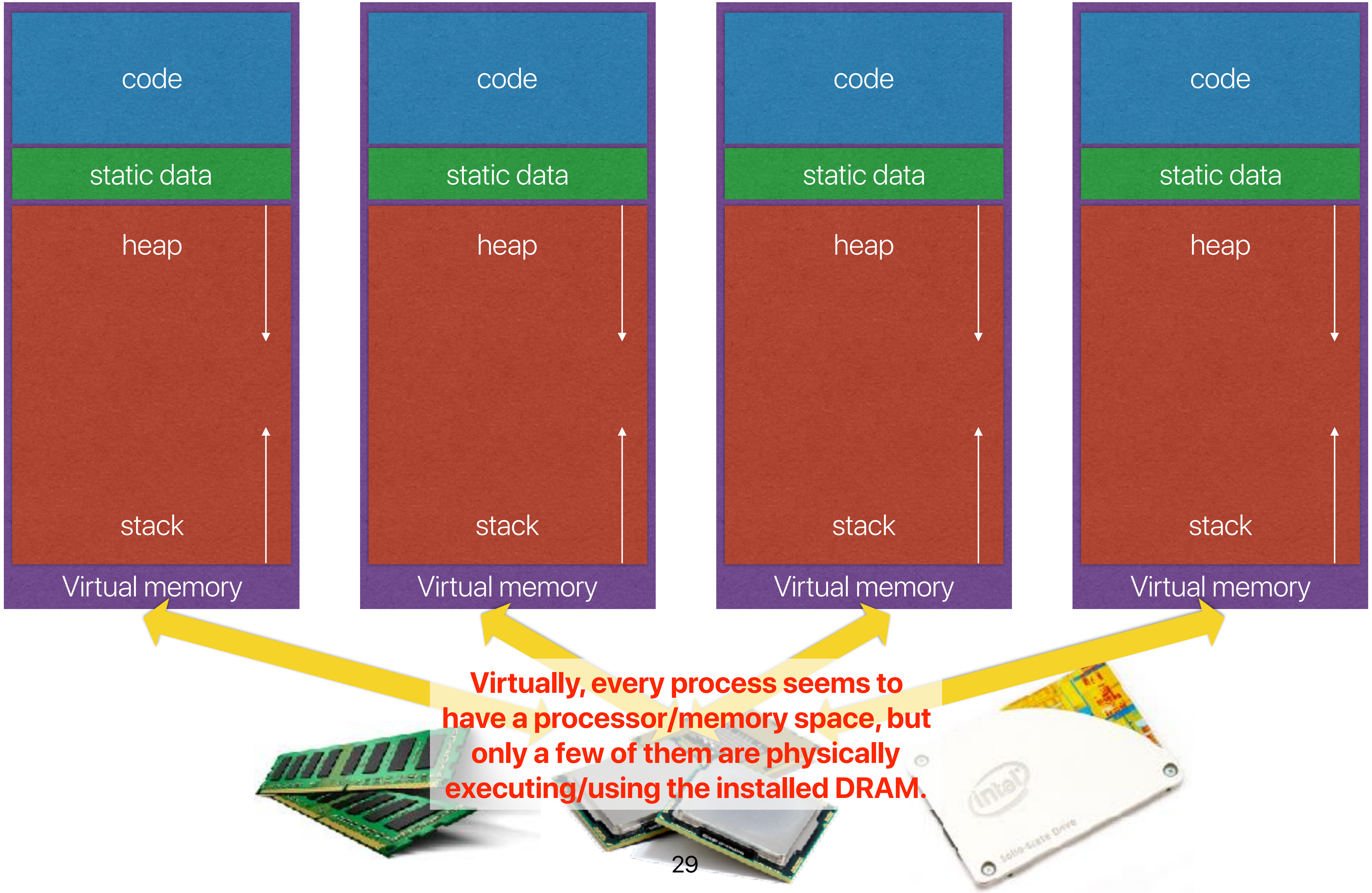
Dynamic allocated data: `malloc()`

Local variables,
arguments



Linux contains a .bss section
for uninitialized global variables

The illusion provided by processes



Virtually, every process seems to have a processor/memory space, but only a few of them are physically executing/using the installed DRAM.

What the OS must track for a process?

- Which of the following information does the OS need to track for each process?
  A. Stack pointer
  B. Program counter
  C. Process state
  D. Registers
  E. All of the above

- You also need to keep other process information like an unique process id, process states, I/O status, and etc…

Process control block

- OS has a PCB for each process
- Sometimes called Task Controlling Block, Task Struct, or Switchframe
- The data structure in the operating system kernel containing the information needed to manage a particular process.
- The PCB is the manifestation of a process in an operating system

Example: `struct task_struct` in Linux

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;
        int on_rq;
        int prio, static_prio, normal_prio;
        const struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
        unsigned int policy;
        int nr_cpus_allowed;
        cpumask_t cpus_allowed;
        pid_t pid;
        struct task_struct __rcu *real_parent;
        struct task_struct __rcu *parent;
        struct list_head children;
        struct list_head sibling;
        ……
        struct list_head tasks;
        ……
        struct mm_struct *mm, *active_mm;
        ……
/* CPU-specific state of this task */
        struct thread_struct thread;
}
```

Process state

Process ID

Virtual memory pointers

Low-level architectural states

- You may find this struct in /usr/src/linux-headers-x.x.x-xx/include/linux/sched.h

Memory pointers in `struct mm_struct`

```
struct mm_struct {
        struct vm_area_struct * mmap;           /* list of VMAs */
        ...
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        ...
};
```

start of heap

end of heap

current stack pointer

Processor states in `struct thread_struct`

```
struct thread_struct {
        struct desc_struct tls_array[GDT_ENTRY_TLS_ENTRIES];
        unsigned long           sp0;
        unsigned long           sp;
#ifdef CONFIG_X86_32
        unsigned long           sysenter_cs;
#else
        unsigned short          es;
        unsigned short          ds;
        unsigned short          fsindex;
        unsigned short          gsindex;
#endif
#ifdef CONFIG_X86_32
        unsigned long           ip;
#endif
#ifdef CONFIG_X86_64
        unsigned long           fs;
#endif
        unsigned long           gs;
        struct perf_event       *ptrace_bps[HBP_NUM];
        unsigned long           debugreg6;
        unsigned long           ptrace_dr7;
        unsigned long           cr2;
        unsigned long           trap_nr;
        unsigned long           error_code;
#ifdef CONFIG_VM86
        struct vm86             *vm86;
#endif
        unsigned long           *io_bitmap_ptr;
        unsigned long           iopl;
        unsigned                io_bitmap_max;
        struct fpu              fpu;
    };
```

Some x86 Register values

Program counter

Virtualization

However, we don’t want everything to pass through this API!

Too slow!!!

Do you really need to track all intermediate states?

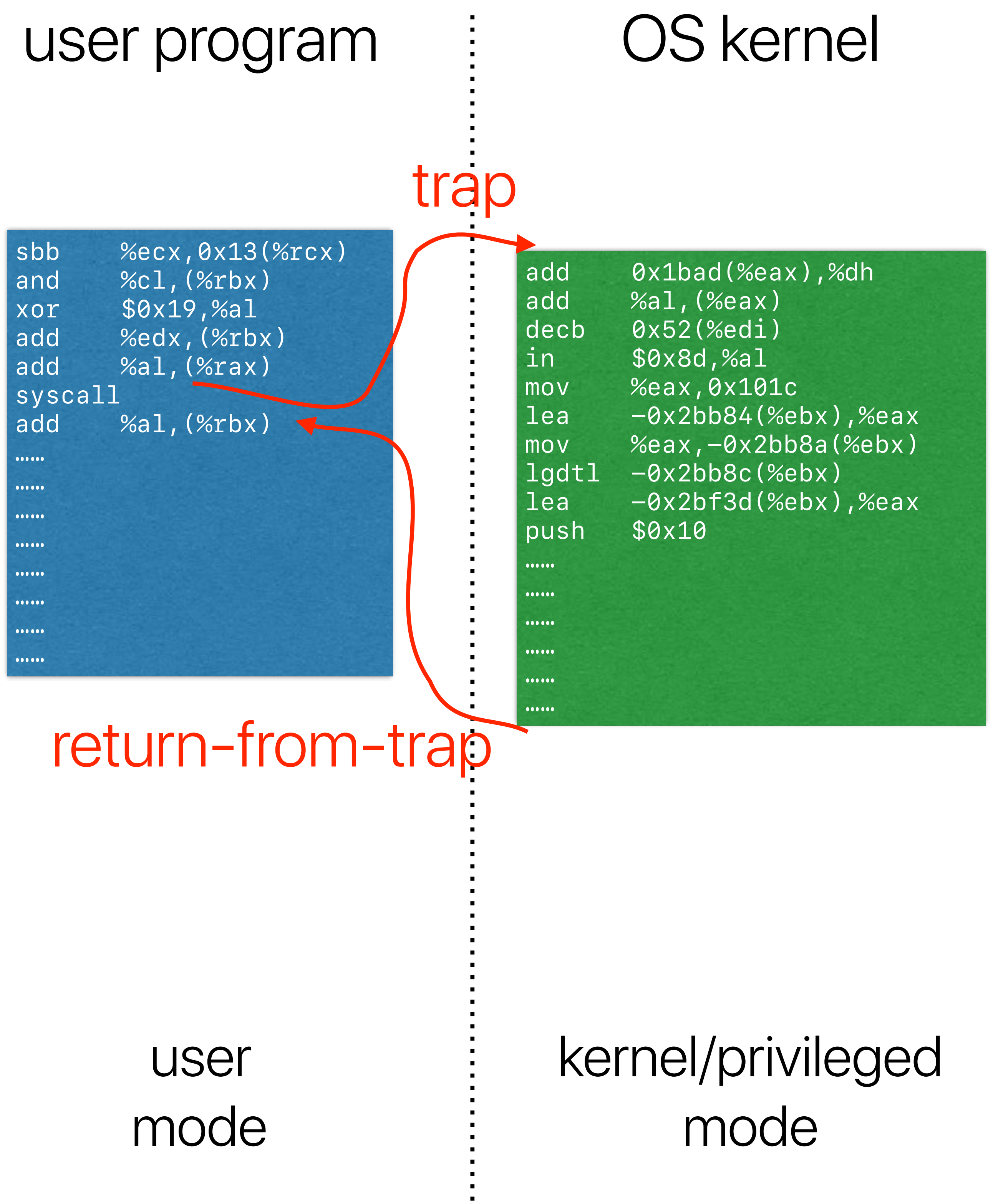
Restricted operations: kernel and user modes

Restricted operations

- Most operations can directly execute on the processor without OS's intervention
- The OS only takes care of protected resources, change running processes or anything that the user program cannot handle properly
- Divide operations into two modes
  - User mode
    - Restricted operations
    - User processes
  - Kernel mode
    - Can perform privileged operations
    - The operating system kernel
- Requires architectural/hardware supports

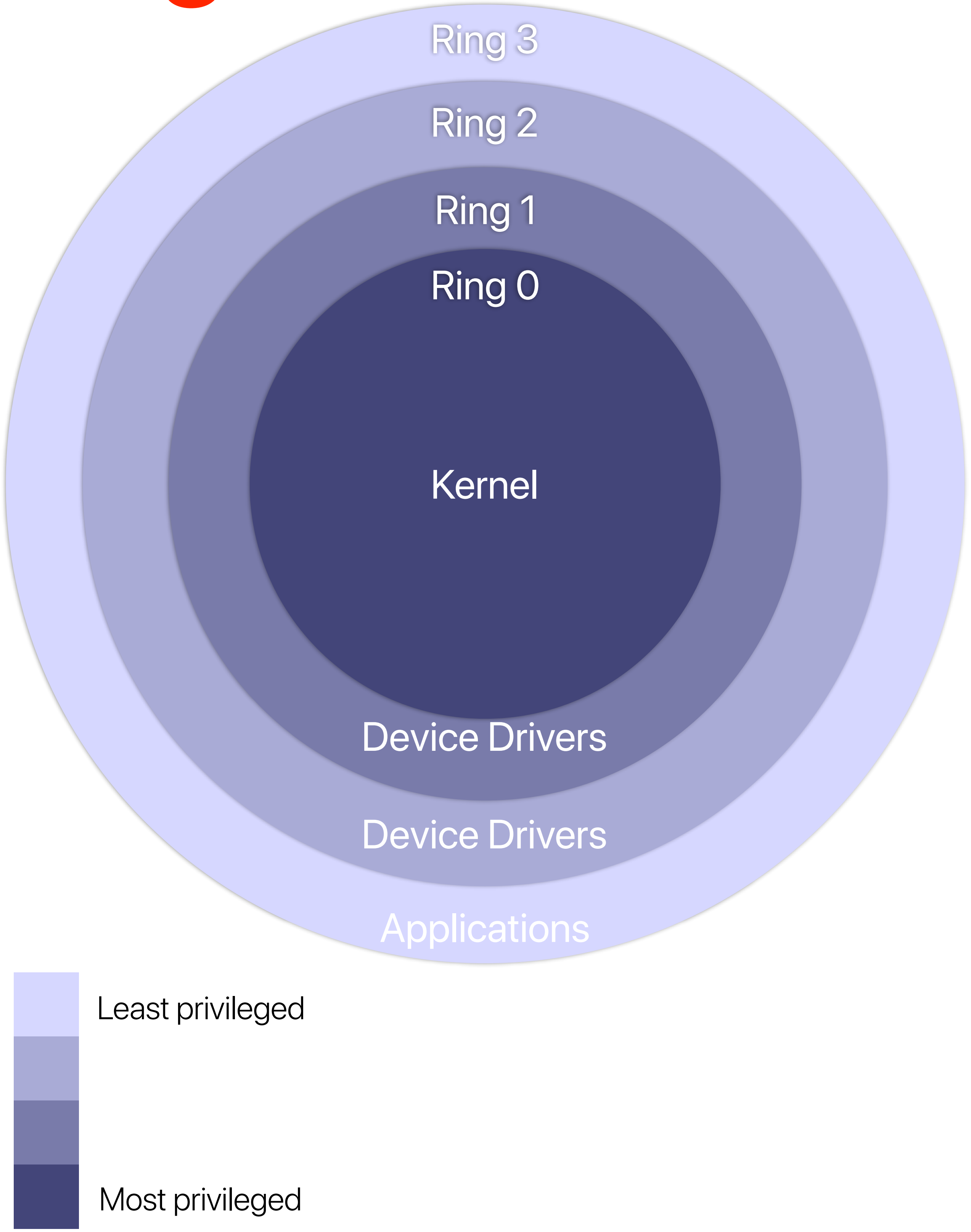
How applications can use privileged operations?

- Through the API: **System calls**
- Implemented in “trap” instructions
  - Raise an exception in the processor
  - The processor saves the exception PC and jumps to the corresponding exception handler in the OS kernel



Architectural support: privileged instructions

- The processor provides **normal** instructions and **privileged** instructions
  - Normal instructions: ADD, SUB, MUL, and etc …
  - Privileged instructions: HLT, CLTS, LIDT, LMSW, SIDT, ARPL, and etc…
- The processor provides different modes
  - User processes can use normal instructions
  - Privileged instruction can only be used if the processor is in proper mode



What is "kernel"

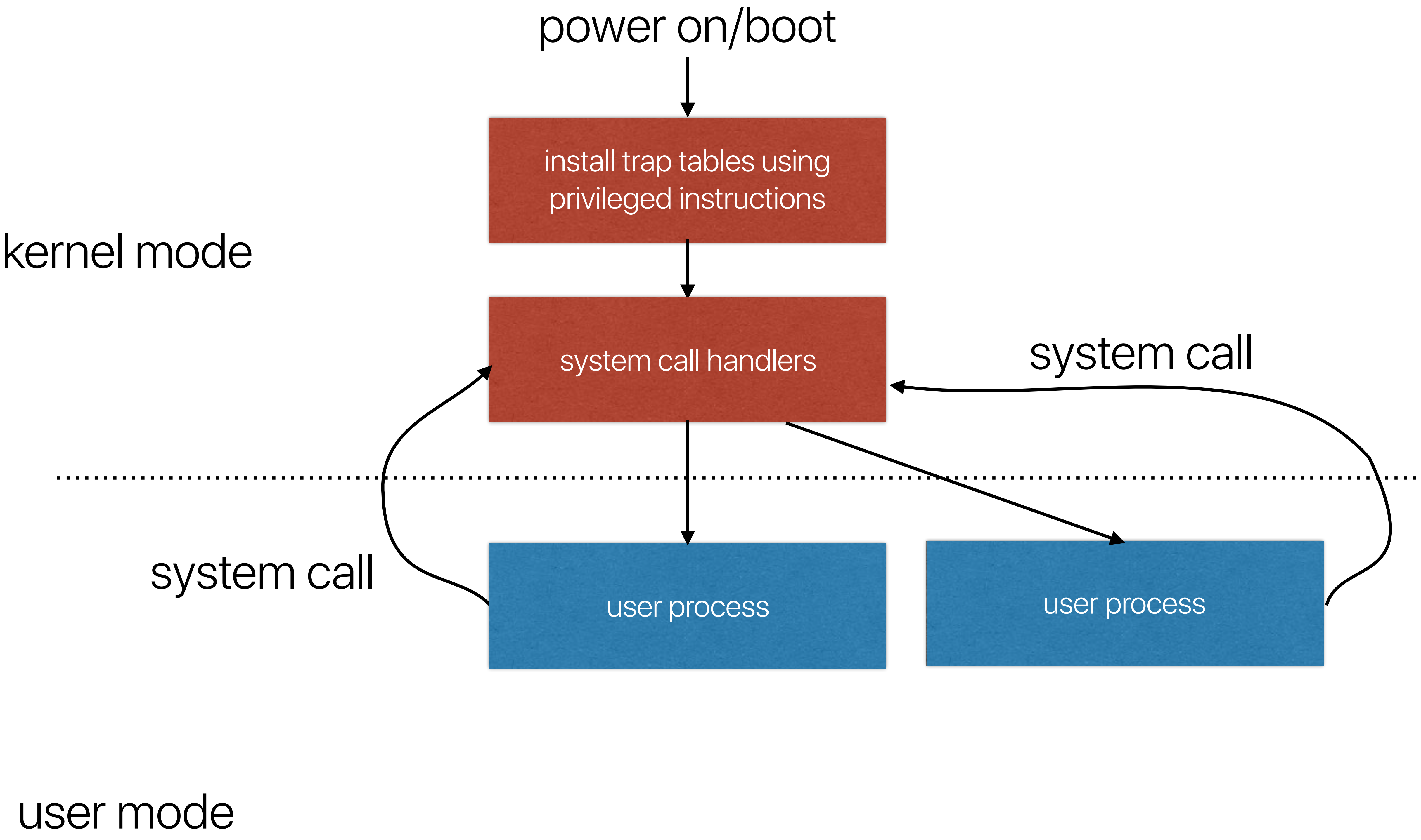
- Which of the following is true about kernel?
  A. It executes as a process — executing kernel function will then require context switch, but context switch also needs to access kernel….
  B. It is always executing, in support of other processes — what if we only have one processor core? You cannot execute any other program…
  C. It should execute as little as possible.
  D. A & B
  E. B & C

- The OS kernel only get involved when necessary
  - System calls
  - Hardware interrupts
  - Exceptions
- The OS kernel works **on behave of** the requesting process — not a process
  - Somehow like a **function call** to a dynamic linking library
  - Preserve the current architectural states and update the PCB
  - As a result — overhead of copying registers, allocating local variables for kernel code and etc…

"A lie doesn't become truth, wrong doesn't become right and evil doesn't become good, just because it is accepted by a majority."

–RICK WARREN

How does the processor knows where to jump to?



Latency Numbers Every Programmer Should Know

Operations	Latency (ns)	Latency (us)	Latency (ms)	
L1 cache reference	0.5 ns			~ 1 CPU cycle
Branch mispredict	5 ns			
L2 cache reference	7 ns			14x L1 cache
Mutex lock/unlock	25 ns			
Main memory reference	100 ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns	3 us		
Send 1K bytes over 1 Gbps network	10,000 ns	10 us		
Read 4K randomly from SSD*	150,000 ns	150 us		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 us		
Round trip within same datacenter	500,000 ns	500 us		
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 us	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000 ns	10,000 us	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20,000 us	20 ms	80x memory, 20X SSD
Send packet CA-Netherlands-CA	150,000,000 ns	150,000 us	150 ms	

Demo: Kernel Switch Overhead

- Measure kernel switch overhead using lmbench http://www.bitmover.com/lmbench/

The overhead of kernel switches/system calls

- On a 3.7GHz intel Core i5-9600K Processor, please make a guess of the overhead of switching from user-mode to kernel mode.
    A. a single digit of nanoseconds
    B. tens of nanoseconds
    C. hundreds of nanoseconds
    D. a single digit of microseconds
    E. tens of microseconds

Operations	Latency (ns)
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 1K bytes over 1 Gbps network	10,000 ns
Read 4K randomly from SSD*	150,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Read 1 MB sequentially from SSD*	1,000,000 ns
Disk seek	10,000,000 ns
datacenter roundtrip	
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA-Netherlands-CA	150,000,000 ns

Announcement

- Two reading quizzes next week
  - We will discuss **4 papers** next week
  - We split them into two since that's probably the first you read papers
- Check your clicker grades in iLearn around next Monday