

W CSE 451: Operating Systems, 26au

L2: I/O, Processes

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head)

Mahir Emran

Ted Guan

Jai Rungrachakanont

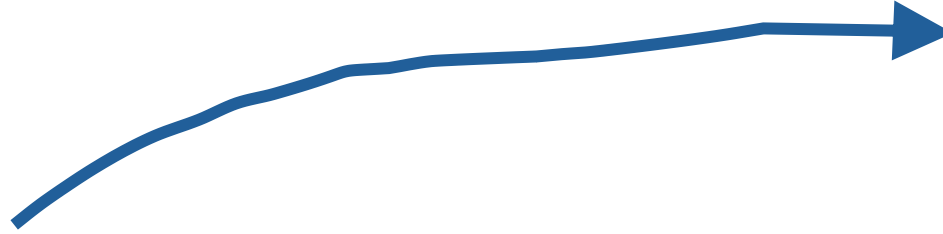
Soham Raut

Sagar Saby

Andrew Wood

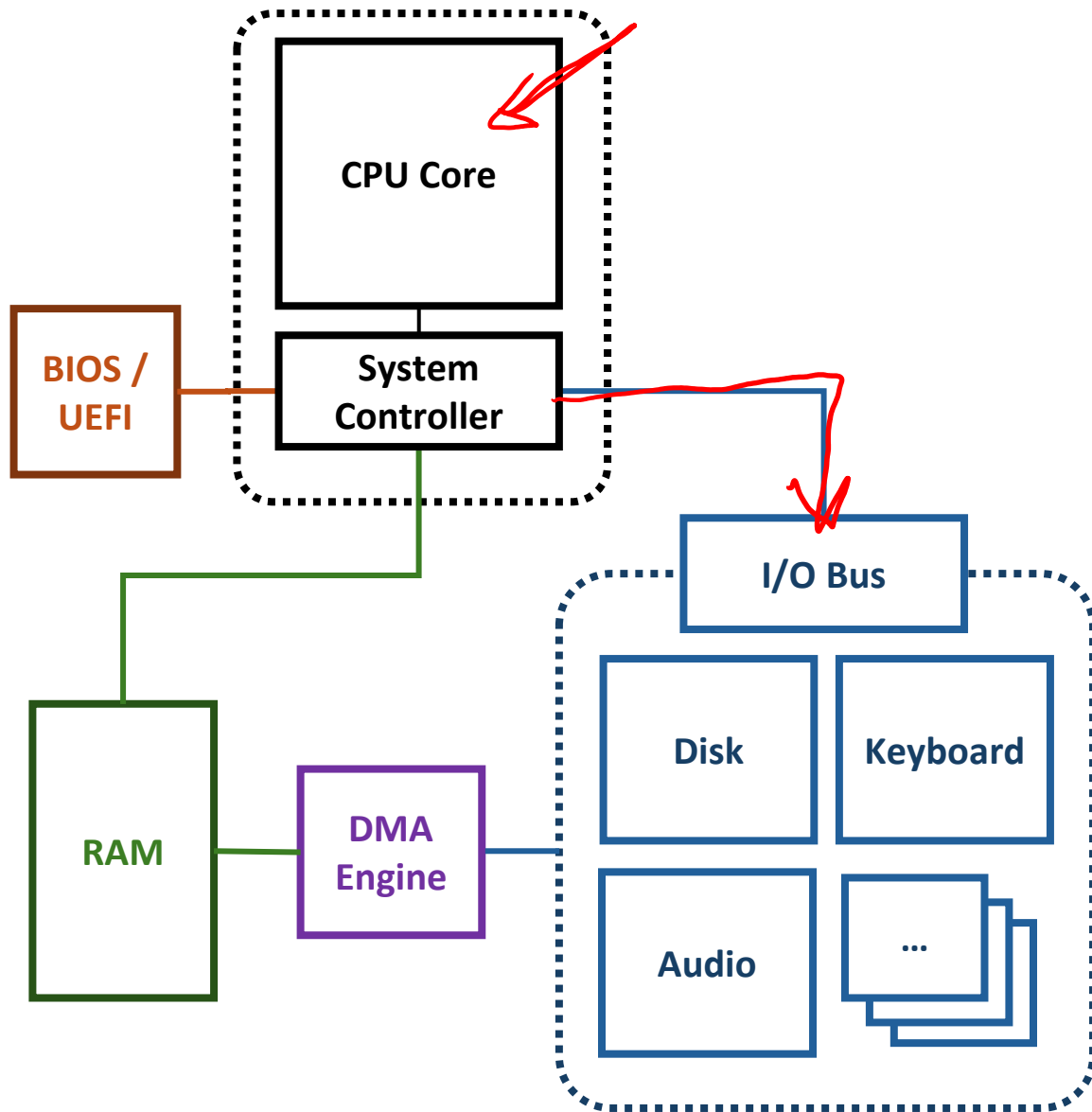
Adapted from slides by Tom Anderson

Administrivia

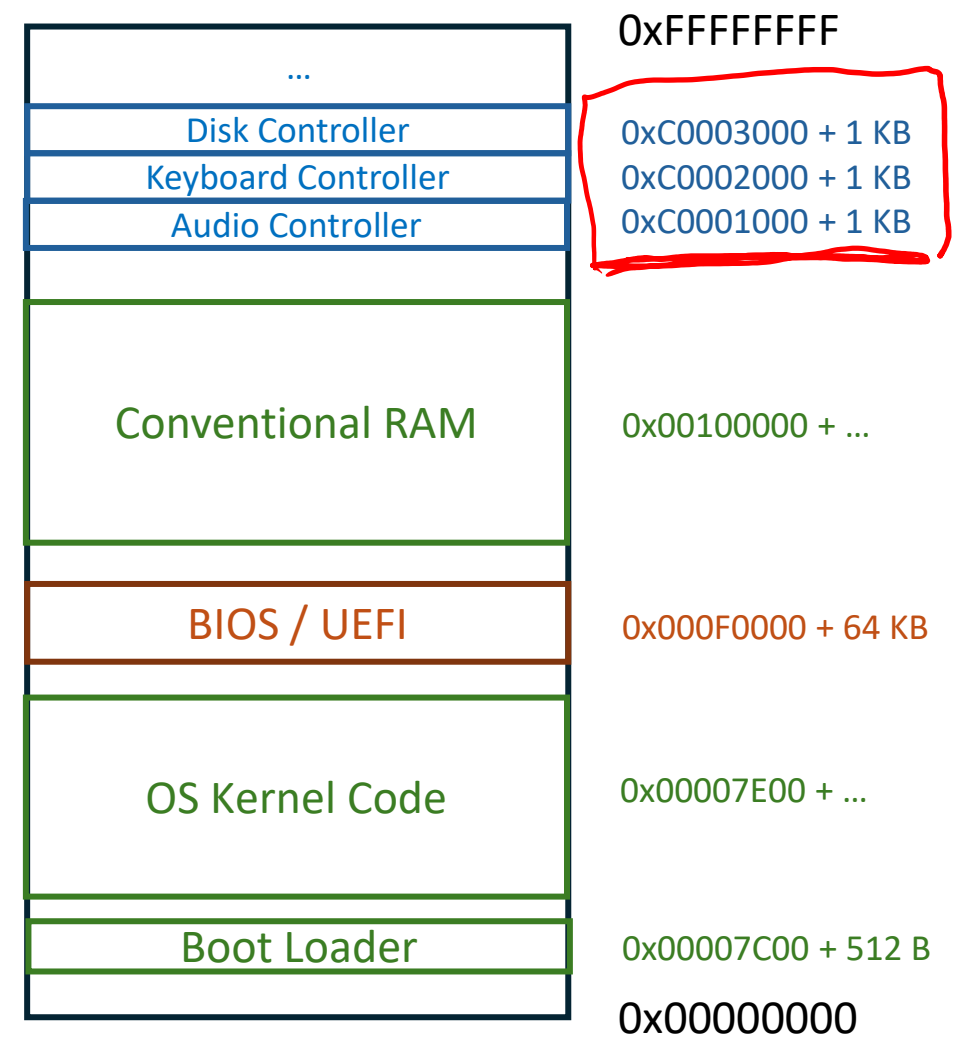


- Fill out yer' partner forms by the end of today.
(one per group)
 - See Ed superthread for finding a partner (thank you Mahir!!)
- ~~A~~ You'll be randomly assigned someone if you don't fill the form out 🐱
- On Monday we'll release Lab 1 and PSet 1

Device I/O



Physical Address Space



Device I/O

- OS kernel needs to communicate with physical devices
 - Network, disk, video, USB, keyboard, mouse, ...
 - Devices operate asynchronously from the CPU
 - Have their own lil processors (“**microcontrollers**”) running their own lil code (“**firmware**”)
- How does the OS communicate with this firmware?
 - Some addresses in the CPU’s physical address space are effectively “wormholed” to these external chips
 - Writing bytes -> sending commands to the firmware
 - Reading bytes -> getting results back from those commands
 - ✱ In C, this looks like **setting a pointer** to a SUPER SPECIFIC but ARBITRARY LOOKING value, and then **dereferencing it**

Waiting for I/O to finish

- Synchronous I/O: **Polling**

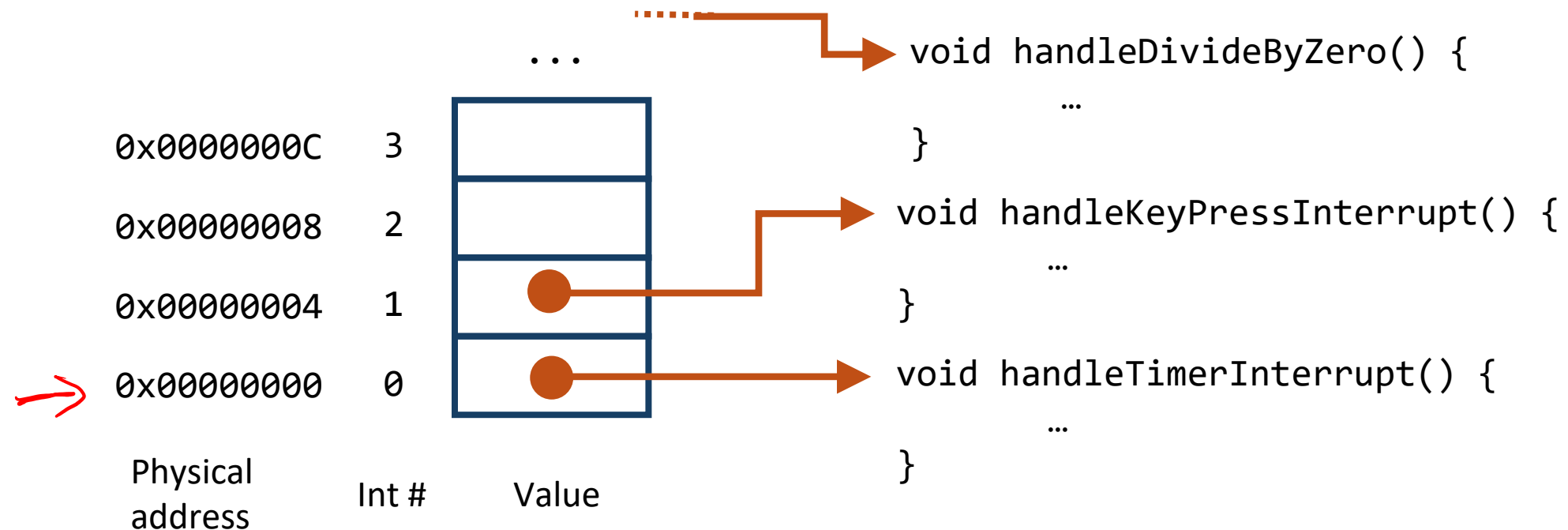
- OS pokes I/O register
- OS sits in a while() loop reading a “status” I/O register (“**polls** it”).
- Loop exits when the “status” register magically becomes **DEVICE_DONE** instead of **DEVICE_WORKING**.
- OS copies relevant data from the rest of the I/O registers into memory

- Asynchronous I/O: **Interrupts**

- OS pokes I/O register
- OS goes on and does something entirely different
- When the device is done, it smacks the CPU (“**triggers an interrupt**”)
- The interrupt hardware forcibly moves the CPU’s instruction pointer to a callback function (the “**interrupt handler**”) and pushes a new stack frame for it.
- When the handler is done, OS carefully returns to the stack frame that got interrupted

Interrupt Nitty-Gritty

- **Interrupt Vector:** an array of function pointers in RAM
 - Each pointer is to a different interrupt handler (eg, callback function)
 - Purpose of each callback hard-coded by hardware



Interrupt Nitty-Nitty-Gritty-Gritty

- See:
 - `xk/kern/vectors.s`
 - `xk/kern/trapasm.s`
 - `xk/kern/trap.c:trap`
 - `xk/inc/trap.h`



Interrupt Masking

- Interrupt handler runs with interrupts off
 - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
 - Eg., when determining the next process/thread to run
 - On x86:
 - CLI: disable interrupts
 - STI: enable interrupts
 - Only applies to the current CPU (on a multicore)
- We'll need this to implement context switching

Oh by the way



- Which of these details are specific to x86, and which are universal?
- Which of these details are there because they allow **hardware simplicity**?

High-bandwidth I/O

“Doing I/O in bulk”

- Direct memory access (**DMA**) engines
 - Dedicated circuits that sit outside of the CPU and copy bytes from external devices into RAM
 - CPU communicates with DMA engines through regular I/O registers
 - Glorified hardware memcpy()

- A simple toy DMA engine:

```
extern volatile uint32_t **ram_ptr;  
extern volatile uint32_t *disk_sector;  
extern volatile uint32_t *n_bytes;  
extern volatile uint32_t *direction; // DMA_READ or DMA_WRITE
```

memory
address
↓

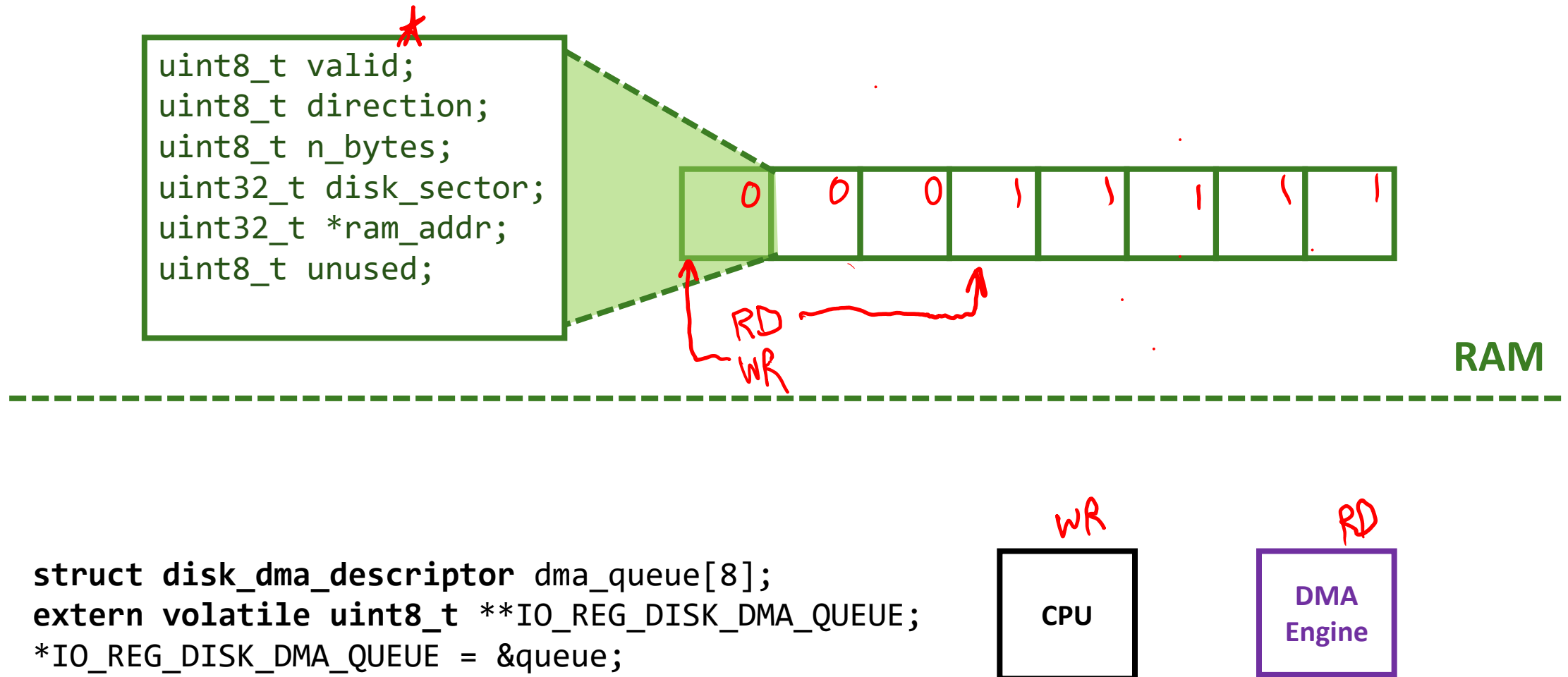
I/O regs
↓

device/HW specific
address
←

Realistic DMA

- Store a queue of DMA transfers for the engine to work on
 - Each entry in the queue represents is a structure called a **buffer descriptor**
 - Descriptor specifies things like RAM address, number of bytes, directionality (where applicable), etc
 - CPU pushes descriptors, DMA engine pops them
- Queue implementation: “**ring buffer**”
 - Fixed-sized array of buffer descriptor structs stored in RAM
 - CPU keeps a “write pointer” to the head of the queue. Advances the pointer on push.
 - DMA engine keeps a “read pointer” to the back of the queue. Advances the pointer on pop.
 - Pointers wrap to beginning of array when they reach the end.
 - Position of ring buffer itself communicated using an I/O register
- CPU interrupt only if buffer empties/fills

Buffer Descriptors Illustrated



Ring Buffer failure modes



https://www.youtube.com/watch?v=Jofn_9rPbzo

Class concluded here on 10/2
and will resume on 10/5

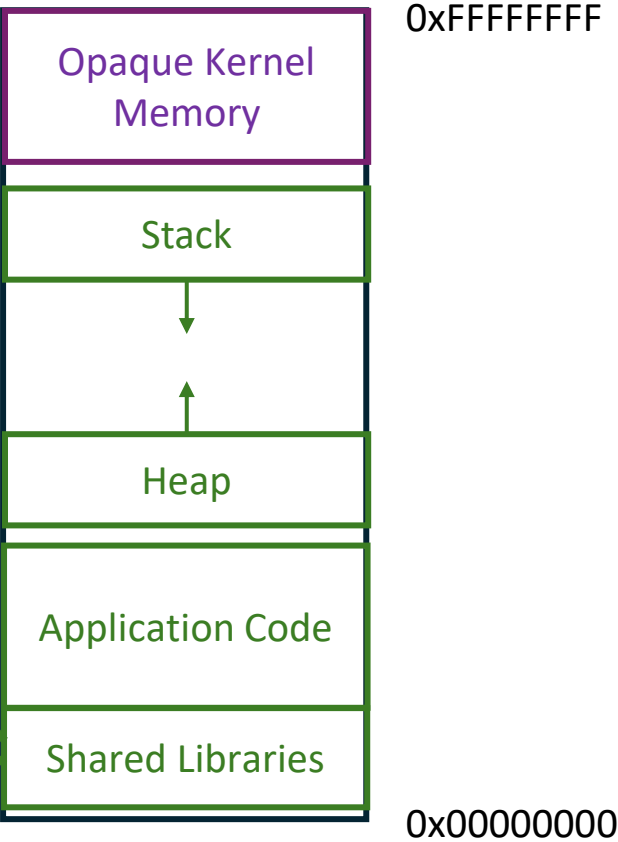
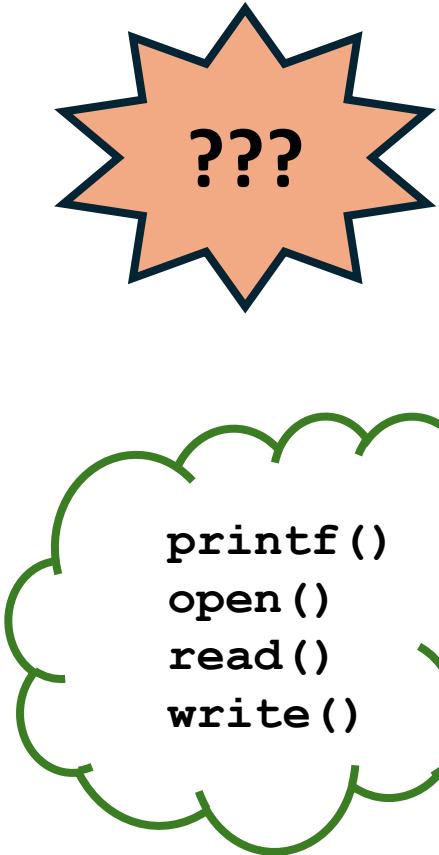
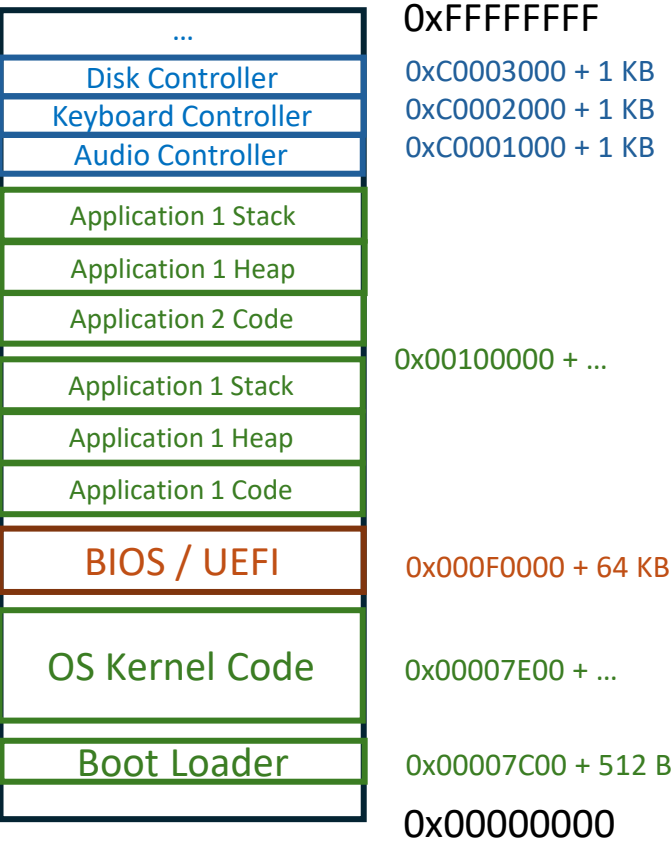


The Kernel Abstraction

Protecting the computer from itself

Physical Address Space

The Illusion of a Computer



Design goal: protection

- Code tends to be **buggy**
- Code can be **malicious**
- What are some things we want to **keep user code from doing?**

Process Abstraction

- Process: an *instance* of a program, with limited rights
 - Cannot modify or read operating system kernel code or data
 - Cannot modify or read I/O controller memory
 - Cannot modify its own rights without asking the kernel
 - Can be “kicked off” of the CPU against its will
 - Etc.
- Process contains one or more threads and an address space
 - Thread: a sequence of instructions within a process
 - Potentially many threads per process
 - Each thread gets a stack. All threads share the heap.
 - Address space: set of rights of a process
 - Memory regions that the process can access
 - Other permissions (e.g., which system calls it can make, what files it can access)

The process abstraction

 Referee

 Illusionist

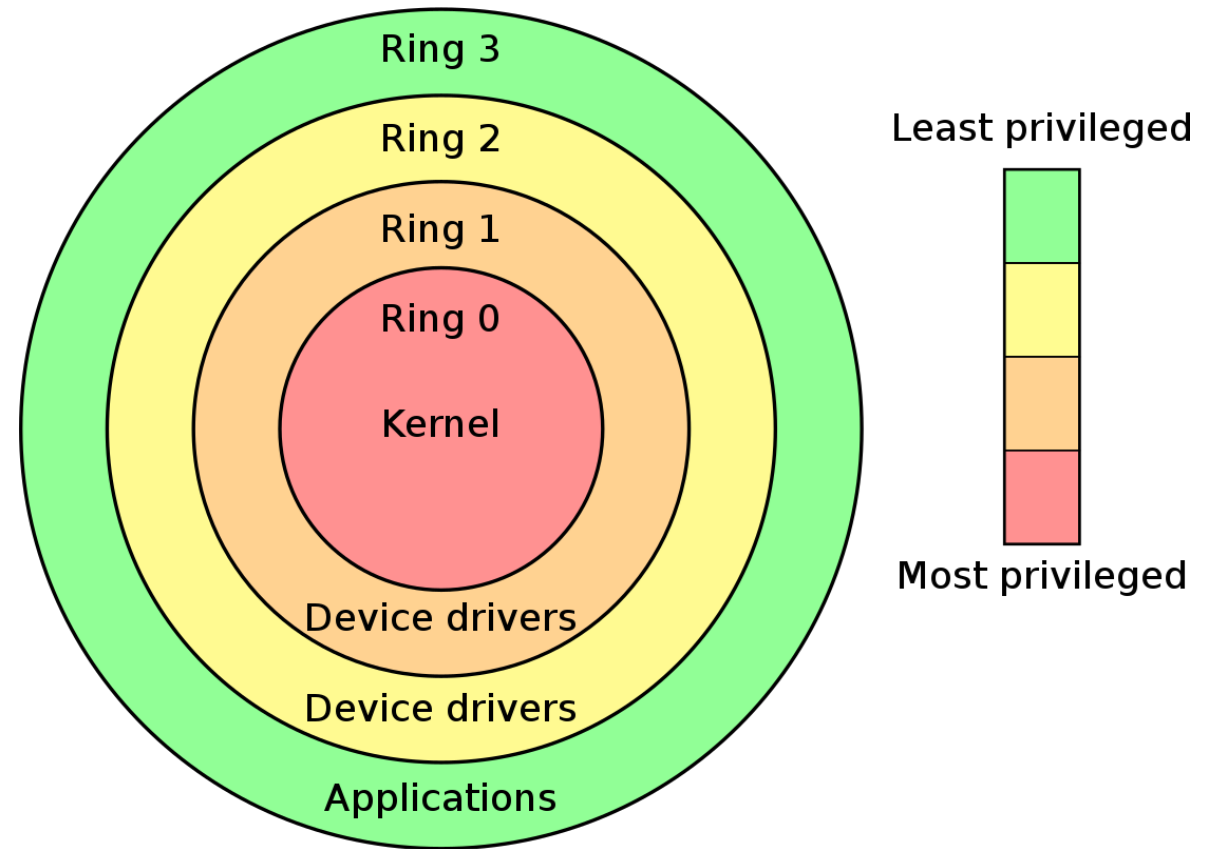
 Glue

CPU “Dual-mode Operation”

- Kernel mode
 - Privilege to do absolutely whatever
- User mode
 - Only privileged to execute a subset of instructions
 - Only privileged to access a subset of physical memory
 - Violating these privileges causes the processor to immediately triggers an interrupt
- On the x86, mode stored in the CPU's EFLAGS register

Privilege levels in x86

- x86 privileges are actually configurable in up eight levels, called “**rings**”
- Most commonly:
 - ring 0 (kernel mode)
 - rings 1 and 2 (elevated system processes)
 - ring 3 (user mode)
- Extra rings help support **virtual machines** (more later)



Hardware Timers

- Hardware components that sit next to the CPU core and periodically interrupt it
 - Whatever the process is doing, control jumps to the kernel's timer interrupt handler
 - Handler checks if current user process has been running too long
 - Puts it to sleep if so
 - Returns control to program if not
- Interrupt frequency set by the kernel
 - Not by user code!
- Interrupts can be temporarily deferred by kernel
 - Not by user code!
 - Crucial for implementing mutual exclusion

User->Kernel Mode Switch

- Many scenarios, but all the same underlying hardware mechanism:
 - Interrupts: triggered by timer and I/O devices
 - *Hardware* exceptions: triggered by unexpected or malicious application behavior (eg, divide-by-zero)
 - System calls: “protected procedure call,” requesting the kernel to do something for the app

“**Trap**”: any user->kernel mode switch

- Idea: the number of places code can transition from user-to-kernel is **very small** and written **very carefully**
- Big fan-in/fan-out
- Key tool of  **OS-As-Referee** 

Some Exceptions

Table 6-1. Exceptions and Interrupts

Vector	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. ¹
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.

[Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1](#) (p 169)

Some More Exceptions

Table 6-1. Exceptions and Interrupts (Contd.)

Vector	Mnemonic	Description	Source
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. ²
18	#MC	Machine Check	Error codes (if any) and source are model dependent. ³
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction ⁴
20	#VE	Virtualization Exception	EPT violations ⁵
21	#CP	Control Protection Exception	The RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at the target of an indirect call or jump.
22-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

[Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1](#) (p 169)

UNIX systems calls (1972)

- Mount, unmount filesystem
- File open/create
- Seek to a file offset
- Read/write file
- Close, delete file
- Change file owner/mode
- Create/delete directory
- Add link to file
- ioctl for I/O device operations
- Process fork
- Process exec
- Exit process; wait for exit
- Create pipe between processes
- Send signal to another process
- Set/mask signal handler
- Extend heap memory region

Nifty trick: “Trap and Emulate”

- OS technique for providing functionality beyond what hardware provides by itself
- Primary tool for ✨ **OS-as-Illusionist** ✨
- Examples:
 - Handle a divide by zero error
 - Emulate floating point unit on hardware without one
 - Interacting with memory regions that haven’t been used yet