Operating Systems
Lecture 2

Boot, Process, Kernel

September 13, 2023
Prof. Mengwei Xu

Recap of Last Course

- OS is a bridge between hardware and apps/users.
- OS is a special software layer that provides and manages the access from apps/users to hardware resources (CPU, memory, disk, etc).

Recap of Last Course

- OS is a bridge between hardware and apps/users.
- OS is a special software layer that provides and manages the access from apps/users to hardware resources (CPU, memory, disk, etc).

- OS is the referee, illusionist, and glue.
- Learning OS is important, useful, and cool.

Recap of Last Course

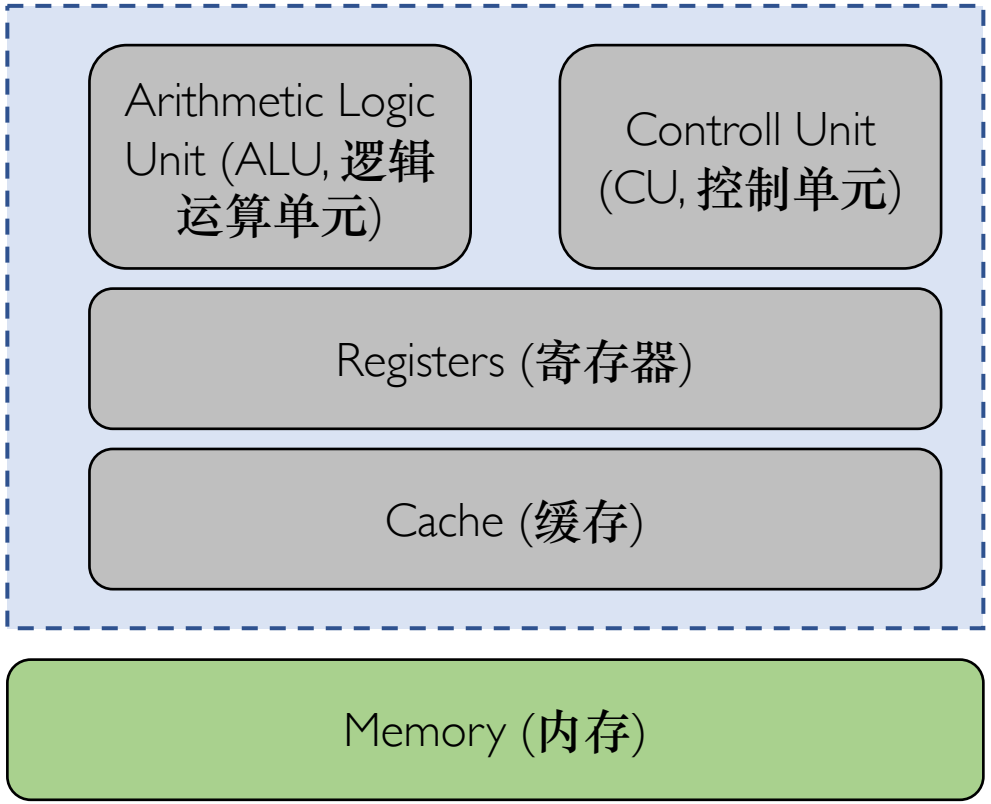
- OS is a bridge between hardware and apps/users.
- OS is a special software layer that provides and manages the access from apps/users to hardware resources (CPU, memory, disk, etc).

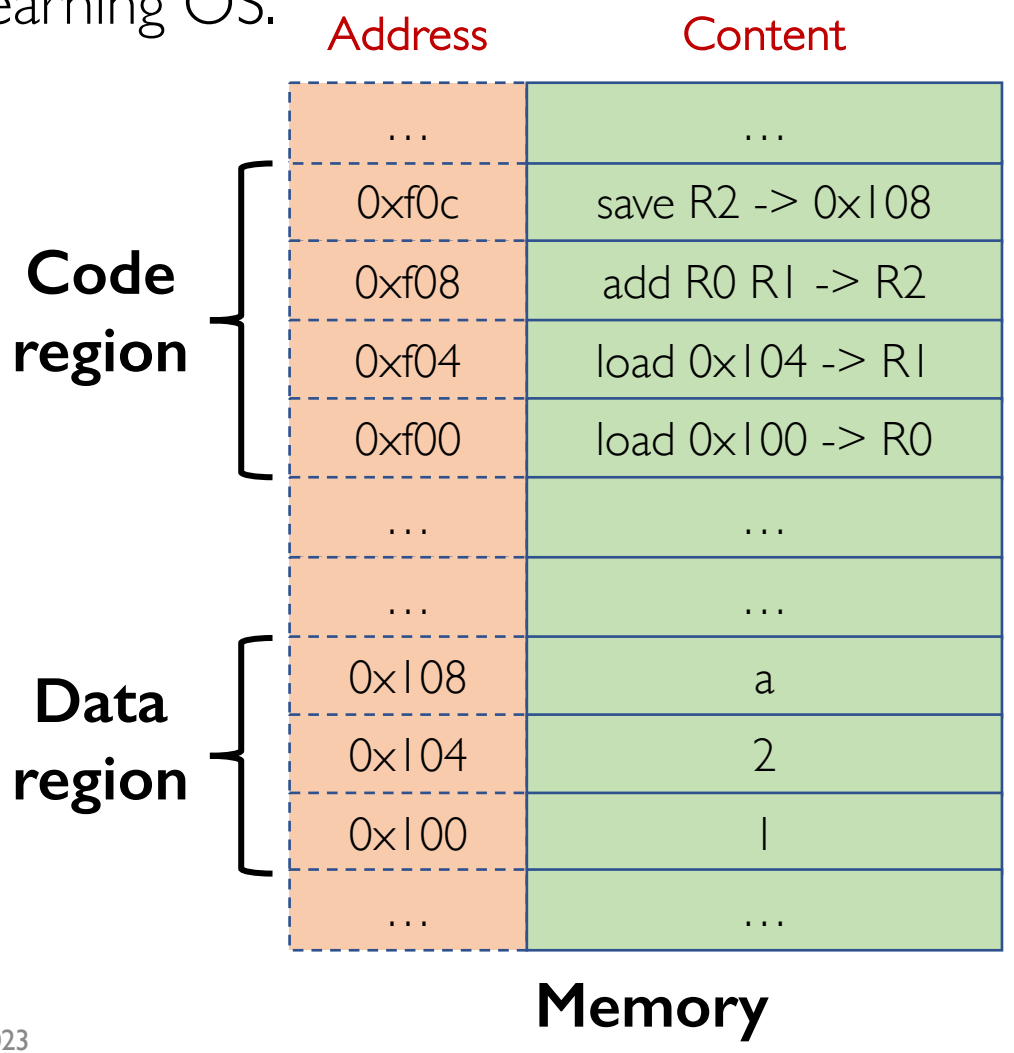
- OS is the referee, illusionist, and glue.
- Learning OS is important, useful, and cool.

- OS evolution: Serial processing -> Simple Batch System -> Multi-programmed Batch Systems -> Time Sharing Systems
  - Adapting to new hardware and workloads

Recap of Last Course

- OS involves extensive interaction with hardware (especially CPU)
  - Understanding their interface is critical to learning OS.





Goals for Today

- How computer boots
- Process: concept and memory layout
- Dual mode: kernel space vs. user space

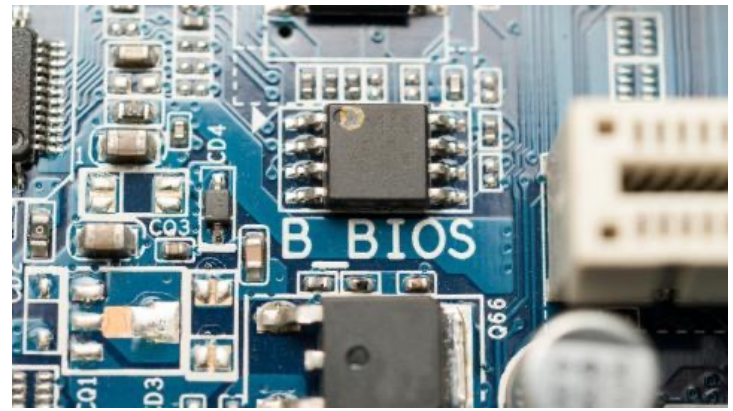
Goals for Today

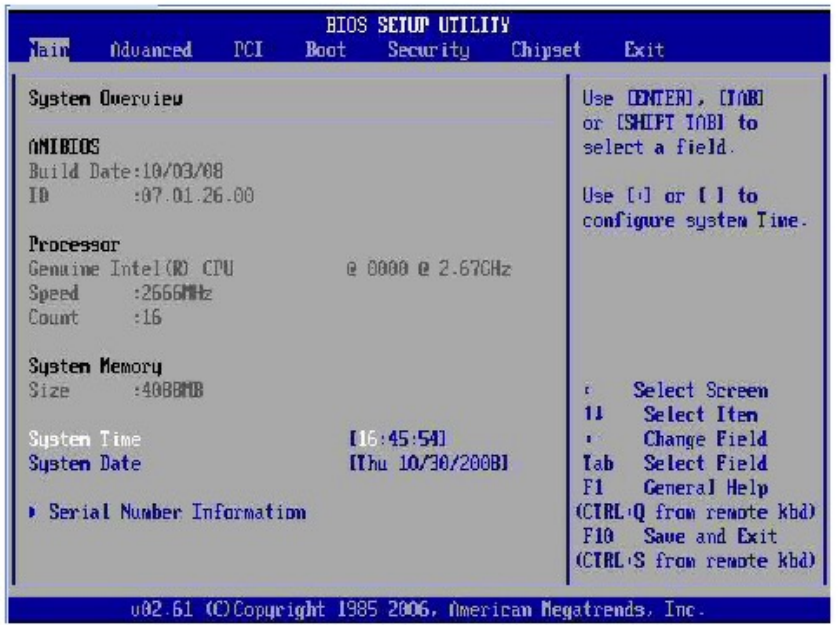
- How computer boots
- Process: concept and memory layout
- Dual mode: kernel space vs. user space

BIOS

- Basic Input Output System (BIOS)
  - A firmware (固件, vs. hw/system/software)
  - Stored on ROM on motherboard

1. Power-on self-test (POST) diagnostics
2. Identify attached hardware and initialize their states
   - VGA display, keyboards, etc..
3. Build HW description for advanced configuration and power interface (ACPI)
   - Defines HW interface between BIOS and OS
4. Load a bootloader from disk to memory
   - Usually the first disk sector (512 bytes)
5. Transfer the control to the bootloader
   - Setting %cs and %ip





Bootloader

- Part of OS

1. Switch from real mode (实模式) to protected mode (保护模式)
   - See next slide
2. Check if kernel image is okay
3. Loading kernel from disk to memory
   - Sector by sector
4. Transfer the control to the "real" OS

Real Mode vs. Protected Mode

- It's about different status the CPU works at
- Historical baggage: CPU needs backward compatibility

Q. Compare Real Mode & Protected mode

SN	Real Mode	Protected mode
01	It this mode processor works as 8086/8088.	It this processor works in full capacity
02	It has only 1MB memory addressing capability	It has more than 1MB to few GB memory addressing capability
03	It handles only one task.	It handles multiple tasks at a time.
04	In this memory address translation not required.	In this memory address translation required.
05	It directly communicate with ports & devices.	It directly communicate with ports & devices through OS.
06	This mode not supports memory management.	This mode supports memory management.
07	It supports less addressing modes & instructions.	It supports more addressing modes & instructions.
08	This mode is for backward capability to support 8086/8088.	This mode processor works in its real power.

Real Mode vs. Protected Mode

- It's about different status the CPU works at
- Historical baggage: CPU needs backward compatibility

6. Pentium 4

Comparison of 8086, 80386, Pentium-I, II and III

Sr. No	Features	8086	80386	Pentium-I	Pentium-II	Pentium-III
1.	Year of Release	1978	1985	1993	1997	1999
2.	Processor Size	16 Bit	32 Bit	32 Bit	32 Bit	32 Bit
3.	Data Bus	16 Bit	32 Bit	64 Bit	64 Bit	64 Bit
4.	Address Bus	20 Bit	32 Bit	32 Bit	32 Bit	32 Bit
5.	Memory Banks	2	4	8	8	8
6.	Memory Size	1 MB	4 GB	4 GB	64 GB	64GB
7.	Pipeline Stages	2	3	5	17	15
8.	ALU Size	16 Bit	32 Bit	32 Bit	32 Bit	32 Bit
9.	Number of Transistors	29 K	275 K	3.1 M	7.5 M	9.5 M
10.	Operating Frequency	6 MHz	33 MHz	100 MHz	450 MHz	450-1400 MHz

BIOS vs. Bootloader

BIOS	Bootloader
Firmware, comes with HW	Software, comes with (or part of) OS
The first software that runs since power on	The first user-defined or user-changeable software that runs since power on
Usually stored on ROM and not changeable	Stored with OS (hard disk, USB, etc)

You must wonder..

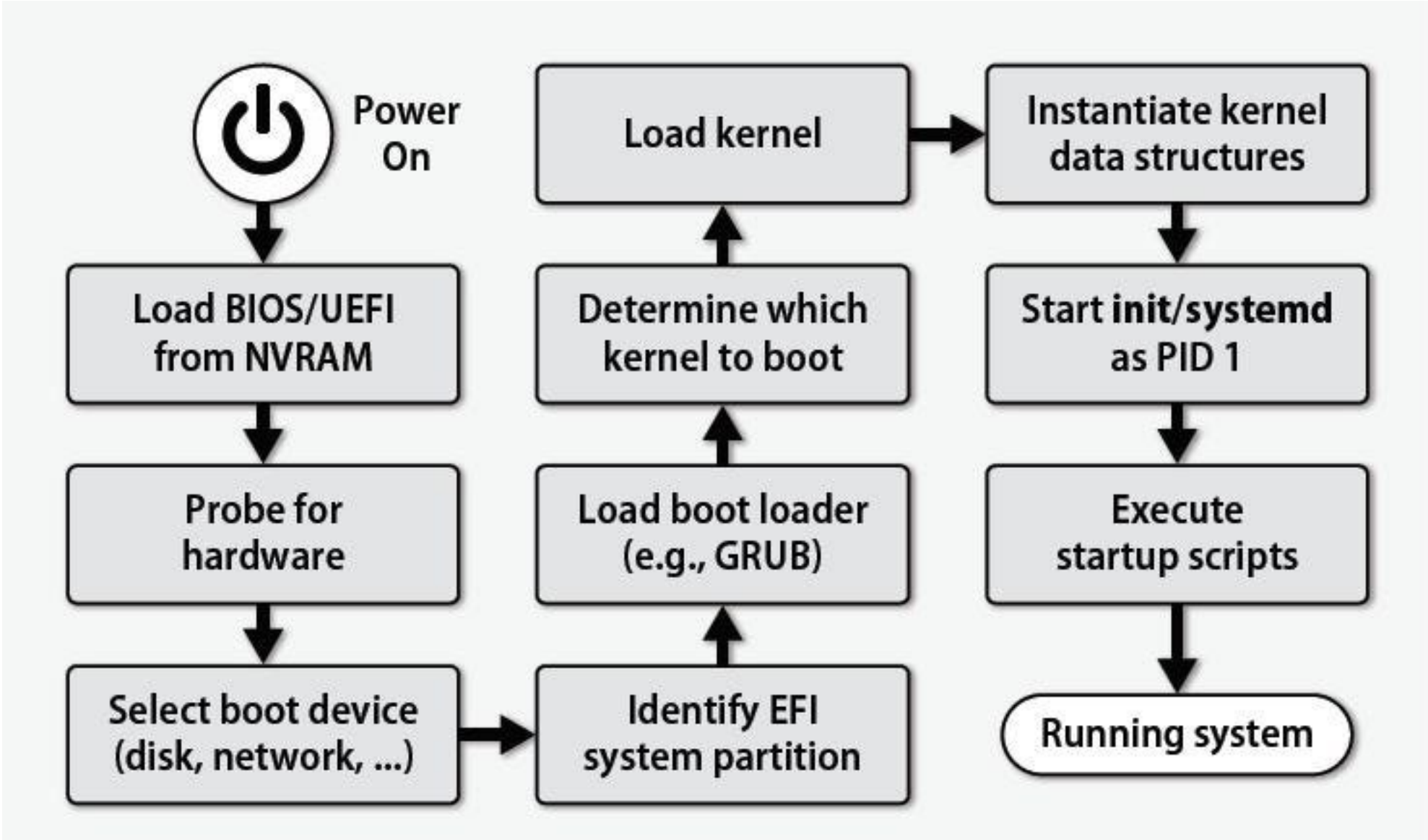
- Why BIOS does not directly load the kernel?

You must wonder..

- Why BIOS does not directly load the kernel?
  - Flexibility and Compatibility
  - Boot Device Detection
  - Boot Manager Functionality
  - Security and Verification
  - Error Handling
  - Ease of Updates

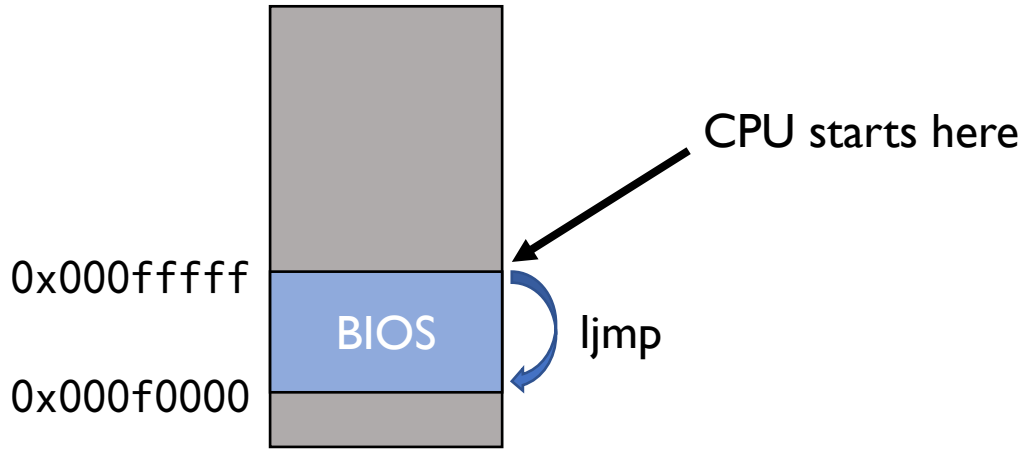
  - The above are summarized by ChatGPT

A Summary of Booting Process



Case Study: Booting of JOS (1/3)

- [f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
  - The first instruction run by CPU
  - Observed through GDB
  - What we learned from it?
    - The IBM PC starts executing at physical address 0x000ffff0, which is at the very top of the 64KB area reserved for the ROM BIOS.
    - The PC starts executing with CS = 0xf000 and IP = 0xfff0.
    - The first instruction to be executed is a jmp instruction, which jumps to the segmented address CS = 0xf000 and IP = 0xe05b.



Case Study: Booting of JOS (2/3)

```
.set PROT_MODE_CSEG, 0x8         # kernel code segment selector
.set PROT_MODE_DSEG, 0x10        # kernel data segment selector
.set CR0_PE_ON,      0x1         # protected mode enable flag

.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  cld                         # String operations increment

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

```
  # Switch from real to protected mode, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to their physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
  movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
  movw    %ax, %ss                # -> SS: Stack Segment

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call bootmain

  # If bootmain returns (it shouldn't), loop.
spin:
  jmp spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULL        # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
  SEG(STA_W, 0x0, 0xffffffff)         # data seg
```

Case Study: Booting of JOS (3/3)

```c
void
bootmain(void)
{
    struct Proghdr *ph, *eph;
    int i;

    // read 1st page off disk
    readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC)
        goto bad;

    // load each program segment (ignores ph flags)
    ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++) {
        // p_pa is the load address of this segment (as well
        // as the physical address)
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
        for (i = 0; i < ph->p_memsz - ph->p_filesz; i++) {
            *((char *) ph->p_pa + ph->p_filesz + i) = 0;
        }
    }

    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void)) (ELFHDR->e_entry))();

bad:
    outw(0x8A00, 0x8A00);
    outw(0x8A00, 0x8E00);
    while (1)
        /* do nothing */;
}
```

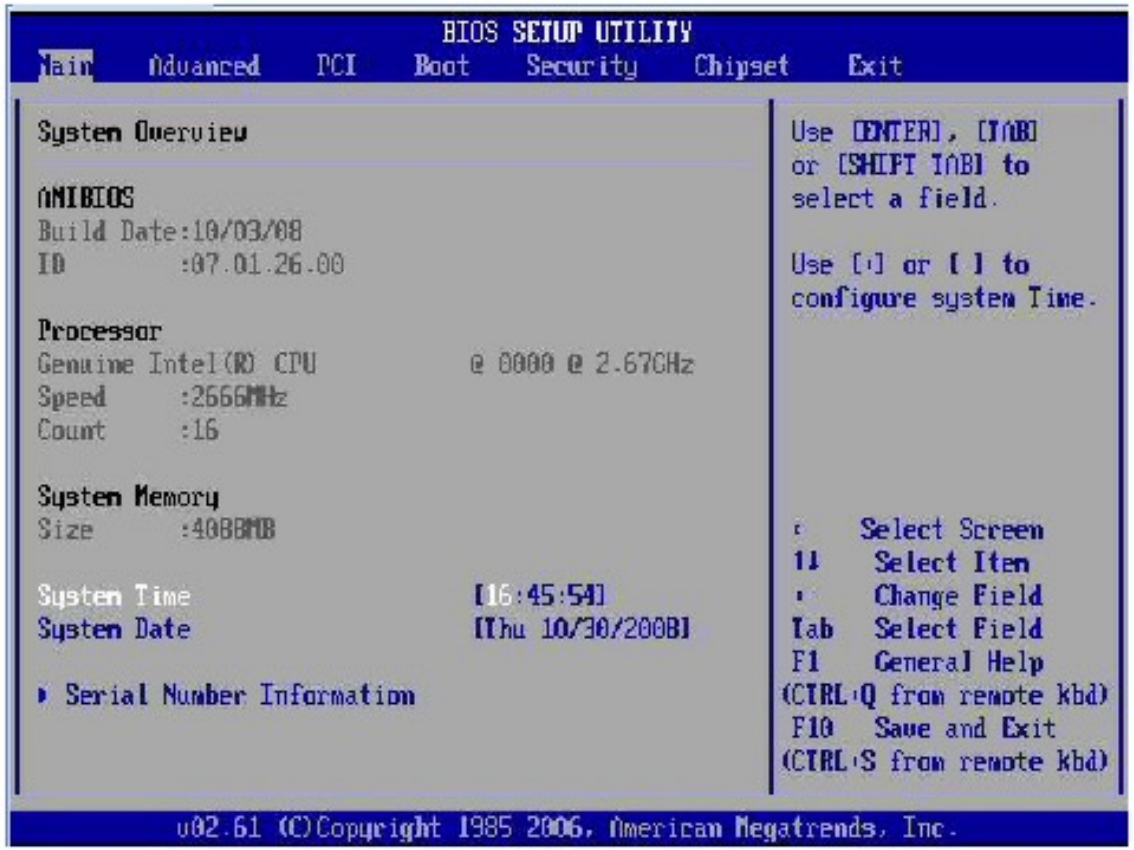
```c
// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked
void
readseg(uint32_t pa, uint32_t count, uint32_t offset)
{
    uint32_t end_pa;

    end_pa = pa + count;

    // round down to sector boundary
    pa &= ~(SECTSIZE - 1);

    // translate from bytes to sectors, and kernel starts at sector 1
    offset = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    while (pa < end_pa) {
        // Since we haven't enabled paging yet and we're using
        // an identity segment mapping (see boot.S), we can
        // use physical addresses directly.  This won't be the
        // case once JOS enables the MMU.
        readsect((uint8_t*) pa, offset);
        pa += SECTSIZE;
        offset++;
    }
}
```

UEFI

- Unified Extensible Firmware Interface (**可扩展固件接口**, UEFI)
- A successor of BIOS
  - It's faster
  - It has filesystem support
  - It can be stored in various places: flash memory on motherboard, hard drive, or even network share
  - It supports more input such as mouse
  - It has secure boot
  - It has better UI
  - Somehow it's more like a "mini OS"

UEFI

- Unified Extensible Firmware Interface (**可扩展固件接口**, UEFI)



Goals for Today

- How computer boots
- Process: concept and memory layout
- Dual mode: kernel space vs. user space

Process (进程)

- Process: the execution of an application program with restricted rights
  - Protection from each other; OS protected from processes
  - Owns dedicated Address Space (later)
  - Contexts of file descriptors, filesystem, etc..
  - One or many threads (later)

- How process differs from program?
  - Process is an *instance* of program, like an object is an *instance* of a class in OOP
  - A program can have zero, one, or many processes executing it

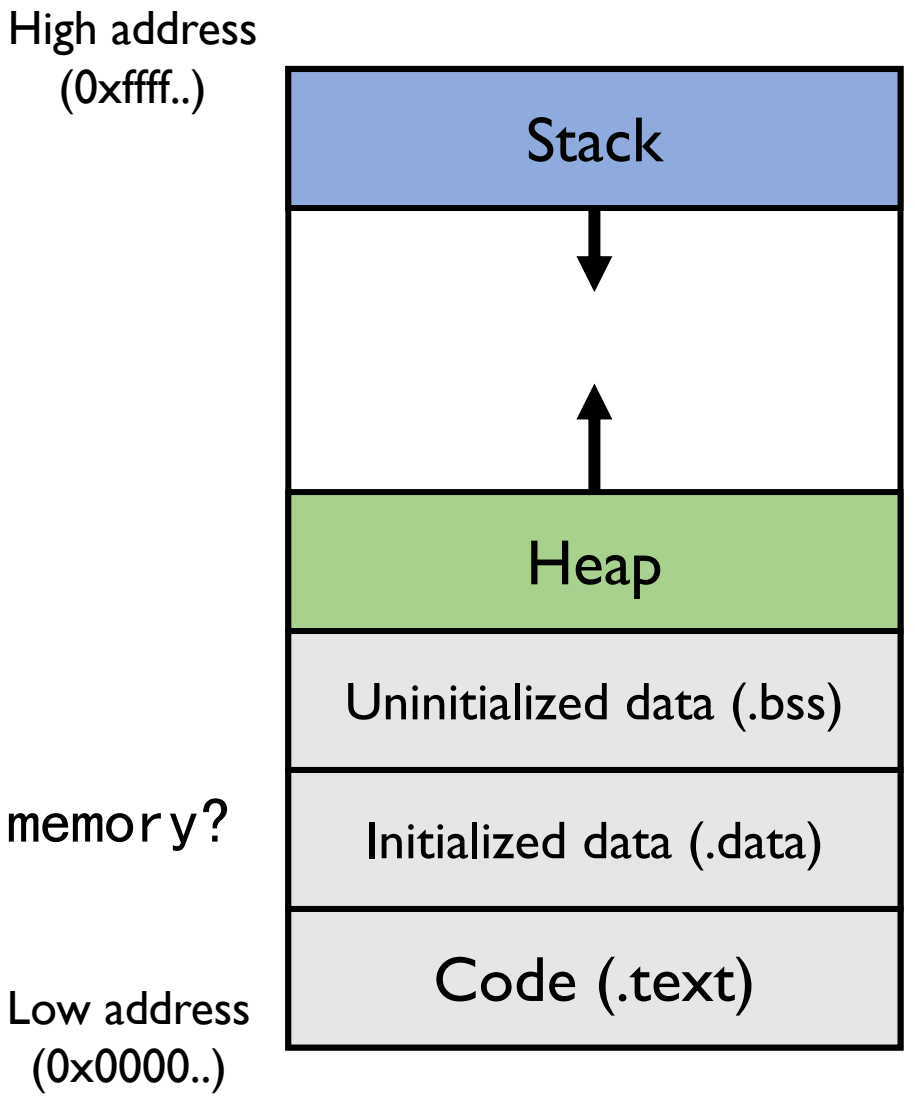
Process Control Block

- Process Control Block (PCB, 进程控制块): a data structure used by Linux to keep track of a process execution
  - Process ID (PID)
  - Process state (running, ready, waiting..)
  - Process priority
  - Program counter
  - Memory related information
  - Register information
  - I/O status information (file descriptors, I/O devices..)
  - Accounting information
- In what case these information is used?

Process in Memory

```
int a = 0;
int b;
void hello() {
    static c = a + b;
    int d;
    int*e = malloc(..);
}
```

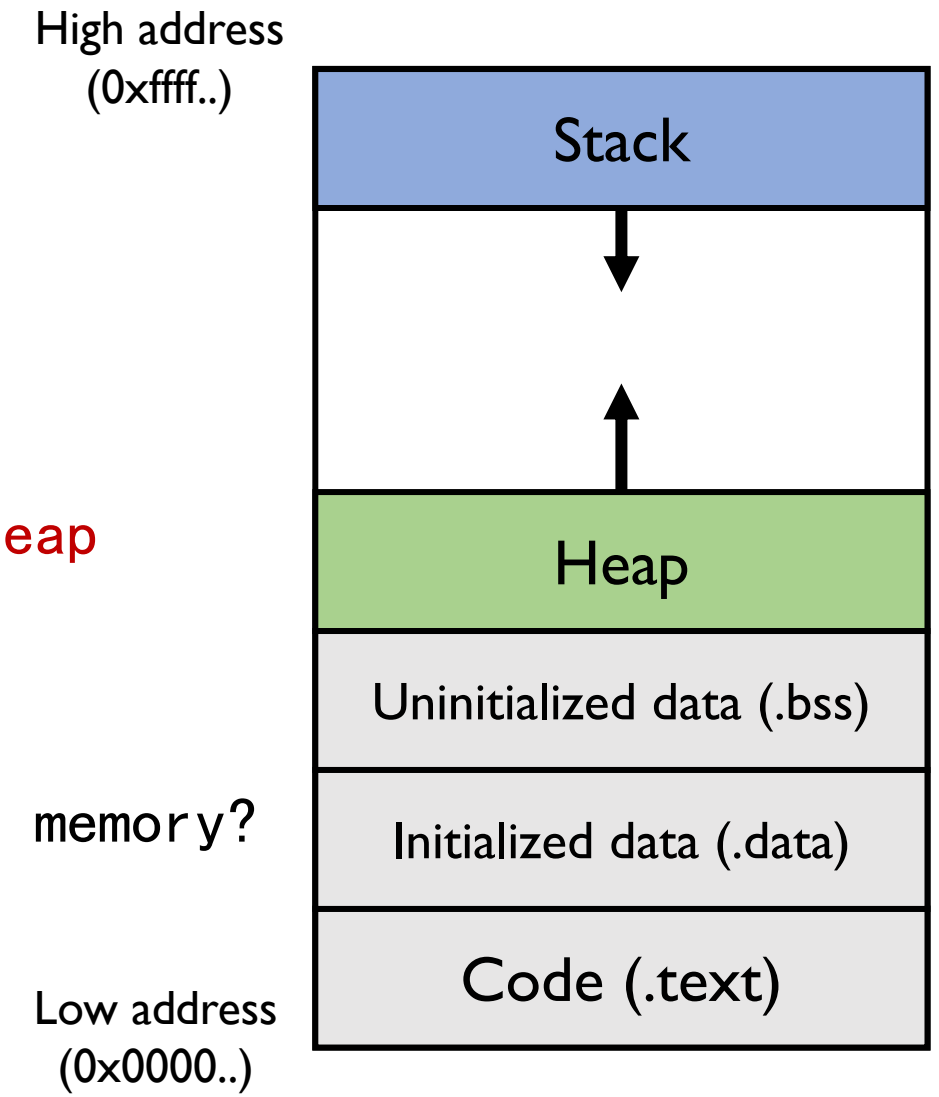
Where are a/b/c/d/e stored in memory?



Process in Memory

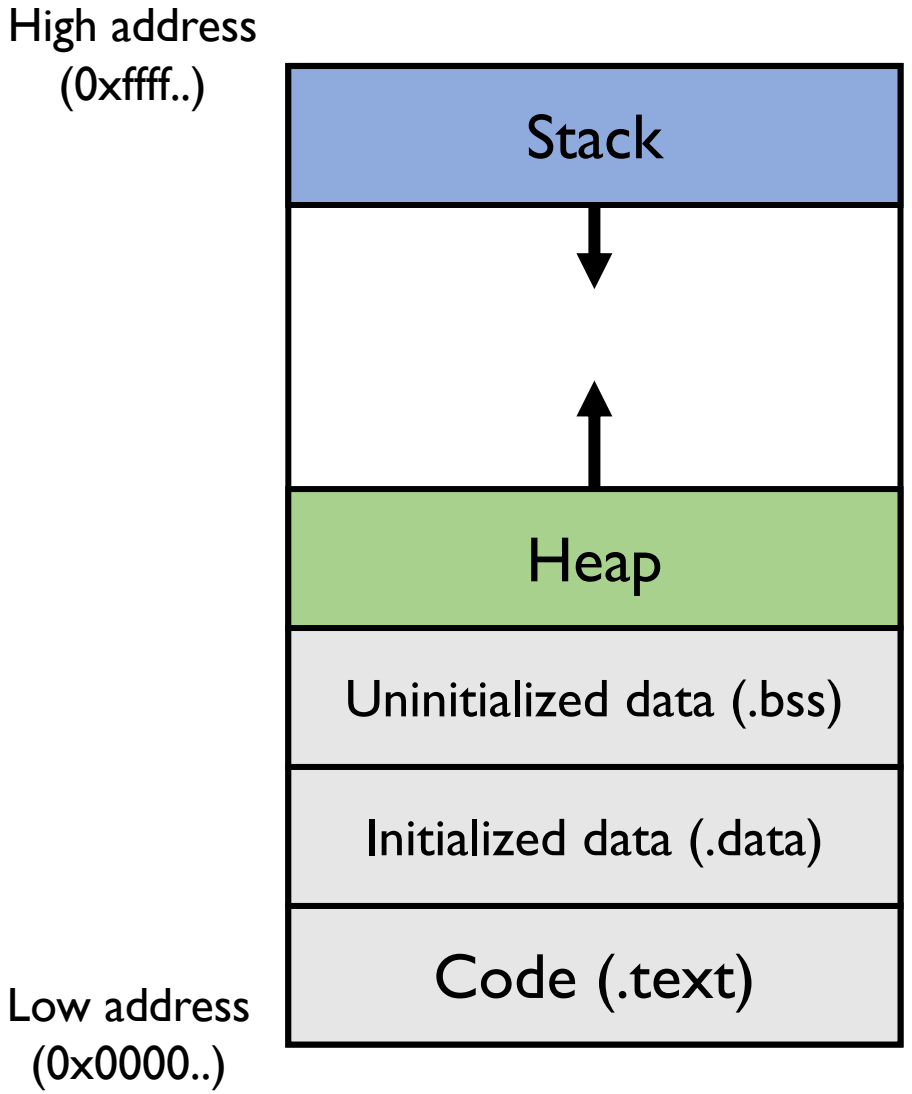
```
int a = 0; // .data
int b; // .bss
void hello() {
    static int c; // .bss
    int d; // stack
    int* e = malloc(..); // heap
}
```

Where are a/b/c/d/e stored in memory?



Process in Memory

- An executable mainly consists of bss, data, and code regions.
- Remember: this memory address is NOT physical!
  - Will learn how it's translated into physical address later.



Process in Memory

- Use *readelf* command to checkout what's in an executable

- The concrete output depends on the compiler
  - What optimization level?
  - Whether debug mode is enabled?
  - ..

```
[root@localhost test]$ readelf -S main
There are 29 section headers, starting at offset 0xca0:
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .gnu.hash         GNU_HASH        08048168 000168 000030 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          08048198 000198 0000d0 10   A  5   1  4
  [ 5] .dynstr           STRTAB          08048268 000268 000183 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          080483ec 0003ec 00001a 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         08048408 000408 000060 00   A  5   2  4
  [ 8] .rel.dyn          REL             08048468 000468 000010 08   A  4   0  4
  [ 9] .rel.plt          REL             08048478 000478 000048 08   A  4  11  4
  [10] .init             PROGBITS        080484c0 0004c0 000017 00  AX  0   0  4
  [11] .plt              PROGBITS        080484d8 0004d8 0000a0 04  AX  0   0  4
  [12] .text             PROGBITS        08048580 000580 000268 00  AX  0   0 16
  [13] .fini             PROGBITS        080487e8 0007e8 00001c 00  AX  0   0  4
  [14] .rodata           PROGBITS        08048804 000804 00001a 00   A  0   0  4
  [15] .eh_frame_hdr     PROGBITS        08048820 000820 000044 00   A  0   0  4
  [16] .eh_frame         PROGBITS        08048864 000864 00010c 00   A  0   0  4
  [17] .ctors            PROGBITS        08049970 000970 00000c 00  WA  0   0  4
  [18] .dtors            PROGBITS        0804997c 00097c 000008 00  WA  0   0  4
  [19] .jcr              PROGBITS        08049984 000984 000004 00  WA  0   0  4
  [20] .dynamic          DYNAMIC         08049988 000988 0000e0 08  WA  5   0  4
  [21] .got              PROGBITS        08049a68 000a68 000004 04  WA  0   0  4
  [22] .got.plt          PROGBITS        08049a6c 000a6c 000030 04  WA  0   0  4
  [23] .data             PROGBITS        08049a9c 000a9c 000004 00  WA  0   0  4
  [24] .bss              NOBITS          08049aa0 000aa0 000098 00  WA  0   0  8
  [25] .comment          PROGBITS        00000000 000aa0 000114 00      0   0  1
  [26] .shstrtab         STRTAB          00000000 000bb4 0000e9 00      0   0  1
  [27] .symtab           SYMTAB          00000000 001128 000510 10     28  53  4
  [28] .strtab           STRTAB          00000000 001638 0003f4 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Goals for Today

- How computer boots
- Process: concept and memory layout
- Dual mode: kernel space vs. user space

Isolation by OS

- How can we protect malicious/harmful behaviors from process?
  - Accessing (or even modifying) the data of other process (or even OS)
  - Let's brainstorm: what evil things can be done without such protection/isolation?

- A straightforward way: let OS take charge of each instruction of process
  - A simulation (or virtual machine) way
  - Too slow
  - Can we do it in hardware?

Dual Mode

- Hardware-assisted isolation and protection
  - User mode (用户态) vs. kernel mode (内核态)
  - Teachers & TAs are in ?? mode, while students are in ?? mode

- What hardware needs to provide?
  - Privileged instructions (特权指令)
  - Memory protection
  - Timer interrupts
  - Safe mode transfer (in next course)

Privileged Instructions (1/3)

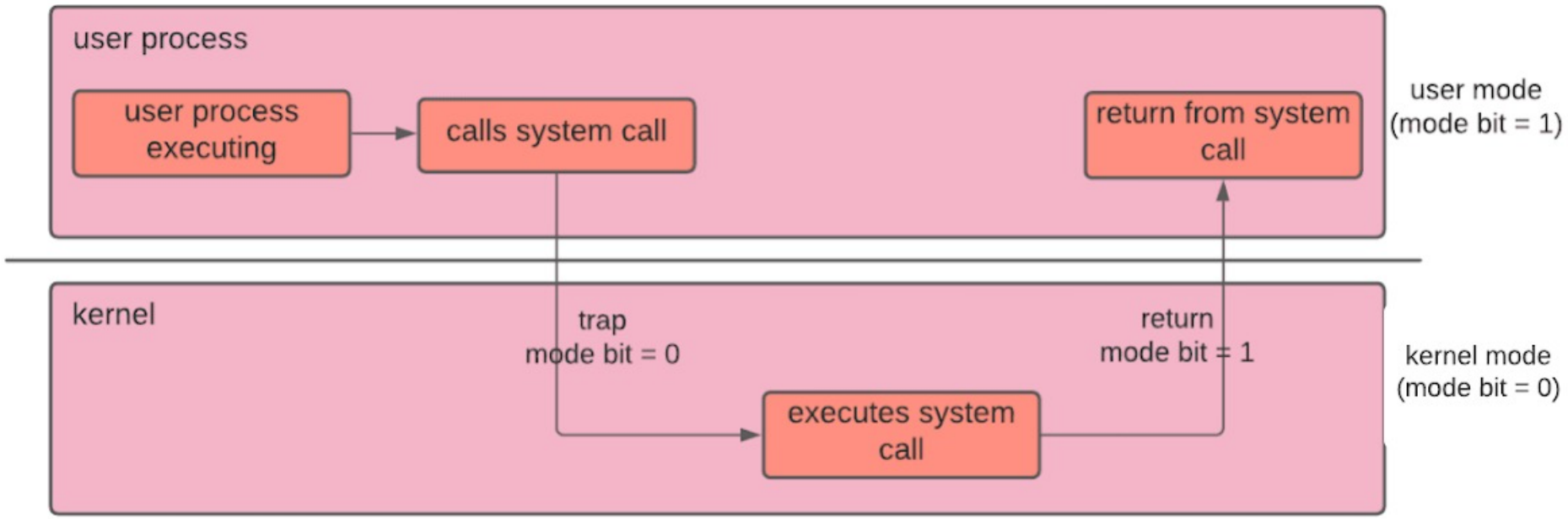
- Instructions available in kernel mode but not user mode

Privileged Instructions	Non-privileged Instructions
I/O read/write	Performing arithmetic operations
Context switch	Call a function
Changing privilege level	Reading status of processor
Set system time	Read system time
..	..

Any instructions that could affect other processes are likely to be privileged.

Privileged Instructions (2/3)

- What if apps need those privileged instructions?
  - Will learn the details later



Privileged Instructions (3/3)

- What if app executes a privileged instruction without permission?
  - Processor detects it in its hardware logic, and throws an exception (next course)
  - Process halted, OS takes over

Privileged Instructions (3/3)

- What if app executes a privileged instruction without permission?
  - Processor detects it in its hardware logic, and throws an exception (next course)
  - Process halted, OS takes over
- Demonstration with assembly code
  - Demonstration without assembly code (e.g., in pure C) is challenging

```
rtos@localhost:~/test $ ./a.out
Illegal instruction
rtos@localhost:~/test $ cat t.c
#include <stdio.h>

int main() {
    int cpsr;

    // Attempt to execute a privileged instruction (MRS - Move to Register from Status)
    // This instruction is only allowed in privileged modes (kernel mode).
    __asm__ __volatile__ ("MRS %0, s3_3_c13_c2_1" : "=r" (cpsr));

    // This code will execute after the privileged instruction above
    // without causing a compilation error.
    printf("Hello, World!\n");

    return 0;
}
rtos@localhost:~/test $
```

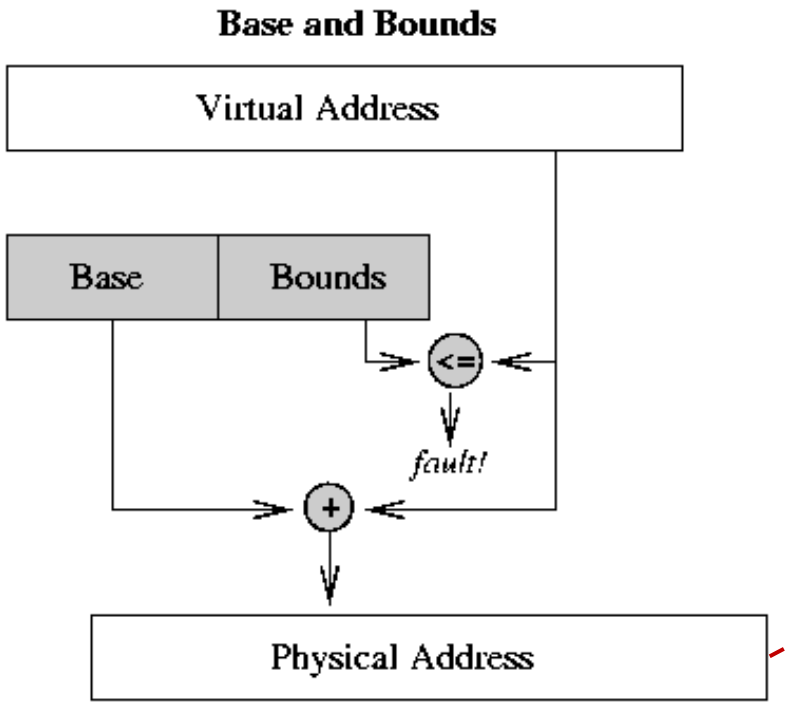
Privileged Instructions (3/3)

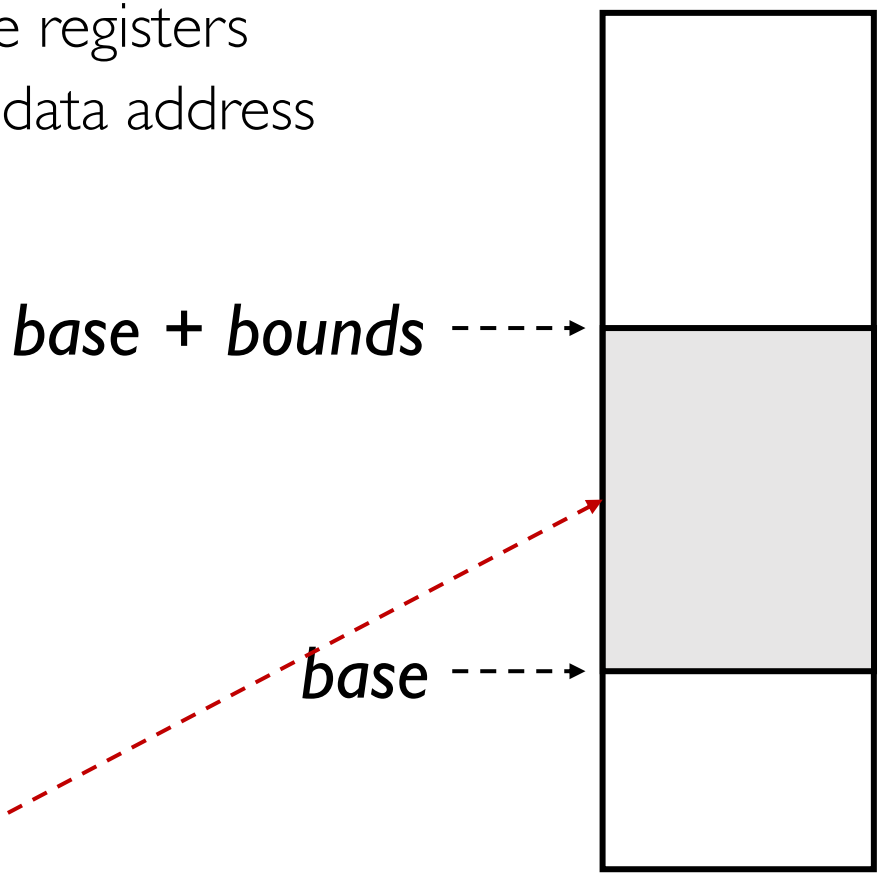
- What if app executes a privileged instruction without permission?
  - Processor detects it in its hardware logic, and throws an exception (next course)
  - Process halted, OS takes over
- Demonstration with assembly code
  - Demonstration without assembly code (e.g., in pure C) is challenging

- The concept of ``execution permission'' also extends to app-level as well, e.g., in Android system that each app requests permissions for read/write, network, etc..
  - How is such permission checking achieved? The same as privileged instructions?

Memory Protection (1/5)

- Segmentation (分段) approach: *base* and *bounds* registers
  - Every memory access is checked on those registers
  - A block copy needs to check each of the data address
  - Kernel mode bypasses this check

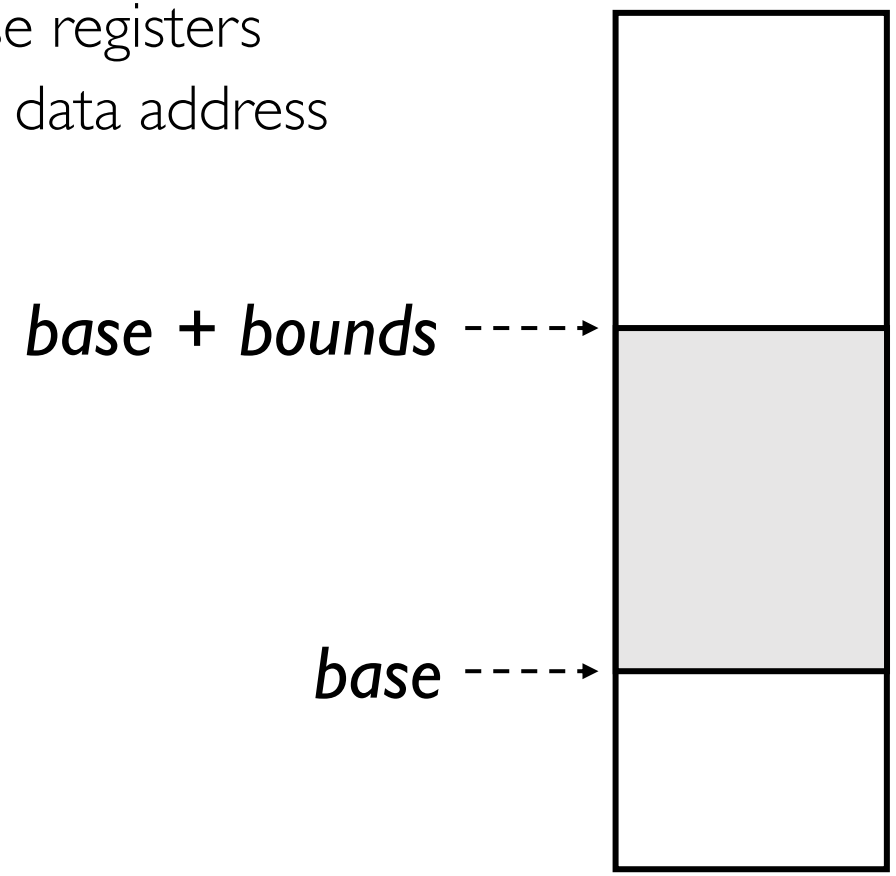




Memory Protection (1/5)

- Segmentation (分段) approach: *base* and *bounds* registers
  - Every memory access is checked on those registers
  - A block copy needs to check each of the data address
  - Kernel mode bypasses this check

- The disadvantages:
  - No expandable heap and stack
  - No memory sharing
  - Memory fragmentation
  - Etc..



Memory Protection (2/5)

- Paging (分页): every memory address a process sees is "*discontinuously*" mapped to a physical address in memory
  - Probably the most important and beautiful concept in OS
  - Involves extensive software-hardware cooperation

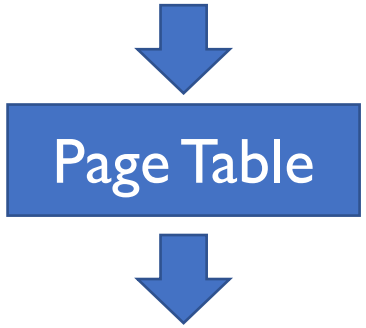
- How to translate virtual address to physical address is determined by OS in kernel mode

- The actual translation process and permission check is done by CPU

Memory Protection (2/5)

- Paging (分页): every memory address a process sees is "*discontinuously*" mapped to a physical address in memory
  - Probably the most important and beautiful concept in OS
  - Involves extensive software-hardware cooperation

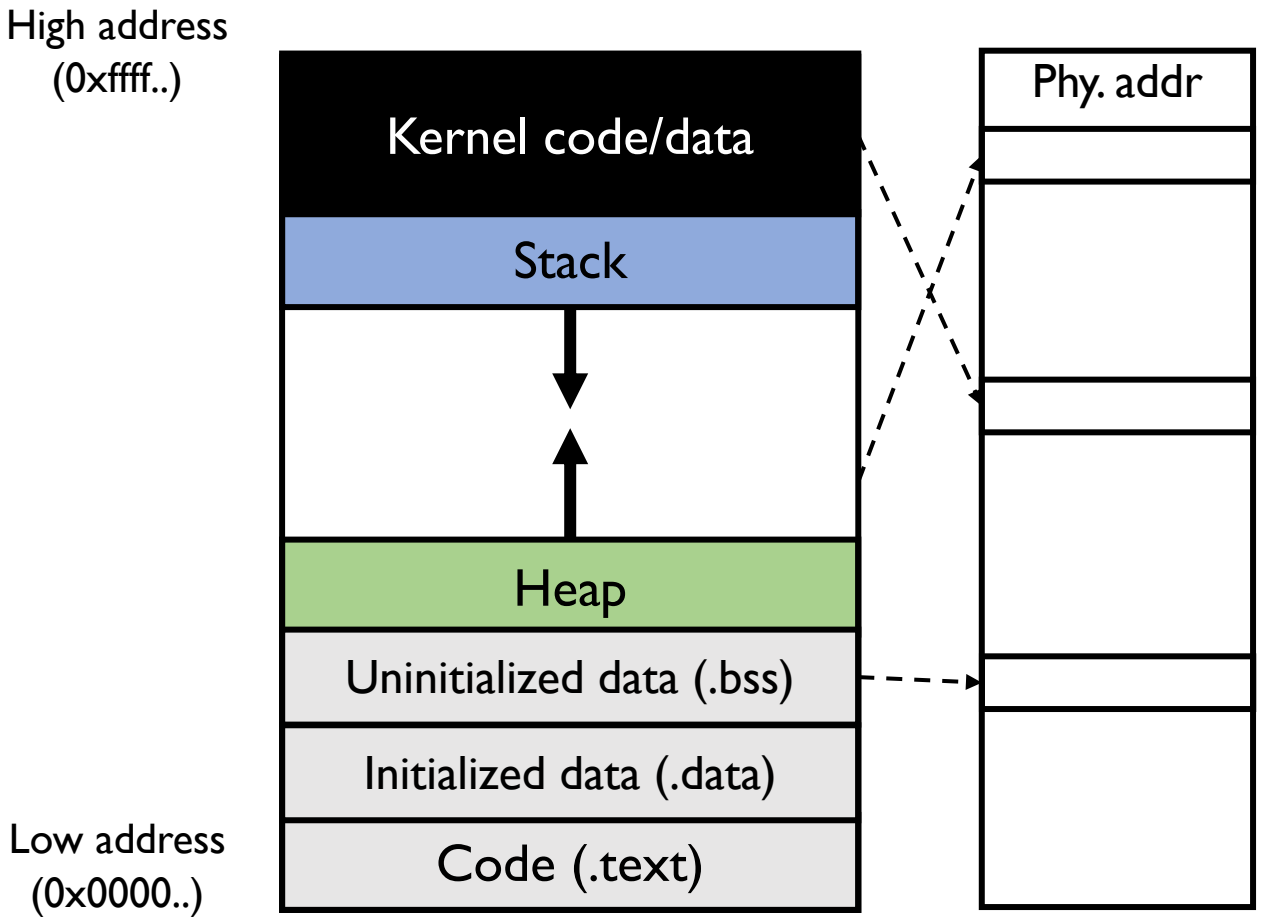
- How to translate virtual address to physical address is determined by OS in kernel mode



- The actual translation process and permission check is done by CPU

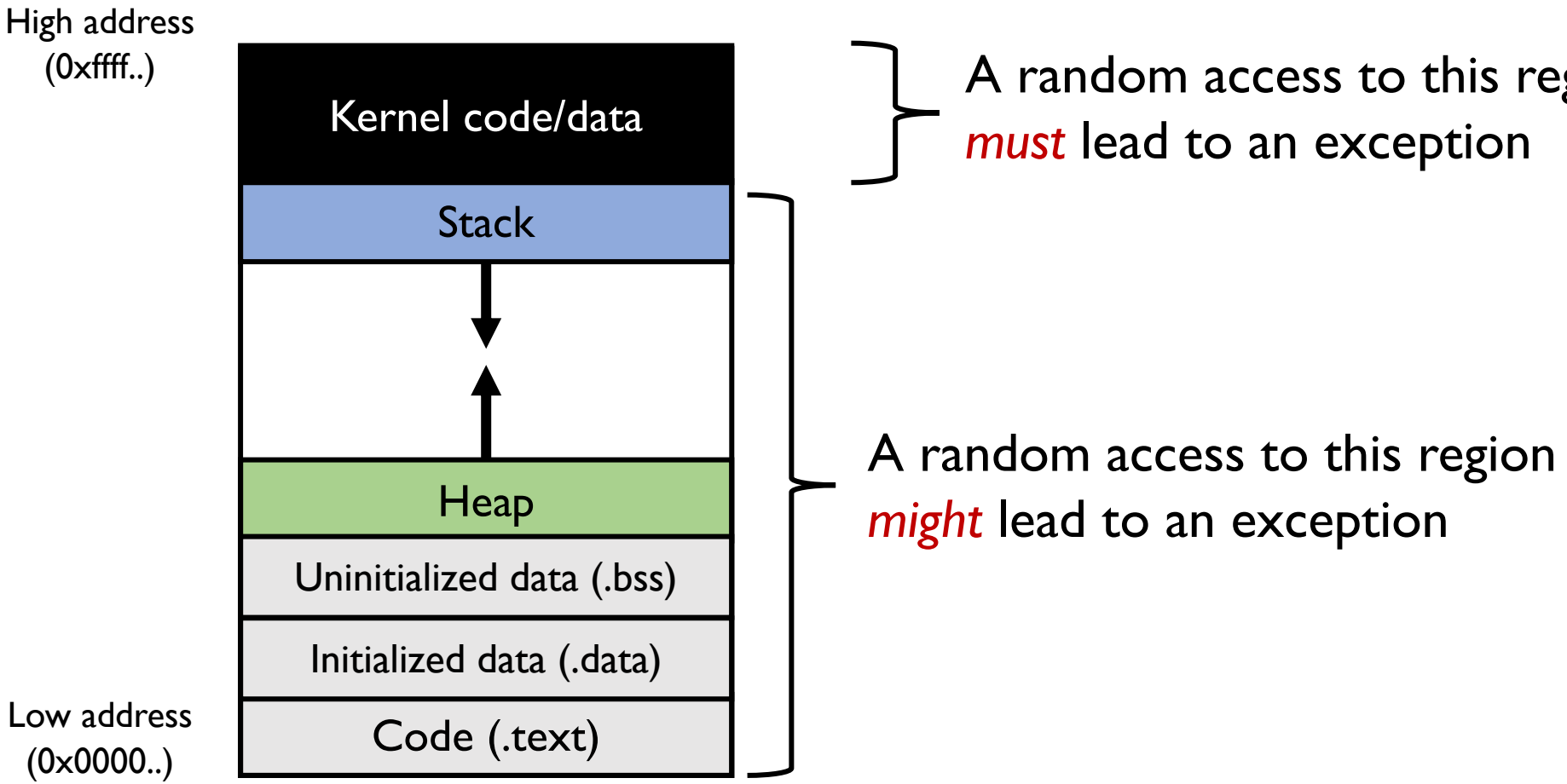
Memory Protection (3/5)

- Paging (分页): every memory address a process sees is "*discontinuously*" mapped to a physical address in memory



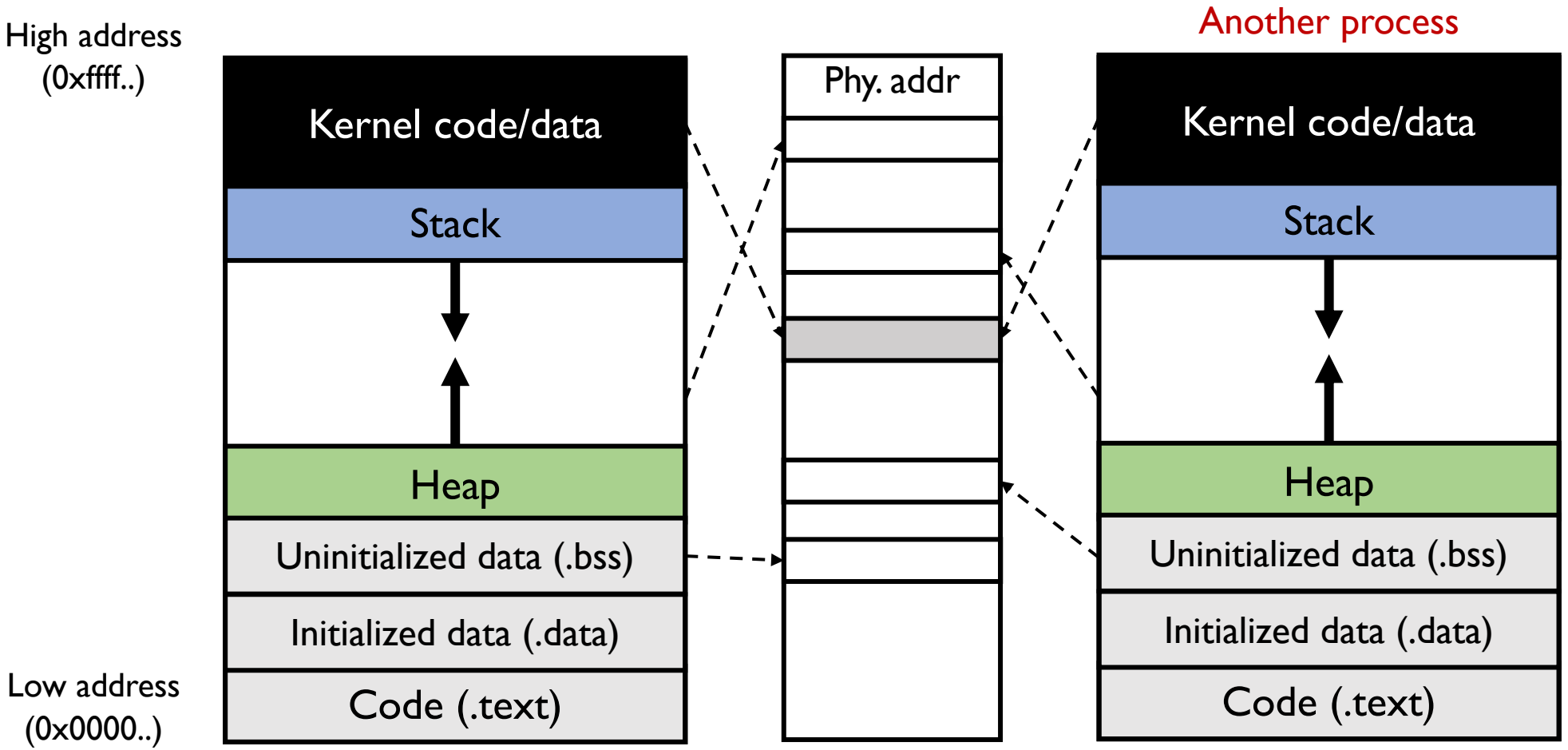
Memory Protection (4/5)

- Paging (分页): every memory address a process sees is "*discontinuously*" mapped to a physical address in memory



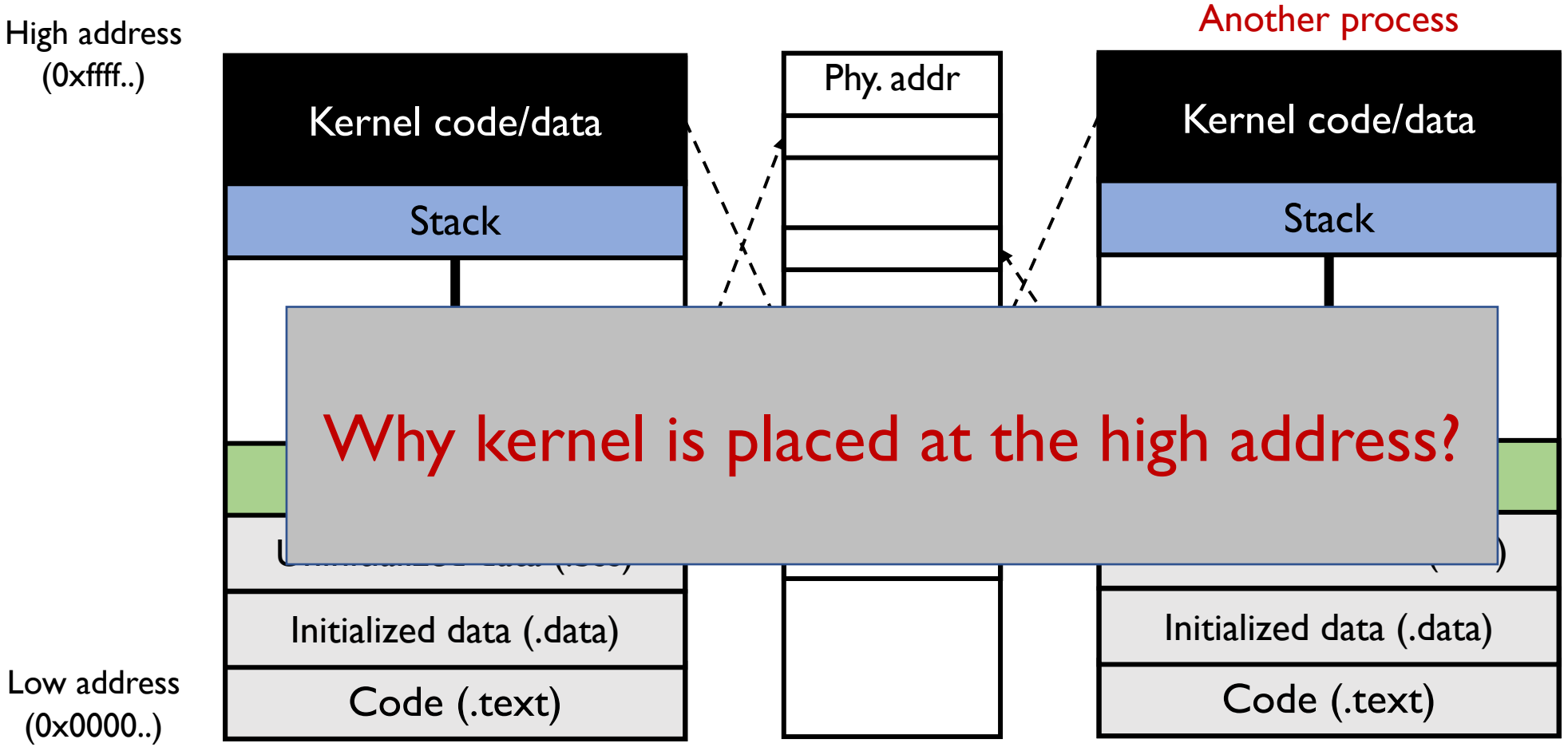
Memory Protection (5/5)

- Paging (分页): every memory address a process sees is "*discontinuously*" mapped to a physical address in memory



Memory Protection (5/5)

- Paging (分页): every memory address a process sees is "*discontinuously*" mapped to a physical address in memory



Timer Interrupts

- A way for OS to regain the control to the CPU
  - An illusion: the program has the full control of CPU
  - Otherwise, it can execute an infinite loop..
  - Hardware timer can only be reset in kernel mode

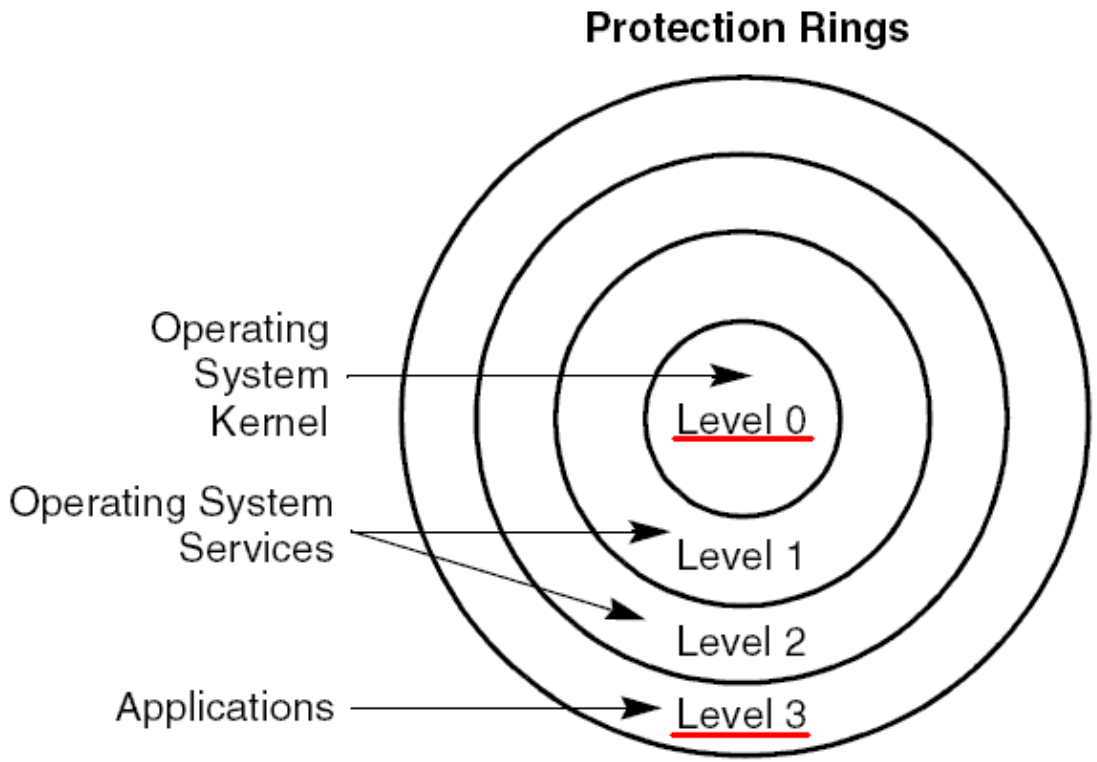
- After timer interrupts, the OS **schedules** another process (could be the same one being interrupted) to run

Current Privilege Level (CPL)

- x86 Architecture uses lower 2-bits in the CS segment register (referred to as the Current Privilege Level bits).
  - Yet most OSes only use level 0 (kernel mode) and level 3 (user mode).

Current Privilege Level (CPL)

- x86 Architecture uses lower 2-bits in the CS segment register (referred to as the Current Privilege Level bits).
  - Yet most OSes only use level 0 (kernel mode) and level 3 (user mode).

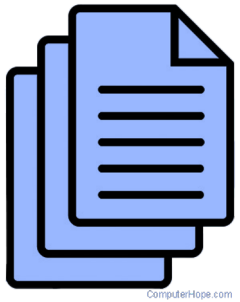


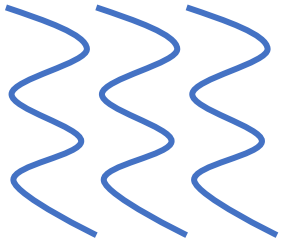
How to switch between user and kernel modes?

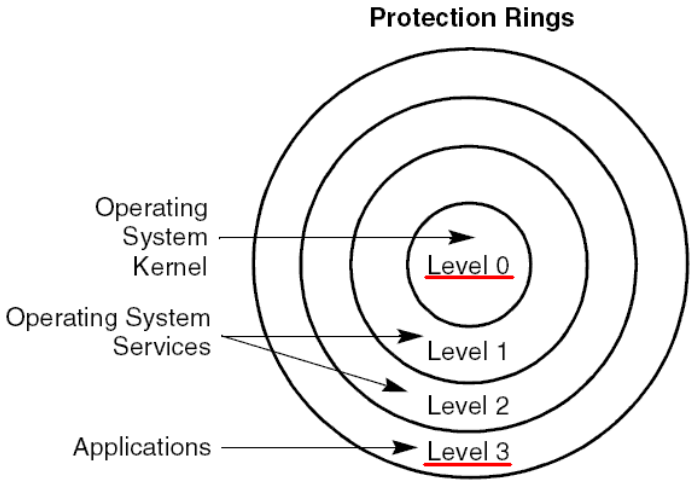
1. `CPL &= 0x0`
2. `CPL &= 0x3`
3. `CPL |= 0x0`
4. `CPL |= 0x3`
5. `CPL &= 0xfffffffc`
6. `CPL |= 0xfffffffc`

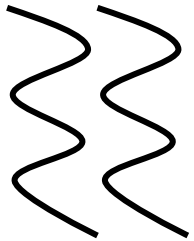
Concepts

- user/app code vs. user/app process vs. user mode
- OS code vs. system process vs. kernel mode









Code

Process

Mode

Some Interesting Questions

- Does user code always run in user process?
- Does user code always run in user mode?
- Does OS code always run in system process?
- Does OS code always run in kernel mode?
- How does code/CPU know if it's in user or kernel mode?

Homework – No Submission Required

- Try to answer the questions in last slide.
- Write some programs that encounter errors due to use of privileged instructions.
- Learn about ELF file format.