CS444/544 Operating Systems II

Prof. Sibin Mohan

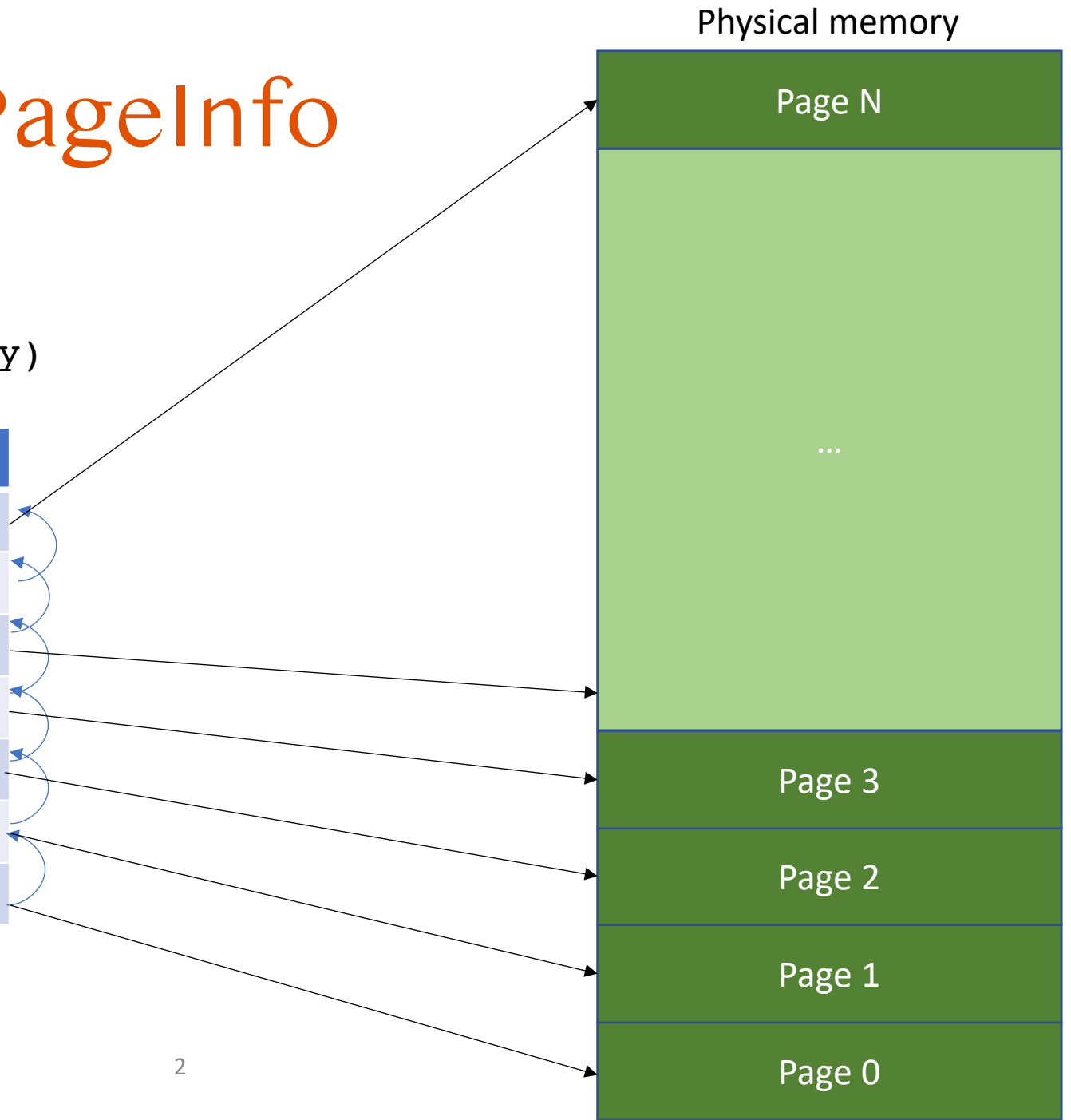
Spring 2022 | Lec6.1: Quiz 1 Review

Adapted from content originally created by: Prof. Yeongjin Jang

Recap: struct PageInfo

```
struct PageInfo * pages (array)
```

idx	pp_ref	pp_link
N	0	
…	0	
…	0	
3	0	
2	0	
1	0	
0	0	



Recap: struct PageInfo

- struct PageInfo *pp
  - The variable typed as struct PageInfo* will point to the address of a struct PageInfo object in pages array
- You can access
  - `pp->pp_ref`
  - `pp->pp_link`
- But you cannot access
  - Physical page via pp

`struct PageInfo * pages (array)`

idx	pp_ref	pp_link
N	0	
…	0	
…	0	
3	0	
2	0	
1	0	
0	0	

Recap: struct PageInfo

- How to get physical address from a struct PageInfo *pp?

- **`page2pa(pp)`**
- **`page2kva(pp)`**
  - Look at the implementation of those functions
  - e.g., **`(pp – pages) << PGSHIFT`**
  - why is this the physical address?
  - **Physical page number = `(pp – pages)`**

- `memset(`**`page2kva(pp)`**`, 0, PGSIZE)`
  - This will zero out corresponding physical page of pp

Checking PTE permissions

`pte_t *p_pte` → pointer to the PTE

`*p_pte` → access the content (values) in p_pte

- To check if p_pte is valid?

```
if ( (*p_pte) & PTE_P )
```

Permission Bit	Meaning	Interpretation
PTE_P	Page Exists [Present]	0: invalid entry 1: valid entry
PTE_W	Page is readable/writable	0: read only 1: writeable
PTE_U	Who can access it [User]	0: kernel [ring 0] only 1: user [ring 3] accessible

Checking PTE permissions

- Apply **bitwise & operation**
  - leave only the bit of our interest
  - **`PTE_P, PTE_W, PTE_U`**

- combine multiple checks

```
int perm = PTE_W | PTE_U | PTE_P
if ( ((*p_pte) & perm) == perm)
```

Permission Bit	Meaning	Interpretation
PTE_P	Page Exists [Present]	0: invalid entry 1: valid entry
PTE_W	Page is readable/writable	0: read only 1: writeable
PTE_U	Who can access it [User]	0: kernel [ring 0] only 1: user [ring 3] accessible

CAVEAT for Lab 2

- **`boot_map_region()`**

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
```

- Make sure that **`va + size`** **does not overflow the 32-bit limit**
- `va = 0xf0000000, size = 0x10000001`
- Then, **`va + size = 1`**

```
>>> va = 0xf0000000 # KERNBASE
>>> size = 0x10000001
>>> hex(va + size)
'0x100000001'
>>> hex((va + size) % 2**32) # in 32-bit machine, we only store 32 bits..
'0x1'
```

From KERNBASE to 2^32-KERNBASE

- **2's complement (signed) number system**
  - For a number N
  - **(2^32 – N) is –N**
- E.g.,
  - `N = 1`
  - `-1 == (int) 0xffffffff`
  - `N = 2`
  - `-2 == (int) 0xfffffffe`
- So, the size of that region is **`-KERNBASE`** in 32-bit

Assertion Errors in Lab2

- There are many **assertions** in the **`check_*()`** functions in lab2
- These functions will check the sanity of your implementation
- If your execution stops due to an assertion,
  - your code manages virtual and physical memory **incorrectly**

- Please try to understand **what each line of code is doing**
  - Don’t look only at the assertion itself
  - You need to know about the execution of the entire test code to debug it

https://gist.github.com/blue9057/5efb9807a9e879c75ee3f1c7add93c08

Learn about the implications of each line in the check_* functions

```
// should be able to map pp2 at PGSIZE because pp0 is
// already allocated for page table. The pde at kern_pgdir[0]
// will manage page table entries from address 0x0 to 0x400000.
// So PGSIZE is 0x1000, so it is still within the first page table
// (that is backed with the physical page of pp0).
assert(page_insert(kern_pgdir, pp2, (void*) PGSIZE, PTE_W) == 0);
// and pp2 must be the physical page for the virtual address at 0x1000
assert(check_va2pa(kern_pgdir, PGSIZE) == page2pa(pp2));
// and pp2 should have a reference count (referred by VA 0x1000).
assert(pp2->pp_ref == 1);
```

```
  assert((pp0 = page_alloc(0)));
  assert((pp1 = page_alloc(0)));
  assert((pp2 = page_alloc(0)));

  // pp0, pp1 and pp2 all should not be NULL and also
  // they need to be unique (pp0 != pp1, pp1 != pp2, pp0 != pp2)
  assert(pp0);
  assert(pp1 && pp1 != pp0);
  assert(pp2 && pp2 != pp1 && pp2 != pp0);

  // temporarily steal the rest of the free pages
  fl = page_free_list;
  page_free_list = 0;
```

USE **GDB**!!!

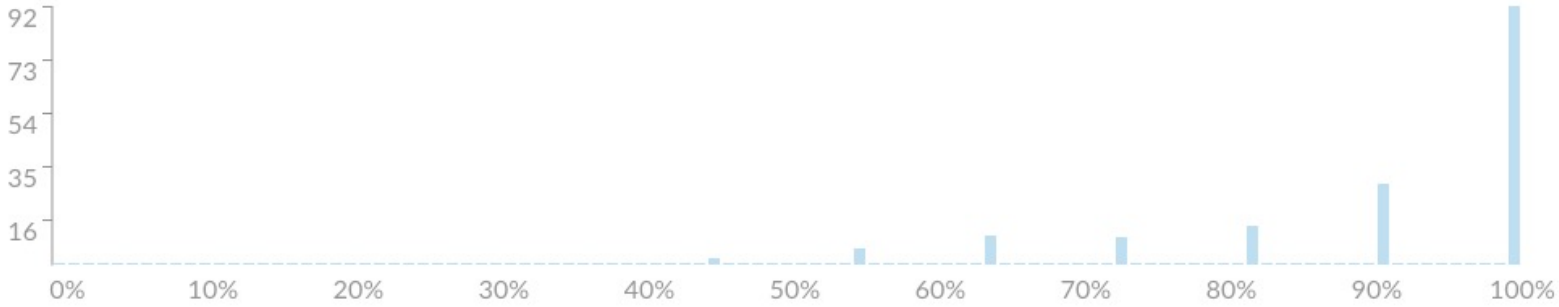
- I do not expect your code **will work in one shot**
- You will do lots of **debugging**
- You can do it with many **`cprintf()`**-s, but better to use GDB

- Write code→don't know why it doesn't work→get debugging help from TA→code works
- This **does not** guarantee that you can write JOS code by yourself

- But if you learn and know how to fix bugs in your code
  - This would be the **most valuable skill that you can learn**

Quiz 1 Statistics

Average Score	High Score	Low Score	Standard Deviation	Average Time
91%	100%	45%	1.51	56:27



Q1: Real-mode Segmentation

1. [Real-mode segmentation] In JOS Lab1, the first instruction that the QEMU emulator runs is from BIOS, and it is stored at [f000:fff0]. The instruction is:

[f000:fff0] 0xffff0: ljmp $0xf000, $0xe05b

This instruction will make the CPU jump on the instruction 0xfe05b, and the next instruction is:

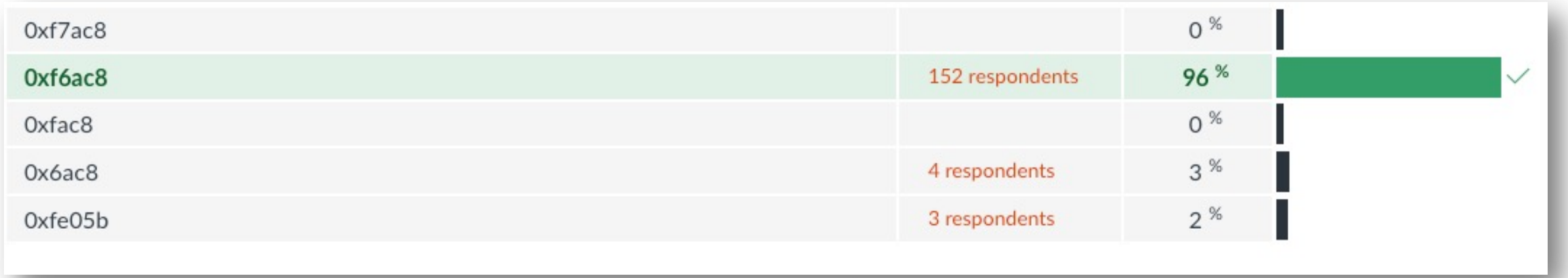
[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x6ac8

Question: Suppose the value of the cs register is 0xf000. Then, what is the address that this instruction read the data from, pointed by %cs:0x6ac8?

(Hint: The address is equivalent to 0xf000:0x6ac8)

Segment * 16 + offset

Q1: Real-mode Segmentation



[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x6ac8

Question: Suppose the value of the cs register is 0xf000. Then, what is the address that this instruction read the data from, pointed by %cs:0x6ac8?

(Hint: The address is equivalent to 0xf000:0x6ac8)

Q2: Real-mode Segmentation & A20

2. [Real-mode segmentation and A20] In the real mode of x86, with A20 disabled, which address does the following segment:offset combination points to?

[f700:f100]

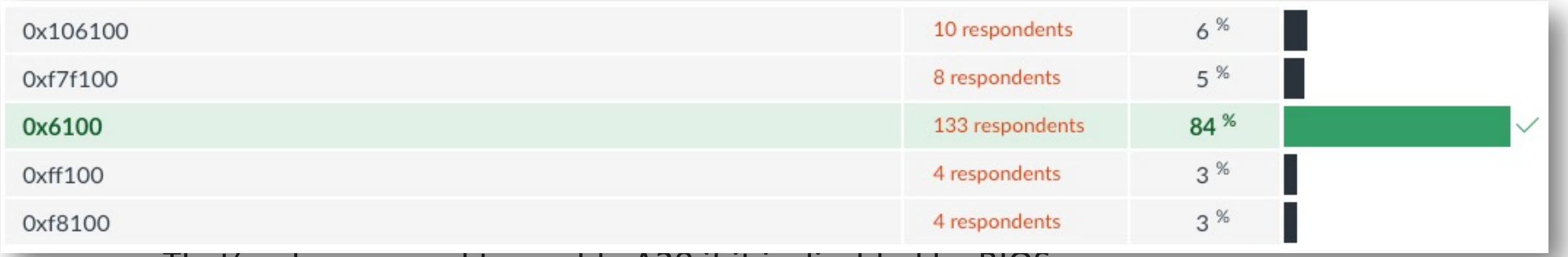
A20 disabled: ignore the address bit **20:** regarding it as 0

- That's why we need to enable A20 if it is disabled by BIOS

- **`f700 * 16 + f100 = f7000 + f100 = 0x106100`**

- Ignore bit at 20: 0x**1**06100
- **`0x6100`**

Q2: Real-mode Segmentation & A20



0x106100	10 respondents	6 %
0xf7f100	8 respondents	5 %
0x6100	133 respondents	84 %
0xff100	4 respondents	3 %
0xf8100	4 respondents	3 %

- That's why we need to enable A20 if it is disabled by BIOS

- **`f700 * 16 + f100 = f7000 + f100 = 0x106100`**

- Ignore bit at 20: 0x**1**06100
- **`0x6100`**

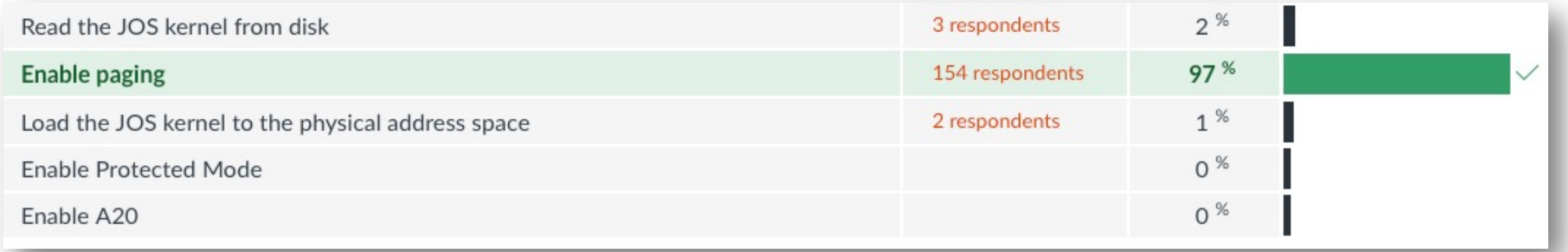
Q3: JOS Bootloader

3. [JOS Bootloader] Which of the following is **NOT** a job that the JOS bootloader does?

Recap - JOS Boot Sequence

- 0xf000:0xfff0 – BIOS
- Loads boot sector – runs 0x7c00
- Enable A20
- Enable protected mode (enabling 4GB memory access)
- Read kernel ELF (Executable Linkable Format)

Q3: JOS Bootloader



Read the JOS kernel from disk	3 respondents	2 %
Enable paging	154 respondents	**97 %** ✓
Load the JOS kernel to the physical address space	2 respondents	1 %
Enable Protected Mode		0 %
Enable A20		0 %

- 0xf000:0xfff0 – BIOS
- Loads boot sector – runs 0x7c00
- Enable A20
- Enable protected mode (enabling 4GB memory access)
- Read kernel ELF (Executable Linkable Format)

Q3: JOS Bootloader

- Where do we enable virtual memory (paging)?
- In **`kern/entry.S`** [JOS Kernel, not the bootloader]

```
# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
movl    %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0

# Now paging is enabled, but we're still running at a low EIP
# (why is this okay?).  Jump up above KERNBASE before entering
# C code.
mov $relocated, %eax
jmp *%eax
```

Q4: JOS Bootloader

4. [JOS Bootloader] The JOS bootloader loads the JOS kernel at the physical address 0x100000. How does JOS decide the address 0x100000 to load the kernel?

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

    // 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);

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

Q4: JOS Bootloader

4. [JOS Bootloader] The JOS bootloader loads the JOS kernel at the physical address 0x100000. How does JOS decide the address 0x100000 to load the kernel?

Lab 1

The boot loader uses the ELF *program headers* to decide how to load the sections. The program headers specify which parts of the ELF object to load into memory and the destination address each should occupy. You can inspect the program headers by typing:

```
$ objdump -x obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x00007108 memsz 0x00007108 flags r-x
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
         filesz 0x0000a300 memsz 0x0000a944 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx
...
```

Q4: JOS Bootloader

4. [JOS Bootloader] The JOS bootloader loads the JOS kernel at the physical address 0x100000. How does JOS decide the address 0x100000 to load the kernel?

All OS kernel must be loaded at 0x100000, similar to that the bootloader is loaded at the fixed physical address 0x7c00.	13 respondents	8 %	
The ELF header of the JOS kernel defines where its code and data should be loaded in memory, and it is 0x100000. The JOS bootloader parses the ELF header to get that address and loads the kernel at 0x100000.	139 respondents	87 %	
After enabling the i386 protected mode, the JOS bootloader can access over 1MB space of physical memory. Because 0x100000 is the start address of that over 1MB address space, the bootloader loads the kernel to that address.	4 respondents	3 %	
The JOS bootloader can load the JOS kernel at any address, and because the bootloader will never use address above 1MB line, so we load that at 0x100000, but in the bootloader we can change it to 0x200000 or an arbitrary address.	3 respondents	2 %	

Q5: x86 Program Stack

5. [JOS Lab1 - x86 program stack] Suppose you have the following code snippet in your JOS. Which will be the correct assignment to print the current base pointer (EBP) and the saved return address (EIP) to the console?

```
int *ebp = (int*) read_ebp();
/* omitting some unnecessary parts */

/* please choose the correct assignment from the listing */
int EBP = ???
int EIP = ???

/* print! */
cprintf("  ebp %08x  eip %08x", EBP, EIP);
```

Q5: x86 Program Stack

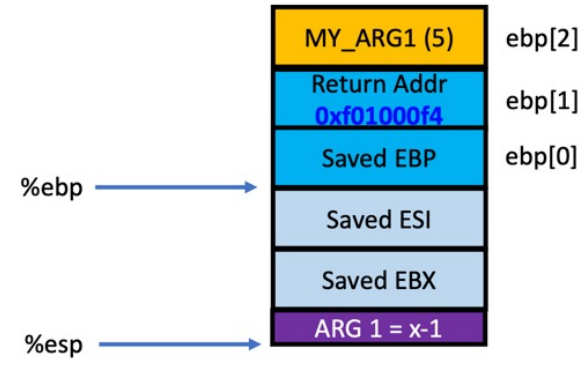
5. [JOS Lab1 - x86 program stack] Suppose you have the following code snippet in your JOS. Which will be the correct assignment to print the current base pointer (EBP) and the saved return address (EIP) to the console?

```
int *ebp = (int*) read_ebp();
/* omitting some unnecessary parts */

/* please choose the correct assignment from
int EBP = ???
int EIP = ???

/* print! */
cprintf("  ebp %08x  eip %08x", EBP, EIP);
```

Hint – Exercise 11

```
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
       kern/monitor.c:143: monitor+106
```

- int *ebp = (int *) read_ebp();
  - cprintf("ebp %08x", ebp)…
- EIP == return address
  - ebp[1] – why?
- Args?
  - Print ebp[2 ~ 6]…



Q5: x86 Program Stack

EBP = ebp; EIP = ebp+1;	5 respondents	3 %	
EBP = ebp[0]; EIP = ebp[1];	2 respondents	1 %	
EBP = ebp; EIP = ebp[1];	148 respondents	93 %	✓
EBP = ebp; EIP = ebp+4;	4 respondents	3 %	

```
int *ebp = (int*) read_ebp();
/* omitting some unnecessary parts */

/* please choose the correct assignment from the listing */
int EBP = ???
int EIP = ???

/* print! */
cprintf("  ebp %08x  eip %08x", EBP, EIP);
```

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Multiply by 2^{12}

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Multiply by 2^{12}

0x1000 * 0x1000 = 0x400000
[actual limit]

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, ~~DPL = 0~~
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Multiply by 2^{12}

0x1000 * 0x1000 = 0x400000
[actual limit]

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

What about privilege level?

0x11111000 + 0x1010 = 0x11112010

Descriptor Privilege Level Defines Ring Level

- CPL = Current Privilege Level
  - Defined in the last 2 bits of the %cs register
  - You can change %cs only via `lcall/ljmp/trap/int`

- Examples
  - %cs == 0x8 == 1000 in binary, last 2 bits are ZERO -> KERNEL!
  - %cs == 0x13 == 10011 in binary, last 2 bits are 3 -> USER!
  - %cs == 0x10 == 10000 in binary, last 2 bits are 0 -> KERNEL!
  - %cs == 0xb == 1011….

TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

GDT index	32-bit Base	20-bit Limit	12-bit Flags
16 USER	0x31310000	0x1000	G=0, DPL=3
8 KERNEL	0x40000000	0x80000	G=1, DPL=0
0 KERNEL	0x0	0xfffff	G=1, DPL=0

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

What about privilege level?

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

What about privilege level?

CPL: last 2 bits of 0x10

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

What about privilege level?

CPL: last 2 bits of 0x10
last 2 bits: **00**

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

What about privilege level?

CPL: last 2 bits of 0x10
last 2 bits: **00 [ring 0]**
ring 0 → can access DPL 0

0x11111000 + 0x1010 = 0x11112010

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

A = invalid access, B = invalid access, C = 0x11111010	6 respondents	4 %
A = 0x11111010, B = invalid access, C = invalid access		0 %
A = 0x11111010, B = invalid access, C = 0x11111010	2 respondents	1 %
A = 0x11111010, B = 0x22222999, C = invalid access	4 respondents	3 %
A = 0x11112010, B = invalid access, C = 0x11111010	1 respondent	1 %
A = 0x11112010, B = 0x22222999, C = 0x11111010	1 respondent	1 %
A = 0x11111010, B = 0x22222999, C = 0x11111010	1 respondent	1 %
A = 0x11112010, B = 0x22222999, C = invalid access	140 respondents	88 %
A = 0x11112010, B = invalid access, C = invalid access	4 respondents	3 %

'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

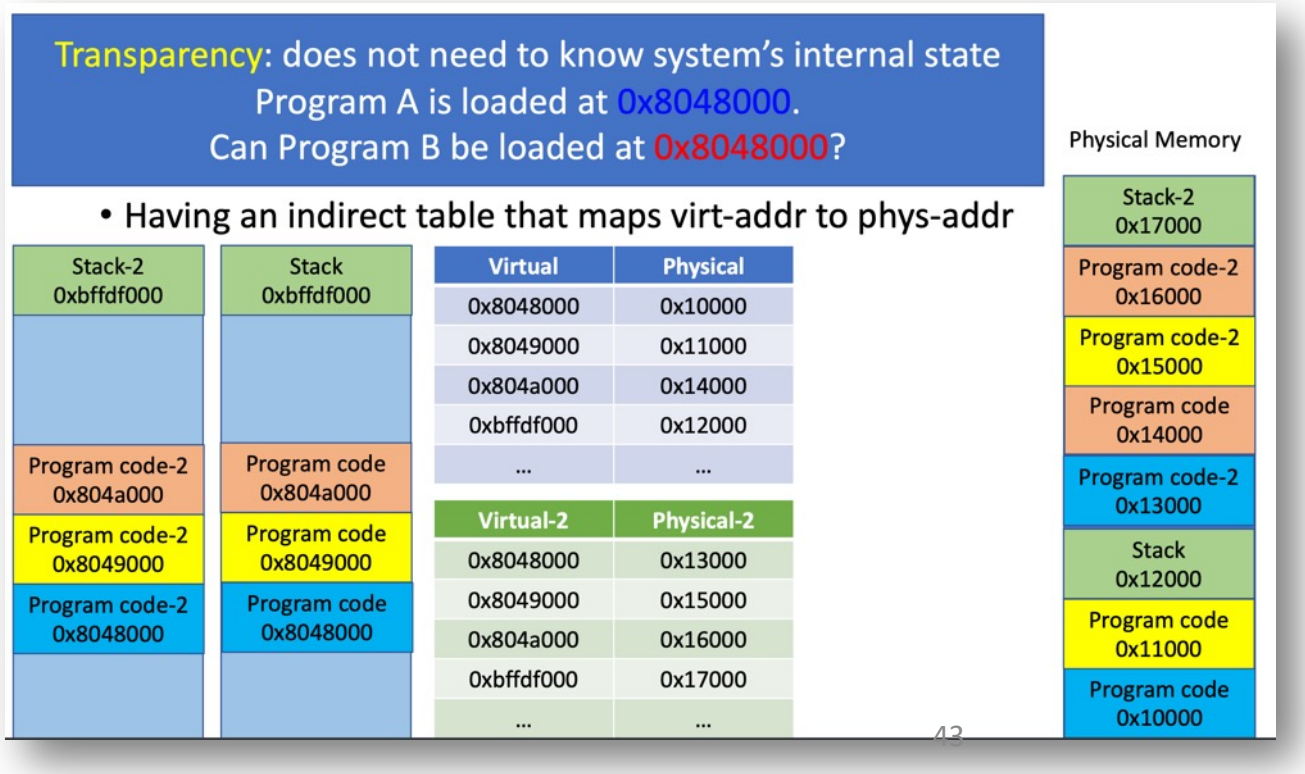
B. [0x8:0x0999]

C. [0x13:0x0010]

Q7: Virtual Memory

7. [Virtual Memory] We have learned three goals of virtual memory, transparency, efficiency, and protection.

Which of the following describes the 'transparency' goal?



Q7: Virtual Memory

7. [Virtual Memory] We have learned three goals of virtual memory, transparency, efficiency, and protection.

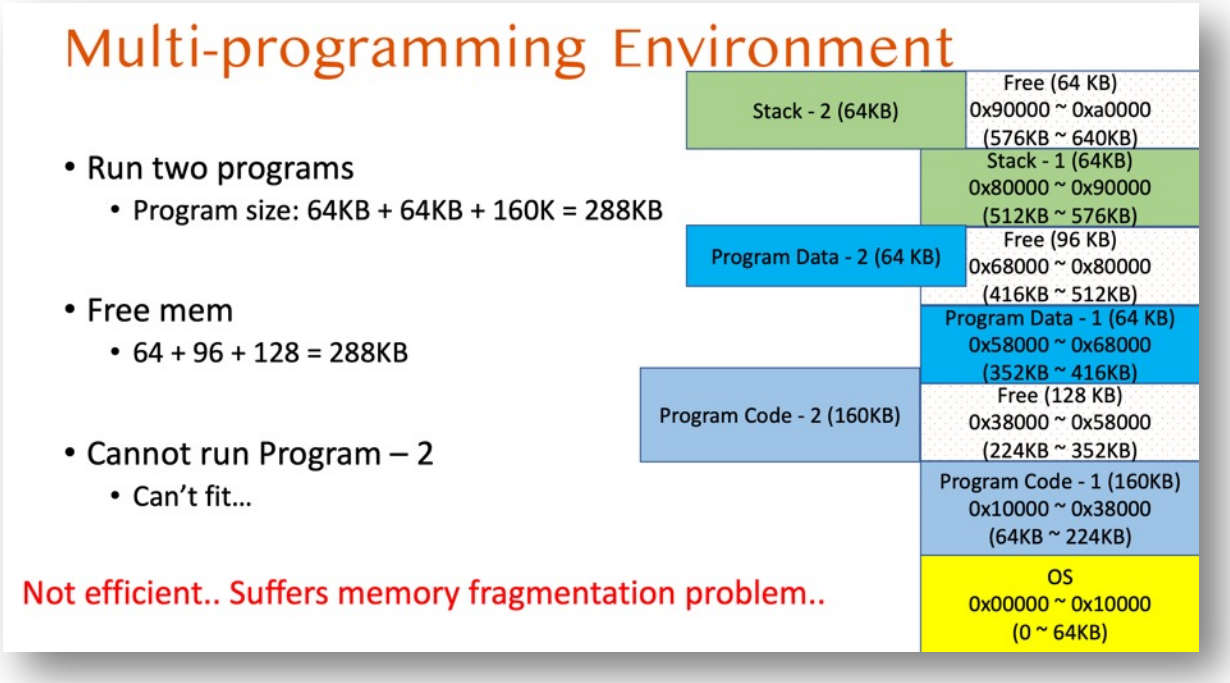
Which of the following describes the 'transparency' goal?

Suppose a program has been loaded to an address 0x8048000. And, you have another program that requires 0x8048000 for its code address. We can't load and run both programs at the same time in the physical memory space, however, we can load both programs at the same virtual address in the virtual memory space.	159 respondents	100 %	
Without virtual memory, a program may access code and data owned by other programs. Virtual memory systems can prevent this by providing an isolated, virtual address space to each program.		0 %	
Physical memory may suffer from memory fragmentation, e.g., cannot utilize the entire free physical memory due to fragmentation. Virtual memory management via paging can resolve such fragmentation issue by breaking down the minimal unit of memory mapping as page, allowing virtually contiguous memory space does not have to be contiguous in physical space.		0 %	

Q8: Virtual Memory

8. [Virtual Memory] We have learned three goals of virtual memory, transparency, efficiency, and protection.

Which of the following describes the 'efficiency' goal?



Q8: Virtual Memory

8. [Virtual Memory] We have learned three goals of virtual memory, transparency, efficiency, and protection.

Which of the following describes the 'efficiency' goal?

Physical memory may suffer from memory fragmentation, e.g., cannot utilize the entire free physical memory due to fragmentation. Virtual memory management via paging can resolve such fragmentation issue by breaking down the minimal unit of memory mapping as page, allowing virtually contiguous memory space does not have to be contiguous in physical space.	158 respondents	99 %	
Suppose a program has been loaded to an address 0x8048000. And, you have another program that requires 0x8048000 for its code address. We can't load and run both programs at the same time in the physical memory space, however, we can load both programs at the same virtual address in the virtual memory space.	1 respondent	1 %	
Without virtual memory, a program may access code and data owned by other programs. Virtual memory systems can prevent this by providing an isolated, virtual address space to each program.		0 %	

Q9: Address Translation

9. [Virtual Memory] We have the following page directory and page table:

Page directory (the presence of each flag means that the flag bit is 1, e.g., PTE_P means Present = 1):

Index	Physical Page Number	Flags
0x0	0x0	PTE_P
...	...	...
0x20	0x444	PTE_P | PTE_W
0x21	0x555	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x400	PTE_P | PTE_U

Page table at 0x444000 (the presence of each flag means that the flag bit is 1, e.g., PTE_P means Present = 1):

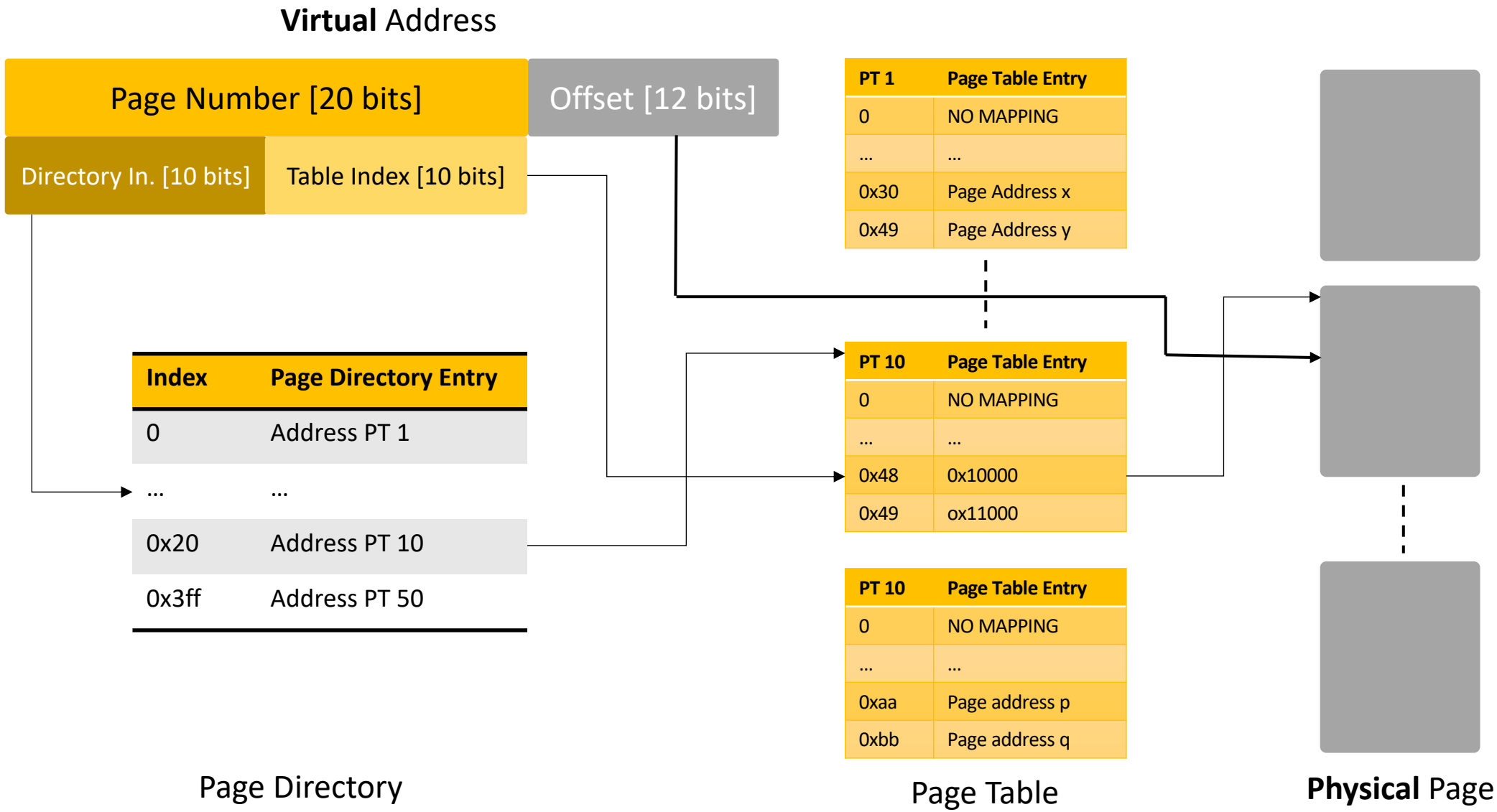
Index	Physical Page Number	Flags
0x0	0x31337	PTE_P
...	...	...
0x20	0x345	PTE_P | PTE_U
0x21	0x678	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x1337	PTE_P | PTE_U

The CR3 register of the CPU points to the address of the page directory specified above.

Question: Which physical address that a virtual address 0x08020567 is translated to?

Hint: Think about what will be the value of PDX(0x8020567) and PTX(0x8020567).

Memory Address Translation



Q9: Address Translation

- 0x**08020**567
- VPN: **0x08020** Offset: 0x567
  - **0000 1000 0000 0010 0000**
- Higher 10-bits (PDX): **0x20**
- Lower 10-bits (PTX): **0x20**

- PPN: 0x**345** Offset: **0x567**
  - 0x**345567**

9. [Virtual Memory] We have the following page directory and page table:

Page directory (the presence of each flag means that the flag bit is 1, e.g., PTE_P means Present = 1):

Index	Physical Page Number	Flags
0x0	0x0	PTE_P
...	...	...
0x20	0x444	PTE_P | PTE_W
0x21	0x555	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x400	PTE_P | PTE_U

Page table at 0x444000 (the presence of each flag means that the flag bit is 1, e.g., PTE_P means Present = 1):

Index	Physical Page Number	Flags
0x0	0x31337	PTE_P
...	...	...
0x20	0x345	PTE_P | PTE_U
0x21	0x678	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x1337	PTE_P | PTE_U

The CR3 register of the CPU points to the address of the page directory specified above.

Question: Which physical address that a virtual address 0x08020567 is translated to?

Hint: Think about what will be the value of PDX(0x8020567) and PTX(0x8020567).

Q9: Address Translation

0x678567	1 respondent	1 %
0x444567	1 respondent	1 %
0x345567	144 respondents	91 %
0x555567	1 respondent	1 %
0x678000	1 respondent	1 %
0x678000		0 %
0x31337567	5 respondents	3 %
0x345000	1 respondent	1 %
0x1337567	5 respondents	3 %

- PPN: 0x**345** Offset: **0x567**
  - **0x345567**

0x0	0x31337	PTE_P
...	...	...
0x20	0x345	PTE_P | PTE_U
0x21	0x678	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x1337	PTE_P | PTE_U

The CR3 register of the CPU points to the address of the page directory specified above.

Question: Which physical address that a virtual address 0x08020567 is translated to?

Hint: Think about what will be the value of PDX(0x8020567) and PTX(0x8020567).

Q10: Address Translation

- 0x**08021**333
- VPN: **0x08021** Offset: 0x333
  - **0000 1000 0000 0010 0001**
- Higher 10-bits (PDX): **0x20**
- Lower 10-bits (PTX): **0x21**

- PPN: **0x678** Offset: **0x333**
  - 0x**678333**

10. [Virtual Memory] We have the following page directory and page table:

Page directory (the presence of each flag means that the flag bit is 1, e.g., PTE_P means Present = 1):

Index	Physical Page Number	Flags
0x0	0x0	PTE_P
...	...	...
0x20	0x111	PTE_P | PTE_W
0x21	0x222	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x400	PTE_P | PTE_U

Page table at 0x111000 (the presence of each flag means that the flag bit is 1, e.g., PTE_P means Present = 1):

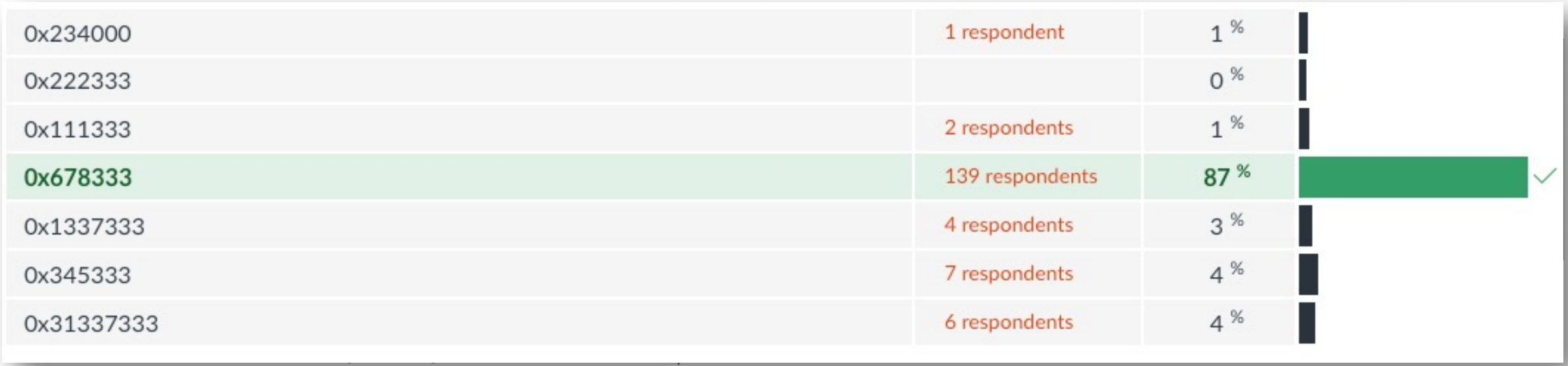
Index	Physical Page Number	Flags
0x0	0x31337	PTE_P
...	...	...
0x20	0x345	PTE_P | PTE_U
0x21	0x678	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x1337	PTE_P | PTE_U

The CR3 register of the CPU points to the address of the page directory specified above.

Question: Which physical address that a virtual address 0x08021333 is translated to?

Q10: Address Translation

10. [Virtual Memory] We have the following page directory and page table:



0x234000	1 respondent	1 %
0x222333		0 %
0x111333	2 respondents	1 %
0x678333	139 respondents	87 % ✓
0x1337333	4 respondents	3 %
0x345333	7 respondents	4 %
0x31337333	6 respondents	4 %

Index	Physical Page Number	Flags
0x0	0x31337	PTE_P
...	...	...
0x20	0x345	PTE_P | PTE_U
0x21	0x678	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x1337	PTE_P | PTE_U

The CR3 register of the CPU points to the address of the page directory specified above.

Question: Which physical address that a virtual address 0x08021333 is translated to?

- PPN: **0x678** Offset: **0x333**
  - 0x**678333**

Q11: Address Translation (Permission Check)

- 0x**08020**212
  - PDX: **0x20**
  - PTX: **0x20**

- PDE → **PTE_P**, **PTE_W**
- PTE → **PTE_P**, **PTE_U**
- Set **intersection** (PDE ∩ PTE) → **PTE_P**

- **Valid (P), not writable, only for ring 0**

11. [Virtual Memory] We have the following page directory and page table:

Page directory (the presence of each flag means that the flag bit is 1, e.g., PTE_P means Present = 1):

Index	Physical Page Number	Flags
0x0	0x0	PTE_P
...	...	...
0x20	0x111	PTE_P | PTE_W
0x21	0x222	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x400	PTE_P | PTE_U

Page table at 0x111000 (the presence of each flag means that the flag bit is 1, e.g., PTE_P means Present = 1):

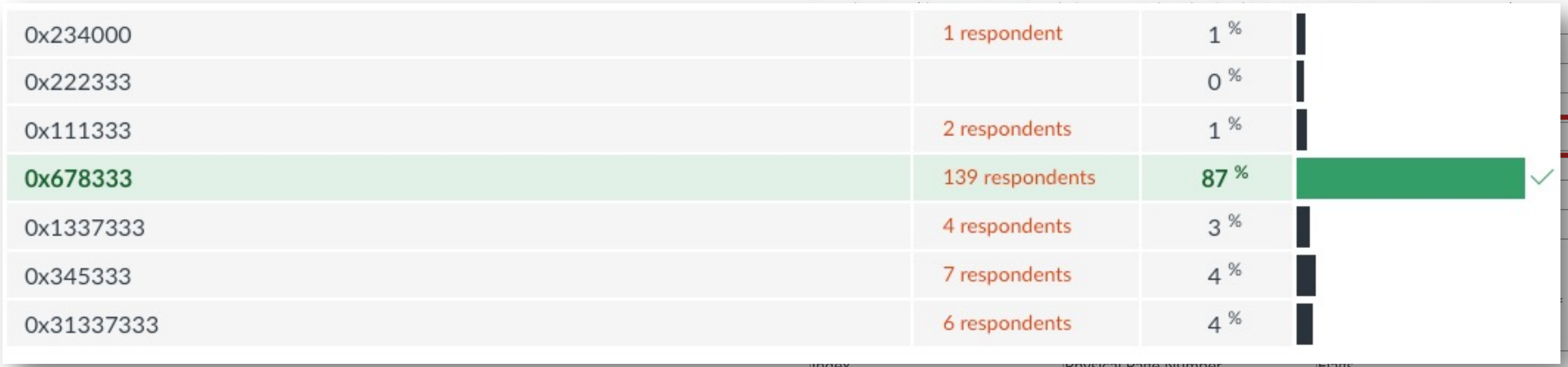
Index	Physical Page Number	Flags
0x0	0x31337	PTE_P
...	...	...
0x20	0x234	PTE_P | PTE_U
0x21	0x567	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x1337	PTE_P | PTE_U

The CR3 register of the CPU points to the address of the page directory specified above.

Question: What is the memory permission applied to the virtual address 0x08020212?

Q11: Address Translation (Permission Check)

11. [Virtual Memory] We have the following page directory and page table:



0x234000	1 respondent	1 %
0x222333		0 %
0x111333	2 respondents	1 %
0x678333	139 respondents	87 %
0x1337333	4 respondents	3 %
0x345333	7 respondents	4 %
0x31337333	6 respondents	4 %

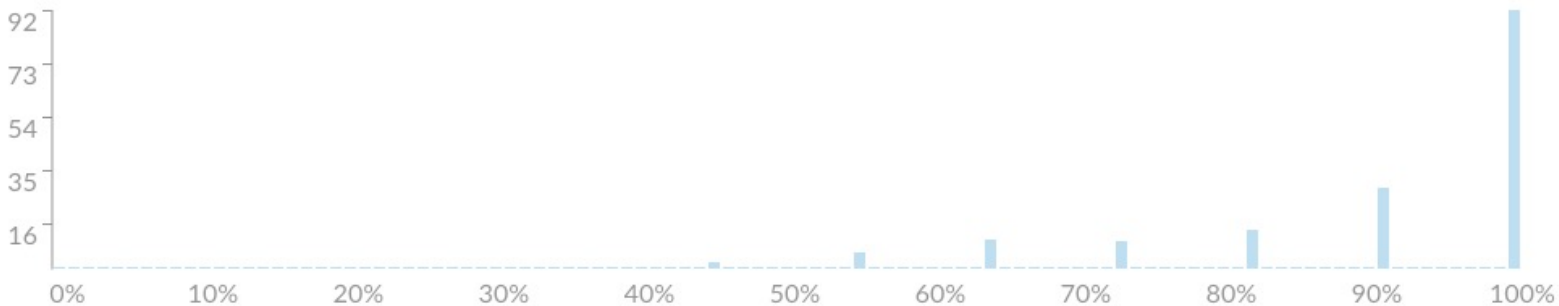
- Set **intersection** (PDE ∩ PTE) → **PTE_P**
- **Valid (P), not writable, only for ring 0**

Index	Physical Page Number	Flags
0x0	0x31337	PTE_P
...	...	...
0x20	0x234	PTE_P | PTE_U
0x21	0x567	PTE_P | PTE_U | PTE_W
...	...	...
0x3ff	0x1337	PTE_P | PTE_U

The CR3 register of the CPU points to the address of the page directory specified above.

Question: What is the memory permission applied to the virtual address 0x08020212?

Average Score	High Score	Low Score	Standard Deviation	Average Time
91%	100%	45%	1.51	56:27



Backup

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

0x22222000 + 0x999 = 0x22222999

CPL? 0x8 = 1000
Last 2 bits = 00
Ring 0, can access DPL 3

Q6: Protected Mode Segmentation

6. [i386 Protected Mode] We have the following Global Descriptor Table (GDT), and it is loaded via lgdt, and the CPU is enabled with the i386 Protected Mode.

Selector	Base	Limit	Flags
0x10	0x11111000	0x1000	G = 1, DPL = 0
0x8	0x22222000	0x1000	G = 0, DPL = 3
0			

Question: In this case, which memory address is accessed by the following [cs segment:offset] ?

(Please choose the correct answer for all A, B, and C, and for the invalid memory access, you must choose 'invalid', not the address, to get points).

Hint: Current Privilege Level (2-bit data) is stored at the last 2 bits of the CS segment register

A. [0x10:0x1010]

B. [0x8:0x0999]

C. [0x13:0x0010]

CPL? 0x13 = 10011
Last 2 bits = 11
Ring 3, cannot access DPL 0

INVALID ACCESS!!