

Quiz 2

Started: May 5 at 9:48pm

Quiz Instructions

CS 444/544 Operating Systems II

Quiz II

Quiz Description

In order to receive the credit, you must answer the question by choosing the answer from the listing. We have an open-material policy on this quiz, so you may refer to slides, textbooks, your JOS source code, your note, or other online materials. However, we strictly prohibit chatting with other students or asking for help online (posting a question, etc.).

In case if you cannot understand any questions in the quiz, please find Yeongjin or any TAs for clarification, but please DO NOT ASK for checking if your answer is correct or not.

Lastly, only two attempts are allowed for this quiz.

Question 1

1 pts



1. [virtual memory mapping in JOS labs] When we create a new page directory for a new environment (in `env_setup_vm()` in `kern/env.c`), we copy page directory entries from the kernel page directory (`kern_pgdir`), from the virtual address `UTOP` to the end of 32-bit virtual address space (2^{32} ; please refer to `inc/memlayout.h` for the detailed virtual memory layout in JOS).

This means that the page directory for the user environment (`e->env_pgdir`) will have valid mappings for the addresses above `UTOP`, which includes the region that is reserved only for the kernel (above `ULIM`, e.g., direct map region above `KERNBASE`, kernel stack, etc.).

However, in JOS, this is managed securely, that means, the user program in an environment can never access memory above `ULIM`.

Regarding this, can you answer how JOS (and your implementation of lab3) manages this securely?

In other words, how can we make user programs be not able to access kernel memory even if mappings for those virtual addresses are valid in the

environment's page directory/table?

- ☐ The statement is wrong. JOS is just an educational OS, so it does not take care of such privilege isolation in its virtual memory management. To enable this, we must run a commercial grade OS, such as Linux, Windows, or MacOS.
- ☐ For kernel-only virtual address space, we set their corresponding page directory or page table entries to have their PTE_U as 0 to remove access from the user privilege level (ring 3). Because our JOS has already did that in lab2, so we are safe to copy entries and re-use them in user environments.
- ☐ CPU knows that any address below ULIM is allowed to the user privilege level (ring 3) by checking only the address value (whether it is below ULIM or not). So copying entries from the kernel page directory will be fine.
- ☐ We should not copy entries from kernel page directory because doing such will automatically allow kernel-memory access from ring 3. To remove access to kernel-only virtual addresses from the user privilege level (ring 3), we should set their corresponding page directory or page table entries to have their PTE_P as 0 to remove access from the user privilege level (ring 3).

Question 2**1 pts**

2. [Calculating memory space overhead in supporting virtual-to-physical translation] A user environment would like to have the following virtual-to-physical mappings.

Area	Starting virtual address (inclusive)	Ending virtual address (exclusive)	Size
.text (code)	0x800000	0x804000	0x4000 (16KB)
.data (initialized data)	0xc00000	0xd00000	0x100000 (1MB)
.bss (uninitialized data)	0x1000000	0x1800000	0x800000 (8MB)

Question: What will be the minimum amount of memory required to support these mappings, assuming that we are using x86, paging, 2-level page table (page directory and then page table), and a 4KB page?

To get the answer, please exclude all kernel and other parts (not shown on the table) of memory allocation, and just counting the memory space used for page

directory and tables for creating mappings in the table.


- ☐ 4 KB (1 page directory)
- ☐ 8 KB (1 page directory, 2 page tables)
- ☐ 20 KB (1 page directory, 4 page tables)
- ☐ 16 KB (1 page directory, 3 page tables)
- ☐ 24 KB (1 page directory, 5 page tables)

Question 3

1 pts

3. [Interrupt/Exceptions/Faults] In JOS lab3, we have a user program, divzero, that generates the divide by zero exception in its execution. Regarding this, we learned that there are two types of exceptions, one is non-recoverable exceptions, and the other is the fault, recoverable exceptions, and the divide by zero exception is not a recoverable exception.

After learning this, Tom VVeller (Disclaimer: not W), a (virtual) smart OSU student in this class, would like to see what will happen in JOS if we treat that exception as recovered. To test this, he changed his JOS implementation to handle the exception by not doing anything at the trap_dispatch and just return back to trap(), letting JOS execute env_run to continue the user execution. Like the following:



```
// dispatch page_fault
switch (tf->tf_trapno) {
    case T_DIVIDE:
    {
        return;
    }
}
```

Question: how Tom's JOS will run with divzero?

- ☐ His JOS will return to the next instruction of the instruction that generated the divide by zero exception. So the user execution has avoided the problematic instruction (that generates the divide by zero exception), and thereby, the user program works correctly.
- ☐ The CPU will block 'iret' in env_run() because it knows that the current trap handling context is for handling a divide by zero exception, which is not a recoverable exception, but his JOS attempted continue the user execution.
- ☐ His JOS will return to the next instruction of the instruction that generated the divide by

zero exception. In this regard, although the user execution has avoided the problematic instruction (that generates the divide by zero exception), the result of the problematic instruction is not reflected, so the user program continues to run but works incorrectly.

- ☐ His JOS will return to the same instruction that generated the divide by zero exception. Because the exception cannot be resolved, it will generate another divide by zero exception, returns, and exception is generated again, so basically it will fall into the infinite loop of handling divide zero exception.

Question 4

1 pts

4. [Trap handling in x86/JOS] The following diagram shows the Trapframe structure in JOS. Which of the following information is stored by CPU, when a trap (interrupt/exception/fault) happens (assuming the trap was generated by a page fault, which is a fault with an error code)?

FYI, all elements in your selection must be stored by CPU automatically to get the point.

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

- ☐ tf_trapno, tf_err, tf_eip, tf_cs, tf_eflags, tf_esp, tf_ss
- ☐ tf_regs, tf_es, tf_ds, tf_trapno
- ☐ The entire TrapFrame structure.
- ☐ All general purpose registers in tf_regs
- ☐ tf_err, tf_eip, tf_cs, tf_eflags, tf_esp, tf_ss

Question 5**1 pts**

5. [Page Fault] Following is the Trapframe structure that we use during handling a page fault. Regarding this, please answer the following questions:

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Question 1) Which element in the Trapframe stores the address of the instruction that generated a page fault?

Question 2) Which register stores "access to which address generated the page fault"?

Question 3) Which element in the Trapframe stores the reason for having the page fault?



☐ 1) tf_eip, 2) cr3, 3) tf_cs

☐ 1) tf_eip, 2) cr2, 3) tf_err

☐ 1) tf_esp, 2) cr2, 3) tf_cs

☐ 1) tf_esp, 2) cr2, 3) tf_eip

☐ 1) tf_eip, 2) cr3, 3) tf_err

☐ 1) tf_eip, 2) cr3, 3) tf_esp

Question 6**1 pts**

6. [Page Fault] Following is the Trapframe that we got during handling a page fault. Regarding this, please answer the following questions:

```
Incoming TRAP frame at 0xf0119f84
TRAP frame at 0xf0119f84
edi  0x00000000
esi  0x00010094
ebp  0xf0119fd8
oesp 0xf0119fa4
ebx  0x00010094
edx  0x00000000
ecx  0x00000000
eax  0x003bc000
es   0x----0010
ds   0x----0010
trap 0x0000000e Page Fault
cr2  0x41414141
err  0x00000002
eip  0xf0103836
cs   0x----0008
flag 0x00000007
```

Question 1) Where does this exception come from? From Ring 3? or Ring 0?

Question 2) Is this exception caused by a write access attempt or a read access attempt?

Question 3) Access to which address caused the access violation?

☐ 1) ring 0, 2) write, 3) 0x41414141

☐ 1) ring 0, 2) read, 3) 0x41414141

☐ 1) ring 3, 2) read, 3) 0xf0103836

☐ 1) ring 3, 2) write, 3) 0xf0103836

☐ 1) ring 3, 2) read, 3) 0x41414141

☐ 1) ring 0, 2) read, 3) 0xf0103836

☐ 1) ring 0, 2) write, 3) 0xf0103836

☐ 1) ring 3, 2) write, 3) 0x41414141

Question 7

1 pts

7. [Preemptive Multitasking] In co-operative multitasking, each user processes are required to yield their execution right to the kernel to support multitasking with a single CPU. This design can enable sharing a CPU for running multiple jobs, however, it could be susceptible to a denial of service (DoS) attack from a user program; that is, if a user program runs an infinite loop, e.g., `while(1);`, then, the kernel will never get a chance to run its code to intervene the user execution.

Regarding this, we have learned the design of preemptive multitasking with interrupt handling. Can you choose which of the following describes how preemptive multitasking works in modern OSes (choose the best answer that describes the concept from the selections)?

- ☐ In contrast to co-operative multitasking, preemptive multitasking is supported by CPU. CPU stops the user execution after running 1 million instructions (or other fixed number of instructions retired) and give the execution right to the kernel. Then, the kernel can decide which other application to schedule, run it, and after running 1 million instructions, do the same thing over and over, and this is how multitasking is supported in the modern OSes.
- ☐ In contrast to co-operative multitasking, preemptive multitasking do not rely on voluntary context-switch. Instead, the OS utilizes a clock hardware that generates a timer interrupt in every 1ms (or some fixed time quantum). Upon the occurrence of the timer interrupt, the CPU stops the user execution and runs the kernel, and then the kernel can decide which application to schedule next and on. Because the interrupt comes asynchronously, this scheme can still switch the execution context from user to kernel to other use program even with one program runs `while(1)`.
- ☐ In contrast to co-operative multitasking, preemptive multitasking requires user programs to run the `yield()` system call (or similar thing) that voluntarily surrenders their execution right in every 1ms (or some fixed time quantum). To avoid having an infinite loop, the OS kernel checks the program if they follows this scheme and blocks their execution at the load time if the loop exists. So there will be no infinite loop like `while(1)` in the user program (it is prohibited by the kernel), and all user program will voluntarily yield their execution.

Question 8

1 pts

8. [User-Kernel Context Switch] In JOS, invoking a system call via calling a library function call (e.g., `cprintf`) will take the following execution path. Regarding the execution, can you answer that at which ring level the CPU runs the following execution?

1) calls `cprintf()`

[Select]

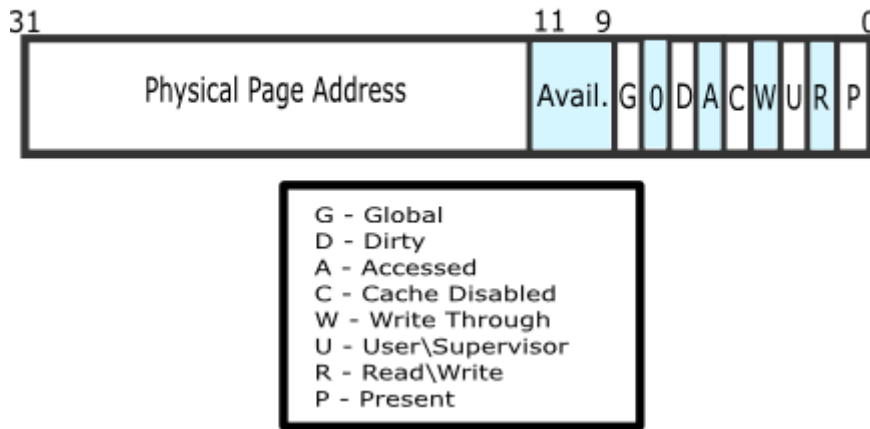


- 2) `cprintf()` calls `sys_cputs()`
- 3) `sys_cputs()` calls `syscall()`
- 4) `syscall()` runs `int $0x30`
- 5) CPU gets a software interrupt and runs `TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)`
- 6) `t_syscall` runs `_alltraps`
- 7) `_alltraps` run `trap()`
- 8) `trap()` runs `trap_dispatch()`
- 9) `trap_dispatch()` calls `syscall()`
- 10) `syscall()` calls `sys_cputs()`
- 11) functions returns until `trap()`, and it calls `env_run()`
- 12) `env_run()` calls `env_pop_tf`, which runs `iret`
- 13) the execution after calling `cprintf` continues

Question 9**1 pts****9. [Handling Copy-on-Write via Page Fault]**

Copy-on-Write is a mechanism to reduce memory usage in modern operating systems.

Page Table Entry



How it works is that, when the operating system has to allocate new memory pages that have the same content in other physical pages (i.e., all 4096 bytes in one page and the other are identical), instead of allocating a new physical page and copy the memory content, the kernel makes the page read-only and maps the same physical page to the newly allocated virtual address. In doing this, modern OSes use a flag bit in the reserved area (bit 10 and 11, marked as Avail. in the figure) as PTE_COW, and set it as 1 (to mark that the page is shared via copy on write) and set PTE_W as 0 (to set the page read-only) for both original and shared pages.

Regarding this, can you answer that which of the following describes how modern OSes support copy-on-write via handling page fault?

- ☐ On a write access to the shared pages via copy-on-write, the CPU will generate a page fault because the page is read-only. So the OS regards a page fault caused by a write access violation (by checking the error code in the Trapframe) as a potential copy-on-write fault (because this includes true write access violations). To prune out true write access violations from the fault, the OS read the page table entry of the faulting address (coming from CR2) and then checks for the flag PTE_COW. If it is 1, then the OS allocates a new physical page, copy the memory content from the shared physical page to the new physical page, and map the new physical page to the virtual address; In doing this, it also removes PTE_COW because now this page is no longer shared. After that, the OS returns to the user execution, and because the OS has resolved the fault at write access, the program can continue to run without having a fault.
- ☐ Please read the other answer and understand how it works!

Quiz saved at 9:48pm

Submit Quiz