> K22 Fall'26 – V. Atlidakis / Quiz-01

Name and sdi _____

1) [12 points: +2 points on each correct answer, -2 points on each incorrect answer]
Choose synchronous or asynchronous.

- System call: synchronous / asynchronous
- Exception of type "abort": synchronous /asynchronous
- Exception of type "fault": synchronous / asynchronous
- Exception of type "trap": synchronous / asynchronous
- Procedure call: synchronous / asynchronous
- Timer interrupt: synchronous / asynchronous

2) [18 points: +2 points on each correct answer, -2 points on each incorrect answer]
Choose Yes or No.

- The bootloader is in ROM: Yes / No
- The stack is a static segment: Yes / No
- POSIX was invented for portability: Yes / No
- A timer helps implement fault isolation: Yes / No
- A process is used to execute a program: Yes / No
- A mode register helps implement preemption: Yes / No
- A user space application can directly call a syscall: Yes / No
- On a TLB miss, a main memory access must follow: Yes / No
- On x86, a double-fault is an exception of the type "fault": Yes / No

3) [10 points] The source code shown below is compiled into "main" and executed, and the identifier of the process executing "main" is 1000. What problem do you see?

$> cat main.c

```
#include <stdio.h>

const char *message = "Hello, World!\n";

void main(void) {
  printf("%p: %s\n", &message, message);
}
```

```
$> gcc -o main main.c
$> ./main
0xaaaae2df0200: Hello, World!

$> cat /proc/1000/maps
aaaae2de0000-aaaae2de1000   r-xp   ...  /home/parallels/main
aaaae2df0000-aaaae2df1000   r--p   ...  /home/parallels/main
aaaae2df2000-aaaae2df3000   rw-p   ...  /home/parallels/main
ffffff6426000-ffffff6447000 rw-p   ...  [stack]
                                   ...
```

Explain here: The pointer "message" seems to belong to a read-only mapping because its memory address (i.e., &message) is shown to be 0xaaaae2df0200. But message is a writable global pointer (const char * makes the characters constant, not the pointer), so its address should be in a writable mapping and not in a read-only mapping because the program is allowed to reassign message (e.g., message = "Goodbye";) and such an assignment must be able to write a new value at &message, which a read-only mapping would prevent.

4) [20 points] The source code shown below is compiled with -O0 into "main" and executed. For each variable, from 1 to 5, choose one region from: (a) .text, (b) .data, (c) .bss, (d) heap, and (e) stack.

$> cat main.c

```
int g;                  /* 1 */
void foo(void) {
   int i = 1;           /* 2 */
   static int s;        /* 3 */
}
void bar(void) {
   static int s = 1;    /* 4 */
}
int main(int argc, char **argv) {
                        /* 5: argv[0] */
   return 0;
}
```

Indicate your choice with a circle:

1) g: a / b / **c** / d / e

2) i: a / b / c / d / **e**

3) foo s: a / b / **c** / d / e

4) bar s: a / **b** / c / d / e

5) argv[0]: a / b / c / d / **e**

5) [10 points] Inspect the source code below and explain in a brief paragraph how you can use it to decide whether your system has its MMU enabled or not.

$> cat main.c

```
int main(int argc, char **argv) {
   int x = atoi(argv[1]);
   printf("%p | %d \n", (void *)&x, x);
}
```

Explain here:

Assume ASLR is off (if you don't know what this is, ignore it for now; if you know what it is, you should know it is easy to turn it off by editing/proc/sys/kernel/randomize_va_space.). Open two terminals and run two copies of main – at the same time – with different command line arguments. If both show the same address for x, but hold different values, then this proves that there are two separate virtual address spaces, which indicates the MMU is enabled.

6) [20 points] There is a processor named di-reversed with a pipeline with the following five stages:

"**IF**: Instruction Fetch" → "**ID**: Instruction decode" → "**EX**: Execute" → "**WB**: Register Write Back" → "**MEM**: Mem. Access".

Observe that the last two stages in the above pipeline are reversed compared to what we discussed in class!

The processor di-reversed supports an instruction called K22PUSH, with syntax K22PUSH <%reg> <const>, which first adds the value const to a 32-bit register %reg, and then saves the result onto the stack.

The semantics of K22PUSH in C code would look like:

- esp -= 4;
- reg += const;
- mem[esp] = reg;

Reversing the last two stages of the pipeline compared to what we have discussed in class was never a good idea.

a) [10 points] Specifically for supporting virtual memory, why is the instruction K22PUSH problematic?

A memory translation (from virtual to physical) for mem[esp] is required. Therefore, the potential for an address translation fault exists. On an address translation fault (remember: fault ⇒ restart at the faulting instruction), the OS will intervene, save the current execution state (i.e., the updated decremented register %esp and the updated incremented register %reg), invoke the appropriate exception handler to handle the fault, and finally, the "faulting" instruction will be restarted. Unfortunately, the state stored by the exception handler has the register %esp already decremented by 4 and the register %reg incremented by 4, and the restart will decrement the register %esp by 4 again and increment the register % reg by 4 again. Thus, the instruction K22PUSH may point to an unintended memory address, depending on whether an address translation fault will be raised, or not.

Assume di-reversed also supports an instruction K22HELPPUSH, with syntax K22HELPPUSH <%reg>, which just "touches" (i.e., accesses) the memory at the address pointed to by <%reg>.

The semantics of K22HELPPUSH in C code would look like:

- (void) mem[reg];

b) [5 points] Explain how you could use the instruction K22HELPPUSH to help with the problem identified in (a)?

Add the instruction K22HELPPUSH <%esp - 4> before any invocation of K22PUSH to prevent the possibility of an address translation fault on the assignment mem[esp] = reg. The pipeline of di-reversed is indeed generally problematic. However, specifically with respect to virtual memory, anyone who sees how K22HELPPUSH can be used to prevent an address translation fault gets full credit.

c) [5 points] Given your modification suggested in (b) to use the command K22HELPPUSH, would you prefer your system to have preemption on or off? And why?

Off. Any solution presented in (b) can be undone with preemption on. If the process executing the program containing the pair of the two instructions gets preempted out at the end of the first instruction (K22HELPPUSH), when the process gets executed back again, it is unclear whether it will get an address translation fault or not.

7) [10 points] Consider the following attempt at a timer interrupt handler, perhaps written by a K22 student using a dangerous mix of inline assembly and C language, as opposed to stand-alone assembly language:

```
static void timer_handler() {
   static unsigned int ticks_since_boot = 0;
   asm("pusha");
   ++ticks_since_boot;
   acknowledge_timer_intr();
   asm("popa");
   asm("iret");
}
```

Assume a 32-bit x86 machine with the stack growing downwards and ignore counter overflows.

- **asm("xxx")**: Asks the compiler to insert the assembly instruction(s) "xxx" at that point in the generated code.
- **pusha:** Saves all 32-bit general-purpose registers onto the stack, and **popa** restores the registers saved by pusha.
- **iret:** Returns from the interrupt by restoring the interrupted execution state.
- **acknowledge_timer_intr():** Sends an acknowledge to the interrupt controller that the current timer interrupt has been handled. (Assume this function is correctly implemented and safe to call from an interrupt context.)

What is the problem with the above implementation? (Ignore the potential for overflow on ticks_since_boot.)

Explain here:

Assume 32-bit x86. When a timer interrupt occurs, the CPU saves EFLAGS, CS, and EIP on the stack. (The hardware does that transparently.) Therefore, to properly return from the interrupt handler, the iret instruction must find those values at the top of the stack to resume the interrupted code. However, the interrupt handler shown above is a normal C function, meaning that the compiler may push a frame pointer, reserve stack space, or change registers before pusha. That is, popa only reverses pusha; but it does not undo the compiler's stack changes, and the inline iret also skips the compiler's function epilogue, so it may pop the wrong values as the return state.