W COMPUTER SCIENCE & ENGINEERING

UNIVERSITY *of* WASHINGTON

CSE 451 Autumn 2015
Final

You have 110 minutes to answer the questions in this quiz. In order to receive credit you must answer the question as precisely as possible.

Some questions are harder than others, and some questions earn more points than others. You may want to skim them all through first, and attack them in the order that allows you to make the most progress.

If you find a question ambiguous, be sure to write down any assumptions you make. Be neat and legible. If we can't understand your answer, we can't give you credit!

Write your name and email address on this cover sheet.

This is an open book, open notes, open laptop exam.

NO INTERNET ACCESS OR OTHER COMMUNICATION.

Name:

Email:

Question:	I	II	III	IV	V	VI	Total
Points:	15	15	25	20	15	10	100
Score:							

I. Warm-up

   (a) (10 points) Circle true or false for each statement (no need to justify your answers here).

True   **False**   When an x86 Bootstrap Processor (BSP) powers on, the BSP is in protected mode with paging enabled.

True   **False**   When an x86 BSP (in protected mode with paging enabled) wakes up an Application Processor (AP), the AP starts code execution in protected mode with paging enabled.

True   **False**   Because the I/O Advanced Programmable Interrupt Controller (IOAPIC) can route interrupts to the BSP but not to the APs, JOS sets up the IOAPIC to route IRQ_KBD and IRQ_SERIAL to the BSP.

True   **False**   In JOS, the kernel will not trigger a page fault when it writes to an arbitrary memory address, because the kernel uses privilege level 0 (ring 0).

True   **False**   Device drivers need to run in privilege level 0 (ring 0) to communicate with the hardware.

(b) (5 points) Ben Bitdiddle is using GDB to single-step over kern/entry.S in JOS:

```
...
# Load the physical address of entry_pgdir into cr3.
# entry_pgdir is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %ecx
movl    %ecx, %cr3
# Enable PSE for 4MB pages.
movl    %cr4, %ecx
orl     $(CR4_PSE), %ecx
movl    %ecx, %cr4
# Turn on paging.
movl    %cr0, %ecx
orl     $(CR0_PE|CR0_PG|CR0_WP), %ecx
movl    %ecx, %cr0
...
```

He notices that before executing the instruction movl %ecx, %cr0, the contents at memory addresses 0x00100000 and at 0xf0100000 are 0x1badb002 and 0x00000000, respectively:

```
(gdb) x/i $pc
=> 0x0010002f:  mov     %ecx,%cr0
(gdb) x/w 0x00100000
0x00100000:     0x1badb002
(gdb) x/w 0xf0100000
0xf0100000:     0x00000000
```

After a single step, both memory addresses contain the value 0x1badb002:

```
(gdb) si
...
(gdb) x/w 0x00100000
0x00100000:     0x1badb002
(gdb) x/w 0xf0100000
0xf0100000:     0x1badb002
```

Ben thinks that the instruction movl %ecx, %cr0 copies the data from memory address 0xf0100000 to overwrite 0x00100000. Do you agree with him? Explain why or why not.

II. Physical world or virtual reality

(a) (10 points) Check either "physical address" or "virtual address" for underlined values from JOS (no need to justify your answers here).

A 256KB memory range starting from 0xfffc0000 in an E820 memory map:
☐ physical address   ☐ virtual address

The kernel pointer kern_pgdir pointing to the kernel's page directory:
☐ physical address   ☐ virtual address

The user pointer pages pointing to read-only copies of the Page structures:
☐ physical address   ☐ virtual address

The value of the %cr3 control register when paging is enabled:
☐ physical address   ☐ virtual address

The value of pcif->reg_base[5], the SATA/AHCI's PCI base address register:
☐ physical address   ☐ virtual address

(b) (5 points) Alyssa P. Hacker is writing a user-space program in JOS:

```
sys_page_alloc(0, (void *)0x1000000, PTE_P|PTE_W|PTE_U);
sys_page_alloc(0, (void *)0x2000000, PTE_P|PTE_W|PTE_U);
strcpy((void *)0x1000000, "hello");
strcpy((void *)0x2000000, "world");
cprintf("%s %s\n", 0x1000000, 0x2000000);
// TODO: your code here
cprintf("%s %s\n", 0x1000000, 0x2000000);
```

The expected output of the program is:

```
hello world
world hello
```

Help Alyssa complete the program by adding some `sys_page_map` call(s) at "TODO". It's okay to ignore the return values of the system call for this question.

The related system calls are:

- **int** sys_page_alloc(**envid_t** envid, **void** *va, **int** perm);

  Allocate a page of memory and map it at va with permission perm in the address space of envid. The page's contents are set to 0. If a page is already mapped at va, that page is unmapped first.

- **int** sys_page_map(**envid_t** srcenvid, **void** *srcva,
                    envid_t dstenvid, **void** *dstva, **int** perm);

  Map the page of memory at srcva in srcenvid's address space at dstva in dstenvid's address space with permission perm; perm has the same restrictions as in sys_page_alloc, except that it also must not grant write access to a read-only page.

III. Truth or dare

Ben is proposing and implementing a few changes to JOS. Please decide whether they are correct or not, and briefly explain why.

(a) (5 points) Ben first writes a user-space program that attempts to access kernel memory above ULIM. Describe what mechanism (in particular, which part of the CPU and which part of JOS) prevents the user space from doing so.

(b) (5 points) Ben is attempting to reduce kernel memory consumption. While reading multicore-JOS he realizes coarse-grain locking ensures only a single CPU can be in the kernel at a time. He now believes that the per-cpu kernel stacks we allocate are an unnecessary pre-caution and based on this analysis decides to remove all but one.

(c) (5 points) Now that Ben has solved his memory consumption problem, he is ready to work on his Lab X Project: a parallel processing framework. He begins by using the copy-on-write fork we implemented for Lab 4. He decides to define a global buffer `static char buf[512]`, which both the parent and child will read and write from to communicate after the fork.

(d) (10 points) Ben enjoys the Lab X demos and wants to port his JOS from x86 to the 64-bit ARM architecture (AArch64). In particular, he is interested in the two-level paging plan. In some way, it looks similar to the two-level paging on x86, with a few differences. For example, the architecture is 64-bit (rather than 32-bit); it uses two registers TTBR0 and TTBR1 to hold base addresses of page tables (rather than one); the page table size in this case is 64 KB (rather than 4 KB). Below is an excerpt from ARM's manual.
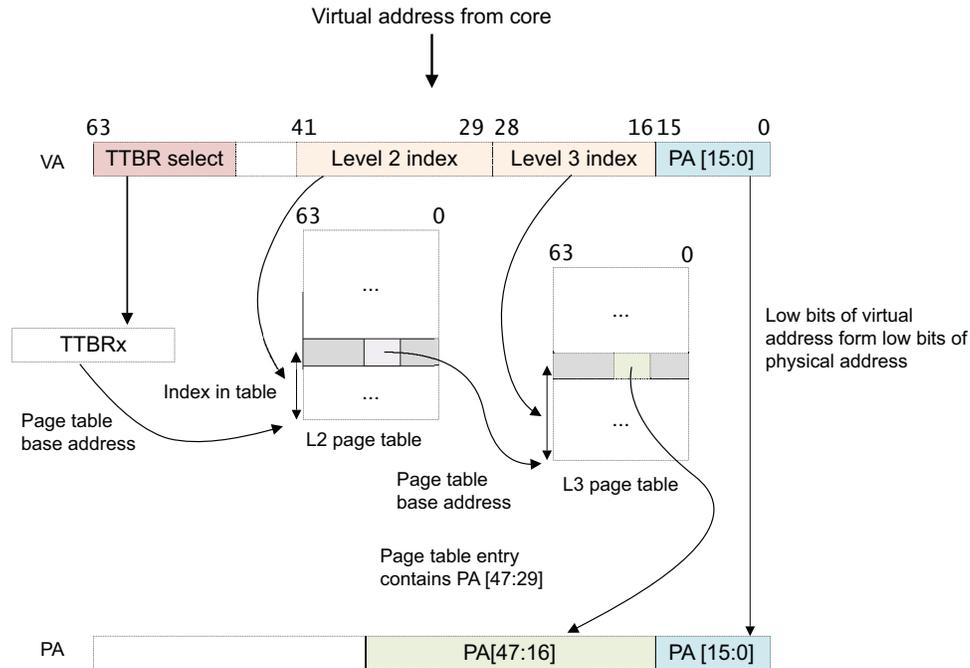
1. If VA[63:42] = 1 then TTBR1 is used for the base address for the first page table. When VA[63:42] = 0, TTBR0 is used for the base address for the first page table.

2. The page table contains 8192 64-bit page table entries, and is indexed via VA[41:29]. The MMU reads the pertinent level 2 page table entry from the table.

3. The MMU checks the level 2 page table entry for validity and whether or not the requested memory access is allowed. Assuming it is valid, the memory access is allowed.

4. In the figure, the level 2 page table entry refers to the address of the level 3 page table (it is a table descriptor).

5. Bits [47:16] are taken from the level 2 page table entry and form the base address of the level 3 page table.

6. Bits [28:16] of the VA are used to index the level 3 page table entry. The MMU reads the pertinent level 3 page table entry from the table.

7. The MMU checks the level 3 page table entry for validity and whether or not the requested memory access is allowed. Assuming it is valid, the memory access is allowed.

8. In the figure, the level 3 page table entry refers to a 64KB page (it is a page descriptor). Bits [47:16] are taken from the level 3 page table entry and used to form PA[47:16]. Because we have a 64KB page, VA[15:0] is taken to form PA[15:0].

9. The full PA[47:0] is returned, along with additional information from the page table entries.

Recall some of the macros you used in your JOS labs on x86:

```
// page directory entries per page directory
#define NPDENTRIES    1024

// page table entries per page table
#define NPTENTRIES    1024

// bytes mapped by a page
#define PGSIZE        4096

// log2(PGSIZE)
#define PGSHIFT       12

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((physaddr_t) (pte) & ~0xFFF)
```

Now help Ben modify these macros for AArch64 (two-level paging, 64 KB page size, assuming `physaddr_t` and `pte` are 64-bit).

IV. Concurrency or consistency

(a) Alyssa is reading Intel's manual on x86's memory consistency model. The manual provides a set of code examples, using the following notational conventions:

- Arguments beginning with an "r", such as r1 or r2, refer to registers (e.g., EAX) visible only to the processor being considered.
- Memory locations are denoted with x, y, z.
- Stores are written as mov [_x], val, which implies that val is being stored into the memory location x.
- Loads are written as mov r, [_x], which implies that the contents of the memory location x are being loaded into the register r.

Your job is to help her understand some of the code examples, by describing an execution of instructions that produces a given result. For example, ④ → ③ → ② → ① is an execution where the instruction ④ running first, followed by ③, ②, and ① in the end.

i. (5 points) Neither loads nor stores are reordered with the same kind of operations.

Processor 0	Processor 1
① mov [_x], 1	③ mov r1, [_y]
② mov [_y], 1	④ mov r2, [_x]

Initially x = y = 0
r1 = 1 and r2 = 0 is not allowed

Alyssa observes r1 = 1 in the end. Describe an execution that produces the result.

ii. (5 points) A store may not be reordered with an earlier locked instruction.

Processor 0	Processor 1
① xchg [_x], r1	③ mov r2, [_y]
② mov [_y], 1	④ mov r3, [_x]

Initially x = y = 0, r1 = 1
r2 = 1 and r3 = 0 is not allowed

Alyssa observes r2 = 1 in the end. Describe an execution that produces the result.

(b) Recall the spinlock (slightly simplified) we used in JOS:

```
struct spinlock {
        uint32_t locked;
};

void spin_lock(struct spinlock *lk)
{
        while (xchg(&lk->locked, 1) != 0) { }
}

void spin_unlock(struct spinlock *lk)
{
        xchg(&lk->locked, 0);
}
```

The function xchg, a wrapper over the corresponding xchg instruction, atomically reads the old value at a given memory address and writes a new value to that address.

```
uint32_t xchg(volatile uint32_t *addr, uint32_t newval)
{
        uint32_t result;
        asm volatile("lock; xchgl %0, %1"
                        : "+m" (*addr), "=a" (result)
                        : "1" (newval)
                        : "cc");
        return result;
}
```

Alyssa is considering some optimizations to the spinlock code, and she needs your help. For each case,

- If you think the optimization is correct, briefly explain why.
- Otherwise, describe an execution that leads to incorrect results; be specific about the number of CPUs/threads and the interleaving.

You may refer to the code examples from part (a).

i. (5 points) Replace xchg with simple memory load and store:

```
void spin_lock(struct spinlock *lk)
{
        while (lk->locked) { }
        lk->locked = 1;
}

void spin_unlock(struct spinlock *lk)
{
        lk->locked = 0;
}
```

ii. (5 points) Keep xchg in spin_lock and use memory store in spin_unlock:

```
void spin_lock(struct spinlock *lk)
{
        while (xchg(&lk->locked, 1) != 0) { }
}

void spin_unlock(struct spinlock *lk)
{
        lk->locked = 0;
}
```

V. The art of persistence

(a) In xv6, each inode has 12 direct blocks and one single indirect block. Since xv6 uses single-sector blocks, each block has 512 bytes. Given that each block number takes four bytes, one single indirect block can hold up to $512/4 = 128$ block numbers. Therefore, each file can grow up to $12 + 128 = 140$ blocks (or $140 \times 512 = 71,680$ bytes).

   i. (2 points) When doing the "big file" exercise, Ben changes xv6 files to have 11 direct blocks, one single indirect block, and one double indirect block. What's the maximum file size (# of blocks)? Write down an expression for computing it (no need to calculate the final value).

   ii. (3 points) Alyssa changes xv6 files to have 10 direct blocks, one single indirect block, one double indirect block, and one triple indirect block. What's the maximum file size this time (# of blocks)? Write down an expression for computing it (no need to calculate the final value).

(b) Recall that the basic disk operations are:

- `read(blkno)`: return the data in block `blkno`.
- `write(blkno, data)`: store `data` in block `blkno`; may be cached/reordered by the disk.
- `flush()`: force data out of the disk cache to physical medium.

We assume `write` is atomic: either the data of an entire block is written on disk, or none.

After reading the Arrakis paper, Alyssa decides to write a simple on-disk data structure, a vector, using these disk operations. Her plan is to store the vector size n (i.e., the number of elements) in disk block 0, and to store the i-th element ($0 \leq i < n$) in block $i + 1$.

One function Alyssa tries to implement is appending an element to the on-disk vector. The pseudo code is the following:

```
append(data):
  n = read(0)           // read the current vector size n
  assert n < N - 1      // assume the disk has N blocks in total
  write(0, n + 1)       // update the vector size to (n + 1)
  write(n + 1, data)    // store the new data in the tail
```

The pseudo code for reading the data of the i-th element is the following:

```
elem(i):
  assert i < read(0)
  return read(i + 1)
```

Initially, the vector is empty: block 0 (i.e., b_0) contains "0"; other blocks (i.e., $b_1, \ldots, b_{N-1}$) contain garbage data (denoted as $\varnothing$).

b_0	b_1	b_2	b_3	$\cdots$	b_{N-1}
0	$\varnothing$	$\varnothing$	$\varnothing$	$\cdots$	$\varnothing$

Alyssa invokes `append("CSE")`, followed by a `flush`. The on-disk vector becomes:

b_0	b_1	b_2	b_3	$\cdots$	b_{N-1}
1	CSE	$\varnothing$	$\varnothing$	$\cdots$	$\varnothing$

i. (5 points) Alyssa continues to invoke append("451"). The power suddenly goes off. List all the possible states of the on-disk vector.

ii. (5 points) The on-disk vector is considered well-formed if:
- the vector size n in block 0 is within the range $[0, N-1]$; and
- elem(i) never returns garbage data, given $0 \leq i < n$.

Based on the pseudo code of append and elem, is the on-disk vector always well-formed, even in the presence of power outages? If yes, briefly explain why. If not, modify append and/or elem to ensure that.

VI. CSE 451

   We would like to hear your opinions. Any answer, except no answer, will receive full credit.

   (a) (2 points) Describe the most memorable bug you have made in the JOS labs.

   (b) (2 points) Are there any topics you would like to see added to or removed from the class?

   (c) (2 points) What is the best aspect of CSE 451?

   (d) (2 points) What is the worst aspect of CSE 451?

   (e) (2 points) Circle true or false for the statement.
       True   False   Princess Leia was held captive in Super Block AA-23.

End of Quiz — Enjoy the break!