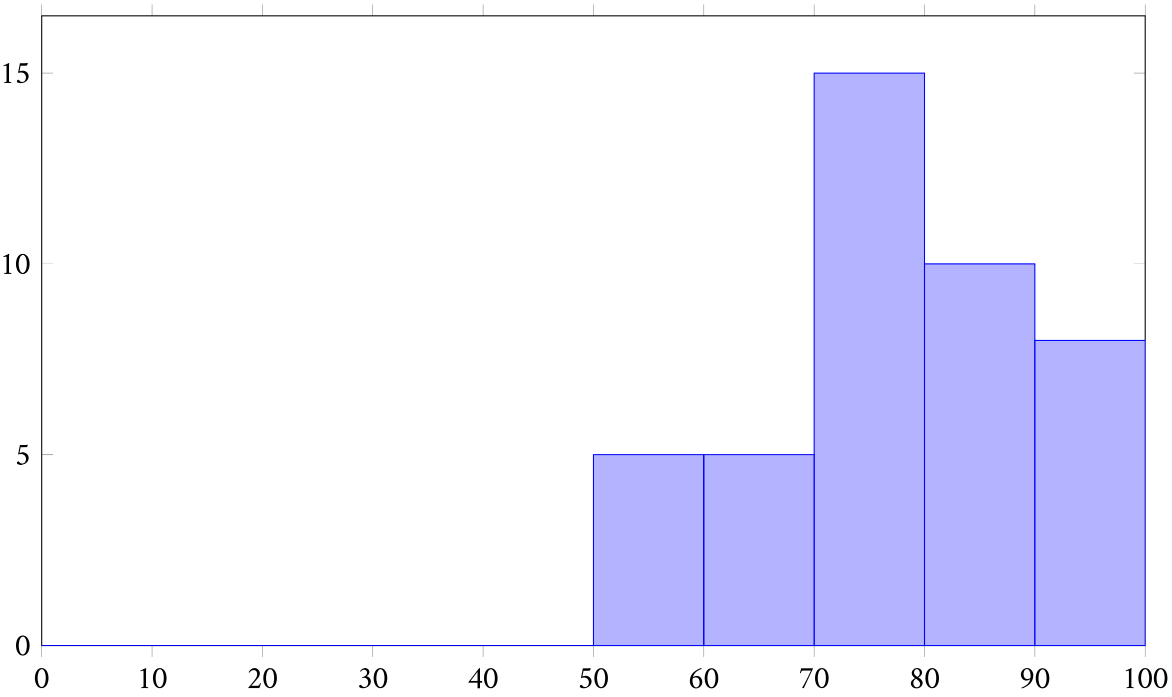
COMPUTER SCIENCE & ENGINEERING

UNIVERSITY *of* WASHINGTON

CSE 451 Autumn 2016

Final Solutions



mean 77.53, median 79, stdev 12.03

I. Warm-up

(a) (15 points) Circle true or false for each statement (no need to justify your answers here).

True **False** In xv6 and JOS, a Bootstrap Processor sends a message (i.e., STARTUP interprocessor interrupt) to wake up an Application Processor (AP). The message contains the physical address the AP should start executing from.

Solution: T

True **False** Before paging is turned on in JOS, setting a breakpoint in GDB at a high address (e.g., `0xf010000c`) will not trigger the breakpoint.

Solution: T

True **False** The JOS kernel will not be interrupted by clock interrupts when it is executing a system call in the kernel, because external interrupts are disabled when in the kernel.

Solution: T

True **False** A Dune process can directly modify the CR3 control register, because it is running in ring 0. As a comparison, a JOS user environment cannot do so because it is running in ring 3.

Solution: T

True **False** Similarly to JOS, the xv6 file system uses inodes to store which file names are associated with a directory.

Solution: F

(b) (8 points) In JOS, the `breakpoint` test case (`user/breakpoint.c`) invokes the `int` instruction, as follows:

```
void umain(int argc, char **argv)
{
        asm volatile("int $3");
}
```

Ben Bitdiddle wants to understand whether this test case generates a break point exception or a general protection fault from user space. Meanwhile, in `inc/mmu.h`, he notices a useful macro `SETGATE` for setting up the IDT:

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   see section 9.6.1.3 of the i386 reference: "The difference between
//   an interrupt gate and a trap gate is in the effect on IF (the
//   interrupt-enable flag). An interrupt that vectors through an
//   interrupt gate resets IF, thereby preventing other interrupts from
//   interfering with the current interrupt handler. A subsequent IRET
//   instruction restores IF to the value in the EFLAGS image on the
//   stack. An interrupt through a trap gate does not change IF."
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, dpl)        { ... }
```

Fill in the blanks given the following definitions of variables and constants:

- `idt`: the interrupt descriptor table.
- `T_BRKPT`: the trap number for breakpoint (i.e., 3).
- `GD_KT`: the global descriptor number for kernel text.
- `Xbrkpt`: the trap handler for breakpoint.

i. If Ben wants the `breakpoint` test case to generate a breakpoint exception, he should set up the IDT for `T_BRKPT` as follows:

`SETGATE(idt[T_BRKPT], ______0______, GD_KT, &Xbrkpt, ______3______);`

ii. If Ben wants the `breakpoint` test case to generate a general protection fault, he should set up the IDT for `T_BRKPT` as follows:

`SETGATE(idt[T_BRKPT], ______0______, GD_KT, &Xbrkpt, ______0______);`

II. A few changes

Ben is proposing and implementing a few changes to JOS. Please help him decide whether these changes are correct or not.

(a) (5 points) Ben doesn't like the performance overhead incurred by system calls through the `int` instruction, dispatching to individual system calls, etc. He decides to move the implementation of `sys_cputs`, including all the code it uses, into user space, allowing user environments to directly call the code. Will his fast `sys_cputs` work? Briefly explain why or why not.

> **Solution:** No - the user space doesn't have the permissions to access the VGA buffer or I/O ports.

(b) (5 points) The use of the big kernel lock guarantees that only one CPU can run the JOS kernel code at a time, but Ben doesn't want to use separate kernel stacks for each CPU. He decides to use a single, shared kernel stack. In addition, instead of calling `lock_kernel()` in the C function `trap()` in `kern/trap.c`, he changes all his trap handlers in `kern/trapentry.S` to acquire the big kernel lock at the very beginning; the lock acquisition code is written in assembly code without using the stack. Is this new plan safe? If so, briefly explain why. If not, describe a scenario in which using a shared kernel stack will go wrong.

> **Solution:** No - this will corrupt the stack, because before entering the trap handler the CPU has already pushed to the stack (old `SS`, `ESP`, etc.).

III. Virtual memory

(a) (15 points) Check either “physical address” or “virtual address” for underlined values from JOS (no need to justify your answers here).

The kernel pointer `envs` pointing to an array of the `Env` structures:

□ physical address ■ **virtual address**

The user pointer at `UENVS (0xeec00000)` pointing to read-only copies of the `Env` structures:

□ physical address ■ **virtual address**

The value of the `%cr2` control register when a kernel page fault happened:

□ physical address ■ **virtual address**

The value of the `%cr2` control register when a page fault happened in user space:

□ physical address ■ **virtual address**

The NVMe disk’s memory-mapped IO region starting from `0xfebf0000`, indicated by its BAR 0:

■ **physical address** □ virtual address

(b) In December 2016, Intel published a white paper, "5-Level Paging and 5-Level EPT," which describes a planned new paging mode for future Intel processors. Alyssa P. Hacker wants to learn more about 5-level paging. Your job is to help Alyssa based on your experience with 2-level paging in JOS. Hint: you may want to read the questions before reading the excerpts.

The following are some (slightly modified) excerpts from Intel's white paper:

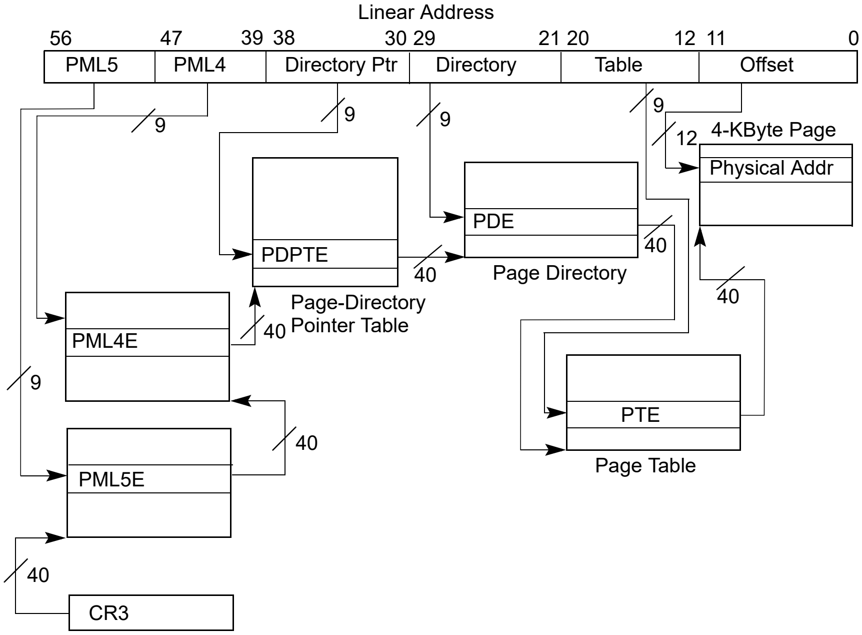
Modern operating systems use address-translation support called **paging**. Paging translates **linear addresses** (also known as **virtual addresses**), which are used by software, to **physical addresses**, which are used to access memory (or memory-mapped I/O).

IA-32e mode is a mode of processor execution that extends the older 32-bit operation, known as legacy mode. In IA-32e mode, linear addresses are 64 bits in size. However, the corresponding paging mode (called **IA-32e paging**) does not use all 64 linear-address bits.

IA-32e paging does not use all 64 linear-address bits because processors limit the size of linear addresses. This limit is enumerated by the CPUID instruction. Specifically, CPUID.80000008H: EAX[bits 15:8] enumerates the number of linear-address bits (the maximum linear-address width) supported by the processor. Processors that support 5-level paging are expected to enumerate this value as 57.

Processors also limit the size of physical addresses and enumerate the limit using CPUID. CPUID.80000008H:EAX[bits 7:0] enumerates the number of physical-address bits supported by the processor, the maximum physical-address width. Processors that support 5-level paging are expected to enumerate values up to 52.

The enumerated limitation on the linear-address width implies that paging translates only the low 57 bits of each 64-bit linear address. After a linear address is generated but before it is translated, the processor confirms that the address uses only the 57 bits that the processor supports. The limitation to 57 linear-address bits results from the nature of IA-32e paging, which is illustrated in the following figure:



The processor performs IA-32e paging by traversing a 5-level hierarchy of paging structures whose root structure resides at the physical address in control register CR3. Each paging structure is 4-KBytes in size and comprises 512 8-byte entries. The processor uses the upper 45 bits of a linear address (bits 56:12), 9 bits at a time, to select paging-structure entries from the hierarchy. The following items describe the translation process in more detail.

- Translation begins by identifying a 4-KByte naturally aligned PML5 table. It is located at the physical address specified in bits 51:12 of CR3. A PML5 table comprises 512 64-bit entries (PML5Es). A PML5E is selected using the physical address defined as follows.
  - Bits 51:12 are from CR3.
  - Bits 11:3 are bits 56:48 of the linear address.
  - Bits 2:0 are all 0.

  Because a PML5E is identified using bits 56:48 of the linear address, it controls access to a 256-TByte region of the linear-address space.

- The next step of the translation process identifies a 4-KByte naturally aligned PML4 table. It is located at the physical address specified in bits 51:12 of the PML5E. A PML4 table comprises 512 64-bit entries (PML4Es). A PML4E is selected using the physical address defined as follows.
  - Bits 51:12 are from the PML5E.
  - Bits 11:3 are bits 47:39 of the linear address.
  - Bits 2:0 are all 0.

  Because a PML4E is identified using bits 56:39 of the linear address, it controls access to a 512-GByte region of the linear-address space.

- Once the PML4E is identified, bits 38:0 of the linear address determine the remainder of the translation process, as illustrated in the figure from the previous page. Similarly to PML5 and PML4, the translation process will identify a page-directory-pointer table, a page-directory table, and a page table.

Because only bits 56:0 of a linear address are used in address-translation, the processor reserves bits 63:57 for future expansion using a concept known as **canonicality**. A linear address is **57-bit canonical** if bits 63:56 of the address are identical. Put differently, a linear address is canonical only if bits 63:57 are a sign-extension of bit 56, which is the uppermost bit used in linear-address translation.

When a 64-bit linear address is generated to access memory, the processor first confirms that the address is canonical. If the address is not canonical, the memory access causes a fault, and the processor makes no attempt to translate the address.

*** END OF EXCERPTS ***

To make it easier to read, integer literals in this page are separated with an underscore. For instance, we use `0x0000_cafe` and `0x0000cafe` interchangeably.

i. (3 points) Check one correct answer: 5-level paging allows up to _______ of linear-address space to be accessed at any given time.

- □ 2^{48} bytes (256 terabytes)
- □ 2^{52} bytes (4 petabytes)
- ■ **2^{57} bytes (128 petabytes)**
- □ 2^{64} bytes (16 exbibytes)

ii. (4 points) Check all that apply: which of the following linear addresses are considered **57-bit canonical**?

- ■ **`0x0000_0000_0000_0000`**
- ■ **`0x00ff_0000_0000_0000`**
- ■ **`0xff80_0000_0000_0000`**
- ■ **`0xffff_ffff_ffff_ffff`**

iii. (10 points) Fill in the blanks with correct answers (no need to justify your answers here). Suppose the value of CR3 is `0x0000_0000_1fff_0000`. Below is the content of the PML5 table at physical address `0x0000_0000_1fff_0000`:

511	`0x0000_0000_0000_0001`
…	…
2	`0x0000_0000_0123_6001`
1	`0x0000_0000_0123_5001`
0	`0x0000_0000_0123_4001`

As described in the white paper, the PML5 table is 4-KBytes in size and has 512 64-bit entries. For instance, the first entry has value `0x0000_0000_0123_4001` and the last entry has value `0x0000_0000_0000_0001`. Now consider the translation of linear address `0x0000_0000_0001_cafe` (ignoring present and permission bits for this question).

- The physical address of the PML5 entry (PML5E) corresponding to linear address `0x0000_0000_0001_cafe` is **`0x0000_0000_1fff_0000`**.

- The physical address of the PML4 entry (PML4E) corresponding to linear address `0x0000_0000_0001_cafe` is **`0x0000_0000_0123_4000`**.

IV. File system

0	1	2	3	32	58	59
boot block	super block	log header	log blocks	inode blocks	free bitmap	data blocks

The disk layout of the xv6 file system is illustrated in the above figure:

- the super block is in block 1;
- the log header is in block 2 and the log is in blocks 3–31;
- inodes are in blocks 32–57;
- the bitmap of free blocks is in block 58; and
- data blocks start from block 59 to end of the disk.

To trace disk writes, Alyssa modifies `iderw()` in the IDE driver code (`ide.c`) to print the block number of each block written. She boots xv6 with a fresh `fs.img`, which initially contains a few files within the root directory (e.g., `README`). She then types in the command "`rm README`" to remove the file. Alyssa observes the following output:

```
$ rm README
write 3
write 4
write 5
write 2
write 59
write 33
write 58
write 2
$
```

(a) (15 points) Match writes to their descriptions: the left side contains the last five writes observed by Alyssa; for each write, choose a letter of the most appropriate description from the right side and fill in the blank. Each letter may be used once, more than once, or not at all.

b write 2
e write 59
f write 33
h write 58
c write 2

a. update the super block
b. mark the transaction as "done" in the log
c. delete the transaction from the log
d. update the root directory's inode
e. update the root directory's data blocks
f. update `README`'s inode
g. update `README`'s data blocks
h. free `README`'s data blocks in the free bitmap

(b) (9 points) Alyssa notices that xv6 does not issue the flush command to the disk. A modern disk controller may reorder writes sent by the driver. For instance, the writes observed by Alyssa are:

write 3, write 4, write 5, write 2, …

The disk controller may execute the writes in a different order, such as:

write 2, write 3, write 4, write 5, …

Alyssa worries that such reordering may violate the correctness of the xv6 file system (e.g., corrupting the file system after a power failure). Since the disk controller will not reorder writes across a flush, she decides to modify xv6 and add a flush after every disk write. This time, running “`rm README`” on a fresh `fs.img` produces the following trace:

write 3
① flush
write 4
② flush
write 5
③ flush
write 2
④ flush
write 59
⑤ flush
write 33
⑥ flush
write 58
⑦ flush
write 2
flush

Alyssa wonders which of these disk flushes are necessary to ensure correctness and which can be safely removed. For each flush (except for the last one), there is a number to the left. List all the flushes that Alyssa must keep, using their numbers (no need to justify your answer).

Solution: ③④⑦

(c) (3 points) Originally, an xv6 inode contains 12 direct block numbers and one singly indirect block number. In the “big file” exercise, Alyssa modifies the xv6 file system to support doubly indirect blocks: now an inode contains 11 direct block numbers, one singly indirect block number, and one doubly indirect block number. In the new file system, will running “`rm README`” on a fresh `fs.img` produce the same number of disk writes? Briefly explain why or why not.

Solution: Yes - ”`rm README`” touches only the data blocks of the root directory. Its size is small in a fresh file system image, so adding doubly indirect blocks won’t affect the writes.

V. CSE 451

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

(a) (2 points) For exercises and labs, did you work in pairs or on your own? Can you tell us why?

Solution: Most worked alone: hard to coordinate on time (13), didn't know others in class (7), could learn more (6), more efficient (4).

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

Solution: Add: networking (9), more file systems (4), more concurrency (4), more security and hacking (3), scheduling (3), among many others (e.g., linking, Rust, drivers, GUI, cloud).

Remove: paper reading (2).

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

Solution: labs (16), understanding OS (15), staff (4).

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

Solution: bugs & debugging (19), no slides (4), final (3).

End — Enjoy the break!