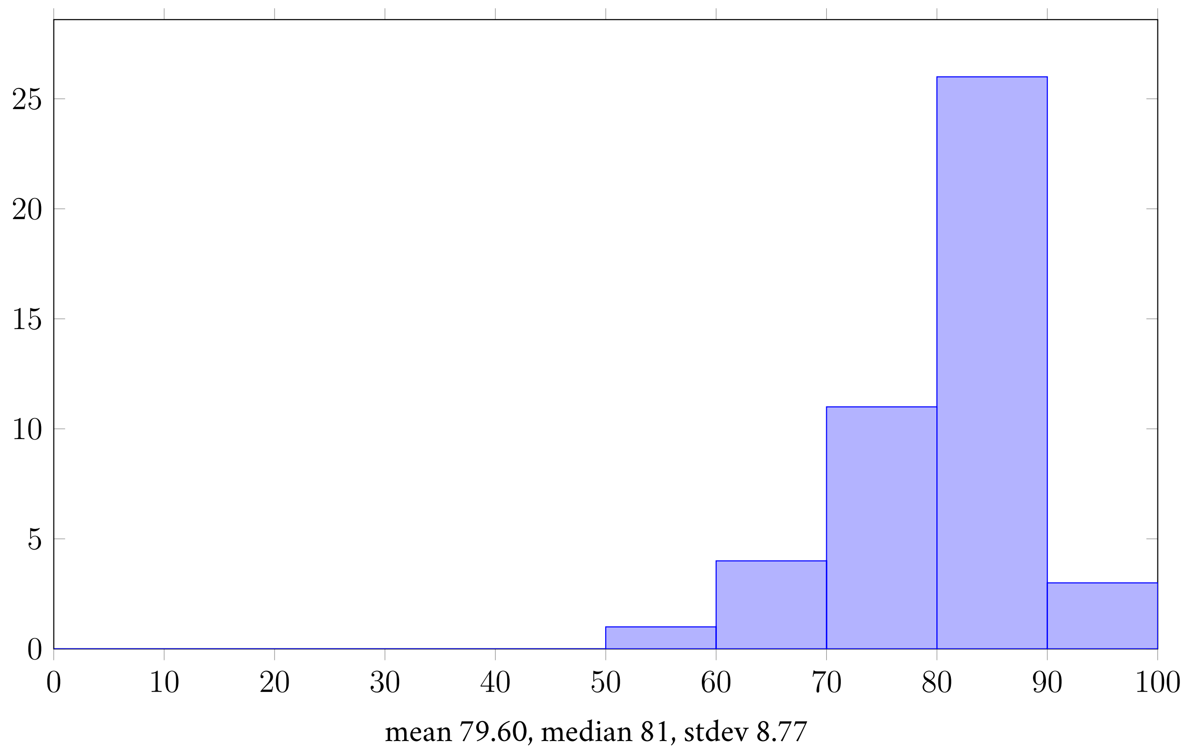
COMPUTER SCIENCE & ENGINEERING

UNIVERSITY *of* WASHINGTON

CSE 451 Spring 2018

Final Solutions



mean 79.60, median 81, stdev 8.77

I. (15 points) Warm-up

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

True **False** The JOS kernel runs above KERNBASE because the x86 CPU requires kernel code to reside in the higher half of the address space.

Solution: F

True **False** The xv6 kernel uses one kernel stack per process, while the JOS kernel uses one kernel stack per CPU.

Solution: T

True **False** The xv6 kernel can take hardware interrupts while running in both kernel and user mode, and the JOS kernel takes hardware interrupts only in kernel mode.

Solution: F

True **False** Meltdown is a bug caused by the OS kernel incorrectly setting the U bit for the kernel portion of the virtual address space in the page table.

Solution: F

True **False** A formally verified OS kernel must be free of bugs, as it provides a mathematical proof showing that its implementation adheres to its specification.

Solution: F

II. (15 points) Exceptions

Ben Bitdiddle is debugging a crash in his JOS. He observes the following output:

```
...
TRAP frame at 0xf021c000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0x00000000
  err  0x00000006 [user, write, not-present]
  eip  0x00800036
  cs   0x----001b
  flag 0x00000082
  esp  0xeebfdfd0
  ss   0x----0023
...
```

To help Ben understand the output, please match the output to their descriptions: for each of the five lines of output observed by Ben, choose a letter of the most appropriate description from below and fill in the blank. Each letter may be used once, more than once, or not at all.

____**i**____ `trap 0x0000000e`
____**c**____ `cr2  0x00000000`
____**g**____ `err  0x00000006`
____**d**____ `eip  0x00800036`
____**e**____ `flag 0x00000082`

a. the physical memory address that the CPU attempts to access when a page fault occurs
b. the physical memory address of an instruction being executed where a page fault occurs
c. the virtual memory address that the CPU attempts to access when a page fault occurs
d. the virtual memory address of an instruction being executed where a page fault occurs
e. the saved value of the EFLAGS register from user space
f. the saved value of the EFLAGS register from kernel space
g. the error code pushed by the CPU
h. the error code pushed by the JOS kernel
i. the exception vector number

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).

A memory range starting from `0x000f0000` in an E820 memory map:

■ **physical address** □ virtual address

`MPENTRY_PADDR` (`0x7000`) at which application processors (APs) start running:

■ **physical address** □ virtual address

The address of the Interrupt Descriptor Table (IDT) loaded into the CPU:

□ physical address ■ **virtual address**

The `struct Trapframe *tf` pointer passed into the `trap()` function in the kernel:

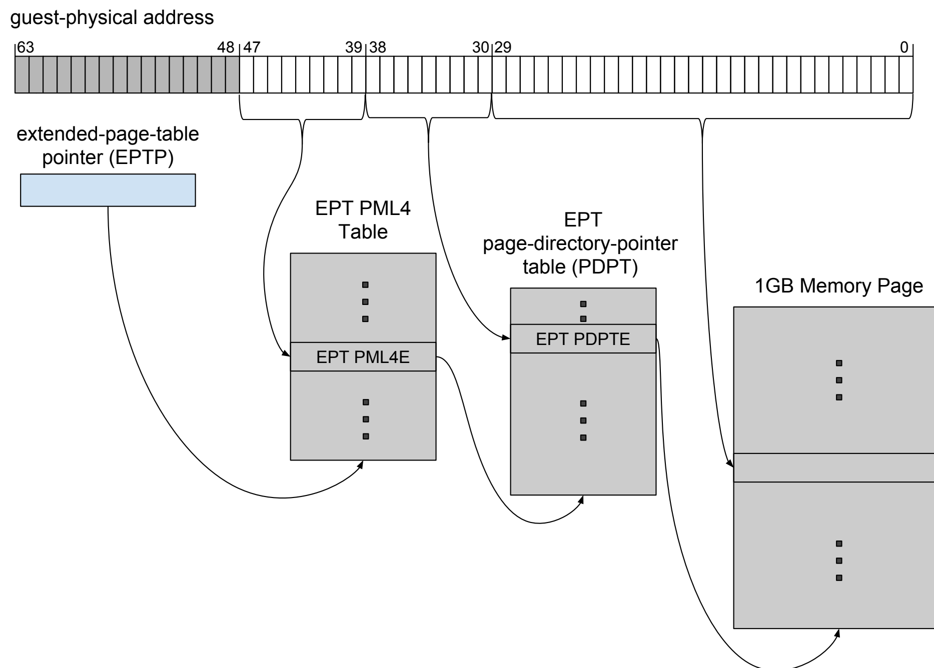
□ physical address ■ **virtual address**

The value of the stack pointer register `%esp` right before the kernel enters the `trap()` function:

□ physical address ■ **virtual address**

(b) After working on the JOS labs, Alyssa P. Hacker decides to develop a hypervisor and plans to use the extended page-table (EPT) to safely isolate multiple virtual machines through the virtualization of physical memory. When EPT is in use, physical addresses in virtual machines (known as **guest-physical addresses**) are translated by traversing a set of **EPT paging structures** to produce physical addresses that are used to access memory.

In particular, Alyssa is interested in a two-level EPT plan: an EPT starts from the extended-page-table pointer (EPTP), which holds the address of the first-level EPT paging structure, the EPT PML4 table (which in turn points to the next level). This is illustrated as follows:



In some way, this plan looks similar to the two-level paging used by JOS, with a few differences. For example, the architecture is 64-bit (rather than 32-bit); the root of the EPT is held in EPTP (rather than CR3); and this EPT controls 1-GB pages (rather than 4-KB pages). Your job is to help Alyssa based on your experience with JOS.

The following is a slightly modified excerpt from Intel's manuals. You may want to read the questions before reading the excerpt.

*** BEGINNING OF EXCERPT ***

The EPT translation mechanism uses only bits 47:0 of each guest-physical address. In this plan, it uses a page-walk length of 2, meaning that 2 EPT paging-structure entries are accessed to translate a guest-physical address. These 48 bits are partitioned by the logical processor to traverse the EPT paging structures:

- A 4-KByte naturally aligned EPT PML4 table is located at the physical address specified in bits 51:12 of the extended-page-table pointer (EPTP), a field in the virtual machine control structure. An EPT PML4 table comprises 512 64-bit entries (EPT PML4Es). An EPT PML4E is selected using the physical address defined as follows:

- Bits 63:52 are all 0.
  - Bits 51:12 are from the EPTP.
  - Bits 11:3 are bits 47:39 of the guest-physical address.
  - Bits 2:0 are all 0.

  Because an EPT PML4E is identified using bits 47:39 of the guest-physical address, it controls access to a 512-GByte region of the guest-physical-address space.

- A 4-KByte naturally aligned EPT page-directory-pointer table (PDPT) is located at the physical address specified in bits 51:12 of the EPT PML4E. An EPT page-directory-pointer table (PDPT) comprises 512 64-bit entries (EPT PDPTEs). An EPT PDPTE is selected using the physical address defined as follows:
  - Bits 63:52 are all 0.
  - Bits 51:12 are from the EPT PML4E.
  - Bits 11:3 are bits 38:30 of the guest-physical address.
  - Bits 2:0 are all 0.

  Because an EPT PDPTE is identified using bits 47:30 of the guest-physical address, it controls access to a 1-GByte region of the guest-physical-address space. Use of the EPT PDPTE depends on the value of bit 7 in that entry:

- If bit 7 of the EPT PDPTE is 1, the EPT PDPTE maps a 1-GByte page. The final physical address is computed as follows:
  - Bits 63:52 are all 0.
  - Bits 51:30 are from the EPT PDPTE.
  - Bits 29:0 are from the original guest-physical address.

- Alyssa does not plan to support the case where bit 7 of the EPT PDPTE is 0.

An EPT paging-structure entry is **present** if any of bits 2:0 is 1; otherwise, the entry is **not present**. A reference using a guest-physical address whose translation encounters an EPT paging-structure that is not present causes an EPT violation.

The following figure gives a summary of the formats of the EPTP and the EPT paging-structure entries (EPT PML4 and EPT PDPTE):

63 … 52	51 50 … 12	11	10	9	8	7	6	5	4	3	2	1	0	
	physical address of EPT PML4 table							0	1	1	1	1	0	EPTP
	physical address of EPT PDPT										X	W	R	PML4E
	physical address of 1 GB page					1					X	W	R	PDPTE

Gray fields are reserved/ignored and must be 0. The fields of R (read access), W (write access), and X (execute access) indicate whether reads, writes, and instruction fetches are allowed from the memory region controlled by this entry, respectively.

*** END OF EXCERPT ***

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

Given a virtual machine, Alyssa plans to construct an EPT to ensure that it cannot access memory of other virtual machines or the hypervisor. More specifically, the EPT consists of one EPT PML4 table and one EPT page-directory-pointer table (PDPT):

- the PML4 table resides at physical address `0x0800_0000`; and
- the EPT PDPT resides at physical address `0x0800_1000`.

Together, the EPT maps guest-physical addresses in the range [`0`, `0x3fff_ffff`] (i.e., the first 1 GB) in the virtual machine to physical addresses [`0x0001_0000_0000`, `0x0001_3fff_ffff`] (i.e., the 1-GB range starting from 4 GB) in the hypervisor. Alyssa wants to grant read, write, and execute accesses to guest-physical addresses in the range [`0`, `0x3fff_ffff`]; accesses to other guest-physical addresses should be denied and cause an EPT violation.

Hint: the binary representation of `0x1e` is `11110`.

i. (5 points) : Check all that apply: using this EPT, which of the following guest-physical addresses will not cause an EPT violation?

- ■ **`0x0`**
- ■ **`0x0800_0000`**
- ■ **`0x0800_1000`**
- ☐ `0x4000_0000`
- ☐ `0x0001_0000_0000`

ii. (5 points) Check one correct answer: the value of the extended-page-table pointer (EPTP) for this virtual machine is _______.

- ☐ `0x1e`
- ■ **`0x0800_001e`**
- ☐ `0x0800_101e`
- ☐ `0x4000_001e`
- ☐ `0x0001_0000_001e`

iii. (10 points) As described in the excerpt, an EPT PML4 table is 4-KBytes in size and has 512 64-bit entries. Fill in the blanks for the first two entries of the EPT PML4 table for the virtual machine (no need to justify your answers here).

511	0x0000_0000_0000_0000
…	…
2	0x0000_0000_0000_0000
1	??
0	??

- The value of entry 0 of the EPT PML4 table is **0x0800_1007**.
- The value of entry 1 of the EPT PML4 table is **0x0**.

IV. (10 points) Locking

Ben Bitdiddle wants to improve the spinlock implementation used in JOS, by adding fairness guarantees. Specifically, he implements a tick spinlock:

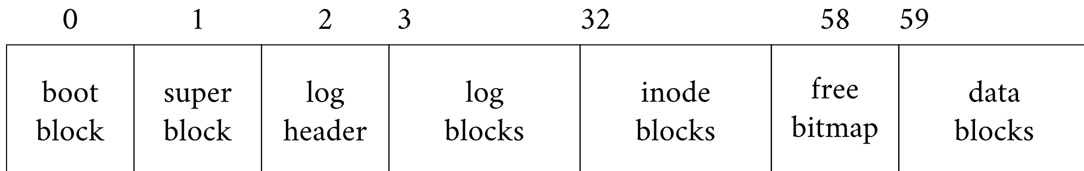
```
struct lock {
        unsigned int next, now_serving;
};

void acquire(struct lock *l)
{
        unsigned int ticket = l->next++;

        while (l->now_serving != ticket)
                ;
}

void release(struct lock *l)
{
        ++l->now_serving;
}
```

He plans to use this modified lock implementation for the multithreaded hash table program from lecture, where two threads concurrently insert keys into a hash table. While running this program on a multicore system, he notices “5644 keys missing” in the output. Please help Ben understand why his lock implementation fails.

Check all possible reasons:

- ■ **there is a race condition between `acquire()` and `release()`**
- ■ **the CPUs have a weak memory consistency model and reorder loads/stores**
- □ the CPUs perform speculative execution and reorder loads/stores
- ■ **the C compiler performs aggressive optimizations as data races are undefined in C**
- □ assembly code is required to implement a lock as there is no safe way to do so in C

V. File systems



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.

Alyssa types in the command "`echo > newfile`" to create an empty file under the root directory, and observes the following output:

```
$ echo > newfile
write 3
write 4
write 5
write 6
write 2
write 36
write 58
write 582
write 32
write 2
```

She uses `ls` to observe that the inode number of `newfile` is 32.

(a) (10 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.

Answer	Write	Description
b	write 36	a. update the root directory's inode
d	write 58	b. update `newfile`'s inode
c	write 582	c. write the root directory's newly allocated data block
a	write 32	d. allocate a data block for the root directory in the free bitmap
e	write 2	e. delete the transaction from the log

(b) (5 points) Alyssa is not happy that creating an empty file currently requires 10 writes. She wants to improve the performance of the xv6 file system by reducing the number of writes. Specifically, she doesn't like the use of the log and decides to change the file system to skip writes to the log (i.e., blocks 2–31) altogether. After the change, creating an empty file requires only the following 4 writes (rather than 10 writes):

a. update the root directory's inode
b. update `newfile`'s inode
c. write the root directory's newly allocated data block
d. allocate a data block for the root directory in the free bitmap

Your job is to help her order the four writes in a way such that no matter when the machine crashes between the writes, the file system is never corrupted (but it can "leak" data blocks and inodes by rendering them unusable). Assume each write is atomic and the disk does not reorder writes.

Give one safe order of the four writes using their letters (no need to justify your answers).

first: ____________ → ____________ → ____________ → ____________ last.

Solution: A safe order should satistfy b → c, c → a, and d → a. Therefore, any of the following three orders is safe:

- b → c → d → a
- b → d → c → a
- d → b → c → a

VI. CSE 451

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

(a) (3 points) What was your favorite topic in CSE 451?

Solution: traps/syscalls (12), virtual memory (11), process management (8), file systems (7)

(b) (3 points) Which topic you would like to see removed for next year?

Solution: booting (5), virtual machines (4), verification (3), papers (3), concurrency (3)

(c) (3 points) Are there any topics you would like to see added to the class?

Solution: networking (14), recent research (4), more virtualization (4), more I/O (4)

(d) (1 point) Circle true or false for the statement.

True **False** The Meltdown exploits "do not have the potential to corrupt, modify or delete data."

End — Enjoy the break!