COMPUTER SCIENCE & ENGINEERING

UNIVERSITY *of* WASHINGTON

CSE 451 Fall 2018

Final

You have 110 minutes to answer the questions in this exam. 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:	20	20	10	20	20	10	100
Bonus Points:	0	0	0	5	0	0	5
Score:							

I. (20 points) Warm-up

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

True **False** Both xv6 and JOS enable virtual memory for user applications and disable virtual memory in the kernel.

True **False** The xv6 kernel can take new timer interrupts while handling a timer interrupt, and the JOS kernel handles each timer interrupt to completion.

True **False** When JOS runs on multiple x86 CPUs, at most one CPU can be in the privilege level of ring 0 (i.e., kernel mode).

True **False** Dune provides applications with access to privileged CPU features, but incurs a performance penalty, such as slower garbage collection.

True **False** One reason that IX outperforms Linux is that its network stack is based on lwIP, which is more optimized than the Linux TCP/IP stack.

II. (20 points) Memory addresses

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

The kernel pointer `pages` pointing to an array of `PageInfo` structures:

☐ physical address ☐ virtual address

The top of the first CPU's kernel stack `KSTACKTOP` (`0xf0000000`):

☐ physical address ☐ virtual address

A memory range starting from `0xf0000000` in an E820 memory map:

☐ physical address ☐ virtual address

The value of `tf_eip` in `struct Trapframe` passed into the `trap()` function in the kernel:

☐ physical address ☐ virtual address

The address of the Interrupt Descriptor Table (IDT) in a Dune process (VMX non-root, ring 0):

☐ physical address ☐ virtual address

III. User space

(a) (5 points) Ben Bitdiddle tries to run the following user program on xv6.

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[])
{
    char *message = "hello";
    int pid;

    pid = fork();
    if (pid > 0) {
        /* parent process */
        message = "world";
        close(1);
    }

    write(1, message, 5);

    if (pid > 0) {
        /* parent process */
        wait();
    }

    exit();
}
```

Assume that file descriptor 1 is connected to the terminal when the program starts, and that `write()` does not output anything when called on an invalid file descriptor. Which of the following outputs may Ben observe? Check all that apply.

- ☐ "hello"
- ☐ "hellohello"
- ☐ "helloworld"
- ☐ "world"
- ☐ "worldhello"
- ☐ "worldworld"

(b) (5 points) After seeing the lab X demos, Ben wants to try fine-grained locking. Recall the multi-threaded hash table from lecture: initially, some keys were missing after being inserted into the hash table; this was caused by a race condition between two threads executing `put()`.

To achieve fine-grained locking, Ben now adds `acquire()` and `release()` calls with a global `lock` to `insert()`, as follows:

```
struct entry {
    int key;
    int value;
    struct entry *next;
};
struct entry *table[NBUCKET];
struct lock lock;
...

void insert(int key, int value, struct entry **p, struct entry *n)
{
    struct entry *e;

    e = malloc(sizeof(struct entry)); /* (1) */
    e->key = key;                     /* (2) */
    e->value = value;                 /* (3) */
    e->next = n;                      /* (4) */

    acquire(&lock);
    *p = e;                           /* (5) */
    release(&lock);
}

void put(int key, int value)
{
    int i = key % NBUCKET;
    insert(key, value, &table[i], table[i]);
}
```

Ben observes that some keys still are missing. Which of the lines marked with `(1)`–`(5)` might have caused data races? Check all that apply.

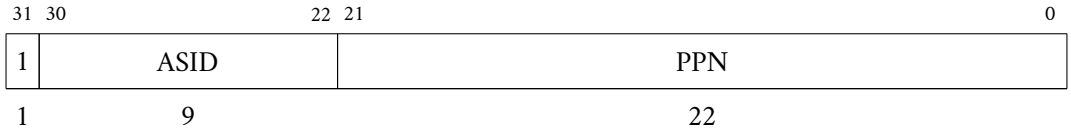
- ☐ (1)
- ☐ (2)
- ☐ (3)
- ☐ (4)
- ☐ (5)

IV. Page tables

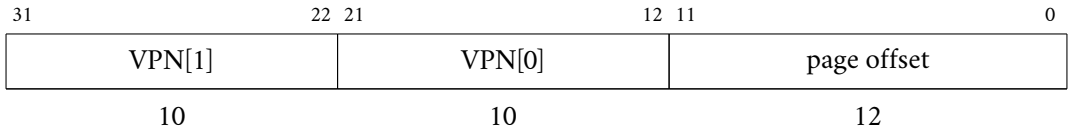
Alyssa P. Hacker plans to port JOS to 32-bit RISC-V, an open-source instruction set architecture also known as RV32. The following is a slightly modified excerpt from the *RISC-V Instruction Set Manual*. Your job is to help Alyssa based on your experience with JOS. You may want to read the questions before reading the excerpt.

RISC-V defines the following privilege levels: machine, supervisor, and user. Machine-mode can be used to manage secure execution environments on RISC-V. User-mode and supervisor-mode are intended for conventional application and operating system usage respectively.

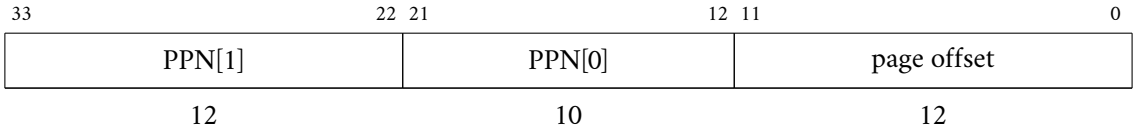
On RV32, the 32-bit Supervisor Address Translation and Protection (`satp`) register controls address translation and protection, formatted as shown below. The `satp` register holds the physical page number (PPN) of the root page table, i.e., its physical address divided by 4 KiB; an address space identifier (ASID), which is reserved and must be zero; and the top bit, which must be 1. Storing a PPN in `satp` supports a physical address space larger than 4 GiB for RV32.



RV32 provides a paged virtual-memory scheme called Sv32. Sv32 supports a 32-bit virtual address space, divided into 4KiB pages. An Sv32 virtual address is partitioned into a virtual page number (VPN) and page offset, as shown below:

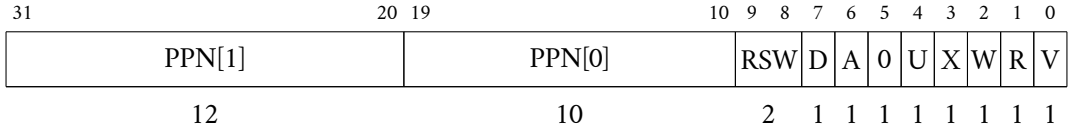


When Sv32 is used, virtual addresses are translated into physical addresses via a two-level page table. The 20-bit VPN is translated into a 22-bit physical page number (PPN), while the 12-bit page offset is untranslated. A resulting physical address is shown below:



Sv32 page tables consist of 2^{10} page-table entries (PTEs), each of four bytes. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the `satp` register.

The PTE format is as follows. The V bit indicates whether the PTE is valid; if it is 0, all other bits in the PTE are don't-cares and may be used freely by software. The permission bits, R, W, and X, indicate whether the page is readable, writable, and executable, respectively. When all three are zero, the PTE is a pointer to the next level of the page table; otherwise, it is a leaf PTE. Writable pages must also be marked readable; the contrary combinations are reserved for future use.



X	W	R	Meaning
0	0	0	Pointer to next level of page table
0	0	1	Read-only page
0	1	0	*Reserved for future use*
0	1	1	Read-write page
1	0	0	Execute-only page
1	0	1	Read-execute page
1	1	0	*Reserved for future use*
1	1	1	Read-write-execute page

The above table summarizes the encoding of the permission bits.

- Attempting to fetch an instruction from a page that does not have execute permissions raises a fetch page-fault exception. Attempting to execute a load instruction whose effective address lies within a page without read permissions raises a load page-fault exception. Attempting to execute a store instruction whose effective address lies within a page without write permissions raises a store page-fault exception.
- The U bit indicates whether the page is accessible to user mode. User-mode software may only access the page when U=1. The supervisor may not execute code on pages with U=1.
- The RSW field is reserved for use by supervisor software.
- Each leaf PTE contains an accessed (A) and dirty (D) bit. When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.
- For non-leaf PTEs, the D, A, and U bits are reserved for future use and must be cleared by software for forward compatibility.

To summarize, a virtual address *va* is translated into a physical address *pa* as follows:

1. Let a be satp.$ppn \times$ PAGESIZE, and let i = LEVELS − 1. (For Sv32, PAGESIZE = 2^{12} and LEVELS = 2.)
2. Let *pte* be the value of the PTE at address $a + va.vpn[i] \times$ PTESIZE. (For Sv32, PTESIZE = 4.)
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, stop and raise a page-fault exception.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. If $i > 0$, this is an unsupported superpage; stop and raise a page-fault exception.
6. A leaf PTE has been found. Determine if the requested memory access is allowed by the *pte.r*, *pte.w*, *pte.x*, and *pte.u* bits. If not, stop and raise a page-fault exception.
7. If $pte.a = 0$, or if the memory access is a store and $pte.d = 0$, raise a page-fault exception.
8. The translation is successful. The translated physical address is given as follows:
   - $pa.pgoff = va.pgoff$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

*** END OF EXCERPT ***

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

(a) (5 points) Check one correct answer. The RV32 architecture supports a physical address space of up to _____ bits.

- ☐ 30
- ☐ 31
- ☐ 32
- ☐ 33
- ☐ 34
- ☐ none of the above

(b) (5 points) Check all that apply. Suppose executing a store instruction in user mode succeeds (i.e., no page faults). In the leaf PTE corresponding to the virtual address the store instruction attempts to write, which of the following fields must be 1 during the execution of the store instruction?

- ☐ V (valid)
- ☐ R (readable)
- ☐ W (writable)
- ☐ X (executable)
- ☐ A (accessed)
- ☐ D (dirty)
- ☐ none of the above

(c) Recall that on x86 the `CR3` register points to a page-directory page, which in turn points to the next level. Alyssa sets up a similar two-level page table on RV32: the `satp` register points to a non-leaf page-table page, which in turn points to a leaf page-table page.

Specifically, the non-leaf page-table page resides at physical address `0x8000_1000`, and the leaf page-table page resides at physical address `0x8000_2000`. This page table maps the 4 MiB range of virtual addresses [`0x0`, `0x003f_ffff`] to physical addresses [`0x8100_0000`, `0x813f_ffff`], with R and W permissions in supervisor-mode only.

Below is the content of the non-leaf page-table page:

1023	`0x0000_0000`
…	…
1	`0x0000_0000`
0	??

Hint: the binary representation of `0x8` is `1000`.

i. (5 points) Check one correct answer. The value of the `satp` register is _____.

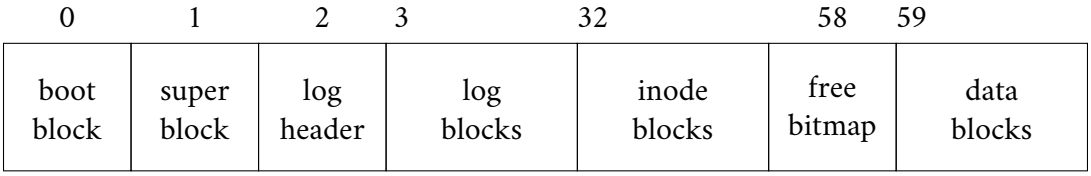
- ☐ `0x8000_1000`
- ☐ `0x1008_0000`
- ☐ `0x1008_0001`
- ☐ `0x8008_0001`
- ☐ none of the above

ii. (5 points) Check all that apply. For entry 0 of the non-leaf page-table page, which of the following fields must be 1?

- ☐ V (valid)
- ☐ R (readable)
- ☐ W (writable)
- ☐ X (executable)
- ☐ A (accessed)
- ☐ D (dirty)
- ☐ none of the above

iii. (5 points (bonus)) Fill in the blank. Assume the RSW field is 0. The value of entry 0 of the non-leaf page-table page is `0x`_____.

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.

A hard link is a directory entry that associates a name with an inode. Initially, the `README` file under the root directory has one hard link. Alyssa types in the command "`ln README ALIAS`" to create another hard link `ALIAS` to the `README` file. This gives the file two hard links (i.e., `README` and `ALIAS`), in other words, two directory entries to the same inode. The `nlink` field in the inode tracks the number of hard links to this inode; in this case, `nlink` changes from 1 to 2 after running `ln`.

To trace disk writes, Alyssa modifies `iderw()` in the IDE driver code (`ide.c`) to print the block number of each block written and observes the following output:

```
$ ln README ALIAS
write 3
write 4
write 2
write 32
write 59
write 2
```

Using `ls`, Alyssa observes the inode number of the root directory is 1 and that of `README` (`ALIAS`) is 2.

(a) (15 points) Match writes to their descriptions: the left side contains the last three writes observed by Alyssa; for each write, choose the 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.

_____ write 32	a. complete the transaction in the log
_____ write 59	b. erase the transaction from the log
_____ write 2	c. update the root directory's inode
	d. update `README`'s inode
	e. write the root directory's data block
	f. write `README`'s data block

(b) (5 points) Alyssa optimizes the xv6 file system by skipping writes to the log (i.e., blocks 2–31) altogether. On the optimized file system, invoking the `ln` command to create a hard link `ALIAS` to `README` (from the original disk state) incurs only the following 2 writes (instead of 6 writes):

```
$ ln README ALIAS
write 32
write 59
```

Assume each write is atomic and the disk may cache and reorder writes. If the power suddenly goes off during the execution of `ln` on the optimized file system, which of the following states may Alyssa observe on disk? Check all that apply.

- ☐ `ALIAS` exists and the `README` file's `nlink` is 1
- ☐ `ALIAS` exists and the `README` file's `nlink` is 2
- ☐ `ALIAS` does not exist and the `README` file's `nlink` is 1
- ☐ `ALIAS` does not exist and the `README` file's `nlink` is 2
- ☐ none of the above

VI. CSE 451

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

(a) (3 points) What is the most important thing you would like to see fixed about CSE 451?

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

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

(d) (1 point) Check one: which of the following is most effective for debugging JOS?

- ☐ running gdb
- ☐ inserting `cprintfs`
- ☐ discussing with others
- ☐ none of the above

End — Enjoy the break!