Operating Systems Final ECE344, Winter 2021

Duration: 3 hours

Examiner: D. Yuan

Instructions

Examination Aids: This is an open book exam.

All questions have been provided in this exam booklet. You need to provide your answers in Quercus. Copy the answers of all sub-questions into the text box of each question.

If any of the questions appear unclear or ambiguous to you, then make any assumptions you need, state them and answer the question that way. If you believe there is an error, state what the error is, fix it, and respond as if fixed.

Please be brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer.

The questions are not necessarily ordered by the difficulty.

All the names in this exam are made-up -- don't read too much into them.

Work independently.

MARKING GUIDE

Q1: 9
Q2: 13
Q3: 7
Q4: 5
Q5: 34
TOTAL: (68)

Question 1 (9 marks): Memory Management (I)
When you clean up your room, you found a laptop from more than 10 years ago. You want to make it useful and install the latest TensorFlood on it. However, TensorFlood won't run on it. You try to understand why and perhaps solve the problem. Here are some specs on this laptop:

Operating system: Doors 7 32-bit with full virtual memory support and the page size of 4KB
CPU: Intal Kore 7 x344: it has a 32-bit processor with hardware-managed TLB
Physical Memory (RAM): 2 GB
Hard drive: 128GB

Answer the following questions:

a. (1 mark) How large is the virtual address space in Bytes of your laptop?

b. (1 mark) What's the number of bits of the Virtual Page Number of your laptop?

According to TensorFlood's website, it requires at least 5GB of RAM to run. Your roommate Dijkstra thinks that it's the reason why it won't run and suggested you upgrade the RAM to 6GB.

c. (1 mark) Do you think upgrading the RAM will work? Why?

d. (1 mark) However, you found that an alternative to TensorFlood, Matlad, which requires 3GB of memory, can run on your laptop without any problem, even with only 2GB of RAM. Why?

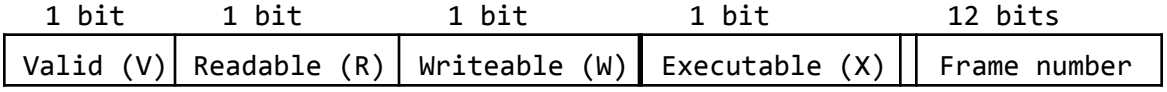
When you browse Heapoverflow, some people suggest that enlarging the page size will allow your laptop to support larger memory. And you found that your laptop's hardware and OS allows you to change the size of page size to 2MB. Dijkstra thinks that by changing the page size to 2MB will allow you to run TensorFlood.

e. (2 mark) Do you think it's true? Why?

f. (3 marks) What other benefits will the larger page size bring?

Question 2 (13 marks): Memory Management (II)

You need to port the virtual memory system for 344OS to a new architecture. The processor has a 16-bit virtual address space. A two-level page table is used, with 16 entries in the page directory (first-level) and 64 entries in the second-level page table. Each page table is page aligned, i.e., its beginning address is at the start of a page. Each page table entry is two bytes wide and has the following format



Note that for both levels of page tables, the last 12 bits of the page table entry are **the page frame (i.e., physical page) number** instead of being the complete physical address.

For page directories, the R, W, and X bits are unused and should be set to zero. For second-level page tables, the R bit indicates whether the page can be read by the thread; the W bit indicates whether the page can be written by the thread; and the X bit indicates whether the page can be used to fetch instructions for execution.

A partial listing of the machine's physical memory is shown below. (For convenience we are showing the contents of memory four bytes at a time. For example, the value at the physical address 0x01 in memory is 0xf7.)

0x00	6b f7 30 7b	**0xa0**	8a be 9b 09	**0x140**	7c a7 00 80
0x04	da 0b 6a ff	**0xa4**	53 f0 13 31	**0x144**	e6 30 d2 71
0x08	9f 07 7a 8e	**0xa8**	b4 38 60 85	**0x148**	0f 1e 21 9e
0x0c	42 47 b8 2e	**0xac**	e3 8e 2b 36	**0x14c**	f1 98 97 6e
0x10	c6 17 d2 41	**0xb0**	34 bc e0 07	**0x150**	c0 12 d0 04
0x14	50 11 f0 00	**0xb4**	0b 3f 30 1a	**0x154**	11 81 d0 72
0x18	b6 ff f2 4c	**0xb8**	16 31 61 7c	**0x158**	96 7b d6 5b
0x1c	f3 ce a4 28	**0xbc**	38 ab 42 06	**0x15c**	e3 bb ef e6
0x20	c0 5a 18 25	**0xc0**	64 0f aa ac	**0x160**	0b a1 c6 d4
0x24	95 95 b9 be	**0xc4**	72 d2 07 b6	**0x164**	c7 1a a3 ca
0x28	0b c2 01 f1	**0xc8**	cc 12 c2 50	**0x168**	6b f7 30 7b
0x2c	c9 f7 f8 88	**0xcc**	05 ca 78 2b	**0x16c**	ad 6c 01 22
0x30	44 16 6e 48	**0xd0**	02 ce 70 3e	**0x170**	5b fd ab 63
0x34	3a d1 2b ca	**0xd4**	d0 df 45 e6	**0x174**	b0 d7 14 06
0x38	2d 15 31 1a	**0xd8**	6b d5 1f 4c	**0x178**	93 12 4c 1f
0x3c	72 be bc b4	**0xdc**	00 a7 80 02	**0x17c**	8b 42 47 f3
0x40	f2 bc 4e 89	**0xe0**	80 03 00 41	**0x180**	ac 3e e7 89
0x44	58 19 a0 2d	**0xe4**	00 16 80 06	**0x184**	78 49 5d ba
0x48	7f f2 ab 03	**0xe8**	00 03 80 00	**0x188**	34 43 4d 8e
0x4c	1a 01 d2 e1	**0xec**	80 05 00 09	**0x18c**	c8 96 d8 40
0x50	84 01 4b 80	**0xf0**	89 2d c6 d5	**0x190**	56 78 69 ea
0x54	19 ba 67 3f	**0xf4**	b2 d3 53 58	**0x194**	90 37 16 89
0x58	78 82 75 95	**0xf8**	8a be 9b 09	**0x198**	f2 89 34 f1
0x5c	84 52 00 08	**0xfc**	e1 7a d9 12	**0x19c**	19 e4 26 16
0x60	30 b5 17 dd	**0x100**	22 9c 18 67	**0x1a0**	17 90 91 a0
0x64	f2 ba 51 0a	**0x104**	b5 37 23 8f	**0x1a4**	1d ad 95 de
0x68	bc 04 ed 19	**0x108**	b6 94 c6 d8	**0x1a8**	af 87 3d d7
0x6c	57 84 dd 3f	**0x10c**	cc f7 3f e8	**0x1ac**	fe 42 dd 41
0x70	5a 2a 3d 4e	**0x110**	13 ef 40 4e	**0x1b0**	52 d2 0b 3a
0x74	b9 a7 89 d8	**0x114**	f4 a1 2c 6a	**0x1b4**	44 06 28 55
0x78	6c d2 d8 65	**0x118**	ed 6e 6e 5c	**0x1b8**	31 f3 b4 ee
0x7c	3a b3 87 81	**0x11c**	ce a2 8b 8f	**0x1bc**	31 56 3d 23
0x80	00 10 00 00	**0x120**	58 ae fe b5	**0x1c0**	8c d1 19 c6
0x84	00 03 80 08	**0x124**	1f a3 0d 82	**0x1c4**	fa 69 8b 2f
0x88	80 f4 80 cc	**0x128**	1a af 41 78	**0x1c8**	75 16 cb f7
0x8c	00 a7 80 0a	**0x12c**	e5 9e cb 97	**0x1cc**	32 5a 6d 2c
0x90	80 04 00 41	**0x130**	fd cd ee f6	**0x1d0**	73 23 20 04
0x94	00 03 80 00	**0x134**	1d e0 0e 18	**0x1d4**	02 55 bb 8d
0x98	80 07 00 09	**0x138**	da 4c 36 a2	**0x1d8**	62 ee 09 20
0x9c	00 03 80 0c	**0x13c**	49 0c 20 9d	**0x1dc**	62 db d1 5c

The 2 bytes of each page table entry are stored in memory in big endian order, i.e., the lower address stores the higher-order byte. For example, the first page table entry at the address 0x88, which has content 0x80f4, has the following page table entry:

Valid	Readable	Writeable	Executable	Frame number
1	0	0	0	0x0f4

a. (2 mark) What is the size of each page in bytes?

b. (2 mark) What is the maximum size of the physical memory it can address in bytes?

c. (3 mark) Assume that the current process's page directory starts at physical address 0x80. List the entries of the page directory here.

Index	Valid	Page Frame Number
0		
1		
2		
3		
4		
5		
6		
7		
8		

9		
10		
11		
12		
13		
14		
15		

d. (6 mark) Translate each of the following virtual addresses from the current process as in (c) into a physical **address (not page frame number)**. The V/R/W/X bits below are the permission bits on the target physical page, not the 0s in the page directory.

Virtual address	Primary page table entry value	Secondary page table entry value	V?	R?	W?	X?	Physical address
0x808d							
0x2142							
0xc112							

Question 3 (6 marks): **OS161**

In OS161, there is a function **`md_forkentry()`** which you needed to implement in lab 3. What is the purpose of this function? List all the writes to the **`trapframe`** that you made in this function, and **describe the purpose of each**.

Question 4 (5 marks): **Scheduling**

Most of the OS schedulers prioritize interactive jobs. How can the OS tell if a process is interactive?

Question 5 (34 marks): **File System and its Synchronization**
In this question, we consider the problems you'll encounter when implementing a real file system. In addition to the inode, file, and directory data structures, the file system also needs the following:

- A super block that contains the metadata of the file system
- An inode map, or imap for short, which maps an inode number to the block number of the inode
- An inode bitmap, which uses 1 bit for each inode number to indicate whether that inode free or in-use
- A free block bitmap, which uses 1 bit for each block to indicate whether the block is free or in-use

The layout of the disk is as following:

Super block (1 block)	imap (a number of blocks)	inode bitmap (a number of blocks)	free block bitmap (a number of blocks)	blocks for inodes, directories, and files

Consider the following file system configurations:

- Block size is *B* Bytes
- The block number is 32-bit long.
- An inode number is also 32-bit long.
- This HDD has *N* available blocks in total
- Each inode occupies one block. The inode structure is the same as taught in the lecture: it has 15 block pointers, among them 12 are direct block pointers, then single, double, and triple indirect.
- Each directory **entry** is of *D* Bytes in size
- For each directory, it will have at least two entries: "." and ".."

Unless otherwise specified, assume the file system is a regular Unix file system instead of being log-structured.

Answer the following questions:

a. (1 marks) How many blocks should you allocate for the inode map and inode bitmap? Explain your answer.

b. (1 marks) How many blocks should you allocate for the free block bitmap?

c. (1 marks) Is there a limit on the total number of files (here assume a directory is also a file), measured in the number of blocks, in this file system on this HDD? If so, what is it? If not, why?

d. (1 marks) Is there a limit on the maximum size of a single file in this file system? If so, what is it? If not, why?

e. (1 marks) Is there a limit to the total number of files in a single directory in this file system, other than the limit in part c (i.e. assume there is no limit in part c)? If so, what is it? If not, why?

f. (2 marks for each subquestion) Assume this file system already has the following files and directories (and there are no other files/directories):
    a. /ece/344/cheat_sheet Occupies 1 block
    b. /covid/stay_home Occupies 2 blocks

Recall that you may want to update the following in a file system operation:

- Free block bitmap
- Inode map (one entry each time)

- Inode bitmap
- Inode (one field each time)
- Directory (one directory entry each time)
- Data block

You can assume that each update to the above is atomic (i.e. it's either done completely or as if it has not happened at all).

You also need to consider *crash consistency*. The system can crash at any given time, and after you reboot, your file system can be in an inconsistent state. For example, if the system crashes when you're creating a file "/a/b", your file system is inconsistent if, after the reboot, you see there is an entry named "b" in the directory "/a", but it points to an invalid inode or a file block that has not been allocated. Specifically, you could get the following possible errors:

A. Data block leak (data block is lost for any future use; occurs when the block is no longer used, but the free block bitmap still marks it as in use)
B. Inode leak (inode is lost for any future use; occurs when the inode is not used, but the inode bitmap still marks it as in use)
C. Inconsistent inode metadata (Some inode field does not match what is stored in data blocks)
D. Data corruption (file system data corrupted, including the above example that /a/b pointing to an unallocated file)

Assume that the only metadata store in the inode is (1) the size of the file, and (2) the last-modification-time of the file.

For each of the following operations, first list all the items that are needed to be updated **in order**, as well as which file/directory it belongs to; then indicate all the possible errors that may occur if a crash happens during the operation. If none, write so. For updates to the inode, specify which fields are updated; if there are multiple fields that are updated, separate them as separate updates. Carefully think about the order of operations, since the errors that could occur depend on the order.

E.g. Modify data of file <u>/covid/stay-home</u>

Need to update:

(1) Data block of stay-home
(2) Inode of stay-home (for updating last modified time)

Errors may occur:

C. (last modified time not updated if a crash happens between the two operations)

I. Append 1 byte to file /ece/344/cheat-sheet (no extra block needs to be allocated)

II. Append 1 block of data to file /covid/stay-home

III. Delete file /covid/stay_home

IV. Create an empty file with the name hacking under /covid/

V. Create an empty directory os161 under /ece/344/

VI. Rename directory /covid/ to /my-life/

g.(1 mark each subquestion) Now assume this file system is log structured. For the same operations, list the data structures that need to be updated **other than the ones you already listed above**. You don't need to consider the order or crash consistency issues.

VII. Append 1 byte to file /ece/344/cheat-sheet (no extra block needs to be allocated)

VIII. Append 1 block of data to file /covid/stay-home

IX. Delete file /covid/stay_home

X. Create an empty file with the name hacking under /covid/

XI. Create an empty directory os161 under /ece/344/

XII. Rename directory /covid/ to /my-life/

h. (2 marks) Now we consider data races on the file system. You notice that if multiple processes are accessing the bitmap (both inode and free data block) at the same time, a race condition may occur. Which file system error (or errors) given in part f may occur if such race condition happens? Explain.

i. (3 marks each subquestion) You are working on a solution to the problem above. You already know that lock could solve this problem. However, you learnt that locks could be slow. You noticed someone mentioned “lockless programming” on Heapoverflow. You decide to try it out. You found that x344 architecture provides the following atomic instructions (uint32_t is 32-bit unsigned integer):

- `uint32_t atomic_and_fetch(uint32_t *ptr, uint32_t val)`
- `uint32_t atomic_or_fetch(uint32_t *ptr, uint32_t val)`

In particular, **atomic_and_fetch** does the following **atomically** (note: it’s an instruction. The code below just shows its semantics.)

```
uint32_t atomic_and_fetch(uint32_t *ptr, uint32_t val){
    *ptr = *ptr & val;
    return *ptr;
}
```

atomic_or_fetch has a similar semantic, except it performs the “|” operation instead of "&".

Another version of these is also available:

- uint32_t **atomic_fetch_and**(uint32_t *ptr, uint32_t val)
- uint32_t **atomic_fetch_or**(uint32_t *ptr, uint32_t val)

In particular, **atomic_fetch_and** has the following semantic:

```
uint32_t atomic_fetch_and(uint32_t *ptr, uint32_t val){
    uint32_t ret = *ptr;
    *ptr = *ptr & val;
    return ret;
}
```

Can you write C code for the following operations **using the above atomic instructions** such that they fix the data races (properly synchronized), **without using any other synchronization primitives?** Assume the inputs of the functions are all valid (i.e. pointers are valid, location is within the boundary, etc.) The argument ***bitmap*** is always pointing to the beginning of the bitmap.

And assume the machine is 32-bit.

I. `void set_at(uint32_t * bitmap, uint32_t n) //set the n-th bit to 1, regardless of its previous value`

II. `void unset_at(uint32_t * bitmap, uint32_t n) //unset the n-th bit. I.e. change it to 0, regardless of its previous value`

lll. `uint32_t` **`allocate_one`**`(uint32_t * bitmap) //find the first 0 bit, change it to 1, and return the location of this bit.`