Operating Systems Final ECE344, Winter 2022

Duration: 2.5 hour, Examiner: D. Yuan

This is an closed book exam.

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.

Be brief and specific as possible. Most questions should only need 1 sentence to answer. **Time is tight.** Marks will be deducted for incorrect statements in an answer.

You'll get 20% of the mark for each question where you leave the answer empty.

*There are **17** total numbered pages, **9** Questions.*

Name: _____

Student Number: _____

	Total Marks	Marks Received
Question 1	6	Zhihao
Question 2	8	Zhihao
Question 3	8	Zhihao
Question 4	5	Rui
Question 5	12	Kia
Question 6	20	Rishi
Question 7	8	Rui
Question 8	35	Robin
Question 9	28	Jack
Total	130	

Question 1 (6 marks): Multiple Choices

(1) (1 mark) The Fast File System (FFS) uses the same logical file system layout as the Unix file system.

(a) True
(b) False

A

(2) (1 mark) The Log Structured File System uses the same logical file system layout as the Unix file system.

(a) True
(b) False

A

(3) (2 marks) Circle all of the correct choices:

(a) The OS must zero a page (i.e., set every bit to zero) when the page is freed
(b) The OS must zero a page when there is a page fault on it
(c) The OS must zero a page when it is swapped in
(d) The OS must zero a page when it is allocated
(e) The OS doesn't ever have to zero a page
(f) None of the above

D

(4) (2 marks) Which of the following are protected instructions

(a) Load
(b) Modifying the PC
(c) Modifying the stack pointer
(d) Reset the CPU timer
(e) None of the above

D

Question 2 (8 marks): Log structured file system

(a) List the pros/cons of a log structured file system design, when compared to a non-log structured Unix file system, on **hard drives**.

Pros:

(1) Great write performance, as it always groups small writes into a large sequential write
(2) Easy to checkpoint, as old version of data are not overwritten

Cons:

(3) Bad read performance if the read isn't cached
(4) Requires garbage collection, which can cause unpredictability to performance
(5) Requires a large buffer cache (won't be effective if the buffer cache is small)

Marking: Correct answering 4 out of 5 points receives full mark.
-1 for each wrong answer.

(b) List all the pros/cons of a log structured file system design, when compared to a non-log structured Unix file system, on **SSD**.

Pros:

(1) Group small writes into large write means it fits perfectly to the write/erase characteristics of SSD
(2) Evenly distributes writes over all blocks, prolong the life span.
(3) Same as (2) above.

Cons:
Same as (4) and (5) above.

Marking: (1) and (2) each worth 2 marks. -1 for each wrong answer. Don't double-penalize (i.e., if they provided a wrong answer in (a), don't penalize if they provide the same wrong answer here.

Question 3 (8 marks): Inode map

(a) (2 marks) What is the inode map (or imap)?

It maps a logical inode number to the physical location (block #).

(b) (3 marks) Discuss the pros/cons of using an imap on a Unix file system for hard drive

Pros:

(1) Can easily relocate an inode. E.g., if a block (that's storing an inode) is corrupted, then we can move the inode to a new block, update the location in the inode map, but without changing any directory contents that stores the inode #.

Cons:

(2) Performance. By adding a level of indirection, we slow things down. Also this imap takes space.

Marking: 1.5 each. -1 for each mistake.

(c) (3 marks) On a log structured file system, if the user modifies a file /a/b/c.txt, discuss which blocks will end up being modified, when the file system (1) has an imap, and (2) doesn't have an imap.

Without imap, any update to /a/b/c.txt will results in a cascading effect that result in

modifications, hence the invalidation, of all the directories (and their inode) that are predecessors of F (/, /a, /a/b, and their inodes).

If we have an imap, then only the block of c.txt and the inode of c.txt will be modified; other blocks remain untouched.

Marking: 1.5 each point (w/ and w/o imap)

Question 4 (5 marks): Page size
Explain the pros/cons of using a large page size versus a small page size.

Pros of large page:

- The same number of TLB entries would cover a larger amount of memory. So less TLB faults, better performance.
- The same number of page table entries would cover a larger amount of memory. So less page faults, better performance.

Cons:

- Internal fragmentation: more wasted space

Marking: 1.5 marks each answer. 0.5 bonus when they get all 3 correct. -1 for each wrong answer.

Question 5 (12 marks): Page Replacement

Suppose there are 4 pages of physical memory, and the memory is initially empty. The system then references virtual pages in the following sequence:

ACBDBAEFBFAGEFA

(a) Fill in the following table to show how the system would fault pages into the four frames if the system uses LRU replacement algorithm: indicate which access results in a page fault. (First two rows filled for you.)

Page Access	Fault/Hit	Page frames
A	F	A
C	F	AC
B		
D		
B		
A		
E		
F		
B		
F		
A		
G		
E		
F		
A		

	LRU		LRU Clock			Belady	
P age Access	**4F 2H 2F 3H 2F 2H Fault/Hit**	**Page frames**	**4F 2H 2F 2H 2F H F H Fault/Hit**	**Page frames (Bold: reference bit = 1,)**	**Clock hand**	**4F 2H 2F 3H F 3H Fault/Hit**	**Page frames**
A	F	A	F	A	A	F	A
C	F	CA	F	AC	A	F	AC
B	F	BCA	F	ACB	A	F	ACB
D	F	DBCA	F	ACBD	A	F	ACBD
B	H	BDCA	H	ACBD	A	H	ACBD
A	H	ABDC	H	ACBD	A	H	ACBD
E	F	EABD **(C)**	F	ECBD **(A)**	E	F	AEBD **(C)**
F	F	FEAB **(D)**	F	EFBD **(C)**	F	F	AEBF **(D)**
B	H	BFEA	H	EFBD	F	H	AEBF
F	H	FBEA	H	EFBD	F	H	AEBF
A	H	AFBE	F	EFBA **(D)**	A	H	AEBF
G	F	GAFB **(E)**	F	EGBA **(F)**	G	F	AEGF **(B)**
E	F	EGAF **(B)**	H	EGBA	G	H	AEGF
F	H	FEGA	F	EGFA **(B)**	F	H	AEGF
A	H	FEGA	H	EGFA	F	H	AEGF

Marking: 4 marks for each subquestion. Follow students' logic: don't cascade the penalization. -0.5 for each mistake.

(b) Show how the system would fault pages into the four frames if the system uses the LRU clock replacement algorithm, using the following table. Also show the location of the clock hand after each access. Assume initially the page frames are allocated in order (the first frame, then the second frame, and so on), and assume initially the clock hand points to the first page frame.

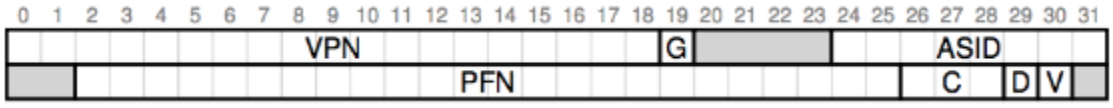
Page Access	Fault/Hit	Page frames	Clock Hand
A			
C			
B			
D			
B			
A			
E			
F			
B			
F			
A			
G			
E			
F			
A			

(c) Show how the system would fault pages into the four frames if the system uses the Belady (i.e., optimal) replacement algorithm. If there are multiple pages that aren't used in the future, evict the earliest one that was **brought into memory**.

Page Access	Fault/Hit	Page frames
A	F	A
C	F	AC
B		
D		
B		
A		
E		
F		
B		
F		
A		
G		
E		
F		
A		

Question 6 (20 marks, 2 marks each): TLB structure

Consider a computer system that has the TLB structure looking like this:



The VPN and PFN fields should be self-explanatory, as should the V field (valid); the ASID field (an address- space identifier field, also known as the PID field); and the D field (dirty); ignore any other fields.

On this system, the OS has a software-managed TLB. Thus, the OS is responsible for installing the correct translation when a TLB miss occurs. When finished with the update to the TLB, the OS returns from a trap, and the hardware retries the instruction.

Unfortunately, right after the OS updates the TLB, sometimes a bit in the entry that was just updated gets flipped and thus the wrong translation ends up in the TLB! (At most one bit could get flipped.) For each of the following fields, both (1) describe what the field is used for and (2) explain what you think would happen if a bit gets flipped in said field:

(a) VPN:

i. What is the VPN field for?

VPN: virtual page number, used as the "tag" of the TLB entry to compare if there is

match with the page number of the address.

ii. What would happen if a bit in the VPN got flipped?

2 possible results: it misses again after the OS retries the instruction. And if the flipped bit turns the VPN to another VPN that will be accessed in the future, it would result in accessing wrong memory address.

(b) PFN:
i. What is the PFN field for?

The target page frame number for the VPN.

ii. What would happen if a bit in the PFN got flipped?

Wrong memory address will be accessed. The process could even access other process's memory.

(c) ASID:
i. What is the ASID field for?

Tag the entry so that only ASID matches the PID, a translation will be used. Avoids TLB flush on context switch.

ii. What would happen if a bit in the ASID got flipped?

The retried instruction will result in a miss; another process's memory access could wrongly access this process's memory.

(d) Valid:
i. What is the Valid bit for?

Indicates whether the translation is valid.

ii. What would happen if the valid bit got flipped?

The retried instruction will result in a miss.

(e) Dirty:
i. What is the Dirty bit for?

Indicates whether the page is modified.

ii. What would happen if the dirty bit got flipped (this flip could occur anytime, not just when OS writes the TLB entry)?

It's used to decide whether a page needs to be copied to disk if being evicted. A bit flip could result in either not copying the page when it should (result in wrong data), or unnecessary disk writes.

Question 7 (8 marks): User and kernel-level threads

(a)(4 marks) What are the advantages and disadvantages of user threads compared to kernel threads?

Advantages of user threads:

- Efficiency: context switch/thread creation/exit happens in user level; no need to trap into the kernel

Disadvantage:

- OS not aware of user threads, so when one thread blocks (make a blocking system call), the entire process gets blocked.

2 marks each.

(b)(4 marks) Assume we can make system calls as fast as procedure calls using some new hardware mechanism (any other kernel-user mode switches are also fast). Would this make one kind of thread clearly preferable over the other? Explain briefly.

Answer: kernel threads would be preferable. the primary disadvantage of kernel threads is the overhead of trapping into kernel to manipulate them, context-switch, etc., and when that is no longer a problem, kernel threads won't have disadvantage compared with user-level threads and they have additional advantages.

Question 8 (35 marks): File system

In this question, we are going to unearth the data and metadata from a simple Unix file system. The file system has a fixed block size of 16 bytes and the disk has only 40 blocks overall. A picture of this disk and the contents of each block is shown below, where each cell represents 4 bytes, and the block ID of the block is at the bottom of each column.

9	0	f26	29	1	27	ece	0	344	1
-	0	24	30	33	0	12	0	10	6
-	0	48	31	0	0	app	0	454	0
-	0	26	32	0	0	4	0	18	0
Blk 0	Blk 1	Blk 2	Blk 3	Blk 4	Blk 5	Blk 6	Blk 7	Blk 8	Blk 9

1	ff26	1	aaaa	244	1	14	0	0	0
2	24	8	bbbb	21	22	0	0	0	0
0	0	16	cccc	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
Blk 10	Blk 11	Blk 12	Blk 13	Blk 14	Blk 15	Blk 16	Blk 17	Blk 18	Blk 19

2418	0	foo	deer	0	1	0	7	3	1111
1245	0	35	34	20	5	0	0	5	2222
eecc	0	dr	abc	3	8	0	0	1	3333
2022	0	1	32	5	0	0	0	0	4444
Blk 20	Blk 21	Blk 22	Blk 23	Blk 24	Blk 25	Blk 26	Blk 27	Blk 28	Blk 29

aaaa	iams	miam	to	ddda	1	0	raaa	0	ece3
bbbb	am.i	that	15	csee	11	0	beei	0	zkll
cccc	amsa	sami	0	abcd	0	0	3456	0	aabb
dddd	m.sa	am!t	0	fghi	0	0	aaaa	0	decc
Blk 30	Blk 31	Blk 32	Blk 33	Blk 34	Blk 35	Blk 36	Blk 37	Blk 38	Blk 39

The bytes in each block are ordered from low to high. For example, for block 3, the first 4 bytes have the value 29, the next 4 bytes 30, the next 4 bytes 31, and the last 4 bytes have the value 32.

The disk is formatted with a simple file system. This file system does not use inode map; instead, it directly stores the address (i.e. block ID) of the inode in the directory entry (as explained next).

The first block (Blk. 0) is a super block. The first 4 bytes is the block ID of the inode of the root directory, which is 9 in this case. The remaining 12 bytes are used for freemap, i.e., a bitmap that uses one bit for each block to indicate whether it's free. We omitted the value of the freemap in the picture above (shown as "-"); later you'll need to provide the value.

The format of an inode is also quite simple:

```
type: 0 means regular file, 1 means directory
direct pointer: the blk number of the first block of file (if there is one)
single-indirect pointer
double-indirect pointer
```

Assume that each of these fields takes up 4 bytes of a block. Each block pointer, whether it's direct, single/double indirect, stores the block ID (and uses 4 bytes). **If a block pointer has value 0**, it indicates the pointer is invalid (i.e., the block isn't allocated).

Finally, the format of a directory is also quite simple. It consists of a number of name/block pointer pairs:

```
name of file/subdirectory (4 bytes)
the block ID (not inode number) of the inode of the file/subdirectory (4 bytes)
```

Note that 0 is not a valid name of the file/subdirectory; we use it to mark the end of the directory.

Answer the following questions:

(a) (2 marks) What's the maximum file size on this file system, on a hard disk that has infinite number of blocks

1 (direct) + 4 (single-indirect) + 16 (double-indirect) = 21 blocks, or 336 bytes.

(b) (8 marks) Provide all the files and directories on this file system. You do not need to enumerate all the directory prefix. For example, say if a file system has the following files/directories "/", "/a/", "/a/b", "/a/b/c", you only need to write "/a/b/c", so that the first 3 directories are implied. **For each answer, specify whether it is a file or a directory.**

```
/ece/454 (file)
/ece/344/f26 (file)
/ece/344/48 (file)
/ece/244 (file)
/app/to/dr (file)
/app/to/foo/ff26 (file)
```

1 mark each. If they get the file name correct but the type wrong, -.5 for each. +2 for getting all correct.

(c) (2 marks) Which files or directories are empty?

```
/ece/454
/ece/344/48
/app/to/dr
```

/ece/244

0.5 mark each.

(d) (3 mark) Are there any (hard) links in this file system? If so, list all of them (and explain which paths are linking to the same file)

/app/to/foo/ff26 (file)
/ece/344/f26 (file)

(e) (4 marks) What's the value of the freemap? Fill in the table below. Each entry represents whether the block is free (1) or not (0). The number below each entry is the block ID that entry represents. The first entry has value 0, because blk 0 is not free (super block).

0																			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

Block 13, 17, 19, 23, 25, 28, 34, 36-39 are free, hence have value 1. Others are in-used (value 0).

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1	0	0	0	0	0	1	0	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

-.5 for each mistake.

(f) (2 mark) What operation/command would cause block 33's content to change to the following (this operation doesn't cause any other blocks to change)?

from
15
0
0

mv /app/to /app/from (changing the name of /app/to to /app/from)

Students don't have to know the exact command (mv).

(g) (2 mark) What operation/command will cause block 7's content to change to the following?

1
0
0
0

This is the (last) data block of /ece/344/f26 (or `/app/to/foo/ff26`). Modify the file's content on that byte.

For question (h) - (j), answer for each operation, which blocks are modified. Show the block contents after the modification. Assume:

- The operations are cumulative
- The file system is not log structured
- If we need to allocate a block, we always allocate the free block with the smallest block ID.

(h) **(4 mark)** The user creates an empty file named "file" under the root directory.

Block 9: change to "1 6 13 0" (allocate a new block 13 as the 2nd blk for / content)
Block 13: “17 0 0 0” indirect
Block 17 (now 2nd data block of / directory): change to "file 19 0 0"
Block 19 (inode of /file): change to "0 0 0 0"

Note: the free blocks can be used for different purposes, e.g., blk 9 could be used as inode of /file.

1 mark for each change.

(i) **(4 mark)** The user removes the file with the longest pathname on this file system, using command "rm XXX" (where XXX is the pathname). Recall that rm uses unlink.

rm `/app/to/foo/ff26`
(1) Blk 11 can be essentially freed
(2) Blk 35 (inode of /app/to/foo) needs to be changed to "1 0 0 0"
(3) Mark blk 11 as free in the freemap

Note, the following answer is also acceptable:
(1) Blk 11 changed to "0 0 0 0". In this case, we don't need to change blk 35, or mark blk 11 as free.

Also, the inode the ff26 (in blk 24) and the contents (blk 20, 29-32, 7) don't need to be changed, nor should they be added to the freemap, because there is still a file linking to it.

(j) **(4 mark)** The user then removes the next file with the longest pathname on this file system.

rm /ece/344/f26

(1) Change blk 2 to "48 26 0 0" (note, if students answer "0 0 48 26", partial 0.5 marks) 1 mark

(2) Mark blk 24 (inode of /ece/344/f26), 3, 5, 27 (indirect) and blk 20, 29-32, 7 (data of /ece/344/f26) as free in the freemap. 3 marks, 0.5 each

Question 9: Concurrency programming (28 marks):

Let's consider a concurrency programming problem that is a variant to the producer/consumer problem we discussed. Similar to that problem, we have:

- Two threads: one producer thread and one consumer thread.
- They share a buffer
- The producer produces resources and adds to the shared buffer

However, there are also some differences from what we discussed:

- The shared buffer is a circular buffer (also known as ring buffer). When the buffer becomes full, the producer doesn't stop producing, but simply "wraps around" by overwriting the oldest element (i.e., the beginning of the buffer array)
- The consumer thread is sleeping most of the time. It is only woken up occasionally by, say, a user or an external program.
- When the consumer is sleeping, the producer is always producing.
- But when the consumer is woken up, it will first stop the producer from producing.
- **Only after the producer is stopped**, the consumer will start to "consume" the items in the buffer, by printing out each item in the buffer from the newest to the oldest. It does NOT delete any items.
- After the printing completes, the consumer will resume the producer's execution, before it (i.e., the consumer) goes back to sleep.

This problem is commonly found in tracing tools, where the tracing thread will continuously emit trace points to record important runtime information of the system into the memory ring buffer, whereas a control-thread will occasionally persist the traces by writing it to the disk or send over the network, when, say, a failure is detected.

(a) (8 marks) Complete this program by **using semaphores**. We already provided the functional code below. Your goal is to complete it by adding semaphores. Write the simplest code possible. Unnecessary or convoluted code will be penalized.

```
Item buffer[SIZE];
int next = 0; // index to the next slot

void producer () {


  while (true) {


    Item item = produce();



    if (next == SIZE) {
       // wrap around
       next = 0;
    }
    buffer[next++] = item;


  }
}


void consumer() {

  while (true) {
    sleep(...);
    // Sleep returns, so we're woken up



    // Print the items from newest to oldest
    for (int i = next-1; i >= 0; i--)
      print (buffer[i]);

    for (int i = SIZE - 1; i >= next; i--)
      print (buffer[i]);



  }
```

```
}

int index = 0;

void producer () {
  while (true) {
    Item item = produce();
    P(sem);
    if (index == SIZE) {
       // wrap around
       index = 0;
    }
    buffer[index++] = item;
    V(sem);
}


void consumer() {

  while (true) {
    sleep(..);
    // We're waken up
    P(sem);
    for (int i = index-1; i >= 0; i--)
      print (buffer[i]);

    for (int i = SIZE - 1; i > index; i--)
      print (buffer[i]);
    V(sem);
  }
}
```

(b) (4 marks) We have learnt in class that using synchronization primitives, such as locks, semaphores, etc., can be slow. Why?

Because the underlying atomic instruction is quite slow: it requires memory fence.

(c) (16 marks) Now your friend Genius wants to rewrite this program using only regular global variables, instead of synchronization primitives (such as semaphore or locks), due to the performance overhead. Genius came up with a few versions and asked you to review the code.

Note that the underlying CPU and compiler could execute instructions out-of-order, i.e., they could reorder instructions as long as such reordering does not affect the correctness of the execution of a **single-thread program (also known as sequential**

program). In other words, if there is a sequential program P, and P' is a variance of P after the CPU/compiler reorders some of P's instructions, P and P' should produce the same output.

Specifically, the CPU/compiler can only reorder any two instructions, from the same function, that do not have a dependency. We say instruction I2 depends on I1, detonated as I1 -> I2, where I1 executes before I2 in the program order, if there is one of the following scenarios:

- Control dependency: I1 is a branch instruction (e.g., if, while, for, etc.), and whether I2 gets executed or not depends on the branch direction of I1.
- Data dependency: I2 uses a variable whose value is set by I1
- Reverse data dependency: I1 uses a variable whose value would be reset by I2 (so if we reorder I2 and I1, I1 would have used a different value).
- Transitive dependency: dependencies are transitive. If there exists another instruction I3, such that I1 -> I3, and I3 -> I2, then I1 -> I2.

Note that the CPU/compiler is not aware of multiple threads. Hence it would not consider any implications of data dependencies between two threads (say one thread reads and another writes to a shared global variable) when considering whether it can reorder instructions (within the same thread). This is true even if the global variable is declared as `volatile`.

Also, assume:

- The compiler/CPU won't remove any code that it deems unnecessary from your program (i.e., the compiler won't perform dead code elimination optimization).
- Concurrent threads run on different cores in parallel
- When one thread modifies a global variable shared by another thread, the new value will eventually be visible to the other thread, but the delay could be arbitrarily long.

To simplify the problem, the consumer doesn't need to worry about re-enabling the producer at the end. It only needs to properly stop the producer before printing the items after the producer is stopped. (The producer will be properly re-enabled by the user or an external program.)

(i) (8 marks) For the following code snippet from Genius, is it correct? Why?

```
1 int next = 0;
2 int flag = 1;
3
4 void producer () {
5   while (true) {
6     if (flag == 1) {
7       Item item = produce();
8
9       if (next == SIZE) {
10         // wrap around
11         next = 0;
12      }
13      buffer[next++] = item;
14    } else {
15      flag = 2;
16    }
17  } // while
18 }
19
20
21 void consumer() {
22   while (true) {
23     sleep(..);
24     // We're woken up
25     flag = 0;
26     while (flag != 2)
27        ;
28
29     // Producer is stopped
30     for (int i = next-1; i >= 0; i--)
31       print (buffer[i]);
32
33     for (int i = SIZE - 1; i >= next; i--)
34       print (buffer[i]);
35
36     // User or external programs can now re-enable the producer.
37   }
38 }
```

No. The compiler/CPU could reorder line 30-34 to before line 25, as there isn't any data dependency between the code blocks of L25-27 and L30-34.

(ii) (8 marks) For the following code snippet from Genius, is it correct? If not, why?

```
int next = 0;
int start = 1;
int stop = 0;

void producer () {
  while (true) {
    if (start) {
      Item item = produce();

      if (next == SIZE) {
         // wrap around
         nextx = 0;
      }
      buffer[next++] = item;
      if (stop) {
        stop = next;
        start = 0;
      }
    } // if (start)
  } // while
}


void consumer() {

  while (true) {
    sleep(..);
    // We're woken up
    stop = 0;
    while (stop == 0)
       ;

    // Producer is stopped
    for (int i = stop - 1; i >= 0; i--)
      print (buffer[i]);

    for (int i = SIZE - 1; i >= next; i--)
      print (buffer[i]);

    // User or external programs can now re-enable the producer.
  }
}
```

No. Line 14 can be broken down to the following instructions:

I1 old_next = next
I2 buffer[old_next] = item
I3 next = old_next + 1

The CPU/compiler can move I2 to bre after L18, hence the consumer could start printing while the producer is still writing to buffer.