CS502 - Operating Systems
2022 Spring Final Exam
Maximum Points: 140.

High-level Instructions

Time Limit: 1 hours 20 mins. The exam will start at 4PM. Canvas site will close at 5:20PM. Please plan ahead to submit.

- There are seven total numbered pages, with 12 questions.
- By design, **there are more questions than can be realistically answered during the exam time. So please try to use time wisely.**
- Please read all questions carefully and follow their respective instructions when answering!
- Please use the provided answer sheet (**2022S_final_worksheet.txt**) to answer the following questions. It is recommended that you do the work **locally** in your computer and save periodically.
- Please submit the worksheet via the Canvas Final Exam Page.
- Open-book policy: **You are free to reference any resources you can find to solve the questions.** However, during the exam, you **must not** engage in any form of communication with other people, regardless of whether they are currently enrolled in this class. Any questions and concerns should be directed to Prof. Guo.
- Good luck and have fun!

Final Exam: Questions

Question 1: Locks

(6 Points) Assume this attempted implementation of a lock:

```c
void init(lock_t *mutex) {
  mutex->flag = 0; // 0 -> lock is available, 1 -> held
}
void lock(lock_t *mutex) {
  while (mutex->flag == 1) ;
  mutex->flag = 1;
}
void unlock(lock_t *mutex)
  mutex->flag = 0;
}
```

Assume 4 threads are competing for this lock. How many threads can possibly acquire the lock? **Explain briefly.**

1-4. There are two possible interpretations of this question. One is "how many threads can acquire the lock at the same time"; the other is "how many threads can acquire the lock at all". Fortunately, answer is the same in this case, because the lock is broken; **it doesn't use an atomic exchange**, and thus has a race condition in its very definition. Multiple threads, calling lock() at the same time, may acquire the lock at the same time! Thus, all five answers are possible.

Grading Notes: 2 points: by answering any number (1, 4]. 4 points: by pointing out the lock implemenation is broken/race condition (2 points); and atomic exchange or hardware instructions (2 points).

Question 2: Semaphore

(10 Points) A Semaphore is a useful synchronization primitive. Which of the following statements are true of semaphores? To answer, first write down **True or False**, then **Explain briefly.**

(a) Each semaphore has an integer value.

True. By definition, each semaphore has a value.

(b) If a semaphore is initialized to 1, it can be used as a lock.

True This is called a binary semaphore.

(c) Semaphores can be initialized to values higher than 1.

True This is useful in some cases as described in the book.

(d) A single lock and condition variable can be used in tandem to implement a semaphore.

True This is true, along with a state variable to track the value of the semaphore.

(e) Calling `sem_post()` may block, depending on the current value of the semaphore

False Only `sem_wait()` blocks, `sem_post()` just does its work and returns.

Grading Notes: 1 point: answer true/false correctly; 1 point for the reason.

Question 3: Critical Section

(12 Points) Assume the following multi-threaded program,

```c
void worker(void *arg) {
  int balance = 0;
  balance = balance + 100;
  printf("balance %d\n", balance);
}

int main(int argc, char *argv[]){

thread_t p1, p2;
thread_create(&p1, worker, NULL);
thread_create(&p2, worker, NULL);
thread_join(p1);
thread_join(p2);

}
```

(a) When this program runs, how many total threads can there be at a given moment in time? **Explain briefly.**

1, 2, or 3; parent thread, then parent + p1, parent+p2, or parent+p1+p2

(b) When this program runs, what value will be printed by the `printf` statement? **Explain briefly.**

100; b/c balance is a local variable

(c) Now consider we make a slight modification (the new code is shown below), when the program runs, what is the final value of balance? **Explain briefly.**

```c
int balance = 0; // global variable

void worker(void *arg) {
  balance = balance + 100;
  printf("balance %d\n", balance);
}

int main(int argc, char *argv[]){

thread_t p1, p2;
```

```
  thread_create(&p1, worker, NULL);
  thread_join(p1); // flipped the order
  thread_create(&p2, worker, NULL);
  thread_join(p2);

  }
```

200; b/c there is no concurrency.

Grading Notes: answer: 2 points; explain: 2 points

Question 4: Deadlock

(9 Points) Deadlock is a classic problem that arises in many concurrent systems with complex locking protocols.

(a) Consider the following code and more than two threads call `foo()`, can this code lead to deadlock? **Explain briefly.**

```
foo() {
  pthread_mutex_lock(&lock1);
  pthread_mutex_lock(&lock2);
  bar();
  pthread_mutex_unlock(&lock2);
  pthread_mutex_unlock(&lock1);
}

bar(){
  pthread_mutex_unlock(&lock1);
  // do some work
  pthread_mutex_lock(&lock1);
}
```

Yes, it can deadlock. One scenario: thread one acquires lock1, and then lock2, then go into bar(), and then release lock1; thread two comes in and grabs lock1; at this point, we have a cycle wait where thread one will wait for lock1 and thread two will wait for lock2.

Grading Notes: answer: 2 points; explaination: 3 points

(b) One way to avoid deadlock is to schedule threads carefully. Assume the following characteristics of threads T1, T2, and T3:

- T1 (at some point) acquires and releases locks L1, L2
- T2 (at some point) acquires and releases locks L1, L3
- T3 (at some point) acquires and releases locks L3, L1, and L4

For which schedule(s) below is deadlock possible? To answer, first write down **the corresponding number(s)**, then **Explain briefly.**

1. T1 and T2 run concurrently to completion, then T3 runs
2. T1, T2, and T3 run concurrently
3. T1 and T3 run concurrently to completion, then T2 runs

Only (2) is deadlock possible. Because only T2 and T3 can deadlock, because they each grab two locks (L1 and L3) but in different orders. So as long as T2 and T3 do not run at the same time!

Grading Notes: answer: 2 points; explaination: 2 points

Question 5: What Prints?

(8 Points) Assume the following multi-threaded code:

```

void *printer(void *arg) {
  char *p = (char *) arg;
  printf("%c", *p);
  return NULL;
}
int main(int argc, char *argv[]) {
  pthread_t p[3];
  for (int i = 0; i < 3; i++) {
  char *c = malloc(sizeof(char));
  *c = 'h' + i; // hint: 'a' + 1 = 'b', etc.
  pthread_create(&p[i], NULL, printer, (void *) c);
}
  for (int i = 0; i < 3; i++)
    pthread_join(p[i], NULL);

    return 0;
}
```

Assuming calls to all library routines succeed, which of the following outputs are possible? To answer, first write down **the corresponding number(s)**, then **Explain briefly.**

1. hij possible; the explaination is smiliar: each thread will be handed a unique memory address contains the initalized character.
2. jih possible
3. jjj not possible
4. iii not possible
5. hhh not possible

Grading Notes: possible/not possible: 1 points; explaination: 3 points

Question 6: Queue-based Lock

(12 Points) Here is a queue-based lock. Assuming a maximum of 3 threads in the system, and further assuming the lock is used "properly" (i.e., threads acquire and release it as expected). What values of m→flag, m→guard, and queue state are possible (at the same time)? To answer, first write down **Possible or Not Possible**, then **Explain briefly.**

```
  typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
 }

 void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        setpark();
        m->guard = 0;
        park();
    }
 }
```

```c
26
27 void unlock(lock_t *m) {
28    while (TestAndSet(&m->guard, 1) == 1)
29        ; //acquire guard lock by spinning
30    if (queue_empty(m->q))
31        m->flag = 0; // let go of lock; no one wants it
32    else
33        unpark(queue_remove(m->q)); // hold lock
34        // (for next thread!)
35    m->guard = 0;
36 }
```

Grading Notes: Possible/Not Possible: 1 point; Explain: 2 points

(a) flag =0, guard=0, queue is empty

Possible. Before any thread does anything.

(b) flag =0, guard=0, queue is not empty

Not Possible. If queue is not empty, that means some other thread is currently holding the flag, i.e., flag=1

(c) flag =1, guard=1, queue is empty

Possible. After one thread calls lock() and before executing line 18.

(d) flag =1, guard=1, queue is not empty

Possible. After the first thread calls lock(), flag =1; the second thread calls lock() and gets added to the queue; and the third thread calls lock() and before executing line 22.

Question 7: Multi-level page table

(12 Points) Assume you have a 15-bit virtual address space, with page size of 32 bytes. Assume further a two-level page table, with a page directory which points to pieces of the page table. The format of both the page directory entry (PDE) and the page table entry (PTE) is the same: a valid bit followed by a 7-bit page frame number. The page directory base register is set to 50 (decimal). The following physical page contents are made available to you:

```
page 8:   16 0e 14 07 07 01 0c 15 03 05 0c 00 19 05 1c 11
          09 02 13 01 0a 1e 19 16 12 13 17 1b 03 1b 1e 12

page 27:  7f 7f 7f 7f 7f 7f 7f 7f 7f f0 7f a4 7f 7f 7f 7f
          7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f

page 50:  7f c0 ea f9 ed 8b db ba d6 c1 84 8a b3 7f da eb
          9a 85 ab 87 e5 97 b1 df 86 ec e7 ad f2 b9 d5 f8

page 86:  13 1b 03 11 1e 12 16 18 0f 08 12 10 0a 1a 0b 0e
          17 19 88 14 07 1a 1c 16 17 0f 0f 12 04 14 1a 05

page 90:  16 0e 14 07 07 01 0c 11 03 05 0c 00 19 05 1c 11
          09 02 13 01 0a 1e 19 16 12 13 17 1b 03 1b 1e 12

page 96:  16 0d 18 10 02 0e 01 1c 1d 0a 09 17 06 05 05 0a
          13 1d 06 1d 11 1b 19 04 14 03 03 0c 17 11 05 1a
```

In translating virtual address 0x2247, which physical pages are accessed? If the address is valid, what final value do you get back? **Show your work.**

pages 50, 86, 8 in order. the final value is 0x15.

steps: This is exactly the same question as the midterm one, except with different physical page contents and the virtual address. 0. 0x2247 in 0b 01000 10010 00111 1. We know we have to access page 50 to get the PDE, and since the top 5 bits translate to 8; that means we get the 8th (counting from 0, 1th, 2nd etc) bytes which is 0xd6. 2. 0xd6 = 0b 1101 0110, and given the PDE format, we know it is valid, and then it points to 0b 101 0110 = 86; so we will access page 86 to get the PTE, and since the next 5 bits translate to 18th, that means we get the 18th (counting from 0, 1th, 2nd etc) bytes which is 0x88. 3. 0x88 = 0b 1000 1000, again it is valid, and it points to page 8 because 0b 000 1000 = 8; so we will access page 8 (the actual data page) and since the last 5 bits translate to 7th, that means we get the 7th (counting from 0, 1th, 2nd etc) bytes which is 0x15.

Grading Notes: physical pages: 6 points (has to in order; each missing physical page deducts 2 points; each incorrect physical page deducts 2 points;)

Steps: 0-4 step each step 1.5 points.

Question 8: Files

(14 Points) For this question, assume a simple disk model where each disk read of a block takes D time units. Also assume the basic layout is quite like the very simple file system. For each question, first write down the answer, then **Show your work.**

(a) **(3 Points)** Assume that all data and metadata begin on disk. Assume further that all inodes are in separate blocks, and that each directory is only one block in size. How long does it take to **read** the file `/a/b/c/d.txt`? Assume the file `d.txt` is two blocks in size and assume after opening the file, we read the file in its entirety.

11D. read root inode, root data block, read a, b, and c, for inode and data block; and finally read the inode for d.txt; and then the 2 data blocks.

(b) **(3 Points)** Now assume a different file `/a/b/c/big.txt` which contains 1024 data blocks. The inode itself stores 10 direct pointers and 2 indirect pointers. Disk addresses are 4 bytes long, and disk blocks are 4KB in size. **After opening it**, how long does it take to read the entire file?

1025D = 1024+1; Each data block can store 1024 addresses; so the file's pointers are stored in one data block. To read the file, it will have to read all data blocks plus the ONE data block for the direct pointer.

(c) **(4 Points)** Now assume a new inode structure is introduced, in which there is only one pointer: a double indirect pointer, which points to the double indirect block, which can point to 1024 indirect blocks, each of which can point to 1024 blocks. After opening the file, how long does it take to sequentially read 40 blocks within a very large file?

42D-43D.

In order to get to each block, we will need to first read the double indirect block (D); then we need to at least get one indirect block and at most two indrect blocks (1D-2D) as each indrect block holds 1024 pointers; last we will read the 40 blocks (40D).

(d) **(4 Points)** How long does it take to read 40 random blocks within a very large file?

81D = 1 + 2x40.

Since it is random, so it is probably going to be in 50 different bocks. In order to get to each block, we will need to first read the double indirect block (D), then for each read, we need to first get read the indirect block and then the data block (2D).

Question 9: RAIDs

(18 Points) In RAIDs, some I/Os can happen in parallel, whereas some happen in sequence. To indicate two I/Os (to blocks 0 and 1, for example) in a flow can happen at the same time, we write: "(0 1)". To indicate two I/Os must happen in sequence (i.e., one before the other), we would use this notation: "0, 1". These flows can be built into larger chains; for example, consider the sequence "(0 1), (2 3)", which would indicate I/Os to blocks 0 and 1 could be issued in parallel, followed by I/Os to 2 and 3 in parallel. We can also indicate read and write operations in a flow with "r" and "w". Thus, "(r0 r1), (w2 w3)" is used to indicate we are reading blocks 0 and 1 in parallel, and, when that is finished, writing blocks 2 and 3 in parallel. For each question, first write down the answer, then **Explain briefly.**

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 1: RAID-4 diagram.

(a) **(3 Points)** First, let us assume a simple disk model where each read and write takes D time units. Assume we have the RAID-4 with one parity block as shown in Figure 1.

Assume we must read blocks 0, 5, 10, and 15 as fast as we can. What is the I/O flow of these blocks? How long does it take to complete these reads?

(r0 r5 r10 r15); basically read in parallel. (A slightly longer than) D.

(b) **(3 Points)** Now assume we must read blocks 0, 4, 8, and 12. What is the I/O flow of these blocks? How long does it take to complete these reads?

r0, r4, r8, r12; basically read in sequence. 4D.

(c) **(4 Points)** Now assume we have a better disk model where it takes S time units to perform a random seek and R time units to perform a full rotation; assume data transfer is free.

Assume we must write block 5 and we are using subtractive parity. What is the I/O flow? How long does it take to this logical write?

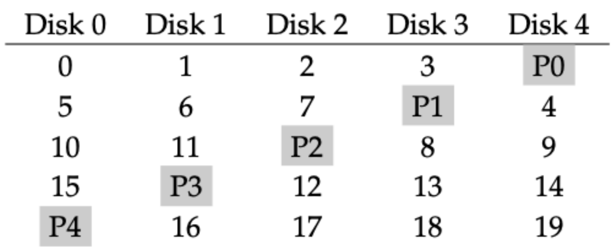
(r5 rP1), (w5, wP1); first two reads, followed by two writes. S + 3R/2; the first read to Disk 1 will take (S + R/2) to read block 5; and then R to write block 5 **because there is no need to seek, only to rotate back to the start of block 5**. This analysis holds for Disk 4 which holds the parity block. Assume the parity calculation takes negligble time.

(d) **(4 Points)** Now assume we must write block 5 and block 11 **as fast as we can** and we are using subtractive parity. What is the I/O flow? How long does it take to these two logical writes?

(r5 rp1 r11), rp2, (w5 wp1 w11), wp2; bascially rp1, rp2, wp1, wp2 are serialized because they need to be performed on the same disk 4. **read two parity blocks first for performance; if we rp1,wp1, we will have to incur an additional rotation.** S+3R/2. First read to disk 4 takes S+R/2, second read to disk 4 takes 0 (transfer is free), the first write takes R, the second write takes 0; assume P1, P2 are sequential read/write.

Grading Notes: The parallelized requests can also be (r5 rp2 r11), the key here is that rp1, rp2, wp1, wp2 are serialized because they need to be performed on the same disk 4.

(e) **(4 Points)** Lastly assume that we changed from a RAID-4 to a RAID-5 system with rotating parity as shown in Figure 2. We must write block 5 and block 11 **as fast as we can** and we are using subtractive parity. What is the I/O flow? How long does it take to these two logical writes?



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 2: RAID-5 diagram.

(r5 rp1 r11 rp2), (w5 wp1 w11 wp2); basically each disk will have one read and one write. So the performance will be S +3R/2 as analyzed similary in (c).

Question 10: Large Files!

(10 Points) Most file systems support pretty large files. In this question, we'll see how big of a file various file formats can support. Assume, for all questions below, that file system blocks are **4KB**. For each question, first write down the answer, then **Explain briefly.**

Grading Notes: 2 points, 4 points, and 4 points each; answers are 2 points, explainations are 2 points.

(a) Assume you have a really simple file system **directFS**, where each inode only has 12 direct pointers, each of which can point to a single data block. Direct pointers are 64 bits in size (8 bytes). What is maximum file size for the **directFS**?

12 * 4KB

(b) Now, assume that a new file system called **indirectFS** that uses direct pointers but also adds indirect pointers and double-indirect pointers. Specifically, an inode within **indirectFS** has 2 direct pointers, 1 indirect pointer, and 1 double-indirect pointer. Pointers, as before, are 8 bytes in size. What is the maximum file size for the **indirectFS**?

number of pointers per block: 4KB/8 = 512

number of identifiable blocks = 512*512 + 512 + 2

number of identifiable blocks * 4KB

(c) Lastly, we have a compact file system, called (you guessed!) **compactFS**, that tries to save as much space as possible within the inode. Thus, to point to files, it stores only a **single pointer** to the first block of the file. However, blocks within **compactFS** store 4KB of user data and a next field (much like a linked list), and thus can point to a subsequent block (or NULL, indicating there is no more data). What is the maximum file size for the **compactFS**?

2^{32} many uniquely identified blocks; 2^{32} x 4KB

Question 11: Journaling

(13 Points) We now turn our attention to journaling file systems, such as Linux ext3. Such file systems use a small "journal" (or "write-ahead log") to record information about pending file system updates before committing said updates, in order to be able to recover quickly from a crash. Assume the standard structures of a file system here: an inode bitmap, a data bitmap, a table of inodes, and data blocks. Assume, for all questions below, that file system blocks are **4KB**.

(a) First, assume the journaling is disabled. A process creates a 2KB file in the root directory (which does not have many entries in it, so there is room for another entry in an existing directory data block). What blocks are written to disk during the creation?

inode bitmap, data bitmap, inode for file, inode for the root, root data block, and the actual 2KB data.

Grading Notes: 0.5 point each; wrong block deduct 1 point; maximum deduction 3.

(b) Now assume data journaling mode, in which all blocks (metadata and data) are first journaled before being updated in place. What exact sequence of writes takes place to the underlying storage device during the file creation described above?

step 1 Journal write: TXB, inode bitmap, data bitmap, inode for file, inode for the root, root data block, and the actual 2KB data;

step 2 Journal commit: TXE

step 3 Checkpoint: Then to the disk proper for all the blocks except TXB and TXE. Free (mark the transaction free in the journal superblock).

Grading Notes: 1 point per step; wrong step order deduct 1 point; maximum deduction 3.

(c) Next assume metadata journaling mode, which only writes metadata to the journal (user file data is written only once as a result). What exact sequence of writes takes place to the underlying storage device during the file creation described above?

The 2KB data is checkpointed directly; Journal write: TXB, inode bitmap, data bitmap, inode for file, inode for the root, root data block;

step 3 Journal commit: wait for the above two steps finished before writing TXE

step 4 Checkpoint: afterwards, write to the disk proper for all the metadata from the journal write step.

Free (mark the transaction free in the journal superblock).

Grading Notes: 1 point per step; wrong step order deduct 1 point; maximum deduction 4.

(d) Assume that a process appends a data block to an existing (small) file. What are the blocks that needed to the journal assuming the data journaling mode as part of this update? What about using the metadata journaling mode?

3 blocks; data journaling: file inode, new data block, data bitmap;

2 blocks metadata journaling: file inode, data bitmap;

Grading Notes: 1.5 points each; wrong block deduct 0.5.

Question 12: TLBs

(16 Points) TLB, a translation-lookaside buffer, is a critical mechanism in supporting memory virtualization.

(a) **(6 Points)** Consider this code snippet. When this code is first run, how many TLB misses will take place? Assume an integer is 4 bytes, and a page size is 1KB. **Explain briefly.**

```c
int i;
int p[512];
for (i = 0; i< 512; i++)
  p[i] = 0;
```

Similar to the Figure 19.2 example in the OSTEP book.

(4 points) First, we need to figure out the number of pages the array p will take up. 4 * 512 = 2KB; so that will either be 2 pages or 3 pages depending on wether the array alignment.

Also, we have the code page (assume 1 page)

So it will incur 3 to 4 times of TLB misses.

Finally, some true or false questions! Which of the following statements are true about TLBs and Multi-level Page Tables? To answer, first write down **True or False**, then **Explain briefly.**

(b) The main reason to have a hardware TLB is to speed up address translation.

True. That is the point of the TLB, to cache address translations and hence speed up the entire process.

(c) Using a multi-level page table increases TLB hit time.

False. TLB hit time is not affected by page-table structure.

(d) The main reason to have a multi-level page table is to save memory space.

True. Yes, compared to linear page table, multi-level page table requires less space to store the virtual-physical page mappings.

(e) A hardware TLB is more flexible than a software-managed one because the former can use any data structure for page data.

False. It is the other way around.

(f) Just like the page table, a hardware TLB entry does not need to contain the VPN.

False. A TLB entry at least needs to contain VPN, PFN, and other bits.