



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:

```
1  void init(lock_t *mutex) {  
2      mutex->flag = 0; // 0 -> lock is available, 1 -> held  
3  }  
4  void lock(lock_t *mutex) {  
5      while (mutex->flag == 1) ;  
6      mutex->flag = 1;  
7  }  
8  void unlock(lock_t *mutex)  
9      mutex->flag = 0;  
10 }
```

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

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.

- (b) If a semaphore is initialized to 1, it can be used as a lock.
- (c) Semaphores can be initialized to values higher than 1.
- (d) A single lock and condition variable can be used in tandem to implement a semaphore.
- (e) Calling `sem_post()` may block, depending on the current value of the semaphore

Question 3: Critical Section

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

```

1  void worker(void *arg) {
2      int balance = 0;
3      balance = balance + 100;
4      printf("balance %d\n", balance);
5  }
6
7  int main(int argc, char *argv[]){
8
9      thread_t p1, p2;
10     thread_create(&p1, worker, NULL);
11     thread_create(&p2, worker, NULL);
12     thread_join(p1);
13     thread_join(p2);
14
15 }
```

- (a) When this program runs, how many total threads can there be at a given moment in time? **Explain briefly.**
- (b) When this program runs, what value will be printed by the `printf` statement? **Explain briefly.**
- (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.**

```

1  int balance = 0; // global variable
2
3  void worker(void *arg) {
4      balance = balance + 100;
5      printf("balance %d\n", balance);
6  }
7
8  int main(int argc, char *argv[]){
9
10     thread_t p1, p2;
11     thread_create(&p1, worker, NULL);
12     thread_join(p1); // flipped the order
13     thread_create(&p2, worker, NULL);
14     thread_join(p2);
15
16 }
```

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.**

```

1  foo() {
2      pthread_mutex_lock(&lock1);
3      pthread_mutex_lock(&lock2);
4      bar();
5  }
```

```

5  pthread_mutex_unlock(&lock2);
6  pthread_mutex_unlock(&lock1);
7  }
8
9  bar(){
10 pthread_mutex_unlock(&lock1);
11 // do some work
12 pthread_mutex_lock(&lock1);
13 }

```

(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

Question 5: What Prints?

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

```

1
2 void *printer(void *arg) {
3     char *p = (char *) arg;
4     printf("%c", *p);
5     return NULL;
6 }
7 int main(int argc, char *argv[]) {
8     pthread_t p[3];
9     for (int i = 0; i < 3; i++) {
10        char *c = malloc(sizeof(char));
11        *c = 'h' + i; // hint: 'a' + 1 = 'b', etc.
12        pthread_create(&p[i], NULL, printer, (void *) c);
13    }
14    for (int i = 0; i < 3; i++)
15        pthread_join(p[i], NULL);
16
17    return 0;
18 }

```

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
2. jih
3. jjj
4. iii
5. hhh

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 $m \rightarrow \text{flag}$, $m \rightarrow \text{guard}$, and queue state are possible (at the same time)? To answer, first write down **Possible or Not Possible**, then **Explain briefly**.

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, getpid());
21         setpark();
22         m->guard = 0;
23         park();
24     }
25 }
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 }
```

- (a) flag =0, guard=0, queue is empty
- (b) flag =0, guard=0, queue is not empty
- (c) flag =1, guard=1, queue is empty
- (d) flag =1, guard=1, queue is not empty

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 |
| 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.**

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.

(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?

(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?

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

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?

(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?

(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?

(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?

(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.

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**.

(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**?

(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**?

(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**?

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?

(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?

(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?

(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?

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.**

```
1  int i;
2  int p[512];
3  for (i = 0; i < 512; i++)
4      p[i] = 0;
```

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.
- (c) Using a multi-level page table increases TLB hit time.
- (d) The main reason to have a multi-level page table is to save memory space.
- (e) A hardware TLB is more flexible than a software-managed one because the former can use any data structure for page data.
- (f) Just like the page table, a hardware TLB entry does not need to contain the VPN.