# CS 537: Introduction to Operating Systems
# Spring 2024: Final Exam

This exam is closed book, closed notes.

No calculators may be used.  All cell phones must be turned off and put away.

You have 2 hours to complete this exam.

Write all of your answers on the accu-scan form with a #2 pencil:

- CANVAS LAST NAME - fill in your last (family) name starting at the leftmost column
- CANVAS FIRST NAME - fill in the first five letters of your first (given) name
- IDENTIFICATION NUMBER - This is your UW Student WisCard ID number
- ABC of SPECIAL CODES - Write your lecture number as a three digit value:
    - 001 (Afternoon Lecture)
    - 002 (Morning Lecture)
- **D of SPECIAL CODES - Write the value 1 for this version of the exam.**

These exam questions must be returned at the end of the exam, but we will not grade anything in this booklet.

You may separate the pages of this exam if it helps you.

Unless stated (or implied) otherwise, you should make the following assumptions:

- The OS manages a single uniprocessor (single core)
- All memory is byte addressable
- The terminology lg means $\log_2$
- $2^{10}$ bytes = 1KB
- $2^{20}$ bytes = 1MB
- Data is allocated with optimal alignment, starting at the beginning of a page
- Assume leading zeros can be removed from numbers (e.g., 0x06 == 0x6).
- Hex numbers are represented with a proceeding "0x"

This exam has 75 questions.  Each question has the same number of points.

Good luck!

# Part 1. Virtualization
## True / False

1. If a PTE is invalid, accessing the corresponding virtual address will trigger a segmentation fault.
**True.**
2. A preemptive scheduler relies on timer interrupts to force processes to yield the CPU.
**True.**
3. Processes running the same program can share memory for the code using virtual memory.
**True.**
4. Since STCF is the ideal scheduler, it is the basis for modern operating system schedulers.
**False, the completion time of processes is unknown and STCF is not implementable in general.**
5. The cost of a context switch is proportional to the number of page table entries in the new process, since they have to be restored.
**False, only the root of the page table hierarchy needs to be restored**

## Multiple Choice

**Address Translation** -- You have a system that uses a linear page table for virtual to physical address translation. The system has the following parameters:

Address Space size: 16 KB
Physical Memory size: 64KB
Page size: 4KB

The format of the page table has the high-order (left-most) bit as the VALID bit. If the bit is 1 then the rest of the entry is the PFN. If the bit is 0 then the page is not valid. The current page table (from entry 0 down to the last entry) is:

0x80000000
0x00000000
0x00000000
0x8000000a

For each virtual address, indicate the physical address it translates to or that it is an invalid address.

6. Virtual Address 0x00001b00 (decimal: 6912)
A. 0x00000b00 (decimal: 2816)
B. 0x00000000 (decimal: 0)
C. 0x0000b00a (decimal: 45066)
D. 0x0000ab00 (decimal: 43776)
E. Invalid Address
**E, Invalid address -- The VPN is 1 and the Valid bit in the second entry of the page table is 0 (not valid).**

7. Virtual Address 0x00003919 (decimal: 14617)
A. 0x00000919 (decimal: 2329)
B. 0x00000519 (decimal: 1305)
C. 0x00003919 (decimal: 14617)
D. 0x0000a919 (decimal: 43289)
E. Invalid Address
**D, 0xa919 -- The VPN is 3 indicating the last entry in the page table. Combining with the offset (0x919) yields 0xa919.**

**CPU Job Scheduling** -- A computer has the following workload. If needed, assume a time-slice of 1 second and **if there is a choice the newest arriving job is selected.**

| Job Name | Arrival Time (seconds) | CPU Burst Time (seconds) |
|---|---|---|
| A | 0 | 8 |
| B | 4 | 4 |
| C | 7 | 6 |

8. Given a FIFO scheduler, what is the turnaround time of job B?
A. 4
B. 6
C. 7
D. 8
E. None of the above
**D, Job B arrived at time 4 and completes running at time 12.**
9. Given a RR scheduler, what is the average response time?
A. 0
B. 2
C. 6
D. 7
E. None of the above
**A, The response times for each job is 0 since the instructions say when there is a choice the newest arriving job is selected so each job runs right when it arrives.**
10. Given a STCF scheduler, what is the average turnaround time?
A. 7
B. 9
C. 14
D. 18
E. None of the above
**B, Job A arrives at time 0 and completes at time 12, Job B arrives at time 4 and completes at time 8, C arrives at time 7 and completes at time 18. (12+4+11)/3 = 9**

# Part 2. Concurrency

## True / False

11. A spinlock is a good choice of lock to protect a critical section that takes 2-3 cycles.
**True, if critical sections are very short the cost of yielding (a context switch) will be higher than spinning.**
12. If two threads simultaneously (that is, without synchronization) increment a memory address by 1 with a load/add/store assembly instruction sequence, the result can be an increment by either 1 or 2 (with no other possibilities).
**True, even if the threads interleave one of the increments will take effect**
13. The primary purpose of mutex locks is to make sure that one thread runs before another.
**False, mutex locks provide mutual exclusion which ensures exclusive access to a critical section. Condition variables and barriers help with ordering threads.**
14. Semaphores can be used to replace condition variables.
**True, the can replace both condition variables and mutex locks**

15. A semaphore used to create mutual exclusion for critical sections should be initialized to 1.
**True**

`

## Multiple Choice

16. Which one of the following is NOT a necessary condition for deadlock to occur?
A. mutual exclusion
B. hold-and-wait
C. no preemption
D. circular wait
E. All the above are necessary
**E, these are the four necessary conditions for deadlock to occur.**
Consider the following producer-consumer code. The producer thread runs for 'loops' number of iterations and is filling a buffer of size 'MAX'. The buffer holds 'count' items. Consider one producer and one consumer thread.

```
void *producer(void *arg) {
   int i;
   for (i=0; i<loops; i++) {
      mutex_lock(&mutex);           //p1
      while (count == MAX)          //p2
         cond_wait(&empty,&mutex); //p3
      put(i);                       //p4
      cond_signal(&fill);           //p5
      mutex_unlock(&mutex);         //p6
}

void *consumer(void *arg) {
   int i;
   for (i=0; i<loops; i++) {
      mutex_lock(&mutex);          //c1
      while (count == 0)           //c2
         cond_wait(&fill,&mutex); //c3
      int tmp = get();             //c4
      cond_signal(&empty);         //c5
      mutex_unlock(&mutex);        //c6
      printf("%d\n",tmp);          //c7
   }
}
```
17. What are the sequence of lines executed when the producer runs for one iteration followed by the consumer? Assume you start from i=0, count=0, and MAX=5.
A. p1 p2 p3 p4 p5 p6 c1 c2 c3 c4 c5 c6 c7
B. p1 p2 p3 p4 p5 p6 c1 c2 c4 c5 c6 c7
C. p1 p2 p4 p5 p6 c1 c2 c4 c5 c6 c7
D. p1 p2 p4 p5 p6 c1 c2 c3 c4 c5 c6 c7
E. None of the above
**C, the producer skips the while loop since the buffer is not full, and the consumer cannot run until**

**the mutex is released**

18. What are the sequence of lines executed when the consumer thread runs first?
A. c1 c2 c3 p1 p2 p4 p5 p6 c2 c4 c5 c6 c7
B. c1 p1 p2 p4 p5 p6 c2 c4 c5 c6 c7
C. c1 c2 c3 p1 p2 p3 p4 p5 p6 c4 c5 c6 c7
D. c1 c2 c4 c5 c6 c7 p1 p2 p4 p5 p6
E. None of the above
**A**
19. This is a correct implementation of the producer/consumer problem.
A. True
B. False
**A, barring any minor syntax errors ;-)**
20. What is the objective of a reader/writer lock?
A. Only one thread of any type can hold the lock at a time
B. Either only one reader thread can hold the lock at a time or multiple writer threads can hold the lock at the same time
C. Either only one writer thread can hold the lock at a time or multiple reader threads can hold the lock at the same time
D. Multiple reader or writer threads can hold the lock at the same time
E. None of the above
**C**

# Part 3. Persistence

## True/False

21. Because of the physical nature of SSDs they only efficiently support log-structured file systems.
**False, SSDs internally have a Flash Translation Layer that maps logical blocks to physical locations in the flash storage, which makes even non log-structured file systems efficient.**
22. In LFS the location of an inode is not fixed and so another level of indirection is used to map an inode number to its location on disk.
**True, this is the imap data structure.**
23. The reason why data journaling waits to write the transaction commit to the journal until all other writes for the transaction to the journal have completed is so that a crash can be detected when writing to the data portion of the file system.
**False, it allows the ability to detect a crash when the transaction was written to the journal.**
24. When fsck discovers an inconsistency between the inode bitmap and the inode table, it is resolved by trusting the information within the inodes.
**True, this minimizes data loss.**
25. The fast file system (FFS) improved performance by keeping inodes close to the data that they referred to and by keeping files in the same directory close to one another.
**True, treating the disk like a disk to minimize seeks is one of the improvements in the FFS.**
26. When a process reads data from the end of a file, the bitmap is consulted to check if the blocks have been allocated.
**False, the bitmap is only for allocating new blocks and the file system assumes any blocks in an inode were correctly marked as allocated.**

27. The system call to open "/foo/bar/blah.txt" requires exactly 5 read accesses to disk. Assume each inode access is one read, and all directories fit into one disk block.
**False, it requires 7 -- Read root inode, root data, foo inode, foo data, bar inode, bar data, where you find the blah.txt entry and then finally read blah.txt inode.**
28. The directory data contains mappings between file names and the associated inode number and this information is stored in the data block region of the file system.
**True.**
29. Extents are an alternative to individual block pointers to data in an inode.
**True, extents are a pointer and a size so if a file is layed out continuously then a single extent is all that is needed.**
30. Assuming a block size of 4KB and 4-byte block pointers, an inode with 12 direct pointers and 1 indirect pointer supports a maximum file size of just over 6MB.
**False, it is just over 4MB (12 + 1024)*4KB = 4.144 MB.**
31. When an indirect pointer is used in an inode, it points to a block of other inodes in the inode table.
**False, it points to a block in the data region of the disk that contains pointers to other blocks in the data region.**
32. The advantage of using a free list rather than a data bitmap for tracking free blocks in a file system is that no search is needed to allocate a block.
**True, a free list makes allocation straightforward. It does make more sophisticated allocation policies more complicated, since scanning the free list is more expensive than with a bitmap.**
33. The execute permission bit for a directory enables a user (or group or everyone) to do things like navigate into the given directory.
**True, If the execute bit is cleared, then the corresponding person or group cannot change into the directory.**
34. One reason that symbolic links cannot refer to a directory is because of the fear of creating cycles in the directory tree.
**False, this is why hard-links cannot refer to a directory. Symbolic links can refer to a directory.**
35. Only when the reference count of a file reaches zero does the file system free the inode and related data blocks.
**True, until the count is zero other directory entries still refer to the inode and the inode and its data are not deleted.**
36. The metadata for a regular file inode includes the number of hard links to the file.
**True, hard links do not create a separate inode, they just create an entry in the directory information and increment the link count in the metadata of the inode.**
37. Opening the same file twice in a process without closing it first will result in multiple references to the same file struct in the open file table.
**False, each file descriptor reference will point to a unique file struct in the open file table.**
38. The file descriptor, otherwise known as the inode number, is the handle used in Unix systems to access files.
**False, the file descriptor is not the same thing as the inode number. The open file table maps file descriptors to inodes for a given process.**
39. The one main difference between RAID levels 4 and 5 is where the parity information is stored.
**True, in level 4 all parity information is stored on a single disk. In level 5, the parity information is rotated across the disks.**
40. Of the RAID levels covered in class, the two that maximize capacity while still providing some type of redundancy are RAID levels 0 (striped) and 5 (rotating parity).
**False, RAID level 0 provides no redundancy.**
41. The SCAN algorithm is optimal in maximizing the throughput rate of a hard disk.
**False, it does not take into account rotational delays, among other things.**

42. The purpose of track-skew is to make sure that sequential reads/writes crossing track boundaries are serviced optimally, not requiring an additional rotation.
**True, skewing consequitive blocks allows for the seek time of the read head to move from one track to the neighboring track.**
43. In a write-back caching disk a write is acknowledged as soon as the data is placed in the cache as opposed to write-through caching.
**True, that is the definition of write-back caching.**
44. If we create a symbolic link to a file, the reference count for that file's inode is not incremented
**True, symbolic links work by name (and can have dangling references if the referenced file is renamed or deleted).**
45. A log-based file system attempts to make all writes sequential.
**True, trying to take advantage of the improved performance of sequential access over random access.**
46. For a 15,000 RPM disk, the average rotation time for a seek is 4ms
**False, it is 2ms. A full rotation takes 4ms = 1/(15,000/60) but a seek from a random location takes half that time on average.**
47. A device driver is software that executes on the microcontroller of a peripheral device.
**False, the driver code still runs on the main CPU, but it controls the peripheral device. The code in a microcontroller is typically called firmware.**
48. Memory-mapped IO and explicit IO instructions are two alternate architectural approaches for how an OS accesses device registers.
**True, either the architecture has explicit IO instructions or it utilizes the same instructions for interacting with memory but has a reserved memory address space that in actuality interacts with device registers.**
49. Programmed I/O was introduced to relieve the OS of transferring large chunks of data to and from a device's registers for it to perform an IO operation.
**False, this is called Direct Memory Access (DMA), not programmed IO.**
50. Polling wastes less CPU time compared to interrupts to determine when a disk drive has finished an IO operation.
**False, it is just the other way around. Interrupts allow the CPU to be utilized while the IO device is busy performing some command.**
51. Over 70% of the Linux kernel code is found in device drivers.
**True, see *An Empirical Study of Operating System Errors* by Andy Chou, et. al.**

## Multiple Choice

An Operating System can perform the following IO Operations:

mkdir() - create a new directory
creat() - create a new, empty file
open(), write(), close() - appends a block to a file (this is one operation)
link(path1, path2) - creates a hard link at path2 to a file (path1)
unlink() - unlinks a file (removing it if refcnt==0)

A file system like the VSFS discussed in class is used on disk, which has an inode bitmap, a data bitmap, an inode table, and data blocks. Both inodes and data blocks are indexed starting at 0. An inode has three fields: type (f for file and d for directory), address of data block (either -1 if no data or a single integer address for the data block), and a reference count saying how many directory entries there are to this inode. Data blocks are indicated by matching square brackets and can contain file data (a single character) or directory information (name-inode pairs). Directory data always fits into a single data block.

Here is the initial state of the file system:

```
inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []
```

52. What IO operation took place to change the state of the file system to the following?

```
inode bitmap  11000000
inodes       [d a:0 r:3][d a:1 r:2][][][][][][]
data bitmap   11000000
data         [(.,0) (..,0) (k,1)][(.,1) (..,0)][][][][][][]
```

A. mkdir("/k")
B. creat("/k")
C. link("/","/k")
D. unlink("/k")
E. None of the above or Can not determine
**A, mkdir("/k") -- a new inode has appeared and its type is d. An entry in the root directory maps that inode to "k". A new data block for this inode containing directory information has appeared.**

53. What IO operation took place to continue changing the state of the file system to the following (from the previous question)?

```
inode bitmap  11100000
inodes       [d a:0 r:3][d a:1 r:2][f a:-1 r:1][][][][][]
data bitmap   11000000
data         [(.,0) (..,0) (k,1)][(.,1) (..,0) (h,2)][][][][][][]
```

A. creat("/h")
B. creat("/k/h")
C. link("/k","/h")
D. link("/k","/k/h")
E. None of the above or Can not determine
**B, creat("/k/h") -- a new file inode has appeared, and in the data for the /k directory (inode 1 and block 1) an entry for "h" pointing to this inode has appeared.**
54. What IO operation took place to continue changing the state of the file system to the following (from the previous question)?

```
inode bitmap 11100000
inodes       [d a:0 r:3][d a:1 r:2][f a:-1 r:2][][][][][]
data bitmap  11000000
data         [(.,0) (..,0) (k,1) (h,2)][(.,1) (..,0) (h,2)][][][][][][]
```

A. link("/k/h","/h")
B. link("/k","/h")
C. open("/k/h"),write("h"),close()
D. open("/k"),write("h"),close()
E. None of the above or Can not determine

**A, we have a new file /h and it points to inode 2, which is what /k/h points to**

55. What IO Operation took place to continue changing the state of the file system to the following (from the previous question)?

```
inode bitmap 11100000
inodes       [d a:0 r:3][d a:1 r:2][f a:2 r:2][][][][][]
data bitmap  11100000
data         [(.,0) (..,0) (k,1) (h,2)][(.,1) (..,0) (h,2)][u][][][][][]
```

A. open("/k/h"),write("u"),close()
B. open("/h"),write("u"),close()
C. open("/k"),write("u"),close()
D. creat("/u")
E. None of the above or Can not determine

**E, it could be either a write to /k/h to /h (A and B) but since those are links to the same inode we cannot tell**

56. What 2 IO operations took place to continue changing the state of the file system to the following (from the previous question)?

```
inode bitmap 11100000
inodes       [d a:0 r:3][d a:1 r:2][f a:2 r:2][][][][][]
data bitmap  11100000
data         [(.,0) (..,0) (k,1) (h,2)][(.,1) (..,0) (s,2)][u][][][][][]
```

A. creat("/k/s")
unlink("/k/h")

B. link("/k/h","/k/s")
unlink("/k/h")

C. unlink("/k/h")
link("/h","/k/s")

D. unlink("/k/h")
mkdir("/k/s")

E. None of the above or Can not determine

**E, both B and C would lead to this situation in different ways (remember the unlink will not remove the file since a reference still exists)**

This same file system may contain errors or inconsistencies in its data structures. For each of the following file system states, identify the error or inconsistency.

57.

```
inode bitmap 11100000
inodes       [d a:0 r:3][d a:1 r:2][f a:2 r:2][][][][][]
data bitmap  11100000
data         [(.,0) (..,0) (k,1) (h,2)][(.,0) (..,1) (h,2)][u][][][][][]
```

A. Inode bitmap mismatch with inode table
B. Inode's refcount is incorrect
C. Directory's data connecting it to the directory tree is incorrect
D. File system is in a consistent state
E. None of the above
**C, The directory data associated with inode at index 1 has incorrect entries for . and .. -- .. should refer to its parent but it doesn't and . should refer to itself but it doesn't.**

58.

```
inode bitmap 11100000
inodes       [d a:0 r:3][d a:1 r:3][f a:2 r:2][][][][][]
data bitmap  11100000
data         [(.,0) (..,0) (k,1) (h,2)][(.,1) (..,0) (h,2)][u][][][][][]
```

A. Inode bitmap mismatch with inode table
B. Inode's refcount is incorrect
C. Directory's data connecting it to the directory tree is incorrect
D. File system is in a consistent state
E. None of the above
**B, inode at index 1 has a refcount of 3 but there are only 2 directory entries referring to the inode.**

59.

```
inode bitmap 11000000
inodes       [d a:0 r:3][d a:1 r:2][f a:2 r:2][][][][][]
data bitmap  11100000
data         [(.,0) (..,0) (k,1) (h,2)][(.,1) (..,0) (h,2)][u][][][][][]
```

A. Inode bitmap mismatch with inode table
B. Inode's refcount is incorrect
C. Directory's data connecting it to the directory tree is incorrect
D. File system is in a consistent state
E. None of the above
**A, Inode bitmap does not match inode table. There are three inodes in use but bitmap only shows 2 in use.**

A journaling system is added to the above file system to prevent inconsistencies. For each of the following IO operations and journal types, of the options listed, identify the one that is written to the journal.

60. creat("/foo") using data journaling.
A. inode bitmap
B. data bitmap
C. a new data block
D. More than one of the above are written to the journal
E. None of the above are written to the journal
**A, creating a file requires reading and updating the inode bitmap, writing a new inode, and writing**

**a directory entry in the directory data where the file is created. All of these writes are metadata and would be in a journal entry when data journaling.**

61. mkdir("/bar") using metadata journaling.
A. data bitmap
B. root's directory data
C. a new data block
D. More than one of the above are written to the journal
E. None of the above are written to the journal

**D, creating a new directory would require adding a directory entry into the parent, creating a new block of directory data, updating both the inode bitmap and data bitmap, and writing a new inode to the inode table. In metadata journaling all of this is written to the journal (the new data block is considered file system data since its part of a directory).**

62. open("/foo"),write("x"),close() using metadata journaling to write to an empty file.
A. root's directory data
B. a new data block
C. inode bitmap
D. More than one of the above are written to the journal
E. None of the above are written to the journal

**E, writing a new block of data requires updating the data bitmap (not an option) updating the inode (not an option) and writing the new data block (not written to the journal when using metadata journaling), so none of the above.**

63. link("/foo","/foo2") using data journaling.
A. inode bitmap
B. root's directory data
C. a new data block
D. More than one of the above are written to the journal
E. None of the above are written to the journal

**B, creating a hardlink requires updating the directory data to contain a new entry, and updating the inode's refcount. Only the root's directory data is an option.**

You decide to change the file system to a log-structured file system. In this file system, each block may contain one of the following:

* A checkpoint region containing block addresses of the live chunks of the imap
* An inode containing its type (regular file or dir), size (in blocks), reference count, and pointers to data blocks
* Directory data (name to inode number mappings)
* A chunk of the imap listing blocks containing inodes (the position (index) within the chunk is the inode number)
* File data

```
INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Notice that the initial file system consists of 4 live blocks.

As the file system is updated by IO operations, you will be presented with the contents of the file system and will be asked to identify which blocks of the file system are live and what IO operations were performed.

64. Three IO operations were performed on the initial file system state to reach the state shown below. Which blocks are live?

```
[   0 ] checkpoint: 16 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] [.,0] [..,0] -- -- -- -- -- --
[   2 ] type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] [.,0] [..,0] [my4,1] -- -- -- -- --
[   5 ] type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] [.,0] [..,0] [my4,1] [ke3,1] -- -- -- --
[   9 ] type:dir size:1 refs:2 ptrs: 8 -- -- -- -- -- -- --
[  10 ] type:reg size:0 refs:2 ptrs: -- -- -- -- -- -- -- --
[  11 ] chunk(imap): 9 10 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  12 ] [.,0] [..,0] [my4,1] [ke3,1] [ka0,2] -- -- --
[  13 ] [.,2] [..,0] -- -- -- -- -- --
[  14 ] type:dir size:1 refs:3 ptrs: 12 -- -- -- -- -- -- --
[  15 ] type:dir size:1 refs:2 ptrs: 13 -- -- -- -- -- -- --
[  16 ] chunk(imap): 14 10 15 -- -- -- -- -- -- -- -- -- -- -- -- --
```

A. 0, 2, 5, 6, 16
B. 0, 8, 9, 10, 16
C. 13, 15
D. 0, 10, 12, 13, 14, 15, 16
E. None of the above

**D, The checkpoint region is live (0), which points to a chunk of the imap (16), which points to inodes 0,1,2 in blocks 14, 10, and 15. These blocks containing inodes point to data blocks (inode 0 points to 12, inode 1 points to nothing, and inode 2 points to 13).**

65. What three IO operations would result in the above file system state?

A. create file /my4
create file /ke3
create file /ka0

B. create file /my4
link file /my4 to /ke3
create dir /ka0

C. create dir /my4
create file /ke3
write data to file /ke3

D. create file /my4
write data to file /my4
create dir /ka0

E. None of the above

**B, find the root inode by going to block 16, inode 14 is the root, 12 is its data. It points to my4, ke3, ka0 - notice that /my4 is inode 1 in block 10 and is a regular file, /ke3 points to the same inode, and /ka0 is inode 2 in block 15 and is a directory.**

66. What block contains the live root inode?

A. 2
B. 5
C. 9
D. 14
E. None of the above

**D, the checkpoint region points to block 16. The 0th index (the root) points to block 14.**

You use an SSD to hold your data. The SSD has 3 blocks with 10 pages per block. Each page holds a single character. The SSD has been used for a while and the current state of the SSD is shown below. The diagram for the SSD contains the following items:

FTL mapping logical pages to physical pages (1st line)
block number (2nd line)
page number (3rd and 4th lines)
state of each page (valid, Erased, or invalid) (5th line)
data stored at each page (6th line)
an indicator(+) if a page is currently live (has an entry in the FTL) (7th line)

Current State of SSD:

```
FTL     1:6   4:5  19:7  20:3

Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvEE iiiiiiiiii iiiiiiiiii
Data  z9fFAdux
Live     + +++
```

The OS can perform 3 operations:

read(logical page #) -- if page is live return the character at the page, otherwise error
write(logical page #, character) -- writes a character to the logical page #.
erase(logical page #) -- removes logical page # from the FTL mapping if there, otherwise error

67. What single OS operation will result in the following SSD state?

```
FTL     1:6   4:8  19:7  20:3

Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvE iiiiiiiiii iiiiiiiiii
Data  z9fFAduxz
Live     +  +++
```

A. write(8,z)
B. read(8)
C. write(4,z)
D. erase(4)
E. None of the above
**C, the mapping for logical page 4 has changed to now map to physical page 8 and the content at that page has the value 'z'.**
68. Continuing making OS operations to the SSD, a read(20) is performed.  What is the return value?
A. z
B. F
C. v
D. error
E. None of the above
**B, the FTL maps logical page 20 to physical page 3 which contains an 'F'.**
69. Another OS operation is performed which changes the state of the SSD again to that shown below. What was the operation?

```
FTL     1:6   2:9    4:8   19:7    20:3
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvv iiiiiiiiii iiiiiiiiii
Data  z9fFAduxza
Live     +  ++++
```

A. write(9,a)
B. erase(4)
C. write(2,z)
D. write(1,9)
E. None of the above
**E, none of the above. A new mapping is in the FTL mapping 2:9 and the content at physical page 9 is 'a'. The operation was write(2,a), which is not an option.**
70. Now the OS performs a write(28,q) operation.  What low-level SSD operations must be performed in order to carry out this OS operation?

A. erase block 1
program q to physical page 10
Enter mapping 28:10 in FTL

B. read physical pages 3, 6, 7, 8, 9
erase block 0
program physical pages F to 10, u to 11, x to 12, z to 13, a to 14, q to 15

C. program q to physical page 10
Enter mapping of 10:28 in FTL

D. erase page 10

program q to physical page 10
Enter mapping 28:10 in FTL

E. None of the above

**A, since the next page is invalid, the block must first be erased, then the first page of the block can be programmed and the mapping entered into the FTL.**

71. Some time later the state of the SSD is as shown below. The garbage collector decides to reclaim block 0. What will the state of the SSD be after the garbage collector has finished?

```
FTL     1:6   2:9   4:13   5:12
        7:11  19:7  20:3   23:14
        28:10
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvEEEEE iiiiiiiiii
Data  z9fFAduxza qh7Et
Live     +  ++ + +++++
```

```
A.
FTL     1:   6   2:   9   4: 13   5: 12
        7: 11  19:   7  20:   3  23: 14
       28: 10
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvEEEEE iiiiiiiiii
Data  z9fFAduxza qh7EtFuxa
Live     +  ++ + +++++
```

```
B.
FTL     1:21   2:24   4:13   5:12
        7:11   19:22  20:18  23:14
        28:10
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvEEEEE iiiiiiiiii
Data  z9fFAduxza qh7Etz9fFA duxza
Live             +++++   +   ++ +
```

```
C.
FTL     1:21   2:24   4:13   5:12
        7:11   19:22  20:18  23:14
```

```
        28:10
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State EEEEEEEEEE vvvvvvvvvv vvvvvEEEEE
Data             qh7Etz9fFA duxza
Live             +++++   +   ++ +

D.
FTL     1:16   2:18   4:13   5:12
        7:11  19:17  20:15  23:14
        28:10
Block 0          1          2
Page  0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvvvvvE iiiiiiiiii
Data  z9fFAduxza qh7EtFuxa
Live             +++++++++
```

E. None of the above

**D, only the live pages are copied, the FTL mappings are updated to point to the new pages.**

# Part 4. Distributed Systems and NFS

72. With the TCP protocol, what does a receiver do if it receives a packet that it has already received?
A. Drop the packet
B. Send an acknowledgement and drop the packet
C. Send an acknowledgement and forward the packet to the application
D. Forward the packet to the application
E. None of the above
**B, send an acknowledgement and drop the packet. The sender must not have received the acknowledgement to the earlier packet and so the receiver needs to send it again, but the data in the packet has already arrived and so the data should not be sent to the application.**

73. What reliability is built into the UDP protocol?
A. Packet loss detection
B. Mechanism for detecting data corruption
C. Guarantees data is received in order it was sent
D. Mechanism for sender to execute code on the receiver
E. None of the above
**B, the checksum is used to detect data corruption during transmission.**
74. In NFS, a client opens a file and obtains file descriptor 145 and a file handle <volume = 1, inode = 23, generation = 2>. After a few seconds, the original client issues a request for this file and the server finds the generation number is 3 and returns an error to the client. What must have happened in the meantime?
A. The server crashed and rebooted
B. File descriptor 145 was closed and reopened with a different file
C. The file for inode 23 was removed and re-allocated for a different file/directory
D. Another client wrote to this file and the original client needs to update its local cached version

E. This situation is impossible, there is a bug in the server

**C, the generation number tracks inode reuse to avoid the client interacting with an unrelated file with the same inode number.**

75. In the NFS protocol, the mkdir operation is not quite idempotent. How does this affect clients?

A. No effect - it only affects performance on an unreliable network
B. The operation can succeed even if the directory already exists
C. The operation can fail and do nothing even though the directory does not exist
D. The operation can fail even though it successfully created the directory
E. None of the above

**D, the issue is if the first time mkdir is issued it succeeds but the ack is lost. The retry will return an error that the directory already exists even though the overall effect is to create the directory.**

**That's it.  Thank you for taking CS 537 and have a great summer!**