# CS 537: Introduction to Operating Systems
# Fall 2015: Midterm Exam #3
# Thursday, November 19th 7:15-9:15

## Persistence

This exam is closed book, closed notes.

All cell phones must be turned off.

No calculators may be used.

You have two hours to complete this exam.

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

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

Please fill in the accu-scan form with your Last Name, First Name and Student Identification Number; remember to fill in the corresponding bubbles as well.

**Good luck!**

**This exam has multiple versions. To make sure you are graded with the correct answer key, you must identify this exam version with a Special Code in Column A on your accu-tron sheet. Your special code is: 2. Be sure to fill in the corresponding bubble as well.**

**The other version of this exam simply places the True/False questions first.**

**Disks. [2 points each]**
Consider a disk with the following characteristics:

- Number of surfaces: 8 (= $2^3$)
- Number of tracks / surface: 512 K (= $2^{19}$)
- Number of bytes / track: 8 MB (= $2^{23}$ bytes)
- Number of sectors / track: 8 K (= $2^{13}$)
- On-disk cache: 16 MB (= $2^{24}$ bytes)

1) How many heads does this disk have?
   a) 1
   b) 2
   c) 4
   **d) 8**
   e) Not enough information or none of the above
   ***8 heads: 1 head per surface***
2) What is the size of each sector?
   a) 512 bytes
   **b) 1KB**
   c) 2KB
   d) 4KB
   e) Not enough information or none of the above
   ***2^23 bytes / track * track/2^13 sectors = 2^10 bytes /sector***
3) How many bytes per cylinder?
   a) 1 MB
   b) 8 MB
   **c) 64 MB**
   d) $2^{42}$ bytes
   e) Not enough information or none of the above
   ***2^23 bytes / track * 8 tracks / cylinder = 2^26 bytes / cylinder***
4) What is the total capacity of this disk?
   a) $2^{42}$ bytes
   **b) $2^{45}$ bytes**
   c) $2^{35}$ bytes
   d) $2^{38}$ bytes
   e) Not enough information or none of the above
   ***2^26 bytes / cylinder * 2^19 cylinders = 2^45 bytes***

Assume the disk head is at cylinder 18 and moving towards higher cylinders. Assume a stream of requests arrives for the following cylinders: `5, 20, 1, 60, 3, 8, 90, 2, 20, 40, 6, 70.`

5) With a **FCFS** scheduling policy, what is the **total seek distance** to service all the requests?
   a) 519
   b) 521
   c) 534
   d) 586
   **e) Not enough information or none of the above**

***Starts at cylinder 18; to go from 18 to 5 to 20 to 1 … to 70 = 13 + 15 + 19 + … + 64 = 474.***

6) With a **SSTF** scheduling policy, what is the **total seek distance**?
   a) 108
   **b) 110**
   c) 119
   d) 150
   e) Not enough information or none of the above

***Schedule: 18 to 20, 20, 8, 6, 5, 3, 2, 1, 40, 60, 70, 90 → 2 + 19 + 89 = 110***

7) With a **SCAN** scheduling policy (bi-directional scanning), what is the **total seek distance**?
   a) 159
   **b) 161**
   c) 166
   d) 168
   e) Not enough information or none of the above

***Schedule: 18 to 90 (servicing requests between 18 and 90) and back down to 1 (servicing requests between 18 and 1) → 72 + 89 + 161***

8) With a **C-SCAN** scheduling policy (uni-directional scanning), what is the **total seek distance**?
   a) 159
   b) 161
   c) 166
   **d) 168**
   e) Not enough information or none of the above

***Schedule: 18 to 90 (servicing requests) then down to 1 (but not servicing requests here), then back to 8 → 72 + 89 + 7 = 168***

**RAID Mapping [3 points each]**

The next questions ask you to translate logical read and write operations performed on top of a RAID system to the physical read and write operations that will be required to the underlying disks. Specifically, for each RAID configuration, translate the logical requests to the physical operations performed on the correct disk number and physical block address (offset). In all cases assume a block and chunk size of 4 KB. These questions are all similar to those available from the homework simulations.

9) **RAID Level: 0**; **Number of Disks: 8;** Random Read from logical block number 58
   a) Read from disk 2, offset 2
   **b) Read from disk 2, offset 7**
   c) Read from disk 7, offset 2
   d) Read from disk 7, offset 7
   e) None of the above

   ***RAID-0 is simple striping;***
   ***58 % 8 (number of disks) = 2 → disk 2***
   ***58 / 8 (integer division) = 7 → block offset 7***

10) **RAID Level: 1; Disks: 8;** Random Write to logical block 29
   a) Write to disk 1 and disk 5 at offset 7
   **b) Write to disk 2 and disk 3 at offset 7**
   c) Write to disk 5 at offset 3
   d) Write to disk 3 at offset 5
   e) None of the above

   ***RAID-1 is mirroring; with 8 disks, it is like having 4 mirrored pairs***

***29 % 4 = 1 → mirrored pair number 1, which are disks 2, 3***
***29 / 4 = 7 → offset 7***

11) **RAID Level: 4; Disks: 4;** Random Write to logical block 50
   a) Write to disk 2, offset 12
   b) Write to disk 2, offset 16
   c) Read from disk 2, offset 16; Write to disk 2, offset 16.
   **d) Read from disk 2 and 3, offset 16; Write to disk 2 and 3, offset 16.**
   e) None of the above

***Since this is a random write, the best approach is to read the old data and the old parity and then flip the parity for every bit that is changed in the new data.***
***With RAID-4, disks 0, 1, 2 are used for data; disk 3 for parity.***
***50 / 3 (number of data disks) = 16 → offset 16***
***50 % 3 = 2 → disk 2 for data***

12) **RAID Level: 5 (Left Symmetric); Disks: 4;** Random Read from logical block 10
   a) Read from disk 0, offset 3
   b) Read from disk 2, offset 2
   **c) Read from disk 2, offset 3**
   d) Read from disk 3, offset 2
   e) None of the above

The pattern for left-symmetric is:

| ***disk 0*** | ***disk 1*** | ***disk 2*** | ***disk 3*** |
|---|---|---|---|
| ***0*** | ***1*** | ***2*** | ***P*** |
| ***4*** | ***5*** | ***P*** | ***3*** |
| ***8*** | ***P*** | ***6*** | ***7*** |
| ***P*** | ***9*** | ***10*** | ***11*** |

13) **RAID Level: 5 (LS); Disks: 4; Sequential Write** to logical blocks 15, 16, and 17.
   a) Write to disks 0, 1, and 3 at offset 5
   **b) Write to disks 0, 1, 2, and 3 at offset 5**
   c) Read and write to disks 0, 1, and 3 at offset 5
   d) Read and write to disks 0, 1, 2, and 3 at offset 5
   e) None of the above

***Continuing into the next group…***

| ***disk 0*** | ***disk 1*** | ***disk 2*** | ***disk 3*** |
|---|---|---|---|
| ***12*** | ***13*** | ***14*** | ***P*** |
| ***16*** | ***17*** | ***P*** | ***15*** |

***Since blocks 15, 16, and 17 are part of the same stripe, the parity block can be calculated directly from those data blocks without needing to read any old data; the data blocks and the parity block for that stripe are simply written to.***

**Basic File System Operations and Data Structures [3 points each]**
These questions ask you to understand how different file system operations lead to different file system data structures being modified on disk. You do not need to consider journaling or crash consistency in these questions. This part is based on the available homework simulations.

This file system supports 7 operations:
- mkdir() - creates a new directory
- creat() - creates a new (empty) file
- open(), write(), close() - appends a block to a file
- link() - creates a hard link to a file
- unlink() - unlinks a file (removing it if linkcnt==0)

The state of the file system is indicated by the contents of four different data structures:
inode bitmap - indicates which inodes are allocated (not shown, because not needed for questions)
inodes - table of inodes and their contents
data bitmap - indicates which data blocks are allocated (not shown)
data - indicates contents of data blocks

The inodes each have three fields: the first field indicates the type of file (f for a regular file, d for a directory); the second indicates which data block belongs to a file (here, files can only be empty, which have the address of the data block set to -1, or one block in size, which would have a non-negative address); the third shows the reference count for the file or directory. For example, the following inode is a regular file, which is empty (address field set to -1), and has just one link in the file system: [f a:-1 r:1]. If the same file had a block allocated to it (say block 10), it would be shown as follows: [f a:10 r:1]. If someone then created a hard link to this inode, it would then become [f a:10 r:2].

Data blocks can either retain user data or directory data. If filled with directory data, each entry within the block is of the form (name, inumber), where "name" is the name of the file or directory, and "inumber" is the inode number of the file. Thus, an empty root directory looks like this, assuming the root inode is 0: [(.,0) (..,0)]. If we add a single file "f" to the root directory, which has been allocated inode number 1, the root directory contents would then become: [(.,0) (..,0) (f,1)]

If a data block contains user data, it is shown as just a single character within the block, e.g., "h". If it is empty and unallocated, just a pair of empty brackets ([]) are shown.

Empty inodes and empty data blocks may not all be shown.

An entire file system is thus depicted as follows:

```
inode bitmap 11110000
inodes       [d a:0 r:6] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] ...
data bitmap  11100000
data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)] [] ...
```

This file system has eight inodes and eight data blocks. The root directory contains three entries (other than "." and ".."), to "y", "z", and "f". By looking up inode 1, we can see that "y" is a regular file (type f), with a single data block allocated to it (address 1). In that data block 1 are the contents of the file "y": namely, "u". We can also see that "z" is an empty regular file (address field set to -1), and that "f" (inode number 3) is a directory, also empty. You can also see from the bitmaps that the first four inode bitmap entries are marked as allocated, as well as the first three data bitmap entries.

Assume the initial state of the file system is as follows:

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

If the file system transitions into each of the following states, what operation or operations must have been performed? The state of the file system is cumulative across questions.

14) File System State:

```
inodes   [d a:0 r:3] [d a:1 r:2] [] [] [] []
data     [(.,0) (..,0) (c,1)] [(.,1) (..,0)] [] []
```

**a) mkdir("/c");**
b) creat("/c");
c) link("/", "/c");
d) mkdir("/c/d");
e) None of the above

***An entry for "/c" exists in the top directory and its inode indicates that it is a directory and not a file.***

15) File System State:

```
inodes   [d a:0 r:3] [d a:1 r:3] [f a:-1 r:1] [] [] []
data     [(.,0) (..,0) (c,1)] [(.,1) (..,0) (q,2)] [] []
```

a) mkdir("/c/q");
**b) create("/c/q");**
c) mkdir("/q");
d) create("/q");
e) None of the above

***An entry for "q" exists in the data for "/c" and q's inode indicates that it is a file.***

16) File System State after **TWO** operations:

```
inodes   [d a:0 r:4] [d a:1 r:4] [f a:-1 r:2] [f a:-1 r:1] [] []
data     [(.,0) (..,0) (c,1) (r,2)] [(.,1) (..,0) (q,2) (j,3)] [] []
```

**a) creat("/c/j"); link("/c/q", "/r");**
b) creat("/c/j"); creat("/r");
c) creat("/r"); link ("/c/q", "/c/j");
d) mkdir("/j"); link("/c/q", "/r");
e) None of the above

***An entry for "j" exists in the directory "/c"; j's inode (3) indicates that it is a file. An entry exists for "r" within the root directory; it is pointing to inode 2 which is also pointed to by "/c/q".***

17) File System State after **TWO** operations:

```
inodes   [d a:0 r:5] [d a:1 r:4] [f a:2 r:2] [f a:-1 r:1] [f a:-1 r:1]
data     [(.,0) (..,0) (c,1) (r,2) (g,4)] [(.,1) (..,0) (q,2) (j,3)] [v]
```

**a) creat("/g");**
**fd=open("/c/q", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);**
b) creat("/c/g");
fd=open("/c/q", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);
c) creat("/c/g");
fd=open("/c/g", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);
d) creat("/g"); creat("c/q");
e) None of the above.

***A new entry "g" exists in "/"; g's inode indicates it is a file. There is now data associated with inode 2, which is pointed to by "/c/q".***

**Links [2 points each]**

Imagine the following commands are run on an FFS-like file system that supports hard and soft links.

```
echo "file" > file
ln file file2
echo "file2" >> file2
mv file file3
echo "file3" >> file3
```

18) What output will you see if you run `cat file2`( assume nothing wrong with whitespace or carriage returns)?
    a) file2\n
    b) file\n file2\n
    **c) file\n file2\n file3\n**
    d) file2: No such file or directory
    e) None of the above

**Imagine "file" points to a newly allocated inode, number I. Since a hard link is used, "file2" will refer to this same inode. Changing the name "file" to "file3" in the directory does nothing to the actual contents of inode I. It does not matter which name is used to refer to inode I.**

19) What output will you see if you run `cat file3` (assume nothing wrong with whitespace or carriage returns)?
    a) file3\n
    b) file\n file3\n
    **c) file\n file2\n file3\n**
    d) file3: No such file or directory
    e) None of the above

Imagine the following commands are run instead on an FFS-like file system:

```
echo "bar" > bar
ln -s bar bar2
echo "bar2" >> bar2
mv bar bar3
echo "bar3" >> bar3
```

20) What output will you see (assume nothing wrong with whitespace or carriage returns) if you run `cat bar2 ?`
    a) bar2\n
    b) bar\n bar2\n
    c) bar\n bar2\n bar3\n
    **d) bar2: No such file or directory**
    e) None of the above

***Since bar2 is only a soft (or symbolic) link, bar2 actually points to the pathname "bar" (and not directly to the same inode). As a result, if bar is deleted or renamed, bar2 will not point to anything valid.***

21) What output will you see (assume nothing wrong with whitespace or carriage returns) if you run `cat bar3 ?`
    a) bar3\n
    b) bar\n bar3\n
    **c) bar\n bar2\n bar3\n**
    d) bar3: No such file or directory
    e) None of the above

***When the command "echo bar2 >> bar2" was executed, bar2 pointed correctly to bar.***

**Crash Consistency [2 points each]**

Imagine you have an FFS-like file system that is creating a new empty file in an existing directory and must update 4 blocks: the directory inode, the directory data block, the file inode, and the inode bitmap. Assume the directory inode and the file inode are in different on-disk blocks. Assume this initial system does not perform any journaling and FSCK is not run. What happens if a crash occurs after only updating the following block(s)?

22) Bitmap

a) No inconsistency (it simply appears that the operation was not performed)
**b) Data/inode leak**
c) Multiple file paths may point to same inode
d) Point to garbage
e) Multiple problems listed above

***The bitmap shows that this inode has been used, but no structure is actually pointing to this inode; therefore, this inode will be lost for any useful purpose (much like a memory leak). Imagine what happens if the user repeatedly retries this file create operation and the file system repeatedly crashes after only the bitmap has been written persistently to disk; each time the file system reboots and tries again, it will allocate another inode by marking the bitmap until all the inodes appear to be used up.***

23) File inode

**a) No inconsistency (it simply appears that the operation was not performed)**
b) Data/inode leak
c) Multiple file paths may point to same inode
d) Point to garbage
e) Multiple problems listed above

***If only the file inode is written, then no directory points to this new inode and the bitmap does not show this inode has been written. The fact that this inode has been written has no effect or impact on any part of the file system. Imagine what happens if the user repeatedly retries this file create operation and only writes out the file inode; each time, this same inode or a different inode may be written to, and the work is simply lost without any other consequences.***

24) Directory inode and Directory data

a) No inconsistency (it simply appears that the operation was not performed)
b) Data/inode leak
c) Multiple file paths may point to same inode
d) Point to garbage
**e) Multiple problems listed above**

***In this case, the directory will point to an inode, but the inode will contain old contents (problem d: garbage); furthermore, since the bitmap is not marked, this same inode may be allocated to other files, in which case there will be multiple file paths that incorrectly point to this inode (problem c). Imagine what happens if the user tries to repeat this operation; the directory entry will exist (so it will not attempt to create the file again) but the inode may be nonsensical.***

25) Bitmap and File inode

a) No inconsistency (it simply appears that the operation was not performed)
**b) Data/inode leak**
c) Multiple file paths may point to same inode
d) Point to garbage
e) Multiple problems listed above

***In this case, no directory entry points to the file inode. Therefore, if the file create operation is retried, a new inode will be allocated leading to an inode leak (identical to question 22).***

26) Bitmap and Directory inode and Directory Data

a) No inconsistency (it simply appears that the operation was not performed)
b) Data/inode leak
c) Multiple file paths may point to same inode
**d) Point to garbage**
e) Multiple problems listed above

***The directory entry points to a garbage inode; if the file create operation is repeated, the directory entry will already exist and so it will not allocate another inode. Because the bitmap shows this inode is already allocated, it will not be allocated to another file.***

27) File inode and Directory inode and Directory Data

a) No inconsistency (it simply appears that the operation was not performed)
b) Data/inode leak
**c) Multiple file paths may point to same inode**
d) Point to garbage
e) Multiple problems listed above

***The directory entry points to a correctly initialized inode, but the bitmap does not show that this inode has been allocated; therefore, this inode may be allocated to another file at some point in the future.***

Assume we've added a basic implementation of full-data journaling to our FFS-like file system and perform the same **file create** operation as above. Assume a **transaction header** block and a **transaction commit** block. Assume each block is written synchronously (i.e., a barrier is performed after every write and blocks are pushed out of the disk cache). If the system crashes after the following number of blocks have been synchronously written to disk, what will happen after the system reboots? (If the number of disk writes exceeds those needed, assume they are unrelated.)

1) 1 disk write (hint: just the transaction header block is written to disk)
   **a) No transactions replayed during recovery; file system in old state**
   b) No transactions replayed during recovery; file system in new state
   c) Transaction replayed during recovery; file system in old state
   d) Transaction replayed during recovery; file system in new state
   e) Transaction replayed during recovery; file system in unknown state

***If only the transaction header is written, the transaction will not be replayed during recovery and the file system will be in the old state (i.e., the four blocks needed to record a file create operation, will not have been updated in their fixed in-place locations).***

2) 4 disk writes (hint: transaction header, plus 3 blocks to journal)
   a) No transactions replayed during recovery; file system in old state
   b) No transactions replayed during recovery; file system in new state
   c) Transaction replayed during recovery; file system in old state
   d) Transaction replayed during recovery; file system in new state
   e) Transaction replayed during recovery; file system in unknown state

***Since the transaction commit block is not written, the transaction will not be replayed during recovery and the file system will be in the old state (i.e., the four blocks needed to record a file create operation, will not have been updated in their fixed in-place locations).***

3) 5 disk writes (hint: transaction header, plus 3 blocks to journal, plus ???)
   **a) No transactions replayed during recovery; file system in old state**
   b) No transactions replayed during recovery; file system in new state
   c) Transaction replayed during recovery; file system in old state
   d) Transaction replayed during recovery; file system in new state
   e) Transaction replayed during recovery; file system in unknown state

***The 5 disk writes will be to the transaction header plus 4 blocks for the journal transaction (inode bitmap, file inode, dir inode, dir data). Since the transaction commit block is not written, the transaction will not be replayed during recovery and the file system will be in the old state (i.e., the four blocks needed to record a file create operation, will not have been updated in their fixed in-place locations).***

4) 6 disk writes
   a) No transactions replayed during recovery; file system in old state
   b) No transactions replayed during recovery; file system in new state
   c) Transaction replayed during recovery; file system in old state
   d) Transaction replayed during recovery; file system in new state
   e) Transaction replayed during recovery; file system in unknown state

***The 6 disk writes now include the transaction commit block. Since the entire transaction is valid, it will be replayed during recovery and the 4 in-place blocks will all be correctly and atomically updated.***

5) 8 disk writes
   a) No transactions replayed during recovery; file system in old state
   b) No transactions replayed during recovery; file system in new state
   c) Transaction replayed during recovery; file system in old state
   d) Transaction replayed during recovery; file system in new state

e) Transaction replayed during recovery; file system in unknown state

***The 8 disk writes include the transaction commit block plus two updates to the in-place blocks. Since the entire transaction is valid, it will be replayed during recovery and the 4 in-place blocks will all be correctly and atomically updated.***

6) 10 disk writes
   a) No transactions replayed during recovery; file system in old state
   b) No transactions replayed during recovery; file system in new state
   c) Transaction replayed during recovery; file system in old state
   d) Transaction replayed during recovery; file system in new state
   e) Transaction replayed during recovery; file system in unknown state

***The 10 disk writes include the transaction commit block plus four updates to the in-place blocks. Since the entire transaction is valid, it will be replayed during recovery and the 4 in-place blocks will all be written to a second time (but this is fine, just some duplicate work).***

7) 11 disk writes
   a) No transactions replayed during recovery; file system in old state
   **b) No transactions replayed during recovery; file system in new state**
   c) Transaction replayed during recovery; file system in old state
   d) Transaction replayed during recovery; file system in new state
   e) Transaction replayed during recovery; file system in unknown state

***The 11$^{th}$ disk write resets the transaction commit block back to zero, indicating that the checkpoint to the in-place blocks completed. Therefore, during recovery, this transaction will not be replayed.***

**True/False about Virtualization [1 point each]**

Designate if the statement is True (a) or False (b).

8) **Cooperative multi-tasking** requires hardware support for a timer interrupt.
   *False; cooperative assumes the running process will voluntarily relinquish the CPU.*

9) A **RR scheduler** may preempt a previously running job.
   *True.*

10) A RR scheduler delivers better average turn-around time then FCFS when the length of the jobs is nearly identical.
    *False; if the job lengths are nearly identical, then with RR the jobs will finish at nearly the same time as one another, which gives them all very poor turn-around time.*

11) With a single linear page table (and no other support), fetching and executing an instruction that performs an add of a constant value to a register will involve exactly **two memory references**.
    *True; single linear page table implies one extra lookup per address translation; fetching the instruction requires one address translation.*

12) Paging approaches suffer from **external fragmentation**, which grows as the size of a page grows.
    *False; paging has fixed-sized pages and thus suffers from internal fragmentation.*

13) A TLB **caches** translations from physical page numbers to virtual page numbers.
    *False; TLB caches translations from virtual page numbers to physical page numbers.*

14) **TLB reach** is defined as the number of TLB entries multiplied by the size of a page.

*True;*

15) A TLB miss is usually faster to handle than a page miss.

*True; missing in the TLB just requires accessing RAM to walk the page tables; handling a page miss requires fetching a page from disk (milliseconds).*

16) A single page can be **shared** across two address spaces by having different entries in two different page tables point to the same physical page.

*True; two different virtual pages in two different address spaces can thus point to the same PPN.*

17) If the **present bit** is clear (equals 0) in a PTE needed for a memory access, the running process is likely to be killed by the OS.

*False; present bit is 0 just means the page is out on disk and needs to be fetched.*

**True/False about Concurrency [2 points each]**

Designate if the statement is True (a) or False (b).

18) The **clock frequency** of CPUs has been increasing exponentially each year since 1985.

*False; in recent years, the improvement is no longer exponential, which is why we need concurrency to see performance improvements.*

19) Context-switching between threads of the same process requires **flushing the TLB** or **tracking an ASID** in the TLB.

*False; threads are part of the same address space so they have the same address translations as one another.*

20) The **hardware atomic exchange** instruction works only on uniprocessor systems.

*False; this would be a rather useless instruction for modern systems!*

21) A lock that performs spin-waiting cannot provide **fairness** across threads (i.e., threads receive the lock in the order they requested the lock).

*False; the famous ticket lock provides fairness and can spin-wait (it just isn't efficient).*

22) Periodically **yielding** the processor while spin waiting reduces the amount of wasted time to be proportional to the duration of a context-switch.

*True; when this process yields because it can't acquire the lock, the scheduler will pick another process to run (that hopefully has useful work to do), at the cost of a context-switch.*

23) A condition variable can be used to provide **mutual exclusion**.

*False; need locks (or monitors) or semaphores for mutual exclusion.*

24) When a thread returns from a call to **cond_wait()** it can safely assume that it holds the corresponding mutex.

*True; calling cond_wait relinquishes the associated mutex, but then reacquires it before returning.*

25) A call to **cond_signal()** will always wake up at least one thread.

*False; it only wakes up a thread if there is a waiting thread (otherwise the signal/wakeup is lost).*

26) Building a condition variable on top of a semaphore is **easier** than building a semaphore over condition variables and locks.

*False; building condition variables is tough due to the possibility of missing the signal just as a process is being put to sleep.*

27) To implement **thread_join()** with a semaphore, the semaphore value should be initialized to 0.

*True; if it is 0, then the exiting thread must call sema_post() to increment the semaphore to 1 so that the call to sema_wait() within thread_join() will proceed.*

28) A **wait-free** algorithm relies on condition variables instead of locks.

*False, they use atomic hardware instructions.*

29) After a process calls **fork()**, the parent and child processes will share the same address space.

*False; the child starts off with a copy of the parent's address space, but it is a copy that will change over time.*

30) A buggy thread may be able to **overwrite** variables belonging to another thread of the same process.

*True; because the threads are in the same address space, it is possible for one thread to read and/or write another thread's variables (even those allocated on the stack).*

31) On a context-switch across threads of the same process, the **general-purpose registers** must be saved and restored.

*True; all the registers are virtualized so each thread appears to have its own set of registers.*

32) A call to **sema_wait()** releases an associated mutex lock and reacquires the lock before returning.

*False; semaphores do not have associated locks.*

**True/False about Persistence [3 points each]**

Designate if the statement is True (a) or False (b).

33) When interacting with a device, it is usually better to use **PIO** than **DMA**.

*False; direct memory access (DMA) is usually much better than programmed I/O (PIO) since the CPU does not need to be involved with every word transferred to the device.*

34) **Device driver code** is software that executes on the microcontroller of a peripheral device.

*False; device driver code is typically part of the OS and runs on the main CPU, though its protocols may be specific to each peripheral device.*

35) With a 15000 RPM disk, the **expected rotation time** for a random access is 4 ms.

*False; 4ms is the full rotation time; the expected rotation time for a random access will be ½ of this full amount. 60 sec / 1 min * 1 min / 15000 revs * 1000 ms / 1 sec = 4 ms per full rotation.*

36) **Transfer time** for disk sectors is significantly longer for random accesses than for sequential accesses.

*False. IO time = seek cost + rotation cost + transfer time; transfer time is constant and corresponds to the maximum bandwidth from the device.*

37) **Track skew** results from the fact that the outer tracks of a disk contain more sectors than the inner tracks.

*False; track skew isn't caused by disk zones; track skew accounts for the fact that seeking to a new track will incur some rotation as well.*

38) **SPTF** scheduling over a set of N requests will order the requests resulting in the minimum possible total positioning time.

*False; SPTF is greedy and just picks the NEXT request to minimize the NEXT positioning time; a greedy schedule does not guarantee the minimum possible total positioning time across the SET of N requests.*

39) **SPTF** scheduling is easier to implement inside of a disk than within the OS.

*True; the disk knows the details of its geometry (e.g., exactly on which track each sector is located) as well as the current head position; therefore, it can more accurately predict positioning time, whereas the OS must guess or approximate some of this information.*

40) A disadvantage of SPTF scheduling is that some requests can **starve**.

*True; a request with a long positioning time may not be scheduled.*

41) A disadvantage of the **SCAN** and **C-SCAN** scheduling algorithms are that they ignore the influence of **rotation time** on positioning cost.

*True; SCAN and C-SCAN just schedule based on the track (or cylinder) number.*

42) A **non-work conserving scheduler** may not schedule available requests even when the disk is idle.

*True; this is the definition.*

43) RAID-0 is also referred to as **striping**.

*True; this is the definition.*

44) With **RAID-1,** the steady-state throughput of random reads and random writes are identical.

*False; RAID-1 or mirroring has two copies of each block; since on a write, both copies must be updated, the throughput of writes is ½ that of reads.*

45) RAID-4 has better **capacity** than RAID-1 and better **reliability** than RAID-0.

*True; RAID-4 has just one parity disk for N data disks, whereas RAID-1 has an extra mirror for every data disk; RAID-0 cannot withstand any disk failures, whereas RAID-4 can handle any single disk failing.*

46) One disadvantage of RAID-4 is that it cannot continue operating if the **single parity disk** fails.

*False; RAID-4 can continue operating (and can reconstruct all necessary data) if any disk fails, whether it i a data disk or a parity disk.*

47) The **capacity** of RAID-4 and RAID-5 systems are identical.

*True; both systems have 1 parity block per stripe of data.*

48) The steady-state throughput for **random writes** with RAID-4 is half the steady-state throughput of random writes on a single disk.

*True; random write to RAID-4 involves both reading and writing to the single parity disk (since double the amount of work, this results in half the effective throughput).*

49) For **random read** operations, RAID-1 delivers better throughput than RAID-5.

*False; they each deliver the same performance since each disk in the system can effectively perform random reads.*

50) Given the metrics of **capacity, reliability, and performance**, RAID-5 is strictly better than RAID-4.

*True or False; threw out this question.*

51) In an FFS-like file system, multiple inodes may point to the **same file descriptor**.

*False; inodes don't point to file descriptors.*

52) In an FFS-like file system, multiple inodes may point to the **same path name**.

*False; path names point to inodes, not the other way around.*

53) An advantage of soft links over hard links is that soft links can be used to refer to **directories**.

*True, soft links can be used for directories while hard links cannot.*

54) **Contiguous allocation** of files to blocks on disk tends to achieve excellent sequential bandwidth.

*True; contiguous allocation forces all the blocks of a file to be allocated contiguously, which gives excellent sequential bandwidth.*

55) A **File-Allocation Table** suffers from external fragmentation.

*False; FAT table assumes fixed-sized pages which have internal, not external, fragmentation.*

56) An inode structure typically contains a field indicating the **name** of this file.

*False; inodes do not contain file names since multiple filenames could point to the same inode.*

57) An inode structure typically contains a field indicating the **time** at which this file was last accessed.

*True, they do…*

58) An inode structure can contain **pointers** to **directory data**.

*True; inodes contain pointers to file or directory data; directories are typically stored just like files, but with a special bit set in the inode.*

59) FFS attempts to put the **inode and data blocks** from the same file in the same cylinder group.

*True; this way, good locality between two related structures.*

60) FFS attempts to put **inodes from files in the same directory** in the same cylinder group.

*True; this way good locality for related files.*

61) When placing parts of a **large file** in a "new" group, FFS looks for a group with more than the average number of free inodes.

*False; the large file is put in a new group, but it looks for a group with more than the average number of data blocks, since this large file will probably consume a lot of data blocks (not inodes).*

62) After a crash, **FSCK** fixes the on-disk state of the file system to match its state before the last update to disk began.
*False; FSCK just tries to make the on-disk state consistent; it doesn't know the correct state of the file system before (or after) an particular update.*

63) After the journal **commit** block is successfully written to disk, a journaling file system can then **checkpoint** the relevant blocks to their final in-place positions.
*True, this is what it does.*

64) **Write-back journaling** forces data blocks to be written to their final in-place positions before committing a journal transaction that refers to those data blocks.
*False, this is the definition of ordered journaling; write-back journaling does not force the data blocks to be written first (which can lead to inconsistencies).*

65) When LFS writes a new copy of a **data block** to a segment, it also writes a new copy of the **inode** that points to that data block.
*True; since the data is in a new location in the log, the pointers to that location that are stored in the inode also have to change; LFS does not overwrite inodes (which would be a random write) and instead writes a new copy of the inode to the log.*

66) When LFS writes a new copy of **inode** to a segment, it also writes a new copy of the **directory** that points to that inode.
*False; LFS handles the fact that the location of the inode changes by having an imap to track the current location of each inode.*

67) LFS periodically **checkpoints imaps** to a known location on disk (alternating between two locations to withstand crashes).
*False; LFS checkpoints pointers to portions of the imap in known locations; the modified imaps themselves are written out to each segment.*

68) When performing **garbage collection**, LFS determines that an **inode** is valid by verifying that the corresponding entry in the imap points to this location.
*True, this is what it does*.

69) When performing **garbage collection**, LFS determines that a **data block** is valid by scanning all valid inodes from the imap and verifying that one of the valid inodes points to this location.
*False; this would be way too slow. Instead, LFS writes segment summary info to each segment that describes each updated data block (i.e., the inode that points to it and its offset in the file).*