CS 537: Introduction to Operating Systems
Fall 2016: Midterm Exam #3
Saturday, December 17th, 2017

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.

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

1. The OS manages a single uniprocessor
2. All memory is byte addressable
3. The terminology lg means $\log_2$
4. 2^{10} bytes = 1KB
5. 2^{20} bytes = 1MB
6. Page table entries require 4 bytes
7. Assume leading zeros can be removed from numbers (e.g., 0x06 == 0x6).
8. Data is allocated with optimal alignment, starting at the beginning of a page or disk block
9. No data begins in any cache in the system
10. If the file system is not specified or implied, assume an FFS-based local file system.
11. Assume NFS refers to NFS version 2 (or NFSv2)

There are 128 questions on this exam.

Good luck!

Virtualization Review [1 point each]
Choose the best answer, either (a) or (b), to make each statement as true as possible.

1. A CPU is typically virtualized with **(a) time-sharing** or **(b) space-sharing.**
2. Memory is typically virtualized with **(a) time-sharing** or **(b) space-sharing.**
3. A disk is typically virtualized with **(a) time-sharing** or **(b) space-sharing.**
4. Another name for a thread is a **(a) Lightweight Process** or **(b) Lightweight Procedure.**
5. The primary problem with direct execution is that **(a) it adds too much performance overhead** or **(b) a process could do something that should be restricted.**
6. The CPU scheduler implements a **(a) mechanism** or **(b) policy.**
7. When a process is waiting to use the CPU, the process is in a **(a) ready state** or **(b) blocked state.**
8. A **(a) RR scheduler** or **(b) SJF scheduler** requires knowledge of the expected run length (or CPU burst time) of each job**.**
9. For a reasonable time-slice duration, RR typically delivers **(a) better** or **(b) worse** average turnaround time than FIFO when the jobs in the workload have the same run lengths.
10. It is extremely fast to allocate (and free) memory that has been allocated on the **(a) stack** or **(b) heap**.
11. There is no fragmentation when memory is allocated with a **(a) stack** or **(b) heap**.
12. **(a) Dynamic** or **(b) Static** memory relocation is associated with a base and bounds register.
13. User processes reference their address space with **(a) virtual** or **(b) physical** addresses.
14. Segmentation can suffer from **(a) internal** or **(b) external** fragmentation.
15. With 1 KB pages**, (a) 8** or **(b) 10** bits of the logical address designate the offset within a page.
16. The **(a) upper** or **(b) lower** bits of a logical address designate the offset within a page.
17. With 4 KB pages and a 32-bit logical address, there can be up to **(a) 2^20** or **(b) 2^22** pages in each address space.
18. The starting address of the running process' page table is stored in a **(a) register** or **(b) a known memory location**.
19. A linear page table for a 32-bit address space, with 4 KB pages, and 8 byte page table entries requires about **(a) 4 MB** or **(b) 8 MB** of memory.
20. For a typical workload, the hit rate of a TLB usually **(a) increases** or **(b) decreases** with larger pages.
21. To use a TLB with ASIDs, each **(a) process** or **(b) thread** must be assigned a unique identifier.
22. TLBs with ASIDs typically have **(a) higher** or **(b) lower** hit rates than TLBs without ASIDs.
23. On a TLB **(a) hit** or **(b) miss,** page tables are accessed.
24. A new page table is created when a **(a) new process is created** or **(b) new page is allocated**.
25. A TLB miss can be handled in about **(a) the same time** or **(b) much faster than** a page fault.
26. The clock algorithm requires a **(a) use bit** or **(b) dirty bit**.

Concurrency Review [3 points each]
Choose the best answer, either (a) or (b), to make each statement as true as possible.

27. Threads in the same process share the same **(a) SP** or **(b) PTBR**.
28. Threads in the same process typically share variables that have been allocated on the **(a) stack** or **(b) heap**.
29. When a **(a) user-level** or **(b) kernel-level** thread blocks on I/O, all the threads from that same process are also blocked.
30. When using a shared circular bounded buffer, consumers must wait until there is a **(a) full** or **(b) empty** element.
31. The **(a) progress** or **(b) bounded waiting** requirement for a correct implementation of mutual exclusion states that if there several simultaneous requests, then (at least) one request must allowed proceed.
32. The progress requirement is equivalent to being **(a) deadlock-free** or **(b) starvation-free**.
33. The atomic xchg hardware instruction sets the value of a variable that is stored in **(a) a register** or **(b) memory**.
34. A high-contention spinlock without a yield() system call can lead to wasting the CPU for an amount of time that is proportional to **(a) a time-slice** or **(b) a context-switch**.
35. If a thread calls signal() on a condition variable for which there are no waiting threads, the calling thread **(a) does nothing** or **(b) waits for another thread to arrive**.
36. After a thread returns from the wait() call on a condition variable, that thread **(a) can** or **(b) cannot** assume that it holds the associated mutex lock.
37. Semaphores are initialized to **(a) 0** or **(b) 1** to provide mutual exclusion.
38. With a reader/writer lock in which readers have priority, **(a) only writers** or **(b) both readers and writers** must wait to acquire the lock if a writer currently has the lock.

An Overview of Persistence [3 points each]
Choose the best answer, either (a) or (b), to make each statement as true as possible.

39. When interacting with a slow device, it is usually better to use **(a) PIO** or **(b) DMA**.
40. SPTF scheduling is easier to implement **(a) inside of a disk** or **(b) within the OS**.
41. A **(a) non-work conserving scheduler** or **(b) work-conserving scheduler** may not schedule available requests even when the disk is idle.
42. RAID-0 is also referred to as **(a) striping** or **(b) mirroring**.
43. With RAID-1, the steady-state throughput of random writes is **(a) identical to** or **(b) less than** that of random reads.

44. RAID-5 has better **(a) capacity** or **(b) reliability** than RAID-1.
45. RAID-5 has better **(a) capacity** or **(b) reliability** than RAID-0.
46. Given a RAID-5 storage system, a large number of disks, and a random write of a block, the most efficient way to update the corresponding parity block involves **(a) reading the data blocks belonging to that stripe** or **(b) reading the old data block and the old parity block**.
47. For workloads generated by LFS, **(a) RAID-1** or **(b) RAID-5** delivers higher performance.
48. A workload with **(a) sequential reads** or **(b) random reads** obtains the same performance on RAID-0 and RAID-5.
49. Creating a **(a) soft link** or **(b) hard link** increments the reference count within an inode.
50. File descriptors are used instead of path names to refer to specific files primarily to **improve (a) performance** or **(b) user convenience**.
51. The **File-Allocation Table** is most directly an optimization of **(a) linked allocation** or **(b) extent-based allocation**.
52. In an FFS-based file system, directories are distinguished from files by information stored in **(a) the directory entries that named them** or **(b) their corresponding inode**.
53. A directory **(a) can** or **(b) cannot** use an indirect block.
54. When creating a new file "bar" in an existing directory "/foo", the **(a) data bitmap** or **(b) inode bitmap** is updated.
55. When creating a new file "bar" in an existing directory "/foo", the disk block containing the inode for "bar" is **(a) written** or **(b) read and written**.
56. When creating a new file "bar" in an existing directory "/foo", foo's data must be **(a) written** or **(b) read and written**.
57. When appending x bytes to an opened file "/test" whose last data block contains space for x bytes, test's inode must be **(a) read** or **(b) read and written**.
58. When a file "/foo/bar" is closed, bar's inode is **(a) not modified** or **(b) written**.
59. As the original FFS aged, it obtained **(a) better** or **(b) worse** performance.
60. In FFS, **(a) appending to** or **(b) overwriting existing data blocks in** a file could cause other existing data in that file to be moved on the disk.
61. When creating a new file, FFS allocates the inode for that file in **(a) a new cylinder group with an above average number of free inodes** or **(b) in the same cylinder group as the parent directory**.
62. After a crash, **FSCK** repairs the on-disk state of the file system to **(a) match its state before the last update to disk began** or **(b) be consistent**.
63. With full data journaling, data and meta-data blocks that are **(a) written** or **(b) read or written** are placed in the journal.

64. **(a) Write-back** or **(b) Ordered** journaling forces data blocks to be written to their final in-place positions before committing a journal transaction that refers to those data blocks.
65. LFS is an example of a **(a) journal-based** or **(b) copy-on-write** file system.
66. LFS optimizes the performance of **(a) writes** or **(b) reads** that are sent to disk.
67. When an existing data block of a file is updated (i.e., overwritten), LFS writes that data block to **(a) the old** or **(b) a new location** on disk.
68. In LFS, directory entries contain filenames and **(a) inode numbers** or **(b) the block addresses of the corresponding inodes**.
69. In LFS, the imap structure is written to **(a) segments** or **(b) checkpoint regions in known locations.**
70. When cleaning old segments, LFS determines that an inode is valid by **(a) looking up this inode in the imap and seeing if it points to the same location** or **(b) verifying if any directory entry contains a reference to this inode.**
71. When cleaning old segments, LFS **(a) may** or **(b) may not** move valid inodes to a new segment.
72. With UDP, a packet may be lost between the sending process and the receiving process and as a result **(a) the receiving process may never receive that message** or **(b) the sending process will retransmit the message.**
73. With UDP, a message may be corrupted between the sending process and the receiving process and as a result **(a) the receiving process may receive a corrupted message** or **(b) the receiving process may never receive the message.**
74. With TCP, messages are delivered in order to the receiver by **(a) sorting packets on the receiver based on their sequence numbers** or **(b) requiring that each packet is acknowledged by the receiver before the next packet is sent**.
75. Servers in **(a) NFS** or **(b) AFS** track state related to currently open files.
76. In NFS, writes must be flushed to the server **(a) every 3 seconds** or **(b) when the file is closed.**
77. In AFS, a client will see changes to a file that was changed on the server **(a) when the client next opens that file** or **(b) when the client next reads a block of that file.**
78. Scalability is improved in AFS by **(a) reducing the interactions between clients and servers** or **(b) allowing clients to obtain updates from other clients.**
79. In **(a) NFS** or **(b) AFS**, the client is responsible for mapping local file descriptors to file handles that are exchanged with the server.
80. In GFS, when a chunkserver crashes, **(a) the number of replicas for a given chunk may be temporarily reduced** or **(b) it notifies the master to make additional copies of its chunks.**
81. In GFS, a client interacts with the master whenever **(a) the client opens a file** or **(b) reads the next chunk from a file.**

82. To improve scalability, GFS attempts to reduce the interactions between **(a) the client and the master** or **(b) the client and chunkservers.**

83. To detect data corruption in a GFS chunk, **(a) each chunkserver** or **(b) the master** stores a checksum for the chunk.

The File System API: Links. [3 points each]
Imagine the following commands are run on an FFS-like file system that supports hard links. Note: the symbols ">>" are used to redirect stdout to append to the specified file (creating that file if it doesn't already exist).

```
echo "file1" >> file1
ln file1 file2
echo "file2" >> file2
rm file1
echo "file2 again" >> file2
echo "file1 again" >> file1
```

84. What output will you see if you run `cat file2` (assume nothing wrong with whitespace or carriage returns)?
    a) file2\n file2 again\n
    b) file1\n file2\n file2 again\n
    c) file1\n file2\n file2 again\n file1 again\n
    d) file2: No such file or directory
    e) None of the above

85. What output will you see if you run `cat file1` (assume nothing wrong with whitespace or carriage returns)?
    a) file1 again\n
    b) file1\n file1 again\n
    c) file2 again\n file 1 again\n
    d) file1\n file2\n file2 again\n file1 again\n
    e) None of the above

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.

86. File System State:

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

a) mkdir("/a");
b) creat("/a");
c) creat("/a"); fd=open("/a"); write(fd, buf, BLOCKSIZE); close(fd);
d) mkdir("/f/a");
e) None of the above

87. File System State:

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

a) creat("/a/d");
b) creat("/d");
c) mkdir("/d");
d) fd=open("/a"); write(fd, buf, BLOCKSIZE); close(fd);
e) None of the above

88. File System State:

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

a) mkdir("/t");
b) creat("/t"); fd=open("/t"); write(fd, buf, BLOCKSIZE); close(fd);
c) creat("/t"); unlink("/a");
d) mkdir("/t"); unlink("/a");
e) None of the above

89. File System State:

```
inodes         [d a:0 r:4] [f a:1 r:1] [d a:2 r:2] [] [] [] [] []
data           [(.,0) (..,0) (t,2) (h,1)] [o] [(.,2) (..,0)] [] [] [] [] []
```

a) mkdir("/h"); creat("/h/o");
b) creat("/h"); fd=open("/h"); write(fd, buf, BLOCKSIZE); close(fd);
c) create("/h");
d) mkdir("/h"); mkdir("/h/o");
e) None of the above.

File system Data Structures and Crash Consistency [26 points total]
Imagine you have an FFS-like file system that is creating a new empty file in an existing directory and must update 6 blocks: the directory inode, two directory data blocks, the file inode, the inode bitmap, and the data bitmap. Assume the directory inode and the file inode are in different on-disk blocks.

90. What is a scenario for which these 6 disk blocks would need to be writen?
    a) The new file is a hard link to an existing file
    b) The size of the directory has grown to require another data block to hold the directory entries
    c) No free data blocks were found in the current cylinder group
    d) No free inodes were found in the current cylinder group
    e) None of the above

Assume this filesystem does not perform any journaling and FSCK is not run. What happens if a crash occurs after only updating the following block(s)?

91. Inode 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

92. 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

93. All bitmaps, directory inode, and all 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

94. File inode, data bitmap, directory inode, and all 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

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 that updates 6 blocks. 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.)

95. 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
96. 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
97. 7 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
98. 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
99. 9 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
100. 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
101. 14 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
102. 15 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

Distributed File System Consistency [25 points total]
The next questions explore the cache consistency behavior of AFS and NFS**.**

The two traces on the next two pages are identical; you should use one trace for answering how AFS behaves and one trace for answering how NFS behaves.

Each trace contains two clients that each generate file opens, reads, writes, and closes on a single file 'a'. The leftmost column shows the server; the next 2 columns show the actions being taken on each of the two clients, c0 and c1. The content of the file is always just a single number. Time increases downwards, **with at least 5 seconds between each operation.**

Opening a file returns a file descriptor, which is the first argument to each call of read, write, and close (i.e., read:fd, write:fd, and close:fd). The write call also designates the new value to be written (i.e., write:fd -> newvalue).

There are two types of questions for you to answer. Questions of the format "`read:fd -> value?`" on c0 and c1 ask you to determine the value that will be read on that client. Questions of the format "`file:a contains: ?`" on the server ask you to determine the value on the server at that point in time.

You may find it useful to record the contents of the file on the server at every "interesting" point in time.

For each protocol, assume that the server and clients have sufficient memory such that no operations are performed unless **required** by the protocol.

Questions 103-115: AFS Protocol

For questions 103-115, your choices are:

a) 0
b) 2
c) 3
d) 5
e) None of the above

Determine the results if this workload is run on top of the **AFS** distributed file system.

```
Server                  c0                          c1
file:a contains:0
                                                    open:a [fd:0]

                        open:a [fd:0]

                                                    read:0 -> value(Q103)
                                                    write:0 -> 2

                        read:0 -> value?(Q104)
                        write:0 -> 3
file:a contains:?(Q105)
                                                    read:0 -> value(Q106)
                                                    close:0
file:a contains:?(Q107)

                                                     open:a [fd:1]
                                                    read:1 -> value(Q108)


                        close:0
file:a contains:?(Q109)

                        open:a [fd:1]

                        read:1 -> value?(Q110)
                        write:1 -> 0


                        close:1
file:a contains:?(Q111)

                                                    read:1 -> value?(Q112)
                                                    write:1 -> 5

                        open:a [fd:2]


                        read:2 -> value?(Q113)

                                                    close:1

                        read:2 -> value?(Q114)
                        close:2
file:a contains:?(Q115)
```

Questions 116-128: NFS Protocol

For questions 116-128, your choices are:

a) 0
b) 2
c) 3
d) 5
e) None of the above

Determine the results if this workload is run on top of the **NFS** distributed file system.

```
Server                  c0                          c1
file:a contains:0
                                                    open:a [fd:0]

                        open:a [fd:0]

                                                    read:0 -> value(Q116)
                                                    write:0 -> 2

                        read:0 -> value?(Q117)
                        write:0 -> 3
file:a contains:?(Q118)
                                                    read:0 -> value(Q119)
                                                    close:0
file:a contains:?(Q120)

                                                     open:a [fd:1]
                                                    read:1 -> value(Q121)


                        close:0
file:a contains:?(Q122)

                        open:a [fd:1]

                        read:1 -> value?(Q123)
                        write:1 -> 0


                        close:1
file:a contains:?(Q124)

                                                    read:1 -> value?(Q125)
                                                    write:1 -> 5

                        open:a [fd:2]


                        read:2 -> value?(Q126)

                                                    close:1

                        read:2 -> value?(Q127)
                        close:2
file:a contains:?(Q128)
```