CS 537 - Introduction to Operating Systems
Fall 2019 Instructor: Andrea Arpaci-Dusseau

Exam 3 — Final Exam: Persistence

Fill in these fields on the scantron form (use #2 pencil):

1. LAST NAME (surname) and FIRST NAME (given name), fill in bubbles
2. IDENTIFICATION NUMBER is your Campus ID number, fill in bubbles
3. Under *F* of SPECIAL CODES, write **7** and fill in bubble **7**

If you miss step 3 above (or do it wrong), the system may not grade you against the correct answer key, and your grade will be no better than if you were to randomly guess on each question. So don't forget!

You may not use notes, books, your neighbors, calculators, or other electronic devices on this exam. Turn off and put away portable electronics.

Use a #2 pencil to mark all answers. When you're done, please hand in these sheets in addition to your filled-in scantron.

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

- Assume leading zeros can be removed from numbers (e.g., 0x06 == 0x6).
- All memory is byte addressable
- The terminology lg means log2
- 2^{10} bytes = 1KB
- 2^{20} bytes = 1MB
- Page table entries require 4 bytes
- Data is allocated with optimal alignment, starting at the beginning of a page
- Persistent storage is provided by a single hard disk drive (HDD).
- The file system is similar to that of FFS.
- The OS is POSIX compliant.
- Journaling is full data journaling.
- NFS refers to NFS version 2 (NFSv2).
- If words are misspelled, assume it is a typo; if there is discrepancy in capitalization or whitespace (including carriage returns), assume it is a typo.

File System API (6 points)

Consider the following commands sent to a UNIX-based system:

```
Line 1: echo 'hello' >> file1
Line 2: ln file1 file2
Line 3: cat file2
Line 4: rm file1
Line 5: cat file2
Line 6: echo 'more' >> file1
Line 7: cat file2
```

Ignore any problems with whitespace or capitalization. Assume ; shows text separated across multiple lines in the shown output.

1. What will be the output from Line 3?

    A. hello
    B. more
    C. more ; hello
    D. hello ; more
    E. ERROR: No such file or directory

2. What will be the output from Line 5?

    A. hello
    B. more
    C. more ; hello
    D. hello ; more
    E. ERROR: No such file or directory

3. What will be the output from Line 7?

    A. hello
    B. more
    C. more ; hello
    D. hello ; more
    E. ERROR: No such file or directory

Assume the `ln` command in Line 2 is changed to `ln -s`.

4. What will be the output from Line 3?

    A. hello
    B. more
    C. more ; hello
    D. hello ; more
    E. ERROR: No such file or directory

5. What will be the output from Line 5?

    A. hello
    B. more
    C. more ; hello
    D. hello ; more
    E. ERROR: No such file or directory

6. What will be the output from Line 7?

    A. hello
    B. more
    C. more ; hello
    D. hello ; more
    E. ERROR: No such file or directory

File System Images (16 points)

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:

```
inodes       [d a:0 r:6] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] ...
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.

For the following questions, begin by assuming you have a file system image in the following state:

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

7. What is /g?

   A. File
   B. Directory
   C. Does not exist in this file system image
   D. Not enough information to determine
   E. None of the above

8. What is /q?

   A. File
   B. Directory
   C. Does not exist in this file system image
   D. Not enough information to determine
   E. None of the above

9. What is /g/d?

   A. File
   B. Directory
   C. Does not exist in this file system image
   D. Not enough information to determine
   E. None of the above

Assume you now execute the command `link(‘‘/u’’, ‘‘/x’’);`

10. What could be the resulting state of the file system image for **inodes**?

    A. No change (includes if the operation was invalid)
    B. `[d a:0 r:6] [d a:1 r:2] [f a:-1 r:2] [f a:-1 r:1] [] [] [] []`
    C. `[d a:0 r:6] [d a:1 r:2] [f a:-1 r:1] [f a:-1 r:2] [] [] [] []`
    D. `[d a:0 r:5] [d a:1 r:2] [f a:-1 r:1] [f a:-1 r:1] [f a:-1 r:1] [] [] []`
    E. None of the above

11. What could be the resulting state of the file system image for **data blocks**?

    A. No change (includes if the operation was invalid)
    B. `[(.,0) (..,0) (g,1) (q,2) (u,3) (x,3)] [(.,1) (..,0)] [] [] [] [] [] []`
    C. `[(.,0) (..,0) (g,1) (q,2) (u,3) (x,4)] [(.,1) (..,0)] [] [] [] [] [] []`
    D. `[(.,0) (..,0) (g,1) (q,2) (u,3)] [(.,1) (..,0) (x,4)] [] [] [] [] [] []`
    E. None of the above

Now, assume you have a different file system image:

```
inodes        [d a:0 r:4] [d a:1 r:3] [f a:-1 r:2] [] [] [] [] []
data          [(.,0) (..,0) (e,2) (o,1)] [(.,1) (..,0) (s,2)] [] [] [] [] [] []
```

12. For this file system image, which two pathnames can be used to access the same file?

    A. /o and /s
    B. /o and /e/s
    C. /e and /o
    D. /e and /o/s
    E. None of the above OR more than one of the above

Assume you now execute the following commands:

```
fd=open(``/e'', O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);
```

13. What could be the resulting state of the file system image for **inodes**?

    A. No change (includes if the operation was invalid)
    B. `[d a:0 r:4] [d a:1 r:3] [f a:-1 r:3] [] [] [] [] []`
    C. `[d a:0 r:4] [d a:1 r:3] [f a:1 r:2] [] [] [] [] []`
    D. `[d a:0 r:4] [d a:1 r:3] [f a:2 r:2] [] [] [] [] []`
    E. None of the above OR more than one of the above

14. What could be the resulting state of the file system image for **data blocks**?

    A. No change (includes if the operation was invalid)
    B. `[(.,0) (..,0) (e,2) (o,1)] [(.,1) (..,0) (s,2) (e,1)] [] [] [] [] [] []`
    C. `[(.,0) (..,0) (e,2) (o,1)] [(.,1) (..,0) (s,2)] [a] [] [] [] [] []`
    D. `[(.,0) (..,0) (e,2) (o,1)] [(.,1) (..,0) (s,2)] [(.,1) (..,0)] [] [] [] [] []`
    E. None of the above OR more than one of the above

File System Data Structures (16 points)

This question asks you about the read and write operations that a file system like FFS must send to disk to update its meta-data and data blocks. You should make the following assumptions:

- Disk blocks are 4 KB
- Nothing begins in the file system buffer cache
- All modified data must be written to the disk (and not just the buffer cache)
- All directory data always fits within 4 KB
- No needed inode is ever in the same block as another needed inode
- No timestamps are updated
- The operations cause no errors (e.g., specified directories exist, files to be created do not already exist)
- There is no journaling or other mechanisms for crash consistency

Assume that you would like to create a new (empty) file `bar` with the pathname `/dir/dir2/bar`. Think about the read and write operations that will be required; it may be useful to write them all out to answer the following questions.

15. How many disk blocks will be **read** to create the file?

    A. 5 B. 6 C. 7 D. 8 E. None of the above

16. How many disk blocks will be **written** to create the file?

    A. 3 B. 4 C. 5 D. 6 E. None of the above

How must the disk blocks containing each of the following file system data structures be accessed to create the file? Possible options are the block A) must only be read, B) must only be written, C) must be both read and written, D) does not need to be accessed. There is no E) option.

17. The block containing the data for dir must be:

    A. read only B. written only C. read and written D. nothing

18. The block containing the inode for dir2 must be:

    A. read only B. written only C. read and written D. nothing

19. The block containing the data for dir2 must be:

    A. read only B. written only C. read and written D. nothing

20. The block containing the inode for bar must be:

    A. read only B. written only C. read and written D. nothing

21. The block containing the data for bar must be:

    A. read only B. written only C. read and written D. nothing

22. A block containing a portion of the inode bitmap must be:

    A. read only B. written only C. read and written D. nothing

23. A block containing a portion of the data bitmap must be:

    A. read only B. written only C. read and written D. nothing

Assume that you would now like to open an existing file `foo` with the pathname `/dir/dir2/foo`; to open the file, you must read the file's inode into memory. Again, assume nothing begins cached in the file system buffer cache. Think about the read and write operations that will be required; it may be useful to write them all out to answer the following questions.

24. How many disk blocks will be **read** to open the file?

    A. 4 B. 5 C. 6 D. 7 E. None of the above

25. How many disk blocks will be **written** to open the file?

    A. 1 B. 2 C. 3 D. 4 E. None of the above

How must the disk blocks containing each of the following file system data structures be accessed to open the file?

26. The block containing the inode for dir2 must be:

    A. read only B. written only C. read and written D. nothing

27. The block containing the data for dir2 must be:

    A. read only B. written only C. read and written D. nothing

28. The block containing the inode for bar must be:

    A. read only B. written only C. read and written D. nothing

29. The block containing the data for bar must be:

    A. read only B. written only C. read and written D. nothing

30. A block containing a portion of the inode bitmap must be:

    A. read only B. written only C. read and written D. nothing

31. A block containing a portion of the data bitmap must be:

    A. read only B. written only C. read and written D. nothing

Data Journaling with Barriers (10 points)

You are using a journaling file system (such as Linux ext3) in full data-journaling mode without the checksum performance optimization (i.e., the transaction commit block does NOT contain a checksum over the blocks of that transaction).

As described in class, each transaction is composed of a single transaction header block (also called transaction descriptor block), multiple meta-data and data blocks in the transaction, and a single transaction commit block; there is also an in-place checkpoint region on the disk.

As you know, some of the transaction blocks and some blocks to the checkpoint region can be written out concurrently to disk. Other blocks cannot be written concurrently: to guarantee crash consistency, the file system must ensure that particular blocks (e.g., A), are persisted to disk before others (e.g., B). Thus, the file system flushes A to disk and sends a barrier operation to the disk before sending B.

What crash consistency errors could occur if the journaling file system does not send a barrier to the disk between the following types of blocks for A and B? Note that a barrier may not be needed between A and B. In each question, assume all other barriers are inserted correctly; that is, each question is independent. Ignore any performance problems.

32. A = Transaction header block; B = First transaction meta-data/data block

    A. No correctness problem occurs
    B. Could replay garbage located in transaction to checkpoint region
    C. Could partially overwrite in-place checkpoint data and not replay transaction
    D. Other correctness problem

33. A = First meta-data/data block in transaction; B = Other meta-data/data blocks in transaction

    A. No correctness problem occurs
    B. Could replay garbage located in transaction to checkpoint region
    C. Could partially overwrite in-place checkpoint data and not replay transaction
    D. Other correctness problem

34. A = Last meta-data/data block in transaction; B = Transaction commit block

    A. No correctness problem occurs
    B. Could replay garbage located in transaction to checkpoint region
    C. Could partially overwrite in-place checkpoint data and not replay transaction
    D. Other correctness problem

35. A = Transaction commit block; B = Blocks to in-place checkpoint region

    A. No correctness problem occurs
    B. Could replay garbage located in transaction to checkpoint region
    C. Could partially overwrite in-place checkpoint data and not replay transaction
    D. Other correctness problem

36. A = Blocks to in-place checkpoint region; B = Transaction commit block

    A. No correctness problem occurs
    B. Could replay garbage located in transaction to checkpoint region
    C. Could overwrite in-place checkpoint data and not replay transaction
    D. Other correctness problem

Distributed File Systems (20 points)

The next questions explore the cache consistency behavior of AFS and NFS.

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

Each trace contains two clients that each generate file opens, reads, writes, and closes on a single file. The 2 columns show the actions being taken on each of the two clients, c0 and c1. Time increases downwards in seconds. Assume the time required to send/receive messages over the network and for the server to respond is negligible.

Opening a file returns a file descriptor which would be used for each call to read, write, and close, but is not shown because only a single file is involved. The file contains multiple blocks. The arguments to read and write show the block number of the file that is being accessed. Note that it does not matter what data is actually read or written.

For each protocol, assume that the server and clients have sufficient memory and disk space such that no operations are performed unless required by the protocol. NFS Hint: Every interaction with the server returns the current attributes for the requested file.

NFS

Time (s)	c0	c1
0.0	open	
1.0	read(1)	
2.0	read(2)	
3.0	read(3)	
10.0	read(2)	
11.0	read(3)	
12.0	write(3)	
13.0		open
14.0		read(3)
15.0		read(1)
16.0		write(1)
17.0		close
18.0	read(1)	
19.0	read(3)	
20.0	write(3)	
21.0		open
22.0		read(3)
23.0		close
24.0	close	
25.0		open
26.0		read(3)
27.0		close

At the time in question, what operation must the NFS client perform?
For the read and open operations, your choices are:

A. Operations performed locally on client machine (i.e., no interaction with server)
B. Send getattr()/stat() request to server; then use local copy of data
C. Send getattr()/stat() request to server; then obtain block from server
D. Obtain block from server (do not need to first send getattr()/stat() request)
E. Obtain whole file from server (do not need to first send getattr()/stat() request)

For the write and close operations, your choices are:

A. Write local copy of block (with no interaction with server)
B. Write local copy of block and also send block to server
C. Send all file changes (or whole file) to server
D. No writing of local copy and no interacting with server

37. Time 1: Read (Client c0 reads block 1 of file)
38. Time 10: Read
39. Time 11: Read
40. Time 12: Write
41. Time 17: Close
42. Time 18: Read
43. Time 19: Read
44. Time 26: Read

AFS

Time (s)	c0	c1
0.0	open	
1.0	read(1)	
2.0	read(2)	
3.0	read(3)	
10.0	read(2)	
11.0	read(3)	
12.0	write(3)	
13.0		open
14.0		read(3)
15.0		read(1)
16.0		write(1)
17.0		close
18.0	read(1)	
19.0	read(3)	
20.0	write(3)	
21.0		open
22.0		read(3)
23.0		close
24.0	close	
25.0		open
26.0		read(3)
27.0		close

At the time in question, what operation must the AFS client perform?
For the read and open operations, your choices are:

A. Operations performed locally on client machine (i.e., no interaction with server)
B. Send getattr()/stat() request to server; then use local copy of data
C. Send getattr()/stat() request to server; then obtain block from server
D. Obtain block from server (do not need to first send getattr()/stat() request)
E. Obtain whole file from server (do not need to first send getattr()/stat() request)

For the write and close operations, your choices are:

A. Write local copy of block (with no interaction with server)
B. Write local copy of block and also send block to server
C. Send all file changes (or whole file) to server
D. No writing of local copy and no interacting with server

45. Time 0: Open (Client c0 opens file)
46. Time 1: Read (Client c0 reads block 1 of file)
47. Time 10: Read
48. Time 12: Write
49. Time 17: Close
50. Time 18: Read
51. Time 19: Read
52. Time 21: Open
53. Time 25: Open

Persistence (1 point per question)

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

I/O Devices including HDDs

54. The I/O bus is usually able to transfer bytes at a higher bandwidth than the memory bus.
55. For handling I/O, polling can be faster than using interrupts.
56. On a system with DMA, the main CPU transfers data from RAM to the peripheral device.
57. To perform a read from hard disk drive, first the desired sector rotates under the disk head and then the disk head seeks to the correct track.
58. For random I/Os on an HDD, one can accurately model the expected seek distance as 1/3 the maximum seek distance.
59. Track skew is used to adjust for the fact that fewer sectors can be placed in inner tracks than outer tracks of a hard disk (with constant density recording).
60. The OS can have multiple outstanding requests to each disk in the system.
61. The SCAN and C-SCAN algorithms help avoid starvation.
62. A work-conserving scheduler always schedules work if it is available.

RAID

63. RAID stands for "Reliable Array of Independent (or Inexpensive) Disks"
64. RAID-0 cannot recover from any failed disk.
65. RAID-0 with $N > 1$ disks provides better sequential write throughput than a single disk.
66. RAID-1 provides better capacity than RAID-5 (for a large number of disks).
67. RAID-1 returns the correct data if one disk has silent data corruption.
68. With RAID-1, updating a random block requires updating two copies of that block.
69. RAID-4 with $N >= 4$ disks provides better throughput for random writes than a single disk
70. With RAID-5, a large number of disks, and a sequential write workload, the most efficient way to update parity is to read the old value of the data and parity.
71. The throughput achieved for a random read workload is equal with RAID-0 and RAID-5.

FS API and Implementation

72. An inode tracks the current offset to read or write the corresponding file.
73. Directories store mappings between names and inode numbers.
74. Inodes for directories may contain pointers to indirect blocks.
75. Directory entries indicate whether each entry refers to a file or a directory.
76. When a process closes a file, any writes to that file that have been buffered in memory are flushed to disk.
77. The rename operation allocates a new inode for the new file name.
78. Assuming 4KB blocks, 4 byte pointers, and inodes containing pointers to 10 direct blocks, 1 indirect block, and 1 double indirect block, the maximum size of a file is 10*4KB + 1K*4KB + 2K*4KB.
79. The reference count for a directory can be more than 1.
80. When a file is unlinked, the corresponding directory entry is removed unless the reference count remains > 0.

FFS

81. The original version of the Fast File System (FFS) achieved worse performance as it aged.
82. In general, increasing the size of disk blocks increases performance, but decreases internal fragmentation.
83. The improved version of FFS placed inodes near their corresponding portion of the bitmap.
84. The improved version of FFS tried to place file data blocks near their corresponding inode.
85. The improved version of FFS tried to place directory data blocks near their corresponding inode.
86. The improved version of FFS tried to place files near other files in the same directory.
87. The improved version of FFS obtained the best performance when applications performed "fragment-sized" writes.

Consistency and Journaling

88. A file system could be in a consistent state if a valid inode points to a data block that is marked free in the data bitmap.
89. A file system could be in a consistent state if two directory entries point to the same valid inode.
90. A file system could be in a consistent state if two inodes point to the same data block.
91. To guarantee atomicity of file system operations, the OS ensures that no crashes or power failurs occur while it modifies on-disk structures.
92. During checkpointing, the contents of a transaction are read from the on-disk journal and written out to their in-place (on-disk) locations.
93. Writing to the on-disk journal is usually faster than writing to the metadata and data's in-place locations on disk.
94. With ordered-mode meta-data journaling, the file system guarantees that data blocks are updated in their in-place locations before the transaction is written to the journal.

LFS

95. LFS first writes all updates to a log and then writes those updates to their in-place locations on disk.
96. LFS was designed to work well with RAID-5.
97. When a data block is updated in LFS, the inode that points to that data block must be updated as well.
98. When a data block is updated in LFS, the inode that points to that data block is written out to the current segment as well.
99. When a data block is updated in LFS, a portion of the imap must be updated as well.
100. When an inode is updated in LFS, the directory that refers to that inode must be updated as well.
101. When an inode is updated in LFS, the directory that refers to that inode must be written out to the current segment as well.
102. When an inode is updated in LFS, a portion of the imap must be updated as well.
103. Updates to data blocks in LFS force changes all the way up to the root of the directory tree.
104. An inode is alive if the imap contains a pointer to this inode.
105. A data block is alive if the segment summary contains a pointer to this data block.

Flash and SSDs

106. With Flash, clients can read in parallel from different channels.
107. With Flash, a block must be in the erased state before a page can be programmed.
108. Single-level-cell (SLC) Flash is more robust and has a longer lifetime than multi-level-cell (MLC) Flash.
109. The time required to program a flash page is similar to that for writing to an HDD sector.
110. An FTL (Flash Translation Layer) tracks which pages belong to which files in the file system.
111. Clients should access an SSD with locality so that page translations hit in the translation cache of the FTL.
112. One of the benefits of a hybrid-mapping FTL is that the FTL tracks fewer mappings than a block-level FTL.
113. Aligned sequential reads perform better than unaligned sequential accesses because the FTL can perform switch merges.
114. FTLs must perform wear leveling because some of the pages in a block may be dead while other pages in that block are not.

Distributed File Systems

115. In the NFSv2 protocol, the client machine specifies the offset for each read and write operation sent to the server.
116. In the NFSv2 protocol, the File Handle returned by open() to the client machine contains the file descriptor stored on the server machine.
117. An application process cannot use the append() system call on an NFSv2 file.
118. In NFSv2, blocks written on a client machine are flushed to the server every 3 seconds.
119. In NFSv2, read() system calls are peformed completely locally on a client machine and do not require interaction with the server.
120. In NFSv2, the server tracks the time each client last made a stat (or getattr) call for each file.
121. In AFS, client A will not see changes made by an other client machine to a file that client A has already opened.
122. In AFS, read() system calls are performed completely locally on a client machine and do not require interaction with the server.

Virtualization (2 points each)

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

123. On a timer interrupt, hardware is responsible for transitioning from user to kernel mode.
124. Increasing the time-slice of an RR scheduler is likely to increase the overhead imposed by scheduling.
125. A Gantt chart illustrates how jobs are scheduled over time
126. With dynamic relocation, the OS determines where the address space of a process is allocated in virtual memory.
127. With dynamic relocation, the OS can move an address space after it has been placed in physical memory.
128. With pure segmentation (with no paging or TLBs), each segment of each process must be allocated contiguously in physical memory.
129. With pure segmentation (with no paging or TLBs), fetching an instruction will involve exactly two memory references.
130. Threads running within the same process are likely to share the same value for the base register, but not the bounds register, in the MMU.
131. Given a constant number of bits in a virtual address, the size of a linear page table increases with more (virtual) pages.
132. Given a fixed number of virtual pages, the size of a linear page table decreases with a smaller address space.
133. Given a two-level page table, there may be two TLB misses when performing a virtual to physical address translation.
134. If the TLB does not support ASIDs, all page table entries (PTEs) are marked invalid on a context switch.
135. With a linear page table, all virtual pages within an address space must be allocated.
136. LRU with N+1 physical pages will contain (cache) the same virtual pages as LRU with N pages, plus the contents of one more virtual page.
137. Demand paging loads the address space of a process into physical memory when that process is started

Concurrency (2 points each)

138. A problem with user-level thread implementations is that if one thread in the application process blocks, then all the threads in that application process are blocked as well.
139. Locks prevent the OS scheduler from performing a context switch during a critical section.
140. The volatile key word before a variable directs the compiler to protect that variable with locks.
141. To minimize wasted CPU cycles on a multiprocessor system, a lock implementation should block instead of spin if the lock will be held by another process for longer than the time required for a context-switch.
142. A ticket lock must atomically increment the ticket number that a thread is given when it tries to acquire the lock.
143. Two-phase waiting assumes that an oracle can predict the amount of time that the lock will be held by another process.
144. When a thread calls cond_wait() on a condition variable, cond_wait() returns immediately if a cooperating thread previously called cond_signal() on that condition variable.
145. The call cond_wait() must release a corresponding mutex.
146. Condition variables can provide both ordering and mutual exclusion.
147. To implement a thread_join() operation with a semaphore, the semaphore is initialized to the value of 0 and the thread_join() code calls sem_wait().
148. With producer/consumer relationships and a circular shared buffer of N elements, producing threads must wait for a consumer when there are no empty elements in the buffer.
149. The safety property for dining philosophers states that no two adjacent philosophers are eating simultaneously.
150. With a reader/writer lock, multiple readers can hold the lock simultaneously.
151. With a wait-free algorithm, a thread that cannot acquire a lock must release any other locks that the thread holds.
152. Deadlock can be avoided if locks are always acquired in a well-defined circular order.