Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: File systems

1. Provide one reason why a DMA-enabled device driver usually gives better performance over a non-DMA interrupt-driven device driver.

   Ans: A DMA driver frees up CPU cycles that would have been spent copying data from the device to physical memory.

2. Which of the following statements is/are true regarding memory-mapped I/O?

   A. The CPU accesses the device memory much like it accesses main memory.

   B. The CPU uses separate architecture-specific instructions to access memory in the device.

   C. Memory-mapped I/O cannot be used with a polling-based device driver.

   D. Memory-mapped I/O can be used only with an interrupt-driven device driver.

   Ans: A

3. Consider a file D1/F1 that is hard linked from another parent directory D2. Then the directory entry of this file (including the filename and inode number) in directory D1 must be exactly identical to the directory entry in directory D2. [T/F]

   Ans: F (the file name can be different)

4. It is possible for a system that uses a disk buffer cache with FIFO as the buffer replacement policy to suffer from the Belady's anomaly. [T/F]

   Ans: T

5. Reading files via memory mapping them avoids an extra copy of file data from kernel space buffers to user space buffers. [T/F]

   Ans: T

6. A soft link can create a link between files across different file systems, whereas a hard link can only create links between a directory and a file within the same file system. [T/F]

   Ans: T (becasue hard link stores inode number, which is unique only within a file system)

7. Consider the process of opening a new file that does not exist (obviously, creating it during opening), via the "open" system call. Describe changes to all the in-memory and disk-based file system structures (e.g., file tables, inodes, and directories) that occur as part of this system call implementation. Write clearly, listing the structure that is changed, and the change made to it.

   Ans: (a) New inode allocated on disk (with link count=1), and inode bitmap updated in the process. (b) Directory entry added to parent directory, to add mapping from file name to inode number.

(c) In-memory inode allocated. (d) System-wide open file table points to in-memory inode. (e) Per-process file descriptor table points to open file table entry.

8. Now, suppose the process that has opened the file in the previous question proceeds to write 100 bytes into the file. Assume block size on disk is 512 bytes. Assume the OS uses a write-through disk buffer cache. List all the operations/changes to various datastructures that take place when the write operation successfully completes.

   Ans: (a) open file table offset is changed (b) in-memory and on-disk inode adds pointer to new data block, and last modified time is updated (c) a copy of the data block comes into the disk buffer cache (d) New data block is allocated from data block bitmap (e) new data block is filled with user provided data

9. Repeat the above question for the implementation of the "link" system call, when linking to an existing file (not open from any process) in a directory from another new parent directory.

   Ans: (a) The link count of the on-disk inode of the file is incremented. (b) A directory entry is added to the new directory to create a mapping from the file name to the inode number of the original file (if the new directory does not have space in its data blocks for the new file, a new data block is allocated for the new directory entry, and a pointer to this data block is added from the directory's inode).

10. Repeat the above question for the implementation of the "dup" system call on a file descriptor.

    Ans: To dup a file descriptor, another empty slot in the file descriptor table of the process is found, and this new entry is set to point to the same global open file table entry as the old file descriptor. That is, two FDs point to same system-wide file table entry.

11. Consider a file system with 512-byte blocks. Assume an inode of a file holds pointers to N direct data blocks, and a pointer to a single indirect block. Further, assume that the single indirect block can hold pointers to M other data blocks. What is the maximum file size that can be supported by such an inode design?

    Ans: (N+M)*512 bytes

12. Consider a FAT file system where disk is divided into M byte blocks, and every FAT entry can store an N bit block number. What is the maximum size of a disk partition that can be managed by such a FAT design?

    Ans: $2^N * M$ bytes

13. Consider a secondary storage system of size 2 TB, with 512-byte sized blocks. Assume that the filesystem uses a multilevel inode datastructure to track data blocks of a file. The inode has 64 bytes of space available to store pointers to data blocks, including a single indirect block, a double indirect block, and several direct blocks. What is the maximum file size that can be stored in such a file system?

    Ans: Number of data blocks = $2^{41}/2^9 = 2^{32}$, so 32 bits or 4 bytes are required to store the number of a data block.

    Number of data block pointers in the inode = 64/4 = 16, of which 14 are direct blocks. The single indirect block stores pointers to 512/4 = 128 data blocks. The double indirect block points to 128 single indirect blocks, which in turn point to 128 data blocks each.

So, the total number of data blocks in a file can be 14 + 128 + 128*128 = 16526, and the maximum file size is 16526*512 bytes.

14. Consider a filesystem managing a disk with block size 2^b bytes, and disk block addresses of 2^a bytes. The inode of a file contains n direct blocks, one single indirect block, one double indirect block, and one triple indirect block. What is the maximum size of a file (in bytes) that can be stored in this filesystem? Assume that the indirect blocks only store a sequence of disk addresses, and no other metadata.

    Ans: Let x = number of disk addresses per block = 2^{b-a}. Then max file size is $2^b*(n+x+x^2+x^3)$.

15. The `fork` system call creates new entries in the open file table for the newly created child process. [T/F]

    Ans: F

16. When a process opens a file that is already being read by another process, the file descriptors in both processes will point to the same open file table entry. [T/F]

    Ans: F

17. Memory mapping a file using the `mmap` system call adds one or more entries to the page table of the process. [T/F]

    Ans: T

18. The read system call to fetch data from a file always blocks the invoking process. [T/F]

    Ans: F (the data may be readily available in the disk buffer cache)

19. During filesystem operations, if the filesystem implementation ensures that changes to data blocks of a file are flushed to disk before changes to metadata blocks (like inodes and bitmaps), then the filesystem will never be in an inconsistent state after a crash, and a filesystem checker need not be run to detect and fix any inconsistencies. [T/F]

    Ans: F (If there are multiple metadata operations, some may have happened and some may have been lost, causing an inconsistency. For example, a bitmap may indicate a data block is allocated but no inode points to it.)

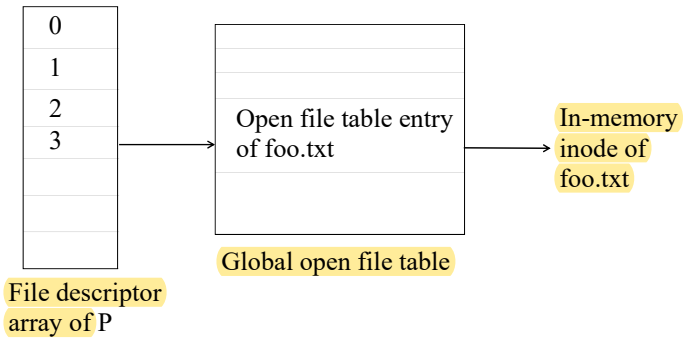
20. Interrupt-based device drivers give superior performance to polling-based drivers because they eliminate the time spent by the CPU in copying data to and from the device hardware. [T/F]

    Ans: F

21. When a process writes a block to the disk via a disk buffer cache using the write-back policy, the process invoking the write will block until the write is committed to disk. [T/F]

    Ans: F

22. Consider a process P that has opened a file `foo.txt` using the `open` system call. The figure below shows the file descriptor array of P and the global open file table, and the pointers linking these data structures.



(a) After opening the file, P forks a child C. Draw a figure showing the file descriptor arrays of P and C, and the global open file table, immediately after the fork system call successfully completes. It is enough to show the entries pertaining to the file `foo.txt`, as in the figure above.

(b) Repeat part (a) for the following scenario: after P forks a child C, another process Q also opens the same file `foo.txt`.

Ans: In (a), the file descriptor arrays of P and C are pointing to the same file table entry. In (b), the file descriptor array of Q is pointing to a new open file table entry, which points to the same inode of the file foo.txt.