Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: xv6 Filesystem

1. Consider two active processes in xv6 that are connected by a pipe. Process W is writing to the pipe continuously, and process R is reading from the pipe. While these processes are alternately running on the single CPU of the machine, no other user process is scheduled on the CPU. Also assume that these processes always give up CPU due to blocking on a full/empty pipe buffer rather than due to yielding on a timer interrupt . List the sequence of events that occur from the time W starts execution, to the time the context is switched to R, to the time it comes back to W again. You must list all calls to the functions `sleep` and `wakeup`, and calls to `sched` to give up CPU; clearly state which process makes these calls from which function. Further, you must also list all instances of acquiring and releasing `ptable.lock`. Make your description of the execution sequence as clear and concise as possible.

   Ans:

   (a) W calls `wakeup` to mark R as ready.
   (b) W realizes the pipe buffer is full and calls `sleep`.
   (c) W acquires `ptable.lock`.
   (d) W gives up the CPU in `sched`.
   (e) The OS performs a context switch from W to R and R starts execution.
   (f) R releases `ptable.lock`.
   (g) R calls `wakeup` to mark W as ready.
   (h) R realizes the pipe buffer is full and calls `sleep`.
   (i) R gives up the CPU in `sched`.
   (j) The OS performs a context switch from R to W and W starts execution

2. State one advantage of the disk buffer cache layer in xv6 besides caching. Put another way, even if there was zero cache locality, the higher layers of the xv6 file system would still have to use the buffer cache: state one functionality of the disk buffer cache (besides caching) that is crucial for the higher layers.

   Ans: Synchronization - only one process at a time handles a disk block.

3. Consider two processes in xv6 that both wish to read a particular disk block, i.e., either process does not intend to modify the data in the block. The first process obtains a pointer to the struct buf using the function “bread”, but never causes the buffer to become dirty. Now, if the second process calls “bread” on the same block before the first process calls ”brelse”, will this second call to “bread” return immediately, or would it block? Briefly describe what xv6 does in this case, and justify the design choice.

Ans: Second call to bread would block. Buffer cache only allows access to one block at a time, since the buffer cache has no control on how the process may modify the data

4. Consider a process that calls the log_write function in xv6 to log a changed disk block. Does this function block the invoking process (i.e., cause the invoking process to sleep) until the changed block is written to disk? Answer Yes/No.

   Ans: No

5. Repeat the previous question for when a process calls the bwrite function to write a changed buffer cache block to disk. Answer Yes/No.

   Ans: Yes

6. When the buffer cache in xv6 runs out of slots in the cache in the bget function, it looks for a clean LRU block to evict, to make space for the new incoming block. What would break in xv6 if the buffer cache implementation also evicted dirty blocks (by directly writing them to their original location on disk using the bwrite function) to make space for new blocks?

   Ans: All writes must happen via logging for consistent updates to disk blocks during system calls. Writing dirty blocks to disk bypassing the log will break this property.

7. (a) Recall that buffer caches of operating systems come in two flavors when it comes to writing dirty blocks from the cache to the secondary storage disk: write through caches and write back caches. Consider the buffer cache implementation in xv6, specifically the `bwrite` function. Is this implementation an example of a write through cache or a write back cache? Explain your answer.

   (b) If the xv6 buffer cache implementation changed from one mode to the other, give an example of xv6 code that would break, and describe how you would fix it. In other words, if you answered "write through" to part (a) above, you must explain what would go wrong (and how you would fix it) if xv6 moved to a write back buffer cache implementation. And if you answered "write back" to part (a), explain what would need to change if the buffer cache was modified to be write through instead.

   (c) The buffer cache in xv6 maintains all the `struct buf` buffers in a fixed-size array. However, an additional linked list structure is imposed on these buffers. For example, each `struct buf` also has pointers `struct buf *prev` and `struct buf *next`. What additional functions do these pointers serve, given that the buffers can all be accessed via the array anyway?

   Ans:

   (a) Write through cache

   (b) If changed to write back, the logging mechanism would break.

   (c) Helps implement LRU eviction

8. Consider a system running xv6. A process has the three standard file descriptors (0,1,2) open and pointing to the console. All the other file descriptors in its `struct proc` file descriptor table are unused. Now the process runs the following snippet of code to open a file (that exists on disk, but has not been opened before), and does a few other things as shown below. Draw a figure showing the file descriptor table of the process, relevant entries in the global open file table, and

the in-memory inode structures pointed at by the file table, after each point in the code marked parts (a), (b), and (c) below. Your figures must be clear enough to understand what happens to the kernel data structures right after these lines execute. You must draw three separate figures for parts (a), (b), and (c).

```
int fd;
fd = open("foo.txt", O_RDWR); //part (a)
dup(fd);                      //part (b)
fd = open("foo.txt", O_RDWR); //part (c)
```

Ans:

(a) Open creates new FD, open file table entry, and allocated new inode.

(b) Dup creates new FD to point to same open file table entry.

(c) Next open creates new open file table entry to point to the same inode.

9. Consider the execution of the system call `open` in xv6, to create and open a new file that does not already exist.

   (a) Describe all the changes to the disk and memory data structures that happen during the process of creating and opening the file. Write your answer as a bulleted list; each item of the list must describe one data structure, specify whether it is in disk or memory, and briefly describe the change that is made to this data structure by the end of the execution of the open system call.

   (b) Suggest a suitable ordering of the changes above that is most resilient to inconsistencies caused by system crashes. Note that the ordering is not important in xv6 due to the presence of a logging mechanism, but you must suggest an order that makes sense for operating systems without such logging features.

   Ans:

   (a)
   - A free inode on disk is marked as allocated for this file.
   - An inode from the in-memory inode cache is allocated to hold data for this new inode number.
   - A directory entry is written into the parent directory on disk, to point to the new inode.
   - An entry is created in the in-memory open file table to point to the inode in cache.
   - An entry is added to the in-memory per-process file descriptor table to point to the open file table entry.

   (b) The directory entry should be added after the on-disk inode is marked as free. The memory operations can happen in any order, as the memory data structures will not survive a crash.

10. Consider the operation of adding a (hard) link to an existing file /D1/F1 from another location /D2/F2 in the xv6 OS. That is, the linking process should ensure that accessing /D2/F2 is equivalent to accessing /D1/F1 on the system. Assume that the contents of all directories and files fit within one data block each. Let $i(x)$ denote the block number of the inode of a file/directory, and let $d(x)$ denote the block number of the (only) data block of a file/directory. Let L denote the starting

block number of the log. Block L itself holds the log header, while blocks starting $L + 1$ onwards hold data blocks logged to the disk.

Assume that the buffer cache is initially empty, except for the inode and data (directory entries) of the root directory. Assume that no other file system calls are happening concurrently. Assume that a transaction is started at the beginning of the link system call, and commits right after the end of it. Make any other reasonable asusmptions you need to, and list them down.

Now, list and explain all the read/write operations that the disk (not the buffer cache) sees during the execution of this link operation. Write your answer as a bulleted list. Each bullet must specify whether the operation is a read or write, the block number of the disk request, and a brief explanation on why this request to disk happens. Your answer must span the entire time period from the start of the system call to the end of the log commit process.

Ans:

- read i(D1), read d(D1)—this will give us the inode number of F1.
- read i(F1). After reading the inode and bringing it to cache, its link count will be updated. At this point, the inode is only updated in the buffer cache.
- read i(D2), read d(D2)—we check that the new file name F2 does not exist in the directory. After this, the directory contents are updated, and a note is made in the log.
- Now, the log starts committing. This transaction has two modified blocks (the inode of F1 and the directory content of D2). So we will see two disk blocks written to the log: write to L+1 and L+2, followed by a write to block L (the log header).
- Next, the transactions are installed: a write to disk blocks i(F1) and d(D2).
- Finally, another write to block L to clear the transaction header.

11. Which of the following statements is/are true regarding the file descriptor (FD) layer in xv6?

A. The FD returned by `open` is an index to the global `struct ftable`.

B. The FD returned by `open` is an index to the open file table that is part of the `struct proc` of the process invoking `open`.

C. Each entry in the global `struct ftable` can point to either an in-memory inode object or a pipe object, but never to both.

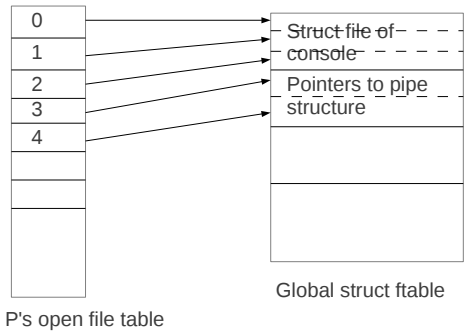
D. The reference count stored in an entry of the `struct ftable` indicates the number of links to the file in the directory tree.

Ans: BC

12. Consider the following snippet of the shell code from xv6 that implements pipes.

```
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
  panic("pipe");
if(fork1() == 0){
close(1);
dup(p[1]);   //part (a)
close(p[0]);
close(p[1]);
runcmd(pcmd->left);
}
if(fork1() == 0){
close(0);
dup(p[0]);
close(p[0]);
close(p[1]);
runcmd(pcmd->right);
}
close(p[0]);
close(p[1]);  //part (b)
wait();
wait();
```

Assume the shell gets the command "echo hello | grep hello". Let P denote the parent shell process that implements this command, and let CL and CR denote the child processes created to execute the left and right commands of the above pipe command respectively. Assume P has no other file descriptors open, except the standard input/output/error pointing to the console. Below are shown the various (global and per-process) open file tables right after P executes the `pipe` system call, but before it forks CL and CR.

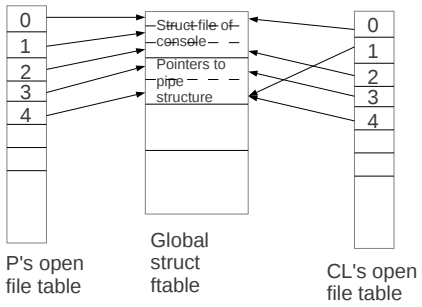


Draw similar figures showing the states of the various open file tables after the execution of lines marked (a) and (b) above. For each of parts (a) and (b), you must clearly draw both the global and per-process file tables, and illustrate the various pointers clearly. You may assume that all created child processes are still running by the time the line marked part (b) above is executed. You may

also assume that the scheduler switches to the child right after fork, so that the line marked part (a) in the child runs before the line marked part (b) in the parent.

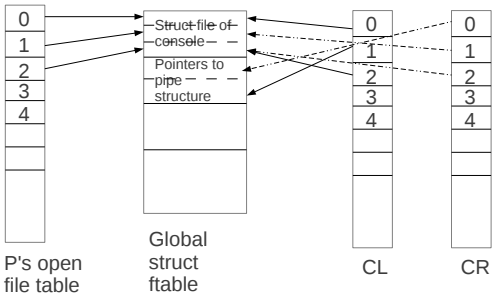
Ans:

(a) The fd 1 of the child process has been modified to point to one of the pipe's file descriptors



by the close and dup operations.

(b) CL and CR are now connected to either ends of the pipe. The parent has no pointers to the



pipe any more.

13. Consider a simple xv6-like filesystem, with the following deviations from the xv6 design. The filesystem uses a simple inode structure with only direct pointers (and no indirect blocks) to track file data blocks. Unlike xv6, this filesystem does not use logging or any other mechanism to guarantee crash consistency. The disk buffer cache follows the write-back design paradigm. Much like xv6, free data blocks and free inodes on disk are tracked via bitmaps. A process on this system performed a `write` system call on an open file, to add a new block of data at the end of the file. The successful execution of this system call changed several data and metadata blocks in the disk buffer cache. However, a power failure occurs before these changed blocks can be flushed to disk, causing changes to some disk blocks to be lost before being propagated to disk.

    (a) Consider the following scenario after the crash. The user reboots the system, and opens the file he wrote to just before the crash. From the file size on disk, it appears that his last write completed successfully. However, upon reading back the data he had written in the last block, he finds the data to be incorrect (garbage). Can you explain how this scenario can occur? Specifically, can you state which changed blocks pertaining to this system call were propagated to disk, and which weren't?

    Blocks correctly changed on disk:

    Blocks whose changes were lost during the crash:

    (b) Repeat the above question when the user finds himself in the following scenario. Immediately after reboot, the user finds that the data he had written to the file just before the crash did survive the crash, and he is able to read back the correct data from the file. However, after a few hours of using his system, he finds that the file now has incorrect data in the last block that he wrote before the crash.

    Blocks correctly changed on disk:

    Blocks whose changes were lost during the crash:

    (c) Now, suppose the filesystem implementation is changed in such a way that, for any system call, changes to data blocks are always written to the disk before changes to metadata blocks. Which of the two scenarios of parts (a) and (b) could still have occurred after this change to the filesystem, and which could have been prevented?

    Scenario(s) that still occur:

    Scenario(s) prevented:

    (d) Now, suppose the filesystem comes with a filesystem checker tool like `fsck`, which checks the consistency of filesystem metadata after a crash, and fixes any inconsistencies it finds. If this tool is run on the filesystem after the crash, which of the two scenarios of parts (a) and (b) can still happen, and which could have been prevented?

    Scenario(s) that still occur:

    Scenario(s) prevented:

    Ans: (a) inode and bitmap changed, data block change lost (b) inode and data block changed, bitmap change lost (causing block to be reused) (c) part a wont happen, b will still occur (d) part a can still happen, b wont occur

14. Which of the following system calls in xv6, when executed successfully, always result in a new entry being added to the open file table?

    (A) `open` (B) `dup` (C) `pipe` (D) `fork`

Ans: AC

15. Consider an in-memory inode pointer `struct inode *ip` in xv6 that is returned via a call to `iget`. Which of the following statements about this inode pointer is/are true?

    (A) The pointer returned from `iget` is an exclusive pointer, and another process that requests a pointer to the same inode will block until the previous process releases it via `iput`
    (B) The pointer returned by `iget` is non-exclusive, and multiple processes can obtain a pointer to the same inode via calls to `iget`
    (C) The contents of the inode pointer returned by `iget` are always guaranteed to be correct, and consistent with the information in the on-disk inode
    (D) The reference count of the inode returned by `iget` is always non-zero, i.e., `ip->ref > 0`

    Ans: BD

16. When is an inode marked as free on the disk in xv6?

    (A) As soon as its link count hits 0, even if the reference count of the in-memory inode is non-zero
    (B) As soon as the reference count of the in-memory inode hits 0, even if the link count of the inode is non-zero
    (C) When both the link count and reference count of the in-memory inode are 0
    (D) Cannot say; the answer depends on the exact sequence of system calls executed

    Ans: C