

SOLUTION
Midterm III
April 30th, 2026

CS162: Operating Systems and Systems Programming

| | |
|---|--|
| Your Name: | |
| SID: | |
| CS162 Login: | |
| TA Name/Section Time | |
| Person to Left, Right, Front, and Back of you: | |

This is a **closed book** exam. You are allowed 3 pages of notes (both sides). You have 170 minutes to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. *Make your answers as concise as possible.* On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

| Problem | Possible | Score |
|---------|----------|-------|
| 1 | 18 | |
| 2 | 16 | |
| 3 | 22 | |
| 4 | 26 | |
| 5 | 18 | |
| Total | 100 | |

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False [18 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Write only in the box. Also, answers without an explanation *GET NO CREDIT*.

Problem 1a[2pts]: The coordinator in two-phase commit (2PC) sends GLOBAL-COMMIT if at least a majority of participants vote VOTE-COMMIT.

True False

Explain: *The coordinator will send GLOBAL-COMMIT only if all of the participants send VOTE-COMMIT. If any participants send VOTE-ABORT, then the coordinator will abort (send a VOTE-ABORT).*

Problem 1b[2pts]: An empty Amazon Kindle gains weight when you add electronic books to it.

True False

Explain: *According to Professor Kubi, a Kindle full of books is in a higher energy state than an empty Kindle, so all things being equal, the higher energy state translates into (a very tiny amount of) weight.*

Problem 1c[2pts]: The FAT file system is faster than the Berkeley Fast File System when performing random accesses to a file.

True False

Explain: *In the FAT file system, files consist of blocks linked together in a linear chain, so that random access requires traversing this linked list to reach the desired data. In contrast, the Berkeley FFS can directly access a random block in the file by consulting the inode pointers.*

Problem 1d[2pts]: Because RAID 5 is an erasure code, it requires individual disks to be able to identify when they have failed using some other mechanism (such as ECC codes on each disk).

True False

Explain: *This is true because the XOR code used in RAID5 is unable to identify which disk has failed. It can only correct for a failed disk's contents after the bad disk has been identified.*

Problem 1e[2pts]: Assume that we have a filesystem using a buffer cache. If a file is created and then immediately deleted, the filesystem must write to the underlying disk multiple times (once on creation and once on deletion) to keep the buffer cache consistent with the disk.

True False

Explain: *As discussed in class, the buffer cache holds dirty pages for a brief period of time before it writes them to disk. Consequently, temporary files may never make it to disk before they are deleted.*

Problem 1f[2pts]: In Pintos the maximum number of user locks that a user thread can hold at one time is 65536 (i.e. 2^{16}).

True False

Explain: *We accepted two reasons here: (1) In Pintos, a user lock is represented by an unsigned char, so there can be no more than 256 distinct user locks. (2) In Pintos, there can be up to 2^{32} unique locks by using the pointer as a unique address to identify each lock.*

Problem 1g[2pts]: Suppose that you and your friend adopted a new IP-layer protocol where every packet has a checksum, and any packet whose contents are changed in transit is detected perfectly. You use this protocol to send a file to a friend, and they reconstruct the file only from packets that pass this checksum. If no additional end-to-end mechanism is used at the sender or receiver, then the friend is guaranteed to receive the file correctly.

True False

Explain: *There are many additional sources of error in the path from source to destination, including memory and software errors. This is one of the essential points of the End-To-End argument, namely the fact that you need end-to-end mechanisms and as a result, pushing error mitigation mechanisms into the network should only be done as a performance enhancement.*

Problem 1h[2pts]: When using RPC to communicate over the network, a client and host/server could share arguments and return values even if the communicating entities have processors with different “endianness” (i.e. x86 vs MIPS).

True False

Explain: *RPC packages include serialization libraries that translate to and from a system-independent data format. Thus, in case of different “endianness”, the serialization libraries take care of translating at either end. We will also take answers which reference the host-to-network and network-to-host macros available in Oses.*

Problem 1i[2pts]: In systems using the network file system (NFS), caches on clients can become temporarily inconsistent with each other (making it appear to different clients that different data is present in a given file).

True False

Explain: *NFS relies on periodic polling to keep caches up to date with the server. For a short period of time after a first client writes data, a second one will continue to see the old data until it polls to detect the change.*

Problem 2: Multiple Choice [16pts]

Problem 2a[2pts]: Which of the following are *true* about the clock algorithm (*Choose all that apply*):

- A: The clock algorithm can be considered to be an approximation to LRU.
- B: The clock algorithm will find and replace the least recently used physical page.
- C: The Clock Algorithm arranges *physical* pages into two groups – an active group that is mapped as “valid” and managed in a FIFO list and an inactive group that is mapped as “invalid” and managed as an LRU list.
- D: The clock algorithm sets the *accessed* bit to 0 in software, and lets the hardware set this bit back to 1.
- E: None of the above.

A: *TRUE. The clock algorithm is able to detect pages that are “old”, namely those that haven’t been accessed in a while. Thus, it is able to find “an old page” rather than “the oldest page”, making it an approximation to LRU.*

B: *FALSE. The clock algorithm is not able to find “the oldest page”, only an old page that hasn’t been accessed in a while.*

C: *FALSE. This is a description of the “second-chance” algorithm, not the clock algorithm.*

D: *TRUE. This is a description of what the clock algorithm does.*

E: *FALSE. For obvious reasons.*

Problem 2b[2pts]: Which of the following is *true* about Pintos’ file system? (*Choose one*):

- A: Directory data (i.e., directory entries) are stored on the directory’s inode on disk.
- B: Two threads can always write to the same file concurrently.
- C: The seek syscall should never require blocks in the file to be allocated or freed.
- D: The directory path “/././” is invalid.
- E: None of the above.

A: *FALSE. Inodes only hold metadata, not data.*

B: *FALSE. The basic file system implementation is not threadsafe; simultaneous modification by concurrent threads can cause inconsistency.*

C: *TRUE. Setting the offset to any location in the file does not cause the underlying blocks allocated of the file to change (this only changes on a write or rollback)*

D: *FALSE. This is false because “.” in the root directory simply points back at the root directory.*

E: *FALSE. For obvious reasons.*

Problem 2c[2pts]: Which of the following are *true* about distributed decision making protocols (*Choose all that apply*):

- A: The PAXOS protocol will block when one of the participants fail.
- B: Byzantine Agreement protocols are able to come to shared decisions among distributed nodes, even if some of them are malicious (compromised and intentionally trying to corrupt the protocol).
- C: Distributed decision making protocols *cannot* be utilized to produce replicated state machines on nodes distributed over the network.
- D: Two-phase commit can block waiting for a failed coordinator to reboot.
- E: None of the above.

- A: *FALSE. PAXOS is, by design, resilient to participant failure.*
 B: *TRUE. This is the basic property of Byzantine Agreement protocols.*
 C: *FALSE. The distributed decision making protocol can be used to agree on (among all participants) the next transition so that all participants can keep their state machines in sync.*
 D: *TRUE. The workers must wait for the final decision from the coordinator, which means that they may be blocked waiting for this decision to be announced.*
 E: *FALSE. For obvious reasons.*

Problem 2d[2pts]: Little's Theorem has the following properties (*Mark all that apply*):

- A: It cannot handle systems with service times that have $C > 1$.
 B: It can be used to compute the average time spent in a queue, given the average length of the queue and average arrival rate.
 C: It shows why the average length of time spent in a queue grows without bound as the system utilization approaches 100%.
 D: It applies only to systems in equilibrium.
 E: None of the above.

- A: *FALSE. Little's theorem works correctly with all probability distributions.*
 B: *TRUE. Since Little's theorem applied to queues is $L_{queue} = \lambda T_{queue}$, you can compute the third variable, given the other two.*
 C: *FALSE. Little's law is too general to explain the arbitrary growth of a queue with probabilistic input probabilities (for instance, it works for a deterministic input/deterministic output system, which can run at 100% without arbitrary queue growth).*
 D: *TRUE. This statement is correct, since Little's law cannot explain transient behavior of a queue (there is no stable average behavior to connect queue length and queuing time).*
 E: *FALSE. For obvious reasons.*

Problem 2e[2pts]: The Meltdown security flaw (made public in 2018) was able to gain access to protected information in the kernel because (*Choose one*):

- A: Some processors with out-of-order instruction execution allowed user-code to *begin* an illegal access to kernel memory in a way which affected cache state – even though the illegal access was not allowed to complete.
 B: A POSIX-compatible system call implemented by multiple operating systems neglected to properly check arguments and could thus be fooled into returning the contents of protected memory to users.
 C: There was a wide-spread bug in the x86 architecture that caused certain loads to ignore kernel/user distinctions in the page table under the right circumstances.
 D: User code could observe the time it took to complete certain system calls and therefore infer the contents of kernel memory.
 E: None of the above.

- A: *TRUE. The Meltdown security flaw involved this timing channel which was observable only on certain processor architectures (Intel, for one) and not others (AMD).*
 B-D: *FALSE: Meltdown was none of these things.*
 E: *FALSE. For obvious reasons.*

Problem 2f[2pts]: Which of the following are true about memory-mapped I/O? (*Choose all that apply*):

- A: Devices which use memory-mapped I/O can be directly controlled from user-mode processes if the kernel maps the I/O address ranges to the process' address space.
- B: Memory-mapped I/O is a software protocol that constructs a coherent distributed shared memory between multiple physical nodes, allowing communication between nodes to occur as reads and writes to the shared memory (address) space.
- C: Memory-mapped I/O is incompatible with DMA.
- D: Memory-mapped I/O is a hardware mechanism that assigns physical memory addresses to I/O devices such that processor read and write operations to these addresses become commands to devices.
- E: None of the above.

- A: *TRUE. Since the ability to control devices that use memory-mapped I/O is through physical addresses, the kernel can hand control to user-mode processes simply by mapping the device addresses into the user's address space.*
- B: *FALSE. This statement describes a distributed shared memory (DSM) protocol, not a memory-mapped I/O protocol.*
- C: *FALSE. This is false because DMA works perfectly well transferring data to and from memory/memory-mapped devices. The DMA controller acts as a special memory copy engine.*
- D: *TRUE. This is a direct description of what memory-mapped I/O does.*
- E: *FALSE. For obvious reasons.*

Problem 2g[2pts]: Which of the following statements about the buffer cache are true? (*Mark all that apply*):

- A: Prefetching subsequent disk blocks into the buffer cache can reduce the overall access time for disk operations
- B: The buffer cache can hold *dirty data*, i.e. data that is more up-to-date than the data on disk.
- C: The buffer cache uses a write-through caching policy.
- D: Because the buffer cache is positioned *between* the system-call interface and the disk, it is possible for different processes to see different contents for shared files.
- E: None of the above.

- A: *TRUE. Since all reads/writes of disk data go through the buffer cache, we can prefetch data from disk when it is convenient (for example but reading the next few blocks whenever we read a particular only) and subsequent reads of this prefetched data will hit in the cache for reduced average access time.*
- B: *TRUE. The buffer cache is a "write-back" cahce. As discussed in class, holding dirty data in the buffer cache for a short period of time after it is written has a variety of performance benefits and is an essential property of the buffer cache..*
- C: *FALSE. The buffer cache is a "write-back" cache, not a "write-through" cache.*
- D: *FALSE. Exactly the opposite is true: because the buffer cache is between the system-call interface and the disk, all writes to the file system are placed in the cache and are thus visible to any readers. Consequently, all processes see the same contents for files.*
- E: *FALSE. For obvious reasons.*

Problem 2h[2pts]: Which of the following statements about the Virtual File System (VFS) are true? *(Mark all that apply):*

- A: VFS is a file system for virtual machines that can be mounted on a host machine.
- B: VFS abstracts away details about the underlying mounted filesystem (such as inode structure, directory format, etc) from the rest of the kernel.
- C: VFS allows you to transparently access a remote file system through the network.
- D: VFS exposes a unique syscall interface for every mounted file system type (e.g. POSIX read vs NFS readAt).
- E: None of the above.

A: FALSE. VFS is not a file system. It is an interface to file systems.

B: TRUE. This is the essential function of VFS: unifying the different interfaces presented by various file systems into a common "unix-like" file system interface.

C: TRUE. This statement describes the original genesis of VFS: it was developed by Sun Microsystems as part of the NFS file system.

D: FALSE. VFS does exactly the opposite: it unifies the different interfaces and internal functionality of various file systems into a shared (common) system-call interface.

E: FALSE. For obvious reasons.

Problem 3: Short Answer Potpourri [22 pts]

For the following questions, please provide as short an answer as you can that actually answers the question. In most cases, this means one or two sentences. *We reserve the right to take off points for answers that are not concise.*

Problem 3a[2pts]: Explain why a pure LRU replacement policy is impractical for demand paging (in virtual memory). Why does the clock algorithm not suffer from this issue?

To implement LRU, you would need to keep physical pages in a linked list which would get rearranged on every memory access, thus turning a single memory access (possibly cached) into a multi-memory access operation (100s of % of overhead). The clock algorithm doesn't have this problem because the software component operates infrequently relative to memory accesses (which simply set the "use" bit in the PTE in hardware at cache access speed).

Problem 3b[3pts]: Explain what DMA is (don't just say what it stands for) and explain how it helps to improve the performance of a file system. How does the processor find out that the DMA has completed?

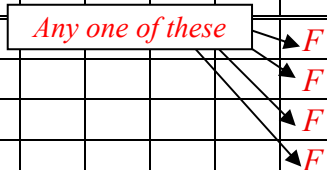
DMA, which stands for "direct memory access", is a mechanism that can directly transfer data from an I/O device to or from memory – without needing the main CPU to transfer each byte (after being set up by the CPU). DMA helps to improve the performance of a file system by freeing the processor to do other things while data is transferred to/from the disk. Typically, the DMA engine signals completion to the CPU via an interrupt.

Problem 3c[4pts]: For the following problem, assume a hypothetical machine with 4 pages of physical memory and 7 pages of virtual memory. Given the access pattern:

A B C D E A A E C F F G A C G D C F

Indicate in the following table which pages are mapped to which physical pages for each of the following policies. Assume that a blank box matches the element to the left. We have given the FIFO policy as an example. You may break ties in any order you wish.

| Access→ | | A | B | C | D | E | A | A | E | C | F | F | G | A | C | G | D | C | F |
|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FIFO | 1 | A | | | | E | | | | | | | | | C | | | | |
| | 2 | | B | | | | A | | | | | | | | | | D | | |
| | 3 | | | C | | | | | | | F | | | | | | | | |
| | 4 | | | | D | | | | | | | | G | | | | | | |
| MIN | 1 | A | | | | | | | | | | | | | | | | | F |
| | 2 | | B | | | E | | | | F | | G | | | | | | | F |
| | 3 | | | C | | | | | | | | | | | | | | | F |
| | 4 | | | | D | | | | | | | | | | | | | | F |
| LRU | 1 | A | | | | E | | | | | | | | A | | | | | F |
| | 2 | | B | | | | A | | | | | | G | | | | | | |
| | 3 | | | C | | | | | | | | | | | | | | | |
| | 4 | | | | D | | | | | | F | | | | | | D | | |



Problem 3d[2pts]: What is the guarantee provided by two-phase commit? Why doesn't it violate the General's Paradox?

The guarantee is that only one of two things happen atomically: (1) All players (coordinator and all workers) will eventually commit or (2) All players (coordinator and all workers) will eventually abort. Two-phase commit does not violate the General's Paradox because it does not specify that commit or abort operations will occur simultaneously, merely that they will eventually happen.

Problem 3e[2pts]: Give one reason why the performance of write operations on a FLASH device such as an SSD might be both *slower* and *more variable* than reads. Note that answers we are looking for are specifically about what happens in the SSD, not in the OS (filesystem, etc).

There are a number of possible answers to this question, for instance: (1) Physically, writes take longer than reads because voltages need to be raised high enough to force electrons to tunnel across the gate and into the trapping plate, which takes time. Reads can just examine voltages in a similar way to DRAM reads and are thus fast. (2) The time to write a page is also slower and more variable than the time to read a page because flash devices cannot overwrite existing pages. Thus, every write must allocate a pre-erased page from a free list, which can take a variable amount of time (because it may become blocked due to ongoing garbage collection and/or erasing operations).

*Note that the answer is **not** because reads can be a byte at a time while writes must be performed in a complete page. SSD devices read and write data in chunks (called "pages").*

Problem 3f[3pts]: Journaling file systems require data to be written twice (once in the journal, then again in the primary file system). How can the file system optimize for good response time? How about good throughput? Explain.

Journaling file systems can use asynchronous write back, meaning changes will not immediately be applied to disk on a commit. This allows for a good response time since committing a transaction simply involves writing a small commit entry to a sequential log. To optimize for good throughput, the file system can batch write-backs to schedule disk accesses more efficiently than immediate individual or small groups of writes.

Problem 3g[2pts]: How could the `mmap()` system call be used to set up shared memory between two processes so that they could share a linked list between them (here sharing means that the two of them could add items to the list and traverse the list assuming that suitable synchronization was employed to avoid inconsistencies). Be explicit about the location in the virtual memory space of each process.

In order for two processes to share memory between them in a way that accommodates shared linked lists, they need to have identical regions of their virtual memory space mapped to a shared chunk of memory. (So that a given chunk of physical memory is mapped to the SAME virtual address in both processes.) One of the ways that you can do this with `mmap()` is by using the same (non-zero) first argument to `mmap()` and the same file and offset in each process.

Problem 3h[2pts]: Consider two queueing systems handling incoming requests with the same memoryless arrival rate λ . If System A has a deterministic (constant) service time and System B has a highly variable service time (many short jobs and a few long ones). Assume that System A and B have the same mean service time. Which system would you expect to have a larger mean queueing delay? Explain why.

Assuming an M/G/1 queueing model, with all things otherwise identical, System B would have a larger mean queueing delay. Consider the $\frac{1}{2}(1 + C)$ term in the queueing delay. For System A, there is no variability in the services times, so $\text{Var}[T_s] = 0 \Rightarrow C = 0$. For System B, the high variability of service times means a larger variance in service time, leading to a (potentially much) larger value of C. The result is that the time in queue is larger for System B.

Intuitively, high variability of services times means a that there will be frequent times in which a long job can occupy the server allowing many jobs to build up in the queue and increasing the average queue wait time.

Problem 3i[2pts]: Name and describe two optimizations introduced in BSD 4.2 by the Berkeley Fast File System (FFS)? Make sure to explain the advantage of each of your optimizations.

There are a number of potential answers here. Here are a few:

- 1. Increased block size from 256bytes=>4K+: This increases performance by causing larger chunks of data to be read sequentially off the disk, thereby decreasing the average time per byte.*
- 2. Partitioning disk into Block Groups: This allows allows better heuristics for data allocation, increasing locality.*
- 3. Distribute inodes among BlockGroups: This allows data to be closer to inodes as well as inodes of a directory to be close to each other, reducing seeks and increasing performance.*
- 4. Using bitmap allocation: This makes it easier to find local runs of free blocks, thereby increasing locality/sequentiality of data.*
- 5. Skip-sector positioning: for slow processors, reduced the chance that the processor would miss chance to read data from disk (so that we wouldn't have to do a complete rotation of disk for each sector).*
- 6. 10% space always reserved: increased probability that there would be runs of empty blocks for better block allocation (and better locality).*
- 7. Prefetch/readahead: Reduce the number of seeks for sequential access patterns.*

[This page intentionally left blank]

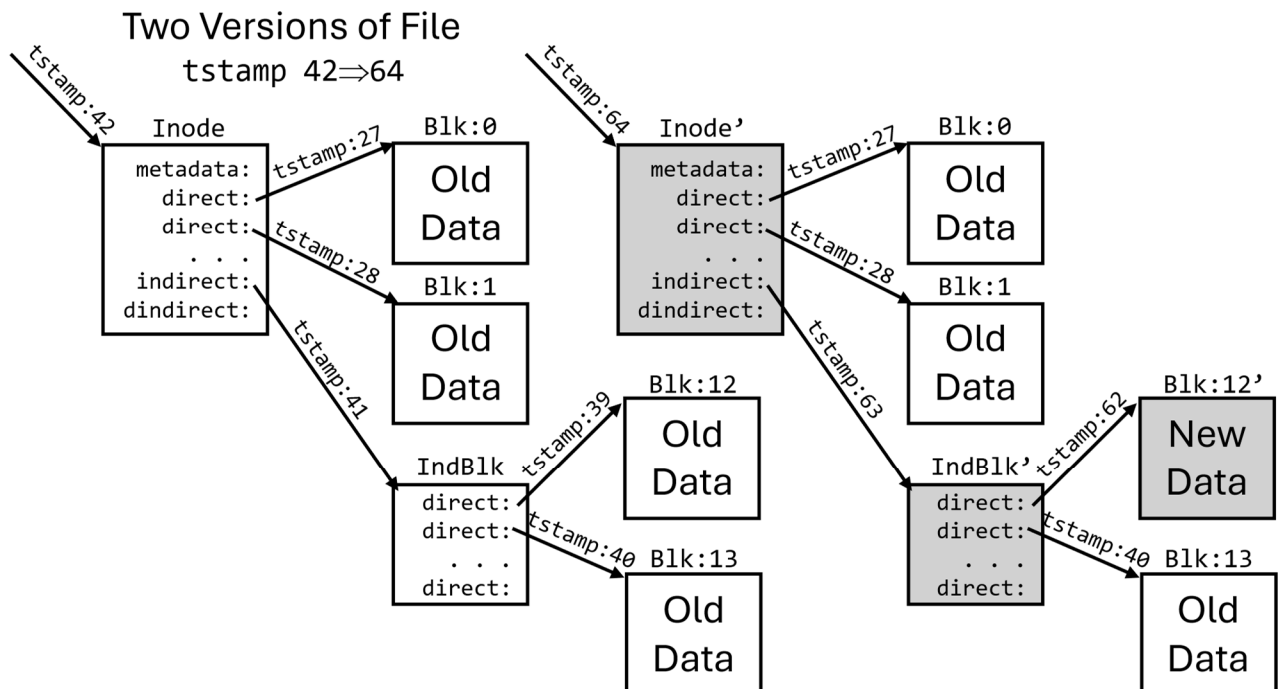
**Problem 4 starts on the flip side!
Do not forget to turn over this page.
We inserted blank page for better page alignment**

Problem 4: ZFS-Lite for Pintos [26pts]

In this problem we design a Pintos file system in the spirit of ZFS, where every update generates a new version of the file using copy-on-write (COW). To make this problem tractable, we produce a file system with only one file in it, but you should be able to see how the ideas generalize. Our filesystem (ZFS-Lite) uses a Linux-style *inode* structure with up to two levels of indirection: each file has *NDIRECT* (defined as 12) direct block pointers, one single-indirect block pointer, and one double-indirect block pointer.

ZFS-Lite has two properties that we want to highlight here. First, we never overwrite data blocks in place. Instead, we create a new block for the data and then update the *inode* and any indirect blocks with *copy-on-write* (COW) by first copying their contents into new blocks before modifying them. When completed, we have a new *inode* for the updated file which can replace the previous *inode* with a single atomic write to disk. This scheme provides clean update semantics (a type of transaction) without a journal. In a full ZFS filesystem, *inodes* are actually called *dnodes* and placed into arrays which are updated via COW with every update and for which updates proceed up to a top block called an “überblock” (superblock). For this problem, we have only one *inode*, a pointer to which gets written into the *superblock* when the file is updated. We place the *superblock* in a well defined place on disk so that we can always read our file after reboot.

Second, ZFS-Lite associates a monotonically increasing “birth time” with every new block. These birth times are kept with *pointers* to blocks within metadata, as shown by the figure below. The following figure shows an original version of the file constructed at *tstamp*=42. At *tstamp*=62, a write of block 12 is requested, which completes at *tstamp*=64. *In grey are the new blocks generated during this write.* Note that datablocks that are unchanged (i.e. blocks 0-11 and 13) are still in use with the new version of the file and retain their original timestamps. Note further that metadata blocks must have timestamps that are later than the blocks that they point to (this really just states that metadata blocks must be newer than the blocks they point to).



Problem 4a[2pts]: Writing new data to a file involves updating multiple blocks. Explain why performing a final atomic write of the `inode pointer` to the `superblock` can be viewed as providing transactional semantics to the file update. Explain any conditions that must be satisfied to view this `superblock` write as a transaction commit.

Although the file update involves multiple steps (allocating and writing blocks, including copying metadata to inode and indirect blocks), none of these steps affect the current copy of the file, which is defined by the old inode. Committing the update involves a single atomic write of the superblock which swaps in the new version of the file in a single operation. Should a crash occur before this final write, then it is as if the pending operations never happened. [All or nothing semantics]. The conditions (already stated) include the fact that none of the pending block allocations/updates should impact the current inode's information and the write to the superblock must be atomic.

Problem 4b[2pts]: A *snapshot* is a read-only version of a file from the past that persists until the snapshot is deleted. You can only create a snapshot of the file at the current time. Explain why we could view the `inode` with `tstamp=42` in the above figure as a *snapshot* of the file, assuming that we keep this pointer in a separate location of the `superblock` so that it does not get overwritten every time that the file is updated. What rule must we follow about garbage collecting old data blocks to make sure that a snapshot remains just that – a valid read-only copy of the file?

The inode with tstamp=42 has all of the data and structure from a version of the file (at time 42). If we keep the pointer to the inode in a separate part of the superblock, it stays available until we choose to delete it. The rule we must follow is that we never delete (garbage collect) data or metadata blocks that are in either the current version of the file or in any snapshots.

Problem 4c[2pts]: Assume that a file has N blocks in it. What is the algorithmic complexity of creating a snapshot of this file with respect to N ? Use Big-O notation. Explain.

Assuming that our system follows the garbage-collection rule of never deleting an in-use block, then we can create a snapshot in $O(1)$ time by just copying the pointer to the inode.

Problem 4d[2pts]: Consider the `inode` with `tstamp=42` in the above figure as the most recent snapshot and the `inode'` with `tstamp=64` as the most recent version of the file. If we update this most recent version of the file (leaving the snapshot alone), we will displace some data or metadata blocks of this file. How will we know whether or not to free these displaced blocks? Explain.

Hint: The time-complexity of deciding whether or not to free a block is independent of the number of blocks in the file or size and number of snapshots.

Simple algorithm: only delete a newly displaced block if it was created (has a tstamp) that is later than the tstamp of the last (most recent) snapshot.

Explanation: Consider the point at which the most recent snapshot was created: assuming that the file and any earlier snapshots were complete (i.e. none of their data or metadata had been prematurely deleted), then we know that this latest snapshot is complete (since it will just have a pointer to the latest inode). Any future updates to the inode will allocate new blocks that are not in any snapshots and have tstamps that are later than the (newest) snapshot. These new blocks are the only ones we can consider deleting if they are displaced, since anything else is still a part of some existing snapshot.

Consider the following definitions for our implementation of ZFS-Lite. You should interpret these in light of your Project 3 knowledge from Pintos. First, we have our new “timestamped pointers”, which combine a sector reference (on disk) with a timestamp. The type is “ts_block_ptr_t”:

```
#define BLOCKSIZE    512                // Sector(==block) size in bytes
typedef uint32_t block_sector_t;        // Pointer to physical sector
typedef uint32_t tstamp_t              // Timestamp

// A pointer to a disk block, stamped with the logical time of creation
typedef struct ts_block_ptr {
    block_sector_t sector;
    tstamp_t      birth;
} ts_block_ptr_t;
```

Here we have modified versions of the inode structures you are used to (using timestamped pointers to sectors instead of just pointers to sectors):

```
// Inode definitions (note that an inode takes up a full BLOCK)
#define NDIRECT    12
#define NINDIRECT  (BLOCKSIZE / sizeof(ts_block_ptr_t))
typedef struct inode_disk { // inodes look like this
    off_t length;
    ts_block_ptr_t direct[NDIRECT];
    ts_block_ptr_t indirect;
    ts_block_ptr_t double_indirect_t;
    /* padding */
} inode_disk_t;

typedef struct indirect_block { // indirect blocks look like this
    ts_block_ptr_t next_ptrs[NINDIRECT];
} indirect_block_t;

typedef struct inode { // File version or snapshot
    ts_block_ptr_t sector;
    inode_disk_t data;
} inode_t;
```

Problem 4e[2pts]: Compute the maximum file size for ZFS-Lite, keeping in mind that $BLOCKSIZE=512$. Please simplify your answer to a number. Show your work.

$$\text{Max File size} = BLOCKSIZE \times (NDIRECT + NINDIRECT + NINDIRECT^2)$$

$$NINDIRECT = \frac{512}{(4+4)} = 2^{9-3} = 64$$

$$\text{Max File size} = 512 B \times (12 + 64 + 64^2) = \frac{1}{2} KB \times (4172) = 2.086 MB$$

Problem 4f[4pts]: Consider a *time-ordered* sequence of snapshots, $SN_t, SN_{t+1}, SA, SB, SC, \dots, SN_{t+k-1}$, with SN_{t+k-1} being the latest. Can you *briefly* describe an algorithm for removing an arbitrary snapshot, say snapshot SB from the middle of the set of snapshots? It should delete any sectors which are no longer needed. SA and SC here are on either side of the snapshot we are deleting. What is the time complexity of your solution?

Time Complexity: $O(N)$, where N is MAX(number of blocks in SB or SC)

Description of your algorithm:

Do a depth-first walk through blocks (data and metadata) of both SB and SC together. For each block, X, delete if:

- (1) $X_{stamp} > SA_{stamp}$ (Block created after SA) and*
- (2) X is not found in similar position still in SC. Note that the only place X could appear in SC is in same position as it was in SB. Further, if it doesn't appear in SC it won't be in any later checkpoints.*

Note that the real ZFS has a much cheaper mechanism for deleting snapshots involving extents – especially important when extended to whole file system (as in 4j).

Consider the following device-level/device driver definitions. We want to define a `write_block()` function that will write a block at a given byte offset of our file. Definitions of interest are as follows. *Assume that device operations (read, write, allocate, release) always succeed.*

```

timestamp_t fs_time;           // Current filesystem time (for timestamps)
extern struct block *fs_device; // Our device

/*
 * Here is our superblock. Contains latest inode pointer and snapshots that
 * we have chosen to write out to disk. It may or may not be written to
 * disk with every update to the file.
 * Note: superblock.current_inode is the ts_block_ptr_t to latest inode
 *       superblock.snapshots[0] is the ts_block_ptr_t to latest snapshot
 */
#define SUPERBLOCK_SECTOR 2 // Some well-defined spot on disk
typedef struct superblk {
    ts_block_ptr_t current_inode;
    ts_block_ptr_t snapshots[NINDIRECT-1]; // Rev order, snapshots[0] latest
} superblk_t;
superblk_t superblock;

// Atomic read/write of one sector
void block_read(struct block *, block_sector_t sector, void *buf);
void block_write(struct block *, block_sector_t sector, const void *buf);

// Assume that allocation and release functions always succeed
void free_map_allocate (size_t cnt, block_sector_t *sector_out);
void free_map_release (block_sector_t sector, size_t cnt);

```

Problem 4g[2pts]: Complete the following implementation of `cow_alloc_write()` which takes a pointer to a buffer full of data, writes this data to a freshly allocated sector on disk, and returns a time-stamped `ts_block_ptr_t` to this sector:

```

static ts_block_ptr_t cow_alloc_write (const void *buf) {
    ts_block_ptr_t p;

    free_map_allocate (1, &p.sector) ;
    block_write(fs_device, p.sector, buf) ;
    p.birth = fs_time ;
    return p;
}

```

Note: the original problem was intended to have `fs_time` be a free-running clock. However, we also accepted solutions that assumed you had to increment time manually:

```

    p.birth = fs_time++

```

[This page intentionally left blank]

**Problem 4h is on the flip side!
Do not forget to turn over this page.
We inserted blank page for better page alignment**

Problem 4h[8pts]: Complete the definition for `write_block()`, below. Note that last step will be to write new version of inode to disk (see figure again):

```
static void write_block(inode_t *inode, size_t pos, const void *data) {
    size_t lbn = pos / BLOCKSIZE;
    inode_disk_t *id = &inode->data;
    ts_block_ptr_t old_data = {0, 0}; // Init to null, since might not use
    ts_block_ptr_t old_ind = {0, 0}; // Init to null, since might not use
    ts_block_ptr_t old_inner = {0, 0}; // Init to null, since might not use
    ts_block_ptr_t old_outer = {0, 0}; // Init to null, since might not use

    // Allocate new sector and fill with data
    ts_block_ptr_t new_data = cow_alloc_write(data);
    if (lbn < NDIRECT) {
        // Write is to a direct block. Save old pointer to possibly free.

        old_data = id->direct[lbn];

        id->direct[lbn] = new_data;
    } else if (lbn < NDIRECT + NINDIRECT) {
        // Write is to indirect block.
        ts_block_ptr_t ind[NINDIRECT] = {}; // Zeroed if no existing block
        if (id->indirect.sector != 0) { // Block exists, so read
            block_read(fs_device, id->indirect.sector, ind);
        }
        // Before replacing, save old pointer to possibly free.

        old_data = ind[lbn - NDIRECT];

        ind[lbn - NDIRECT] = new_data;

        old_ind = id->indirect;

        id->indirect = cow_alloc_write(ind);
    } else {
        // Write is to double-indirect block
        ts_block_ptr_t outer[NINDIRECT] = {}; // Zeroed if no double-indirect block
        ts_block_ptr_t inner[NINDIRECT] = {}; // Zeroed if no single-indirect block
        // Compute index of lbn in double (outer) and single-indirect (inner)

        size_t off = lbn - DIRECT - NINDIRECT;

        size_t outer_idx = off / NDIRECT;

        size_t inner_idx = off % NDIRECT;

        if (id->double_indirect.sector != 0) { // Block exists, so read
            block_read(fs_device, id->double_indirect.sector, outer);
        }
        if (outer[outer_idx].sector != 0) { // Block exists, so read
            block_read(fs_device, outer[outer_idx].sector, inner);
        }
    }
}
```

```

    old_data = inner[inner_idx]
                ;
    inner[inner_idx] = new_data
                ;
    old_inner = outer[outer_idx]
                ;
    outer[outer_idx] = cow_alloc_write(inner)
                ;
    old_outer = id->double_indirect
                ;
    id->double_indirect = cow_alloc_write(inner)
                ;
}

// Finally, allocate new sector for inode and write to disk
ts_block_ptr_t old_inode = inode->sector;

inode->sector = cow_alloc_write(id)
                ;

// put into superblock and save superblock before releasing old inode
superblock->current_inode = inode->sector;
block_write(fs_device, SUPERBLOCK_SECTOR, superblock);
cow_free(old_inode); // Release old inode

// Free any non-zero blocks not in use by any snapshots
cow_free(old_data);
cow_free(old_ind);
cow_free(old_inner);
cow_free(old_outer);
}

```

Problem 4i[2pts]: Considering your answer from (4d), fill in the missing blanks for the `cow_free()` function which will conditionally free (i.e. release) a disk sector displaced when updating the contents of a file. *Hint: Your answer requires a time that is independent of the size or number of snapshots. Possibly helpful: time of most recent snapshot: `superblock.snapshots[0].birth`*

```

static void cow_free(block_ptr_t p) {
    if (p.sector == 0)
        return;

    if (p.birth <= superblock.snapshots[0].birth)
        _____
        return;
        _____
    free_map_release(p.sector, 1);
        _____
}

```

Problem 4j[3pts Ex Credit]: Explain how you would generalize the above implementation of ZFS-Lite to include a full file system (with directories) and for which snapshots are across the whole file system, not just for a single file.

The trick here is to assign unique identities (innumbers) to inodes each time a file or directory is created. The simplest way is with a monotonically increasing integer such that the root directory inode could have a well-known ID, say ID 0. Then, instead of placing pointers to inodes directly into the superblock (as above), we insert them into a “mapping tree” of some sort built from indirect blocks (many options here!), with the pointer to the top block of the tree inserted into the superblock. Imagine, for instance, an NINDIRECT-ary tree where the leaf blocks hold pointers to inodes, with inumber 0 at far left of tree, etc..

Properties:

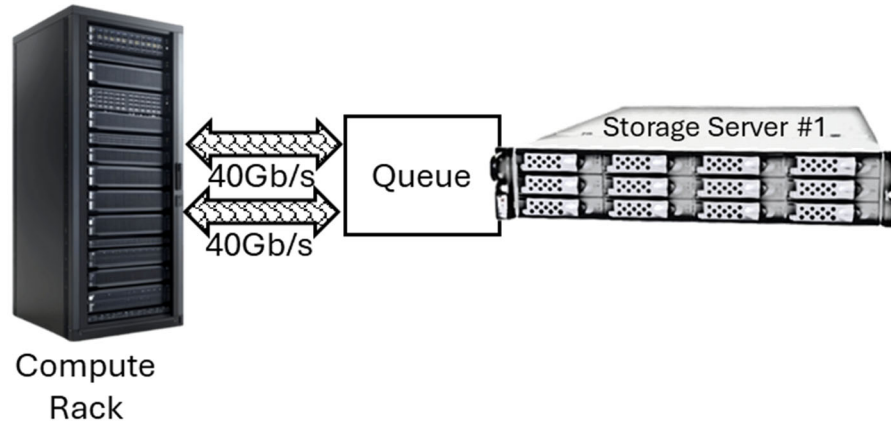
- 1) Now, directories are just files (as usual) mapping names to innumbers.*
- 2) We can look up file or directory by inumber simply by looking it up in the “mapping tree” to get pointer to inode (completed with tstamp).*
- 3) Update of a file proceeds as earlier, updating the inode pointer. Then, we replace the old pointer entry for the inode in the mapping tree by using COW and continuing with all the internal blocks of tree up to superblock. Write of superblock gives transactional update just as before.*
- 4) Note that the inumber for a file stays the same even as we update the file. Hence all directories containing the file stay the same (do not need to be changed just because the file is updated).*
- 5) Checkpoints of whole file system take $O(1)$ time and are just copies of pointers to mapping tree.*

[This page intentionally left blank]

Problem 5 starts on the flip side!
Do not forget to turn over this page.
We inserted blank page for better page alignment

Problem 5: RAID6 File System Performance [18pts]

NOTE: IN THE FOLLOWING PROBLEM, ALL OF THE NUMBERS HAVE BEEN CHOSEN SO THAT ANSWERS CAN BE COMPUTED WITHOUT A CALCULATOR



In this problem, we will consider a network-based storage system consisting of a storage server connected to a high-performance compute system via high-bandwidth optical links.

The Storage server is characterized as follows:

- Contains two links connecting it to a high bandwidth network switch. Each link handles 40Gb/s of network traffic and is routed through a queue to handle bursts of traffic.
- Contains 12 disks, each of which is 50TiB in size
- Uses disks that rotate at 15,000RPM, have a data transfer rate of 131072 KB/s (for each disk), have a 2.4ms average seek time, and a 4096 Byte sector size. *Note* $131072/4096 = 32$.
- Each disk has a SCSI interface with a $100\mu\text{s}$ controller command time. Assume that a group of consecutive sectors can be fetched with a single request.
- Disks are combined into RAID6 partitions of 6 disks each, which means that each partition has 4 data disks and 2 parity disks. It also means that there are 2 independent partitions/server.
- All RAID processing is done via internal buses in the storage server.

Because the disks are configured in RAID6 partitions, the system will spread consecutive sectors of a file across multiple disks when writing a large file, and read them back in a similar fashion. Specifically, the first sector worth of data in the file will be read from disk zero (0). The second sector will be read from disk one (1). Eventually it will wrap around to disk 0, after reading *one sector from each data disk*. Consequently, to read sector X from the file, the system will contact disk $(X \bmod N)$, where N is the number of data disks in the strip.

Possibly helpful numbers include:

$$\begin{array}{ll}
 4096 \times 16 = 65536, & 1/8 = 0.125 \\
 4096 \times 32 = 131072, & 1/16 = 0.0625 \\
 4096 \times 64 = 262144, & 1/32 = 0.03125 \\
 4096 \times 128 = 524288 & 1/64 = 0.015625
 \end{array}$$

Each of the following answers should be simplified to a single number without rounding. However, to get full credit you must show your work.

Problem 5a[4pts]: What is the average *service time* to retrieve one sector from a single disk starting at a random location on disk? *Hint: there are four terms in this service time.*

$$\begin{aligned} T_{ser} &= T_{controller} + T_{seek} + T_{rotate} + T_{transfer} = \\ &2.4ms + 0.1ms + \frac{1}{2} \left(\frac{60000 \text{ ms/m}}{15000 \text{ r/m}} \right) + \left(\frac{4096 \text{ B} \times 10^3 \text{ ms/s}}{131072 \times 10^3 \text{ B/s}} \right) = \\ &2.5ms + 2ms + 0.03125ms = 4.53125 \text{ ms} \end{aligned}$$

Problem 5b[2pts]: Write an equation for the average *service time* to read a contiguous set of X sectors from one of the RAID6 partitions as a single request. Assume that disks are independent of one another so that accesses for independent disks can be initiated in parallel. Your equation will be a 2-term latency function as discussed in class with a constant *overhead* (intercept) and a linear term that depends on X (although this term will involve a “ceiling” function).

$$T_{ser} = 4.5ms + 0.03125 \left\lceil \frac{X}{4} \right\rceil ms$$

Problem 5c[2pts]: Suppose that we frequently access very large files in chunks of 64 sectors at a time from a partition. What is the average *service time* for such a request (assume that you start reading in a random part of the file)? *Hint: Use your equation from (5b).*

$$T_{ser} = 4.5ms + 0.03125 \left\lceil \frac{64}{4} \right\rceil ms = 5ms$$

Problem 5d[2pts]: *Ignoring queueing effects*, what is the maximum number of these large (64 sector) I/O requests that can be made per second (IOPS) from any one server partition? Does this exceed our network bandwidth to the switch?

$$IOPS = \frac{1}{T_{ser}} = \frac{1}{5 \text{ ms/I/O}} \times \frac{1000ms}{s} = 200 \text{ IO s}^{-1}$$

Problem 5e[3pts]: Consider a series of 64-sector requests to a single partition. Treat the entire system as a M/G/1 queue, with a coefficient of variance of $C=1.5$. Also assume that we do not want to have our total response time (time spent in queue + service time), to be more than 50% larger than the raw service time. What is the *utilization* of the RAID6 server at this point? What is the maximum rate of requests that we can sustain? Show your work. You can simplify your answers to rational numbers (i.e. a fractions).

The time in the queue (which is M/G/1) is 50% of service time $T_{ser} \Rightarrow$

$$T_q = \frac{u}{1-u} \times \frac{1}{2} (1+C) \times T_{ser} = 50\% \times T_{ser}$$

$$\Rightarrow \frac{2.5}{2} \left(\frac{u}{1-u} \right) = 0.5 \Rightarrow 5u = 2(1-u) \Rightarrow 7u = 2$$

$$\Rightarrow u = \frac{2}{7}$$

So, at this utilization, the rate of requests is $200 \frac{10}{s} \times \frac{2}{7} = \frac{400}{7} IOPS$

Problem 5f[3pts]: What is the service time for *modifying a single sector* in a file at a random place in one of our RAID6 servers? Explain why this number is different from your answer to (5a) and how you might avoid the extra overhead when you are appending to a large file one block at a time (the way that files normally grow in Unix-like OSes).

In order to modify a single sector, we need to update the parity blocks associated with the appropriate strip. The simplest thing is to just read the whole data strip, insert changes, then write back to full strip. So, we want $2 \times T_{ser}$, using the single-block timing from 5a = 9.0625ms

We also accepted an answer that observed that the writeback would require a complete rotation to accomplish, adding an extra $\frac{1}{2} T_{rotate} = 2ms$. So, answer = 11.0625ms

We avoid this overhead by keeping data in a non-volatile RAM (NVRAM) until we have a full strip worth (4 blocks), after which we can write data and parity blocks in parallel to the disks in parallel. This would avoid the factor of 2. If we had a really large NVRAM buffer, we could, in principle save up a large block of data (64 sectors) to get the higher data rate of 5d.

Problem 5g[2pts]: If one of the disks in a partition fails, explain what happens when we try to read data that was on that failed disk. Ignoring queuing effects, what might get in the way of achieving the same maximum read IOPS that we computed in (5d)?

The problem states that we have a failed data disk. So, we would have to read all 5 remaining disks in parallel and perform the inverse transform that generated the parity disks in order to reconstruct the missing data. If we were not able to perform the inverse function at full speed, we might end up reading more slowly, but this slowdown would not be due to the physical disk reading process.

As an aside, many RAID6 coding schemes generate one of the parity blocks via XOR, just like RAID5, in which case the "inverse" is really easy (assuming only one disk is failed): just XOR other disks to get missing data. You were not required to know this.

[Scratch Page (please remove)]

[Scratch Page (please remove)]