CS240
Advanced topics in operating systems
Midterm Exam – Thursday, Dec 5, 2024

OPEN BOOK, OPEN NOTES,
NO ELECTRONIC DEVICES

Your Name: Answers

SUNet ID: root `@stanford.edu`

In accordance with both the letter and the spirit of the Stanford Honor Code, I neither received nor provided any assistance on this exam.

Signature: ____________________

- You have 80 minutes to complete the exam.
- Answer 6 out of the 9 questions. You can earn a maximum of 60 points.
- Please **cross out the 3 questions you do not want graded** (or we will grade the first 6 you mark).
- Follow all **boldface** instructions. Some multiple choice questions require a justification for any credit.
- Keep your answers concise. We will deduct points for a correct answer that also includes incorrect or irrelevant information. State any big assumptions you make.

1	/10
2	/10
3	/10
4	/10
5	/10
6	/10
7	/10
8	/10
9	/10

1. [10 points]:

You have a program that creates a file, writes data to the file, then exits, never calling `fsync()`. A power outage takes down your machine immediately after the program exits. Once power is restored and the file system is repaired with fsck, in which cases is the file contents guaranteed to reflect all the data your program wrote to it?

Mark all cases in which the file will always contain the data written to it:

- ☐ **A** The file was created on a local file system mounted with the default options (noasync, nosync) meaning not every write goes to disk immediately, but also the file system is intended to be recovered after a crash.
- 🟥 **B** The file was created on an NFS file system and the server lost power at the same time as the client.
- 🟥 **C** The file was created on an NFS file system but the server was on a UPS (uninterruptible power supply) and never lost power or rebooted.
- 🟥 **D** The file was created on xsyncfs, the program ran in the foreground on your terminal, and you noticed that the program exited right before the power went out.
- ☐ **E** The file was created on xsyncfs but the program ran in the background so there was no indication that it exited.

Justify your answer:

Answer:

Without fsync or the sync mount option, the local file system will not write the data to disk.

In NFS, all write RPC must store data stably on disk before replying to the client, and all writes must complete before the file is closed (on exit from the program). Hence, regardless of whether the server loses power, it will have stored the file on disk before the program exits.

In xsyncfs, if there's no way to notice whether the program has completed (output, network packet, rocket launch), the file system need not have written the data to disk.

2. [10 points]:

The NFS paper states they flush data back on `close()` rather than on each write.

Explain what problem this can cause for file system errors and quickly sketch how you could apply ideas from the Speculator system mentioned in the xsyncfs paper (and class) to potentially handle this problem.

Answer:

Problem: when a program does a write() system call to write data, the actual NFS write operation occurs (much) after the write() call returns. Thus, there's no callsite to return any error that occurs (e.g., media failure, permenent server crash, quota exceeded).

Use the checkpoint/restore from speculator as follows:

1. *At each NFS write() call, checkpoint the state and continue executing.*
2. *When the write occured, if there was no error, delete the checkpoint it corresponded to. If there was an error, rollback to the write() call and return the error.*

If errors are rare and checkpoints were cheap this could potentially still give benefits from delayed writeback.

3. [**10 points**]**:**

The xsyncfs guys use the webcam of your laptop to see when you are looking at your screen and, if not, remove all dependencies on output printed to the the terminal (but not for other operations).

1. **For Figure 4 and 5 in the xsyncfs paper: do you expect this hack to make much of a performance difference? (Why or why not?)**
2. **Give a situation where this change can violate externally synchronous behavior.**

Answer:

Not much gain is possible as xsyncfs is already close to the optimal" performance. In Figure 4 xsync is almost the same performance as the "gold standard" of a completely asynchronous file system and in Figure 5 is very close to the even gold-er standard of a ramFS file system that is entirely in main memory. There isn't much fat left.

As for violating external synchrony, suppose it sees you aren't watching, allows a `printf` *message "data is committed" to escape before the operation has hit disk, but then you look back and see a "committed" message. As soon as the webcam notices you looking back, it will attempt to flush to disk, but the system could then crash before the data is written out.*

4. **[10 points]:**

   OS-GPT claims: "If you changed LFS to track external synchrony as in xsyncfs, it won't improve performance or correctness much."

   Is this statement always true, sometimes true, or never true? Please provide concrete details from the papers or experiments to justify your answer.

Answer:

1. *Since LFS batches into large segments (equivalant to an enormous group commit) and delays for 30 seconds we would expect it won't perform worse than xsyncfs on Figure 4 and 5 in the xsyncfs paper.*

   However, it is unlikely to beat xsyncfs by an appreciable margin since (1) in Figure 4 xsync is almost the same performance as the "gold standard" of a completely asynchronous file system and (2) is very close to the even gold-er standard of a ramFS file system that is entirely in main memory. There isn't much fat left.

2. *If an LFS application calls sync() or fsync() it should improve since these would block LFS (though it would essentially group commit the result in a partial segment write).*

3. *Correctness will improve. xsyncf preserves external syncrony, and (if correct) won't lose file system mutation that occurred before a user-visible screen output or network. The original LFS doesn't guarantee this at all: an application could lose 30 seconds of file system mutations after sending many messages or outputs after performing them.*

5. [10 points]:

Recall our informal description of a journaling file system: to atomically transition from the old file system state (FS_{old}) to a new file system state (FS_{new}) you *conceptually*:

1. Compute the value of each live byte in FS_{new} that differs from FS_{old} along with its on-disk disk location.
2. Write this set of differences to the next location in a persistent journal on stable storage, along with any additional information you might need (such as a timestamp).

If your hard-disk provides both 512-byte atomic sector writes and write barriers (as in the xsyncfs paper) explain the steps needed to :

1. **Write the journal and file system data and metadata to disk such that the file system can recover from any crash that happens during this process.**
2. **During recovery after a crash: what steps are required before it can delete the log after replaying it? (Hint: make sure you can recover if it crashes while doing so).**

Answer:

1. *You first have to write the journal entries completely to disk (i.e., wait til all writes complete) along with a way to know that all entries were written completely (checksum is easiest, but ordered writes and timestamp can work, too).*

2. *After the journal entries are persistently written to disk you can then flush out the file system data and metadata (in any order). Note: This data must be completely written to disk before the journal can be discarded later.*

3. *When you recover, as you roll forward and apply the changes, you can only delete the journal only after ensuring these changes are flushed out to disk (persistent). If not, a crash during recovery will cause you to corrupt your file system (we found this bug in every file system we checked back in the mid 2000s).*

6. [**10 points**]:

In Figure 1 of the map/reduce paper consider the operations "**(3) read**" , "**(4) local write**", "**(5) remote read**" and "**(6) write**".

Answer the following, and justify your answers:

1. **Why is operation (4) local and operation (5) remote? What type of operation (local, remote, other) are (3) and (6)?**
2. **Given their system assumptions: what is a reasonable ordering of these four operations from cheapest to most expensive?**

Answer:

1. *(4) is local bc the mapper writes to the local disk. in general, (5) is remote because a reducer typically has to read from many mapper jobs. note, however, in degenerate cases this could potentially be a local read when the reducer only has to read from a single mapper, which happened to execute on the machine it runs on as well.*

   (3) is often a local read when the master can scheduler the mapper on teh same machine that stores a chunk. Since each chunk is replicated (eg 3x) there is a reasonable chance of this. If it can't be local, the master attempts to make it be semi-local by running on a machine close by.

   (6) has a remote write component since the write is replicated. Also, this is much more expensive than a regular write since there is overhead cooridnating the replicas (voting etc).

2. *for local: my guess is that (4) local write should be fastest since asynchronous. (3) when the read is local since its a compulsorary fetch and you can't mask the latency for that job.*

   we would accept either order.

   for remote: (5) the remote fetch is cheaper than the remote replicated write (6) since the bw is 2x for (6) + overhead of coordination. you can see this in the graphs.

7. **[10 points]:**

   At the end of Section 3.2 of *A Comparison of Software and Hardware Techniques for x86 Virtualizaton*, Adams and Agesen write, "we observe that BT is not required for execution of **most** user code on most guest operating systems" [emphasis added].

   Assume you run an exokernel-based operating system as a guestOS. How would you expect their observation to change? (Justify your answer!)

   Give an example of something unprivileged code could do that would prohibit directly executing it.

Answer:

Library operating systems do operating system type operations, so you'd expect them to need more interposition. One example if the exposure of physical names such as physical page numbers.

The exokernel could give unprivileged libOS code read-only access to its page tables so that the user code can efficiently check accessed and dirty bits. (Fast access to dirty bits can speed up some garbage collection algorithms.)

The guest OS could give user code write access to other privileged data structures, such at the global descriptor table or task switch segment.

The guest OS could give user code access to virtual addresses that are reserved for the VMM. In particular, the trick of limiting segment registers in kernel requires virtual addresses to be unused, which works because most 32-bit guest OSes don't use the top 4 MiB of virtual memory. Segmentation cannot limit access to lower addresses, and without page protection user code may be able to bypass segment protection by writing to the local descriptor table.

8. [10 points]:

The paper *A Comparison of Software and Hardware Techniques for x86 Virtualizaton* describes a very useful trick of how they optimize guestOS code translation by assuming it is "innocent until proven guilty".

Two parts:

1. **Give a clear intuition for the optimization with a common example.**
2. **Using this intuition, explain what Figure 1 (section 3.3) is intended to show, especially why the 'jmp' instruction was inserted.**

Answer:

NOTE: This answer needs some tuning.

Loads/stores to "regular" memory don't need BT translation, but those to page tables do. It's not obvious from examining the load/store instruction which case they fall into. Since most memory operations are not to page tables (or similar things like DMA or device memory) the software VMM assumes they are "innocent" and can be run IDENT (roughly full speed).

However, if they do access page tables they will result in a trap. After a threshold number of traps (could be 1) the VMM will rewrite the memory instruction to be check of the memory type and, if it needs special handling, a direct call to the appropriate routine.

Figure 1 shows the translation cache with different code fragments. Initially it has translated the code IDENT (left side). After it determines the instruction is not innocent it regenerates a code fragment that does the appropriate simulated action and inserts a jump from the old fragment to the new one. It does not rewrite the old fragment in general b/c the size of the block in bytes will likely (always?) be too small.

9. [**10 points**]:

Assume you have two routines that sort integer arrays:

```
void sort_simple(int *v, unsigned n);
void sort_fast(int *v, unsigned n);
```

where `n` is the number of elements in the given array `v`.

Write a simple test harness (some pseudo-code is ok) that when run using KLEE will either prove that both implementations give the same sorted result for all integer arrays of size `n=1` up to `n=10` or if a difference is found for a given size, cause KLEE to emit an error.

Answer:

```
for(int n = 1; n <= 10; n++) {
    int nbytes = n*4;
    int *a1 = malloc(nbytes);
    int *a2 = malloc(nbytes);

    // make two copies so that sorting one won't change the
    // other but they have the same initial constraints.
    klee_mark_sym(a1, nbytes);
    memcpy(a2,a1, n*4);

    sort_simple(a1,n);
    sort_fast(a1,n);

    if(memcmp(a1,a2,nbytes) != 0)
        panic("error␣for␣size␣n=%d\n", n);
    else
        printf("all␣arrays␣of␣size␣n=%d␣are␣sorted␣correctly!\n", n);
 }
```

Extra space for notes or overflow:

Answer: