08/12/2018

CH 42 – Crash Consistency: FSCK and Journaling

- The main issue we are focussing over here is how to make the FS consistent with all the crashes happening. The main components that need to be updated when a write happens to the system are the inode bitmap, the data bitmap, inode and the data itself. These are three separate writes, and the system can perform only one write at a time, hence the crash could happen anytime causing the FS to be inconsistent in state.
- Few scenarios are as follows. *Only **one write** goes through*: **Data is written**. In this case, the FS is not in an inconsistent state, because a write did not happen, but the user has lost his data. **Inode is Written**. In this case the inode would point to the disk space, but it would only read garbage value. Further, the data bitmap says that data block is not allocated, but the inode says otherwise. This results in the FS being inconsistent. **Bitmap is Written**. In this case, the FS is inconsistent again, since bitmap says, some data block is allocated, but no inode is pointing to it – could lead to **space leak**. *Only **Two Writes** go through*. **Inode + Bitmap**: In this case, the FS won't be inconsistent, but the data would be garbage. **Inode + Data**: In this the FS would be inconsistent, as the data bitmap, does not recognize that data block is allocated. **Bitmap + Data**: In this case FS would be inconsistent since, data is written and is recognized as being allocated, but no inode is pointed to it, resulting in the FS not knowing which file the data belongs to.
- The first solution proposed is to use a file checker that would check the consistency of the FS *before* the FS is mounted and made available. It would check for several things including: **Superblock sanity checks**: which involved confirming the mapping described in the SB and the actual mappings that FSCK calculated. **Free Blocks**: fsck checks the inode, (double) indirect blocks, to get an overview of block allocation, and build a bitmap and confirms the same with the data and inode bitmaps. If inconsistent, it is resolved by using the info from inodes. **Inode State**: fsck checks for corrupted inodes, and clears them if found and updates the bitmap. **Inode Links**: The link count for each inode is calculated by fsck by building a directory tree, and verified. If a mismatch is found, corrective actions are taken, if allocated node is discovered, but no directory points to it –> *lost and found*. **Duplicates**: if duplicate inodes referring to same block are found, one could be cleared or could be given its own inode. **Bad Blocks**: Pointing outside of the valid range, can be cleared.
  - As can be seen this solution can be very slow, and the performance would be prohibitive when the disk grows in capacity.
- The next solution that we talk about is the Write-ahead logging/**Journaling**. In this all we do is create a journal buffer to hold all the information of the operation that the disk is about to perform, before actually performing. It would consist of a TxB block with a **TransactionID**, to indicate the beginning of the transaction, followed by all the operations, and finally ending with a TxE block, to indicate the end of the transaction, with a **TransactionID**. The operations that are written to this journal block, are then written to the disk one by one, and not together as that could lead to problems, as it would be susceptible to disk scheduling, and a crash could lead to missing out on some transaction. To avoid this problem, the disk would first write all the blocks except the TxE block using what is called **checkpointing**, which would be to write out he contents of the update to the final disk location. Hence the steps that would be involved are – **Journal write**, **Journal commit**, and **Checkpointing**. If the file system crashes at any point, the steps would be re-done (**redo logging**).
- **Limiting the journal**: Journal size should be limited because, if not then the amount of tasks that the journal would hold, that would need checkpointing would grow to a point that it would no longer be able to hold transactions, and even in case of a crash, the recovery would take a lot of time. To address this, we would bake the journal circular and adding a Journal Superblock which would keep information about the tasks that haven't been check-pointed yet, this in turn would help clear out the tasks that have been checkpoints so that the journal could be cleared for newer tasks to be added.
- **Metadata Journaling**: The approach here (ordered journaling) is a bit different as what is followed above, in way that the data being written to the disk is no longer a part of the journaling process, and is written before the journaling process even begins. This reduces the cost of the double writing (first to the journal and then again to the disk). The data is written before the journalling, process as otherwise the inode could be pointing to garbage values.
- Backpointer-based consistency makes sure that the data block being pointed by the inode, has a back pointer to the inode. The FS can be sure of consistency if both the forward-pointer and the back pointer are in tandem.

CH 43 – Log-Structured File Systems

- Ideal Filesystem would hence use sequential disk **+** focus on increasing the write performance **+** it would work on performing well on common workloads **+** work well with raids
- Log-structured File System
  - It would first buffer all the updates + metadata in an in-memory segment. Once the memory segment is full, the content is written to disk in one long sequential part of the disk. **NOTE** : LFS never over-writes existing data, but rather always writes segments to free locations.
- Writing to disk Sequentially
  - The basic idea is to load all the data blocks and the corresponding meta-data blocks including the inode blocks in the in-memory segment before writing the data down in a sequential manner.
  - Assume a single data block to be written, once it is written in the in-memory block, the corresponding inode block would be written right next to it, while it would have a pointer which would point back to the data block. The data block is usually 4KB in size, whereas inode is around 128 bytes in size.
- Writing to disk sequentially and efficiently
  - The basic idea is to use the concept of **write buffering** which makes use of an in-memory buffer for buffering the writes first, and once the buffer is full, then sequentially write out the content of the in-memory segment to the disk.
  - For example if we have data blocks 0,1,2,3,4 and the corresponding inode blocks. Then we would first write down all the data blocks, then all the inode blocks, and then create a link back from the inode blocks to the corresponding data blocks.
- How much to buffer?
  - The data that would be needed to be buffered for optimal performance would depend on the disk
  - The factors it would depend on are:
    - Peak transfer rate (P), Time it takes to position the head (T), The effective bandwidth of peak (F)
    - Therefore,
      - $D = \frac{F}{1-F} * R_{Peak} * T_{Position}$, Where D is the data that can be buffered for optimal performance
- Inode map
  - Since, inode in LFS is scattered all throughout the disk, we need an inode map takes the inode number as input and produces the disk address of the most recent version of the inode. Any time an inode is written the imap is updated. And the imap is written write next to the newly edited/updated data
- Checkpoint region
  - The checkpoint region contains the pointers to the latest pieces of the inode map and thus the inode map pieces can be found by reading the CR first. The CR is updated every 30 seconds or so to avoid it affecting the performance
- Reading a file
  - So what happens is first the OS would load up the CR and search for inode map. Once the imap is loaded, it would locate the latest version of the inode using the inode number that it received for the file that it wants to read. Once it has the inode number and the has loaded the inode it would directly load up to that position in the disk because the inode would be pointing back towards the data block to be read
- Garbage collection
  - A noticeable problem with LFS is that though it works on improving the write performance by continuously updating the files in the system, the older versions of the files are still in the filesystem which are scattered. These are the garbage values. One could argue that since, these files are nothing be older versions of the same file, a versioning system could be created to allow the user to restore the older version of the files. However, the problem is that LFS only keeps one live version of the file, and the older versions are truly garbage now, since they cannot be recovered. Hence, something called garbage collection should be performed to collect all the older version of all the files in the system and cleared out, so that the file system can be made free for other purposes.
  - However, the garbage cleaner cannot just move out on a cleaning spree, as that could lead to creation of free holes in the system. Hence, the LFS aims to go segment by segment thus clearing up large chunks of space for subsequent writing
  - There would be a segment summary block which would be used to determine the liveness of a block, if the map in the SS points exactly to the exact address on the disk, it would be a live block, else it would be a dead block.
  - It is better to clean out cold segment firsts, than to clean out hot segments, which would be used more frequently.

CH – 44 Flash Based SSD

- Flash chips are organized into banks or planes, accessed in two different sized units – blocks and pages. Within each bank there are large number of blocks, which in turn have a large number of pages.
- Flash operations – **Read**: Can access any location uniformly. **Erase**: Before writing to a page, you need ERASE the entire block (setting bits to 1). Remember to copy important content, before erasing. **Program**: Change bits from 1s to 0s. Can also cause some neighboring bits to flip – **read disturbs / program disturbs**.
- Flash Translation Layer provides the control logic need for orchestrating device operations. It would take the read and write requests on logical blocks, and translate them into low-level read, erase, and program commands. Another goal is to maintain high performance and reliability, and it can be done by reducing **write amplification**, that is, write issued by FTL / write issued by client.
- **Log-Structured FTL**– In this a write to a block would be appended to the next free spot available to the currently-being written block. To allow for reads, a mapping table is made. The drawback of this form of FTL is that it leads to Garbage, and periodically, garbage collection needs to be performed. **Garbage Collection**: Read in live pages from the block, write those live pages the log, & then erase the block. To determine the block liveness, one can refer the table mapping.
- Page Level mapping can be very space consuming, alternatively we can use Block-Level Mapping(BLM), but there are some performance tradeoffs. In BLM we would need a **chunk number** and an **offset**. Chunk Number would point at the block, and then offset can be used to map into the page we're looking for.
- **Hybrid Mapping**: It has a page level mapping called **log table** and a per-block mapping called the **data table**. **Switch merge** –> convert log table and data table mappings into one data table reading. **Partial merge** –> Have both log table and data table values.
- **Wear Leveling** – Spread the writes throughout the SSD. Log-Structured FTL does a good job, but for long-lived data FTL must periodically re-write the content elsewhere, making a block available again.

Data integrity and protection

- Disk failure – Latent Sector errors (LSEs) and block corruption. **LSE**: In this the sectors of the disk would be damaged in some way. The disk would deploy some error correcting code to correct the error. Similarly a block might get **corrupted**, and this would be considered as silent faults since they would be hard to detect, as the disk would not indicate any problem.
- To solve LSE, some redundancy mechanism can be used to reconstruct the lost data with RAID where in parity information can be used or the mirrored information can be used to reconstruct the data lost.
- To detect corruption, checksum for the data can be calculated to detect the corrupt data, and then recovery as before can be used to reconstruct the lost data. The input would be the data, and the output would be the summary of data, for example, when XOR is calculated over the data, if on later access, the checksum calculated before the corruption and after the corruption would be different, this would help detect the corruption. Ways of calculating checksum, XOR, Addition, Fletcher's algorithm, Cyclic Redundancy Check. Checksum can either be stored along with the data, (Update needs 1 R and 1 W), whereas it can also be allocated a new block for N data blocks – (Update needs 1 R, 2 W)
- The detection of corruption using checksum usually happens during the read, when the client reads the checksum stored, and compares it with the checksum that it would calculate. If they're different that's when the client knows that the data is corrupted. To identify **Misdirected Writes** we can add a physical identifier to the checksum. Checksum however, cannot help with **lost writes**, other mechanism such as **write verify** or **read-after-write** can ensure that no writes are lost.
- Disk also deploys **disk scrubbing** a mechanism that would periodically read through every block of the system and verify the checksum.
- Checksum also induces **overheads** specifically with **space** and **time**. Time: overhead can be reduced by combining the aspect of copying the data (kernel cache to user buffer) and calculating the checksum.
-

CH 49 – NFS

- Focus – Transparency, Performance, and simple + fast crash recovery.
  - Solution is to use steels protocol for transfer of information. In a stateless protocol, the client would send all the information that would be necessary to carry out an operation.
  - In case of a stateful, protocol, if the server crashes, the server will have to run some kind of recovery to be able to understand subsequent client requests. Which adds to the overhead.
- **NFSv2**: File handle is how the server sends the client the information needed for access to a particular file. It consists of three identifiers. **Volume ID** – It helps to identify the file system/partition. The **inode no**. – helps identify the particular file. And **Generation number** is needed when reusing an inode. The client needs to obtain the File Handle from the server using the LOOKUP (one request per level in the DIR a/b/c.txt will need 3 requests.) protocol, and the connection is established with the MOUNT protocol. Once the client has the File handle, it can issue the READ and WRITE commands to the server.
- In case of a server crash, the NFSv2 protocol, just **retries** the sent requests, if it receives no response from the server, this possible because the requests are **idempotent**, meaning that the effect of performing the same operation multiple times is equivalent to the effect produced when the request is processed once. READ, WRITE, GETTATR, etc are idempotent, but CREATE or MKDIR is not.
- The client uses **caches** for for caching the file data and metadata. Thus, while the first access is expensive, subsequent accesses are serviced quickly out of memory. The client also uses the cache for buffering writes before actually sending the **WRITE** request to the server, that is, it would decouple the writes to the server allowing for an **immediate success** being returned to the application. This could lead to **cache inconsistencies** when multiple clients are involved – **update visibility** when the client gets old copy and is solved by **flush on close**/**close to open**, and **stale cache**, when the client is working on a version of a file that is old, to solve this, the client would first send a GETATTR request to confirm whether it has the latest copy before accessing the file. But this could lead the server to be overwhelmed with GETATTR requests, hence an **attribute cache** was used with a timeout value, to update the attribute periodically.
- Also to avoid inconsistencies, the server must commit each write to a stable storage before informing the client of success.
-

CH 50 – AFS

- The main focus of AFS was to introduce **scale** into a distributed file system such that a server can support as many clients as possible. Also, AFS is very different from NFS, such that in NFS cache consistency is hard to describe since it depends directly on low level implementation details including client side cache timeout intervals. In AFS cache consistency is simple and readily understood: when a file is opened a client will *generally* get the latest consistent copy from the server.
- **AFS Version 1**: In this version, when an *Open()* call is made to the server, with the entire filename being sent to the server, the server sends the entire file to the client, which is then ***written*** to the local disk of the client. Any subsequent *Read()* or *Write()* functions are redirected to the **local disks** thereby increasing the efficiency and avoiding the network transfer of files and the content. Once the operation is completed *AFS Client* would check if the file was modified, and if so it flushes the new version back to the server with the path and a ***Store*** *protocol message*. Next time the file is accessed by this client, it would use the ***TestAuth*** *protocol message* in order to determine whether the file has been updated, since the last time the client accessed it. If so it would fetch and store the file in the client local disk, if not the client would directly access the cached version of the file, thereby avoiding the network transfer.
- **Problems with version 1**: **(1)** – **Path traversals were costly**: When performing the Fetch and Store protocol request, the entire path traversal sent by each of the client made the server spend too much of its CPU time simply walking down directory paths. **(2)** – **Too many TestAuth messages**: Like in NFS with GETATTR, AFSv1 generated a large amount of traffic only to check whether a local file was valid or has been modified, Thus, this resulted in there servers spending too much of their time on telling the clients whether it was *OK* to use the local cached copy of the file. **(3) Load was not balanced across Servers**: Solved with Volumes. **(4)** The server used a single distinct process per client thus inducing context switching and other overheads. All these problems resulted in limiting the scalability of the system.
- **AFS Version 2**: A notion of **callback** is introduced. In this the client assumes that a file is valid until the server tells it otherwise. This is to reduce the TestAuth message protocol crowding on the server side. Also, AFSv2 introduced the notion of file identifier which includes – **volume identifie**r, **file identifier** and a ***uniquifier*** which enabled the reuse of the volume and file IDs when a file is deleted. Thus instead of sending the entire path for a file the client would walk the pathname one piece at a time caching the results and reducing the load on the server. The key difference from NFS is that with each fetch of a path the AFS client would establish a callback with the server
- **Cache Consistency**: Because of callbacks discussed in the previous point, the notion of cache consistency is simplified. There are two important cases to consider – Consistency between processes on different machines, and consistency between processes on the same machine. **On Different Machines** –When a client gets a copy of the file, it would update the visibility and invalidate any cached copies, once the update (if any) is completed to the file, then the connection with server is broken for callbacks. When the client wants to restart working on the file it would initiate a new callback connection. This step ensures that that client will no longer read stale copies of the file and subsequent opens on those clients will require a re-fetch of the file. Thereby making a consistent copy of the latest version. **On Same Machine** – in this case writes to a file are immediately made visible to other local processes. **Also there is a notion of last writer wins**. Meaning that if there are multiple clients editing the same file, the client which closes the file last would be the one whose changes would be saved to the AFS permanent storage.
- Performance of AFSv2: **First**, in many cases, the performance of each system is roughly equivalent. **Second**, difference arises during a large-file sequential re-read (Workload 6). Because AFS has a large local disk cache, it will access the file from there when the file is accessed again. NFS, in contrast, only can cache blocks in client memory; as a result, if a large file (i.e., a file bigger than local memory) is re-read, the NFS client will have to re-fetch the entire file from the remote server. **Third**, we note that sequential writes (of new files) should perform similarly on both systems (Workloads 8, 9). AFS, in this case, will write the file to the local cached copy; when the file is closed, the AFS client will force the writes to the server, as per the protocol. NFS will buffer writes in client memory, perhaps forcing some blocks to the server due to client-side memory pressure, but definitely writing them to the server when the file is closed, to preserve NFS flush-on-close consistency. However, realize that it is writing to a local file system; those writes are first committed to the page cache, and only later (in the back-

ground) to disk, and thus AFS reaps the benefits of the client-side OS memory caching infrastructure to improve performance. **Fourth**, we note that AFS performs worse on a sequential file over- write. Overwrite can be a particularly bad case for AFS, because the client first fetches the old file in its entirety, only to subsequently overwrite it. NFS, in contrast, will simply overwrite blocks and thus avoid the initial (useless) read. **Finally**, workloads that access a small subset of data within large files perform much better on NFS than AFS. In these cases, the AFS protocol fetches the entire file when the file is opened; unfortunately, only a small read or write is performed. Even worse, if the file is modified, the entire file is written back to the server, doubling the performance impact. NFS, as a block-based protocol, performs I/O that is proportional to the size of the read or write.

- **Other Improvements**: **(1)** AFS provides a true global namespace to clients, thus ensuring that all files were named the same way on all client machines. NFS, in contrast, allows each client to mount NFS servers in any way that they please, and thus only by convention would files be named similarly across clients. **(2)** AFS also takes security seriously, and incorporates mechanisms to authenticate users and ensure that a set of files could be kept private if a user so desired. NFS, in contrast, had quite primitive support for security for many years. **(3)** AFS also includes facilities for flexible user-managed access control. Thus, when using AFS, a user has a great deal of control over who exactly can access which files. NFS, like most UNIX file systems, has much less support for this type of sharing. **Finally**, as mentioned before, AFS adds tools to enable simpler management of servers for the administrators of the system. In thinking about system management, AFS was light years ahead of the field.

Virtual Machines Monitor

- **Virtualizing CPU**: In this, the basic technique that is used is limited direct execution. A virtual machine monitor must perform a **machine switch** between running virtual machines. Thus, when performing such a switch, the VMM must save the entire machine state of one OS (including registers, PC, and unlike in a context switch, any privileged hardware state), restore the machine state of the to-be-run VM, and then jump to the PC of the to-be-run VM and thus complete the switch.
- The VMM cannot execute in kernel mode, but when the OS it is hosting first boots up, it would record the location of the trap handler of the hosted OS. Now, when the VMM receives a trap from a user process running on the given OS, it knows exactly what to do: it jumps to the OS's trap handler and lets the OS handle the system call as it should.
- The OS on the VMM, cannot be running in kernel mode either, it runs on a special mode called **supervisor mode**, which allots it extra memory than usual, which is used by the OS for its data structures. Hence when a process on the hosted OS makes a **(1)** System Call. **(2)** The the process is Trapped, and the VMM calls the OS trap handler. **(3)** The actual OS' Trap Handler would respond to the trap, by decoding the trap and executing the system call & when it is done, it would issue a return from Trap to go back to the VMM. **(4)** The VMM would then do the real return from trap to the process on the hosted OS. **(5)** Which would then finally, resume execution.
- To allow for virtualizing of memory, another layer of virtualization needs to be added so that multiple OSes can share the actual physical memory. This extra layer of virtualization makes "physical" memory a virtualization on top of what the VMM refers to as **machine memory**. Each OS maps virtual-to-physical addresses via its per-process page tables; the VMM maps the resulting physical mappings to underlying machine addresses via its per-OS page tables.
- The steps are as follows. **(1)** When a process on the hosted OS Loads something from memory, it would be a TLB miss, which would result in a trap. **(2)** This trap is handled by the VMM which would call into the actual OS' TLB handler. **(3)** The OS TLB miss handler would extract the VPN from the VA, do a page lookup and get the PFN, if valid and update the TLB. **(4)** The VMM would now update the TLB because the OS, during "boot", tried to install its own trap handlers. The OS TLB miss handler then runs, does a page table lookup for the VPN in question, and tries to install the VPN-to-PFN mapping in the TLB. However, doing so is a privileged operation, and thus causes another trap into the VMM. At this point, the VMM plays its trick: instead of installing the OS's VPN- to-PFN mapping, the VMM installs its desired VPN-to-MFN mapping. After doing so, the system eventually gets back to the user-level code, which retries the instruction, and results in a TLB hit, fetching the data from the machine frame where the data resides. **(5)** These per-machine page tables need to be consulted in the VMM TLB miss handler in order to determine which machine page a particular "physical" page maps to, and even, for example, if it is present in machine memory at the current time, to reduce the cost associated with TLB misses, a software TLB was introduced. **(6)** The return from trap in the previous results in the VMM returning a return from trap to the hosted OS, which then **(7)** resumes execution.
- The VMM first consults its software TLB to see if it has seen this virtual-to-physical mapping be- fore, and what the VMM's desired virtual-to-machine mapping should be. If the VMM finds the translation in its software TLB, it simply installs the virtual-to-machine mapping directly into the hardware TLB, and thus skips all the back and forth in the control flow above
- When a VMM is running underneath two different OSes, one in the idle loop and one usefully running user processes, it would be useful for the VMM to know that one OS is idle so it can give more CPU time to the OS doing useful work.
- **Demand Zeroing**: The reason for doing so is simple: security. If the OS gave one process a page that another had been using *without* zeroing it, an information leak across processes could occur, thus potentially leak- ing sensitive information. Unfortunately, the VMM must zero pages that it gives to each OS, for the same reason, and thus many times a page will be zeroed twice, once by the VMM when assigning it to an OS, and once by the OS when assigning it to a process.