Friday, December 7, 2018

CH 38, 39, 40, 41, 42

CH 38 – RAID

- **Redundant Array of Inexpensive disk** provide several advantages over a normal disk, them being – **performance**, **reliability**, and **capacity**. It provides all these benefits *transparently*, meaning that to the OS it would still act as any other disk which provides the added benefit of *deployability* which helps to easily replace existing disks with RAID.
- With RAID, the internals are quite complex, it has a **micro controller** that runs firmware to direct the operations of RAID, **volatile memory** (DRAM) to buffer data blocks to be read/written. Alternatively, **non-volatile memory** to buffer writes, an even specialized parity calculation. Finally, **disks** for storage.
- Fault model being considered in this chapter is **fail-stop**. Meaning, that a disk can be either working or failed. No intermediate states are possible. Hence when we evaluate the different approaches of building a RAID< we will look into three aspects – **Capacity**, **Reliability** and **Performance**.
- **RAID 0 (Striping)**: In this form of RAID, the data is striped across different disks. This striping is based on how much chunk is to be placed on a single disk. RAID 0 provides an upper bound in **performance** and **capacity**. To get the location of the Disk and Offset for the blocks and getting to know where they are stored in the RAID we would use the following equation to get the disk number – $Disk = A \% number\_of\_disks$, and to calculate the offset we use the following equation $Offset = \frac{A}{number\_of\_disks}$ where A is the logical block address. Smaller chunk size would improve parallelism of reads and writes to a single file. While, larger chunks would achieve a higher throughput.
  - RAID 0 Analysis – Capacity provided is excellent, as it sums to N disks each of size B blocks, which delvers $N . B$ blocks of useful capacity. Reliability is excellent too, however, it should be noted that any disk failure would lead to complete data loss. Finally performance is excellent both in terms of Random and Sequential workloads.
- **RAID 1 (Mirroring)**: In this form of RAID, there is a mirroring that happens across the disks. This results in a lower capacity available for use but higher reliability as compared to RAID 0. Also, when considering the performance, writes are slower – $(\frac{N}{2} . S)/(\frac{N}{2} . R)$ where S and R are sequential and random workloads, because the written need to be replicated to another disk, to which the current disk copies to. Also, RAID 1, can handle 1 disk failure to up to $\frac{N}{2}$ disk failures, depending on which disk it is. Finally, the read performance depends on what type of read operation is being carried out. If it sequential reads $(\frac{N}{2} . S) MB/s$, it would not perform any better than other disks, but in case of random reads $(N . R) MB/s$, the operation can split among the N disks, and hence achieve a higher reading speed. Also, since the writes here are being made to two disk, there is a possibility of something bad happening before both the writes are completed, one way to solve this problem is to make use of a **write-ahead log**, which would keep a log of what the RAID is about to do.
- **RAID 4 (Saving Space with Parity)**: In this form of RAID we have an extra disk which would save the parity information pertaining to each of the corresponding blocks. The parity would be calculated by using a simple XOR function which would return 1, if there are even number of 1's and 0 otherwise. Though this method helps in saving space, the tradeoff here is the additional performance that would be incurred because of parity calculation. Also, this parity information can be used to reconstruct data if *at most* one disk is damaged (Reliability). The capacity associated with RAID 4 is $(N - 1) . B$. Finally, when considering performance, the sequential read performance would be $(N - 1) . S$ MB/s. In case of sequential writes, RAID 4 can perform full-stripe write which would mean that it would calculate the Parity as soon as it receives the write information, and then parallely write all the information to the disk $(N - 1) . S$ MB/s. Random Reads, would be similar to sequential reads, as in it wouldn't consider the parity disk for read performance calculation. Random Writes can be performed using two methods, **Additive parity** or **subtractive parity** method. In the additive method, you would read in all the value from the blocks under consideration and then use them to calculate the parity information. This is compute heavy. Subtractive method makes use of the block that needs to be overwritten, ad compare it with the

current information at hand, if they're same the parity block would remain the same, if not the parity bit would have to be flipped, that is, $P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old}$. The problem with RAID 4 is that, even if accessing the disks in parallel is possible, update to the Parity Disk cannot be made parallel, in cases, where small writes are made to different disks. This is known as the **small-write problem**.

- **RAID 5 (Rotating Parity)**: In this form of RAID, it is identical to RAID 4, except for the fact, that the parity information of each block is rotated amongst the different disks. This results in better random read performance, as we can now utilize all the disks. Finally, there is a noticeable improvement in random writes as it now allows for parallelism.

CH 39 – Interlude: Files and Directories

- The files in the system have an associated low-level name called the inode number. Just like Files, the Directories also have a low-level name, but its content are quite specific – user-readable name, low-level name. Note that, each entry in a directory either refers to another file or some other directory. One can place directories within directories to form trees, just note that a directory of name **foo** cannot have a file with the same name.
- **Creating Files**: This is done by using the *open()* call, which takes in several different parameters. *O_CREAT* to create file if it does not exist. *O_WRONLY* would be to ensure the file is Write only. *O_TRUNC* would be to truncate the size of the file to zero if it already exists, thus removing any existing content. *S_IRUSR*, *S_IWUSR* would be to associate permissions. One important aspect of *open()* is what it returns – **File Descriptor** which would be integer, private per process which would be used to access the files.
- **Reading and Writing Files**: Using something as simple as the *cat* command to print the content of the files on the terminal requires lots of processing at the backend. **First**, the file would be opened, this would return the file descriptor integer. **Second**, the file descriptor would then be used to read the file, the first parameter would be the FD. Please note that, every file has three open files – **Standard input**, **Standard Output**, and **Standard Error**, hence the FD for a file cannot be 0, 1, or 2. The next parameter would be the buffer that would be used to store the content that would be printed on the screen, and the their parameter would be the buffer size. **Third**, the content would be written on the terminal using the *write* command, which would use the FD as 1, as described above. To which it would write out the content (2nd parameter), the third param would be the size of the content being written. Once complete, the *read* function would be run again, for more content, if not, it would simply return *0* thereby calling the *close* function.
- While reading a file if you need to jump in the buffer to read specific parts or to find something in a file, the *lseek(FD, offset, whence)* system call can be used in which whence is used to describe how the seek [**Not DISK SEEK**] would take place. Hence, once *lseek* is used (to jump to offset 200), and then we use the *read* with the buffer size as set to *50,* we would jump to 250 position in the buffer.
- When writing data to a file, the data is first written out to a buffer, which due to performance reasons would then write the data out to the actual file later. In cases, where this eventual guarantee is not enough $fsync(int\ fd)$ can be used to force the writing immediately.
- Cannot edit the low-level info of directory, it would be done when some file inside the directory is edited. When we **create** a new file, what we are doing is **linking** a file to the directory, and hence to delete the file from the directory we need to call the ***unlink*** function to delete the connection between the file and the directory. When we use the *ln* (**link**) command to link two files, if you edit one of them, the other would have the same content. If you remove one of the files, the other file would still exist. However, if you use **symbolic links**, it would have different inode number, plus if you remove the file that is being linked, the linked file would cease to exist but still appear in the console. Spooky!
- **Permission bits** – It is represented by 10 bits, the first bit is to show the type of the file *f, d, L* for file, directory and symbolic link respectively. The next 9 bits are for permissions, 3 bits each for the Owner, Group and Others. **Read is specified by 4, Write by 2, and Executable by 1**.

CH 40 – File System (FS) Implementation

- Two basic components of a FS are data structures, and access methods. The on-disk organization of data structure of the FS is as follows. First the disk is divided into **blocks**. Each of these blocks would be of fixed sized. The next thing on a disk is the **user data**, for which space would be reserved. For all the files and directories in the disk, we need a data structure to store information about it so that they can be easily accessed. This information is stored in the **inode** or **inode table**. Next we need information about whether each *inode* or *data region* are free (0) or allocated (1), for this we would use **inode bitmap** and **data bitmap**. Finally, there would be another block that would be allocated to a **Superblock** which would give the information about the number of *inodes* and *data blocks* in the FS.
- Inode mainly consists of all the associate metadata that would help describe a file – including but not limited to time of modification, type, size, number of blocks allocated, protection etc. $block = (inumber * sizeof(inode\_t)) / blockSize$
  $sector = ((blk * blockSize) + inodeStartAddr) / sectorSize.$ Indirect pointers can be used to point towards aa block with additional pointers. To support even larger files we can have double-indirect pointers.
- **Reading a file from the disk** – For this one needs to start at the very start, that is, the root directory, but even for that inode number would be needed, but is already defined and known by the OS, once you've read the inode for the Root, the OS then reads the data content of the root inode, and then it would get the information of the foo directory. Then, the OS would read the content of the Foo inode, to get information about the bar file in the foo directory, for which would have to read the data content of the foo directory, once it has read the bar inode, it would then start reading the data at the different blocks, and then concurrently update the last-accessed value in the bar inode with a write call, this would be repeated until all the data in the bar file has been read.
- **Writing To disk**: Writing to a disk is a much more complex process, which would involve at least 5 I/O operations. First would be read the data bitmap, the second would be to update the data bitmap, the third and fourth world be to read and then write the inode, and finally one would be to write the data content itself. The traversal of the path to the foo directory would be same as above, once there the OS would have to update the inode bitmap with a read and a write call to update the task of creating a file, and then concurrently writing in the foo's data to reflect the file. Then OS would read and write the bar's inode to reflect that the file bar has been created, and reflect his change in the foo inode with the size or any other information. The next steps would be to first read thee bar's inode, and then read and update the data bitmap followed by writing the content to the bar's data blocks, followed by an update to the bar's inode to update the associated metadata.

CH 41 – Locality and the Fast File System

- The concept of FFS is similar to that described in the previous chapter, the only difference is that the tracks that equidistant from the center across multiple disk would belong to a single Cylinder Group, and each of these groups would have the information about the **inode bitmap**, **data bitmap**, **data**, and the **superblock**.
- FFS allocates directories in the following way. It would store a directory in a cylinder with the least number of directories and a high number of free inodes, and put the directory data and inode in that group. This would help balance directories across groups and also be able to allocate a bunch of files.
- With files, FFS tries to do two things, it would make sure to allocate the data blocks of a file in the same group to avoid longer seek times. Next it places all the files that are in the same directory in the cylinder group of the directory they are in
- One thing to keep in mind is about the large files. In the current aspect of things if we encounter a very large file it would occupy almost the entire group, thus leaving no space for other related files. Hence the solution to this problem would be to divide this data amongst different groups in a way that seek time does not affect the sequential read time. Hence, the way to do this would be select a chunk size that is large enough to amortize the costs associated with the seek of data.
- Finally, FFS introduced a new disk layout, in which sectors were alternatively placed, to allow for more time for the disk to respond to a read request which previously could've been missed if the files were placed in consecutive sectors. Doing this allowed FFS to avoid the additional overhead that would be associated with the rotation of the disk if it had missed the first read. There were other usability improvements as well such as long names (as opposed to the traditional 8 characters). Finally, a new concept was introduced – symbolic links; which allowed the user to create an "alias" to any other file or directory on the system and are much more flexible than the hard links. Also, FFS introduced an atomic *rename()* operation for renaming files.