Exploring Error Checking and Correction Protocols

HAORAN YU

Data may become corrupted under many situations. For example, while transmitting a message over the network, noise could introduce error making the message unreadable by the receiver; a disk storing files could fail and lose part or all of the files. In this project, I explored and modified the Xv6 kernel slightly, to allow services such as checking for error in corrupted data and correcting the error. At the user level, multiple system calls were introduced to perform distinct services at the kernel level. Several protocols for error correction were implemented, including RAID level 1 and RAID level 4. An attempt was made to implement Reed-Solomon erasure code, but it was not successful.

ACM Reference Format:
Haoran Yu. 2010. Exploring Error Checking and Correction Protocols. *ACM Comput. Entertain.* 9, 4, Article 39 (March 2010), 4 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

To guarantee reliable service of modern computers, data must not be corrupted and remain undetected. Systems need to follow error-checking protocols to monitor for error in files being read from disk or from the network. Once the error is discovered, an efficient protocol should be able to correct the error using a reasonable amount of computational resources. Erasure code is a method to check and correct error by storing redundant data derived from the primary file data. The redundant data contain sufficient amount of information for an algorithm to determine whether there is an error in the primary data. Erasure code transforms a file of size k to a file of size n. The redundant data is of size n-k, the size varies depending on the erasure code protocol used.

Optimal erasure code can recover an error in a file of size k using only the redundant n-k data. There are many optimal erasure codes implemented, one such code is parity. Parity is binary data computed from performing bit-level operation on the primary file data. For example, given two blocks of primary data, an XOR operation can be performed between the two blocks to compute the parity data. This technique can be used for checking large amount of data at a coarse-granularity level. Another optimal erasure code technique is Reed-Solomon, which computes the n-k data by performing matrix multiplication between the primary data and a predefined encoding matrix. The n-k data is appended to the primary data, and error can be detected by performing another matrix multiplication between the data of size n with a decoding matrix. An example of error checking using Reed-Solomon erasure code will be given later in this report.

Author's address: Haoran Yu, haleyew@cs.ubc.ca.

2 IMPLEMENTATION

Since this project did not have a specific research question to begin with, I will not talk about the implementation instead. I will also discuss the system design principles that I applied and what desired properties does the principles imply.

A total of three protocols were explored and were partially implemented. The first two are closely related to redundant array of independent disks (RAID). RAID techniques create redundant data on backup disks, and restores from the backup disks when primary data disks fail and lose data. Two less-common RAID protocols, RAID level 1 and RAID level 4, are discussed below. RAID level 1 simply mirrors data from one disk to the other disk, so that file stored on disk1 has an exact copy on disk2. RAID level 4 breaks a file of more than a block large into multiple blocks, and stores each block on separate disks. For example, if a file contains 2 blocks and 2 primary data disks are available, then store the first block on disk1 and the second block on disk2. There is also a backup disk to store the redundant data. As discussed above, an XOR operation can be performed between the two blocks to create parity data. The parity data is stored on the backup disk. When any of the three disks fail, simply use the other two disks to perform an XOR to restore the lost data on the failed disk.

The interesting question arises when considering the correlation between speed of recovery and how much computational resources were used. That is, as the number of backup disks used increase, the speed of recovery increases. In RAID level 4, if the number of blocks per file is very large and we have a large number disks, then we can use RAID level 5 to increase performance. For each pair of blocks, we perform an XOR operation to create the parity data and store it on alternating disks rather on a single backup disk. This way, if multiple blocks need to be restored, we can avoid queueing the request on the same backup disk because parity data number 1 is stored on diskA and parity data number 2 is stored on diskB. Unfortunately, I could not implement RAID 5 because I could not figure out how to create a large number of virtual disks, this is discussed next.

To implement the first two protocols, I ran Xv6 on an emulation layer, QEMU. I am running QEMU on Ubuntu, on a 64-bit architecture machine of Intel i5 with 4 cores. I compile Xv6 and load the operating system as an image on QEMU. Any files stored on the filesystem are stored on a second virtual disk, so a total of 2 virtual disks were used. I created 2 extra virtual disks by creating 2 more images, this is done in the Makefile. I then loaded the 2 images as disks by modifying the QEMU options. A total of 4 virtual disks were used, but I cannot load a large number of virtual disks on QEMU due to the limit of maximum number of virtual disks allowed. One possible solution is to edit a system file on Ubuntu /etc/security/limits.d to increase the limit but it might not be safe, therefore I did not increase the limit and only used 4 virtual disks. However, three files disks is sufficient to demonstrate RAID level 4.

In Xv6, I added additional system calls to allow the user to specify which disk to write a file to. I duplicated the open() system call and added an extra parameter, the device number. Procedures below the system call look up the parent directory of the specified file and return an in-memory copy of the inode, and if the CREATE flag is set at the system call then create the file and return the inode. Although initially I thought that Xv6 is a message-driven kernel, but this is not the case because each procedure simply calls another procedure. Therefore the different procedure calls could be interleaved and cause concurrency issues. Access to an inode must be limited so that only one process modify it at a time. When using an inode, first lock it, and when finished using the inode, release the lock. It is also interesting to observe how inodes are recycled using a free list. When a new inode is returned upon file creation, I can modify the contents of the struct. For example, I can change the struct member ip->dev to the device I want the file to be written to.

The kernel procedure pushes the contents of the in-memory inode to disk, and therefore will be pushing to the correct disk.

To demonstrate how RAID level 1 works, I write the file to disk1 at the user level, then I write the same file to disk2. When something wrong happens to disk1‘s file, I simply read the file from disk2 and write the file to disk1 again. All of the recovery process happens at the user level, and this follows the end-to-end argument. The kernel is just an intermediate step for transmitting a file from the user to disk, and thus does not make any decisions for the user.

Next, to demonstrate RAID level 4, I added additional system calls. Each system call presented to the user can only be found in a header file at the user level specifying the data type of each parameter. The SYSCALL assembly code pushes the data to the user process stack, which is popped from the stack at the kernel. The kernel procedure knows what each of the parameter means because of the order that the data appears on the stack. For example, the syscall open() knows that the first item is a pointer to char and it specifies the path of the file. Each syscall has a number, which is registered in a syscall table. The table stores the corresponding kernel procedure that services that particular system call.

I created a syscall to initialize block striping in the kernel, and then another syscall to perform block striping at the kernel using the initialized parameters. A third syscall, restore() is implemented without specifying how to restore the file because the parameters are already set at the kernel level, therefore the kernel already knows how to restore the file. This is the principle of separation of policy from mechanisms. At the user level, the user specifies the policy for the kernel to follow when restoring a specific file, this information is passed to the kernel. The kernel only offers the mechanism for restoring a file from disks, the user specifies what disks are used to restore the file. For example, in the demo I created, the user breaks the file into blocks, combines all the odd-numbered blocks together and all the even-numbered blocks together. Then the user passes parameters to the kernel to specify that the given file uses disk1 and disk3 as data disks and disk2 as the parity disk. Then the two parts of the file are created and written to disk1 and disk3. Next, a syscall is requested to build the parity data, the kernel then performs XOR between data on disk1 and disk3 and stores the parity on disk2. Upon failure of disk1, the user simply calls restore(), specifying which disk failed. The kernel knows how to restore the data for disk1 because of the policy specified by the user upon file creation.

All implemented system calls interact closely with the filesystem and do not perform unsafe operations such as trying to read contents of a recycled page. As it seems, when a page is recycled, the content of the page is not zeroed out by Xv6‘s virtual memory management. I manually zero out each buffer I allocate on the stack, so that procedures read and write correct data from the buffers. Lastly, I note that it is much easier to work with inodes than working with files. An inode contains all the information I need about a file, but each read and write operation much be part of a syscall transaction.

Lastly, I will discuss the third protocol for error detection and error correction. In the first two protocols, I simply assumed that the user knows when there is an error in the data, and perform error correction by executing the RAID protocols. However, it is possible to detect error using RAID as well. For example, for RAID level 4, when a file is read from disk1, I must perform XOR using the other two disks to check that the result computed by the XOR is the same as disk1‘s data. If one disk contains error, additional efforts are needed to detect which disk contains the error. The Reed-Solomon erasure coding can both check for error and correct error using the primary data of size k and backup data of size n-k.

To explain Reed-Solomon, we need to define symbol and finite field. If a symbol has a size of 8-bits then the symbol is called a byte, the symbol has 256 unique values. A symbol of size 3 bits has 8 unique values, and the unique values define the finite field. Using the example of 3-bit symbol, a message of size 15 bits is broken into 5 pieces, each piece is 3-bits. Then based on the position that the piece appears in the message, it is encoded as a monomial term in a polynomial, where the coefficient of the term is the value of the 3-bit symbol and the exponent is the position of the piece in the message. Next, we define another polynomial, the generator polynomial, which is used to generate the parity data. We perform a matrix operation between the message polynomial and the predefined generator polynomial. The parity of size n-k is created and added to the message. To perform error checking, perform the matrix operation between the n-size message-plus-parity with the generator polynomial again. If there are no error, the result of the operation should return 0, otherwise we have an error and the result is non-zero. To correct the error, many complex steps were taken, but there is a shortcut if the symbol size and the message size are both small. We can store a lookup table, with the key being the error returned by the Reed-Solomon error checking and the value being the solution of how to correct the error. The reasoning is that for Reed-Solomon, each error is unique relative to the position of the error in the message as well as the unique value that the original message at that position was mutated to. Although with this simplification, I still found it difficult to implement Reed-Solomon in Xv6.

A simplified view of Reed-Solomon encoding is to perform long division between the message polynomial and the generator polynomial, with the XOR operation to compute the remainder at each shifting step in the division algorithm. This simplification allows me to compare Reed-Solomon with Circular Redundancy Check. Both perform polynomial divisions between the message and the generator polynomial. However, due to the simplification, I can only detect error and I do not have enough information to correct the error. In my demo, I show that a message I read from disk is encoded using the generator polynomial and the backup data is appended. The n-sized message is sent to the user. Next, I detect an error at the user level. Assuming that the error occurred during message transmission, to correctly the error, I simply read the data from disk again. Note that the error detection protocol in Reed-Solomon can be implemented in addition to the RAID error-correction protocols to allow both error checking and correction.

3 CONCLUSION

To conclude this report, I explored the filesystem aspects of the Xv6 kernel, made modifications to allow error checking protocol such as Reed-Solomon and error correction protocol such as RAID level 1 and 4 to take place. The lesson learned from this project is that the small Xv6 kernel has very limited functionality. The user library has only a dozen of libc procedures, and the kernel also has the minimal procedures to permit debugging via console logging. Therefore for a kernel to work well, it should have higher complexity. Due to the limited time spent on the project, no performance evaluation were done. However, it would be desirable to measure how long it takes to perform the XOR operations, the error checking procedure, as well as the recover() system call relative to normal file read/write time.

4 ACKNOWLEDGEMENT

Special thanks to Stewart Grant for suggesting a project on RAID, and a great thank you to Andy Warfield for teaching this thought-provoking course.