CSE 506: Operating Systems
Make the File System writable and Crash tolerant using Journaling

Writable File System

We have implemented our own version of the writable file system. We started off from File Systems readable code and implemented in the branch “Lab 7”.

BitMap

Bitmap guides us to whether a disk block is free or in use. We have considered 3rd block in the file system to maintain the Bitmap. We are clearing the values in blocks 0 and 1. We allocate a new block to write to a file, by getting an available block and clearing the corresponding Bitmap entry for that file block.

Dirty Blocks

We are making use of the PTE_D "dirty" bit to keep track of dirty pages in the block cache. This means there has been some modification to the page and it has not been written back to the file. After writing this back to disk we make the page Undirty again.

File Creation

We create a file by passing the O_CREAT argument. We initially walk the path of the directory structure to locate the location of its creation. If available, we associate a file name to an existing file structure in that directory. We initialize its size to zero and assign the file type as a file.

File Truncate

We truncate a file by passing the O_TRUNC argument. Truncating a file means truncating all the blocks allocated to it. We start truncating from the offset of the file specified as an argument. If 0 is specified we will remove all the blocks of the file.

Requests from Client to File System

We have implemented the following additional requests by the client

- FSREQ_SET_SIZE
    - Sets the size metadata parameter for the file

- FSREQ_WRITE
    - We copy the contents of the required buffer into a file whose position is at the offset specified.If the write crosses the block size, then we allocate a new block and write the remaining contents of the buffer into that file.

- FSREQ_FLUSH
    - We use this to flush the changes made to a file to the disk. We initially check the corresponding dirty pages of the file. If dirty, then we calculate the appropriate sector number of the block and write to it. After it is successfully written we undirty the page again so that it can be used again.

- FSREQ_REMOVE
    - This is used to remove a file or truncate a file. To achieve this we are removing all the blocks of the file. We do this by clearing the bitmap entries of both the direct and indirect blocks.

- FSREQ_SYNC
    - This is used to write all dirty pages accross the file system to the disk. All the dirty pages are flushed to the disk one by one.

These are used by the client to communicate to the File System by IPC requests. We have implemented corresponding handlers to them.

Test Scripts to test Writable

We have implemented the following test scripts to check the working of the writable file system.

1. user/testfile.c

We are checking the below functionalities in this file. We are initially redoing the readable checks to make sure the writable code has not corrupted earlier functionality.

a. Check read for a non existing file

The file 'not-found' when opened with O_RDONLY should return -E_NOT_FOUND. This file is not present in the directory path.

b. Check File read for the file 'newmotd'

Open file 'newmotd' in O_RDONLY mode. Check the File stat parameters. Read the file and write to the buffer. Compare and check whether the file read is good. If good return 'file_read' is good and close the file.

c. Create a new File

Open a new file called 'new-file' using O_CREAT | O_RDWR

d. Check File Write

Write the content of the buffer to the file. Compare the contents of the file to the contents. If they match display "file write is good"

e. Check File Write for Big Files with Indirect blocks

We create a big file with the arguments O_CREAT | O_WRONLY. Compare the contents of the buffer and contents of the big file by reading. If they match, the file write is correct and print "large file is good"

From the above tests we conclude that the writable file system is working in our version of JOS.

Making the File System Crash Tolerant using Journaling

If a crash occurs during the File Write, it may make the File system inconsistent. If the inconsistency exists we will experience abnormalities in future file operations or it will make few blocks unusable.

To solve this problem and to make the File system tolerant we will make use of Journaling. In journaling we will write all the file operations we conducted into a log. This journal log helps us to check all operations before a crash until a previously saved checkpoint. Based on the actions we can reverse or redo the change to the disk and make the file system consistent again.

To make the File system consistent our main aim is to preserve the metadata of a file in the journal log. Next in priority is to maintain data consistency and the data can be written to the journal or to a separate log. If a crash occurs the user may lose some or none of the data, but the disk or the file system will be in a consistent state. We can achieve this and go quickly to a consistent state by undoing or redoing transactions from the journal log.

Journal Structure:

Our implemented journal is a **Circular Log**. This is in line to most of the implementation where the log is not ever growing, instead overwrites after it reaches the end of the journal log. To not overwrite required journal records for recovery, we are allocating two blocks of space to the journal file. This space is allocated on the last two block of the File system, the 10238th and 10239th block.

Reading the Journal

To maintain the circular structure, we maintain two pointers for the journal; namely the start and the end. The start pointer keeps track of the point from which a journal is read in case of a crash. The end pointer specifies till where it has to be read. Initially the start pointer is pointing to the start of the journal. After we check point (Flush all the dirty pages to the disk) we iterate the start pointer to the location of the checkpoint. The start and end of the journal file is very similar to queue start and end i.e start is 0 and end is 0 initially and end+=sizeof(jrdwr_t) on addition of every entry.

Journal Structure

The below is sample of our journal record displayed in on the console.

```
CheckPointing
0:1:1017fa00
2:1:1017fa00:442
7:1:1017fa00:442
5:1:18000:512:98304
4:1:1017fa00:98816
8:0:1017fa00
CheckPointing
0:1:1017fa00
5:1:18200:512:98816
4:1:1017fa00:99328
8:0:1017fa00
CheckPointing
0:1:1017fa00
5:1:18400:512:99328
4:1:1017fa00:99840
8:0:1017fa00
CheckPointing
0:1:1017fa00
5:1:18600:512:99840
4:1:1017fa00:100352
8:0:1017fa00
CheckPointing
0:1:1017fa00
5:1:18800:512:100352
4:1:1017fa00:100864
8:0:1017fa00
CheckPointing
0:1:1017fa00
5:1:18a00:512:100864
4:1:1017fa00:101376
8:0:1017fa00
CheckPointing
0:1:1017fa00
```

Field 1

The first field indicates the type of file operation. The file operations are defined in an enum as follows.

```
typedef enum {
        JSTART,
        JREMOVE_FILE,
        JBITMAP_CLEAR,
        JBITMAP_SET,
        JSETSIZE,
        JWRITE,
        JCOMMIT,
        JASSIGN,
        JDONE,
} jtype_t;
```

The JSTART indicates the start of a file operation. Every file operation will have this. JdREMOVE_FILE indicates that the file has been removed.

The JREMOVE_FILE indicates when the file is removed. We achieve this by removing all the blocks of the file. The blocks are removed one by one. The Filename is initialized to null and the file record will be completely terminated.

JBITMAP_CLEAR indicates that the Bitmap has been cleared in the cases of truncate of file blocks or due to write discrepancies.

JBITMAP_SET indicates that the Bitmap has been set when a new file is written. One more instance is when we writing a big file. We write completely to the current block and and after its completion initialize to a new block and write into it. The Bitmap for the new block is set.

JFILE_SET_SIZE indicates a write has been completed to the file and the size has been set. When we write a new file, the size of the file is being set. If we append data to an existing file then this field is updated indicating that the append was successful.

JCOMMIT indicates that the changes have been written to disk. All the dirty pages of the current file are considered for this operation. The appropriate sector number is found and the content in updated using the ide_write. This operation takes a long time to complete compared to all the other operations as it has to update in the disk.

This operation happens only when the counter for the file is 0. Once this operation is complete, it is safe to assume that the file is committed to the disk and consistent.

JWRITE indicates that the write has completed. It indicates the offset and the length of the file actually written. Based on the change of the offset parameter and length we will able to determine if there was any data inconsistency in case of a crash.

JASSIGN indicates that a free block has been allocated from the Bitmap. But, it is yet to be pointed to the file pointer. The JASSIGN confirms that this block is indeed assign to a File pointer. There is a time gap when implemented, thus explaining the requirement of JASSIGN.

JDONE indicates that the file operation is complete.

FIELD 2

This field indicates the number of active transaction on a file. The count is incremented by 1 for every JSTART operation and decremented by 1 for every JEND operation. If the count is 0 at the end the JEND then we have the option to commit the file to the disk.

FIELD 3

This field indicates the File ID of the the current File operation. This field along with the count ensures the number of active transactions of that file.

FIELD 4

This field indicates the block number of the current file operation. This field is returned only for a few operation like when the Bitmap is cleared or when it is set. In case of JWRITE this field indicates the length of the written file.

FIELD 5

This field indicates the File Offset in case of a JWRITE. This field along with the length of the file is helpful in determining whether data consistency is present.

Checkpointing

This point indicates that all the dirty files across the file system is written to disk. At this point it is safe to assume that the file system is consistent. The start pointer point is incremented to this point as it will only consider operations for recovery only after the checkpoint. We are also occasionally stopping and checkpointing if required.

Journal Case Writes:

Each file operation will contain JSTART, which will be written to the journal. The value of the count of the file will be incremented for each JSTART. The count indicates the current number of active transactions on a file.

Case 1:

The first distinction comes if the crash happens before the transaction is written to the log. In this case we have no option but to skip the pending operation as nothing could be written in the journal.

If a crash occurs after the operation has been written to the log, then we have a chance to undo/redo the operations. As the operations types are less and the write back of the journal to the disk is important, we write the changes to the disk immediately.

Case 2:

Create a File

Creating a file is a single instruction. There are no specific cases. It is safe to checkpoint the file system at this point.

Case 3:

Write to a new File

Writing to new file after the file is created is a complex process. It involves three steps, namely

- Bit map is set
- Block is allocated
- Size value is updated

If the Bitmap is set and the block and size value is not present in the journal, then undo the Bitmap value. To undo, we have initialized the start pointer which is either at the beginning of the file or at the latest checkpoint. The journal is started to be checked from this point. We write into the journal on the initialization of the new Bitmap for this block, the alloc block and the size that is finally written to this block of the file.

Case 4:

If the Bitmap and the block is allocated but the size has not been set. We can enquire this from the Bitmap set and block statement in the journal. Then undo the allocated block and the bitmap value. Similarly again, we go to the start pointer which is either at the beginning of the file or at the latest checkpoint. The start point is manipulated such that it is either at the checkpoint or at the beginning of the file. We also return a failed result to the user application.

Case 5

If the block is allocated but if the corresponding bitmap is unset, then something went wrong in the previous crash and we have to clear the block by undoing the operation. To undo, we go to the start pointer which is either at the beginning of the file or at the latest checkpoint. The start point is manipulated such that it is either at the checkpoint or at the beginning of the file. We also return a failed result to the user application.

Case 6

Write to an Existing File

If we append text to an existing file and the size does not cross the block boundary. Then the size parameter is written to the journal file. If it is not present we simply skip the process and return a failed result to the user application.

Case 7

If we append text to an existing file and the file size crosses the block size. Then we allocate a new block to the file. We write into the journal on the initialization of the new Bitmap for this block, the alloc block and the size that is finally written to this block of the file.

Case 8

If the existing file crosses the block size boundary and a crash creates the block to extend the file but the bitmap is not set. In that case undo the block allocation. This is similar to the undo operation seen in Case 5.

Case 8

If the existing file crosses the block size boundary and a crash creates the block to extend the file and the bitmap is set but the file is not set. In that case undo the block allocation and setting the bitmap. This is similar to the undo operation seen in Case 4.

Case 9

Remove a File

To remove a File we ensure that all blocks of the file are remove. If the crash occurs in the middle of the removal, we make sure the remaining blocks are intact. The remove entry is written to the journal. Post this the journal end value will be present and the count of the active transactions on the file will be decremented by 1.

Note: We always make sure journal is committed to disk first before writing any other data to the disk.

Journaling Modes

We are able to Journal in two modes ASCII mode and BINARY mode. Ascii is human readable and binary is disk saving and program compatible. Please specify the JOURNAL_ISBINARY macro to false in case if you need human readable ascii storing of the journal and normal reading from the user space displays the journal in ascii.

```
We can Journal in either binary or ascii
  The format is as follows
  <modification-type>:<file-id>:<any-other-arguments-if-needed>
  modification-type: It specifies the type of change occured in the filesystem
                    that caused this entry (Look at enum ftype_t in fs/fs.h
                    for all the type).
  <any-other-arguments>: Please look at the entry specific arguments in jrdwr_t
                        in fs/fs.h for all the arguments.
```

Time Difference When Writing to Disk

We can observe a significant time difference when writing to disk compared to the memory. We are printing the journal in the console after completing all transactions. We are writing/* statements into the journal and flushing them to the disk individually.

Simulating a Crash

We are able to simulate a crash by opening the file CRASHFILEPATH. This file makes sure we Fsync everything first and then we execute a panic to simulate the crash.

Handling Simultaneous access to a File

We have designed a unique way to handle multiple active transaction in the same file. And to prevent the file from committing. We write a count value to the journal, which has the count of all active transaction on a file. Active transactions are nothing but, those with JSTART but not with the corresponding JEND entry. If the count to a file is 0 on an JEND operation only then the file can be committed.

Test Scripts to test Journaling

```
$make clean
$make
```

To test the writeable FS

```
$ make run-testfile-nox
```

To test Journaling

(which is exposed to the user for the observing purpose)

```
$ make run-jprintjournal-nox
```

To test the creation of the new files

```
$ make run-jaftercreate-nox
```

To remove ‘new-file’

```
$make run-jremovefiletest-nox
```

To check the removal of ‘new-file’

```
$make run-jafterremf-nox
```