PROJECT REPORT

ON

A CUSTOM FILE SYSTEM FOR IN MEMORY DISK

Submitted by: Jayesh Kshirsagar (153050013) Pratik Satapathy(153050036)

1) PROBLEM DESCRIPTION AND GOALS

Development of a custom file system, which can operate on an in-memory disk.

Building an in memory disk:

Allocation or RAM memory for the in memory disk

Copy partition information to the disk

Format one partition of the disk, using a developed program

Build a device driver for accessing the disk

Building a custom file system implementing VFS standards:

Creation of custom superblock and inode structure.

Overriding of file operations, inode operations and superblock operations to interact with the in-memory disk.

2) DESIGN/IMPLEMENTATION CHALLENGES

→Exploration of how a disk is partitioned by partition tables, structure of partition tables.

→Working of a block device driver which will handle the in memory disk

→Exploration of how the custom developed filesystem will connect to the in memory disk(registering the file system and mounting it with the block device).

→Exploration on how to format a partition of the disk with a file system (storing superblock related information on the disk to be read by the file system when mounted)

→Storage organisation of metadata information of stored files on the disk.(what is the content of first block..second block….on wards)

→ How to make the file system persistent over mount ,unmount operation.

DESIGN

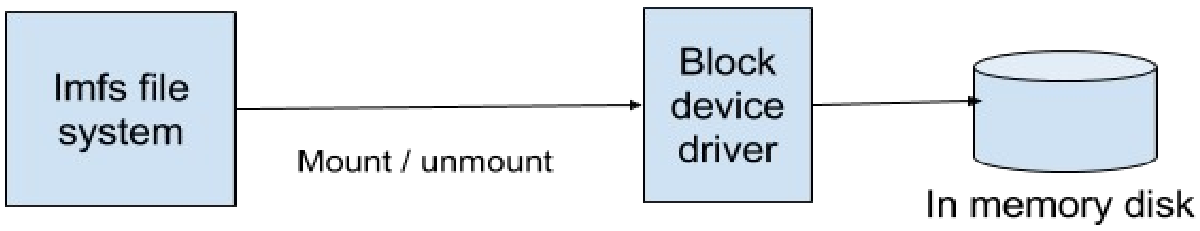
1) **IN MEMORY DISK DESIGN:**

A block device is registered with block device operation implemented.

At block device driver init, a block of memory is allocated using vmalloc.

We create a gendisk instance by setting all the properties of the allocated memory. We initialize the gendisk -> queue to a custom method which handles read and write to the memory allocated for disk.

High level overview of the setup



Partition table information is written to first block of the disk.

On successful creation of disk we format one of the partition of the disk as per our custom file system metadata using a custom partition format program.

This generates the following metadata organisation structure on the formatted partition

Metadata organisation structure on the formatted disk partition

Size of blocks : 4KB

Block 0 **[meta data]**	Block 1 **[meta data]**	Block 2 **[meta data]**	Block 3 **[user data]**	Block 4 **[user data]**
Imfs Superblock Structure Store (contains custom superblock structure)	Imfs all Inode structure store (contains all custom inode structures that are present)	Root inode Data block (contains name to inode num mapping of child files to root directory)	Data block of inode 1	Data block of inode 2

A creation of new file affects each of the meta data block and the particular block where its data will be stored.

2) IMFS FILE SYSTEM DESIGN:

The developed file system supports

- Creation of one or more files
- Read operation of created files
- List files

Metadata block1: File system mounts by reading the superblock data from the 1st block of the partition. Superblock contains file system level details such as block size,file count, free block lists etc.

Metadata block2: This block contains inode structures of all the files present. Each inode structure has the mapping of inode number and its corresponding data block. Any newly created file's inode structure needs to be stored at the second block. Its data is to be stored at the corresponding data block number of its inode structure.

Metadata block3: A name to inode number list is to be maintained at block 3 for ls operation. Any new file's entry has to be made at block 3 for "ls" operation.

User blocks: blocks from block number 4 onwards are used for user data storage, the first user level file's data is kept at block 4. The second user level file's data is kept at block 5 and so on..

This also states that the number of files in the system is capped by the number of inode structures in block 2 (i.e. blocksize / sizeof (inode_structure))

As only one data block is allotted per file, the file content can not exceed one block size i.e **4KB**

IMPLEMENTATION DETAILS:

These are main components of implementation :

1) Implementation of in-memory disk: For this we referred the code at http://opensourceforu.com/2012/02/device-drivers-disk-on-ram-block-drivers/. This pointer was provided in project list. We had to do several modifications and corrections in this code to make it usable for our purposes. Following are the related files:

- ram_device.h : This header file contains prototypes of functions which together implement the low level interface for the im-memory disk.

- ram_device.c : This is implmentation of ram_device.h. It defines the function which copy data to and from the in-memory disk.
- ram_block.c : This is the driver for this in-memory disk. It performs following important functions:
  - It initializes the ram_disk and registers it as block device to obtain the major number. Kernel represents each disk as struct gendisk. This module creates this struct, intializes request queue for this block device and also defines the request handler for the request queue.
  - In this way interfacing of this device with kernel is complete, and as a result it is shown /dev/ .
- Partition.c and partition.h : These programs create partitions on the ram_disk by creating a partition table at the end of boot record.

In summary, we create a in-memory representation of disk, specify a driver to handle I/O to this disk.

2) Implementation of a custom file system for this in-memory disk.

We call our file system 'imfs' (and hereafter refer it so). Following are the related files and functions:

- imfs.h : This gives declarations of on-disk VFS objects such as superblock and inode.
  - Struct imfs_disk_superblock : This keeps record of number of inodes on the disk and also the next free block.
  - Struct imfs_disk_inode: Main piece of imformation that this struct holds is the data block number which holds the data of this inode.
  - Struct imfs_dir_record: Data of the directories is represented by this struct. Each struct basically contains inode number and file name of one file in the directory.
  - Along with these structs following constants related to file system are also defined here:
    - BLOCK_SIZE : Logical unit of data transfer for the file system.
    - SB_BLOCK : Location of superblock on the disk.
    - INODE_STORE: Location of the data block which contains all the inodes in the imfs.

        - ROOTDIR: Location of rootdir on the disk.
        - MAX_INODES: Maximum number of files in the imfs.
- The ram-disk must be formatted to contain the superblock and root inode at their specified locations. This is performed by prep_dick.c, which is a simple C program and it utlizes normal I/O interface to write this information on ram_disk. This also serves as basick test if the disk I/O for ram_disk works correctly or not.
- Super.c : This is the implmentation of imfs. It defines struct file_system_type for imfs and registers the file system. Following is the detailed description of various structs and and functions in this file.
    - struct imfs_file_operations : This is file_operations struct for files in imfs,and it provides pointers to imfs_read and imfs_write functions for read and write on file respectively.
    - struct imfs_inode_ops : This is inode_operations struct for inodes in imfs,and it provides pointers to imfs_create and imfs_lookup and imfs_mkdir functions.
    - struct imfs_dir_operations : This is file_operations struct for directories in imfs,and it provides pointers to imfs_iterate function.
    - struct imfs_super_ops : This is superblock_operations struct for imfs,and it provides pointers to imfs_put_super function.
    - imfs_create: It creates new inode in the parent directory. For imfs, it involves creating a disk inode with appropriate parameters and storing it in inode_store on ram_disk.
    - imfs_mkdir: For now,it is placeholder function, however we can easily extend imfs to support directories.
    - imfs_lookup: This function performs lookup in the parent directory to check if the given child exists in it. For imfs, it involves finding datablock for parent inode, and going through all the imfs_dir_record structs to find entry for child.
    - imfs_read: For imfs, read involves fetching datablock for given file from ram_disk and passing it to user.
    - imfs_write: For imfs, write involves updating data block for the given file.
    - imfs_iterate: For imfs, it involves fetching data block for given directory and iterating through all it's records.

It can be observed that keeping all the metadata related to filesystem and individual files and directories is real challenge here.

Experimentation

We try to evaluate project on following main standards:

1) Correctness: To test correctness we performed following experiments:

- File I/O operations in mounted directory on single file using C program and using command line.

- Creation of multiple files, writing/reading to/from them.

- For all experiments, we verified that data is going to correct block using xxd command. e.g., data for first file should go into 4th disk block(or block number 3). We verified it using xxd on ram_disk.

- Listing directory contents.

- To check whether on-disk structures are being updated correctly we performed following experiment:
    - Mount the file system.
    - Create many files (say,5-6).
    - Write to them.
    - Unmount the file system.
    - Mount filesystem again.
    - Now all the previously creates files should be visible and holding correct content.

2) Boundary conditions:

We tested following boundary conditions.

- We specify maximum number of files which can exist on ram-disk and we test whether we can create more number of files than it. Imfs successfully rejects request to create files more than specified maximum.

- We allocate one block of size 4KB to each file or directory. We tested whether it is possible to write block of size greater than 4KB(say 6KB) to a file. In this case imfs successfully truncates block of size greater than 4KB.