CSC 714 : Project Report 2

Gayathri TK
gtambar@ncsu.edu

Jayush Luniya
jrluniya@ncsu.edu

RTFS : Real Time File System

Project Url

http://www4.ncsu.edu/~jrluniya/rt/

Introduction to Flash File Systems

An efficient and reliable file storage system is important in wireless sensor network so that data can be logged for later asynchronous delivery. In addition to this, incorporating a web server on the intermediate relay device in the sensor network provides a mechanism for allowing users to perform data collection using any standard browser. The web server uses a file system to store embedded web pages in RAM, flash, or on disk. File systems provide capabilities for changing web pages dynamically and maintaining dynamic objects.

Flash memory is a type of electrically erasable programmable read-only memory (eeprom). Because flash memories are nonvolatile and relatively dense, they are now used to store files and other persistent objects. Flash, like earlier eeprom devices, suffers from two limitations. First, bits can only be cleared by erasing a large block of memory. Second, each block can only sustain a limited number of erasures, after which it can no longer reliably store data. Due to these limitations, flash file systems should use sophisticated data structures and algorithms, which support efficient not-in-place updates of data, reduce the number of erasures, and level the wear of the blocks in the device.

RTFS Architecture

1. Assumption

- Once a file is created, the file will only be appended and data in the file will not be modified.
- To keep the filesystem simple, we specify an upper bound on the maximum file size and the number of entries in a directory (configurable).
- To provide time bounds on the filesystem access time, we also place an upper bound on the maximum depth of the filesystem.
- We don't need to maintain filesystem access rights, ownership, access timestamps etc in our filesystem.

2. Design Issues

- Not-In-Place Updates: Flash memory can be read or programmed a byte or a word at a time in a random access fashion, but it must be erased a "block" at a time. Once a byte has been programmed, it cannot be changed again until the entire block is erased. In other words, flash memory (specifically NOR flash) offers random-access read and programming operations, but cannot offer random-access rewrite or erase operations.
- Wear Leveling: Flash exhibits a unique property, in that each flash memory erase-unit can only endure a limited number of erasures, typically about 10,000 operations. Repeated erasures to the same unit will quickly exhaust the lifetime of a flash erase-unit. To ensure that no single flash unit reaches its lifetime limit before the rest of the flash erase-units, it is important to ensure that erase-write cycles are evenly distributed around the flash; a process normally called wear levelling.

- Erase Unit Reclamation: Over time, the flash device accumulates obsolete sectors and the number of free sectors decreases. To make space for new blocks and for updated blocks, obsolete sectors must be reclaimed. Since the only way to reclaim a sector is to erase an entire unit, reclamation (sometimes called *garbage collection*) operates on entire erase units.
- Efficient RAM Usage: Filesystems designed for small embedded systems must content with another challenge, extreme scarcity of resources, especially RAM. Hence the filesystem should be primitive file system with low metadata overhead.
- Filesystem Consistency: Traditional file systems provide satisfactory performance in the transfer of data, but power interruption or system failure can cause corruption of the file system, possibly rendering the device inoperative. Hence our filesystem should ensure that the filesystem remains consistent across power failures.
- Block Mapping: One approach to using flash memory is to treat it as a block device that allows fixed-size data blocks to be read and written, much like disk sectors. However, mapping the blocks onto flash addresses in a simple linear fashion presents two problems. First, some data blocks may be written to much more than others, which leads to frequently-used erase units wearing out quickly, slowing down access times, and eventually burning out. The second problem that the identity mapping poses is the inability to re-write data blocks smaller than a flash erase unit. Suppose that the data blocks that the file system uses are 4 KB each, and that flash erase units are 128 KB each. If 4 KB blocks are mapped to flash addresses using the identity mapping, writing a 4 KB block requires copying a 128 KB flash erase-unit to ram, overwriting the appropriate 4 KB region, erasing the flash erase unit, and rewriting it from ram. These problem can be addressed by using a more sophisticated block-to-flash mapping scheme and by moving around blocks.

3. Methodology

Our approach to the issues enlisted above was to either avoid the problem from occurring or by finding a simplistic solution to the problem.

- We deliberately excluded some common file-system features that we felt were not essential for embedded system, and which would have complicated the design or would have made the file system less efficient. Hence we won't maintain filesystem access rights, ownership, access timestamps etc in our filesystem.
- We have made the assumption that once a file is created, the file will only be appended and data in the file will not be modified. Further, we assume that a directory entry and file control block has fixed number of pointers to data block. This makes our file system metadata static and hence the metadata is never re-written into. Hence we avoid the problem of in-place updates and wear-levelling completely.
- Instead of using a linear block mapping technique our approach uses an efficient block apping technique as described in Section 3.1.
- Also, the filesystem efficiently utilizes the RAM as it stores just a virtual-to-physical mapping table explained in Section 3.1.
- Since we do not modify or create multiple copies of any existing data or metadata. Hence the filesystem consistency is ensured.
- Finally, the issue of erase-unit reclamation can be considered as future work and beyond the scope of the project. An efficient erase-unit reclamation algorithm can be easily incorporated in the existing design as we maintain information about the obsolete sectors.

3.1 Efficient Block Mapping Technique

The entire flash memory is divided into erase-units. Each erase-unit can be visualized to be composed of *fixed sized sectors*. Each sector can be uniquely identified by sector-number identifier. We also introduce the concept of virtual block numbers. Each sector is mapped to a virtual block number there-by abstracting the physical location of the sector. This enables file modifications and wear-leveling to be incorporated in the design if desired.

Data Structures for Mapping:

- Direct Map: Direct Maps facilitate efficient mapping of virtual blocks to sectors by maintaining arrays that store in the ith location the sector number that currently contains block i. Direct Maps are stored in RAM.
- Inverse Map: Inverse Maps allow efficient mapping of sectors to virtual blocks. This is achieved by storing the virtual block number in the sector header.

The indirect map is stored on the flash device itself and ensures that sectors can always be associated with the virtual blocks that they contain. The main use of the inverse map is to reconstruct a direct map during filesystem initialization. The direct map, which is stored in RAM, allows the system to quickly find the sector that contains a given block.

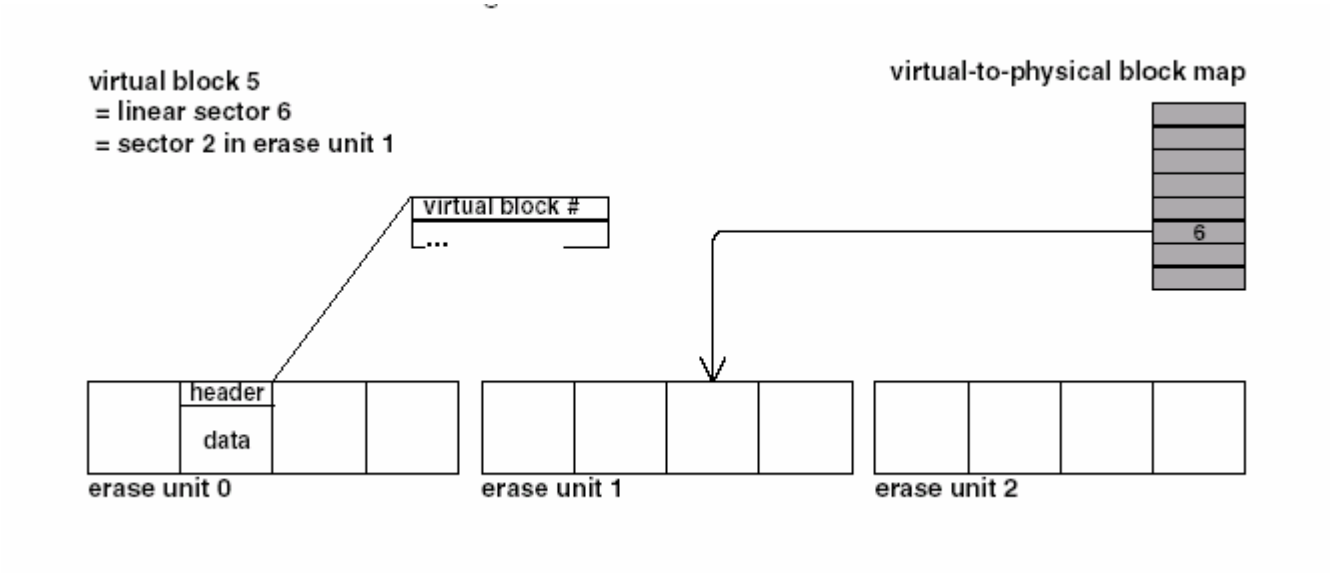

Figure 1: Block mapping in a flash device. The gray array on the left is the virtual block to physical sector direct map, residing in ram. Each physical sector contains a header and data. The header contains the index of the virtual block stored in the sector. The virtual block numbers in the headers of populated sectors constitute the inverse map, from which a direct map can be constructed.

Sector Types:

Each sector can store one of the following information

- Directory Control Block (DCB): This sector would store the directory name and the fixed list of virtual block pointers to the other directory control blocks or file control blocks. The value of the virtual block entries is initialized to all 1's (value after erasure of the erase-unit).
- File Control Block (FCB): Similar to DCB, FCB also store the filename and list of virtual block pointers to the data blocks. The value of the virtual block entries is initialized to all 1's (value after erasure of the erase-unit). Note that the file size is not stored hence avoiding the need to modify the FCB as the file is being written to. However, the size of the file can be calculated using the direct-map table at run-time.
- Data Block (DB): Data block just contains the data associated with the file. The data block which is not full can be identified by the value 0xFFFF (the value after erasure of the erase-unit). Hence, if the data contains 0xFFFF it should be delimited by an escape sequence. Using this we can calculate the file size at filesystem initialization.

Sector Header:

Field	Purpose
Used Bitmap	To identify if the sector is free, in-use or obsolete
Virtual Block Number	Inverse Map of Sector Number to Virtual Block Number
Block Type	Differentiate between DCB, FCB and DB

Used Bitmap	Virtual Block Number	Comments
0xFFFF	0xFFFF	Block is free
0xFFFF	Value != 0xFFFF	Block in use
0x0000	Don't Care	Block is obsolete

Direct Map Table Entry:

The direct map table is indexed by the virtual block number and contains the following fields.

Field	Purpose
Sector Index	Physical Sector Number corresponding to the Virtual Block Number
Sector Type	Sector Type DCB, FCB and DB
Data Size	Amount of valid data in the data block DB. Don't care for DCB and FCB.

Filesystem Initialization:

During filesystem initialization, the direct map table is allocated in RAM and the size of this table is equal to the total number of physical sectors. All the sectors are scanned sequentially and the direct map table is populated using the inverse map entry in each sector. Also if the sector type is a Data Block, the data block is scanned to find the amount of valid data in that data block. This information is also updated in the corresponding direct map table entry.

Filesystem Framework:

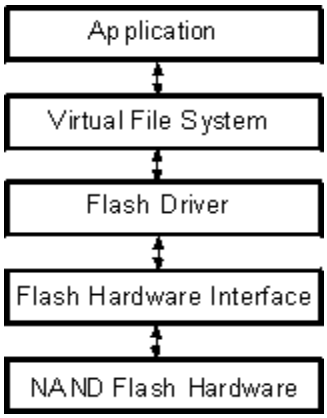

Figure 2: Filesystem Framework

Filesystem APIs:

We are currently working on deciding the Filesystem APIs that we will be implementing on the project and will be included in the next deliverable.

Individual Contributions

Since this project milestone involved working on the design of RTFS we chose to work together reading the literature about Flash File Systems, Renesas MC16 Architecture, coming up with the design for RTFS, and writing the project report.

Weekly Milestones

Week	Date	Milestone
Week 1	11/08/05	Decide upon filesystem APIs, data structures and implementation details
Week 2	11/15/05	Implementation of the Flash Driver Module
Week 3	11/22/05	Implementing File System APIs in the Virtual Filesystem Module
Week 4	11/25/05	Testing and Benchmarking

References

[1] FAT Filesystem http://users.iafrica.com/c/cq/cquirke/fat.htm

[2] Transaction-Safe FAT File System http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcemain4/html/cmcontransaction-safefatfilesystem.asp

[3] Reliable File Systems for Windows CE

[4] Flash Filesystems for Embedded Linux Systems http://www.linuxjournal.com/node/4678/print

[5] TargetFFS: An Embedded Flash File System http://www.blunkmicro.com/ffs.htm

[6] Renesas M16C Architecture http://www.renesasinteractive.com

[7] Algorithms and Data Structures for Flash Memories http://www.cs.tau.ac.il/~stoledo/Pubs/flash-survey.pdf

[8] A Transactional Flash File System for Microcontrollers http://www.cs.tau.ac.il/~stoledo/Pubs/usenix2005.pdf

[9] ELF: An Efficient Log-Structured Flash File System For Micro Sensor Nodes http://www.cs.colorado.edu/~rhan/Papers/sensys_elf_external.pdf