Jackson Paull

JHP2539

14 March 2025

ECE 381L - RTOS

Lab 4 Report

1. **OBJECTIVES**

Lab 4 was focused on the design and implementation of a nonvolatile filesystem that the OS can interface with via SSPI. This report details the implementation of a linux-based filesystem which scales incredibly well to large disk parititons.

1. **HARDWARE DESIGN**

None for this lab

1. **SOFTWARE DESIGN** - *Note: See GitHub, tag ‘lab4-release’ for source code files.*

I implemented a simplified version of the Linux filesystem. For this implementation, there is fixed overhead (in terms of peak RAM requirements) for any connected disk. Removing iNode magic and replacing each magic byte with increasing orders of indirect pointers allows the potential size per file to grow to several terabytes. Only one bitmap sector is required for every 4095 data sectors (roughly a 0.02% allocation, one eighth the overhead of a linked list implementation)

iNodes – Information actually stored on disk (and what is stored in RAM)

A screenshot of a computer program

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.

*Figure 1. iNodes on Disk (left) and in RAM (right)*

iNodes are a data block which contains information about where a file’s data is located on disk. The majority of the parameters are self-explanatory. ‘DP’ stands for ‘Direct Pointers,’ which are bytes that immediately point to a data sector. Small files will only use Direct Pointers. Single Indirect Pointers (SIP) and Double Indirect Pointers (DIP) are increasing orders of indirect pointers. SIP points to a block of direct pointers. DIP points to a block of SIPs.

DP – 62KB max file size

SIP – 126KB max file size

DIP - 8.3MB max file size

Everything is a file

Files and directories are the same in this implementation, only differing in the structure of what is stored in their data nodes.

Files store whatever information they want on a byte level.

Directories store ‘directory entries’ which holds the filename and a pointer to the sector.

Directories also store entries for ‘.’ and ‘..’ by default. There is no error checking for removing these entries, thereby allowing a user to disconnect their directory from its parent potentially leading to a hard fault and corruption. If a user knows enough to intentionally remove these entries, I believe that it is their fault and undefined behavior is just punishment for their actions.

Globally allocated buffers and mutexs

Since we don’t have a heap implemented for this lab, all of the data manipulation was done through a fixed number of globally allocated buffers, each with their own mutex.

There exist 5 buffers of block size, 1 pool for iNodes in memory.

The buffers are for

* Zeros
* Databuffer 1
* Databuffer 2
* Pathbuffer (for interpreter commands with long paths)
* Bitmapbuffer

This totals 5KB of RAM overhead for maintaining the filesystem (for a maximum of 10 simultaneously open files). This number decreases dramatically when a heap is implemented as the only buffer worth storing in RAM permanently is the bitmap, and all of the other buffers are only needed during file operation which can be allocated and freed on a short timescale.

Issues with stack overflow

It is worth noting that due to the limited stack size allocated to all threads (static 128 for this implementation) any thread which attempts to push too many things onto the stack will corrupt other global variables. Most commonly this corrupted the periodic threads data structure, creating hard faults when the OS attempts to launch these threads. Hence a user must be careful with calling filesys functions while their stack is almost full. These issues will be mitigated once the heap is implemented, as the buffers for files (which are commonly created as local variables – though they only take 8 bytes) can be allocated and freed via the heap.

1. **MEASUREMENT DATA**

|  |  |  |
| --- | --- | --- |
| SPI\_Clock\_Rate | KBps Read (KBps) | KBps Write |
| *996.016 Hz* | *304.48* | *303.95* |

Table 1. Disk Profiling Data

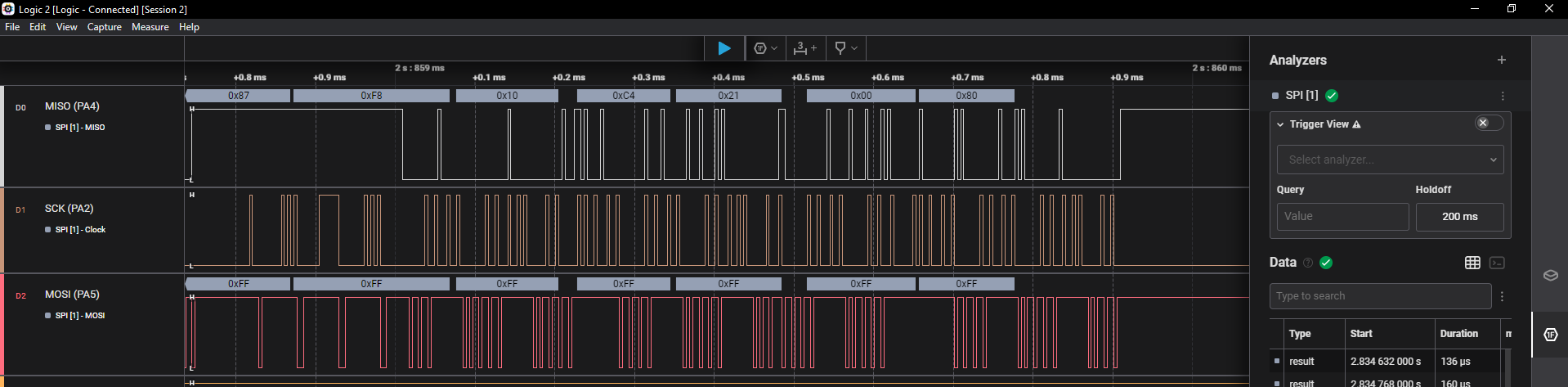


Figure 2. Example SPI Packets

1. **ANALYSIS AND DISCUSSION**
2. *Does your implementation have external fragmentation? Explain with a one sentence answer.*
   1. No. Sectors are allocated via a bitmap which simply finds the first allocable data sector and then pointers to these sectors are stored in iNodes. These sectors do not need to be continuous (though in practice they often are) hence there is no external fragmentation.
3. *If your disk has ten files, and the number of bytes in each file is a random number, what is the expected amount of wasted storage due to internal fragmentation? Explain with a one sentence answer.*
   1. BLOCK\_SIZE/2 = 256 bytes of internal fragmentation.
4. *Assume you replaced the flash memory in the SD card with a high-speed battery-backed RAM and kept all other hardware/software the same. What read/write bandwidth could you expect to achieve? Explain with a one sentence answer.*
   1. Assuming that the RAM card can operate at the bus clock frequency (80MHz) then we can transfer a maximum of 1 byte / 8 cycles (via sspi) hence 10MBps is an upper bound on the data rate. Realistically one half of this rate is probably realistic (5MBps).
5. *How many files can you store on your disk? Briefly explain how you could increase this number (do not do it, just explain how it could have been done).*
   1. 4095 free sectors / min 2 sectors per (nonempty) file = 2047 files.
   2. Increasing the number of sectors allocated to the bitmap (and modifying the root dir header sector) can increase this number to be limited by the disk size and not software.
6. *What is the maximum disk size and file size that your file system supports? Briefly explain what aspects in your file system limit maximum file and disk size.*
   1. Using 1 bitmap sector = 2048KB = 2MB partition. Allocating more sectors for the bitmap increases this as well to theoretically infinite disk size.
7. *Does your system allow for two threads to simultaneously stream debugging data onto one file? If yes, briefly explain how you handled the thread synchronization. If not, explain in detail how it could have been done. Do not do it, just give 4 or 5 sentences and some C code explaining how to handle the synchronization.*
   1. Truly simultaneous streams are impractical – it is a violation of the reader writer problem.
   2. My filesystem avoids this issue by allowing multiple threads to write to files concurrently, but uses semaphores to prevent simultaneous writes. Any two programs which do work, then stream data, then do more work and so forth will only lock the iNode for writing during the actual writing process and therefore their writes will be properly interleaved.