Section 2: Week 5: Algorithm Implementation

Nate Bachmeier

TIM-8110: Programming Languages and Algorithms

May 19, 2019

North Central University

# Algorithm Implementation

The operating system’s file system is responsible for persisting bytes streams to disk and retrieving it at a later point. While this in a simple concept its actual implementation is foreign to me. Online sources suggested that starting with FAT is ideal as it shares many design patterns of modern file systems, without all the bells and whistles. Afterwards a transparent compression feature was added to the implementation, based on Huffman Compression Algorithm.

## File Allocation Tables (FAT)

When a file is created it will span one or more physical blocks on the disk drive. There are several reasons that can prevent a file from being written across a single continuous block sequence. For instance, the physical disk buffer the disk queue might interweave heavy disk I/O from multiple parallel processes. The file could also be too large and require fragmentation to fit.

To address these challenges File Allocation Tables (FAT) can be used to build a map of where each page of the file is located. Each entry in the FAT contains a pointer to the next entry that continues it, along with various metadata such as permissions and time stamps. The reader can then reassembly the file by traversing the linked list structure until an End of File (EOF) marker is detected.

## Huffman Compression

An obvious advantage of compressed data is that it is smaller and requires fewer resources. This can improve the performance of network transmissions and increases the available space on a disk drive. One method for implementing compression is with the Huffman Compression algorithm. It was invented in the mid- 50s as a mechanism for variable bit encoding, where the most frequently used characters use the least number of bits.

The algorithm begins by counting the frequency of each value within an input byte array. These counts are then used to build a sorted binary tree with individual byte values as the leaves. When traversing the tree to the leaf nodes, each branch is either going left or right. Those decisions can be encoded as binary values. This binary path becomes the code word for compressing the value.

For instance, if a leaf node has the letter T and is arrived by Left, Left, Right, and Left; then it would have a code word of 1101. If the value T was persisted in ASCII that would be 0101 0100 – twice the bit count.

Finally, a new output byte array is created and populated with metadata for reconstructing the code book. Next the input byte array is encoded to the end of the output byte array. To decompress the metadata is read to reconstruct the codebook, then the compressed bits are decoded one-by-one back.

# Algorithm Efficiency

## FAT

To retrieve a file from FAT is very efficient taking time proportional to number of blocks containing the file. The system also leverages O(1) complexity for traversing the table. This is accomplished by creating all entries up front and then directly indexing into the entries array.

There is some overhead to finding the head of a file, especially in deep paths. The system must first request the root directory file, then parse the DirectoryEntry structures. These will contain pointers to child DirectoryEntry objects which are stored in separate directory files. This process must recurse until eventually finding the requested file head.

There can also be challenges with finding a free location to write the page. The worst case would occur when the disk is full, and all FAT Entries need to be checked. Typically, there are dirty pages or available locations. These help to keep the amortized cost down.

## Huffman Compression

One of the key challenges to the Huffman algorithm is that it needs to perform two passes through the input byte array. This is acceptable for most short messages but would not be useful across enormous files such as Virtual Hard Disk (VHD). The tree can have up to 2^8 leaves. This can lead to a lot of required comparisons as it decodes one character at a time. A previous example described encoding the letter ‘T’ as 1101, which means requires 4 decisions of left or right before reaching the leaf.

# Improving the Algorithms

## FAT

The FAT algorithm does not consider the physical movement of the disk controller arm. This can be expensive on mechanical disks as it needs to seek across the disk and reassembly the file. The Linux file system ext3 addressed this by adding a journaling feature which buffers huge memory segments to the disk in one shot. Then a background operation redistributes the journal contents to more optimal locations.

Another potential improvement is to redesign the Directory Entry files so that the recursive traversal is not needed. Perhaps the system can load them into memory during initialization. The loading algorithm could preemptively read directory files that are frequently accessed and likely to be queried.

The implementation for this course also included a Queue<int> dirty pages, which is appended by the DeleteFile operation. This improves the WriteFile as it can find available pages in the queue before it begins seeking for new pages.

## Huffman Compression

One of the limitations of the Huffman compression algorithm is that it only encodes a single value at a time. Instead it could focus on multi-byte sequences and further improve compression rates.

It would also be more efficient to only pass through the data once, like many stream compressors. This could be accomplished by determining the frequency weights ahead of time. For example, if the use case is predominately English text then ‘e’ will be more frequently used than ‘z’. The final compression might not be the perfect though it would allow for larger data sizes.

# Issues and Observations

Implementing the Huffman algorithm was a little challenging due to the multiple transformation steps required. This was addressed by using the dynamic programming approach and placing a method stub anywhere that was not defined. By breaking down the description into smaller and smaller concise problems it was possible to continue making progress.

Given more time it would have also been nice to redo the compression algorithm in a low-level language like C, instead of C#. This is because the algorithm makes heavy use of bits and pointer mathematics.

The Linux kernel includes a reference implementation of FAT which leverages libfs to expose a lightweight file system. In a later course that focuses on operating systems specifically, it could be a useful to revisit this area.

# Resources

Cantrell, G. (2014, October 30). *FAT File System Explained.* Retrieved from YouTube: https://www.youtube.com/watch?v=HjVktRd35G8

Georgia Tech. (2016). *File Allocation Table.* Retrieved from YouTube: https://www.youtube.com/watch?v=V2Gxqv3bJCk

Minnaard, W. (2014). The Linux FAT32 allocator and file creation order reconstruction. *Digital Investigation 11*, 224-233.

Sedgewick, R. (2014). *Algorithms, Fourth Edition.*