Operating Systems

ENGR-3950U-001

SneakyFS Design Brief

Created by

|  |  |  |
| --- | --- | --- |
| **Jonathan Gillett** | **100437638** | **jonathan.gillett@uoit.net** |
| **Joseph Heron** | **100425488** | **joseph.heron@uoit.net** |
| **Daniel Smullen** | **100440203** | **daniel.smullen@uoit.net** |

|  |  |  |
| --- | --- | --- |
|  |  |  |

Instructor: Dr. Kamran Sartipi

Document Revision 2.0

11/26/2012

Table of Contents

[Overview 3](#_Toc341723051)

[Key Features 3](#_Toc341723052)

[Unique Allocation Structure 3](#_Toc341723053)

[Journaling Using Copy-On-Write 3](#_Toc341723054)

[Transparent File System Encryption 5](#_Toc341723055)

[Object Oriented Design Principles Implemented Using C 5](#_Toc341723056)

[Universal Unique Identifiers 6](#_Toc341723057)

[Error Handling 7](#_Toc341723058)

[Unit Testing Framework 7](#_Toc341723059)

[Performance Metrics 7](#_Toc341723060)

[Read Complexity 7](#_Toc341723061)

[Write Complexity 7](#_Toc341723062)

[Memory Overhead 8](#_Toc341723063)

[Design Statistics 8](#_Toc341723064)

[Maximum Disk Size 8](#_Toc341723065)

[Maximum File Size 9](#_Toc341723066)

[Block Allocation Methodology 10](#_Toc341723067)

[Hybrid Linked/Indexed Model 10](#_Toc341723068)

[Overview of Main File System Functions 11](#_Toc341723069)

[Figure 1 - Copy-on-write Journaling 4](#_Toc341723070)

[Figure 2 - Using Structs as Objects 5](file:///C:\Users\100440203\Documents\Operating%20Systems%20-%2040863.201209-201209.40863-ENGR-3950U-001\sfs%20report\sneakyFS%20design%20brief.docx#_Toc341723071)

[Figure 3 - Structure of Blocks on Disk 10](#_Toc341723072)

[Figure 4 - Hybrid Block Allocation Scheme 11](#_Toc341723073)

# Overview

SneakyFS is a UNIX file system which operates on a simulated hard disk. It uses a hybrid indexed and linked allocation method to efficiently use disk space, support large disks, lengthy directory trees, and large file sizes. It incorporates various innovations such as transparent encryption, journaling, and universal unique identifiers. In terms of low-level design and implementation, it incorporates various advanced programming techniques which extend the functionality of the C language beyond its original scope of usage – this includes the implementation of object oriented design principles by repurposing C data structures and pointers.

## Key Features

The following section details key features which SneakyFS implements. These features set the file system apart from traditional file system implementations such as NTFS, ext2, and FAT32 by incorporating new innovations.

### Unique Allocation Structure

Using index block data structures which link data block locations to inodes on disk, files of a virtually unlimited length can be indexed under one contiguous data structure. As each index block becomes full, it links to an additional index block which extends the addressable storage space of each file. The first index in the structure is linked to by the file’s inode. Because each of these data structures is indexed, the actual blocks on disk can be anywhere. Since the index data structure can link to an infinite length of additional index blocks, data blocks can be continually indexed until the entire disk is full. As a result, the maximum supported file size spans nearly the entire size of the disk, minus the overhead requirement of one block for an inode, and the linked index blocks. For a detailed description of the allocation structure, see Block Allocation Methodology.

### Journaling Using Copy-On-Write

Journaling ensures that new data that is being written to disk (or changes to the file system) do not disrupt the integrity of existing files on disk, and linkages to the newly modified or created data do not occur until the moment that the new data has been verified as written correctly. This way, if any failure occurs during the write or modification process, the data is immediately restored to the previous state. This journaling is also incorporated into the data structure which tracks the free blocks available on disk. If a failure to write or modify a file occurs, every system attribute will be successfully reset to the state prior to the attempted operation.



Figure 1 - Copy-on-write Journaling

### Transparent File System Encryption

SneakyFS automatically encrypts all blocks of data being written to disk. When reading data, the blocks are decrypted using an encryption key specified by the user. In order to determine whether the correct key was used, a checksum operation is performed on the data, and matched with the checksum stored in the file’s inode. If the checksum matches, the file is readable. If the checksum does not match, the file was not decrypted correctly – either the key provided was invalid or the data is corrupt.

### Object Oriented Design Principles Implemented Using C

Under normal circumstances, the C programming language does not support object oriented programming – that is to say, there is no definition for classes, object attributes, or methods. These entities are logical constructs, however, so in order to increase encapsulation and coherency within SneakyFS source code, these concepts were captured using clever manipulation and usage of supported C data structures.

#### Using Structs as Objects for Encapsulation of Data

In C, structs define an encapsulation of variables inside a single data structure. Arrays are not necessarily separate objects by themselves; rather they are contiguously allocated chunks of memory which are accessed via pointers. Pointer arithmetic is used to traverse the array members using their known size in memory. C cannot return array types from functions, but pointers can be returned. Therefore, by cleverly allocating pointers within data structures, memory can be allocated and accessed using pointers and functions can use these pointers to access array members. Further, by encapsulating these complex pointer and array structures within a struct, an entire object can be created. By declaring these instances static, a primitive form of information hiding appears. Using these techniques in combination, pseudo-objects can be passed back and forth between functions within a limited scope.

Figure 2 - Using Structs as Objects



This technique was used in the design of SneakyFS with respect to the free block list data structure, the inode data structure, the system wide open file table, and the super block data structure in the source code.

In all of the data structures within SneakyFS, arrays of all data types which use dynamic memory allocation (or function as pseudo-objects) are pedantically terminated. This is done in order to guarantee that the length of all data structures can always be determined, which prevents segmentation fault errors in functions that traverse buffers of unspecified lengths.

#### Function Pointers and Visibility in C

In C, classes do not exist, and there is no equivalent entity. Therefore, it is impossible to create interface objects or use information hiding to make operations private within the scope of a single module. However, by using clever manipulation of pointers, the address of a function declaration on the execution stack can be determined and this address can be pointed to in order to move the execution to a different point in the program, and the visibility of these pointers can be controlled within a single scope. Therefore, information hiding and privatization of functions can occur on a logical basis within the code.

This technique was used within the unit testing suite in the source code.

#### Generic Programming

Generic programming uses a type system which is statically typed – that is to say, the type of every variable must be assigned. In SneakyFS, several critical internal functions use void pointers in order to facilitate any data type. This is used in operations such as concatenation of two variables with the same but unspecified type.

This technique was used as part of the library of global internal functions within.

### Universal Unique Identifiers

SneakyFS uses an external library for generating and validating universal unique identifiers (UUIDs). These are used within inode pseudo-objects in order to distinguish them by some means other than their name, or temporal attributes such as date created. Since a near-infinite number of UUIDs can be created, each file can be tagged with a UUID in order to ensure that the file can be identified uniquely. This is also used within the super block in order to give a logical volume a unique identifier as well. One important attribute of UUIDs are that they must follow a strict form guideline which minimizes the likelihood that a corrupted block of data would be read in as a unique identifier. This is used as a check to determine whether a disk has been initialized with SneakyFS. If the structure of the block in the default superblock location does not have a valid UUID, the disk does not pass the validation tests for mounting and should be reinitialized.

This technique was used within the super block data structure, and the inode data structures.

### Error Handling

All functions within SneakyFS have documented return values, and prior to any function return call errors are generated. The file system incorporates an error handling framework that ensures that any fatal internal error crashes the program, preventing damage or corruption to the file system or its entities. Non-fatal errors and ‘success’ error values are also handled inside the program, outputting pertinent details regarding the conditions of the error to the console.

The definitions for errors and their messages are defined inside, and further information on the error definitions is available within the user manual.

### Unit Testing Framework

In order to test the underlying functionality of high level functions, a unit testing framework was created for SneakyFS that was built upon as the functionality grew. Using this approach, each fundamental operation or helper function for an operation could be tested sequentially, and validated as correct. As each function passes validation, combining the functions in higher-level entities and functions ensured that the data on disk and in memory was manipulated correctly for all use cases.

The unit testing framework has a specially defined flag in which allows users to enter testing mode. The individual test cases within the framework are aggregated into, with example templates that are provided for various unit tests in order for developers to extend and implement their own unit tests.

## Performance Metrics

SneakyFS was designed from the ground up for use with the latest generation of storage technologies, particularly solid state devices. For these devices, disk fragmentation is not an issue and this was a specific design consideration when implementing the processes for writing files to disk, as well as the journaling scheme. SneakyFS is designed to be as space efficient as possible, and mitigate vulnerability to data corruption.

The main performance metrics are as follows:

### Read Complexity

All read operations within the disk occur in – that is to say, linear time complexity per operation. Reading a block on disk is an operation which occurs in constant time. Reading an inode and interpreting the contents also occurs in constant time. The linkage to the index structure and traversal of the index structure invokes a linear chain of operations up until the length of the index. The resultant complexity in the worst case is purely linear, based on the length of the file in data blocks.

### Write Complexity

Writing a new file is an operation which occurs in constant time, however, rewriting and rebuilding the index blocks for a file occurs in – linear time complexity. Traversing an index and generating or rebuilding an index is functionally the same operation. The way in which they differ is their input. Rebuilding an index linearly creates an index block data structure given the location of the data blocks on disk. Generating an index linearly creates an index block data structure and returns the data blocks on disk. This utilizes the traversal of the index block, which is also used in read operations. All of these occur in linear time complexity.

### Memory Overhead

Disk operations in SneakyFS all require movement of data into and out of buffers in order to manipulate the data before it is written to disk, or displayed to the console. No additional memory overhead is required for manipulation of data blocks besides one additional data block in memory at any given instant in order to manipulate the previous block, or to act as a null terminator. By default, data blocks are defined as 128 bytes. Thus, the memory overhead in the worst case is defined as double the amount of blocks being manipulated plus one additional block. When manipulating files, this means memory overhead of in the worst case. For very small files (in the order of one or two blocks), this will generally give an overhead of roughly 384 to 512 bytes in the default case – this is very minimal.

## Design Statistics

### Maximum Disk Size

Maximum disk size for SneakyFS is calculated using the largest addressable block index on disk. Since the file system uses 32-bit unsigned integers for addressing blocks, the theoretical maximum disk size with the default block size of 128 bytes is calculated as follows:

For larger block sizes, the theoretical maximum disk size increases accordingly. Blocks also use 32-bit unsigned integers for addressing individual bytes in a block. Therefore, the theoretical maximum block size is calculated as follows:

Therefore, the resultant absolute theoretical maximum disk size is:

### Maximum File Size

Each file requires an overhead requirement of 1 block reserved for its inode, as well as an index block data structure in order to index all of the data blocks which encompass a file. The number of index blocks required to address a file’s data blocks is given as follows:

#### Minimum Overhead on Disk

The disk itself has a requirement that the first block be designated the super block. There must also be a free block list data structure written to disk which requires the following number of blocks:

Using the default file system values of 512 and 128 for the number of blocks and block size, respectively, this yields a minimum overhead on disk of:

#### Overhead of Maximum File

Given the amount of index blocks required, and the additional overhead requirement, using the example of the default file system values of 512 and 128 for the number of blocks and block size respectively, this yields a maximum file size of:

# Block Allocation Methodology

Different file systems use different methodologies for allocating blocks on disk. Traditional implementations use contiguous allocation, linked allocation, or indexed. SneakyFS uses a combination of the best attributes of the latter two, creating a new hybrid methodology that allows for large file sizes, and for quick traversal of directory structures. This methodology reuses data structures for file data indexes and directory indexes.

## Hybrid Linked/Indexed Model

The linked allocation model stores one attribute within an inode data structure to link the inode to a data block. A finite number of linkages can exist within one inode in order to link a data block to the next, with each data block linking to the next block in the chain. The indexed model uses a designated block which stores a sequenced list of indices that represent locations on disk - traversing these indices in sequence yields the data contents of a file. The hybrid model developed for SneakyFS uses designated index blocks which are linked together, allowing for a near-infinite length index structure, which can link to a near-infinite number of indices.



Figure 3 - Structure of Blocks on Disk



Figure 4 - Hybrid Block Allocation Scheme

# Overview of Main File System Functions

Please refer to the reference manual for detailed descriptions of the functionality of all underlying processes within SneakyFS. Detailed call graphs, and implementation specifics are provided.