The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2013

COMP3301/COMP7308
Assignment 3 — File systems

Version 2.0 – Full specification

Due: 8pm Friday 25th October 2013
Weighting: 30% (100 marks total)

NOTE THAT YOU MUST GET AT LEAST 50% FOR THIS ASSIGNMENT TO PASS THE COURSE.

Introduction

The goal of this assignment is to give you practical experience modifying an existing file system for Linux.

You will need to complete the file systems practical before attempting this assignment, as it instructs you how to clone the *ext2* file system that you will be using for the assignment.

You will complete this assignment using the UML Linux kernel available on moss.labs.eait.uq.edu.au.

This assignment must be completed individually. You should read and understand the school's policy on plagiarism and student misconduct, which is available in the course profile.

Overview

The assignment is split into two distinct parts. You are required to implement both parts in the same file system driver, and they must interact as described below.

When loaded, your file system driver should register itself with type *ext3301*.

It may be useful to mount any file systems you create with the debug mount option while testing. This will cause the module to output extensive information to the kernel's ring buffer describing what it is doing. You may add new calls to write debugging information out (using the existing ext2 debug calls), but please remove any calls to printk() before submitting your assignment (if you added any).

Part A — Immediate files

In the unmodified ext2 file system, regular files are stored in data blocks on a block device (often a hard disk). Pointers to these data blocks live in the inode structure as single, double and triple indirect pointers. These pointers take up 60 bytes of space in the inode, but some (if not all) are often unused if the file is small. For more information on the inode structure, see *the Documentation/filesystems/ext2.txt* file inside the kernel source tree.

Immediate files are a way of storing the contents of a small file directly in the inode structure, in the unused block pointers, rather than in data blocks. Files up to 60 bytes can have their contents stored in these pointers, and no data blocks need to be allocated. Files over 60 bytes cannot fit in the inode, so they need to be stored in allocated blocks, and the block pointers need to be used to point to these data blocks (just like in the existing ext2 file system).

Your task for this part of the assignment is to implement immediate files in the *ext3301* file system. New files should be created as immediate files (which have the file type value as specified below), and continue to be immediate files until their contents can no longer fit in the inode itself (by growing to over 60 bytes). When this happens, the file should be transformed into a regular file and its contents be transferred to data blocks (you will need to allocate these before transferring). You need to ensure that when this happens the block pointers are updated to be valid, so further references to the file succeed in accessing the data blocks.

When a file is truncated below 60 bytes, you must convert the file back into an immediate file by transferring the contents of its data blocks into the inode structure. Ensure you free any data blocks used.

You do not have to implement immediate files for special file types (e.g. block device) — you will not be tested on this.

You cannot modify any code outside of the kernel, and since file types are declared in the system header file linux/fs.h (search for DT_REG), you must define a new immediate file type inside your module. Pick a new file type number, and define the type in the form of:

```
#define DT_IM   X
```

(where X is a unique integer. It is not as simple as picking an unused number; you will need to do some testing to see what works.)

Ensure that DT_IM is defined as specified in one of your source files so the marker can recognise an immediate file during marking.

Note: since we will be modifying parts of how the inode structure is represented, the e2fsck tool will probably fail or detect errors when there aren't any. This is expected behaviour and you do not need to worry about fixing this.

Your file system implementation must still be able to mount existing ext2 file systems created with the mkfs.ext2 tool, so make sure you do not modify the inode structure itself (do not add or remove any fields, or change the size of the inode).

Tips

You can cast the member of the inode structure that stores block pointers to an unsigned char * and use it as a contiguous piece of memory for storing immediate files in. This way you do not have to access the pointer fields directly.

When a file is marked as immediate, you need to ensure no part of the file system attempts to access the block pointers (since they will effectively contain garbage).

Part B — File encryption

The *ext3301* file system must support a very naive encryption scheme. Any files under the /encrypt directory (if it exists) of a given *ext3301* file system need to be encrypted. This will either occur on disk (if the file is a regular file), or in the inode structure if the file is marked as immediate. Files outside /encrypt are to remain as plaintext.

Your file system driver must support an extra mount option called key. This specifies the encryption key that will be used during encryption and decryption for the given mount of the file system. The key will be given in hex format, and you can use sscanf() to parse the option and extract the key. You should only use the 8 least significant bits of the key (meaning it can only be in the range 0x0 to 0xFF). If no key option is specified at mount time, then the encryption key defaults to 0x0 (which disables encryption) and data should be passed through the file system as-is.

It is valid to mount a file system with one key, write some data to it, and then remount it with a different key (even though this would result in garbage when reading).

Moving a file into the /encrypt directory should trigger its contents to be encrypted, and moving a file outside of this directory should trigger its contents to be decrypted (with whatever key the file system is mounted with). In Linux, moving a file within the same file system is implemented using a link and then an unlink on the inode — the data blocks are not moved.

You do not have to handle hard links or symbolic links in this file system.

The encryption algorithm is a simple substitution cipher, where each byte is XORed with the key:

Encryption: $Ci = Pi \oplus k$

Decryption: $Pi = Ci \oplus k$

(where *k* is the encryption key given when mounting the file system.)

You should ensure the key does not get written to disk — it must stay in memory at all times (so do not store it in the inode).

All normal files in the directory tree under /encrypt should be encrypted.
You can choose to encrypt directory files or not – whichever you find simpler.

Interactions between immediate files and encryption

Your file system needs to support the different combinations of both immediate files and file encryption. For instance it is perfectly valid to have an immediate file that is encrypted (and the encrypted data needs to be stored in the inode as described above).

Code compilation

Your implementation must compile as a Linux kernel module (with a .ko extension). It must compile and be loadable on the UML kernel. See the kernel module practical for information on Makefiles for kernel modules.

Your code will be marked by downloading your 'a3" repository directory on subversion.

Your module will be built by running the following in the a3 repository directory:

```
make
```

Make only needs to generate the .ko file, all the rest (loading kernel module, mounting filesystem, etc.) will be done by the tester during testing.

Coding style

There is not a specified coding style for the assignment, however please note the following minimum expectations:

- The name, author, creation date should be included as a file header to any source files.
- It is bad practice to have arbitrary constants (such as 60) hard-coded. They should be defined as constants.
- You should have enough comments so that the purpose of blocks of code is easily understandable.

You might like to consider using the Linux kernel coding style document available at http://www.kernel.org/doc/Documentation/CodingStyle, but this is not essential.

Submission and Version Control

The due date for this assignment is 8pm Friday 25th October 2013. Submission made after this time will incur a 10% penalty per day late   Any submissions more than 4 days late will receive 0 marks. Of course extensions may be given for medical or other genuine reasons.

This assignment must be submitted through EAIT's Subversion system. The repository URL

for this assignment is (where s4xxxxxx is your student number):

https://source.eait.uq.edu.au/svn/comp3301-s4xxxxxx/a3

or

https://source.eait.uq.edu.au/svn/comp7308-s4xxxxxx/a3

Submissions will be retrieved from your repository when the final cutoff date has been reached. Your submission time will be taken as the most recent revision in the above repository directory.

Details on subversion repositories in EAIT Faculty are at:

http://student.eait.uq.edu.au/software/subversion/

Marking Scheme

10% - for Coding style, readability, organisation

50% - immediate files

Approximate grading scale:-

50/50 – everything works correctly
40/50 – mostly works but errors prevent fully correct operation
30/50 – only partially works, significant errors
10/50 – relevant code provided, and code compiles, but doesn't work

40% - encrypted files

40/40 – everything works correctly
30/40 – mostly works but errors prevent fully correct operation
20/40 – only partially works, significant errors
10/40 – relevant code provided, and code compiles, but doesn't work

Revision History:
Version 1: Draft Spec
Version 2: 30/9: Some URL errors corrected, minor clarifications of marking scheme, etc.