The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2012

COMP3301/COMP7308
Assignment 3 — File systems

Due: 8pm Friday 26th October 2012
Weighting: 25% for COMP3301 students (100 marks total)

Version 1.0 — 1 October 2012

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 *ext3301* file system that you will be using for the assignment.

You will complete this assignment on the virtual machine image provided on the course website. More information on this image is available under the Resources page.

This assignment may be completed **individually** or in **self-selected groups of two**. Please read the section on group work if you decide to work in groups of two. Although you may share code with your team member (if working in groups of two), it is still considered cheating to look at another student's code or allow your code to be seen or copied by anyone but your partner. You should be aware that all code submitted may be subject to automated checks by plagiarism-detection software. You should read and understand the school's policy on 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. It will be automatically extraced so you may be penalised if the automated script cannot find it.

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 against the key:

Encryption: $C_i = P_i \oplus k$

Decryption: $P_i = C_i \oplus k$

(where k is the encryption key given when mounting the file system.) Note that this is a different encryption algorithm from assignment 2.

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).

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).

Short-response questions

1. Discuss what happens in your implementation if you attempt to create a symbolic link from:
   (a) outside the encrypted directory to a file inside the directory (for instance `/foo` → `/encrypt/foobar`)
   (b) inside the encrypted directory to a file outside it (for instance `/encrypt/foobar` → `/foo`)

   Explain why each of these either works or does not work, and explain why.

2. Discuss what happens in your implementation if you attempt to create a hard link from:
   (a) outside the encrypted directory to a file inside the directory (for instance `/foo` → `/encrypt/foobar`)
   (b) inside the encrypted directory to a file outside it (for instance `/encrypt/foobar` → `/foo`)
   (c) another file system to a file inside the encrypted directory

   Explain why each of these either works or does not work, and explain why.

3. What would happen if you run `e2fsck` over an *ext3301* file system that:
   (a) contains only regular files
   (b) contains only encrypted files
   (c) contains only immediate files

   Discuss the behaviour you see and suggest why this happens.

Code compilation

Your implementation must compile as a Linux kernel module (with a `.ko` extension). It must compile and be loadable on the version of the virtual image provided on the website. See the kernel module practical for information on Makefiles for kernel modules.

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

```
make
```

Coding style

Your solution must conform to the Linux kernel coding style document available at `http://www.kernel.org/doc/Documentation/CodingStyle` (or `Documentation/CodingStyle` in the kernel source tree). You may wish to run your code through the `checkpatch` tool to validate that your solution adheres to the coding style. The following arguments may be useful to you (where `FILE` is the C source file you wish to check):

```
checkpatch --no-tree --no-signoff -f --no-summary FILE
```

The tool is available to download from the Resources page of the course website. If you download from another source, make sure you use version 0.32 as this is the version we will use when marking.

You may also find the `indent(1)` tool useful with the following arguments:

```
indent -kr -i8 FILE
```

Please note that these are tools only — they should not be considered perfect. Always double-check the results by hand to be sure.

Group work

If you decide to work in groups of two, **one** member of the group should be chosen to host the repository and that student should e-mail one of the tutors (or if unavailable the lecturer) **no later than seven days before the due date**. After this cutoff it will be assumed that you are working individually and you will be marked as such. Both members of a group will receive the same mark, and any complaints or problems should be directed to the lecturer who will treat each case confidentially.

The tutors and lecturer's e-mail addresses can be found on the course website.

Submission and Version Control

The due date for this assignment is **8pm Friday 26th October 2012**. Submission made after this time will incur a 10% penalty per day late (weekends are counted as 1 day). Any submissions more than 4 days late will receive 0 marks. No extensions will be given without supporting documentation (i.e. medical certificate or family emergency)—should such a situation occur you should e-mail the course coordinator as soon as possible.

This assignment must be submitted through ITEE's Subversion system. The repository URL for this assignment is (where *s4123456* is your student number):

`https://svn.itee.uq.edu.au/repo/comp3301-s4123456/a3`

If you are working in groups of two, only one student should use their repository. When submitting group membership to the tutor, you should nominate the student that will host the repository for the group. Permissions will be then set accordingly so you can use your own account. This means both students will checkout and commit to the same repository.

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.

You are required to make regular commits to your repository as a demonstration of your work. Subversion history will be considered in marking. Do not submit your assignment with a single, large commit. You will be penalised for doing so.

For information regarding ITEE's Subversion system, see `http://student.eait.uq.edu.au/software/subversion/`.

Answers to the short-response questions should be provided in a `responses.txt` file in the specified repository directory.

Assessment Criteria

Grade band	Immediate files (40 marks)		File encryption (30 marks)		Short-response answers (15 marks)		Coding style and comments (10 marks)		Version control (5 marks)	
Excellent	Correct implementation of immediate files with no errors. Clear understanding of file systems and assignment concepts.	40	Correct implementation of file encryption with no errors. Encryption/decryption is handled correctly with no errors.	30	Clear explanations and descriptions of answers given. No incorrect or misleading explanations. Student clearly grasps the assignment concepts.	15	Coding style applied consistently and without any error. Code has meaningful comments where appropriate, with no complex sections left uncommented.	10	Evidence of continual progress through version control history.	5
Very good	Correct implementation of immediate files with very few to no errors. Clear understanding of file systems and assignment concepts, with only minor issues.	36 32	Correct implementation of file encryption with very few to no errors. Encryption/decryption is handled correctly with only minor issues.	28 24	Clear explanations and descriptions of answers given. Very few incorrect or misleading explanations. Student clearly grasps the assignment concepts.	13 11	Coding style applied consistently with few errors. Code has meaningful comments where appropriate, with some complex sections left uncommented.	9 8	Evidence of good progress through version control history.	4
Good	Correct implementation of immediate files with few errors. Good understanding of file systems and assignment concepts, with some issues present.	28 24	Correct implementation of file encryption with few errors. Encryption/decryption is handled with few errors.	22 18	Explanations and descriptions of answers given. Some incorrect or misleading answers. Student has a fair grasp of the assignment concepts.	10 9	Coding style applied with some consistency and with some errors. Code has a fair amount of meaningful comments where appropriate, with some complex sections left uncommented.	7 6	Evidence of some progress through version control history.	3
Satisfactory	Partial implementation of immediate files with some errors. Basic understanding of file systems and assignment concepts.	20	Implementation of file encryption is incomplete with some errors. Encryption/decryption has some errors.	15	Some answers given with some incorrect or misleading answers. Basic explanations given. Student has a basic understanding of assignment concepts.	8	Fair attempt to apply coding style, but some errors or not much consistency. A basic attempt to place meaningful comments throughout code.	5	Little evidence of progress through version control history.	2
Poor	Basic attempt to implement immediate files but has significant errors. Little evidence of an understanding of file systems and assignment concepts.	16 12 8 4	Basic attempt to implement file encryption with significant errors. Encryption/decryption has major errors and/or is mostly incomplete.	13 10 7 4	Very few answers given. Many incorrect or misleading answers. Student has a poor grasp of assignment concepts.	6 4 2	Basic attempt to apply coding style, with many errors or no consistency. Very few meaningful comments in code.	4 3 2 1	Very little to no evidence of progress through version control history.	1
Very poor	No attempt made to implement immediate files.	0	No attempt made to implement file encryption or encryption/decryption.	0	No attempt made at short-response questions.	0	No attempt made to apply coding style standards or comment code.	0	No evidence of progress through version control history.	0

Penalty:	**Comments:**
Total mark: **/100**	