3005: Coursework 2

devised by Mark Handley
M.Handley@cs.ucl.ac.uk

DEADLINE

The deadline for this coursework will be posted on the Moodle course web page.

PLAGIARISM

Plagiarism is when you present someone else's work as your own. While you are encouraged to discuss this assignment with your colleagues, the work that you submit must be your own.

Plagiarism is cheating!

Your attention is drawn to the statement on plagiarism at:

http://www.cs.ucl.ac.uk/students/teaching_matters/#c10837

If evidence of plagiarism is found, disciplinary action will be taken.

A FILESYSTEM IN A FILE

This goal of this coursework is to give you hands-on experience of the issues that need to be handled within a filesystem, without all the complexity of a real filesystem. To do this we will use a filesystem in a file.

A filesystem in a file actually has practical uses too. For example, MacOS X supports the concept of a file vault, which is an encrypted file system in a file, keeping the contents secure. The MacOS file vault is mounted just like a normal filesystem, so all the normal commands and GUI work for copying files in and out.

Unfortunately it would make our implementation too complex to mount it as a normal filesystem, so we'll have to use special commands for copying files in and out of the filesystem, creating directories, listing files, etc.

Our virtual filesystem is going to be loosely based on the Linux Ext3 filesystem, which is fairly typical of a Unix filesystem, but with extensions for journalling. We will however simplify many aspects, because we simply do not need all the attributes Ext3 provides.

The user interface should consist of the following command programs:

`jfs_format <volumename> <size>`

This command creates a new filesystem in the file called `volumename` with the specified `size`.

`jfs_fsck <volumename>`

This command checks the filesystem in the file called `volumename` and reports any errors or inconsistencies it finds. It does not however repair those inconsistencies.

`jfs_ls <volumename>`

This command lists all the files in the filesystem called `volumename`, similar to "`ls -lR`" on a Unix system.

`jfs_mkdir <volumename> <dirname>`

This command creates a new directory called `dirname` in the filesystem called `volumename`.

`jfs_copyin <volumename> <fromfile> <tofile>`

This command copies the file `fromfile` from the regular filesystem into our filesystem in a file, storing it in `tofile`. The filename `tofile` should be a full pathname within the filesystem in a file.

`jfs_copyout <volumename> <fromfile> <tofile>`

This command copies the file `fromfile` from our filesystem in a file to the regular filesystem, storing it in `tofile`. The filename `fromfile` should be a full pathname within the filesystem in a file.

`jfs_rm <volumename> <filename>`

This command deletes the file `filename` from the filesystem in a file, freeing up space and the i-node for new files.

`jfs_checklog <volumename>`

This command checks the journal logfile for the filesystem in a file, applying any unwritten changes to make the filesystem consistent. See below for more details.

You will be provided with the source code for the memory mapping of the filesystem's file, together with disk block emulation, and many routines that are common across all the commands. You will also be provided with the full C source code for all the commands *except* `jfs_rm` and `jfs_checklog`.

This source code will be available from the following URL:
`http://nrg.cs.ucl.ac.uk/mjh/3005/cw2/`

THE QUESTION

Part 1

For part 1, you must implement `jfs_rm`. The program must successfully remove a file from any directory (including the root directory) on the virtual filesystem, leaving the filesystem in a consistent state. It should work, irrespective of whether the file was the first file in that directory, the last file, or somewhere in between.

Part 2

The filesystem implements a very simple journalling mechanism. There is a logfile in the root directory called ".log". All writes are buffered in memory until a `jfs_commit()` is called. The writes are then written to the logfile, followed by a *commit block*. The commit block contains a magic number (so you know it's a commit block not a regular block), a list of where the blocks in the logfile should be on disk, and a checksum. After the commit block has been written, the writes are written to their final destination on disk. Finally the commit block is cancelled by overwriting it, clearing a flag to indicate that the logfile entries are no longer needed.

Your task is to write a program, `jfs_checklog`, that reads the logfile and fixes the disk as appropriate.

Of course the files should not get inconsistent unless the system goes down part way through writing changes. To facilitate testing, you can set the shell environment variable `CRASH_AFTER` to indicate the number of written blocks to wait before crashing. For example, if you use sh or bash as your shell type:

```
./jfs_format disk1
CRASH_AFTER=13; export CRASH_AFTER
./jfs_copyin disk1 README foo
```

This will format a new filesystem called disk1, set the code to crash after 13 writes, then attempt to copy the README file into the filesystem. The code will crash part way through, leaving the filesystem inconsistent.

What to hand in:

1. The C source code for `jfs_rm` and `jfs_checklog`. Source code must be properly commented, sensibly indented, and printed out in a fixed-width font.
2. For each of the commands you implement, provide up to half a page of concise description of how your solution performs its task.
3. If you have to modify any of the supplied code, hand in a printout of the functions that you changed, indicating clearly which parts you changed. If you do not need to modify the supplied code, you do not need to hand it in.

Zip the above files and submit the collection on Moodle

FILESYSTEM STRUCTURE

Although the filesystem consists of a large chunk of contiguously mapped memory, the utilities provided give a block abstraction, so you should only use `read_block()` and `write_block()` to modify the stored filesystem.

Thus the virtual disk consists of a sequence of numbered 512 byte blocks.

The filesystem code you will be given then divides these blocks up between *block groups*, each consisting of 256 blocks.

At the start of each block group is a superblock giving filesystem metadata, followed by a bitmap of free i-nodes and a bitmap of free blocks. The bitmaps and the supernode are all small enough that they fit together into the first block of the block group.

Following the superblock are a set of blocks reserved for i-nodes. Each i-node takes up 64 bytes, so 8 i-nodes fit in a block. We reserve 64 i-nodes per block group, so 8 blocks are used to hold i-nodes. The remainder of the blocks in a block group can be used to hold blocks of files or blocks of directories. Thus the structure of the disk looks like:

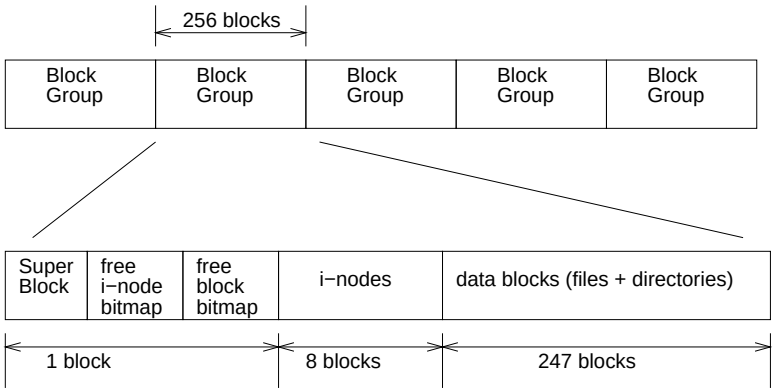

Figure 1: Basic Virtual Disk Layout

i-node numbers and block numbers are global across the virtual disk. Thus the first i-node in the first block-group would be i-node 0, and the first i-node in the second blockgroup would be i-node 64.

(Continued overleaf)

An i-node has the following structure:

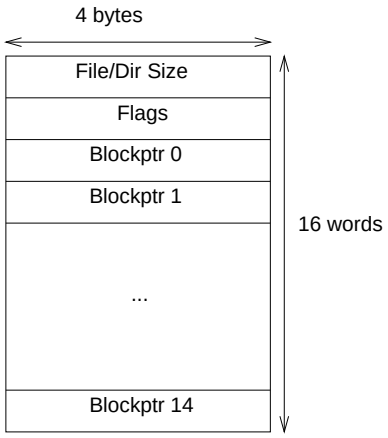

Figure 2: Structure of an inode

This is a rather simplified i-node, as it does not support indirect blocks. Thus the largest file this can store is 7168 bytes. For bonus marks, you may wish to extend this model to support indirect blocks.

A directory consists of a single block with multiple directory entries in it (in a real filesystem, a directory would be able to flow across multiple blocks, but currently it doesn't in this simplified version). A directory entry has the following structure:

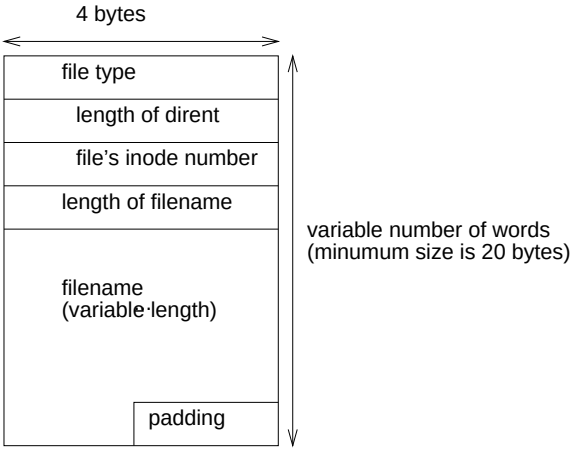

Figure 3: Structure of a single Directory Entry