Very Very Simple File System (VVSFS)

Jack Kilrain (u6940136) Daniel Herald (u7480080) Angus Atkinson (u7117106)

22 October 2023

Contents

1 Overview

This report discusses the implementation of several operations into VVSFS, the Very Very Simple File System. Our group completed all four baseline requirements (unlinking, renaming, inode stats, filesystem stats), the advanced task (indirect blocks), and two extensions (hard/soft links and block/character devices).

2 Baseline

2.1 Unlink Dentries and Removing Directories

2.1.1 Unlink

Unlinking a dentry from a directory involves not only finding and deleting the dentry, but also potentially deleting the data block containing that dentry. Therefore, our implementation splits `unlink` into two stages:

1. Finding the dentry

2. Removing the dentry

To support find and remove operations, we created a helper datatype (called `struct bufloc_t`) which stores block and dentry indices for ease of use. In addition, this structure can optionally retain the buffer object and dentry object (depending on the flags `BL_PERSIST_BUFFER` and `BL_PERSIST_DENTRY`), to avoid having to re-index the buffers later on a subsequent write. These buffers can also be resolved at a later time if necessary.

The method `vvsfs_find_dentry` is used to search through the directory's data blocks and populate a `bufloc_t` instance with the relevant information. Search operations merely involve a linear traversal of the blocks bound to the directory `inode`, attempting to match the name and parent inode of a dentry to the target one.

The `bufloc_t` structure is then forwarded to the `vvsfs_delete_entry` method to perform the actual removal. Removing a dentry requires care to be taken in cases where the dentry is the:

1. Last dentry in the last block
2. Non-last dentry in last block
3. Any position dentry in non-last block

In the first case, we can simply evict the dentry (by zeroing it) and we are done. If there are no dentries left in the block, we can also deallocate it by removing it from the `inode->i_data` array.
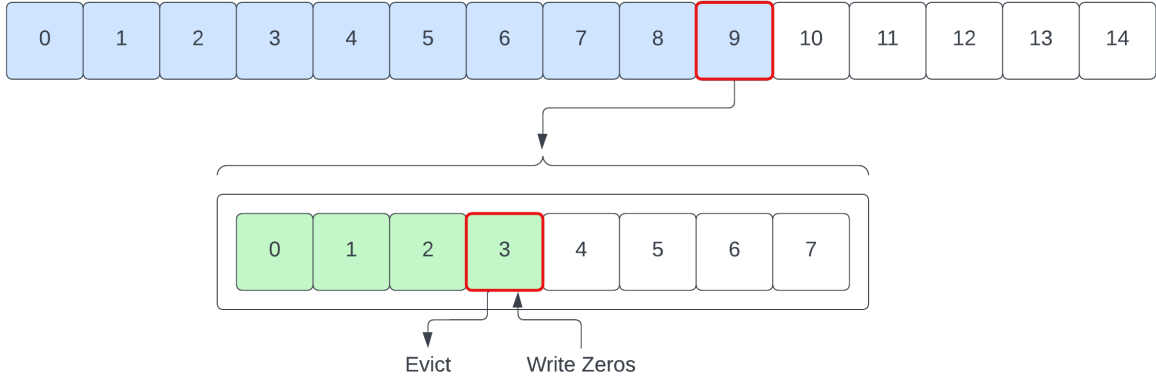

Figure 1: Last block last dentry

The second case requires moving the last dentry in the last block to fill the hole of the one being evicted. We do this with a memmove to overwrite the target dentry, then zero the old location of the last dentry in the last block.
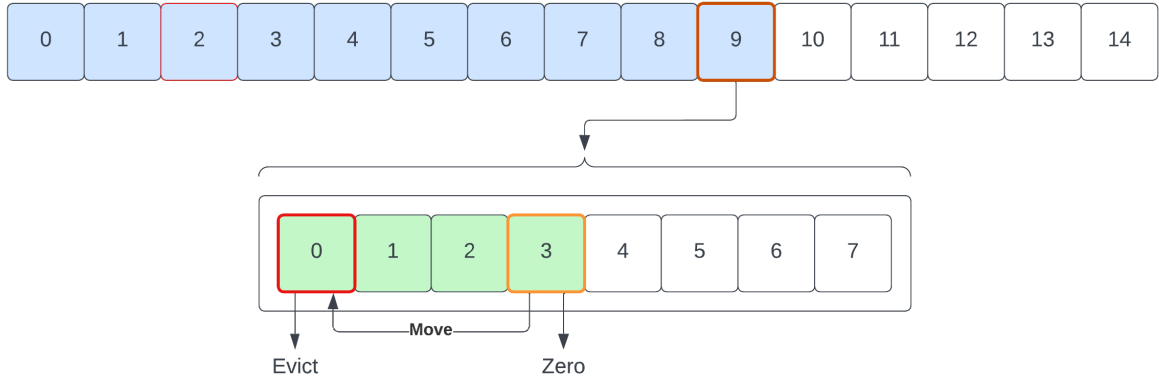
Figure 2: Last block non-last dentry

In the last case, we need to move the last dentry from the last block to our current block to fill the hole. This requires multiple buffer locations to be open at once, in order to move and then zero the old location of the last dentry in the last block.
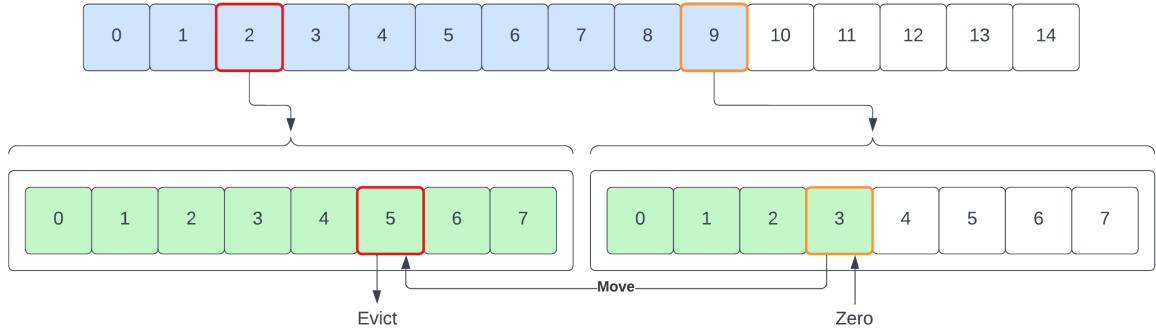

Figure 3: Non-last block non-last dentry

Once all shifting and operations have been performed with dentries and blocks, the inode matching the dentry is deallocated (using the bitmap) and returned to the available pool.

2.1.2 Directory Removal

In order to perform directory removal, we rely on the `vvsfs_unlink` operation discussed above. But before performing an arbitrary unlink on the dentry matching the directory, there are some checks needed. Specifically, we need to ensure that the directory is empty, which is done by traversing the dentries within the directory and verifying that it is empty (or only contains reserved entries). Reserved implies any of the following:

1. It is either a `.` or `..` entry (these are not persisted to disk, so are not a problem, but are checked regardless). In the `..` case, we also verify that the parent inode matches the inode stored in the dentry.
2. Reseved inode number, specifically `0` as the root.

If any of these checks fail, we return `-ENOTDIR` when not a directory and `-ENOTEMPTY` when it is not empty. Assuming these pass, we forward the dentry to the `vvsfs_unlink` call for removal.

2.2 Renaming

We implement rename operations for files and directories via the `vvsfs_rename` method. There are two main cases for the rename operation: those where the destination dentry exists (because there is already a file/folder with the same name in the destination folder), and those where the destination does not exist.

In the simple case, where the destination dentry is not pre-existing, we can simply add a new dentry to the destination folder (setting the inode number to be that of the file to rename), and deallocate the old dentry from the source folder. Since these operations of finding, adding and removing dentries from folders are the same as those needed for the `vvsfs_unlink` function, we reuse many of the helper functions described in the section above. This eliminates code duplication and reduces the chance of bugs, as the dentry addition/removal logic is now tested via a wider variety of usage patterns.

Additional complications can emerge when the destination dentry already exists. There are many rules which govern whether a rename operation is allowed to overwrite an existing file or directory (e.g. depending on whether the destination directoy is empty). We consulted the man page for `rename()` and Michael Kerrisk's book *"The Linux Programming Interface"* to gain a deep understanding of these requirements, and carefully implemented checks to prevent invalid renames. The process of actually renaming the file is relatively similar to beforehand, with the one exception being that it is also necessary to decrement the link count in the existing file's inode. If the destination dentry was they only hard link to the existing file, it's link count will now be zero, which causes the file to be deleted. This process is illustrated in Figure 4 below.
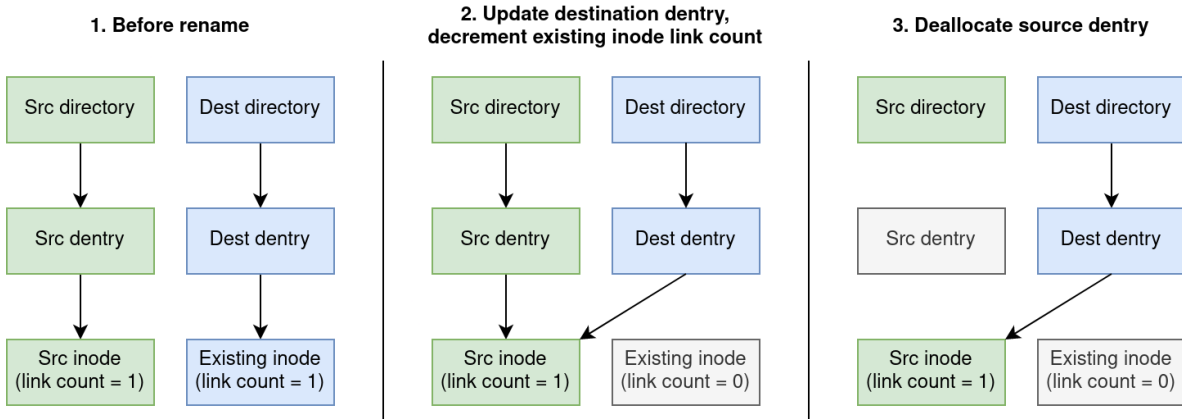
Figure 4: Renaming a file to overwrite an existing file at the destination

2.3 Inode Attributes

As part of the baseline tasks, we've added support for the following inode attributes:

- `GID`: group id
- `UID`: user id
- `atime`: access timestamp
- `ctime`: change timestamp
- `mtime`: modified timestamp

The `vvsfs_inode` structure was the only place we needed to update, adding in the necessary fields. The current inode size (`VVSFS_INODESIZE`) already provisions space for all potential fields, so no updates were needed for the code which handles moving inodes between memory/disk.

In order to load these fields, we updated the `vvsfs_iget` method to write the `atime/ctime/mtime` fields to the inode structure from the disk inode. Note that we also set the nanoseconds to 0 (`tv_nsec` field on `i_atime`) to be consistent with `ext2` and `minixfs`. Additionally, the `vvsfs_write_inode` method was updated to perform the inverse, writing the fields from the VFS inode to the disk inode. Finally, we did not implement `setattr/getattr` since there was nothing additional we desired beyond the generic VSF implementation.

During the initial development of the inode attributes, position dependent behaviour was not accounted for in the struct packing. This problem was encountered when we originally ordered our fields the same as Minix. This caused subtle bugs and data corruption. To combat this, we stored the new fields after the provided fields. This also ensures full binary compatibility with the default VVSFS implementation, with the only consequence being potentially weird statistical data.

During testing, we discovered that the Linux kernel has provisions to prevent disk thrashing when updating the `atime` field often. In order to override this behaviour and force the kernel to write through to disk, the `strictatime` parameter was included as a mounting option for our testing scripts.

2.4   Supporting FS Stats

The `struct kstatfs` layout details many fields that are normally dependent on the creation and mounting options for file systens. However, VVSFS is very simple, and consequently almost all of the fields are static values that can be assigned directly. Specifically, all of the following fields are statically defined in VVSFS:

- `f_blocks = VVSFS_MAXBLOCKS` (we always configure the FS with the same block count)
- `f_files = VVSFS_IMAP_SIZE * VVSFS_IMAP_INODES_PER_ENTRY` (the inode bitmap is a fixed size)
- `f_namelen = VVSFS_MAXNAME` (constant for any configuration of VVSFS)
- `f_type = VVSFS_MAGIC` (VVSFS-specific constant)
- `f_bsize = VVSFS_BLOCKSIZE` (static value that is not configurable on mount or creation)

The key fields are `f_bfree` (which is equal to `f_bavail` since we have no non-superuser scoped blocks) and `f_ffree` which are populated by traversing the bitmaps to count how many bits are set. This is done by looping over the bytes, and bitmasking and counting bits. It is also possible to do this more efficiently using compiler (e.g. GCC `__popcountdi2`) or hardware intrinsics (e.g. x86 `popcnt`) for counting the set bits in a fixed sized integer. However, these are not easy to implement within Kernel modules and don't add any significant speedups for an uncommon syscall.

3   Advanced

3.1   Indirect Blocks

There are three main areas that need to change to support indirect blocks: listing dentries, adding a new dentry and unlinking a dentry.

The first case of listing dentries is the most straightforward. Previously, we iterated over all the blocks in `i_data`; now, we traverse only the first 14 and load the dentries into memory. Then, we check if a 15th exists (indicating indirect blocks are present), which is then buffered. Next, we perform the same iteration over the block addresses within the indirect block (via offset using `sizeof(uint32_t) * <iteration count>`). The iteration count (inclusive) is determined by subtracting 15 from the total block count. We then traverse through the dentries within each block, iterating 8 times (no. dentries) per block until the last entry where we use the total dentry count modulo 8 as the limit. Note that due to our coding conventions, there will never be a case where the indirect block is allocated with zero block entries within it.

When adding a new dentry, we first calculate the new block position and dentry offset using the following:

```
uint32_t dentry_count = inode->i_size / VVSFS_DENTRYSIZE;
uint32_t new_dentry_block_pos = dentry_count / VVSFS_N_DENTRY_PER_BLOCK;
uint32_t new_dentry_block_off = dentry_count % VVSFS_N_DENTRY_PER_BLOCK;
```

If the `new_dentry_block_pos` is greater or equal to the current block count, we need to allocate a new block. This is done via `vvsfs_assign_data_block`, which first determines if we are allocating a direct block, in which case it does and returns the new block address. If we are allocating an indirect block, we first check if we have an indirect block allocated already, if we don't then it is created. Next we create a new block to store as the first indirect block and put it's address in the indirect block. Lastly we return the first indirect block address.

If we don't need a new block, we call to `vvsfs_index_data_block` to retrieve the data block address for the block associated wtih `new_dentry_block_pos`. After this, we buffer the block, and write the dentry to the offset denoted by `new_dentry_block_off` which is guaranteed to be free, semantically guaranteed by our code's logical flow.

From here on out, we initialise the dentry with the required data as necessary, including name, inode number, etc.

Lastly, unlinking a dentry requires an interesting set of changes. We need to extend our previous 'dentry shifting' implementation to work with spatially-discontiguous (but logically contiguous) arrays. That is, the indirect and direct blocks are continugous notionally, as an array, whereby the indirect blocks require more calculation and logic to index into. We need to support three main cases, for dentry movement between:

1. Direct blocks only (using existing logic)
2. Indirect only (dentry moves between indirect blocks)
3. Indirect to direct (dentry moves between indirect and direct blocks)

The first case is straightforward, as we can check the block count, if it is less than 15, we delegate to the old logic. In the second case, we check that the block index (stored in `struct bufloc_t` passed as an argument from `vvsfs_find_entry`), is greater than or equal to 15, in which case we are only moving dentries within indirect blocks. From here we apply the same logic as the first case, but operate over indirect blocks instead of direct blocks. In the third case, we need to apply the same logic from the old direct-only implementation, but buffer both direct and indrect blocks at the same time.
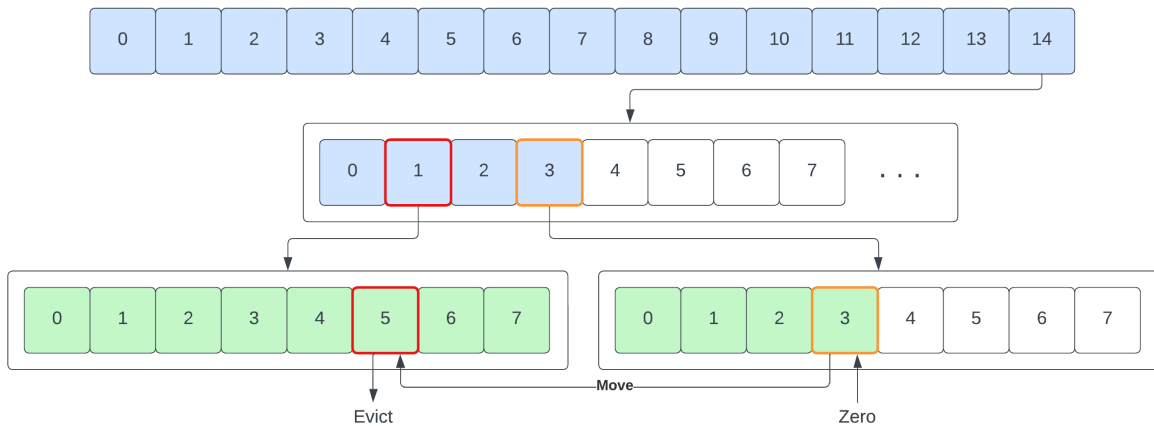

Figure 5: Indirect to indirect

In the second and third cases, a situation exists where the only remaining dentry of the last (but not only) indirect block is moved, requiring deallocation of that indirect block but not the indirect addresses block.
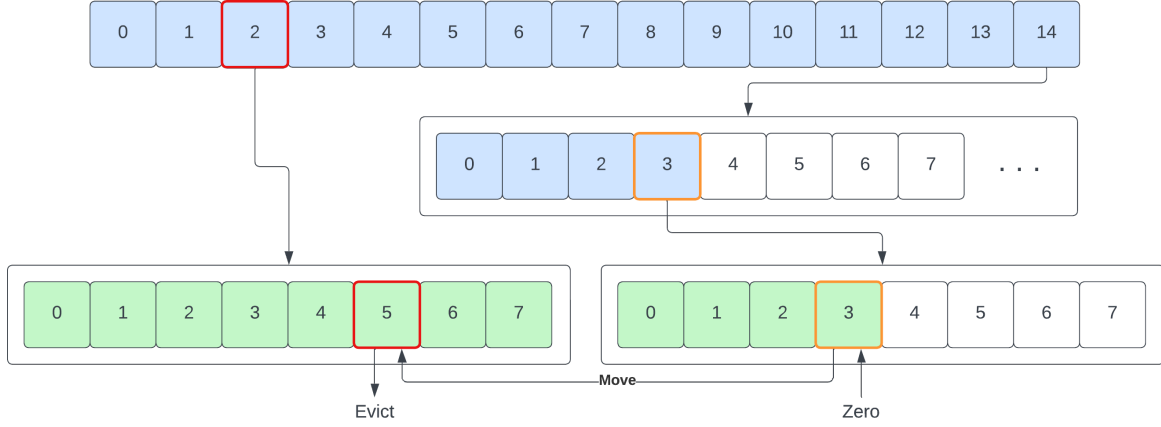

Figure 6: Indirect to direct

Another possibility in the second and third cases is that we could move the only dentry in the only indirect block to a direct block. In this case, it is necessary to deallocate both the first indirect block and the 15th direct block (which is the indirect addresses block).
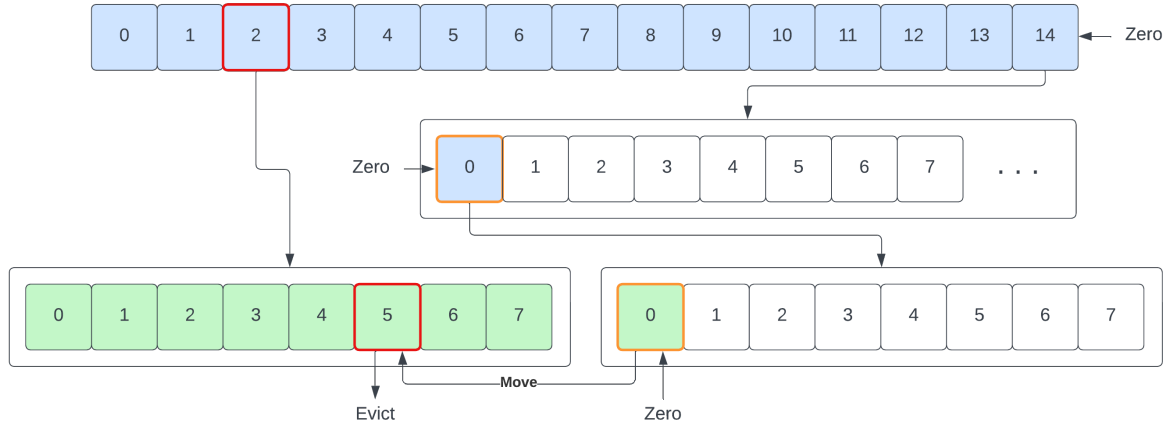

Figure 7: Indirect last to direct

4 Extensions

4.1 Hardlinks

At this stage, our code was based on the asumption that any given file can only be referenced from a single place. That is, all inodes persist a link count of 1, to themselves. The most notable methods affected by this assumption are `vvsfs_rename` and `vvsfs_unlink`, which need to remove a link to an inode. We chose to implement a common method which decrements the link count and conditionally frees the inode data blocks if the new link count is zero.

Having handled proper inode destruction when the link count reaches zero, we turned to the case of adding a new hardlink. We reused the code for creating a new file, but additional modifications were made to increment the link count of a given inode via `inode_inc_link_count`. This was followed by an `ihold` to inform the VFS that the link counts for the inode have changed.

4.2 Symbolic Links

To grasp the initial concepts and implementation details involved with symlink support in a filesystem, we examined the `ext2` and `minixfs` implementations. The most notable points was their use of `page_get_link` and `page_symlink`. We followed this implementation, regarding the given `file` structure as the target. However, this brought a large complication, which was that `vvsfs_write_end` assumes `file` is always null. After extensive bug hunting in our code, we discovered that `page_symlink` always calls `vvsfs_write_end` with a null file parameter as it utilised anonymous memory maps for the file itself. In order to fix this, we changed the usage of `file->f_inode` to `mapping->host` to retrieve the inode pointer. This was reported to the forum which lead to it being fixed upstream in the assignment template.

To implement symlinks, we created a new inode utilising the `S_IFLNK` flag to mark it as a symlink. Then, we invoked `page_symlink` to create a new anonymous memory map and write the symbol name (the original file location) to it, completing the link. Lastly, a dentry is created in the parent directory of the symlink location to act as the binding between the anomymous page in the inode and the parent directory.

4.3 Special Devices

Implementing special devices was very similar to the third baseline task of storing more attributes. This is because we simply needed to store a new attribute `i_rdev`, update the `iget` method to correctly instantiate a new inode, and update the `vvsfs_write_inode` to sync this new information to the disk. After this we created `vvsfs_mknod` based on `vvsfs_create`, and updated the `vvsfs_new_inode` helper method to be able to create a special node.

5 Testing

We created our own test suite (found in `vvsfs/vvsfs_tests`), which can be run using the script `run_all_tests.sh`. The suite is composed of a set of helper scripts that provide automatic generation of a test environment, and an assertion framework to provide nice error messages. We used this suite as part of a test driven development methodology (where we would create tests for expected behaviour and build new features to make them pass), as well as for regression testing (whenever we discovered/fixed a bug, we would write a new test to prevent it from occuring again).

There is one bug found by our test suite which we were unable to fix before submission - the assertion on line 60 of `test_mv_overwrite.sh`. This test involves creating two files in the same directory, and then renaming one to the other, causing an overwrite. Our assertions to check that the move was successful initially pass, however after a remount, the destination file contents are wiped. This behaviour is non-deterministic, and usually does not occur until the test suite has been run 3-4 times on a fresh VM image. Our investigations have led us to believe that the provided template code allocates the same data block twice in some cases; however despite extensive attempts by all of our team members and our tutor (Felix) to try and diagnose the issue, we were unfortunately unable make any progress.

In addition to our own suite, we used the pjdfstest filesystem test suite to check our implentation for POSIX compliance and various other edge cases. By the end we passed all tests with the following exceptions:

1. Large file tests (2gb).

2. Folder link count checks. These failed because our filesystem does not keep track of the reserved `.` & `..` files in directories, which was an implementation decision we made based a note posted by Alwen on Ed.

3. The filesystem does not correctly update `ctime` on truncation.

4. The filesystem does not store high presision time, only seconds like minix & ext2.

Finally, we ran our filesystem through xfstests and fs-fuzz test suites. This uncovered a large number of concurrency issues which we believe are related to the provided assignment template code. We suspect they may be related to the double block allocation issue mentioned above. If we had more time we would try to find the cause and solve these synchronisation related problems.