HW11 Solutions

1. Necessity of File System State

No, it's not fundamentally necessary. This is because the system could, on restart, traverse the entire file system (starting from the root directory). This would involve marking as in-use any inode or block that is reachable. (This of course would be inefficient.)

2. File System Operations

2.1. Creating /foo/bar

Assume that the <fname,ino> mapping in a directory lives in the directory's inode:

- R inode of '/'
- R inode of '/foo'
- R free inodes and pick one as '/foo/bar'
- W mark inode not free
- W inode of '/foo' (add <bar,ino>)
- R free sectors and pick one
- W inode of '/foo/bar'
- W data to the data block of '/foo/bar'
- W mark data block not free

2.2. Write Operations Order

We consider only the writes. Principle: We should refer to a inode/data block only after the inode/data block is properly setup.

1. W data to the data block
2. W inode to reference the data block
3. W mark the inode not free
4. W mark the data block not free
5. W inode of '/foo' (add <bar,ino>)

2.3. FSCK Requirements

In order to get 2.2 to work, we know fsck at least needs to:

1. Examine all allocated inodes, check the pointers, if it points to a block marked as free in the free block bitmap, modify the bitmap.
2. Examine the directory hierarchy, any unreachable and allocated inode will be freed.

2.4. Logging Protocol

Assume redo-only logging:

1. W (Log) BEGIN_TX
2. W (Log) log a write to data block, need to log blk# and data written to the blk
3. W (Log) log a write to an inode, need to log i-number and values written to this inode
4. W (Log) log a write to free inode bitmap (Not necessary)
5. W (Log) log a write to free disk blk bitmap (Not necessary)
6. W (Log) log a write to inode of '/foo', need to log the blk# and content written to it
7. W (Log) END_TX
8. W data to the data block
9. W inode to add the data block
10. W mark the inode not free
11. W mark the data block not free
12. W inode of '/foo' (add /foo/bar)

In terms of the ordering: (1 || 2 || 3 || 4 || 5 || 6) < 7 < (8 || 9 || 10 || 11 || 12)

By which we mean:

- 1-6 can be done in any order, but all have to happen before 7
- 8-12 can be done in any order, but all have to happen after 7
- 4, 5 are not necessary since 2 and 3 have already logged enough information for 4, 5

For recovery: The system should scan through all log entries. Starting from the beginning of the log towards the end, the system should replay all log entries for transactions that have an END_TX record with an id that matches the BEGIN_TX record.

3. Directory Operations

Assume that directories store their entries inside the directory inode. Further assume that the FS performs garbage collection in the background (to recycle inodes and data blocks), so we don't have to worry about bitmaps. Then the operations are:

- Write a new directory inode for /home/alice that is a copy of the old one (call the old one D), except removing the entry <lab.c, NUM>, where NUM is the inode number. Call this new directory inode D'.

- Write a new directory inode for /home that is a copy of the old one, except <alice,D> is replaced with <alice,D'>. call the new directory inode for /home E' and the old one E.
- Likewise for /: write a copy of the inode for / (call the old one F), replacing <home,E> with <home,E'>. Call the new inode F'.
- *Rewrite* (not copy) the uberblock to point to F', rather than F.

The only ordering constraint is that the last one (the uberblock) happens after all of the others. The others can happen in any order.

4. Redo-Undo Logging Protocol

The redo-undo logging protocol allows the file system to checkpoint (flush to the file system data structures) updates from *uncommitted* operations. If there were a crash during an uncommitted operation, which the file system had partially applied, then a recovery protocol without the undo pass would leave the file system in an inconsistent state (with the uncommitted operation partially applied).