Orphan file

In unix there can inodes that are unlinked from directory hierarchy but that are still alive because they are open. In case of crash the filesystem has to clean up these inodes as otherwise they (and the blocks referenced from them) would leak. Similarly if we truncate or extend the file, we need not be able to perform the operation in a single journalling transaction. In such case we track the inode as orphan so that in case of crash extra blocks allocated to the file get truncated.

Traditionally ext4 tracks orphan inodes in a form of single linked list where superblock contains the inode number of the last orphan inode (s_last_orphan field) and then each inode contains inode number of the previously orphaned inode (we overload i_dtime inode field for this). However this filesystem global single linked list is a scalability bottleneck for workloads that result in heavy creation of orphan inodes. When orphan file feature (COMPAT_ORPHAN_FILE) is enabled, the filesystem has a special inode (referenced from the superblock through s_orphan_file_inum) with several blocks. Each of these blocks has a structure:

Offset	Type	Name	Description
0x0	Array of __le32 entries	Orphan inode entries	Each __le32 entry is either empty (0) or it contains inode number of an orphan inode.
blocksize-8	__le32	ob_magic	Magic value stored in orphan block tail (0x0b10ca04)
blocksize-4	__le32	ob_checksum	Checksum of the orphan block.

When a filesystem with orphan file feature is writeably mounted, we set RO_COMPAT_ORPHAN_PRESENT feature in the superblock to indicate there may be valid orphan entries. In case we see this feature when mounting the filesystem, we read the whole orphan file and process all orphan inodes found there as usual. When cleanly unmounting the filesystem we remove the RO_COMPAT_ORPHAN_PRESENT feature to avoid unnecessary scanning of the orphan file and also make the filesystem fully compatible with older kernels.