Ext4, btrfs, and the others

Jan Kára
SUSE Labs, Novell
Lihovarská 1060/12
190 00 Praha 9
Czech Republic
jack@suse.cz

1 Abstract

In recent years, quite a few has happened in the Linux filesystem scene. Storage is becoming ever larger, solid state disks are becoming common, computers are joined into clusters sharing a storage... That brings new challenges to the filesystem area and new filesystems are developed to tackle them. In this paper we present a design and compare two general purpose filesystems under development: ext4 and btrfs. They are considered as the most probable successors of the current filesystems such as ext3 or reiserfs.

2 Ext4

Ext4 is a direct successor of the well-known ext3 filesystem. Ext4 is implemented as fully backward compatible. I. e., current ext3 filesystem can accessed and modified using ext4 filesystem driver. As the filesystem driver refrains from using incompatible features unless explicitely told so, you can later mount the filesystem as ext3 again. However you will not be able to use most of the advantages of ext4 without enabling incompatible features so this behavior is useful mostly during the transition period.

2.1 Disk layout

The basic disk layout of ext4 is similar to the one of ext3. Following a traditional Unix filesystem design, each file and directory in the filesystem is described by an object called *inode*. Inode contains information about its type (file, directory, symbolic link, etc.), file size, access rights, owner, and where data is stored. All inodes in the filesystem have the same size which is configurable when creating the filesystem. The inode size is a power of two between 128 bytes and the filesystem block size. Current default inode size is 256 bytes. Inode size above 128 bytes is needed to store inode timestamps with a sub-second precision and to allow storage of extended attributes (such as ACLs) in the inode (extended attributes which do not fit into the inode are stored in a special block).

One of the key differences between ext3 and ext4 is the way how numbers of blocks with file or directory data are stored. Ext4 remembers numbers of blocks with data in *extents*. An extent is a continuous run of physical blocks carrying data for a continuous run of logical file blocks. So for example a single extent describes that data from logical blocks $3 - 10$ of file *foo* are located in physical blocks $2101 - 2108$. Such description is much more space efficient than simple block pointers used by ext3 on an unfragmented filesystem as a length of an extent can be upto 2^{15} blocks in ext4.

To allow fast lookup which physical block describes a given logical block, extents are kept in a b-tree indexed by the starting logical block number of an extent. The root node of the b-tree is stored in the inode itself and can contain six extents (or indices to lower levels). If the extents do not fit into the inode, other nodes of the b-tree are stored in blocks allocated for this purpose. When a node of the b-tree gets

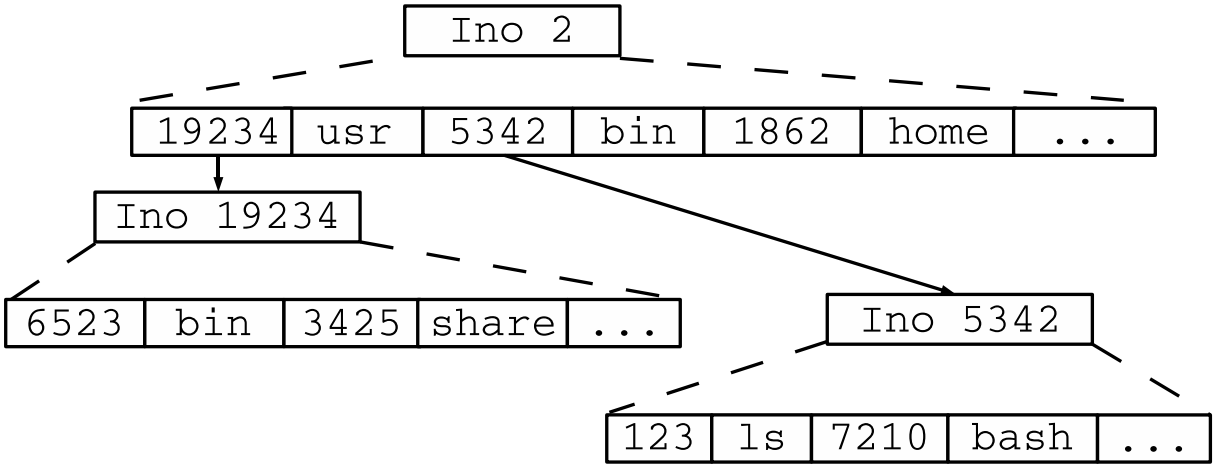

Figure 1: Directory structure of an ext4 filesystem

empty, the block is released. Currently, optimizations of the b-tree such as merging of index blocks or reduction of a depth are not implemented.

Format of directories is exactly the same in ext4 as it is in ext3. So let us just quickly remind it. Following a traditional ext3 design, directory is stored as a file which contains directory entries instead of data. Each directory entry contains a name of a file, directory, symbolic link, etc. and a number of inode which contains details about it (see Figure 2.1). To speedup searching in large directories, an *htree* feature can be enabled. With this feature, a search tree of directory entries is maintained. The nodes of the tree are hidden in a special directory entries and thus this feature is fully backward compatible.

Now let us review the global filesystem structure. The filesystem is divided into *block groups* (or simply groups). The number of blocks in a group is exactly the number of bits in a filesystem block so that bitmap of used blocks for a group fits into one filesystem block. Thus for a common 4 KB block size, the size of a group is 128 MB. Each group contains a sequence of blocks in which inodes are stored (this area is called *inode table*), a bitmap of used blocks and a bitmap of used inodes. A well-known limitation of ext3 filesystem which is kept in ext4 is that the size of inode table in a group and thus the maximum number of inodes in the filesystem is set when creating the filesystem and cannot be changed later. A new feature of ext4 called *flexible block groups* is that blocks for inode table and bitmap blocks need not be inside the block group. This allows for more flexible placement of these metadata (used for example to achieve lower filesystem check time). Another important improvement of ext4 over ext3 is that all block numbers in the filesystem are stored as 48 bit numbers. Thus the maximum size of the filesystem with common 4 KB blocksize has been raised from 16 TB to 1 EB (2^{60} bytes).

First few blocks of the first group contain a *superblock* and *group descriptors*. In the superblock is stored basic information about the volume such as filesystem a block size, a number of blocks, a number of inodes, supported filesystem features, a number of free blocks, etc. In group descriptor are stored a number of a block with block bitmap, a number of a block with inode bitmap, a number of the first block of inode table, a number of free blocks in the group, a number of free inodes in the group, etc. A copy of the superblock and group descriptors used to be stored in the beginning of all the other groups. In last few years, *sparse super* feature is commonly used and thus a copy is kept only in the second group and then every group whose number is odd and divisible by three, five, or seven.

2.2 Journaling

There are some improvements in the area of journaling as well. Ext4 uses new journaling layer JBD2 which supports 64-bit block numbers. Another new feature of JBD2 is journal checksumming. A CRC32 checksum is computed over all blocks in each transaction (these are just metadata blocks in case of

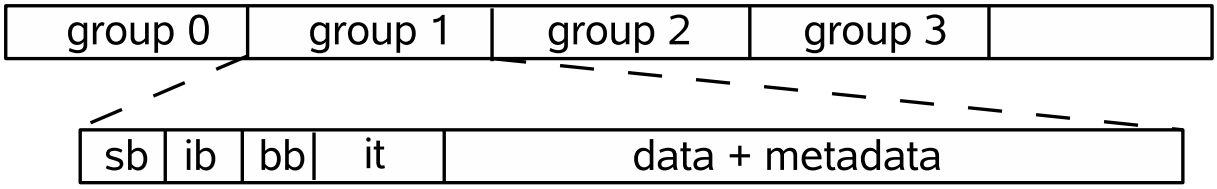

Figure 2: Global layout of an ext4 filesystem. *sb* denotes blocks containing (a copy of) superblock and group descriptors in the picture. *ib* denotes a block containing group inode bitmap, *bb* denotes a block containing group block bitmap, and *it* denotes blocks containing inode table.

common ordered or writeback journaling modes). When replaying the journal after a crash, the checksum is verified to detect whether a journal data has not been overwritten by invalid data while the power was failing (this is a common failure mode of a PC class hardware). If the verification fails, journal replay is stopped. Note that this feature does not prevent filesystem corruption in case of a corrupted transaction[1].

2.3 Allocation algorithms

Another key difference of ext4 to ext3 is a different strategy of block allocation. Ext4 supports so called *delayed allocation*. The principle of this feature is that during a write system call, filesystem just estimates how many new blocks it will need to allocate. It reserves this amount of blocks but does not allocate any particular blocks yet. The real allocation of blocks is done when memory management decides to flush written data from memory to disk. This delaying of allocation allows for better decision about the final size of the file and thus for better allocation decisions:

- More blocks can be allocated at once thus reducing fragmentation and CPU utilization.
- Writes happening at random offsets (possibly also by different threads) are joined into one monotone writeout sequence.
- In case of short-lived temporary files, filesystem can avoid allocating any blocks at all.

A disadvantage of this feature is that in case of a crash, larger amount of data is lost compared to standard ordered mode without delayed allocation. With a standard system configuration, writeout of data happens once every 30 seconds and it need not write all of the written data. On the other hand in ordered mode without delayed allocation, data is guaranteed to hit the disk after 5 seconds from the write.

To reduce fragmentation and to decrease CPU utilization when allocating large numbers of blocks, ext4 features a new block allocator *mballoc*. The core of mballoc is a plain buddy allocator [7]. The bitmap of buddy blocks is generated as needed from the on disk bitmap and is cached in page cache so that it gets automatically released under memory pressure. Because buddy allocation provides chunks of blocks larger than requested, blocks unused by the current allocation are added to inode's preallocation list. When a subsequent allocation happens at a logical block which is covered by some extent in the preallocation list, this extent is used to satisfy the allocation.

To pack small files together and thus avoid fragmentation of free space, mballoc also keeps a list of preallocated extents for each locality group (a *locality group* is defined by a CPU the allocation is running on). When a file is smaller than 16 blocks (tunable via `/sys/fs/ext4/<dev>/mb_stream_req`), mballoc does not add unused blocks to a per-inode list of preallocated extents but rather to a list of preallocated extents of the current locality group. When a subsequent block allocation for a small file happens, mballoc first checks whether it can be satisfied from the preallocation list of the current locality group.

[1] Part of the metadata from the transaction could have been already written to the filesystem before the crash by kernel. Thus when we fail to replay the transaction, the invariant that either none or all blocks in a transaction are written is violated and filesystem is left in an inconsistent state.

Before we start looking for a space for a file, we try to estimate the final file size and allocate blocks for that size. Searching for free blocks happens in several rounds. In each round we walk all the groups in the filesystem starting with the goal group (that is a group where the last allocation of blocks for the file happened) looking for a satisfactory free extent. The rounds differ by requirements they put on the length of the free extent. In the first round, we require a completely free buddy to satisfy the allocation. In the second round, we are satisfied with a free extent that has at least size of a stripe of a raid array and is properly aligned (this round actually happens only for small files and when stripe size is set in the filesystem). In the third round, we consider all free extents. We try to find a free extent which has enough blocks to satisfy the allocation. If there is no such free extent, we just take the largest one. Otherwise we take the smallest extent that is still able to satisfy the allocation. As doing such full scan is rather CPU intensive, allocator stops searching for a free extent after considering `mb_max_to_scan` extents and simply takes the best extent found so far. Also when the allocator already found extent which is large enough to satisfy the allocation, it stops searching for better extent after considering `mb_min_to_scan` extents. These two parameters can be tuned via `/sys/fs/ext4/<dev>/`.

2.4 Other features

Ext4 also supports fallocate system call. That enables user space application to explicitely preallocate space to a file. This system call is more efficient than writing zeros to the file[2]. It does not really write the data blocks, it merely allocates them and marks them as uninitialized so that user sees zeros when the blocks are read. Ext4 stores the uninitialized bit in its extent structure.

Another useful feature of ext4 is a support for online defragmentation. In the current 2.6.31 kernel, only a simple ioctl `EXT4_IOC_MOVE_EXT` is supported. It atomically copies data from a source file to a target file and makes inode of the source file point to the data blocks of the target file. In the development tree are patches which implement ioctls allowing control of allocation algorithms and thus defragmentation program can allocate target file at desired location and possibly make much more sophisticated allocating decisions than a kernel can do.

3 Btrfs

Btrfs (an acronym from b-tree filesystem, pronounced as “butter eff ess”) is a filesystem written from scratch. Its development was started by Chris Mason in 2007. So far it is under intensive development and does not have a completely fixed on disk format (although changes are scarce now) but most of the important features are already there.

3.1 B+trees

The core data structure of btrfs is *B+tree* [8]. It is a tree which stores index keys in internal nodes of the tree and data associated with keys is stored in the leaves. All filesystem metadata is stored in this tree (this design is similar to the one used by reiserfs). Modifications of the tree happen in a copy-on-write manner. I. e., when a block is to be changed, it is written to a new location instead of modifying it in place. This is an alternative way to the currently standard journaling how to keep filesystem easily recoverable after a crash. Naturally, copy-on-write handling requires that all references to the modified block are rewritten as well. As classical B+trees have links from each leaf to its neighbors (see Figure 3.1), copy-on-write handling of such trees would be impractical (the change would propagate via links to all the leaves and thus a whole tree would have to be written to a new location). Thus btrfs uses B+trees without linked leaves. See Figure 3.1 for an example of a copy-on-write modification of a B+tree without leaf links.

A key in the filesystem tree is comprised of three components:

- *objectid*: Each logical object in a filesystem (inode) is assigned a unique object id. This id is similar to inode number in ext4.

[2] Only when the filesystem supports it. Otherwise glibc just falls back to writing zeros to the file

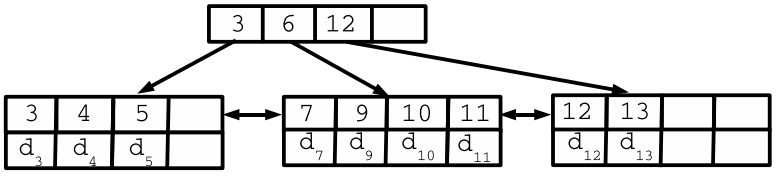

Figure 3: A classical B+tree. Numbers are the tree keys, d_i are data associated with key i.

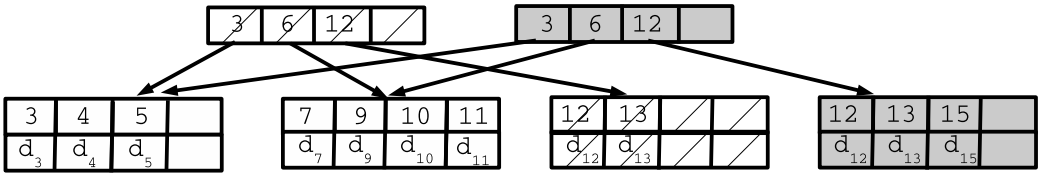

Figure 4: Insert of a key 15 to a B+tree without leaf links handled in a copy-on-write manner. Grey boxes denote newly allocated blocks, hatched boxes denote blocks that are no longer used by the tree. Note that the root block has been allocated in a new place because a reference to a leaf block had to be changed.

- *type*: Type of the item - for example stat data (information about file size, owner, permissions, etc.), directory entries, file data, or extent (describes position of file data).
- *offset*: Offset of the item (used for example for an offset of data an extent is describing, or for a hash of a name described by a directory entry).

As objectid forms the most significant bits of the key, items related to the same inode are packed close together in the tree. Therefore all the filesystem metadata related to one file can be read effectively.

In fact there are more B+trees in the filesystem. There is the main one with items containing inodes, directories, inline data, extents describing position of data, etc. Then there are five other trees carrying special items. We describe some of them below.

3.2 Directories and small files

Directories are formed by a set of directory entry items with the objectid of the directory inode. Each directory item contains all directory entries that have the same hash of their name (CRC32 is used for hashing)[3]. This scheme allows for an efficient lookup even for very large directories. Also, because of the natural tree-like structure of the filesystem, creations and deletions of files are fast even in large directories as well. For some applications it is advantageous to return directory entries from a readdir system call in an order which generates as sequential access to disk as possible (e. g. for those applications that intend to scan the whole directory). Therefore btrfs keeps for each directory entry a special index item with a secondary entry index. Readdir system call uses this secondary index to iterate over directory entries in a directory. Currently, the implementation of the secondary index is rather simplistic (it is simply increased by one when an entry in a directory is created) but an improvement is planned.

In btrfs small files are not stored in a dedicated data blocks but rather directly in the tree in data items. This reduces time to access such files (as the data is stored close to metadata) and also saves on disk space. The maximum size of files stored this way is configurable `max_inline` mount option and is of course also limited by the maximum size of the tree leaf (one block). Default maximum size of files stored in the tree is 4 KB.

[3] So far the case that directory entries with the same hash do not fit into the tree item is unhandled.

3.3 Snapshots

One of the less common features btrfs offers is an efficient creation and handling of writable filesystem snapshots (also called clones). When a snapshot is created it is exact copy of a filesystem at that point in time stored in a subdirectory of the filesystem. From that moment on, modifications to the original and the snapshot are separate (i. e., changes to the snapshot cannot be seen in the original and vice versa). Hence, from user point of view, snapshotting behaves as if a block device underlying the filesystem has been atomically copied to a new block device at the moment the snapshot has been taken and then mounted to a directory of the parent filesystem. Snapshots are implemented by referencing the root of the filesystem tree (before a directory in which snapshot should reside is created) from a freshly created directory. Thus snapshot and the original share all unchanged nodes of the tree and unchanged data extents. Each extent (describing either data or tree node) holds a reference count (number of references to it) to allow easy detection when data / metadata described by it are no longer needed and can be freed. For details about reference counting, we refer reader to the paper of Ohad Rodeh [4] where reference counting algorithms used by btrfs are described.

In the above we described snapshotting of the whole filesystem. In fact btrfs supports creation of snapshots for arbitrary directory or file in the filesystem.

The snapshot mechanism is also used to implement fast recovery after a crash. Filesystem automatically takes unnamed snapshots approximately every 30 seconds and after a crash filesystem is recovered to the state of the last fully created snapshot. Old unnamed snapshot gets removed as soon as a new one is created.

3.4 Checksumming

To detect hardware errors, btrfs checksums all data and metadata. Checksums are independent of the data position so writes ending in a different location are not detected by this mechanism. Each tree block contains a checksum in its header. Currently, only CRC32 is supported but the disk format is designed to support different checksums and there are 32 bytes reserved for a tree block checksum. Checksum of ordinary data is computed for each data block (4 KB) separately. Checksums are stored in a separate checksum tree on disk. The tree is indexed by offset of data block in the filesystem and checksums of several blocks are merged into a single checksum item to reduce overhead of item headers.

3.5 Multiple device support

Btrfs provides also powerful handling of multiple devices. Administrator can add and remove devices from a pool of devices used by a filesystem online. Space is allocated for filesystem's use in *chunks*. Each chunk can be either directly mapped part of a device from the pool, a part of a device which is mirrored to another location, or it can be created by combining parts of several devices in one of the three modes — RAID0 (standard striping), RAID1 (mirroring), RAID10 (stripe of mirrors). The size of chunks is a few GB and they together form a single logical address space of blocks. All tree handling code and extent pointers use these logical addresses. Information about chunks is stored in a special filesystem tree. Superblock contains enough information so that addresses in a tree with chunks can be properly translated on mount.

Remarkable is that removal of devices from the storage pool is quite efficient. Because the filesystem keeps references from allocated extents to items which use them, it is an easy task to move all the data and metadata from one device to another one and update all references properly.

3.6 Allocation algorithms

Similarly to ext4, btrfs supports delayed allocation to decrease file fragmentation. Other allocation algorithms are different. On disk, free space is kept in a dedicated tree of free extents. In memory, btrfs caches information about free space. It keeps a red-black tree of extents of free space as long as its size

is less than 16 KB per GB (i. e., as long as it is half of the memory a simple bitmap would use). When memory used by the red-black tree would cross this boundary, only bitmaps are added to the tree to track free space. Thus filesystem uses at most 32 KB per GB to cache free space information. Keeping free space in form of extents is beneficial not only to save memory but mainly to speedup searching for free space. Creation of in-memory data structures for tracking free space from on-disk b+tree of free extents is performed on demand by dedicated kernel threads.

An algorithm searching for free space has three different modes:

- Normal mode for standard rotating media.
- SSD mode (enabled by `-o ssd` mount option) for SSD disks which handle random writes well (like Intel SSDs).
- SSD spreading mode (enabled by `-o ssd_spread`) mount option for SSD disks that do not handle random write pattern very well.

Btrfs differentiates between allocation of space for data and for metadata. Depending on the purpose allocation and on allocation mode, algorithm either directly searches for a continuous extent of free space in each suitable allocation group (a *group* in btrfs corresponds to a chunk described above; see below which groups are considered suitable) or it performs a cluster allocation. A *cluster* is a sequence of extents that are close together and contain at least requested number of bytes plus some amount depending on the allocation purpose and allocation mode. In the cluster allocation suitable block groups are searched for a cluster of free space and then allocation algorithm looks for an extent large enough to satisfy the allocation in the found cluster. If such extent does not exist, current cluster is discarded and the search for a free cluster continues. When a suitable extent is found, remaining extents in the cluster are stored and consulted in the subsequent cluster allocation before a search for a free cluster starts.

Before we explain in which cases which allocation strategy is used, let us explain which allocation groups are suitable. Allocation groups are of three types:

- System group — used for allocation of a chunk tree
- Metadata group — used for allocation of other filesystem trees
- Data group — used for allocation of regular file data

Searching for suitable group happens in five rounds. In the first four rounds, we perform allocation only from allocation groups of appropriate type. In the first round, we consider only allocation groups that have already free space information cached in memory. In the second, round we consider also allocation groups that are just being cached by a caching threads. In the third round, if we failed to find a free space in a group and its free space information is not fully fetched, we wait until the caching of free space is done and then rescan the group again. In the fourth round, we try to allocate a new chunk from a device pool to create new allocation group and then retry scanning. In the fifth round, we scan all allocation groups regardless of their type.

Now let us explain which allocation strategy is used when. For normal allocation mode (rotational media), data is allocated by the simple strategy which just searches for large enough free extent. Metadata is allocated by the cluster allocation strategy where cluster is required to contain requested number of free bytes plus 64 KB. For SSD and SSD spreading allocation mode both data and metadata allocations use cluster strategy (we keep one cluster for subsequent data allocations and one cluster for subsequent metadata allocations). A cluster is required to have requested number of bytes plus 2 MB of free space in SSD spreading mode. In normal SSD mode, the cluster is required to have $\max(bytes, (bytes + 2\text{MB})/4)$ bytes of free space for data allocation and $\max(bytes, (bytes + 2\text{MB})/16)$ bytes of free space for metadata allocation. An exception to these rules is the fifth scanning round in which we also relax allocation policies and are satisfied with any extent of large enough size. Finally, if allocation didn't succeed in any of the five rounds, we set the amount of required space to one disk block and retry the whole allocation procedure.

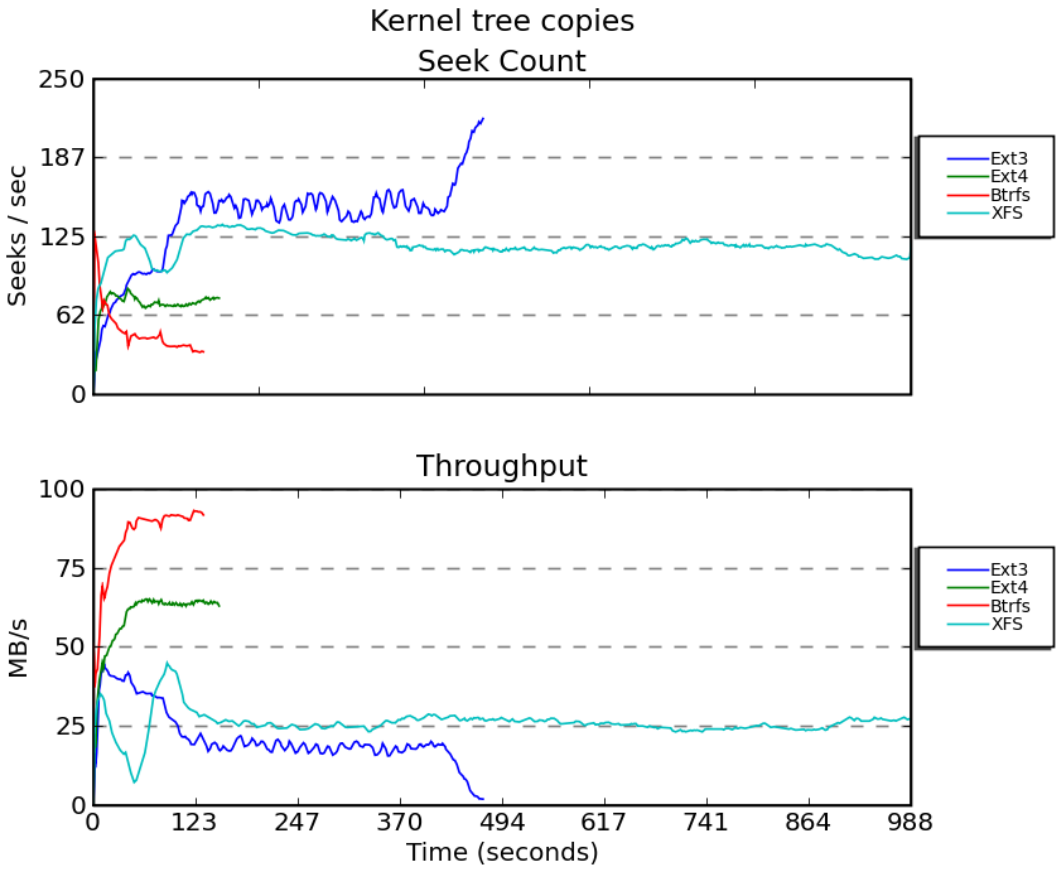

Figure 5: Creation of 30 kernel trees.

3.7 Other features

A few other btrfs features which deserve mentioning:

- Support for simple online defragmentation (`BTRFS_IOC_DEFRAG`). It is implemented by atomically copying the whole file into a new location.
- Support for fallocate system call in a similar way as in ext4.
- Support for transparent online compression and decompression of data using zlib.

4 Performance comparison

In this section we present a short performance comparison of ext4 and btrfs. The graphs also include other filesystems to give a broader idea of filesystems performance.

First we present three results from a desktop with two core CPU and a single SATA drive running 2.6.29 kernel.

The first test involves creation of 30 kernel trees. We see from results in Figure 5 that btrfs takes about 10% less time than ext4 to perform the task.

The second test measures reading of all files in a directory with 64000 files. Results in Figure 6 show that XFS outperforms btrfs quite a bit, ext3 and ext4 are far behind. This is caused by the fact that ext3 and ext4 return directory entries sorted by the hash value and not by inode number and thus reading the directory requires much more seeking.

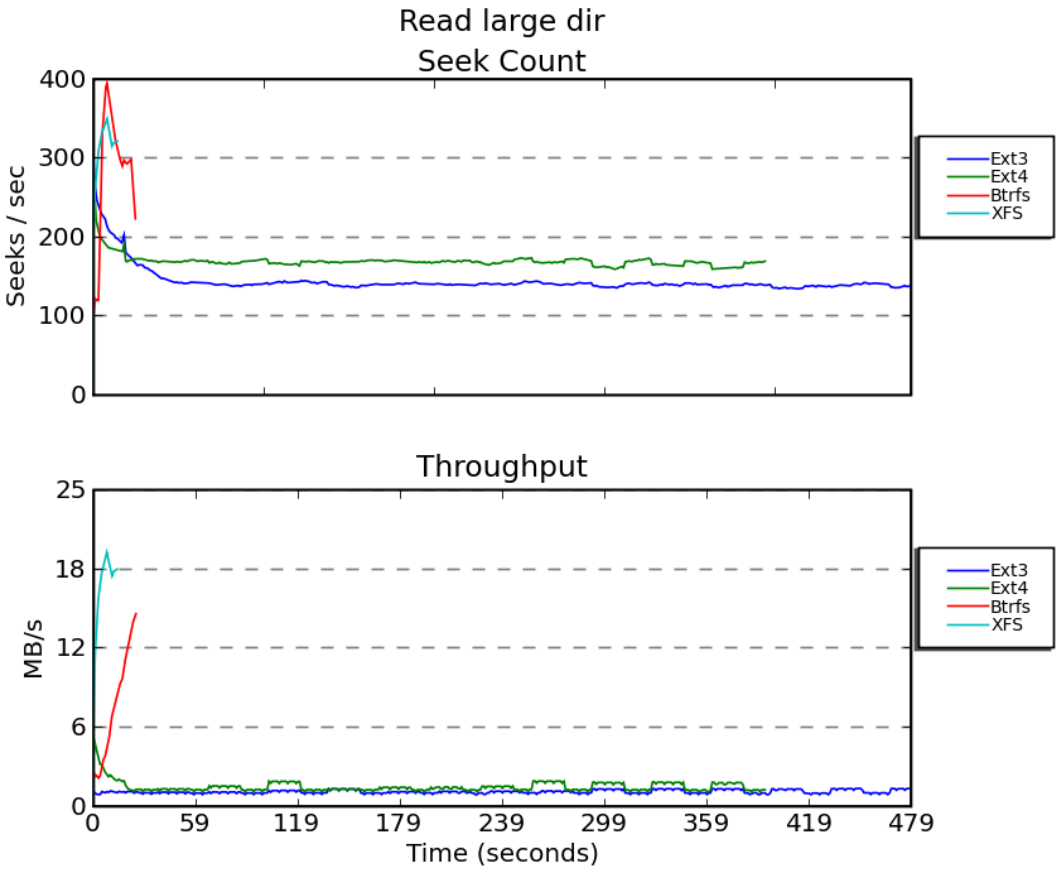

Figure 6: Reading of directory with 64000 files.

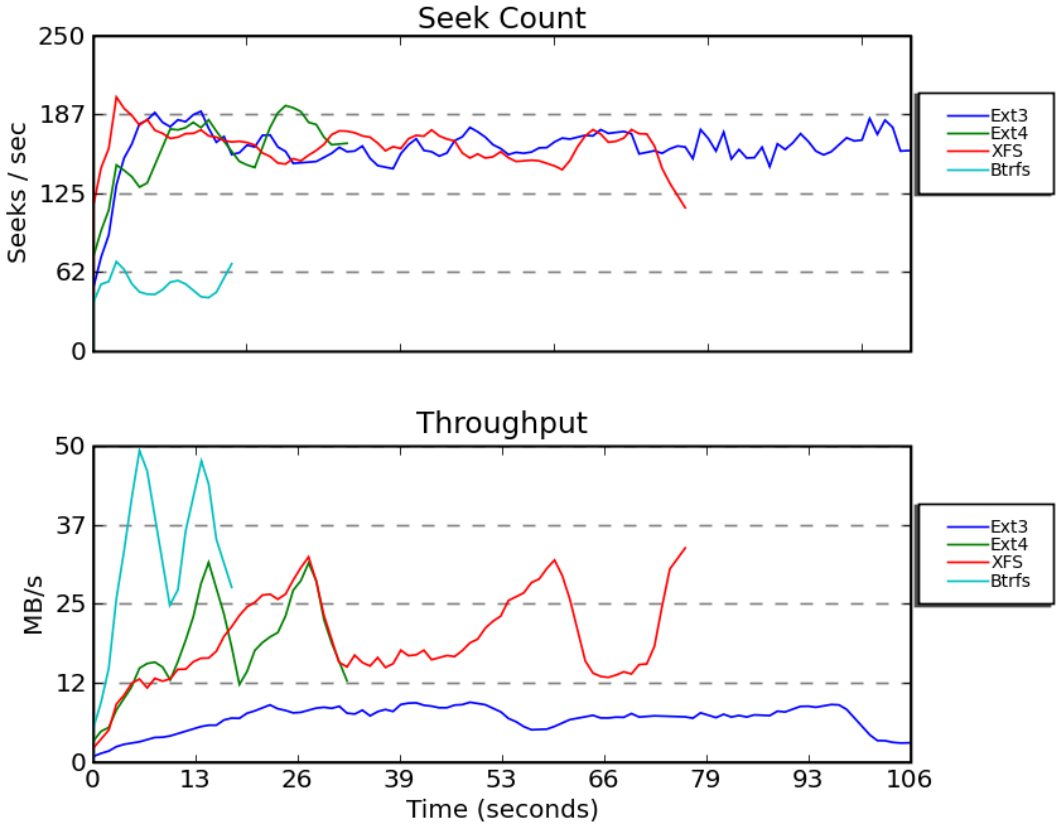

Figure 7: 100 threads writing synchronously 100 files.

In the third test 100 threads are writing and syncing 100 files. We see from Figure 7 that btrfs benefits from its special code handling syncing and outperforms all other filesystems.

Next we present three tests from FFSB suite. These tests were performed on a server class hardware with RAID array (see [6] for details) with kernel 2.6.30-rc7.

In the first test each of the 16 threads was creating 1 GB file using 4 KB writes. Results in Figure 8 show that XFS clearly wins, ext4 comes second and btrfs with ext3 are quite below. We'd like to remark that in newer kernels streaming performance of btrfs has been improved so that with copy-on-write disabled it matches the speed of XFS.

The second test simulates mail server workload. In this test the filesystem is seeded with 1 milion files spread across 1000 directories. Each of the 16 threads then randomly reads, writes, or deletes files. Results of the test are in Figure 9. In this workload ext3 performs the best and ext4 is close behind. XFS and btrfs end up quite below.

In the third test 1024 files of size 100 MB are created. Each of the 16 threads then writes 5 MB of data at random location into these files. From the results in Figure 10 we see that ext3 clearly wins. Btrfs with copy-on-write enabled is far below but with copy-on-write disabled it even outperformed ext4. To give a broader view of the filesystem area we also briefly introduce three other filesystems under development — reiser4, OCFS2, and ubifs.

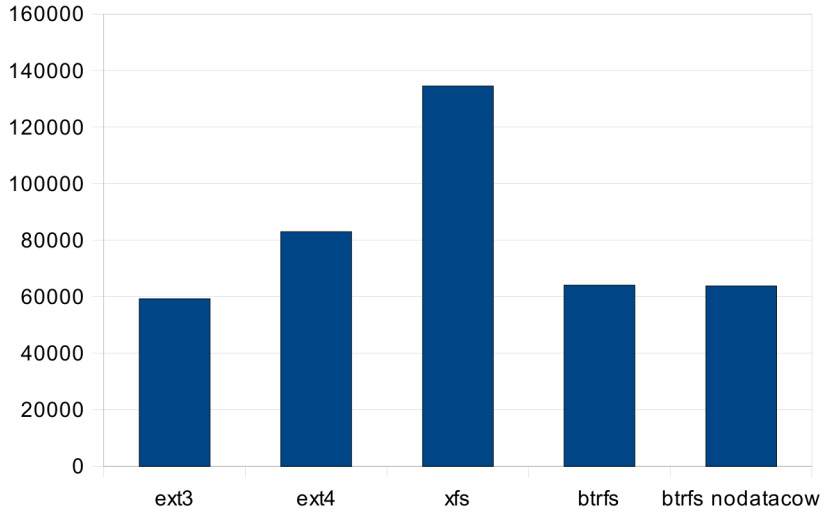

Figure 8: 16 threads creating 1 GB files.

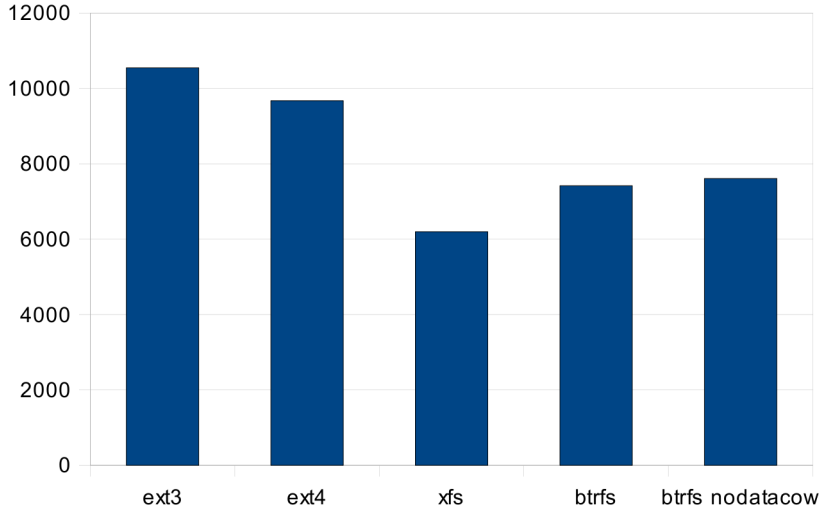

Figure 9: 16 threads simulating mail server load.

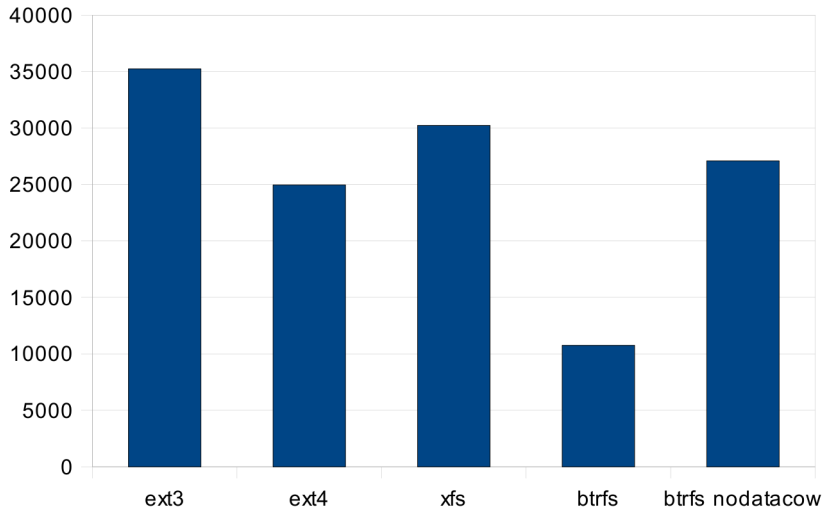

Figure 10: 16 threads randomly writing to files.

5 Other filesystems under development

There are also other filesystems under development in Linux. In this section we briefly introduce some of them to the reader.

5.1 Reiser4

Reiser4 is general purpose filesystem, a successor of the well-known reiserfs filesystem. It is being developed since 2004 but till now is not present in mainline kernel. The basic structure of the filesystem is similar to the old reiserfs — all filesystem data and metadata is stored in one B+tree. The filesystem is kind of hybrid between a standard journaling filesystem and a copy-on-write filesystem. For some kinds of modifications it performs copy-on-write and for other kinds it performs standard journaling. Reiser4 also supports transparent encryption and compression. The code is designed to allow easy extension or replacing of functionality through *plugins* (essentially the filesystem is separated into several modules with defined interfaces). The future of the filesystem is uncertain as btrfs seems to have most of the advantages of Reiser4 design and seems to attract more community attention.

5.2 OCFS2

OCFS2 is a filesystem designed to be used in clusters. I. e., in an environment where a single storage is attached via fiberchanel, SAS, or by other means to several computers (nodes). Nodes are additionally connected via network and want to concurrently access and modify the filesystem. Such setup is advantageous for example when several web servers are needed to handle load of a single site, or when several machines form a frontend to a database.

The on disk format follows traditional Unix design in its basic principles. Filesystem uses inodes, directories are just files with directory entries, etc. Differences from traditional design are for example that inodes are dynamically allocated, extents are used to track file's data, journaling is used to recover after a crash. Synchronization of computers accessing the filesystem is performed by DLM (distributed lock manager). Of course such synchronization implies a performance penalty in case two computers try to operate in the same area of the filesystem. The filesystem design tries to minimize such conflicts. So for example each node has its private journal in which it records changes, each node has reserved different area on disk from which it allocates blocks, etc.

Although OCFS2 possesses all features to be used as a general purpose filesystem and cluster locking can be turned off, the performance of the filesystem still is not at the level of other filesystems that focus on optimizing the single computer access.

5.3 Ubifs

Ubifs is a filesystem that has been designed for solid state devices. It does not work on top of a standard block device but on top of UBI layer, which is a layer accessing directly raw flash and performs tasks like wear-leveling and such.

Ubifs does not have the scalability issues old flash filesystems such as JFFS2 had. Most notably the mount time and memory consumption does not depend on the flash size (but the initialization time of the underlying UBI layer still depends linearly on the flash size) and the filesystem is designed to scale well to flashes of size several hundred GB. Similarly to btrfs, the filesystem utilizes B+trees to keep track of filesystems metadata and modifies the tree in a copy-on-write manner (which is dictated by properties of flash devices which do not allow modify-in-place).

Among other interesting ubifs features belong online compression and checksumming of stored data.

6 Acknowledgment

The author would like to thanks Chris Mason for providing the performance data and for consultations when writing this paper.

References

[1] Valerie Aurora: History of btrfs, http://lwn.net/Articles/342892/

[2] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya: *State of the Art: Where we are with the Ext3 filesystem*, OLS 2005

[3] Chris Mason: Btrfs, talk at Linux Plumbers conference, Portland, Oregon, 2008

[4] Ohad Rodeh: B-trees, Shadowing, and Clones. ACM Transactions on Storage, Volume 3, Issue 4, 2008, p. 1–27

[5] Sources of Linux kernel

[6] Btrfs FFSB benchmarks, http://btrfs.boxacle.net/

[7] http://en.wikipedia.org/wiki/Buddy_memory_allocation

[8] http://en.wikipedia.org/wiki/B+tree

[9] http://www.linux-mtd.infradead.org/doc/ubifs.html

[10] http://www.linux-mtd.infradead.org/doc/JFFS3design.pdf