Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Online ext2 and ext3 Filesystem Resizing

Andreas E. Dilger
adilger@clusterfs.com,
http://www-mddsp.enel.ucalgary.ca/People/adilger/

Abstract

It is difficult to predict the future, yet this is what you have to do each time you partition a disk. There are several HOWTOs giving advice on ways to partition a disk for Linux, yet everyone's usage pattern is different. Invariably, your filesystems fill up, often in the middle of doing something important.

With the advent of Linux LVM in 1999, Linux was finally getting to the stage where one could add space to "partitions" dynamically. The missing link was allowing the widely-used ext2 filesystem to grow to use newly added disk space without having to kill your applications, unmount the filesystem, do an offline resize (which was in its infancy at that time also), and remount the filesystem.

We start with a brief overview of the layout of the ext2 filesystem to give an understanding of the constraints behind the design of the online ext2 resizer. The three ext2 resizing scenarios are discussed, and implementation of each case is presented. The rationale behind offline filesystem preparation is given, and the (incompatible and yet unimplemented) alternative is presented. We continue with the requirements for ext3 online resizing discuss how this leads to a totally different implementation.

1 Introduction

One of the many reasons why a system stops doing the job it is intended to do is because it runs out of space in an important filesystem. While it is possible to increase the size of a partition or disk which is in use (via software or hardware RAID, LVM [LVM], and more recently EVMS [EVMS]) you normally have to stop applications and unmount ext2 and ext3 filesystems in order for the filesystem itself to be resized to take advantage of this increased space. To avoid an interruption to the system (and applications, and users), one has to be able to grow ext2 and ext3 filesystems while they are mounted and in active use (i.e. read and write operations in progress, current working directory of a process, etc).

The `GNU ext2resize` package [resize] is GPL licensed code which contains the `ext2online` tool and a kernel patch, which together allow increasing the size of a mounted filesystem without interruption to processes using that filesystem. In addition, the `ext2resize` tool also allows you to grow and shrink an unmounted filesystem. The `ext2prepare` tool is also part of `GNU ext2resize`, and is discussed later. While `ext2online` only allows one to increase the size of a mounted filesystem, in a vast majority of cases it is increasing the free space in a filesystem which is the critical operation

needed to keep an application running. In rare cases you might need to shrink one filesystem in order to grow another, but given the ease of increasing the size of a mounted filesystem on a system using LVM and online filesystem resizing, there is little need to make filesystems too large for their anticipated short-term growth as was needed previously.

The bulk of the ext2 online resizing kernel code was written in the fall of 1999 for the 2.0.36 and 2.2.10 kernels, but has remained relatively unchanged through all of the kernels since then, with only minor changes to locking and patch context. The 2.4 "code freeze" of the fall of 1999 prevented the patch from being added to the kernel at that time, and ongoing stabilization of 2.4 and other tasks have prevented me from doing more than minor code maintenance for the most part. The requirement to have online resizing for ext3 was the first restructuring of the kernel code, and involved a complete code rewrite. It is anticipated that the ext2 and ext3 online resizing code will be submitted for inclusion into the 2.4 and 2.5 kernels some time in the summer of 2002. The user space tools that form the `GNU ext2resize` package have undergone a slow evolution during their lifetime to support newer ext2 features such as large files and offline resizing of ext3 filesystems, as well as having more complete support for unusual filesystem layouts such as RAID stripe aligned metadata and filesystems whose inode tables are not at the same offset in every group.

In this paper we focus primarily on the *online* (mounted) aspect of filesystem resizing. For *offline* (unmounted) filesystem resizing, there are additional aspects of resizing, such as inode renumbering, moving the contents of data blocks and the inode table, and renumbering the data block pointers within an inode. The `ext2resize` tool can do all of these things. In order to keep the amount of kernel code to a minimum (and to make it actually work) these aforementioned operations are never done during online filesystem resizing.

2 Anatomy of an ext2 File System

In order to understand the constraints under which the ext2 filesystem resizer operates, we must first have some understanding of the on-disk filesystem layout. For the purposes of filesystem resizing, the ext2 and ext3 on-disk layouts are identical. We do not cover all aspects of the ext2 filesystem layout, such as inodes and directories, because for the purposes of online filesystem resizing those details are mostly irrelevant. They are covered in many other general ext2 papers [ext2].

The on-disk layout of the ext2 filesystem is strongly influenced by the layout of the BSD Fast File System. The disk/partition is divided into one or more sections, called *block groups*. Block groups are of a fixed size, determined at filesystem creation time, and all contain the same number of blocks, except the last block group which may have fewer blocks. By default, ext2 block groups are created at their maximum size (32MB for the default 4kB blocksize), and are numbered from the beginning of the filesystem starting with 0.

Each block group contains several key pieces of filesystem metadata, as shown in

Figure 1. For each block group, there is one block which is the *block bitmap*, one block which is the *inode bitmap*, and one or more blocks which make up the *inode table*. In addition, there may be a copy of both the filesystem *superblock* and the filesystem *group descriptor table* in a block group. Whether a block group will contain either a primary or

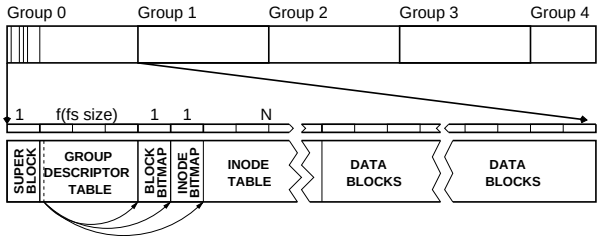

Figure 1: Block Group Layout

backup superblock and group descriptor table depends on the group number and/or parameters at filesystem creation time.

The block bitmap describes the allocation status of all data and metadata blocks within that block group. If a bit is set, this indicates that a block is in use as either a data or metadata block, and if it is clear then the block is available for allocation. Since the block bitmap is limited to a single block in size, this imposes the maximum size of a block group - the number of bits which will fit in a single filesystem block is 8 times the blocksize, and this is the maximum number of blocks that can be in a single group. For the last group in a filesystem, the bits representing blocks past the end of the filesystem will be set (marked in use) so that the kernel does not need to special-case the search for free blocks in the last group.

The inode bitmap describes the allocation status of the inodes in its group's inode table. Since the inode bitmap is limited to a single block in size, this imposes the maximum number of inodes that can be allocated in a single group. If there are less than the maximum number of inodes in a group, the bits corresponding to non-existent inodes will be set (in use).

The inode table contains one or more blocks which hold the inode data. Each group has the same number of blocks in the inode table, and this number is determined at filesystem creation time. Multiple inodes are packed into each inode table block, and fill the block completely, so this imposes the minimum number of inodes that can be allocated in each group - the number of inodes that fill a single block. The maximum number of inodes in each group is the same as with the maximum number of blocks in each group - the number of bits that fit within a single filesystem block, so 8 times the blocksize.

The superblock contains critical filesystem configuration parameters (e.g. blocksize, total number of blocks and inodes, group size, number of inodes in each group, etc.) and also dynamic filesystem status (e.g. the number of free blocks and inodes, the number of times the filesystem was mounted, the error status, the last time it was checked, etc). The primary superblock is located at 1024 bytes from the start of the filesystem, and is 1024 bytes in size. There are backup copies of the superblock stored in block groups[1] with numbers which are integer powers of 3, 5, and 7 (i.e. 1, 3, 5, 7, 9, 25, 27, 49, ...). Under normal operation, only the primary copy of the superblock is ever used, and the backups are only needed by `e2fsck` in case the primary copy is corrupted or overwritten.

The group descriptor table contains one or more blocks which hold *group descriptors*. A group descriptor contains the location of the block bitmap, inode bitmap, and the start of the inode table for its block group. It also contains the count of free blocks, free inodes, and allocated directories for its group. There is a group descriptor for each group in the filesystem, so

[1] When the ext2 on-disk layout was first developed, backup copies of the superblock were placed in *every* block group. For large filesystems, the amount of space consumed by the backup superblocks and group descriptor tables became too large, so modern ext2 filesystems only place backups in *sparse* groups (i.e. as described here).

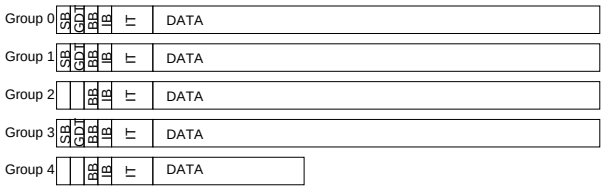

Figure 2: Several block groups make up filesystem

the number of blocks that make up the group descriptor depends on the number of groups in the filesystem, which in turn depends on the size of the filesystem. Because the group descriptor table is critical in locating the filesystem metadata, backup copies of the group descriptor table are placed in the same groups as backup superblocks.

Figure 2 shows a filesystem with several block groups. Note that all but the last group have the same number of blocks. The inode table is the same size in each block group, and each block group has both an inode and block bitmap. The group descriptor table is the same size in each group, if it exists. While this example shows the most common case of the bitmaps and inode table in the same location in each group, the ext2 format allows the bitmaps and inode table each to be in any (non-overlapping) location within the group. The actual location within each group is solely determined by the entries in the group descriptor table for that group. The backup superblock and group descriptor table must be located in the first blocks of the group so that they can be located in case of filesystem corruption. By default `mke2fs` will create the bitmaps and inode table in the same position within each group.

3 Three Resizing Scenarios

3.1 Common Resizing Operations

There are several operations that are common to all of the growth scenarios discussed here:

- Increasing the total number of filesystem blocks in the primary and backup superblocks by the number of blocks added to the filesystem. Since it is possible that the primary superblock may be corrupted at some later date, we also need to update the backup superblocks to reflect the new size of the filesystem. Otherwise, `e2fsck` would believe that all block numbers higher than the old filesystem size are invalid and the data therein would be discarded.
- Increase the count of free filesystem blocks in the primary superblock and group descriptor for that group. The count of free blocks, like other dynamic ext2 metadata, does not necessarily need to be updated in the backup superblocks or group descriptors. The total and per-block-group count of free filesystem blocks can be recovered by counting the bits set in each of the block bitmaps, and is only really a convenience for efficient `statfs()` and `ENOSPC` implementation.
- Increase the number of reserved filesystem blocks proportional to the number of new blocks added to the filesystem.
- In order to notify the filesystem that it should begin its resizing operation, the filesystem is remounted with the option `-o remount,resize=<new size>`. While this seems somewhat awkward, it does have the benefit that it can be done

from the command-line with only the `mount` command. It is also very practical from the point of view that the resize operation uses all of the same checks and setup code in `ext2_setup_super()` as the initial `mount` call.

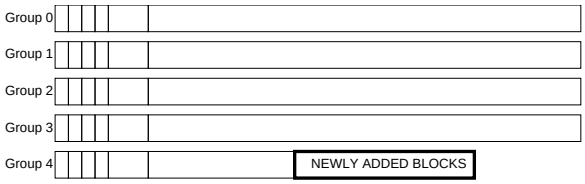

Figure 3: Adding blocks to a single group

Since the goal of online resizing is to allow the filesystem to continue to be used while the resizing operation is being done, we need to make sure that we do the appropriate locking of the filesystem structures that we update. Currently, there is only a single lock for the superblock and all of the group descriptors, so we only need to hold this superblock lock to ensure our operations are safe. Also, the critical filesystem values are each stored in only one place so we don't need to ensure consistency between multiple data fields of different in-memory data structures. Obviously we want to hold the superblock lock for as little time as possible to avoid excluding other processes from doing filesystem operations. It is anticipated that in the future the filesystem locking will become more fine-grained to allow increased parallelism in block and inode allocation. This will likely be done by having a read/write lock for each group's block bitmap, inode bitmap, and group descriptor, separate from the superblock lock, and may be added to the kernel during 2.5 development.

One other aspect of increasing the filesystem size which makes it relatively trouble-free is the fact that all of the blocks which are added to the system are new. This means that there can not be any users of these blocks or pages or buffers mapped to them, so we do not need to be concerned with hashing or locking or other such aspects of block I/O which may cause deadlocks or data corruption. This is one of the major reasons why shrinking a mounted filesystem is completely impractical. In limited cases it might be possible to shrink a mounted filesystem by a single group, the nature of the ext2 block and inode allocation algorithms mean that the last group will almost always have inodes and blocks allocated in them. While it might be possible to relocate data blocks on a mounted filesystem (excluding issues such as `FIBMAP` of those blocks exporting the block numbers to user space), the relocation of inodes is even more problematic because of the use of inode numbers in directories and NFS file handles, let alone the locking issues involved.

3.2 Adding Blocks to a Single Block Group

The first and simplest filesystem growth scenario is adding blocks to the end of a single block group, as shown in Figure 3. In order to efficiently and safely implement the operations of updating the block bitmap and increasing the free blocks count in the superblock and group descriptor, we take a short cut and create a fake inode which spans the newly-added blocks at the end of the last block group. All we have to do now is increase the total number of filesystem blocks in the superblock, and delete the fake inode via `ext2_free_blocks()`. This will take care of updating the bitmaps and the free blocks count in the superblock and group descriptor. The fake node is only created in-memory and only has enough fields filled in to satisfy those accessed by `ext2_free_blocks()` and what it calls. This also has the advantage that any locking changes which take place in the ext2 code will be handled for us.

We can easily do everything we need for this resizing operation from within the kernel, since it has no more impact on performance than deleting a file of the same size (less actually, since we don't need to update the on-disk inode data). Since this resize operation is virtually identical to deleting a file, it is almost impossible for it to fail, excluding bugs in the core ext2 code. This resizing operation needs no additional updates to the on-disk metadata. Depending on the blocksize of the filesystem, this scenario would allow us to grow a filesystem up to the next 8MB, 16MB, or 32MB boundary (for 1kB, 2kB, or 4kB blocks, respectively) for all filesystems.

The only operation which is left to user-space is that of copying the new primary superblock to the backup superblock locations, if any. Since the backup superblocks are never accessed by the kernel, there is no problem writing them directly to the block device from user space. If the resize command is done manually via the `mount` command instead of `ext2online`, the superblock is not copied to the backup locations. Under all but the most extreme failure conditions that will be OK, and the ability to do a filesystem resize using an available tool is convenient. The backup superblocks will be updated the next time `e2fsck` does a full check of the filesystem.

3.3 Adding a Block Group Within a Group Descriptor Block

The second filesystem growth scenario is that of adding a new group to the filesystem, as shown in Figure 4, after the end of a full block group. Each block group must have a block bitmap, inode bitmap, and inode table, and possibly a backup superblock and group descriptor table. This means that we need to add

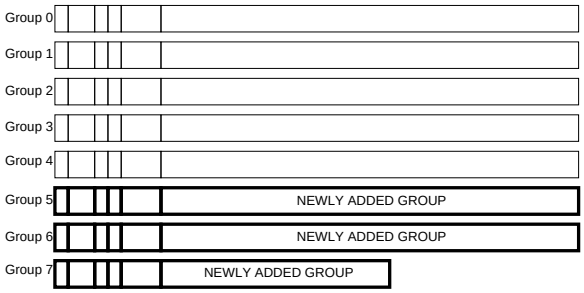

Figure 4: Adding new block groups within the same descriptor block

at least enough new blocks to the filesystem to hold all of this metadata before we can actually increase the size of the filesystem. Since we are adding an inode bitmap and inode table, we also need to update the total number of inodes in the primary and backup superblocks, and the number of free inodes in both the primary superblock and group descriptor for the new group. All of these operations can be done by simply updating the appropriate fields in the superblock while holding the superblock lock.

There are two things that distinguish this case from the first case:

- We need to create a new block bitmap, inode bitmap, and an inode table for each group added to the filesystem. The bitmaps have to be filled to reflect the availability of blocks and inodes within the new group.

- We need to add a new entry to the group descriptor table for the new group. The group descriptor table is stored on disk and is accessed in the kernel via an array of buffer heads. If we are adding a group that fits within the last group descriptor block then we do not need to add a new buffer head, but we do need to increase the number of groups in the filesystem so that the block and inode allocation

routines know to check for free blocks in the new group(s).

In order to minimize kernel code growth and in-kernel processing overhead, the creation of the block bitmap, inode bitmap, and inode table are done from user space before the kernel is told about the new group(s). This is perfectly safe even on a busy filesystem because the blocks being modified from user space are beyond the end of the existing filesystem, so there will not be any other processes reading from or writing to these blocks.

The creation of the new group descriptor entry is slightly more problematic, because it is sharing a block with other group descriptors that are in active use. Reading the block to user space, modifying it, and writing it back to disk cannot be done safely on an active filesystem because it would lose any updates that had been done by the kernel since the group descriptor block had been read. Fortunately, existing kernels have cache coherency between the block device and the buffer heads in the kernel, so it is possible with ext2 to write a new group descriptor entry into the last group descriptor block and have it visible to the kernel without corrupting the existing group descriptors. The new group descriptor entry has the correct values for the free block and inode counts of that group, in addition to the location of the bitmaps and inode table.

The new group descriptors are written to the disk from user space before the resize operation takes place, as are the bitmaps and inode table. The filesystem is notified via the `mount` parameter of the new filesystem size and adds the new blocks and inodes to the free and total block and inode counts in the superblock. As the group's free block and inode counts are added into the totals in the superblock (under the superblock lock, of course), the new groups are immediately available to the filesystem for allocation. Again, once the kernel is finished its resizing operation the new superblock and group descriptor table is copied to the backup locations, if any. Since we need to create the bitmaps and inode table from user space, it is not practical to do this resize operation with anything other than the `ext2online` tool, which also handles the updating of the backup superblock and group descriptor tables once the resize operation has completed successfully.

Resizing the filesystem in this manner allows it to add groups until the last group descriptor block is full. Since each group descriptor is a fixed size (32 bytes), the number of group descriptors that fit into a single filesystem block depends on the blocksize. Likewise the size of each group also depends on the blocksize. The boundaries for group descriptor blocks are 256MB, 1GB, and 16GB for 1kB, 2kB, and 4kB blocks, respectively. This allows one to resize any default 4kB blocksize filesystem a considerable amount without any prior preparation.

The failure scenarios for this resizing case are very minimal. The in-kernel code basically is just adding the new block and inode counts from the group descriptors into the superblock. Although the group descriptors, bitmaps, and inode table were just written from user space, the kernel does some validity checks of everything before activating each group to avoid any problems.

3.4 Adding a Block Group in a New Group Descriptor Block

The final resizing scenario happens when the last block in the group descriptor table is full. This brings up one of the major limitations im-

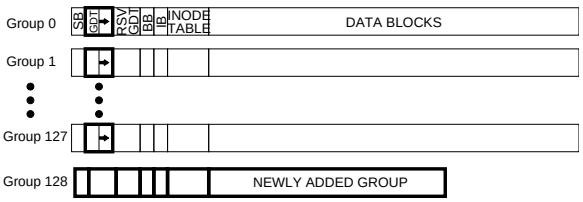

Figure 5: Adding a new block group in a new group descriptor block

posed by keeping on-disk compatibility with the existing ext2 format. Because the number of groups in the filesystem is linearly dependent on the filesystem size, and we keep all group descriptors in each group descriptor table, eventually we need to allocate new blocks for the group descriptor table in each group that has a copy. As Figure 1 shows, the default configuration of metadata within a group is to have the superblock and group descriptor table first (these two *have* to be first in the group), with the block bitmap, inode bitmap, and inode table following immediately afterward. This means that the group descriptor table cannot normally allocate a new block at the end because the block bitmap is using this block.

Fortunately, the design of the ext2 on-disk layout allows us to circumvent this problem by moving the bitmaps and inode table further from the start of the group to allow the group descriptor table to grow, as shown in Figure 5. Since the location of the bitmaps and inode table for each group can be set in the group descriptor, it is simply a matter of creating a filesystem with the bitmaps and inode table offset from the end of the group descriptor table. For existing filesystems, the `ext2resize` package has an `ext2prepare` tool which will relocate the bitmaps and inode table. Since the operation of relocating the bitmaps and inode table is also necessary for resizing the filesystem (for the same reason - because the group descriptor table will grow to need the blocks they occupy) the `ext2prepare` code can re-use most of the same code as `ext2resize`. The moving of the bitmaps and inode table must be done while the filesystem is unmounted.

The second issue for this case is how to reserve the blocks after the end of the group descriptor table so that they are not allocated to regular files. One way would have been to store an extra field in the superblock which holds the number of reserved blocks. If the blocks were set unused in the block bitmap, kernels and `e2fsck` that do not know about this scheme would happily assign those blocks to files and potentially prevent the filesystem from being resized later. If the blocks were set as used in the block bitmap, then an `e2fsck` which didn't know about this scheme would think they were unused and clear them the next time it checked the filesystem. Instead, one of the reserved ext2 inodes (#7) was used to hold these reserved group descriptor blocks. To both the kernel and `e2fsck` inode #7 is using the reserved group descriptor blocks and everyone is happy.

In order to do an online resize when we need to add a new group descriptor block, we perform all of the steps as before:

- Write a group descriptor, block bitmap, inode bitmap, and inode table for each group to the disk from user space.
- Tell the filesystem to grow to the new size.
- The kernel adds the new inodes and blocks from each group to their respective total and free counts in the superblock.
- Update the backup superblocks and group descriptors from user space once the resize has completed successfully.

There are two additional steps which must be

done in order to handle the new group descriptor block:

- The kernel must read the new block into a buffer and add it into the array of group descriptor block buffer heads before adding these groups to the filesystem. In order to minimize kernel code growth, the existing code to allocate the array and read the buffers was extracted into a function which could start reading the group descriptor table at a non-zero offset. The existing buffers are left untouched in order to avoid having other parts of the filesystem code with pointers to no-longer-existing buffer data. Only the group descriptor array is re-allocated at the new size and filled in with pointers to the existing and new buffers. A failure during this part of the resizing information will simply cause the resize not to happen, as the old array is not freed until after the new array and buffer(s) have been allocated.

- We must transfer the blocks from the reserved inode to the group descriptor table. This is actually relatively simple. Because the blocks are already marked as in-use in the block bitmap, we do not need to change it for blocks assigned to the group descriptor table. The number of blocks in the group descriptor table is calculated from the total number of blocks in the filesystem, so we do not need to update the superblock to reflect this (we have already updated the total number of blocks). The only thing that remains, for consistency, is to deallocate the blocks from the reserved inode. This can be done from user-space by writing directly to the block device after the resize is complete, because this reserved inode is not accessible from the mounted filesystem. A failure at this point will mean that `e2fsck` will complain about blocks shared between the reserved inode and the group descriptor table, and will automatically clean it up.

3.5 Incompatible Online Resizing

Adding the reserved group descriptor blocks to the reserved inode is done by `ext2prepare` while the filesystem is unmounted as it moves the bitmaps and inode table out of the way. The number of blocks to be reserved for each group descriptor table copy is calculated from the desired future maximum filesystem size given by the user on the command line. For the default 4kB block size, we only need to reserve 1 block for each 16GB of future growth, so the overhead of reserving enough blocks for a 2TB filesystem is fairly minimal —only 512kB per group descriptor copy. The requirement for users to prepare a filesystem while it is unmounted, before doing large online resizes is a fairly minimal price to pay in order to keep 100% forward and backward compatibility with the existing ext2 filesystem layout. This requirement could be removed by having `mke2fs` create new filesystems that already have these blocks allocated to the reserved inode.

The alternative to doing offline filesystem preparation is to make an incompatible change to the on-disk layout when we run out of space in the last group descriptor block. This change would involve storing new group descriptor blocks at the beginning of the first group that needs a new block for its group descriptor. The backup of this block would be stored in the second group that needs this group descriptor block. This has the added benefit that the group descriptor of a group is relatively closer to the groups that it describes, which may reduce seeking some small amount. The major drawback of this scheme is that it results in a filesys-

tem that can not be mounted by older kernels, nor can older ext2 filesystem tools work with it. It also adds some small amount of additional code to the kernel to deal with the two different layouts of the group descriptor table, although this is fairly minimal. There is also a proposed filesystem change to allow larger contiguous extents on the disk to be allocated which would also benefit from this format change, so there may be additional justification for making such an incompatible change.

Given that people use online filesystem resizing when they are running out of space, they may choose to pay this penalty in order to keep their system running smoothly. The potential drawbacks are only a problem if you have back-level filesystem tools or need to boot an older kernel. This problem could be mitigated by adding functionality to `ext2resize` or `ext2prepare` to remove the incompatibility from unmounted filesystems after the fact. This would be done by moving the bitmaps and inode table out of the way and moving the new group descriptor block(s) to their normal position, in a manner very similar to resizing the filesystem.

4 Online Resizing an ext3 Filesystem

The ext3 filesystem is the journaling version of ext2[ext3]. Interestingly, although ext2 and ext3 share a virtually identical on-disk format (the ext3 journal is simply a regular file stored inside the filesystem), the requirement for a 100% consistent filesystem in the face of a crash at any point during the resize made the ext3 online resize implementation totally different from that of ext2. Writing into the filesystem from user space for ext2 online resizing is not safe to do with an ext3 filesystem because the journal layer may make copies of a buffer while it is being written to disk, so there is no guarantee of cache coherency between user space access to the block device and kernel space[2]. The requirement that the resize operation be atomic and leave the filesystem correct also precludes writing directly to the block device, because these writes will not be in the journal.

Instead, all of the operations which were formerly done inside journaled transactions[3] in the kernel. The user space code is still responsible for determining the locations of the metadata structures within each group, but the kernel code is responsible for creating the actual data. This turns out to be less complex than initially thought, as most of the operations are simply to `memset()` the buffers to zero (for the inode table and group descriptor and most of the bitmaps), then use the kernel bitmap handling routines to mark appropriate bits set, and finally `memset()` any large parts of the bitmap to 0xffffffff as necessary. Since we are creating the data in the kernel, we do not need to verify its correctness, excluding a limited number of parameters passed from user space.

The addition of each group is placed into its own journal operation to avoid trying to create too large of a single transaction. Transactions which do not need to be atomic because they do not affect filesystem recovery such as zero-

[2] In fact, the ext3 journaling layer has a large number of assertions embedded in it which will catch buffers which enter the ext3 “food chain” incorrectly, to avoid data corruption. This prevents such access as a rule, rather than letting it succeed most of the time but fail mysteriously at other times.

[3] Actually, the ext3 journal layer has a larger operation called a *transaction*. What is referred to here as a transaction is actually called a journal *handle* in the code, but transaction more clearly represents the operation being described and the actual functionality is the same.

ing the blocks in the inode table and copying data to the backup superblock and group descriptor tables are put into separate transactions to allow other filesystem operations to happen concurrently. Those operations which need to be atomic, such as moving blocks from the reserved inode to the group descriptor table and updating the superblock to include new groups, are done within a single transaction. Because the journal imposes ordering between transactions, it is enough that a previous transaction was closed and a new one opened to ensure that the previous operations will exist after a crash if any following operation also exists.

In order to keep in-kernel processing to a minimum, the layout of the reserved group descriptor inode was changed from that used with ext2. Under ext2 the updating of the reserved inode was done in user space and searching the inode for blocks that had been added to the group descriptor table was acceptable. With ext3 the layout was changed to allow group descriptor blocks to be put into use by updating only a single inside the transaction, and without searching the inode. This required a change to the user space `ext2prepare` tool to set up the new format. I took advantage of this format change to also add an ext2 compatible feature flag to the ext2 superblock, so that `e2fsck` could properly detect and verify the more rigid layout requirements for the reserved inode.

Figure 6 shows the layout of the reserved GDT blocks in the inode. The reserved blocks are arranged such that the double indirect inode block points to the primary copy of each reserved group descriptor block, which is an indirect inode block. The reserved group descriptor blocks are shown numbered starting with `X`, which is the block number for the block immediately following the last group descriptor block currently in use. The offset of the primary reserved group descriptor block is `X mod (blocksize/4)`. which allows us

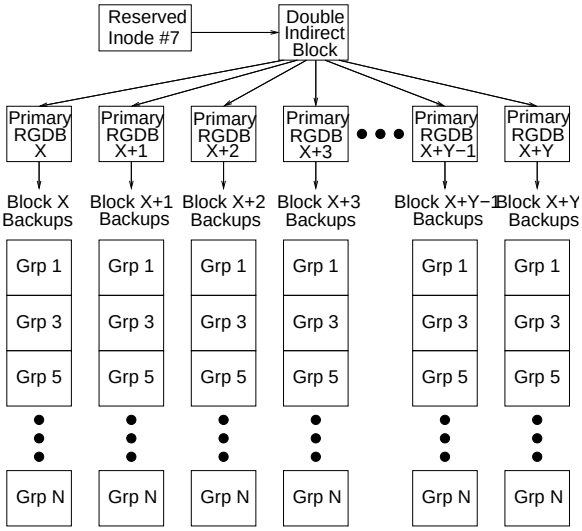

Figure 6: New arrangement of blocks in the reserved inode

to locate each reserved group descriptor block and its backups without any searching.

All of the backup copies for this group descriptor block are leaf blocks attached to the indirect (primary) block. There is one backup group descriptor block for each of the "sparse" groups which have a backup superblock and group descriptor table. Using this layout allows us to transfer the reserved descriptor blocks from the reserved inode to the group descriptor table by simply zeroing out the indirect block pointer in the double indirect block. This is done within the same journal transaction as creating the group descriptor data in the primary descriptor block and adding that group to the filesystem, so if there is any error during this process the filesystem remains consistent.

There is one additional point which needs to be handled by the ext3 resizing code which cannot exist in ext2. In the case where a resize operation was completed in the journal, but the system crashes before the resize is flushed from the journal to the filesystem, the filesystem size in the superblock will still reflect the old filesystem size. Journal recovery is normally

done by e2fsck, but for the root filesystem or other ext3 filesystems mounted without an fsck, journal recovery is done after the superblock has been read from the disk. We detect this case by comparing the filesystem size from before journal recovery to that after journal recovery, and if the filesystem has grown we need to read any additional group descriptor blocks from disk in the same manner we would do in a normal resizing operation.

5 Conclusion

The ext2 filesystem resizing code is a fairly mature piece of code, even thought it has not been included in the stock kernel yet. While the thought of resizing the filesystem while it is mounted and in use is somewhat daunting, the actual simplicity of the resizing operations gives little room for error. In fact, other than minor math errors in updating the group or superblock inode or block counts during development (which are easily detected and/or fixed by both the kernel and e2fsck) , I have never had a reported filesystem corruption from the online resizing.

The advent of online resizing for ext3 does add additional complexity to the kernel code, but the requirements for high-availability systems demand both online resizing and journaling support, so the extra overhead can be justified. Since the online resizing code is only used very rarely, it would be possible to put the bulk of this code into a separate module that is only loaded when a resize operation is done. The cleaner layout of the reserved inode and the e2fsck support for it mean that it is desirable to change the ext2 online resizing support over to use the new inode format also. In the short term this is accomplished by an updated ext2online user tool.

As was previously mentioned, the GNU ext2resize package is available from Sourceforge at http://sf.net/projects/ext2resize/ in .tgz and .rpm formats. There is also a very low volume mailing list dedicated which can be accessed from this same page. General ext2 and ext3 filesystem design and coding discussions take place on the ext2-devel@lists.sourceforge.net mailing list.

Special thanks go to Ted Ts'o and Stephen Tweedie, who helped me understand ext2 and the kernel when I was first trying to learn what kernel programming was all about, and all the interesting ext2/ext3 discussions we've had since then. Lennert Buytenhek was the original author of the GNU ext2resize code and libext2resize, and this gave me the foundation on which to build the user tools for online resizing and offline filesystem preparation.

Thanks also go to Miguel de Icaza, whose ext2-volume patch[volume] gave me a rough idea of where I should look at in the kernel to find the ext2 filesystem code and how it all fit together. The ext2-volume code was written to allow one to concatenate full ext2 filesystems together to form a single filesystem, prior to the availability of MD RAID and LVM in the kernel, but had the major drawbacks that it only supported offline resizing, and produced a totally incompatible filesystem.

References

[resize] Lennert Buytenhek, Andreas Dilger, *GNU ext2resize*, http://sf.net/projects/ext2resize/

[ext2] Rémy Card, Theodore Ts'o, Stephen

Tweedie, *Design and Implementation of the Second Extended Filesystem*, `http://e2fsprogs.sf.net/ext2intro.html`, (1994)

[LVM] Michael Hasenstein, *The Logical Volume Manager*, `http://www.sistina.com/lvm/lvm_whitepaper.pdf`, (2000)

[EVMS] Ben Rafanello, John Stiles, Cuong H. Tran, *Emulating Multiple Logical Volume Management Systems within a Single Volume Management Architecture*, `http://oss.software.ibm.com/developerworks/opensource/evms/doc/EVMS_White_Paper1.ps.gz`, (2000)

[ext3] Stephen Tweedie, *Journaling the ext2fs Filesystem*, LinuxExpo ’98 Proceedings, `ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/journal-design.ps.gz`, (1998)

[volume] Miguel de Icaza, *Ext2fs volume support*, `http://www-mddsp.enel.ucalgary.ca/People/adilger/online-ext2/ext2-volume-0.1.tar.gz`, (1997)