ext4 online defragmentation

Takashi Sato
NEC Software Tohoku, Ltd.
sho@tnes.nec.co.jp

Abstract

ext4 greatly extends the filesystem size to 1024PB compared to 16TB in ext3, and it is capable of storing many huge files. Previous study has shown that fragmentation can cause performance degradation for such large filesystems, and it is important to allocate blocks as contiguous as possible to improve I/O performance.

In this paper, the design and implementation of an online defragmentation extension for ext4 is proposed. It is designed to solve three types of fragmentation, namely single file fragmentation, relevant file fragmentation, and free space fragmentation. This paper reports its design, implementation, and performance measurement results.

1 Introduction

When a filesystem has been used for a long time, disk blocks used to store file data are separated into discontiguous areas (fragments). This can be caused by concurrent writes from multiple processes or by the kernel not being able to find a contiguous area of free blocks large enough to store the data.

If a file is broken up into many fragments, file read performance can suffer greatly, due to the large number of disk seeks required to read the data. Ard Biesheuvel has shown the effect of fragmentation on file system performance [1]. In his paper, file read performance decreases as the amount of free disk space becomes small because of fragmentation. This occurs regardless of the filesystem type being used.

ext3 is currently used as the standard filesystem in Linux, and ext4 is under development within the Linux community as the next generation filesystem. ext4 greatly extends the filesystem size to 1024PB compared to 16TB in ext3, and it is capable of storing many huge files. Therefore it is important to allocate blocks as contiguous as possible to improve the I/O performance.

In this paper, an online defragmentation feature for ext4 is proposed, which can solve the fragmentation problem on a mounted filesystem.

Section 2 describes various features in current Linux filesystems to reduce fragmentation, and shows how much fragmentation occurs when multiple processes writes simultaneously to disk. Section 3 explains the design of the proposed online defragmentation method and Section 4 describes its implementation. Section 5 shows the performance benchmark results. Existing work on defragmentation is shown in Section 6. Section 7 contains the summary and future work.

2 Fragmentations in Filesystem

Filesystem on Linux have the following features to reduce occurrence of fragments when writing a file.

- Delayed allocation

  The decision of where to allocate the block on the disk is delayed until just before when the data is stored to disk in order to allocate blocks as contiguously as possible (Figure 1). This feature is already implemented in XFS and is under discussion for ext4.

- Block reservation

  Contiguous blocks are reserved right after the last allocated block, in order to use them for successive block allocation as shown in Figure 2. This feature is already implemented in ext4.

Although ext4 and XFS already have these features implemented to reduce fragmentation, writing multiple files in parallel still causes fragmentation and decrease in file read performance. Figure 3 shows the difference in file read performance between the following two cases:

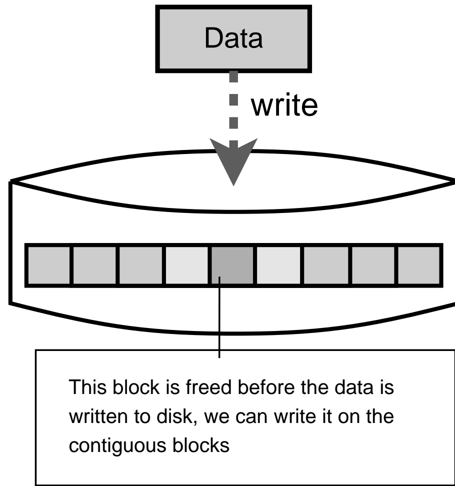


Figure 1: Delayed allocation

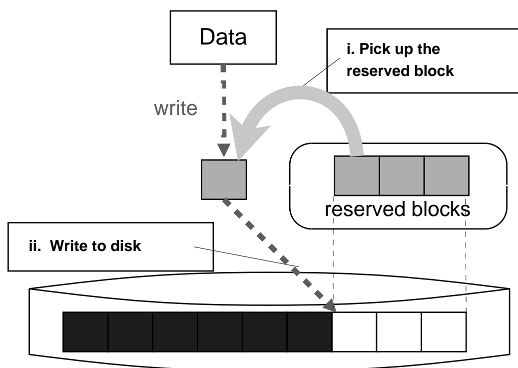


Figure 2: Block reservation

- Create the file by writing 32 1GB files sequentially
- Create the file by writing 32 1GB files from 32 threads concurrently

File read performance decreases about 15% for ext3, and 16.5% for XFS.

Currently ext4 does not have any defragmentation feature, so fragmentation will not be resolved until the file is removed. Online defragmentation for ext4 is necessary to solve this problem.

3 Design of Online Defragmentation

3.1 Types of Fragmentation

Fragmentation could be classified into the following three types [2].

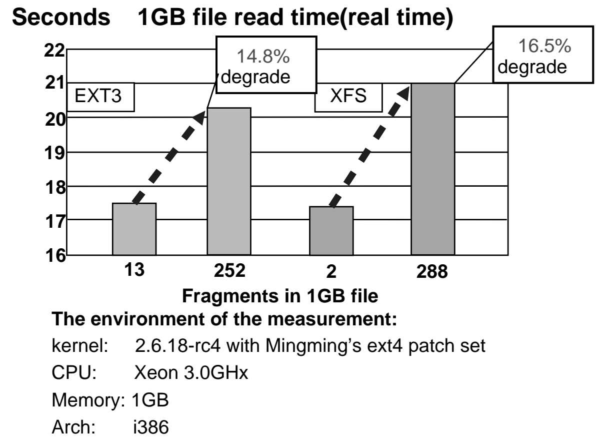


Figure 3: The influence of fragments

- Single file fragmentation
  Single file fragmentation occurs when a single file is broken into multiple pieces. This decreases the performance of accessing a single file.
- Relevant file fragmentation
  Relevant file fragmentation occurs when relevant files, which are often accessed together by applications are allocated separately on the filesystem. This decreases the performance of applications which access many small files.
- Free space fragmentation
  Free space fragmentation occurs when the filesystem has many small free areas and there is no large free area consisting of contiguous blocks. This will make the other two types of fragmentation more likely to occur.

Online defragmentation should be able to solve these three types of fragmentation.

3.2 Single File Fragmentation

Single file fragmentation could be solved by moving file data to contiguous free blocks as shown in Figure 4.

Defragmentation for a single file is done in the following steps (Figure 5).

1. Create a temporary inode.
2. Allocate contiguous blocks to temporary file as in Figure 5(a).

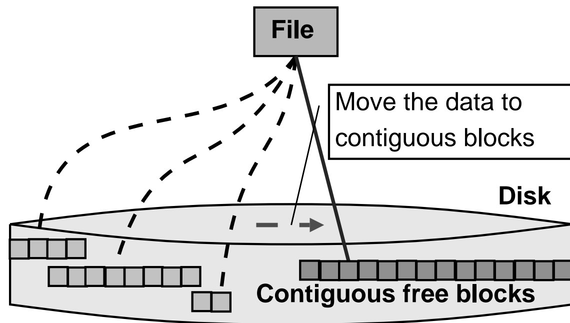


Figure 4: Defragment for a single file

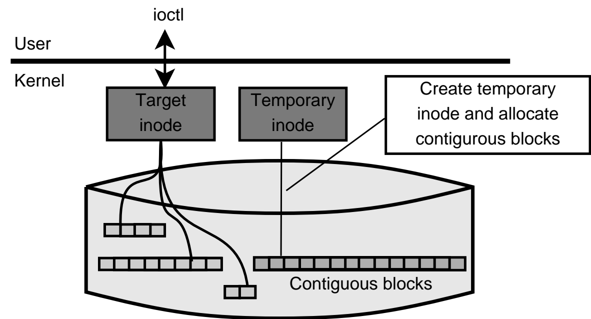
3. Move file data for each page as described in Figure 5(b). The following sequence of events should be committed to the journal atomically.
   (a) Read data from original file to memory page.
   (b) Swap the blocks between the original inode and the temporary inode.
   (c) Write data in memory page to new block.

3.3 Relevant File Fragmentation

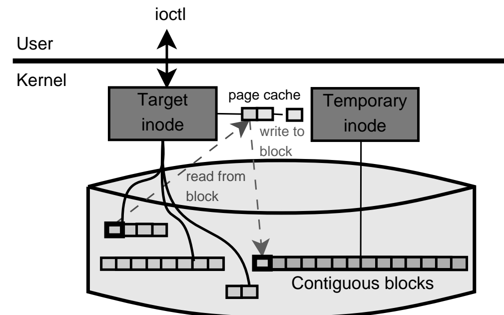
Relevant file fragmentation could be solved by moving the files under the specified directory close together with the block containing the directory data as presented in Figure 6.

Defragmentation for relevant files is done in the following steps (Figure 7).

1. The defragmentation program asks the kernel for the physical block number of the first block of the specified directory. The kernel retrieves the physical block number of the extent which is pointed by the inode of the directory and returns that value as in Figure 7(a).
2. For each file under the directory, create a temporary inode.
3. The defragmentation command passes the physical block number obtained in step 1 to the kernel. The kernel searches for the nearest free block after the given block number, and allocates that block to the temporary inode as described in Figure 7(b).
4. Move file data for each page, by using the same method as for single file defragmentation.



(a) Allocate contiguous blocks



(b) Replace data blocks

Figure 5: Resolving single file fragmentation

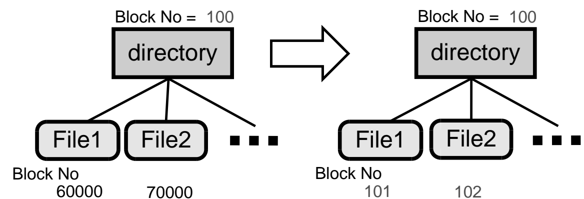


Figure 6: Defragment for the relevant files

5. Repeat steps 2 to 4 for all files under the specified directory.

3.4 Free Space Fragmentation

If the filesystem has insufficient contiguous free blocks, the other files are moved to make sufficient space to allocate contiguous blocks for the target file. Free space defragmentation is done in the following steps (Figure 8).

1. Find out the block group number to which the target file belongs. This could be calculated by using the number of inodes in a group and the inode number as below.

$$groupNumber = \frac{inodeNumber}{iNodesPerGroup}$$

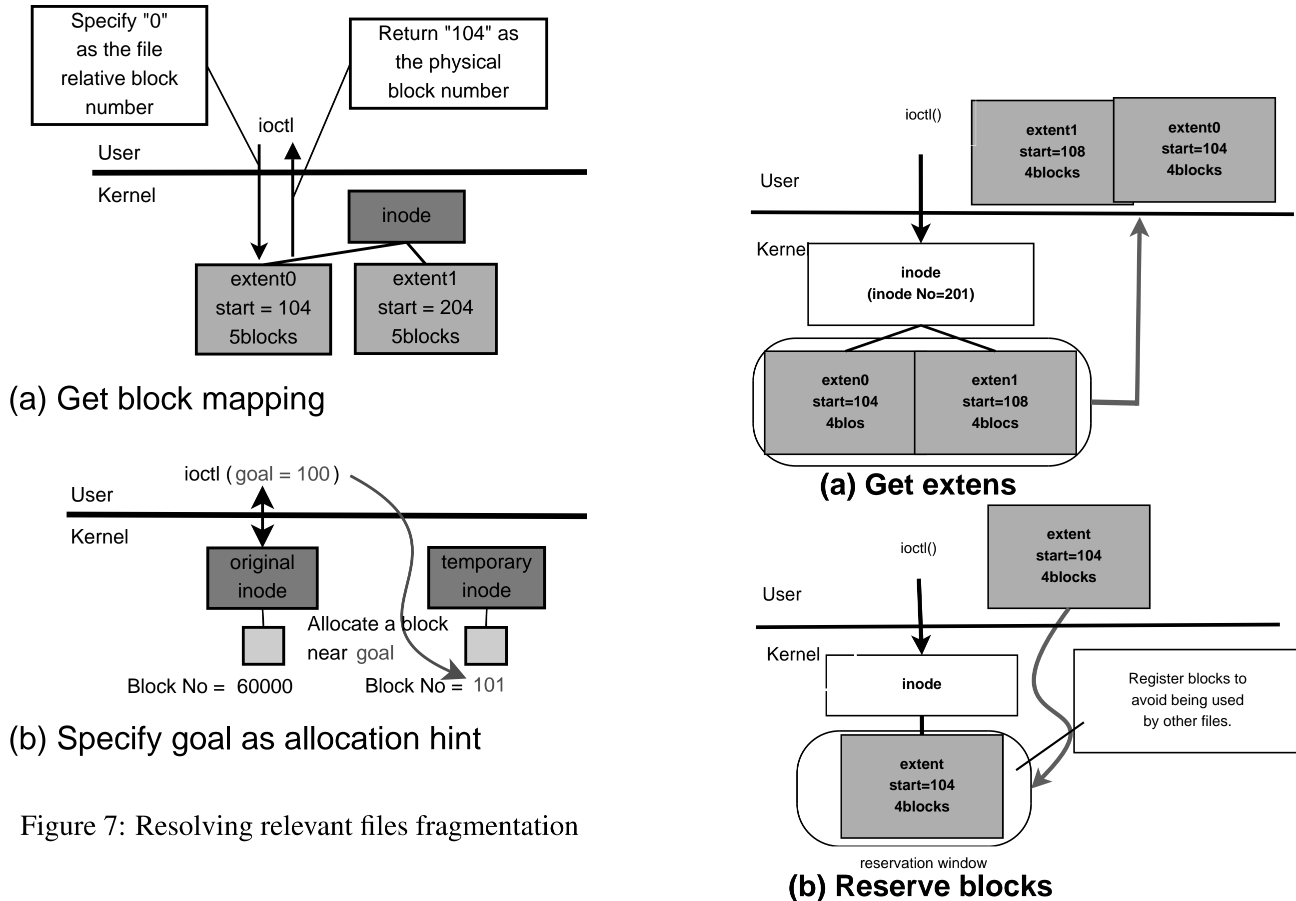


Figure 7: Resolving relevant files fragmentation

2. Get the extent information of all the files belonging to the target group and choose the combination of files to evict for allocating contiguous blocks large enough to hold the target file as in Figure 8(a).

3. Reserve the chosen blocks so that it will not be used by other processes. This is achieved by the kernel registering the specified blocks to the reservation window for each file as presented in Figure 8(b).

4. Move the data in the files chosen in step 2 to other block groups. The destination block group could be either specified by the defragmentation command or use the block group which is farthest away from the target group as in Figure 8(c). The file data is moved using the same method as for single file defragmentation.

5. Move the data in the target file to the blocks which have been just freed. The kernel allocates the freed blocks to the temporary inode and moves the file data using the same method as for single file defragmentation as in Figure 8(d).

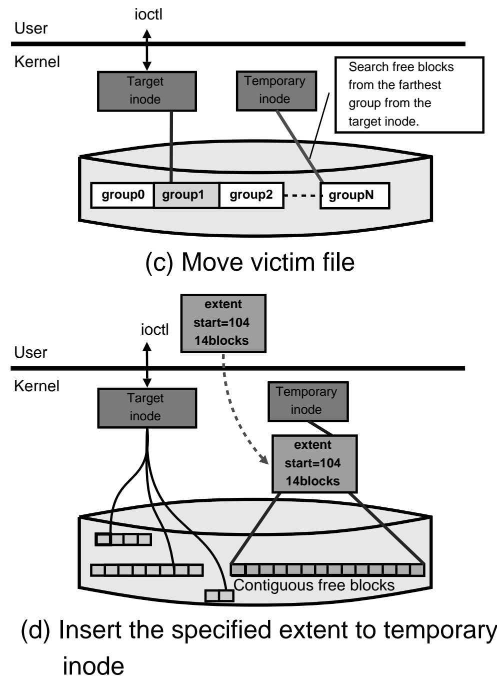


Figure 8: Resolving free space fragmentation

4 Implementation

The following five ioctls provide the kernel function for online defragmentation. These ioctls use Alex Tomas's patch [3] to implement multi-block allocation feature to search and allocate contiguous free blocks. Alex's multi-block allocation and the proposed five ioctls are explained below.

The ioctls described in Section 4.3 to 4.6 are still under development, and have not been tested. Also, moving file data for each page is currently not registered to the journal atomically. This will be fixed by registering the sequence of procedures for replacing blocks to the same transaction.

4.1 Multi-block allocation

Alex's multi-block allocation patch [3] introduces a bitmap called "buddy bitmap" to manage the contiguous blocks for each group in an inode which is pointed from the on-memory superblock (Figure 9).

The buddy bitmap is divided into areas to keep bitmap of contiguous free blocks with length of powers of two, e.g. bitmap for 2 contiguous free blocks, bitmap for 4 contiguous free blocks, etc. The kernel can quickly find contiguous free blocks of the desired size by using the buddy bitmap. For example, when the kernel requests 14 contiguous free blocks, it searches in the area for 8, 4, and 2 contiguous free blocks and allocates 14 contiguous free blocks on disk.

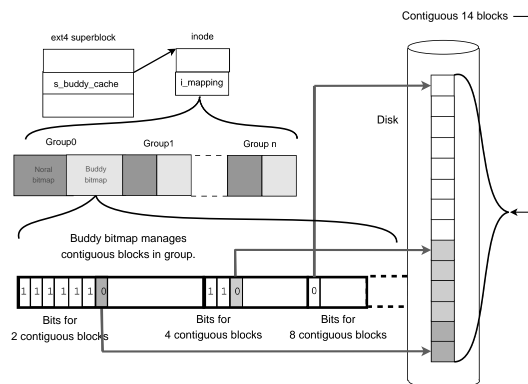


Figure 9: Multi-block allocation

4.2 Moving Target File Data (EXT4_IOC_DEFRAG)

This ioctl moves file data from fragmented blocks to the destination. It uses the structures `ext4_ext_defrag_data` and `ext4_extent_data` shown in Figure 10 for input parameters. The behavior of this ioctl differs depending on the type of defragmentation it is used for.

- Single file fragmentation
  Both the start offset of defragmentation (*start_offset*) and the target size (*defrag_size*) need to be specified. When both *goal* and *ext.len* are set to 0, the kernel searches for contiguous free blocks starting from the first block of the block group which the target file's inode belongs to, and replaces the target file's block with the free blocks.

- Relevant file fragmentation
  In addition to *start_offset* and *defrag_size*, *goal* should be set to the physical block number of the first block in the specified directory. When *goal* is set to a positive number, the kernel searches for free blocks starting from the specified block, and replaces the target file's blocks with the nearest contiguous free blocks.

- Free space fragmentation
  In addition to *start_offset* and *defrag_size*, *ext* should be set to the information of the extent which represents the new contiguous area for replacement. When *ext.len* is set to a positive number, the kernel replaces the target file's blocks with the blocks in *ext*

Since I am still designing the implementation of the following four ioctls, I haven't tested them yet.

4.3 Get Group Information (EXT4_IOC_GROUP_INFO)

This ioctl gets the group information. There is no input parameter. The kernel gets the number of blocks in a group (*s_blocks_per_group*) and the number of the inodes (*s_inodes_per_group*) from ext4 memory superblock (*ext4_sb_info*) and returns them with the structure (*ext4_group_data*) in Figure 11.

blocks_per_group is not used in the current implementation, but it is returned in case of future use.

4.4 Get Extent Information (EXT4_IOC_GET_EXTENTS_INO)

This ioctl gets the extent information. The structure shown in Figure 12 is used for both input and output. The command sets the first extent number of the target extent to *entries*. The kernel sets the number of returned extents upon return. Since there might be very large number of extents in a file, the kernel returns extents up to *max_entries* specified as input. If the number of extents is larger than *max_entries*, the command can get all extents by calling this ioctl multiple times with updated *entries*.

4.5 Block Reservation (EXT4_IOC_RESERVE_BLOCK)

This ioctl is used to reserve the blocks, so that it will not be used by other processes. The blocks to reserve is specified in the extent information (*ext4_extent_data*). The kernel reserves the blocks using the existing block reservation function.

4.6 Move Victim File (EXT4_IOC_MOVE_VICTIM)

This ioctl moves a file from the block group where it belongs to other block groups. *ext4_extents_info* structure is used for input parameter. *ino* stores the inode number of the target file, and *entries* holds the number of extents specified in *ext*. *ext* points to the array of extents which specify the areas which should be moved, and *goal* contains the physical block number of the destination.

The kernel searches for a contiguous free area starting from the block specified by *goal*, and replaces the target file's blocks with the nearest contiguous free blocks. If *goal* is 0, it searches from the first block of the block group which is farthest away from the block group which contains the target file's inode.

5 Performance Measurement Results

5.1 Single File Fragmentation

Fifty fragmented 1GB files were created. Read performance was measured before and after defragmentation. Performance measurement result is shown in Table 1. In this case, defragmentation resulted in 25% improvement in file read performance.

	Fragments	I/O performance (Sec)
Before defrag	12175	618.3
After defrag	800	460.6

Table 1: The measurement result of the defragmentation for a single file

5.2 Relevant File Fragmentation

The Linux kernel source code for 2.6.19-rc6 (20,000 files) was extracted on disk. Time required to run find command on the kernel source was measured before and after defragmentation. Performance measurement result is shown in Table 2. In this case, defragmentation resulted in 29% performance improvement.

	I/O performance (Sec)
Before defrag	42.2
After defrag	30.0

Table 2: The measurement result of the defragmentation for relevant files

Defragmentation for free space fragmentation is still under development. Performance will be measured once it has been completed.

6 Related Work

Jan Kara has proposed an online defragmentation enhancement for ext3. In his patch [4], a new ioctl to exchange the blocks in the original file with newly allocated contiguous blocks is proposed. The implementation is in experimental status and still lacks features such as searching contiguous blocks. It neither supports defragmentation for relevant files nor defragmentation for free space fragmentation.

During the discussion in linux-ext4 mailing list, there were many comments that there should be a common interface across all filesystems for features such as free block search and block exchange. And in his mail [5], a special filesystem to access filesystem meta-data was proposed. For instance, the extent information could be accessed by reading the file data/extents. Also the blocks used to store file data could be exchanged to contiguous blocks by writing the inode number of the temporary file which holds the newly allocated contiguous

```
struct ext4_ext_defrag_data {
        // The offset of the starting point (input)
        ext4_fsblk_t start_offset;
        // The target size for the defragmentation (input)
        ext4_fsblk_t defrag_size;
        // The physical block number of the starting
        // point for searching contiguous free blocks (input)
        ext4_fsblk_t goal
        // The extent information of the destination.
        struct ext4_extent_data ext;
}

struct ext4_extent_data {
        // The first logical block number
        ext4_fsblk_t block;
        // The first physical block number
        ext4_fsblk_t start;
        // The number of blocks in this extent
        int          len;
}
```

Figure 10: Structures used for ioctl (EXT4_IOC_DEFRAG)

```
struct ext4_group_data {
        // The number of inodes in a group
        int inodes_per_group
        // The number of blocks in a group
        int blocks_per_group;
}
```

Figure 11: Structures used for ioctl (EXT4_IOC_GROUP_INFO)

```
struct ext4_extents_info {
        //inode number (input)
        unsigned long long ino;
        //The max number of extents which can be held (input)
        int   max_entries;
        //The first extent number of the target extents (input)
        //The number of the returned extents (output)
        int   entries;
        //The array of extents (output)
        //NUM is the same value as max_entries
        struct ext4_extent_data ext[NUM];
}
```

Figure 12: Structures used for ioctl (EXT4_IOC_GET_EXTENTS_INO)

blocks to data/reloc. During the discussion, two demerits were found. One is that common procedures across all filesystems are very few. The other is that there are a lot of overheads for encoding informations in both userspace and the kernel. Therefore this discussion has gone away. This discussion is for unifying interface and is not for the implementation for the online defragmentation which is explained by this paper.

7 Conclusion

Online defragmentation for ext4 has been proposed and implemented. Performance measurement has shown that defragmentation for a single file can improve read performance by 25% on a fragmented 1GB file. For relevant file fragmentation, defragmentation resulted in 29% performance improvement for accessing all file in the Linux source tree.

Online defragmentation is a promising feature to improve performance on a large filesystems such as ext4. I will continue development on the features which have not been completed yet.

There are also work to be done in the following areas:

- Decrease performance penalty on running processes
  Since it is possible for defragmentation to purge data on the page cache which other processes might reference later, defragmentation may decrease performance of running processes. Using fadvice to alleviate the performance penalty may be one idea.
- Automate defragmentation
  To reduce system administrator's effort, it is necessary to automate defragmentation. This can be realized by the following procedure.
  1. The kernel notifies the occurrence of fragments to user space.
  2. The user space daemon catches the notification and executes the defragmentation command.

References

[1] Giel de Nijs, Ard Biesheuvel, Ad Denissen, and Niek Lambert, "The Effects of Filesystem Fragmentation," in *Proceedings of the Linux Symposium*, Ottawa, 2006, Vol. 1, pp. 193–208, `http://www.linuxsymposium.org/2006/linuxsymposium_procv1.pdf`.

[2] "File system fragmentation." `http://en.wikipedia.org/wiki/File_system_fragmentation`.

[3] Alex Tomas. "[RFC] delayed allocation, mballoc, etc." `http://marc.info/?l=linux-ext4&m=116493228301966&w=2`.

[4] Jan Kara. "[RFC] Ext3 online defrag," `http://marc.info/?l=linux-fsdevel&m=116160640814410&w=2`.

[5] Jan Kara. "[RFC] Defragmentation interface," `http://marc.info/?l=linux-ext4&m=116247851712898&w=2`.

Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*