Introduction

Andreas Dilger, Kalpak Shah

18 June 2007

This document covers the user interface and internal implementation of an efficient fragmentation reporting tool for ext3/4. This will include addition of a FIEMAP ioctl to fetch extents and changes to filefrag to use this ioctl. The main objective of this tool is to efficiently and easily allow inspection of the disk layout of one or more files without requiring user access to each of the underlying OST devices. There are three major components to this design - the ldiskfs interface, the Lustre RPC transport for the extent mapping information, and the llite interface.

1 Requirements

The tool should be efficient in its use of RPCs, even for large files. The FIBMAP ioctl is not suitable for use on large files, as this can result in millions or even billions of RPCs to get the mapping information for a single file. It should be possible to get the information about an arbitrary-sized extent in a single RPC, and the kernel component and user tool should efficiently use this information.

The user interface should be simple, and the output should be easily understood - by default the filename(s), a count of extents (for each file), and the optimal number of extents for a file with the given striping parameters. The user interface will be "`filefrag [options] filename{filename ...}`" and will allow retrieving the fragmentation information for one or more files specified on the command-line. The output will be of the form:

```
/path/to/file1: extents=2 optimal=1
/path/to/file2: extents=10 optimal=4
```

2 Functional specification

The FIEMAP ioctl (FIle Extent MAP) is similar to the existing FIBMAP ioctl block device ioctl used for mapping an individual logical block address in a file to a physical block address in the block device. The FIEMAP ioctl will return the logical to physical mapping for the extent that contains the specified logical byte address.

```
struct fiemap_extent {
    __u64 fe_offset;/* offset in bytes for the start of the extent */
    __u64 fe_length;/* length in bytes for the extent */
    __u32 fe_flags; /* returned FIEMAP_EXTENT_* flags for the extent */
    __u32 fe_lun;   /* logical device number for extent(starting at 0)*/
};
struct fiemap {
    __u64 fm_start;         /* logical byte offset (in/out) */
    __u64 fm_length;        /* logical length of map (in/out) */
    __u32 fm_flags;         /* FIEMAP_FLAG_* flags (in/out) */
    __u32 fm_extent_count;  /* extents in fm_extents (in/out) */
    __u64 fm_unused;
    struct fiemap_extent fm_extents[0];
};
```

In the ioctl request, the fiemap struct is initialized with the desired mapping information.

```
fiemap.fm_start = {desired start byte offset, 0 if whole file};
fiemap.fm_length = {length of mapping information in bytes, ~0ULL if whole file}
fiemap.fm_extent_count = {number of fiemap_extents in fm_extents array};
fiemap.fm_flags = {flags from FIEMAP_FLAG_* array, if needed};
ioctl(fd, FIEMAP, &fiemap);
for (i = 0; i < fiemap.fm_extent_count; i++) {
     Process extent fiemap.fm_extents[i];
}
```

The logic for the filefrag would be similar to above. The size of the extent array may be extrapolated from the filesize and/or FIEMAP can be called repeatedly with an increasing start offset for each ioctl. The fm_start for the next ioctl can calculated from the fiemap for the last ioctl.

```
fm_start = fiemap.fm_start + fiemap.fm_length + 1
```

We do this until we find an extent with FIEMAP_EXTENT_LAST flag set. We will also need to re-initialise the fiemap flags, fm_extent_count, fm_end.

The FIEMAP_FLAG_* values are specified below. If FIEMAP_FLAG_NUM_EXTENTS is given then the fm_extents array is not filled, and only fm_extent_count is returned with the total number of extents in the file. Any new flags that introduce and/or require an incompatible behaviour in an application or in the kernel need to be in the range specified by FIEMAP_FLAG_INCOMPAT (e.g. FIEMAP_FLAG_SYNC and FIEMAP_FLAG_NUM_EXTENTS would fall into that range if they were not part of the original specification). This is currently only for future use. If it turns out that

FIEMAP_FLAG_INCOMPAT is not large enough then it is possible to use the last INCOMPAT flag 0x01000000 to incidate that more of the flag range contains incompatible flags.

```
#define FIEMAP_FLAG_SYNC        0x00000001 /* sync file data before map */
#define FIEMAP_FLAG_HSM_READ    0x00000002 /* get data from HSM before map */
#define FIEMAP_FLAG_NUM_EXTENTS 0x00000004 /* return only number of extents */
#define FIEMAP_FLAG_INCOMPAT    0xff000000 /* error for unknown flags in here */
```

The returned data from the FIEMAP ioctl is an array of fiemap_extent elements, one per extent in the file. The first extent will contain the byte specified by fm_start and the last extent will contain the byte specified by fm_start + fm_len, unless there are more than the passed-in fm_extent_count extents in the file, or this is beyond the EOF in which case the last extent will be marked with FIEMAP_EXTENT_LAST. Each extent returned has a set of flags associated with it that provide additional information about the extent. Not all filesystems will support all flags.

FIEMAP_FLAG_NUM_EXTENTS will return only the number of extents used by the file. It will be used by default for filefrag since the specific extent information is not required in many cases.

```
#define FIEMAP_EXTENT_HOLE      0x00000001 /* has no data or space allocation */
#define FIEMAP_EXTENT_UNWRITTEN 0x00000002 /* space allocated, but no data */
#define FIEMAP_EXTENT_UNMAPPED  0x00000004 /* has data but no space allocation */
#define FIEMAP_EXTENT_ERROR     0x00000008 /* mapping error, errno in fe_offset. */
#define FIEMAP_EXTENT_NO_DIRECT 0x00000010 /* cannot access data directly */
#define FIEMAP_EXTENT_LAST      0x00000020 /* last extent in the file */
#define FIEMAP_EXTENT_DELALLOC  0x00000040 /* has data but not yet written */
#define FIEMAP_EXTENT_SECONDARY 0x00000080 /* data in secondary storage */
#define FIEMAP_EXTENT_EOF       0x00000100 /* if fm_start + fm_len is beyond EOF */
#define FIEMAP_EXTENT_UNKNOWN   0x00000200 /* in-use but location is unknown */
```

FIEMAP_EXTENT_NO_DIRECT means data cannot be directly accessed (maybe encrypted, compressed, etc.)

FIEMAP_EXTENT_ERROR and FIEMAP_EXTENT_DELALLOC flags should always be returned with FIEMAP_EXTENT_UNMAPPED also set. So some flags are a superset of other flags. FIEMAP_EXTENT_SECONDARY may optionally include FIEMAP_EXTENT_UNMAPPED.

Inside ext4, this can be implemented for extent-mapped files by calling something similar to the existing ext4_ext_ioctl() for EXT4_IOC_GET_EXTENTS but with a different callback function. Or the ext4_fiemap() function can be called directly from the ioctl code if the latest extents patches do not have ext4_ext_ioctl().

For block-mapped files it is possible to simulate this behaviour by looping through ->bmap. Maybe the user should be notified by setting a flag(EXT4_FLAG_BMAP) in fm_flags that bmap is being used instead of the more efficient FIEMAP. Existing tools like bmap can can fallback to using FIBMAP and lustre tools will only work for extent-based files and will return EOPNOTSUPP.

The basic logic for the lustre part of FIEMAP will be as follows:

```
ioctl -> llite -> lov -> osc 1 -> ost 0 -> obdfilter 0 -> ldiskfs 0
                      -> osc 2 -> ost 1 -> obdfilter 1 -> ldiskfs 1
                      .....       .....     .....          .....
                      -> osc N -> ost N -> obdfilter N -> ldiskfs N
```

The *_get_info methods can be used for the communication between the different components of lustre.

In the lov layer, we find the ost_idx's over which the file is striped. Then we calculate the fm_start and fm_length for each object and do a obd_get_info() for each OST over which the file is striped. Then the information obtained from different OSTs will be put together in one fiemap structure for sending to the calling application.

For Lustre, the OST index (not the stripe index) will be stored into fe_lun, and the extents will be returned with ost-local offset values and in stripe order instead of in file offset order due to the undesirable interleaving of the stripes that would cause many more stripes to be shown than are actually allocated.

3 Use cases

1) Files containing holes including an all-hole file.

2) File having an extent which is not yet allocated.

3) Proper working with fm_start + fm_len beyond EOF.

4) Test proper reporting of preallocated extents.

5) Have non-zero fm_start and non-~0ULL fm_end. This can be tested by having fm_count = 1 and forcing many ioctls.

6) If one or more of the OSTs across which the file is striped is down, then fiemap should return with information of the available OSTs.

7) If there is an error mapping an in-between extent then the later extents should be returned.

4 Logic specification

```
struct fiemap_internal {
    struct fiemap *fiemap_s;
    struct fiemap_extent fm_extent;
    char *cur_ext_ptr;
    unsigned int current_extent;
    int err;
};
/*
 * Callback function called for each extent to gather fiemap
 * information.
 */
int ext4_ext_fiemap_cb(struct inode *inode, struct ext4_ext_path *path,
                       struct ext4_ext_cache *newex,
                       struct fiemap_internal *fiemap_i)
{
    struct fiemap *fiemap_s = fiemap_i->fiemap_s;
    struct fiemap_extent *fm_extent = &fiemap_i->fm_extent;
    int current_extent = fiemap_i->current_extent;
    unsigned long blksize = inode->i_sb->s_blocksize;

    if (fiemap_i->err) {
        fm_extent->fe_offset = fiemap_i->err;
        fm_extent->fe_flags |= FIEMAP_EXTENT_ERROR;
        return EXT_CONTINUE;
    }

    /*
     * We only need to return number of extents.
     */
    if (fiemap_s->fm_flags & FIEMAP_FLAG_NUM_EXTENTS)
        goto count_extents;
    if (current_extent >= fiemap_s->fm_extent_count)
        return EXT_BREAK;
    /*
     * Cleanup old data in fiemap_i->fm_extent.
     */
    memset(fm_extent, 0, sizeof(struct fiemap_extent));
    fm_extent->fe_offset = newex->ec_start * blksize;
    fm_extent->fe_length = newex->ec_len * blksize;

    if (newex->ec_type == EXT4_EXT_CACHE_GAP)
        fm_extent->fe_flags |= FIEMAP_EXTENT_HOLE;
    /*
     * Mark this fiemap_extent as FIEMAP_EXTENT_EOF if it's the end of
     * file
     */
    if ((newex->ec_block + newex->ec_len) * blksize >= inode->i_size)
          fm_extent->fe_flags |= FIEMAP_EXTENT_EOF;
    if (!copy_to_user(fiemap_i->cur_ext_ptr, fm_extent,
                      sizeof(struct fiemap_extent))) {
        fiemap_i->cur_ext_ptr += sizeof(struct fiemap_extent);
    } else {
        fiemap_i->err = -EFAULT;
        return EXT_BREAK;
    }
count_extents:
    fiemap_i->current_extent++;
    /*
     * Stop if we are beyond requested mapping size but return complete
     * last extent
     */
    if ((newex->ec_block + newex->ec_len) * blksize >=
        fiemap_s->fm_length)
        return EXT_BREAK;
    return EXT_CONTINUE;
}
int ext4_fiemap(struct inode *inode, struct file *filp,
                unsigned int cmd, unsigned long arg)
{
    unsigned int extent_count;
    struct fiemap *fiemap_s;
    struct fiemap_internal fiemap_i;
    struct fiemap_extent *last_extent;
    ext4_fsblk_t start_blk;
    size_t num_bytes;
    int err = 0;
    if (!(EXT4_I(inode)->i_flags & EXT4_EXTENTS_FL))
        return -EOPNOTSUPP;
    if (!access_ok(VERIFY_WRITE, arg, sizeof(struct fiemap)))
        return -EFAULT;

    /*
     * Fetch only the extent_count first so we can know the number of
     * bytes we have to get from userspace.
     */
    err = get_user(extent_count,
                   &((struct fiemap __user *)arg)->fm_extent_count);
    if (err)
         return err;
    num_bytes = sizeof(struct fiemap);
```

```
        fiemap_s = (struct fiemap *) kmalloc(num_bytes, GFP_KERNEL);
        if (copy_from_user(fiemap_s, (struct fiemap __user *)arg, num_bytes))
             return -EFAULT;

        if (!access_ok(VERIFY_WRITE, arg, num_bytes +
                       fiemap_s->fm_extent_count *
                       sizeof(struct fiemap_extent)))
            return -EFAULT;
        start_blk = fiemap_s->fm_start >> inode->i_sb->s_blocksize;
        fiemap_i.fiemap_s = fiemap_s;
        fiemap_i.cur_ext_ptr = (char *)(arg + sizeof(struct fiemap));
        fiemap_i.current_extent = 0;
        fiemap_i.err = 0;
        /*
         * Walk the extent tree gathering extent information
         */
        mutex_lock(&EXT4_I(inode)->truncate_mutex);
        err = ext4_ext_walk_space(inode, start_blk , EXT_MAX_BLOCK,
                                  (void *)ext4_ext_fiemap_cb, &fiemap_i);
        mutex_unlock(&EXT4_I(inode)->truncate_mutex);
        if (err)
            return err;
        fiemap_s->fm_extent_count = fiemap_i.current_extent - 1;
        last_extent = &fiemap_i.fm_extent;
        last_extent->fe_flags |= FIEMAP_EXTENT_LAST;
        fiemap_s->fm_length = last_extent->fe_offset +
                                           last_extent->fe_length;
        err = copy_to_user((void *)arg, fiemap_s, num_bytes);

        return err;
    }
```

5 State management

5.1 State invariants

5.2 Scalability & performance

FIEMAP is expected to speedup filefrag by a very large factor by allowing packing of information about GBs of data in a single RPC. To further speed up the performance, we have the EXT4_FLAG_NUM_EXTENT flag with which only the number of extents per OST would be reported without filling the extent information.

5.3 Recovery changes

5.4 Locking changes

5.5 Disk format changes

5.6 Wire format changes

5.7 Protocol changes

FIEMAP will be added to *_get_info methods and swabbing of the fiemap structures would have to be done accordingly.

5.8 API changes

5.9 RPCs order changes