The basics of file systems

Different ways to store an integer

Most significant bytes come first (**big-endian**)

u32 x = 0x1A2B3C4D;

On the disk:

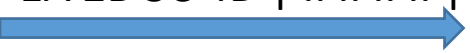
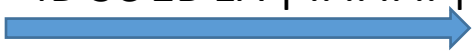
1A 2B 3C 4D | .. .. | ..

byte offsets grow left-to-right

Used by:

- PowerPC
- Itanium

Different ways to store an integer

Most significant bytes come first (**big-endian**)	Least significant bytes come first (**little-endian**)
u32 x = 0x1A2B3C4D;	u32 x = 0x1A2B3C4D;
On the disk: 1A 2B 3C 4D | .. .. | ..  byte offsets grow left-to-right	On the disk: 4D 3C 2B 1A | .. .. | ..  byte offsets grow left-to-right
Used by: • PowerPC • Itanium	Used by: • x86, • ARM.

Different ways to store an integer

Most significant bytes come first (**big-endian**)	Least significant bytes come first (**little-endian**)
u32 x = 0x1A2B3C4D;	u32 x = 0x1A2B3C4D;
On the disk: 1A 2B 3C 4D | .. .. | .. → byte offsets grow left-to-right	On the disk: 4D 3C 2B 1A | .. .. | .. → byte offsets grow left-to-right
Used by: • PowerPC • Itanium	Used by: • x86, • ARM.

Remark: PowerPC, Itanium, ARM, MIPS are bi-endian. They can use both little-endian and big-endian byte orders.

Quiz: what byte order does IP use?

How do we transfer data between little-endian and big-endian systems?

We must choose a byte order for the serialised data. When serialising, we convert the host byte order to the selected one:

```
dmap_ext_t ext = {
        .slice_id = it->last_slice_id, .wr_seq = UINT64_MAX, .item_id = item_id,
        .ext = { .offs = offs < max_ext_len ? 0 : (offs - max_ext_len), .len = 0 }
};
struct dmap_ext_ondisk dsk;
dmap_ext2ondisk(&dsk, &ext);
......

void dmap_ext2ondisk(struct dmap_ext_ondisk *dsk, const dmap_ext_t *ext)
{
        dsk->wr_seq = cpu_to_be64(ext->wr_seq);
        dsk->slice_id = cpu_to_be32(ext->slice_id);
        dsk->item_id = cpu_to_be64(ext->item_id);
        dsk->ext_offs = cpu_to_be64(ext->ext.offs);


        /* pack extent len and deleted bit into 3 bytes */
        u32 len = ext->ext.len;
        dsk->ext_len[0] = (len >> 16) & 0xFF;
        dsk->ext_len[1] = (len >> 8) & 0xFF;
        dsk->ext_len[2] = len & 0xFF;
}
```

When deserialising data, we do the reverse byte order conversion.

Different struct layouts

The declaration of struct dmap_ext_ondisk

```
struct dmap_ext_ondisk {
        be64            item_id;
        be64            ext_offs;
        u8              ext_len[3];
        be64            wr_seq;
        be32            slice_id;
} __attribute__((packed));
```

Different struct layouts

The declaration of struct dmap_ext_ondisk

```
struct dmap_ext_ondisk {
        be64            item_id;
        be64            ext_offs;
        u8              ext_len[3];
        be64            wr_seq;
        be32            slice_id;
} __attribute__((packed));
```

A naïve declaration:

```
struct dmap_ext_ondisk {
        long        item_id;
        long        ext_offs;
        char        ext_len[3];
        long        wr_seq;
        int         slice_id;
}
```

Different struct layouts

The declaration of struct dmap_ext_ondisk

```
struct dmap_ext_ondisk {
        be64            item_id;
        be64            ext_offs;
        u8              ext_len[3];
        be64            wr_seq;
        be32            slice_id;
} __attribute__((packed));
```

A naïve declaration:

```
struct dmap_ext_ondisk {
        long        item_id;
        long        ext_offs;
        char        ext_len[3];
        long        wr_seq;
        int         slice_id;
}
```

How is this struct laid out in the memory on x86_64?

8 bytes	item_id
8 bytes	ext_offs
3 bytes	ext_len
8 bytes	wr_seq
4 bytes	slice_id

Different struct layouts

The declaration of struct dmap_ext_ondisk

```
struct dmap_ext_ondisk {
        be64            item_id;
        be64            ext_offs;
        u8              ext_len[3];
        be64            wr_seq;
        be32            slice_id;
} __attribute__((packed));
```

A naïve declaration:

```
struct dmap_ext_ondisk {
        long        item_id;
        long        ext_offs;
        char        ext_len[3];
        long        wr_seq;
        int         slice_id;
}
```

How is this struct laid out in the memory on x86_64?

8 bytes	item_id	8 bytes	item_id
8 bytes	ext_offs	8 bytes	ext_offs
3 bytes	ext_len	3 bytes	ext_len
8 bytes	wr_seq	5 bytes	padding
4 bytes	slice_id	8 bytes	wr_seq
		4 bytes	slice_id
		4 bytes	padding

Different struct layouts

The declaration of struct dmap_ext_ondisk

```
struct dmap_ext_ondisk {
        be64            item_id;
        be64            ext_offs;
        u8              ext_len[3];
        be64            wr_seq;
        be32            slice_id;
} __attribute__((packed));
```

A naïve declaration:

```
struct dmap_ext_ondisk {
        long        item_id;
        long        ext_offs;
        char        ext_len[3];
        long        wr_seq;
        int         slice_id;
}
```

How is this struct laid out in the memory on x86_64?

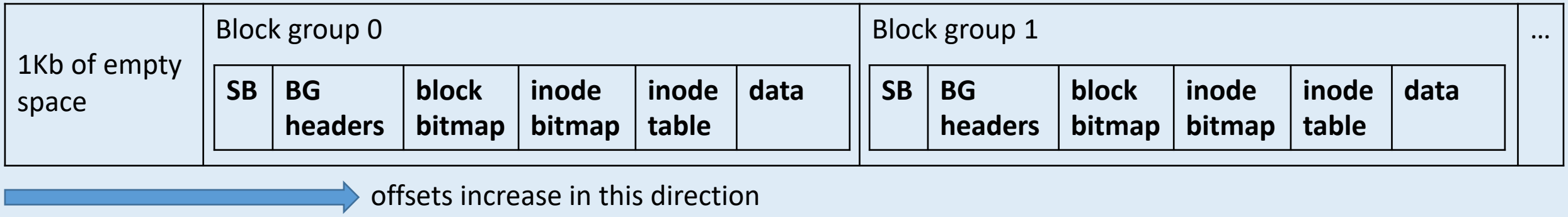
8 bytes	item_id
8 bytes	ext_offs
3 bytes	ext_len
8 bytes	wr_seq
4 bytes	slice_id

And on x86_32?

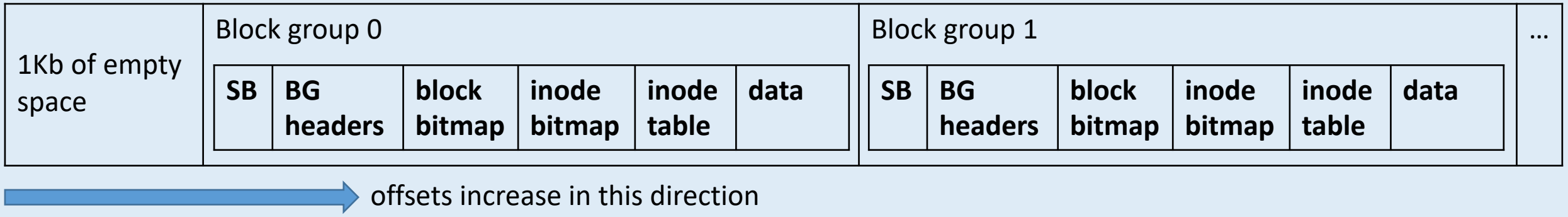
8 bytes	item_id
8 bytes	ext_offs
3 bytes	ext_len
5 bytes	padding
8 bytes	wr_seq
4 bytes	slice_id
4 bytes	padding

4 bytes	item_id
4 bytes	ext_offs
3 bytes	ext_len
1 bytes	padding
4 bytes	wr_seq
4 bytes	slice_id

The structure of ext2

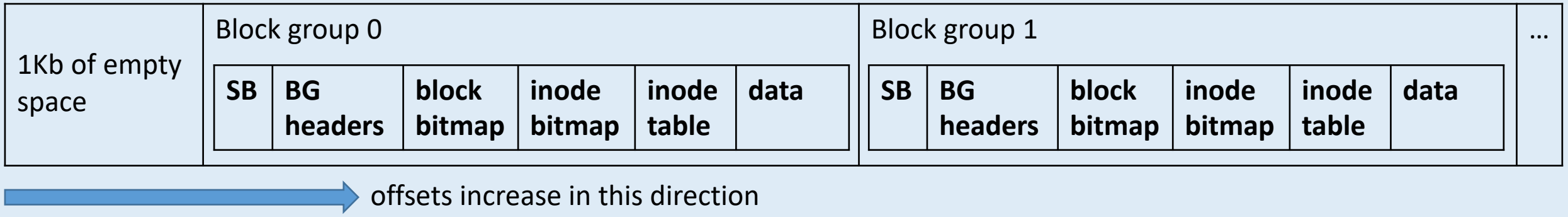


The structure of ext2



The Superblock (SB) contains information about a file system in general: the total size, the size and the number of blocks, etc.

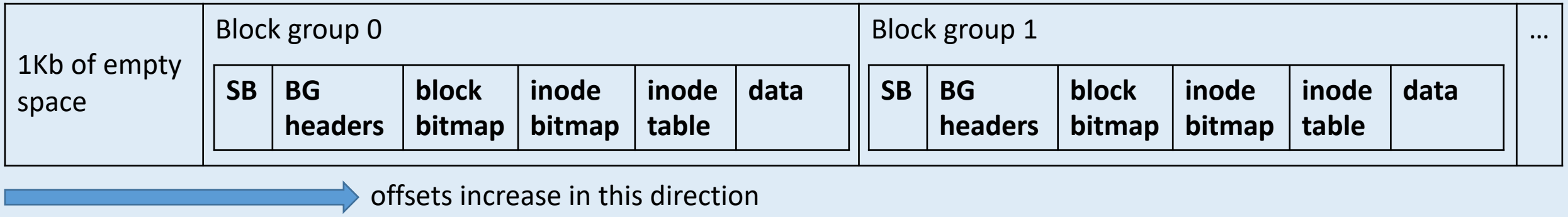
The structure of ext2



The Superblock (SB) contains information about a file system in general: the total size, the size and the number of blocks, etc.

Block Group Headers contain information about individual block groups like the number of free blocks and inodes.

The structure of ext2

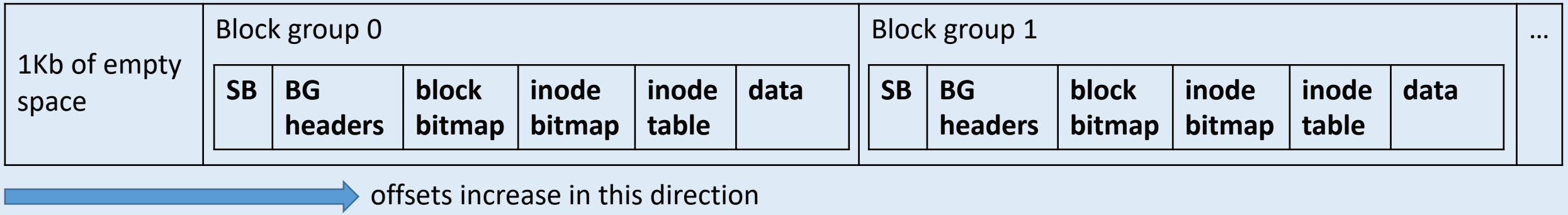


The Superblock (SB) contains information about a file system in general: the total size, the size and the number of blocks, etc.

Block Group Headers contain information about individual block groups like the number of free blocks and inodes.

Remark: splitting a file system into multiple block groups has several advantages. First, this improves the locality. As long as a FS can allocate blocks within one block group, this decreases the seek time. Second, the metadata of a single BG is small enough to fit to RAM. Third, growing such FS is trivial.

The structure of ext2

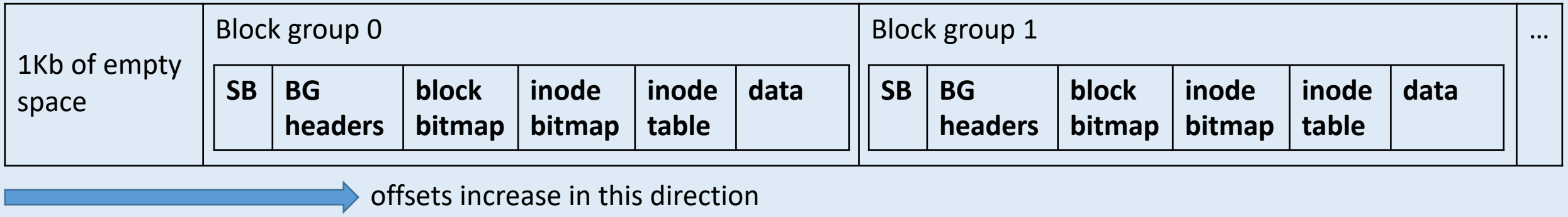


The Superblock (SB) contains information about a file system in general: the total size, the size and the number of blocks, etc.

Block Group Headers contain information about individual block groups like the number of free blocks and inodes.

Block bitmaps are bit arrays that track which blocks are free and which blocks are in use. Ext2 space allocation granularity is 1 block (typically, 4k).

The structure of ext2



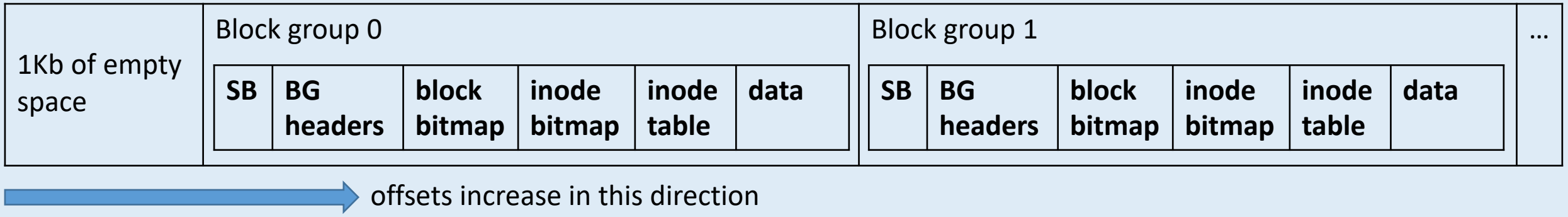
The Superblock (SB) contains information about a file system in general: the total size, the size and the number of blocks, etc.

Block Group Headers contain information about individual block groups like the number of free blocks and inodes.

Block bitmaps are bit arrays that track which blocks are free and which blocks are in use. Ext2 space allocation granularity is 1 block (typically, 4k).

Quiz: why does ext2 track allocated space so coarsely? Every file, even a short one, takes at least 4k on the disk. That seems wasteful.

The structure of ext2



The Superblock (SB) contains information about a file system in general: the total size, the size and the number of blocks, etc.

Block Group Headers contain information about individual block groups like the number of free blocks and inodes.

Block bitmaps are bit arrays that track which blocks are free and which blocks are in use. Ext2 space allocation granularity is 1 block (typically, 4k).

Inode bitmaps are bit array that track which inodes are free and which inodes are in use. An inode (Index Node) is a structure that describes a file in ext2.

Inode table is a disk area that contains all inodes. They are stored as an array of equally-sized structures.

Index nodes (src/linux/fs/ext2/ext2.h)

An ext2 inode stores the properties of a file and lists disk blocks that hold the file's content:

```
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
        __le32  i_osd1;
        __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
        __le32  i_file_acl;     /* File ACL */
        __le32  i_dir_acl;      /* Directory ACL */
        __le32  i_faddr;        /* Fragment address */
        __le8   i_osd2[12];
};
```

Index nodes (src/linux/fs/ext2/ext2.h)

The array ext2_inode->i_block[] holds the list of blocks that comprise the file.

```
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
        __le32  i_osd1;
        __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
        __le32  i_file_acl;     /* File ACL */
        __le32  i_dir_acl;      /* Directory ACL */
        __le32  i_faddr;        /* Fragment address */
        __le8   i_osd2[12];
};
```

Index nodes (src/linux/fs/ext2/ext2.h)

The array ext2_inode->i_block[] holds the list of blocks that comprise the file.
How does ext2 support files that are longer than 15 blocks?

```
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
        __le32  i_osd1;
        __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
        __le32  i_file_acl;     /* File ACL */
        __le32  i_dir_acl;      /* Directory ACL */
        __le32  i_faddr;        /* Fragment address */
        __le8   i_osd2[12];
};
```

Index nodes (src/linux/fs/ext2/ext2.h)

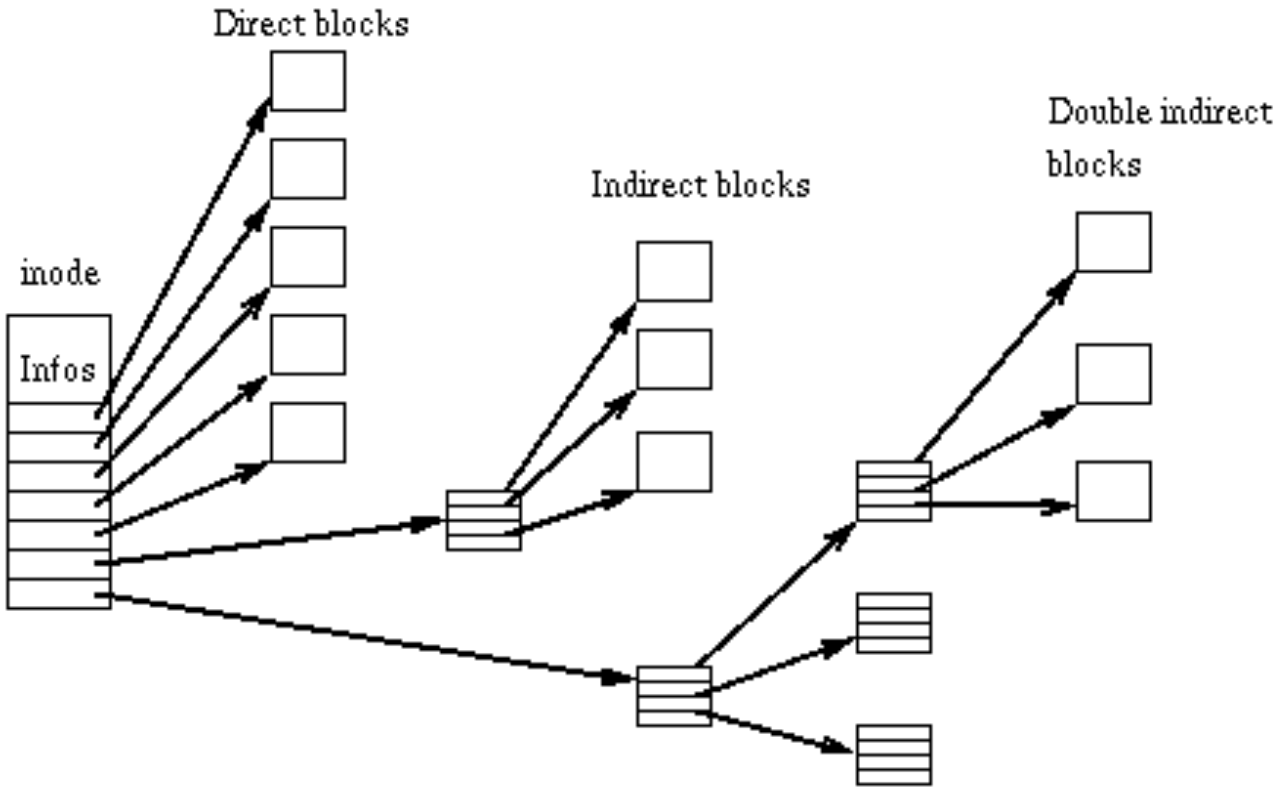
The array ext2_inode->i_block[] holds the list of blocks that comprise the file.
How does ext2 support files that are longer than 15 blocks?

The last 3 entries of i_block[] are **indirect**. They point to blocks that are not file data. They point to block lists.

Index nodes (src/linux/fs/ext2/ext2.h)

The array ext2_inode->i_block[] holds the list of blocks that comprise the file.
How does ext2 support files that are longer than 15 blocks?

The last 3 entries of i_block[] are **indirect**. They point to blocks that are not file data. They point to block lists.
i_blocks[12] is a single-indirect block, i_blocks[13] is double-indirect, and i_blocks[14] is triple-indirect.



Delayed allocation and extent trees

File systems want to store files as contiguous areas on disks. This way accessing a file produces less random IO.

Delayed allocation and extent trees

File systems want to store files as contiguous areas on disks. This way accessing a file produces less random IO.

Recall that writes from a userspace application do not hit the file system immediately. Instead, they are buffered in the page cache.

Delayed allocation and extent trees

File systems want to store files as contiguous areas on disks. This way accessing a file produces less random IO.

Recall that writes from a userspace application do not hit the file system immediately. Instead, they are buffered in the page cache.

This enables **delayed allocation**. When a file system has more file content buffered, it can allocate longer contiguous areas of disk.

Delayed allocation and extent trees

File systems want to store files as contiguous areas on disks. This way accessing a file produces less random IO.

Recall that writes from a userspace application do not hit the file system immediately. Instead, they are buffered in the page cache.

This enables **delayed allocation**. When a file system has more file content buffered, it can allocate longer contiguous areas of disk.

Suppose a file is 48K long and is stored contiguously. What does `i_blocks[]` contain?

```
i_blocks[] = {N, N+1, N+2, N+3, …}
```

This becomes worse yet for longer files.

Delayed allocation and extent trees

File systems want to store files as contiguous areas on disks. This way accessing a file produces less random IO.

Recall that writes from a userspace application do not hit the file system immediately. Instead, they are buffered in the page cache.

This enables **delayed allocation**. When a file system has more file content buffered, it can allocate longer contiguous areas of disk.

Suppose a file is 48K long and is stored contiguously. What does `i_blocks[]` contain?

```
i_blocks[] = {N, N+1, N+2, N+3, …}
```

This becomes worse yet for longer files.

Contiguous blocks of a file are called **extents**. They have a very compact representation: `{offset, length}`.

Delayed allocation and extent trees

File systems want to store files as contiguous areas on disks. This way accessing a file produces less random IO.

Recall that writes from a userspace application do not hit the file system immediately. Instead, they are buffered in the page cache.

This enables **delayed allocation**. When a file system has more file content buffered, it can allocate longer contiguous areas of disk.

Suppose a file is 48K long and is stored contiguously. What does `i_blocks[]` contain?

```
i_blocks[] = {N, N+1, N+2, N+3, …}
```

This becomes worse yet for longer files.

Contiguous blocks of a file are called **extents**. They have a very compact representation: `{offset, length}`.

Delayed allocation often decreases the number of extents in a file. Often, a file has only one extent. The list of extents in a file can be stored much more compactly.

Delayed allocation and extent trees

File systems want to store files as contiguous areas on disks. This way accessing a file produces less random IO.

Recall that writes from a userspace application do not hit the file system immediately. Instead, they are buffered in the page cache.

This enables **delayed allocation**. When a file system has more file content buffered, it can allocate longer contiguous areas of disk.

Suppose a file is 48K long and is stored contiguously. What does `i_blocks[]` contain?

```
i_blocks[] = {N, N+1, N+2, N+3, …}
```

This becomes worse yet for longer files.

Contiguous blocks of a file are called **extents**. They have a very compact representation: `{offset, length}`.

Delayed allocation often decreases the number of extents in a file. Often, a file has only one extent. The list of extents in a file can be stored much more compactly.

Files with <= 9 extents keep the list of their extents in `i_blocks[]`. Ext4 stores bigger extent lists as B-trees (will see it later).

Sparse files

Extents in ext4 are more complicated. They are triples of `{logical_offset, length, physical_offset}`.

- `logical_offset` shows where the extent is located in the file,
- `physical_offset` shows where the data of the extent is located on the disk.

One can construct a file with an extent tree that has the following two entries:

- `{logical_offset = 0,  length = 4K, physical_offset = X},`
- `{logical_offset = 8K, length = 4K, physical_offset = Y}.`

Sparse files

Extents in ext4 are more complicated. They are triples of `{logical_offset, length, physical_offset}`.

- `logical_offset` shows where the extent is located in the file,
- `physical_offset` shows where the data of the extent is located on the disk.

One can construct a file with an extent tree that has the following two entries:

- `{logical_offset = 0,  length = 4K, physical_offset = X},`
- `{logical_offset = 8K, length = 4K, physical_offset = Y}.`

What will the following reads do?

1. `pread(fd, buf, 4096, 0),`
2. `pread(fd, buf, 4096, 4096),`
3. `pread(fd, buf, 4096, 8192).`

Sparse files

Extents in ext4 are more complicated. They are triples of `{logical_offset, length, physical_offset}`.

- `logical_offset` shows where the extent is located in the file,
- `physical_offset` shows where the data of the extent is located on the disk.

One can construct a file with an extent tree that has the following two entries:

- `{logical_offset = 0,  length = 4K, physical_offset = X},`
- `{logical_offset = 8K, length = 4K, physical_offset = Y}.`

What will the following reads do?

1. `pread(fd, buf, 4096, 0),`
2. `pread(fd, buf, 4096, 4096),`
3. `pread(fd, buf, 4096, 8192).`

1. reads the first extent,
2. reads zeroes,
3. reads the second extent.

Quiz: how does ext2 represent sparse files? It has only `i_blocks[]`.

Sparse files and thin-provisioned storage

What is the use case for sparse files?

Sparse files and thin-provisioned storage

What is the use case for sparse files? – Storing images of disks of virtual machines.

Typically, a user allocates a large disk to a VM, but only a part of it is really used. Parts that were never written to contain zeroes and need not be stored. They are represented as holes in sparse files.
VM images that use this technique are called **thin-provisioned**.

Remark: this is the same idea as the memory overcommit in Linux.

Sparse files and thin-provisioned storage

What is the use case for sparse files? – Storing images of disks of virtual machines.

Typically, a user allocates a large disk to a VM, but only a part of it is really used. Parts that were never written to contain zeroes and need not be stored. They are represented as holes in sparse files.
VM images that use this technique are called **thin-provisioned**.

Remark: this is the same idea as the memory overcommit in Linux.

See also:

1. `st_size`, `st_blocks` and `st_blksize` in `struct stat`,
2. `ioctl(FS_IOC_FIEMAP)` and `ioctl(FIBMAP)`,
3. `lseek(SEEK_HOLE)` and lseek(`SEEK_DATA`),
4. `man 2 fallocate`.

Inline files

Recall that ext4 allocates only whole blocks to files. Very small files can avoid this penalty.

Files that have <= 60 bytes are stored directly in `i_blocks[]`. Such files are called **inline**.

Extended attributes

Recall that there is a mechanism to attach an Access Control List to a file. An ACL is an array of instructions “user X has access Y”.
See

- `man 5 acl`,
- `man 1 getfacl`,
- `man 1 setfacl`.

How does ext4 store the ACL of a file?

Extended attributes

Files can have **extended attributes (xattrs)** attached to them. See

- `man 2 fsetxattr`,
- `man 2 fgetxattr`,
- `man 1 setfattr`,
- `man 1 getfattr`.

Extended attributes are much like file content. The difference is that xattrs cannot be accessed randomly. One can only read the whole of an xattr and replace it.

Extended attributes

Files can have **extended attributes (xattrs)** attached to them. See

- `man 2 fsetxattr`,
- `man 2 fgetxattr`,
- `man 1 setfattr`,
- `man 1 getfattr`.

Extended attributes are much like file content. The difference is that xattrs cannot be accessed randomly. One can only read the whole of an xattr and replace it.

In the inode table each entry consists of struct ext4_inode that describes the inode itself, and a variable-length array that describes extended attributes:

```
struct ext4_itable_entry {
        struct ext4_inode               inode;
        struct ext4_xattr_ibody_header xattr_ibody_header;
        struct ext4_xattr_entry         xattrs[];
}
```

Extended attributes

Files can have **extended attributes (xattrs)** attached to them. See

- `man 2 fsetxattr`,
- `man 2 fgetxattr`,
- `man 1 setfattr`,
- `man 1 getfattr`.

Extended attributes are much like file content. The difference is that xattrs cannot be accessed randomly. One can only read the whole of an xattr and replace it.

In the inode table each entry consists of struct ext4_inode that describes the inode itself, and a variable-length array that describes extended attributes:

```
struct ext4_itable_entry {
        struct ext4_inode               inode;
        struct ext4_xattr_ibody_header xattr_ibody_header;
        struct ext4_xattr_entry         xattrs[];
}
```

There are more usages to xattrs:

- SELinux keeps security labels in xattrs,
- larger inline files (ext4-specific).

Directories in ext2

A directory is stored as a file. This file has the lower byte of `->i_mode` set to `EXT2_FT_DIR`.
Ext2 parses the content of such files as a list of variable-length records that have the following format:

- Each entry begins with a header

```
struct ext2_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[];                 /* File name, up to EXT2_NAME_LEN */
};
```

- The header is followed by a string (not null-terminated) that stores the file name.

Directories in ext2

A directory is stored as a file. This file has the lower byte of `->i_mode` set to `EXT2_FT_DIR`.
Ext2 parses the content of such files as a list of variable-length records that have the following format:

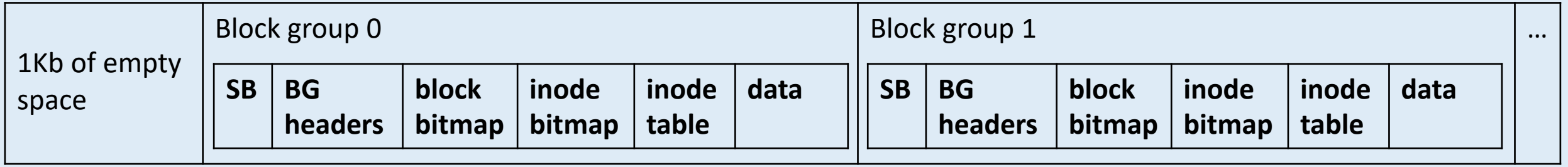
- Each entry begins with a header

```
struct ext2_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[];                 /* File name, up to EXT2_NAME_LEN */
};
```

- The header is followed by a string (not null-terminated) that stores the file name.

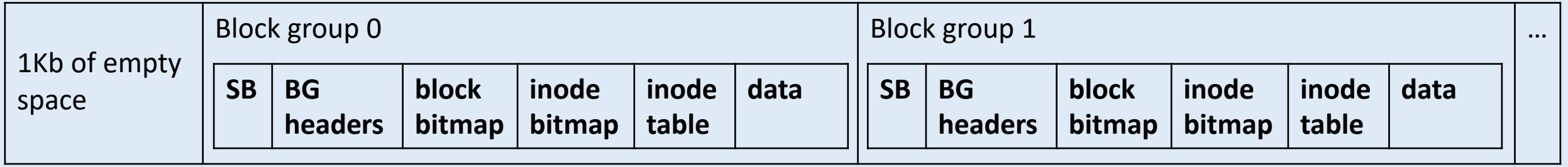
Remark: a directory entry never crosses the block boundary. The value of `->rec_len` of the last record in a block is selected so that the record extends precisely up to the block end.
Remark: if the field `->inode` is zero, then ext2 assumes that the current record is the last one in the current block.

Superblock



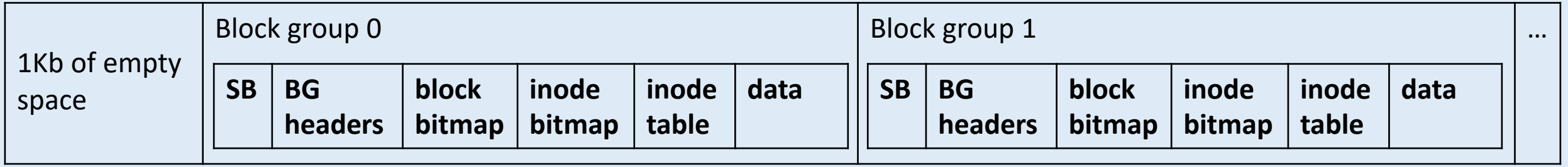

```
struct ext2_super_block {
  __le32 s_inodes_count;      /* Inodes count */
  __le32 s_blocks_count;      /* Blocks count */
  __le32 s_r_blocks_count;    /* Reserved blocks count */
  __le32 s_free_blocks_count; /* Free blocks count */
  __le32 s_free_inodes_count; /* Free inodes count */
  __le32 s_first_data_block;  /* First Data Block */
  __le32 s_log_block_size;    /* Block size */
  __le32 s_log_frag_size;     /* Fragment size */
  __le32 s_blocks_per_group;  /* # Blocks per group */
  __le32 s_frags_per_group;   /* # Fragments per group */
  __le32 s_inodes_per_group;  /* # Inodes per group */
  __le32 s_mtime;             /* Mount time */
  __le32 s_wtime;             /* Write time */
  __le16 s_mnt_count;         /* Mount count */
  __le16 s_max_mnt_count;     /* Maximal mount count */
  __le16 s_magic;             /* Magic signature */
  __le16 s_state;             /* File system state */
  __le16 s_errors;            /* Behaviour when detecting errors */
  __le16 s_minor_rev_level;   /* minor revision level */
  __le32 s_lastcheck;         /* time of last check */
  __le32 s_checkinterval;     /* max. time between checks */
  __le32 s_creator_os;        /* OS */
  __le32 s_rev_level;         /* Revision level */
  __le16 s_def_resuid;        /* Default uid for reserved blocks */
  __le16 s_def_resgid;        /* Default gid for reserved blocks */
  __le32 s_first_ino;         /* First non-reserved inode */
  __le16 s_inode_size;        /* size of inode structure */
  __le16 s_block_group_nr;    /* block group # of this sb */
  __le32 s_feature_compat;    /* compatible features */
  __le32 s_feature_incompat;  /* incompatible features */
  __le32 s_feature_ro_compat; /* readonly-compatible features */
  __u8   s_uuid[16];          /* 128-bit uuid for volume */
  char   s_volume_name[16];   /* volume name */
  char   s_last_mounted[64];  /* directory where last mounted */
  __le32 s_algorithm_usage_bitmap; /* For compression */
```

Superblock



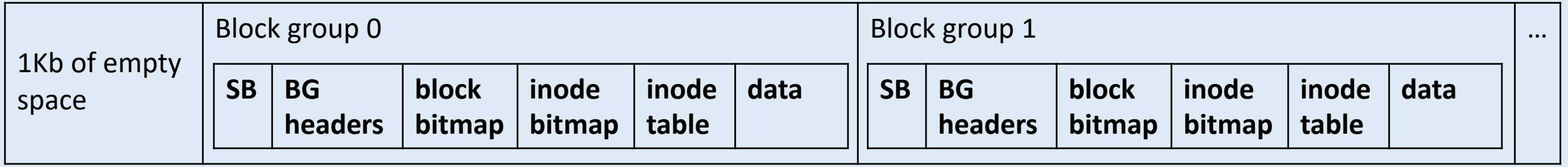

```
struct ext2_super_block {
  __le32 s_inodes_count;      /* Inodes count */
  __le32 s_blocks_count;      /* Blocks count */
  __le32 s_r_blocks_count;    /* Reserved blocks count */
  __le32 s_free_blocks_count; /* Free blocks count */
  __le32 s_free_inodes_count; /* Free inodes count */
  __le32 s_first_data_block;  /* First Data Block */
  __le32 s_log_block_size;    /* Block size */
  __le32 s_log_frag_size;     /* Fragment size */
  __le32 s_blocks_per_group;  /* # Blocks per group */
  __le32 s_frags_per_group;   /* # Fragments per group */
  __le32 s_inodes_per_group;  /* # Inodes per group */
  __le32 s_mtime;             /* Mount time */
  __le32 s_wtime;             /* Write time */
  __le16 s_mnt_count;         /* Mount count */
  __le16 s_max_mnt_count;     /* Maximal mount count */
  __le16 s_magic;             /* Magic signature */
  __le16 s_state;             /* File system state */
  __le16 s_errors;            /* Behaviour when detecting errors */
  __le16 s_minor_rev_level;   /* minor revision level */
  __le32 s_lastcheck;         /* time of last check */
  __le32 s_checkinterval;     /* max. time between checks */
  __le32 s_creator_os;        /* OS */
  __le32 s_rev_level;         /* Revision level */
  __le16 s_def_resuid;        /* Default uid for reserved blocks */
  __le16 s_def_resgid;        /* Default gid for reserved blocks */
  __le32 s_first_ino;         /* First non-reserved inode */
  __le16 s_inode_size;        /* size of inode structure */
  __le16 s_block_group_nr;    /* block group # of this sb */
  __le32 s_feature_compat;    /* compatible features */
  __le32 s_feature_incompat;  /* incompatible features */
  __le32 s_feature_ro_compat; /* readonly-compatible features */
  __u8   s_uuid[16];          /* 128-bit uuid for volume */
  char   s_volume_name[16];   /* volume name */
  char   s_last_mounted[64];  /* directory where last mounted */
  __le32 s_algorithm_usage_bitmap; /* For compression */
```

Superblock




```
struct ext2_super_block {
  __le32 s_inodes_count;      /* Inodes count */
  __le32 s_blocks_count;      /* Blocks count */
  __le32 s_r_blocks_count;    /* Reserved blocks count */
  __le32 s_free_blocks_count; /* Free blocks count */
  __le32 s_free_inodes_count; /* Free inodes count */
  __le32 s_first_data_block;  /* First Data Block */
  __le32 s_log_block_size;    /* Block size */
  __le32 s_log_frag_size;     /* Fragment size */
  __le32 s_blocks_per_group;  /* # Blocks per group */
  __le32 s_frags_per_group;   /* # Fragments per group */
  __le32 s_inodes_per_group;  /* # Inodes per group */
  __le32 s_mtime;             /* Mount time */
  __le32 s_wtime;             /* Write time */
  __le16 s_mnt_count;         /* Mount count */
  __le16 s_max_mnt_count;     /* Maximal mount count */
  __le16 s_magic;             /* Magic signature */
  __le16 s_state;             /* File system state */
  __le16 s_errors;            /* Behaviour when detecting errors */
  __le16 s_minor_rev_level;   /* minor revision level */
  __le32 s_lastcheck;         /* time of last check */
  __le32 s_checkinterval;     /* max. time between checks */
  __le32 s_creator_os;        /* OS */
  __le32 s_rev_level;         /* Revision level */
  __le16 s_def_resuid;        /* Default uid for reserved blocks */
  __le16 s_def_resgid;        /* Default gid for reserved blocks */
  __le32 s_first_ino;         /* First non-reserved inode */
  __le16 s_inode_size;        /* size of inode structure */
  __le16 s_block_group_nr;    /* block group # of this sb */
  __le32 s_feature_compat;    /* compatible features */
  __le32 s_feature_incompat;  /* incompatible features */
  __le32 s_feature_ro_compat; /* readonly-compatible features */
  __u8   s_uuid[16];          /* 128-bit uuid for volume */
  char   s_volume_name[16];   /* volume name */
  char   s_last_mounted[64];  /* directory where last mounted */
  __le32 s_algorithm_usage_bitmap; /* For compression */
```

Superblock




```
struct ext2_super_block {
  __le32 s_inodes_count;      /* Inodes count */
  __le32 s_blocks_count;      /* Blocks count */
  __le32 s_r_blocks_count;    /* Reserved blocks count */
  __le32 s_free_blocks_count; /* Free blocks count */
  __le32 s_free_inodes_count; /* Free inodes count */
  __le32 s_first_data_block;  /* First Data Block */
  __le32 s_log_block_size;    /* Block size */
  __le32 s_log_frag_size;     /* Fragment size */
  __le32 s_blocks_per_group;  /* # Blocks per group */
  __le32 s_frags_per_group;   /* # Fragments per group */
  __le32 s_inodes_per_group;  /* # Inodes per group */
  __le32 s_mtime;             /* Mount time */
  __le32 s_wtime;             /* Write time */
  __le16 s_mnt_count;         /* Mount count */
  __le16 s_max_mnt_count;     /* Maximal mount count */
  __le16 s_magic;             /* Magic signature */
  __le16 s_state;             /* File system state */
  __le16 s_errors;            /* Behaviour when detecting errors */
  __le16 s_minor_rev_level;   /* minor revision level */
  __le32 s_lastcheck;         /* time of last check */
  __le32 s_checkinterval;     /* max. time between checks */
  __le32 s_creator_os;        /* OS */
  __le32 s_rev_level;         /* Revision level */
  __le16 s_def_resuid;        /* Default uid for reserved blocks */
  __le16 s_def_resgid;        /* Default gid for reserved blocks */
  __le32 s_first_ino;         /* First non-reserved inode */
  __le16 s_inode_size;        /* size of inode structure */
  __le16 s_block_group_nr;    /* block group # of this sb */
  __le32 s_feature_compat;    /* compatible features */
  __le32 s_feature_incompat;  /* incompatible features */
  __le32 s_feature_ro_compat; /* readonly-compatible features */
  __u8   s_uuid[16];          /* 128-bit uuid for volume */
  char   s_volume_name[16];   /* volume name */
  char   s_last_mounted[64];  /* directory where last mounted */
  __le32 s_algorithm_usage_bitmap; /* For compression */
```

/etc/fstab and FS UUIDs

`/etc/fstab` lists mount options for frequently used file system. If a file system is listed in `/etc/fstab`, one can mount it simply with

```
# mount /path/to/mount/point
```

instead of

```
# mount -t <fstype> -o <mntopts> <blkdev> /path/to/mount/point
```

/etc/fstab and FS UUIDs

`/etc/fstab` lists mount options for frequently used file system. If a file system is listed in `/etc/fstab`, one can mount it simply with

```
# mount /path/to/mount/point
```

instead of

```
# mount -t <fstype> -o <mntopts> <blkdev> /path/to/mount/point
```

Two examples of /etc/fstab:

```
# / was on /dev/sda1 during installation
UUID=c113b43a-734a-40b4-a082-1175b620fe90 /    ext4 errors=remount-ro 0  1
# swap was on /dev/sda5 during installation
UUID=d4bd2d13-6e3c-4c3f-ba4c-b2f33d6fcdff none swap sw                0  0
```

```
# / is /dev/sda1
/dev/sda1 /    ext4  errors=remount-ro 0  1
# swap is /dev/sda5
/dev/sda5 none swap  sw                0  0
```

/etc/fstab and FS UUIDs

`/etc/fstab` lists mount options for frequently used file system. If a file system is listed in `/etc/fstab`, one can mount it simply with

```
# mount /path/to/mount/point
```

instead of

```
# mount -t <fstype> -o <mntopts> <blkdev> /path/to/mount/point
```

Two examples of /etc/fstab:

```
# / was on /dev/sda1 during installation
UUID=c113b43a-734a-40b4-a082-1175b620fe90 /    ext4 errors=remount-ro 0  1
# swap was on /dev/sda5 during installation
UUID=d4bd2d13-6e3c-4c3f-ba4c-b2f33d6fcdff none swap sw              0  0
```

```
# / is /dev/sda1
/dev/sda1 /    ext4  errors=remount-ro 0  1
# swap is /dev/sda5
/dev/sda5 none swap  sw              0  0
```

Quiz: how are the two fstabs different?

/etc/fstab and FS UUIDs

`/etc/fstab` lists mount options for frequently used file system. If a file system is listed in `/etc/fstab`, one can mount it simply with

```
# mount /path/to/mount/point
```

instead of

```
# mount -t <fstype> -o <mntopts> <blkdev> /path/to/mount/point
```

Two examples of /etc/fstab:

```
# / was on /dev/sda1 during installation
UUID=c113b43a-734a-40b4-a082-1175b620fe90 /    ext4 errors=remount-ro 0  1
# swap was on /dev/sda5 during installation
UUID=d4bd2d13-6e3c-4c3f-ba4c-b2f33d6fcdff none swap sw                0  0
```

```
# / is /dev/sda1
/dev/sda1 /    ext4  errors=remount-ro 0  1
# swap is /dev/sda5
/dev/sda5 none swap  sw                0  0
```

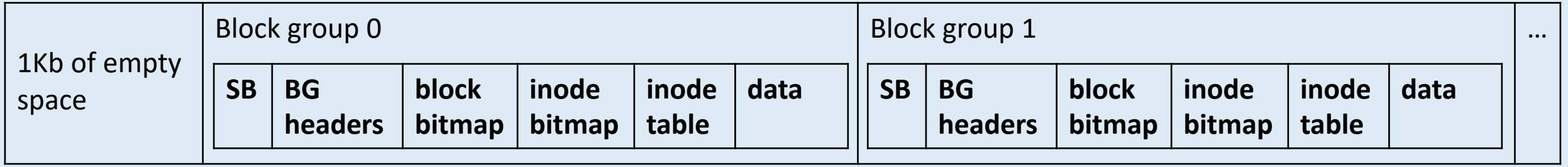
Quiz: how are the two fstabs different?

Quiz: what happens to FS UUIDs when a VM is cloned? See also https://lwn.net/Articles/923969/

See also:

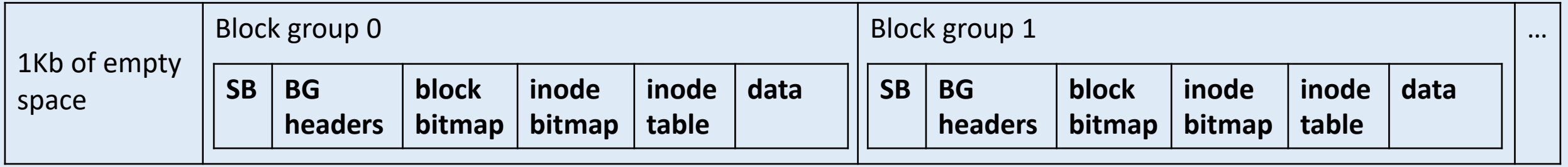
- man 8 lsblk
- man 8 blkid

Superblock




```
struct ext2_super_block {
  __le32 s_inodes_count;      /* Inodes count */
  __le32 s_blocks_count;      /* Blocks count */
  __le32 s_r_blocks_count;    /* Reserved blocks count */
  __le32 s_free_blocks_count; /* Free blocks count */
  __le32 s_free_inodes_count; /* Free inodes count */
  __le32 s_first_data_block;  /* First Data Block */
  __le32 s_log_block_size;    /* Block size */
  __le32 s_log_frag_size;     /* Fragment size */
  __le32 s_blocks_per_group;  /* # Blocks per group */
  __le32 s_frags_per_group;   /* # Fragments per group */
  __le32 s_inodes_per_group;  /* # Inodes per group */
  __le32 s_mtime;             /* Mount time */
  __le32 s_wtime;             /* Write time */
  __le16 s_mnt_count;         /* Mount count */
  __le16 s_max_mnt_count;     /* Maximal mount count */
  __le16 s_magic;             /* Magic signature */
  __le16 s_state;             /* File system state */
  __le16 s_errors;            /* Behaviour when detecting errors */
  __le16 s_minor_rev_level;   /* minor revision level */
  __le32 s_lastcheck;         /* time of last check */
  __le32 s_checkinterval;     /* max. time between checks */
  __le32 s_creator_os;        /* OS */
  __le32 s_rev_level;         /* Revision level */
  __le16 s_def_resuid;        /* Default uid for reserved blocks */
  __le16 s_def_resgid;        /* Default gid for reserved blocks */
  __le32 s_first_ino;         /* First non-reserved inode */
  __le16 s_inode_size;        /* size of inode structure */
  __le16 s_block_group_nr;    /* block group # of this sb */
  __le32 s_feature_compat;    /* compatible features */
  __le32 s_feature_incompat;  /* incompatible features */
  __le32 s_feature_ro_compat; /* readonly-compatible features */
  __u8   s_uuid[16];          /* 128-bit uuid for volume */
  char   s_volume_name[16];   /* volume name */
  char   s_last_mounted[64];  /* directory where last mounted */
  __le32 s_algorithm_usage_bitmap; /* For compression */
```

Superblock




```
struct ext2_super_block {
  __le32 s_inodes_count;      /* Inodes count */
  __le32 s_blocks_count;      /* Blocks count */
  __le32 s_r_blocks_count;    /* Reserved blocks count */
  __le32 s_free_blocks_count; /* Free blocks count */
  __le32 s_free_inodes_count; /* Free inodes count */
  __le32 s_first_data_block;  /* First Data Block */
  __le32 s_log_block_size;    /* Block size */
  __le32 s_log_frag_size;     /* Fragment size */
  __le32 s_blocks_per_group;  /* # Blocks per group */
  __le32 s_frags_per_group;   /* # Fragments per group */
  __le32 s_inodes_per_group;  /* # Inodes per group */
  __le32 s_mtime;             /* Mount time */
  __le32 s_wtime;             /* Write time */
  __le16 s_mnt_count;         /* Mount count */
  __le16 s_max_mnt_count;     /* Maximal mount count */
  __le16 s_magic;             /* Magic signature */
  __le16 s_state;             /* File system state */
  __le16 s_errors;            /* Behaviour when detecting errors */
  __le16 s_minor_rev_level;   /* minor revision level */
  __le32 s_lastcheck;         /* time of last check */
  __le32 s_checkinterval;     /* max. time between checks */
  __le32 s_creator_os;        /* OS */
  __le32 s_rev_level;         /* Revision level */
  __le16 s_def_resuid;        /* Default uid for reserved blocks */
  __le16 s_def_resgid;        /* Default gid for reserved blocks */
  __le32 s_first_ino;         /* First non-reserved inode */
  __le16 s_inode_size;        /* size of inode structure */
  __le16 s_block_group_nr;    /* block group # of this sb */
  __le32 s_feature_compat;    /* compatible features */
  __le32 s_feature_incompat;  /* incompatible features */
  __le32 s_feature_ro_compat; /* readonly-compatible features */
  __u8   s_uuid[16];          /* 128-bit uuid for volume */
  char   s_volume_name[16];   /* volume name */
  char   s_last_mounted[64];  /* directory where last mounted */
  __le32 s_algorithm_usage_bitmap; /* For compression */
```

Compat, ro-compat, incompat features

Compat features: older implementations of ext4 can read and modify a file system that uses such features.

RO-compat features: older implementations can only read a file system.

Incompat features: older implementations cannot mount a file system.

Compat, ro-compat, incompat features

Compat features: older implementations of ext4 can read and modify a file system that uses such features.

RO-compat features: older implementations can only read a file system.

Incompat features: older implementations cannot mount a file system.

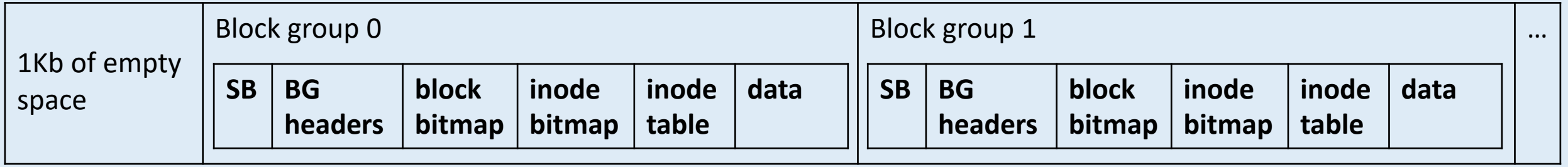
This is not the only classification of features. QCOW2 has compat-discard features. For example, a QCOW2 image may have a CBT map (Changed Block Tracking Map). Older QEMU versions that do not support this feature may simply delete a CBT map and then use the image.

Compat, ro-compat, incompat features

Compat features: older implementations of ext4 can read and modify a file system that uses such features.

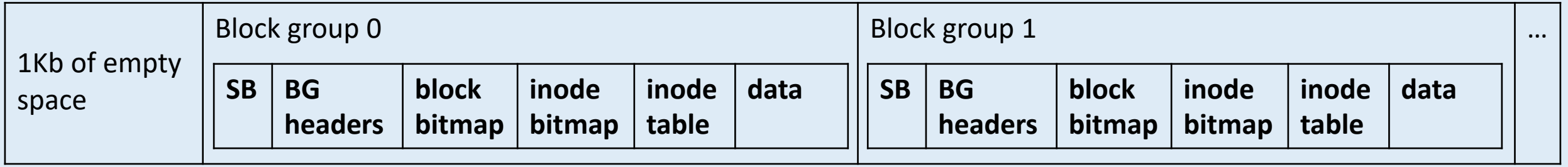
- EXT4_FEATURE_COMPAT_DIR_PREALLOC
- EXT4_FEATURE_COMPAT_HAS_JOURNAL
- EXT4_FEATURE_COMPAT_EXT_ATTR
- EXT4_FEATURE_COMPAT_RESIZE_INODE

EXT4_FEATURE_COMPAT_RESIZE_INODE



When a file system is created, it needs to write BG headers in every block group of the file system.

EXT4_FEATURE_COMPAT_RESIZE_INODE

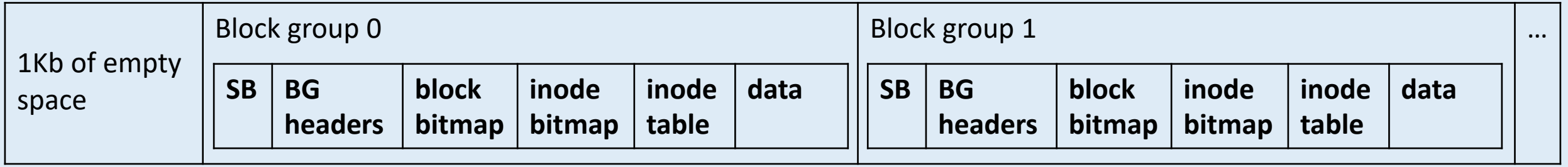


When a file system is created, it needs to write BG headers in every block group of the file system.

This is not friendly towards thin-provisioned disks.

Quiz: how many extents does a thin-provisioned disk is likely to have after `mkfs.ext2`?

EXT4_FEATURE_COMPAT_RESIZE_INODE



When a file system is created, it needs to write BG headers in every block group of the file system.

This is not friendly towards thin-provisioned disks.

Quiz: how many extents does a thin-provisioned disk is likely to have after `mkfs.ext2`?

File systems with EXT4_FEATURE_COMPAT_RESIZE_INODE initialise only a small portion of block groups during `mkfs`.
The rest are initialised when first needed.

Quiz: why is resize_inode a compat feature?

Compat, ro-compat, incompat features

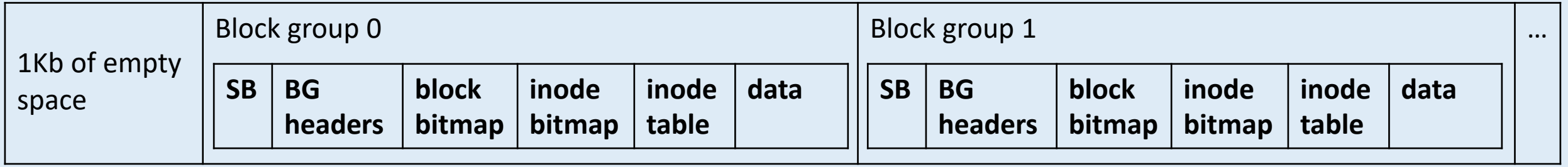
Compat features: older implementations of ext4 can read and modify a file system that uses such features.

- EXT4_FEATURE_COMPAT_DIR_PREALLOC
- EXT4_FEATURE_COMPAT_HAS_JOURNAL
- EXT4_FEATURE_COMPAT_EXT_ATTR
- EXT4_FEATURE_COMPAT_RESIZE_INODE

RO-compat features: older implementations can only read a file system.

- EXT4_FEATURE_RO_COMPAT_LARGE_FILE
- EXT4_FEATURE_RO_COMPAT_QUOTA
- EXT4_FEATURE_RO_COMPAT_SPARSE_SUPER

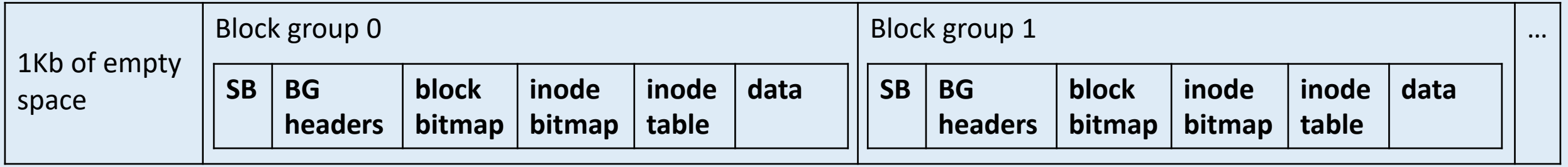
EXT4_FEATURE_RO_COMPAT_SPARSE_SUPER



Each BG contains copies of

- the SB,
- BG headers of every group.

EXT4_FEATURE_RO_COMPAT_SPARSE_SUPER



Each BG contains copies of

- the SB,
- BG headers of every group.

File systems with EXT4_FEATURE_RO_COMPAT_SPARSE_SUPER place copies of the superblock only in a few block groups.

Compat, ro-compat, incompat features

Compat features: older implementations of ext4 can read and modify a file system that uses such features.

- EXT4_FEATURE_COMPAT_DIR_PREALLOC
- EXT4_FEATURE_COMPAT_HAS_JOURNAL
- EXT4_FEATURE_COMPAT_EXT_ATTR
- EXT4_FEATURE_COMPAT_RESIZE_INODE

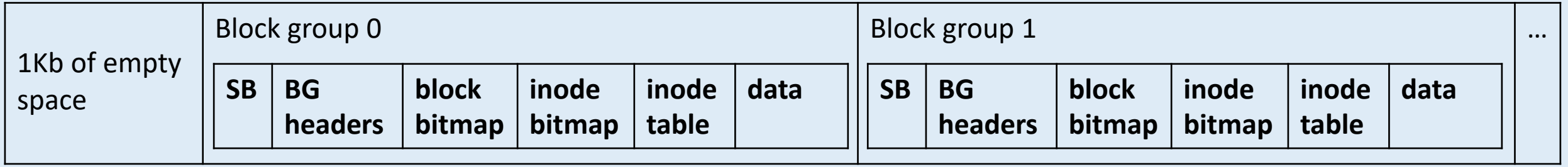
RO-compat features: older implementations can only read a file system.

- EXT4_FEATURE_RO_COMPAT_LARGE_FILE
- EXT4_FEATURE_RO_COMPAT_QUOTA
- EXT4_FEATURE_RO_COMPAT_SPARSE_SUPER

Incompat features: older implementations cannot mount a file system.

- EXT4_FEATURE_INCOMPAT_EXTENTS
- EXT4_FEATURE_INCOMPAT_INLINE_DATA
- EXT4_FEATURE_INCOMPAT_COMPRESSION
- EXT4_FEATURE_INCOMPAT_ENCRYPT
- EXT4_FEATURE_INCOMPAT_META_BG
- EXT4_FEATURE_INCOMPAT_FLEX_BG

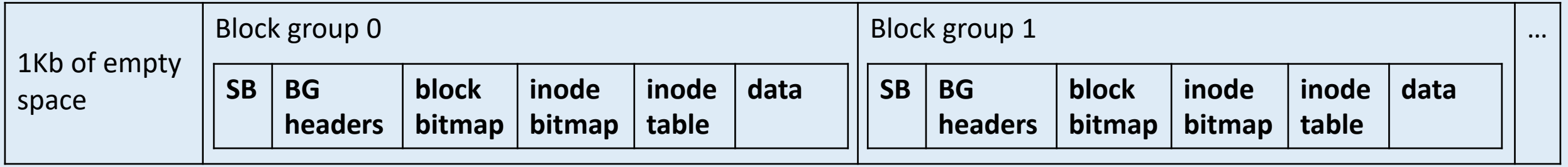
EXT4_FEATURE_INCOMPAT_META_BG



Each BG contains copies of

- the SB,
- BG headers of every group.

EXT4_FEATURE_INCOMPAT_META_BG



Each BG contains copies of

- the SB,
- BG headers of every group.

EXT4_FEATURE_INCOMPAT_META_BG decreases the number of copies of BG headers.
BGs are grouped into “meta groups” that have 64 BGs (4096/64 if the block size and the BG header size are default).
Within a meta group only BGs 0, 1 and 63 contain copies of BG headers.

See also

- http://www.nongnu.org/ext2-doc
- https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout
- http://wiki.osdev.org/Ext2

- https://lwn.net/Articles/322823/

To do at home

An ext2 file system has a file that is 1024 blocks long. The blocks are located sequentially. One block is 4K bytes long.

How much time will it take to read that file? Assume a typical HDD and consider the following two scenarios:

1. Blocks are read one at a time: map the logical offset within a file to the LBA, read that block, move on to the next one.
2. Read the inode and the block with indirect pointers into the RAM first, make a batch request to read file blocks, and issue a read of 4 megabytes in one request.