

Основы построения файловых систем



Два способа записать целое число в память или на диск

Два способа записать целое число в память или на диск

В начале идут старшие байты (**Big-endian**)

u32 x = 0x1A2B3C4D;

На диске:

1A 2B 3C 4D | | ..
 нумерация байтов на диске

Используется в:

- PowerPC
- Itanium

Два способа записать целое число в память или на диск

В начале идут старшие байты (**Big-endian**)

u32 x = 0x1A2B3C4D;


На диске:
1A 2B 3C 4D | | ..
 нумерация байтов на диске

Используется в:

- PowerPC
- Itanium

В начале идут младшие байты (**little-endian**)

u32 x = 0x1A2B3C4D;

На диске:
4D 3C 2B 1A | | ..
 нумерация байтов на диске

Используется в:

- x86

Два способа записать целое число в память или на диск

В начале идут старшие байты (**Big-endian**)

u32 x = 0x1A2B3C4D;

На диске:

1A 2B 3C 4D | | ..
 нумерация байтов на диске


Используется в:

- PowerPC
- Itanium

В начале идут младшие байты (**little-endian**)

u32 x = 0x1A2B3C4D;

На диске:

4D 3C 2B 1A | | ..
 нумерация байтов на диске

Используется в:

- x86

Примечание: PowerPC, Itanium, ARM, MIPS на самом деле bi-endian, т.е. умеют работать как с little-endian, так и big-endian данными.

Как обмениваться данными между little-endian и big-endian системами?

При сохранении преобразовать данные из host byte order в некоторый фиксированный:

```
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;
}
```

При чтении данных проделать обратное преобразование.

Как обмениваться данными между little-endian и big-endian системами?

Определение 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));
```

Как обмениваться данными между little-endian и big-endian системами?

Определение 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));
```

Более простой способ

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


Как обмениваться данными между little-endian и big-endian системами?

Определение struct dmap_ext_ondisk

Более простой способ

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

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

Как структуры будут выглядеть в памяти на x86_64?

8 байт	item_id
8 байт	ext_offs
3 байта	ext_len
8 байт	wr_seq
4 байта	slice_id

Как обмениваться данными между little-endian и big-endian системами?

Определение struct dmap_ext_ondisk

Более простой способ

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

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

Как структуры будут выглядеть в памяти на x86_64?

8 байт	item_id	8 байт	item_id
8 байт	ext_offs	8 байт	ext_offs
3 байта	ext_len	3 байта	ext_len
8 байт	wr_seq	5 байт	padding
4 байта	slice_id	8 байт	wr_seq
		4 байта	slice_id
		4 байта	padding

Как обмениваться данными между little-endian и big-endian системами?

Определение struct dmap_ext_ondisk

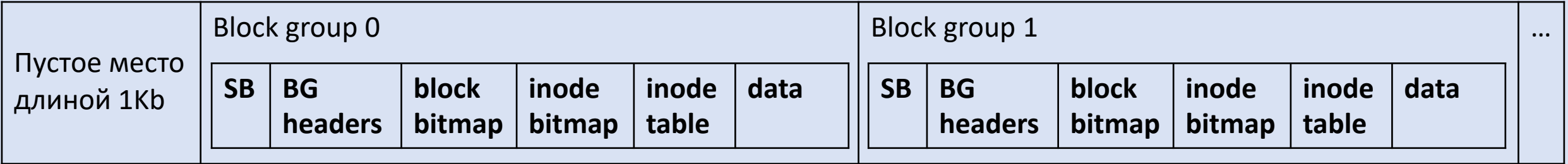
Более простой способ


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

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

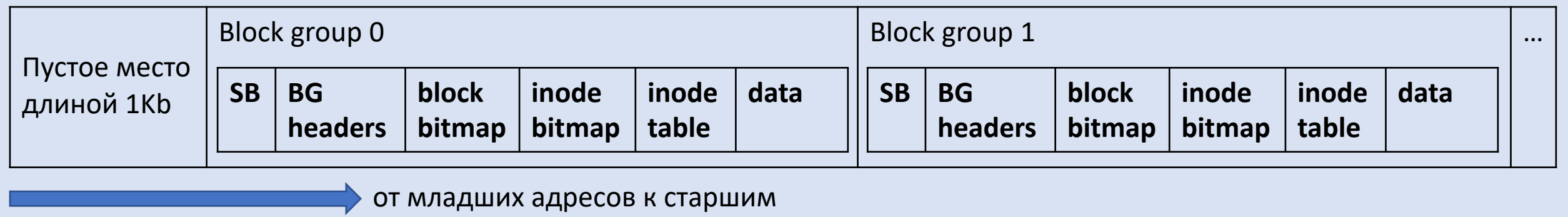
Как структуры будут выглядеть в памяти на x86_64?				А как на x86_32?	
8 байт	item_id	8 байт	item_id	4 байта	item_id
8 байт	ext_offs	8 байт	ext_offs	4 байта	ext_offs
3 байта	ext_len	3 байта	ext_len	3 байта	ext_len
8 байт	wr_seq	5 байт	padding	1 байт	padding
4 байта	slice_id	8 байт	wr_seq	4 байта	wr_seq
		4 байта	slice_id	4 байта	slice_id
		4 байта	padding		

Устройство ext2 в целом



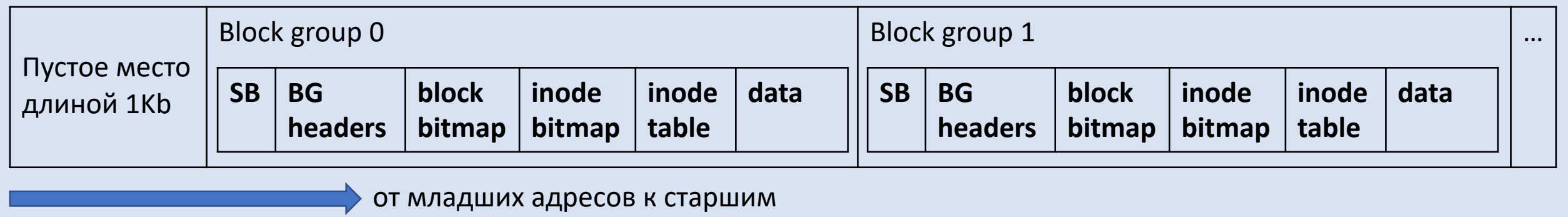
 от младших адресов к старшим

Устройство ext2 в целом



Superblock (SB) содержит информацию о файловой системе в целом: её размер, размер и число блоков и т.п.

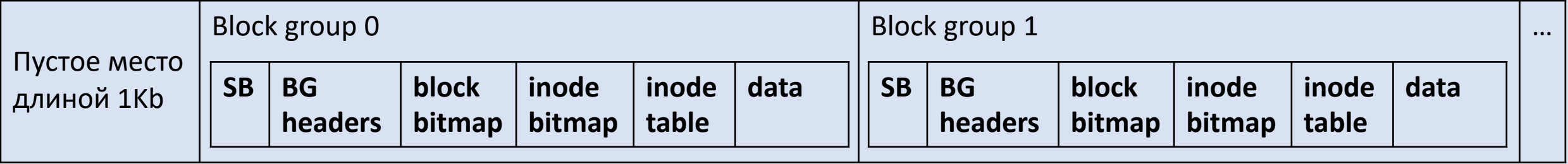
Устройство ext2 в целом




Superblock (SB) содержит информацию о файловой системе в целом: её размер, размер и число блоков и т.п.

Block Group Header содержит информацию об отдельной группе блоков: число свободных блоков и инод.

Устройство ext2 в целом



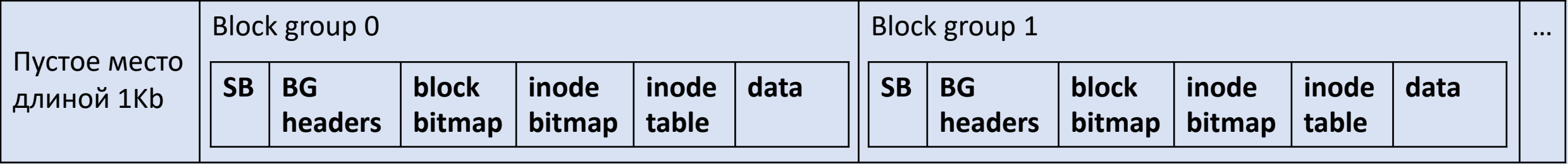
 от младших адресов к старшим


Superblock (SB) содержит информацию о файловой системе в целом: её размер, размер и число блоков и т.п.

Block Group Header содержит информацию об отдельной группе блоков: число свободных блоков и инод.

Замечание: ФС разделяется на множество блочных групп для того, чтобы увеличить локальность доступа: пока блоки удаётся выделять в пределах одной группы, уменьшается расстояние, на которое надо двигать читающие головки, плюс есть шанс уместить все метаданные одной блочной группы в памяти.

Устройство ext2 в целом



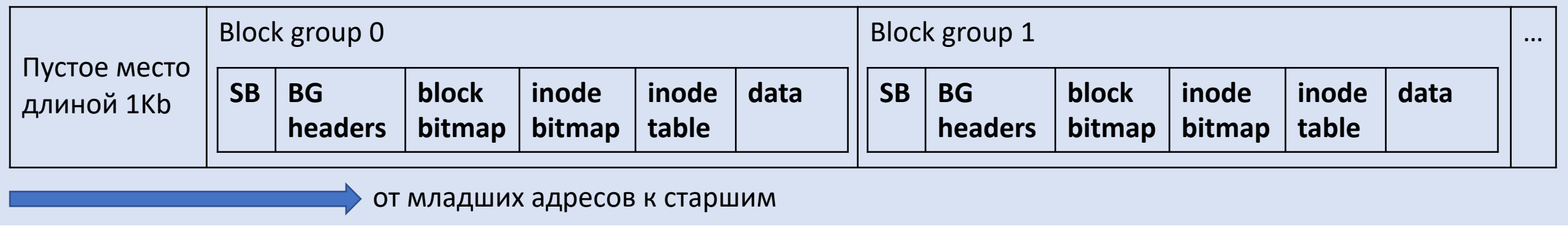
 от младших адресов к старшим

Superblock (SB) содержит информацию о файловой системе в целом: её размер, размер и число блоков и т.п.

Block Group Header содержит информацию об отдельной группе блоков: число свободных блоков и инод.

Block bitmap – это битовый массив, определяющий, которые из блоков заняты, а какие свободны. Блок – это минимальная единица выделения места на ФС.

Устройство ext2 в целом



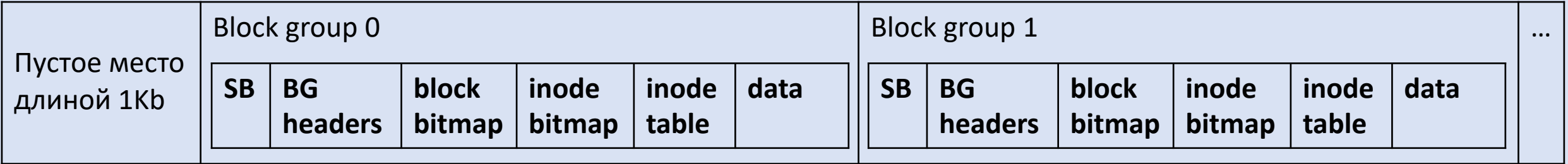
Superblock (SB) содержит информацию о файловой системе в целом: её размер, размер и число блоков и т.п.


Block Group Header содержит информацию об отдельной группе блоков: число свободных блоков и инод.

Block bitmap – это битовый массив, определяющий, которые из блоков заняты, а какие свободны. Блок – это минимальная единица выделения места на ФС.

Вопрос: почему место выделяется блоками а не байтами? Казалось бы, выделять как минимум 4K на файл, даже если он короткий – это излишне расточительно.

Устройство ext2 в целом



 от младших адресов к старшим

Superblock (SB) содержит информацию о файловой системе в целом: её размер, размер и число блоков и т.п.

Block Group Header содержит информацию об отдельной группе блоков: число свободных блоков и инод.

Block bitmap – это битовый массив, определяющий, которые из блоков заняты, а какие свободны. Блок – это минимальная единица выделения места на ФС.

Inode bitmap – это битовый массив, определяющий, который из инод заняты, а какие свободны. Inode (Index node) – это структура, описывающая один файл на ext2.

Inode table – это область на диске, хранящая содержимое инод. Они расположены как непрерывный массив.

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

В ext2 информация о свойствах файла и его расположении на диске содержится в структуре index node:

```
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)

В `ext2_inode->i_block[]` хранится список блоков, которые составляют файл.

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

В `ext2_inode->i_block[]` хранится список блоков, которые составляют файл. Но в этом массиве 15 элементов. Как быть с файлами, которые длиннее 15 блоков?

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

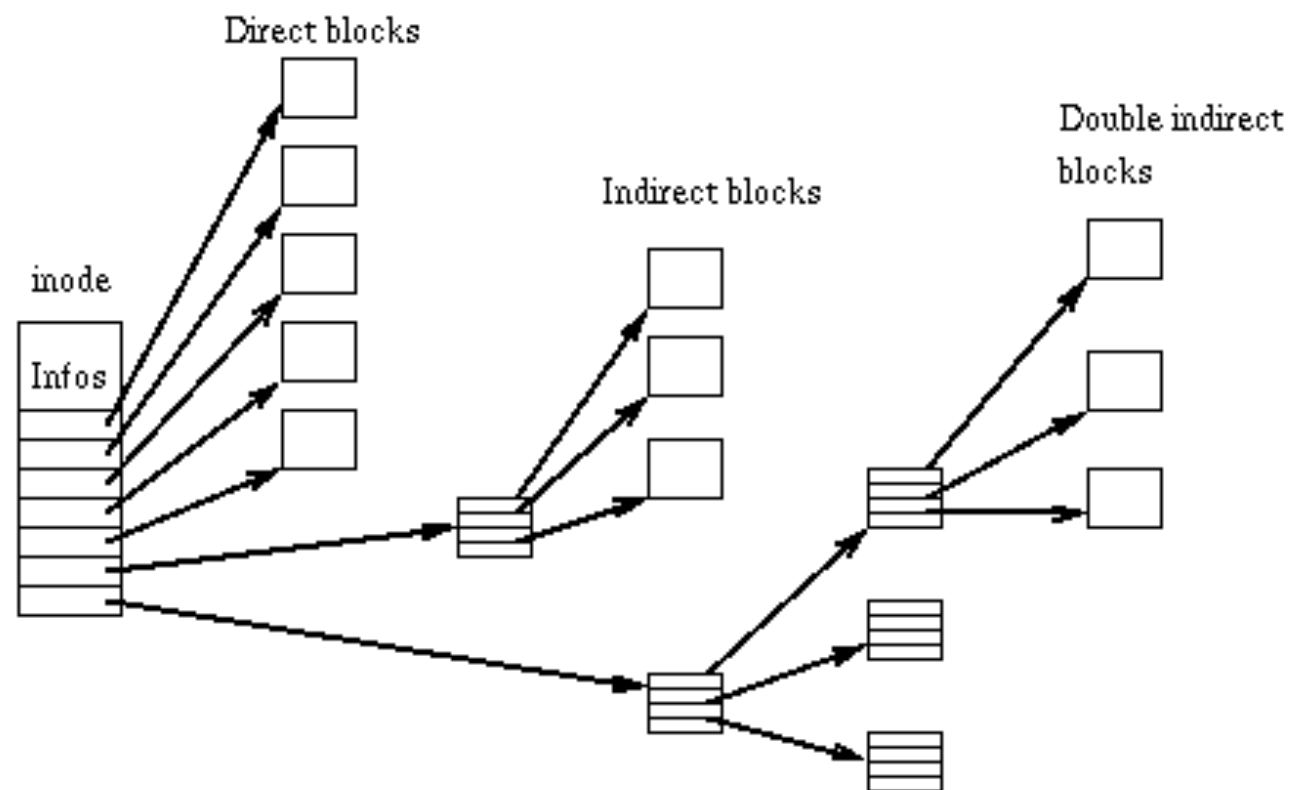
В `ext2_inode->i_block[]` хранится список блоков, которые составляют файл. Но в этом массиве 15 элементов. Как быть с файлами, которые длиннее 15 блоков?

Последние три элемента в `i_block[]` косвенные, т.е. указывают на блоки, которые сами являются списками блоков.

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

В `ext2_inode->i_block[]` хранится список блоков, которые составляют файл. Но в этом массиве 15 элементов. Как быть с файлами, которые длиннее 15 блоков?

Последние три элемента в `i_block[]` косвенные, т.е. указывают на блоки, которые сами являются списками блоков. Они имеют уровни косвенности 1, 2 и 3, соответственно.



Каталоги в ext2

Каталог хранится в файле специального типа (у которого младший байт `i_mode` равен `EXT2_FT_DIR`).

Содержимое файла представляет собой последовательность записей переменной длины:

- В заголовке записи стоит

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

- После `struct ext2_dir_entry_2` следует имя файла.

Каталоги в ext2

Каталог хранится в файле специального типа (у которого младший байт `i_mode` равен `EXT2_FT_DIR`).

Содержимое файла представляет собой последовательность записей переменной длины:

- В заголовке записи стоит

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

- После `struct ext2_dir_entry_2` следует имя файла.

Примечание: запись об элементе каталога никогда не пересекает границы блока; `rec_len` у последней записи подбирается так, чтобы она заканчивалась точно на границе.

Ещё примечание: если поле `inode` равно нулю, то считается, что список в текущем блоке закончился.

Каталоги в ext2

Каталог хранится в файле специального типа (у которого младший байт `i_mode` равен `EXT2_FT_DIR`).
Содержимое файла представляет собой последовательность записей переменной длины:

- В заголовке записи стоит

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

- После `struct ext2_dir_entry_2` следует имя файла.

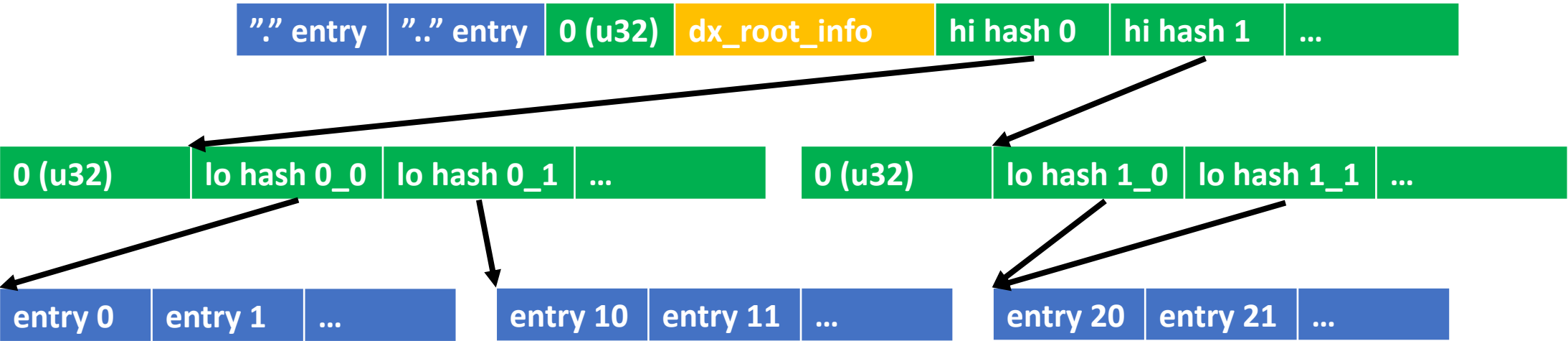
Примечание: запись об элементе каталога никогда не пересекает границы блока; `rec_len` у последней записи подбирается так, чтобы она заканчивалась точно на границе.

Ещё примечание: если поле `inode` равно нулю, то считается, что список в текущем блоке закончился.

Как быть с удалением элементов каталога?

Каталоги в ext3 (hash indexed dirs)

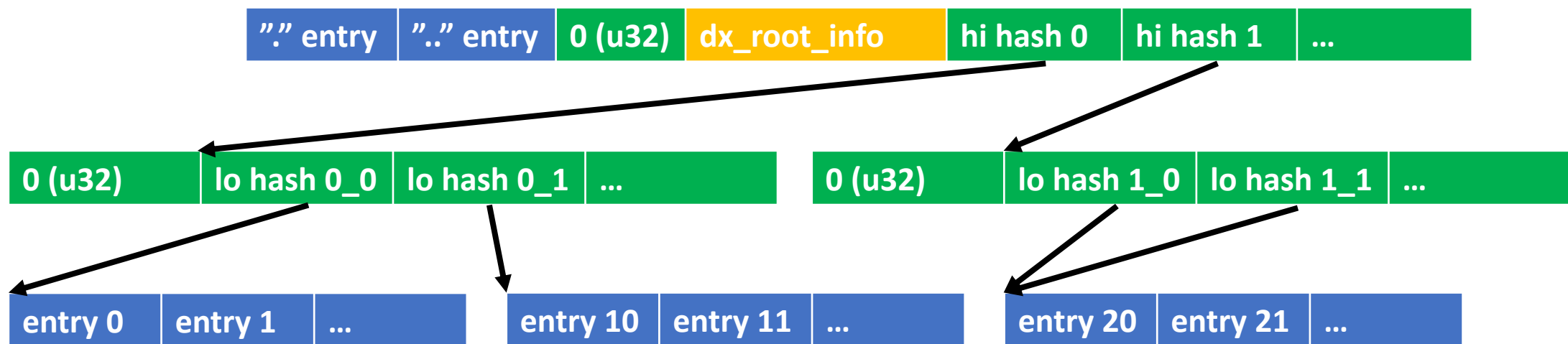
Для больших каталогов используется следующее представление*:



* Каждый узел изображённого дерева занимает один блок на диске.

Каталоги в ext3 (hash indexed dirs)

Для больших каталогов используется следующее представление*:

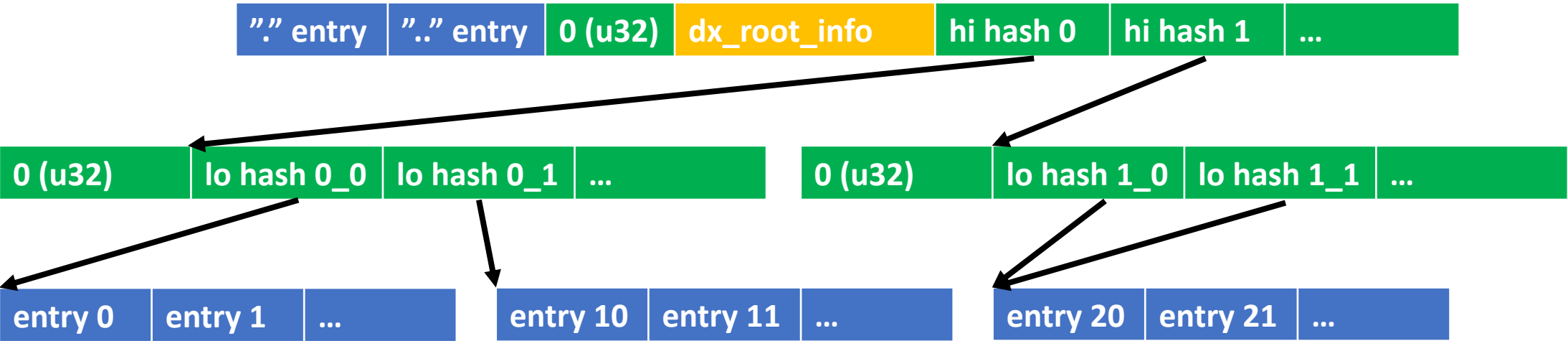


- Если много имён имеют совпадающий хеш и их список не умещается в один блок, то в карте “хеш → номер блока” ставится флаг «список продолжается в следующем блоке».
- Разные хеши могут ссылаться на один блок.

* Каждый узел изображённого дерева занимает один блок на диске.

Каталоги в ext3 (hash indexed dirs)

Для больших каталогов используется следующее представление*:



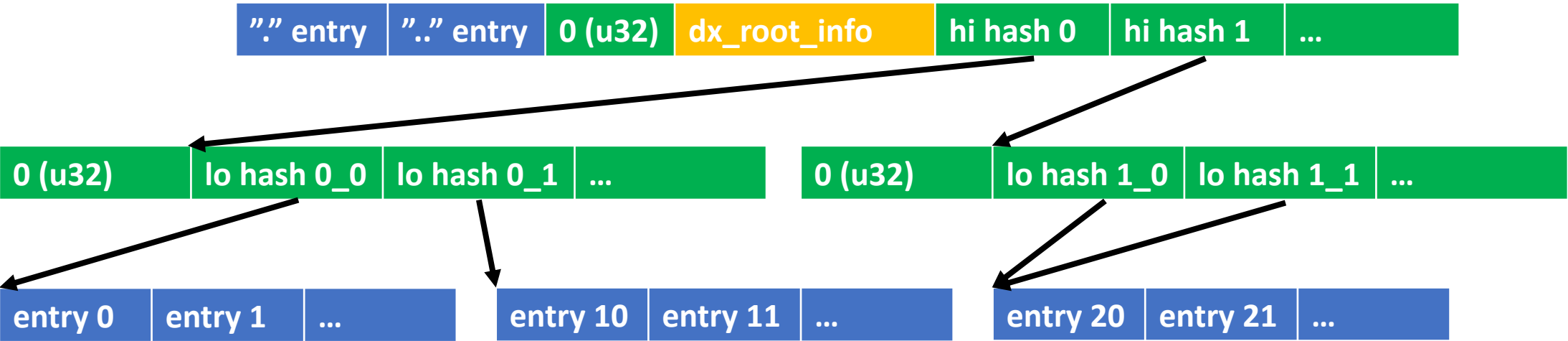
- Если много имён имеют совпадающий хеш и их список не умещается в один блок, то в карте “хеш → номер блока” ставится флаг «список продолжается в следующем блоке».
- Разные хеши могут ссылаться на один блок.

Изображённые выше блоки на диске располагаются подряд (и составляют один файл).

* Каждый узел изображённого дерева занимает один блок на диске.

Каталоги в ext3 (hash indexed dirs)

Для больших каталогов используется следующее представление*:



- Если много имён имеют совпадающий хеш и их список не умещается в один блок, то в карте “хеш → номер блока” ставится флаг «список продолжается в следующем блоке».
- Разные хеши могут ссылаться на один блок.

Изображённые выше блоки на диске располагаются подряд (и составляют один файл).

Нулевые записи в блоках нижнего уровня и нулевое 4-байтовое значение в корневом блоке поставлены затем, чтобы алгоритм линейного поиска из ext2 увидел правильный список элементов (вспоминаем, что элемент с нулевым полем `inode` – это признак «в этом блоке больше нет записей»).

** Каждый узел изображённого дерева занимает один блок на диске.*

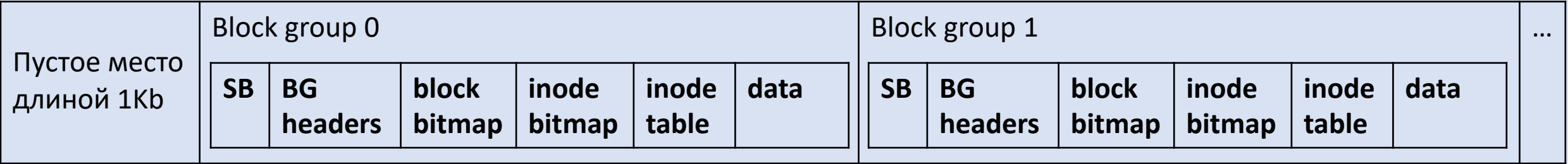
Superblock

Пустое место длиной 1Kb	Block group 0						Block group 1						...
	SB	BG headers	block bitmap	inode bitmap	inode table	data	SB	BG headers	block bitmap	inode bitmap	inode table	data	

```
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

Пустое место длиной 1Kb	Block group 0						Block group 1						...
	SB	BG headers	block bitmap	inode bitmap	inode table	data	SB	BG headers	block bitmap	inode bitmap	inode table	data	

```
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

Пустое место длиной 1Kb	Block group 0						Block group 1						...
	SB	BG headers	block bitmap	inode bitmap	inode table	data	SB	BG headers	block bitmap	inode bitmap	inode table	data	

```
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

Пустое место длиной 1Kb	Block group 0						Block group 1						...
	SB	BG headers	block bitmap	inode bitmap	inode table	data	SB	BG headers	block bitmap	inode bitmap	inode table	data	

```
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: старые реализации ext2 могут и читать, и писать на такую файловую систему.

RO-compat features: старые реализации могут корректно читать такую ФС, но писать в неё уже нет.

Incompat features: старые реализации не могут смонтировать такую ФС.

Compat, ro-compat, incompat features

Compat features: старые реализации ext2 могут и читать, и писать на такую файловую систему.

RO-compat features: старые реализации могут корректно читать такую ФС, но писать в неё уже нет.

Incompat features: старые реализации не могут смонтировать такую ФС.

Compat-discard features (QCOW2): старые реализации могут и читать, и писать, но должны обнулить указатели на структуры, которые они не поддерживают.

Пример: CBT map (Changed Block Tracking map).

Compat, ro-compat, incompat features

Compat features: старые реализации ext2 могут и читать, и писать на такую файловую систему.

- EXT4_FEATURE_COMPAT_DIR_PREALLOC
- EXT4_FEATURE_COMPAT_HAS_JOURNAL
- EXT4_FEATURE_COMPAT_EXT_ATTR
- EXT4_FEATURE_COMPAT_DIR_INDEX (hash directories)

Compat, ro-compat, incompat features

Compat features: старые реализации ext2 могут и читать, и писать на такую файловую систему.

- EXT4_FEATURE_COMPAT_DIR_PREALLOC
- EXT4_FEATURE_COMPAT_HAS_JOURNAL
- EXT4_FEATURE_COMPAT_EXT_ATTR
- EXT4_FEATURE_COMPAT_DIR_INDEX (hash directories)

Ro-compat features: старые реализации могут корректно читать такую ФС, но писать в неё уже нет.

- EXT4_FEATURE_RO_COMPAT_SPARSE_SUPER
- EXT4_FEATURE_RO_COMPAT_HUGE_FILE
- EXT4_FEATURE_RO_COMPAT_QUOTA

Compat, ro-compat, incompat features

Compat features: старые реализации ext2 могут и читать, и писать на такую файловую систему.

- EXT4_FEATURE_COMPAT_DIR_PREALLOC
- EXT4_FEATURE_COMPAT_HAS_JOURNAL
- EXT4_FEATURE_COMPAT_EXT_ATTR
- EXT4_FEATURE_COMPAT_DIR_INDEX (hash directories)

Ro-compat features: старые реализации могут корректно читать такую ФС, но писать в неё уже нет.

- EXT4_FEATURE_RO_COMPAT_SPARSE_SUPER
- EXT4_FEATURE_RO_COMPAT_HUGE_FILE
- EXT4_FEATURE_RO_COMPAT_QUOTA

Incompat features: старые реализации не могут смонтировать такую ФС.

- EXT4_FEATURE_INCOMPAT_COMPRESSION
- EXT4_FEATURE_INCOMPAT_JOURNAL_DEV
- EXT4_FEATURE_INCOMPAT_EXTENTS
- EXT4_FEATURE_INCOMPAT_INLINE_DATA
- EXT4_FEATURE_INCOMPAT_ENCRYPT

Пример (почти) compat feature: 32-битные UID и GID владельца

Напоминание: хвост struct ext2_inode выглядит так:

```
    __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];
};
```

Операционные системы, которые не используют поле `osd2`, должны сохранять его без изменений.

Пример (почти) compat feature: 32-битные UID и GID владельца

Для linux хвост struct ext2_inode выглядит так:

```
__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 */
union {
    struct {
        __u8      l_i_frag;        /* Fragment number */
        __u8      l_i_fsize;       /* Fragment size */
        __u16     i_pad1;
        __le16    l_i_uid_high;    /* these 2 fields      */
        __le16    l_i_gid_high;    /* were reserved2[0] */
        __u32     l_i_reserved2;
    } linux2;
    } osd2;                      /* OS dependent 2 */
};
```

Дополнительное чтение

- <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/>

Домашнее задание

На разделе ext2 расположен файл длиной 1024 блока, блоки которого идут подряд. Один блок имеет размер 8192 байт.

Сколько времени потребуется (для типичного HDD), чтобы прочесть этот файл в следующих случаях:

1. Чтение выполняется по одному блоку за итерацию (по логическому смещению блока определили его номер на диске, прочли блок, перешли к следующему),
2. Содержимое иноды и блоков с указателями зачитывается в память целиком, формируются большие запросы на чтение данных, эти запросы исполняются.

1. Почему htree directories (EXT4_FEATURE_COMPAT_DIR_INDEX) – это compat feature?
2. Разобраться с mkfs.ext2, создать образ ext2, и написать программу, которая
 - Перечислит элементы в любом каталоге по номеру его inode,
 - (*) Перечислит элементы в любом каталоге, заданном путём,
 - Прочтёт файл, заданный номером его inode,
 - (*) Прочтёт файл, заданный путём.
3. (***) Реализовать модуль для FUSE, который примонтирует образ ext2 в режиме только для чтения.