演習2の解答例

2006年12月22日

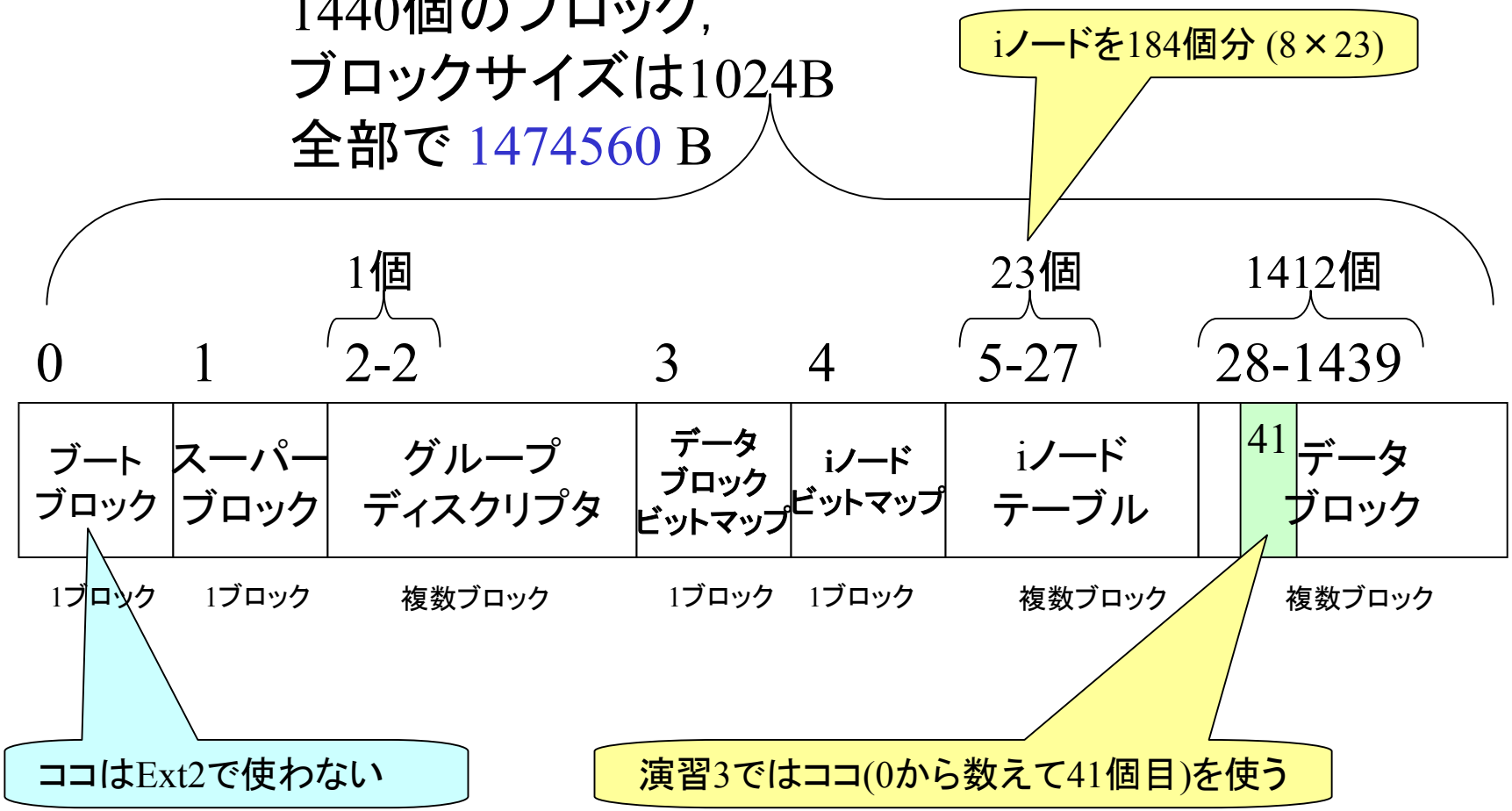
海谷 治彦

目次

- 演習2編
  - Ext2の構造，ext2_dir_entry_2
  - ディレクトリエントリの終端は？
  - 解答例1 ブロック毎にちまちま読んでいく
  - 解答例2 1.5MBくらいのデータ，いっきに読み！
- 機種依存の話
  - バイトオーダー問題
  - Padding問題

演習で使うFSの構造 (ブロックグループが1個)

1440個のブロック,
ブロックサイズは1024B
全部で 1474560 B



演習2の大まかな流れ

- ブロック41を取り出す.
  - サンプル getblock.c (任意のブロックを切り出すもの)
- その中身を構造体 ext2_dir_entry_2 に従い解析する.
  - この構造体は可変長なのでタチが悪い.
  - 個々のインスタンスの長さは読み込んで，メンバー rec_lenの値を読まないと解らない.
  - よって，まずは本構造体が最大長であると仮定して，ext2_dir_entry_2 構造体に読み込み，rec_len の長さを得て，次のインスタンスまでポインタを進めればよい.
    - 上記がヒントですが，さらに多少の工夫が必要.

ext2_dir_entry_2

型: 1 通常ファイル, 2 ディレクトリ, 7 シンボリックリンク 他

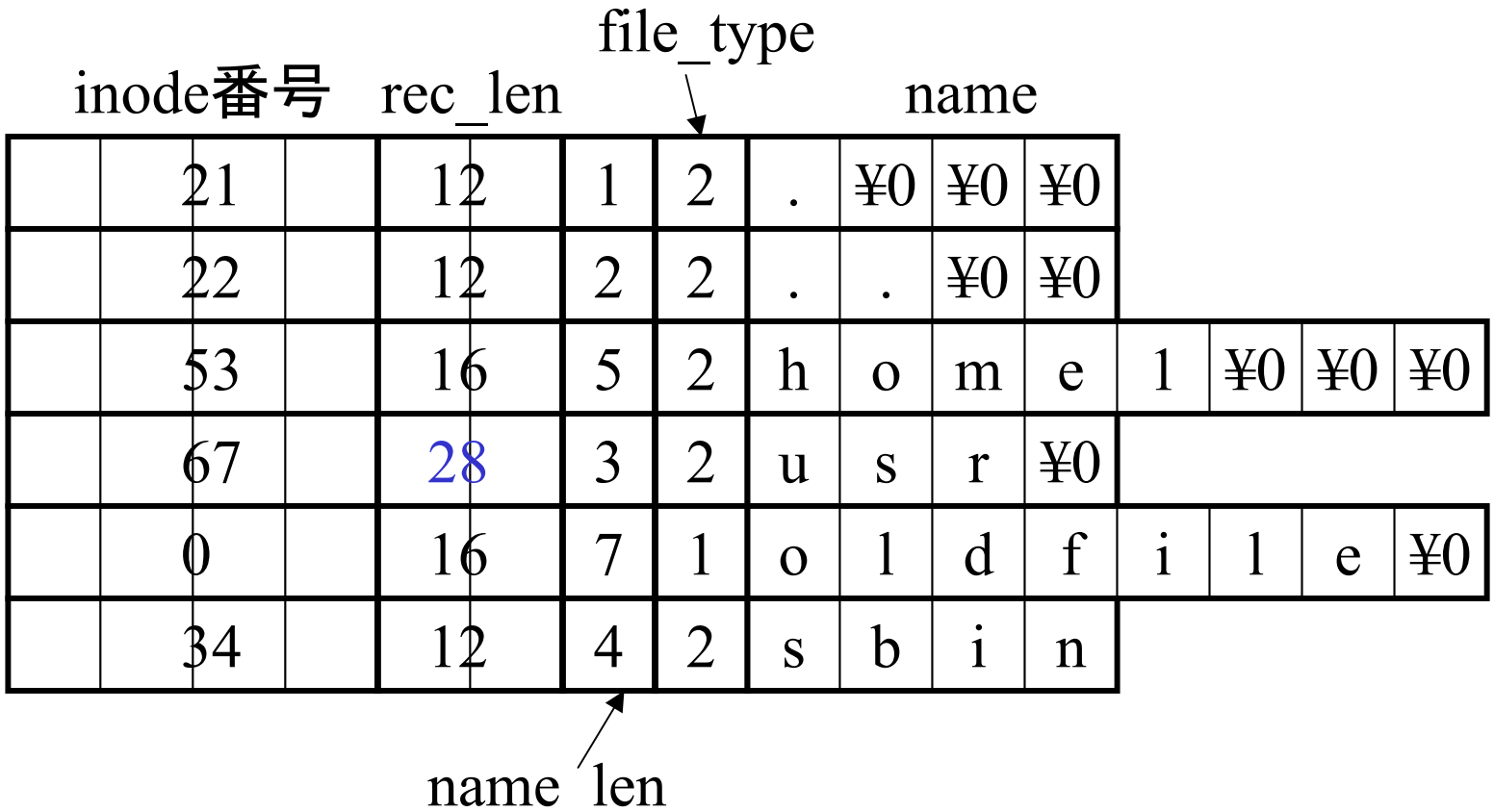
```
// include/linux/ext2_fs.h の501行目

struct ext2_dir_entry_2 {
    __u32   inode; // そのディレクトリのiノード番号
    __u16   rec_len; // 構造体のサイズ, name[]のため可変
    __u8    name_len; // ファイル名の長さ ¥0 は含まず
    __u8    file_type;              // ファイルの型番号
    char    name[EXT2_NAME_LEN]; // ファイル名
};
```

通常, 255

効率化のため4の倍数長になっている. 不要な部分には ¥0 文字が詰めてあるはずなんだが..

例: とあるデータブロックの中身

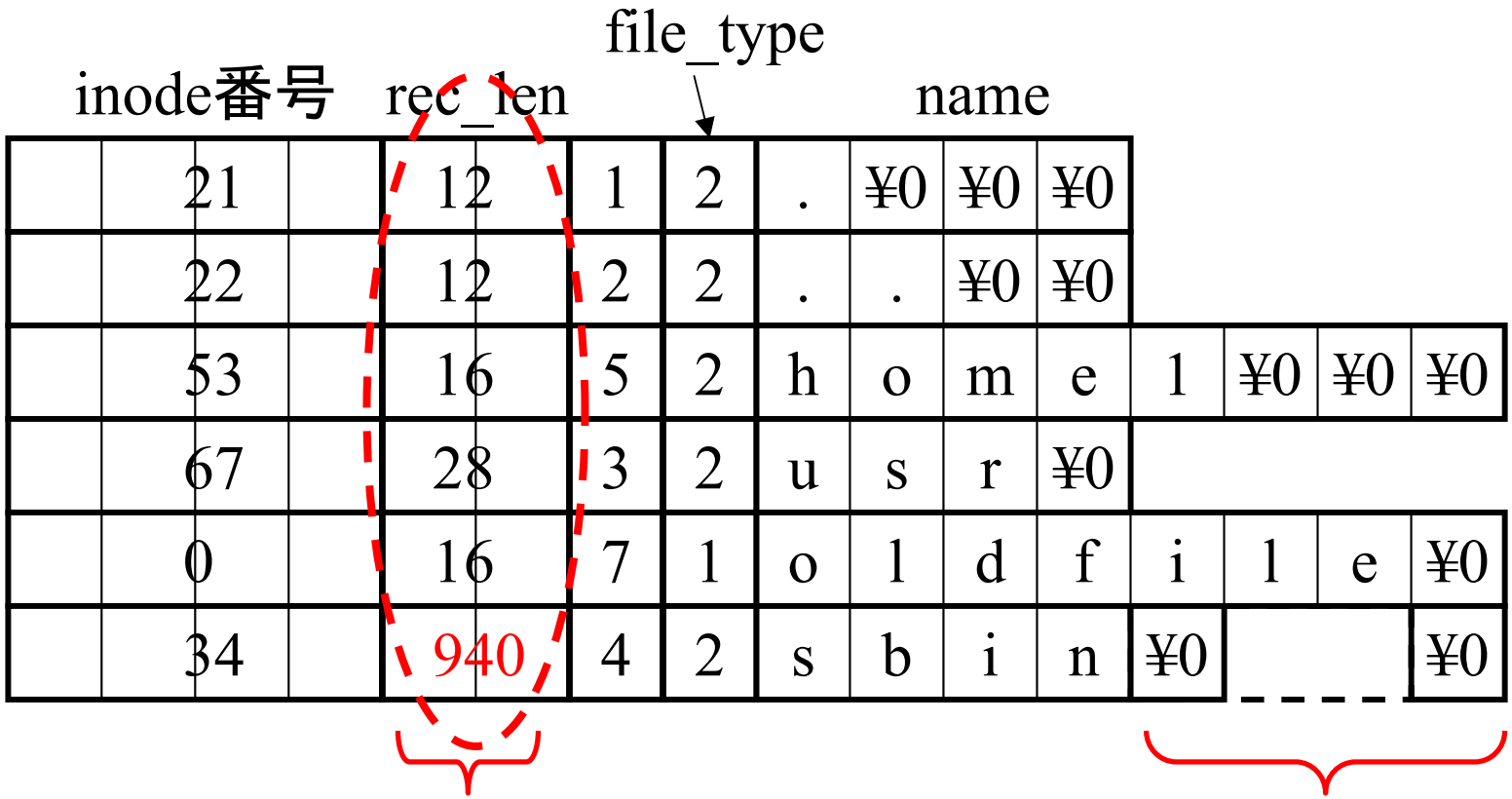


inode番号	rec_len	name_len	file_type	name
21	12	1	2	. ¥0 ¥0 ¥0
22	12	2	2	. . ¥0 ¥0
53	16	5	2	h o m e 1 ¥0 ¥0 ¥0
67	28	3	2	u s r ¥0
0	16	7	1	o l d f i l e ¥0
34	12	4	2	s b i n

ディレクトリ・ブロックの終端は？

- inodeにも，ディレクトリデータが入ったブロックにも，ディレクトリ数のデータは入っていない.
- じゃ，どうやってディレクトリのリストの末尾をみつけるかというと・・・・ 次のページ参照.
- 結果として，各エントリのサイズの合計がブロックサイズに一致した時が末尾ということになる.

例: とあるデータブロックの中身



全部足して，ブロックサイズ(1024B)になるように最後のエントリのサイズが設定されている.

結果として，末尾に詰め物(Padding)がされている.

演習でのブロックの場合

```
inode = 12, (.)          (lec_len =  12)
inode =  2, (..)         (lec_len =  12)
inode = 21, (exercise1)  (lec_len =  20)
inode = 22, (os1.pdf)    (lec_len =  16)
inode = 23, (index.html) (lec_len =  20)
inode = 24, (chroot.txt) (lec_len =  20)
inode = 17, (image)      (lec_len = 924)
```

足して 1024B

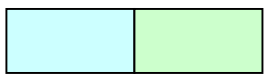
バイトオーダー問題

- 前述のようにBigEndianマシンの場合，提供したイメージがLittleEndianを想定しているので，バイトオーダー変換をしないといけない.
- オーダー(順序)がひっくり返るのは，int とか short とかの単純データ型内のみです.
  - 構造体の中全体がひっくり返っているわけではない.
- ま，自分でオーダーをひっくり返してもいいけど，こちらからひっくり返すマクロを提供した.

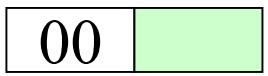
rev16

- #define rev16(x) (((x&0x00ff)<<8) | ((x&0xff00)>>8))
- 普通のマクロ，rev32もやり方は同じ.

元データx (16bit = 2byteを想定)

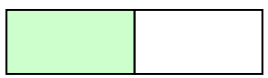


x & 0x00ff は以下



x & 0xff00 は以下

<<8 は8bit左シフト



>>8 は8bit右シフト

| は論理(というかbit)和

解答例1 (ex3.c)

- 基本的にはサンプルプログラムをベースにブロック単位にデータを読む方法ととる.

```
unsigned char* get_dir_entry(
    struct ext2_dir_entry_2* de, unsigned char* block, int* resta
)
```

blockで指定されたアドレスからext2_dir_extry_2構造体を切り出して，結果を *deに入れる.
blockのデータがあと何Bのこっているかのカウンタを *resta に保持.
返り値は次に構造体を読み出すべきアドレスの先頭．読みきった場合，NULLが返る.
de->nameに対しては終端文字を埋め込む．(実はちょっと危険)

解答例2 (ex3b.c)

- このファイルシステムは所詮，たった1.5Mしかないので，ちまちまブロック単位に読まず，一気に全データを配列blocksに読み込むプログラム.
  - ま，lseek()を使うというのもアリなんですが・・・・
- 実は演習4ではこのスタイルのほうが断然便利.

```
// 1024*1440 = 1474560
#define ALLBLOCKSIZE 1474560
#define BLOCKSIZE 1024
#define BLOCKNUMBER 1440

// ココに全部よみこんじゃう
unsigned char blocks[ALLBLOCKSIZE];

unsigned char* nth_block(int n){ // ブロック先頭を得る関数
   if(n<0 || BLOCKNUMBER<=n) return NULL; // out of range
   return blocks+(n*BLOCKSIZE);
}
```

機種依存問題について

バイトオーダー

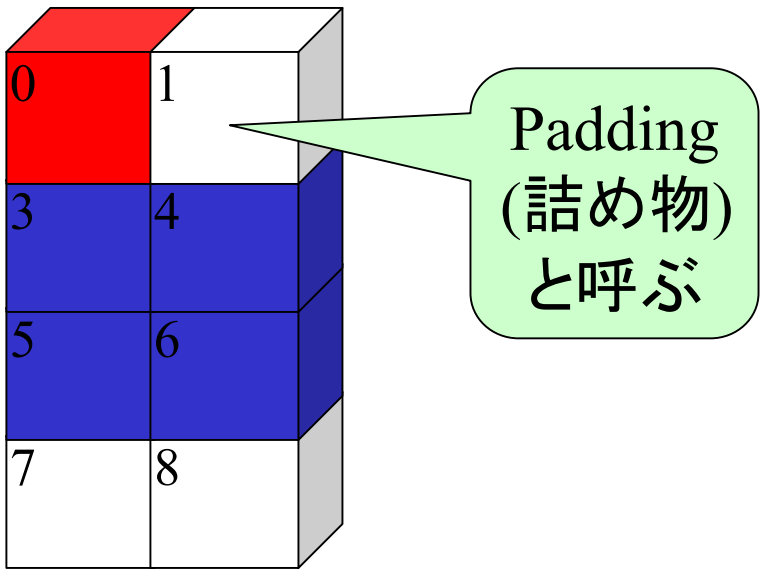
- 2バイト以上のデータを保存したり通信転送したりする場合，最上位のデータから送る場合と，最下位から送るかの違いがCPUによってある.
- 最上位: Big Endian と呼ぶ
  - PPC (Power PC, Macに搭載)のCPUはこっち
- 最下位: Little Endian と呼ぶ
  - i386 (WinPC)のCPUはこっち
- i386の場合は順序がひっくり返っている！

境界整列問題

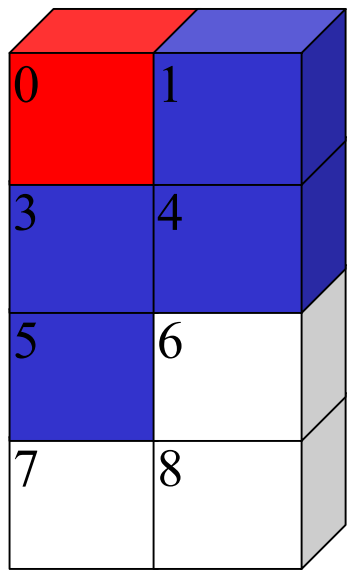
- データの境界が必ず2バイト単位になるように内部的に調整しているようなマシンがある. (e.g. mc68000等)
- その場合，この演習でやったbcopy()やmemcpy()を使った戦略は破綻する.

例 メモリ上の配列配置

```
struct funny{
    char flag;      long int value;
};  に対して,
```



とあるマシン(MPU)
たとえばMC6800



別のマシン(MPU)
たとえば8086