

## Lokalne systemy plików Linuksa

### W głąb struktur

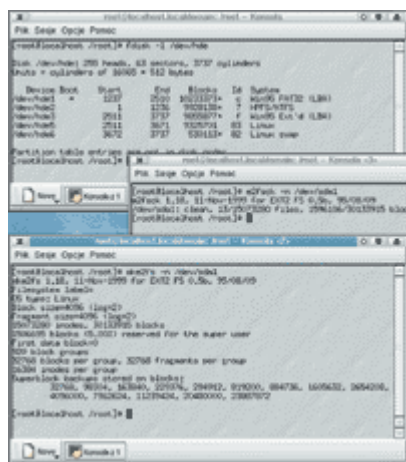
Autor: [Łukasz Spufiński](#)

Linux staje się systemem coraz bardziej skalowalnym, porównywalnym z komercyjnymi wersjami Uniksa oraz nowymi wydaniem Windows. W nowym jądrze zniesiono ograniczenie maksymalnej liczby procesów - teoretycznie zależy ona od rozmiaru pamięci operacyjnej, ale z analizy źródeł wynika, że na platformie Intel można, bez obawy o stabilność, utworzyć do 32 767 jednocześnie działających procesów. Ponadto zwiększono skalowalność SMP oraz w znacznej mierze zmodyfikowano obsługę plików.

W Linuksie za obsługę plików odpowiedzialny jest zestaw funkcji jądra o wspólnej nazwie VFS (Virtual File System). Do funkcji VFS odwołują się programy użytkownika, gdy chcą wykonać jakąkolwiek operację na pliku. Oczywiście nie muszą się one odwoływać bezpośrednio do tych funkcji, ale mogą korzystać z biblioteki glibc (GNU Library for C), która pośredniczy między programami użytkowników a skomplikowanymi funkcjami jądra.

Natomiast wirtualny system plików do realizacji operacji na pliku musi posłużyć się funkcjami konkretnego systemu plików, np. ext2 czy FAT. W ramach VFS odbywa się pierwszy etap buforowania plików - buforowanie nazw katalogów oraz i-węzłów.

### Od nazwy do tabeli plików procesu



Narzędzia ext2: fdisk (partycjonowanie dysków), mke2fs (formatowanie partycji ext2) i e2fsck (sprawdzanie partycji ext2)

Dobrym przykładem zastosowania systemu VFS jest funkcja systemowa `sys_open()` [plik źródłowy: `fs/open.c`]. W celu odnalezienia pliku na dysku odwołuje się ona do funkcji `filp_open(const char * filename, int flags, int mode)` [plik źródłowy: `fs/open.c`]. Z kolei ta funkcja odwołuje się do specjalnego, systemowego bufora nazw katalogowych (ang. directory entry cache). Rekord bufora nazw ma postać struktury widocznej na [listingu 1](#) [plik źródłowy: `include/linux/dcache.h`].

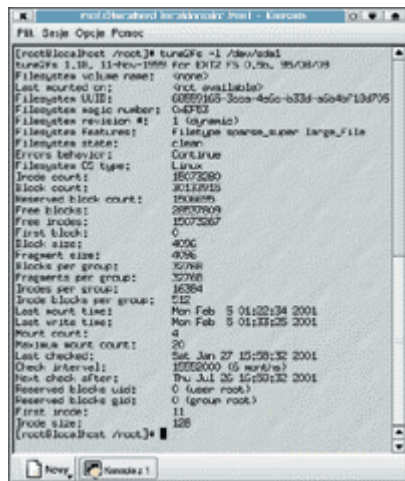
Zawartość bufora nazw jest odczytywana i aktualizowana poprzez wywołanie `open_namei(const char * pathname, int flag, int mode, struct nameidata *nd)` [plik źródłowy: `fs/namei.c`]. Dla już istniejącego pliku algorytm działania tej funkcji można sprowadzić do dwóch punktów:

1. Wywołanie `path_init(const char *name, unsigned int flags, struct nameidata *nd)`, czyli analiza ścieżki dostępu pod kątem zamontowanych systemów plików oraz właściwości procesu, który odwołał się do nazwy.
2. Wywołanie `path_walk(const char * name, struct nameidata *nd)`, czyli odczyt kolejnych poziomów drzewa katalogowego w celu dotarcia do docelowego pliku.

Druga z podanych funkcji korzysta z i-węzłów, czyli ze struktur, które są związane z docelowymi systemami plików. Konkretny i-węzeł opisuje pojedynczy obiekt w systemie plików, np. plik, katalog bądź skrót (link symboliczny).

`path_walk()` rozpoczyna poszukiwanie pliku od nazwy oraz i-węzła, który wskazał `path_init()` w zmiennej pośredniej `nd`. Jak można się domyślić, wykonywanie tego typu operacji ułatwia to, że wszystkie i-węzły wskaziwane przez

składowe d\_inode rekordów bufora nazw znajdują się w innym buforze i mają postać struktury z [listingu 2](#).



Tuning partycji ext2: tune2fs

Po znalezieniu pliku w buforze nazw funkcja `filp_open()` wywołuje funkcję `dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags)`, której zadaniem jest wypełnienie struktury typu `file` [plik źródłowy: `include/linux/fs.h`] - patrz [listing 3](#).

Jak widać, pozycja kursora w pliku (składowa `f_pos`) jest typu `loff_t`. Typ ten zależy od platformy [plik źródłowy: `include/asm-xxx/posix_types.h`]. W IA32 (i386) jest on zdefiniowany jako `long long`, a więc jako 64-bitowy. Stąd można uznać, że VFS Linuksa ma cechy 64-bitowego systemu plików.

Cała operacja kończy się zaktualizowaniem tabeli deskryptorów plików procesu przez wywołanie funkcji `fd_install()` [plik źródłowy: `include/linux/file.h`] - patrz [listing 4](#).

Jak sugeruje argument `fd` funkcji, w 32-bitowym systemie pojedynczy proces teoretycznie może otworzyć do 65 536 plików. Jednak przy standardowej konfiguracji jądra alokowana jest mniejsza pamięć - do 1024 deskryptorów na proces [plik źródłowy: `include/linux/fs.h`]:

```
#define INR_OPEN 1024
```

Liczba ta jest brana pod uwagę przy ustalaniu domyślnych limitów procesu [plik źródłowy: `include/asm-xxx/resource.h`] - [listing 5](#).

Dla każdego nowego zadania limity są przypisywane składowej `rlim` [plik źródłowy: `include/linux/sched.h`] - [listing 6](#). Sama tabela deskryptorów procesu jest przechowywana w składowej `files` jako struktura typu `files_struct` ([listing 7](#)).

## Wracając do i-węzłów...

Warto zwrócić uwagę na to, że znaczna część składowych struktur wirtualnego systemu plików to po prostu wskazania do funkcji implementowanych przez docelowy system plików, np. w strukturze typu `inode` wskazanie `i_op->lookup(struct inode *, struct dentry *)` udostępnia funkcję wyszukującą plik o zadanej nazwie w docelowym systemie plików.

Obsługa poszczególnych rodzajów systemów plików (`vfat`, `ext2`, `ntfs`, itd.) jest rejestrowana w formie jednokierunkowej listy ([listing 8](#)).

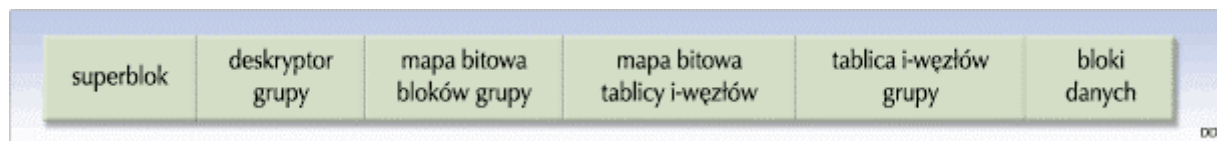
Składowa `super_block *(*read_super)(struct super_block *, void *, int)` to zdefiniowana w docelowym systemie plików funkcja, która zwraca strukturę typu `super_block` [plik źródłowy: `include/linux/fs.h`] - [listing 9](#).

Struktura ta szczegółowo definiuje docelowy system plików, który można zamontować w ramach VFS. Wykaz dostępnych systemów plików jest określony listą struktur typu `vfsmount` [plik źródłowy: `include/linux/mount.h`] - [listing 10](#).

## Po prostu ext2

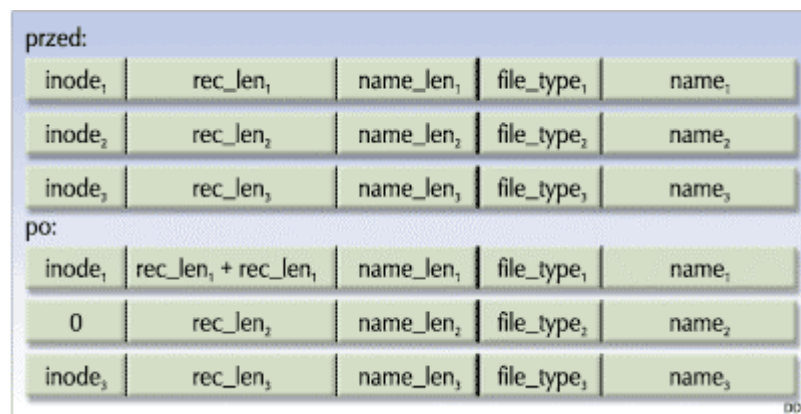
Podstawowym systemem plików Linuksa, a jednocześnie wzorcem strukturalnym dla VFS jest system `ext2`. Jego

projekt oparto na BSD FFS (Fast File System). Tak jak w FFS, partycja systemu plików ext2 została podzielona na mniejsze części. O ile jednak w BSD były to grupy cylindrów, to dla ext2 stworzono już strukturę mniej związaną z fizyczną warstwą systemu, tj. grupę bloków. W pierwszej wersji ext2 każda grupa bloków zawierała dodatkowy blok, tzw. superblok. W tym bloku zapisywane były informacje dotyczące całego systemu plików, np. liczba bloków w grupie, całkowita liczba bloków w systemie plików, rozmiar bloku i wersja ext2.



Grupa bloków ext2

Aktualne wydania systemu pozwalają zaoszczędzić miejsce na partycji i wymagają tylko, aby superblok znajdował się w pierwszej, drugiej, 3x+1, 5x+1 i 7x+1 grupie bloków. Superblok ext2 jest dokładnie zdefiniowany w pliku źródłowym `include/linux/ext2_fs.h` - [listing 11](#).



Zawartość katalogu przed i po usunięciu wpisu

Dla samej grupy ważniejsze od superbloku są następujące po nim, a przed konkretnymi blokami danych elementy:

- deskryptor grupy (patrz [listing 12](#));
- mapa bitowa grupy (zajętość bloków w grupie);
- mapa bitowa tablicy i-węzłów (zajętość i-węzłów w tablicy i-węzłów w grupie);
- tablica i-węzłów grupy.

Wszystkie mapy w grupie mają rozmiar jednego bloku, tak więc mapa może opisywać maksymalnie tyle obiektów (np. bloków), ile wskazuje stała rozmiar\_bloku\_w\_bitach. Blok Ext2 ma taki sam rozmiar w całym fizycznym systemie plików i może wynosić 1, 2 lub 4 kB. Bloków może być maksymalnie 232, a więc maksymalny rozmiar systemu plików Ext2 to  $(232) \times 4 \text{ kB} = 16 \text{ TB}$ . Zakładając, że grupa bloków może zawierać maksymalnie rozmiar\_bloku\_w\_bitach bloków, maksymalny rozmiar grupy może wynieść 128 MB (4096 bajtów  $\times$  8 bitów daje 32 768 - jest to liczba bloków, które można opisać w mapie bitowej; każdy blok może mieć 4096 bajtów, stąd wynika maksymalny rozmiar grupy).

Po mapie bitowej bloków grupy znajduje się mapa bitowa i-węzłów. Mapa ta opisuje tablicę i-węzłów rozpoczynającą się w następnym bloku grupy. Tablica i-węzłów zawiera - jak sama nazwa wskazuje - i-węzły, czyli praktyczną realizację obiektów (plików, katalogów) znanych z VFS. Obecnie każdy i-węzeł zajmuje 128 bajtów (patrz [listing 13](#)), a więc przy mapie bitowej mającej 32 768 pozycji tablica i-węzłów zajmuje 4 MB ze 128 MB grupy (co można potraktować jako ok. 3-procentową stratę pojemności dysku).

Tablica wykorzystywanych bloków to - zgodnie z ideą i-węzłów - bezpośrednie wskazania do 12 bloków danych, jedno wskazanie do bloku pośredniego (zamiast danych zawiera wskazania do kolejnych bloków), jedno wskazanie do bloku podwójnie pośredniego (zawierającego wskazania do bloków pośrednich) oraz jedno wskazanie do bloku potrójnie pośredniego (zawierającego wskazania do bloków podwójnie pośrednich). Wszystkie te wskazania są 32-bitowe, a więc pojedynczy obiekt (katalog, plik) może mieć rozmiar  $(12 + (\text{rozmiar\_bloku}/4) + (\text{rozmiar\_bloku}/4)^2 + (\text{rozmiar\_bloku}/4)^3) \times \text{rozmiar\_bloku}$ , co teoretycznie - przy bloku o rozmiarze 4096 bajtów - daje  $(12 + 1024 + 1\,048\,576 + 1\,073\,741\,000) \times 4096 = 4\,402\,345\,000\,000$  bajtów = 4,1 TB.

Pierwszy i-węzeł w systemie plików dotyczy oczywiście katalogu głównego (root). W ext2 katalogi występują w

formie jednokierunkowej listy struktur - patrz [listing 14](#).

Względne położenie kolejnej pozycji na liście jest zapisywane w składowej `rec_len`. Usunięcie wpisu z katalogu polega na wyzerowaniu wartości `inode`. Wartość `rec_len` rekordu poprzedzającego jest powiększana o wartość `rec_len` aktualnie kasowanej pozycji. Później taka niewykorzystana przestrzeń jest zapelniana podczas operacji, np. wydłużania nazwy pliku/katalogu lub wstawiania nowej pozycji do katalogu.

### [archiwum.pckurier.pl](http://archiwum.pckurier.pl)

- [Elastyczne jadro Pingwina](#) (Kernel 2.4), nr 24/2000, str. 75, ID = 838
- [Bez fanfar](#) (Kernel 2.4), nr 2/2001, str. 12, ID = 4544
- [System na bazie](#) (system OIFS), nr 26/2000 str.92, ID = 4468
- [Zgodni sąsiedzi](#) (multiboot), nr 4/2000, str. 70, ID = 182
- [Skalowalne, wydajne i stabilne](#) (systemy 64-bitowe), nr 26/00, str. 52, ID = 917

W celu ograniczenia fragmentacji zewnętrznej (rozrzutu bloków związanych z tym samym i-węzłem) bloki przydziela się najbliższej danego i-węzła oraz jego pozostałych bloków. Algorytm przydziału bloków najpierw poszukuje w mapie bitowej grupy całego wolnego bajta (8 bloków), a dopiero później, gdy nie znajdzie takiej wolnej przestrzeni, szuka najbliższego wolnego bitu. Przy zamykaniu pliku bądź katalogu zarezerwowane "na wyrost" bloki są zwalniane. Sam dostęp do bloków danych odbywa się za pomocą kolejnej warstwy buforowania - a mianowicie buforów blokowych. Bufor taki jest opisany strukturą [plik źródłowy: `include/linux/fs,h`] widoczną na [listingu 15](#).

Jak wynika z tej struktury (położenie bloku na fizycznym dysku, czyli numer 512-bajtowego sektora, opisuje liczba typu `long`, a więc na platformie IA32 - 32-bitowa), bufor blokowy ogranicza rozmiar systemu plików do  $(232) \times 512$  bajtów = 2 TB.

W jednym z przyszłych numerów omówimy i porównamy bardziej zaawansowane systemy plików Linuksa (SGI XFS, IBM JFS, ReiserFS) oraz rozwiązania Software RAID i LVM (Linux Volume Manager).

==

## Listing 1

```
struct dentry {
    int d_count; /* licznik odwo•a• do nazwy */
    unsigned int d_flags; /* znaczniki AutoFS i NFS */
    struct inode * d_inode; /* do jakiego i-w•z•a odnosi si• nazwa */
    struct dentry * d_parent; /* katalog macierzysty nazwy */
    struct dentry * d_mounts; /* je•eli nazwa jest punktem montowania innego
        systemu plików, to jest to wskazanie na korze• zamontowanego systemu plików */
    struct dentry * d_covers; /* je•eli nazwa jest korzeniem zamontowanego systemu plików,
        to jest to wskazanie na katalog, pod którym zamontowano ten system plików */
    struct list_head d_alias; /* dwukierunkowa lista nazw odnosz•cych si•
        do tego samego i-w•z•a d_inode */
    struct qstr d_name; /* nazwa */
    unsigned long d_time; /* czas ostatniego odwo•ania do nazwy */
    struct dentry_operations *d_op; /* operacje na katalogach zdefiniowane
        w docelowym systemie plików */
    struct super_block * d_sb; /* wskazanie na superblok docelowego systemu plików */
    unsigned long d_reftime; /* czas od ostatniego odwo•ania do nazwy */
    void * d_fsdata; /* dane specyficzne dla docelowego systemu plików */
    unsigned char d_iname[DNAME_INLINE_LEN]; /* zast•puje d_name przy krótkich nazwach */
};
```

## Listing 2

```
struct inode {
    struct list_head i_hash, i_list, i_dentry, i_dirty_buffers;
    unsigned long i_ino;
    atomic_t i_count;
    kdev_t i_dev;
    umode_t i_mode;
    nlink_t i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    kdev_t i_rdev;
    loff_t i_size;
    time_t i_atime, i_mtime, i_ctime;
    unsigned long i_blksize, i_blocks, i_version;
    struct semaphore i_sem, i_zombie;
    struct inode_operations *i_op;
    struct file_operations *i_fop;
    struct super_block *i_sb;
    wait_queue_head_t i_wait;
};
```

```

struct file_lock *i_flock;
struct address_space *i_mapping;
struct address_space i_data;
struct dquot *i_dquot [MAXQUOTAS];
struct pipe_inode_info *i_pipe;
struct block_device *i_bdev;

unsigned long i_dnotify_mask;
struct dnotify_struct *i_dnotify;
unsigned long i_state;

unsigned int i_flags;
unsigned char i_sock;

atomic_t i_writecount;
__u32 i_generation;
union {
    struct minix_inode_info minix_i;
    struct ext2_inode_info ext2_i;
    ...
    struct proc_inode_info proc_i;
    struct socket socket_i;
    struct usbdev_inode_info usbdev_i;
    void *generic_ip;
} u;
};

```

### Listing 3

```

struct file {
    struct list_head f_list;
    struct dentry *f_dentry;
    struct vfsmount *f_vfsmnt;
    struct file_operations *f_op;
    atomic_t f_count;
    unsigned int f_flags;
    mode_t f_mode;
    loff_t f_pos;
    unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    struct fown_struct f_owner;
    unsigned int f_uid, f_gid;
    int f_error;
    unsigned long f_version;
    void *private_data;
};

```

### Listing 4

```

static inline void fd_install(unsigned int fd, struct file * file)
{
    struct files_struct *files = current->files;

    write_lock(&files->file_lock);
    if (files->fd[fd])
        BUG();
    files->fd[fd] = file;
    write_unlock(&files->file_lock);
}

```

### Listing 5

```

#define INIT_RLIMITS \
{ \
    { RLIM_INFINITY, RLIM_INFINITY }, \
    { RLIM_INFINITY, RLIM_INFINITY }, \
    { RLIM_INFINITY, RLIM_INFINITY }, \
    { _STK_LIM, RLIM_INFINITY }, \
    { 0, RLIM_INFINITY }, \
    { RLIM_INFINITY, RLIM_INFINITY }, \
    { 0, 0 }, \
    { INR_OPEN, INR_OPEN }, \
    { RLIM_INFINITY, RLIM_INFINITY }, \
    { RLIM_INFINITY, RLIM_INFINITY }, \
    { RLIM_INFINITY, RLIM_INFINITY } \
}

```

### Listing 6

```

struct task_struct {
    ...
    /* limits */
    struct rlimit rlim[RLIM_NLIMITS];
    unsigned short used_math;
    char comm[16];
    /* file system info */
    int link_count;
    struct tty_struct *tty; /* NULL if no tty */

```

```

    unsigned int locks; /* How many file locks are being held */
    /* filesystem information */
    struct fs_struct *fs;
    /* open file information */
    struct files_struct *files;
    ...
};

```

## Listing 7

```

#define NR_OPEN_DEFAULT BITS_PER_LONG
struct files_struct {
    atomic_t count;
    rwlock_t file_lock;
    int max_fds, max_fdset, next_fd;
    struct file ** fd; /* ostatnio wykorzystywany fragment tabeli */
    fd_set *close_on_exec, *open_fds, close_on_exec_init, open_fds_init;
    struct file * fd_array[NR_OPEN_DEFAULT];
};

```

## Listing 8

```

struct file_system_type {
    const char *name;
    int fs_flags;
    struct super_block *(*read_super) (struct super_block *, void *, int);
    struct module *owner;
    struct vfsmount *kern_mnt;
    struct file_system_type * next;
};

```

## Listing 9

```

struct super_block {
    struct list_head s_list;
    kdev_t s_dev;
    unsigned long s_blocksize;
    unsigned char s_blocksize_bits, s_lock, s_dirt;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dquot_operations *dq_op;
    unsigned long s_flags, s_magic;
    struct dentry *s_root;
    wait_queue_head_t s_wait;
    struct list_head s_dirty; /* lista fizycznych i-w•z•ów wymagaj•cych aktualizacji */
    struct list_head s_files;

    struct block_device *s_bdev;
    struct list_head s_mounts; /* lista zamontowanych systemów plików */
    struct quota_mount_options s_dquot; /* opcje dot. kwot */
    union {
        struct minix_sb_info minix_sb;
        struct ext2_sb_info ext2_sb;
        ...
        void *generic_sbp;
    } u;
};

```

## Listing 10

```

struct vfsmount {
    struct dentry *mnt_mountpoint; /* punkt montowania w buforze nazw */
    struct dentry *mnt_root; /* korze• w buforze nazw */
    struct vfsmount *mnt_parent; /* system plików, pod jakim ten system zosta• zamontowany */
    struct list_head mnt_instances;
    struct list_head mnt_clash;
    struct super_block *mnt_sb; /* superblok */
    struct list_head mnt_mounts;
    struct list_head mnt_child;
    atomic_t mnt_count;
    int mnt_flags;
    char *mnt_devname; /* nazwa urz•dzenia, na którym znajduje si•
        system plików, np. /dev/hda5 */
    struct list_head mnt_list;
    uid_t mnt_owner;
};

```

## Listing 11

```

struct ext2_super_block {
    __u32 s_inodes_count; /* liczba i-w•z•ów */
    __u32 s_blocks_count; /* liczba bloków */
    __u32 s_r_blocks_count; /* liczba bloków zarezerwowanych dla wybranego u•ytkownika */
    __u32 s_free_blocks_count; /* liczba wolnych bloków */
    __u32 s_free_inodes_count; /* liczba wolnych i-w•z•ów */
    __u32 s_first_data_block; /* pierwszy blok danych */

```

```

__u32 s_log_block_size; /* rozmiar bloku */
__s32 s_log_frag_size; /* rozmiar fragmentu */
__u32 s_blocks_per_group; /* liczba bloków w grupie */
__u32 s_frags_per_group; /* liczba fragmentów w grupie */
__u32 s_inodes_per_group; /* liczba i-w•z•ów w grupie */
__u32 s_mtime; /* czas ostatniego montowania */
__u32 s_wtime; /* czas ostatniego zapisu */
__u16 s_mnt_count; /* licznik montowa• */
__s16 s_max_mnt_count; /* maksymalna warto• licznika montowa• */
__u16 s_magic;
__u16 s_state; /* stan systemu plików */
__u16 s_errors;
__u16 s_minor_rev_level; /* podwersja systemu plików */
__u32 s_lastcheck; /* czas ostatniej weryfikacji systemu plików */
__u32 s_checkinterval; /* maksymalny odst•p pomi•dzy weryfikacjami */
__u32 s_creator_os; /* system operacyjny, pod jakim utworzono system plików */
__u32 s_rev_level; /* wersja systemu plików */
__u16 s_def_resuid; /* domy•lny id ...*/
__u16 s_def_resgid; /* ...i domy•lny gid u•ytkownika dla rezerwowych bloków */
};
...
};

```

## Listing 12

```

struct ext2_group_desc {
    __u32 bg_block_bitmap; /* blok mapy bit. bloków grupy */
    __u32 bg_inode_bitmap; /* blok mapy bit. tablicy i-w•z•ów grupy */
    __u32 bg_inode_table; /* I blok tablicy i-w•z•ów */
    __u16 bg_free_blocks_count; /* l. wolnych bloków w grupie */
    __u16 bg_free_inodes_count; /* l. wolnych i-w•z•ów w grupie */
    __u16 bg_used_dirs_count; /* l. katalogów w grupie */
    __u16 bg_pad;
    __u32 bg_reserved[3];
};

```

## Listing 13

```

/*
 * Constants relative to the data blocks
 */
#define EXT2_NDIR_BLOCKS 12
#define EXT2_IND_BLOCK EXT2_NDIR_BLOCKS
#define EXT2_DIND_BLOCK (EXT2_IND_BLOCK + 1)
#define EXT2_TIND_BLOCK (EXT2_DIND_BLOCK + 1)
#define EXT2_N_BLOCKS (EXT2_TIND_BLOCK + 1)
struct ext2_inode {
    __u16 i_mode; /* rodzaj */
    __u16 i_uid; /* m•odsze 16-bitów w•a•ciciela */
    __u32 i_size; /* rozmiar */
    __u32 i_atime; /* czas ostatniego dost•pu */
    __u32 i_ctime; /* czas utworzenia */
    __u32 i_mtime; /* czas ostatniej modyfikacji */
    __u32 i_dtime; /* czas usuni•cia */
    __u16 i_gid; /* m•odsze 16-bitów grupy */
    __u16 i_links_count; /* licznik dowi•za• */
    __u32 i_blocks; /* licznik bloków */
    __u32 i_flags; /* znaczniki pliku */
    union {
        struct {
            __u32 l_i_reserved1;
        } linux1;
        struct {
            __u32 i_block[EXT2_N_BLOCKS]; /* wskazania do bloków */
            __u32 i_generation; /* wersja pliku (NFS) */
            __u32 i_file_acl; /* lista ACL pliku */
            __u32 i_dir_acl; /* lista ACL katalogu */
            __u32 i_faddr; /* adres fragmentu */
        } linux2;
    } osd1; /* pierwsza zale•no• od systemu operacyjnego */
    __u32 i_block[EXT2_N_BLOCKS]; /* wskazania do bloków */
    __u32 i_generation; /* wersja pliku (NFS) */
    __u32 i_file_acl; /* lista ACL pliku */
    __u32 i_dir_acl; /* lista ACL katalogu */
    __u32 i_faddr; /* adres fragmentu */
    union {
        struct {
            __u8 l_i_frag; /* numer fragmentu */
            __u8 l_i_fsize; /* rozmiar fragmentu */
            __u16 l_i_pad1, l_i_uid_high, l_i_gid_high;
            __u32 l_i_reserved2;
        } linux2;
    } osd2; /* druga zale•no• od systemu operacyjnego */
};

```

## Listing 14

```

#define EXT2_NAME_LEN 255
struct ext2_dir_entry_2 {
    __u32 inode; /* numer i-w•z•a */
    __u16 rec_len; /* rozmiar rekordu */
    __u8 name_len; /* d•ugo• nazwy */
    __u8 file_type;
    char name[EXT2_NAME_LEN]; /* nazwa pliku */
};

```

## Listing 15

```
struct buffer_head {
    struct buffer_head *b_next;
    unsigned long b_blocknr; /* numer bloku logicznego */
    unsigned short b_size; /* rozmiar bloku logicznego */
    unsigned short b_list;
    kdev_t b_dev; /* urz•dzenie blokowe */
    atomic_t b_count; /* u•ytkownicy korzystaj•cy z bloku */
    kdev_t b_rdev; /* urz•dzenie fizyczne */
    unsigned long b_state; /* mapa bitowa buforu */
    unsigned long b_flushtime; /* czas, po jakim blok powinien by• zaktualizowany na dysku */

    struct buffer_head *b_next_free;
    struct buffer_head *b_prev_free;
    struct buffer_head *b_this_page; /* lista buforów znajduj•cych si•
        na tej samej stronie pamieci */
    struct buffer_head *b_regnext; /* lista •da• o bufor */
    struct buffer_head **b_pprev;
    char * b_data; /* dane bloku */
    struct page *b_page; /* strona pamieci bufora */
    void (*b_end_io)(struct buffer_head *bh, int uptodate); /* funkcja
        aktualizacji bloku na dysku */
    void *b_private;
    unsigned long b_rsector; /* po•o•. bloku na fizycz. dysku */
    wait_queue_head_t b_wait;

    struct inode * b_inode;
    struct list_head b_inode_buffers; /* dwukierunkowa
        lista bloków z i-wz•ami, wymagaj•cymi aktualizacji na dysku */
};
```