WIRTUALNY SYSTEM PLIKÓW VFS

Plan wykładu

- Wirtualny system plików (VFS) – podstawowe koncepcje
- Struktury danych VFS
  - Obiekty superbloku
  - Obiekty i-węzła
  - Obiekty pliku
  - Obiekty pozycji w katalogu
- Rejestrowanie typu systemu plików w VFS
- Montowanie systemu plików w VFS
- Przykład działania VFS

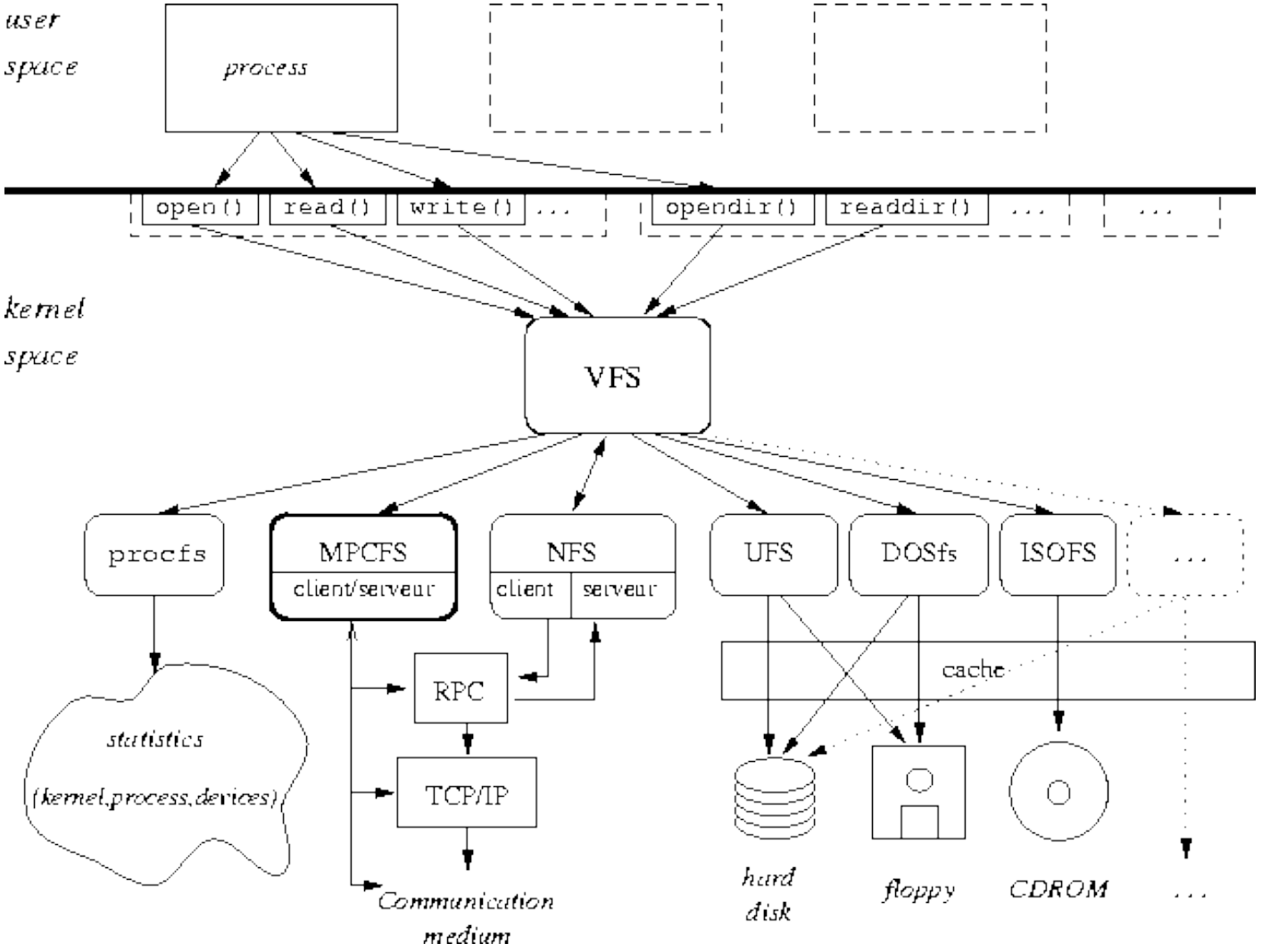
Wirtualny system plików (VFS) – podstawowe koncepcje

- **Wirtualny system plików (VFS) – podstawowe koncepcje**
- Struktury danych VFS
  - Obiekty superbloku
  - Obiekty i-węzła
  - Obiekty pliku
  - Obiekty pozycji w katalogu
- Rejestrowanie typu systemu plików w VFS
- Montowanie systemu plików w VFS
- Przykład działania VFS

Po co kolejna warstwa pośrednia?

- Dla użytkownika system plików systemu Linux wygląda jak hierarchiczne drzewo katalogów o semantyce systemu Unix.
- Wewnątrz systemu Unix zastosowano warstwę abstrakcji, nazwaną wirtualnym systemem plików **VFS** (ang. *Virtual File System*), pozwalającą na zarządzanie różnymi systemami plików.
- **VFS** dostarcza jednolity interfejs wspólny dla wszystkich systemów plików obsługiwanych przez jądro systemu operacyjnego.
- Początkowo wirtualny system plików został wprowadzony w **BSD** w celu obsługi **NFS** (ang. *Network File System*) – opartego o **UDP** protokołu zdalnego udostępniania systemu plików.

Wirtualny system plików VFS

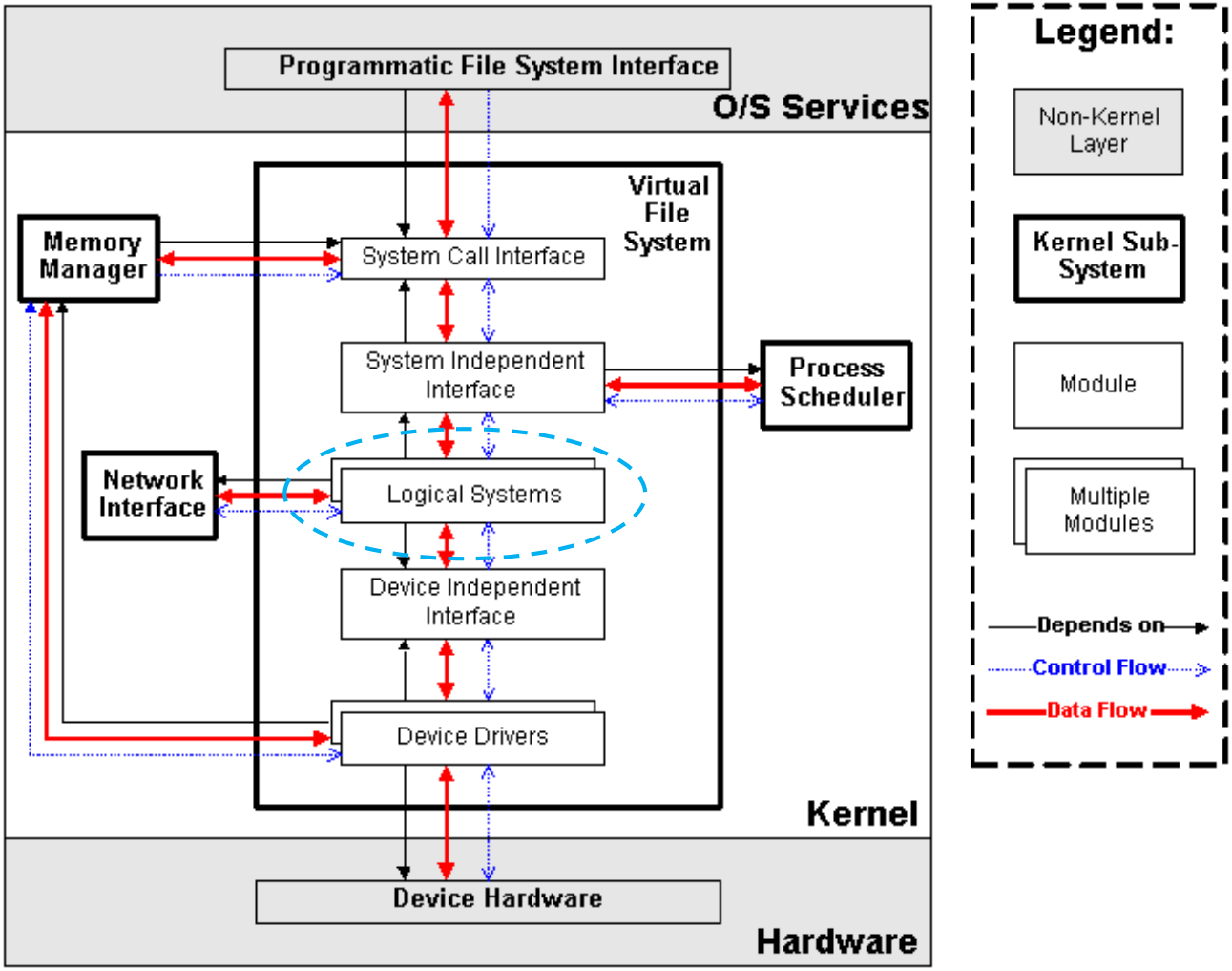


Klasy obsługiwanych systemów plików

Główne klasy systemów plików obsługiwanych przez VFS:

1. **Dyskowe systemy plików**: natywne systemy Linux (**Ext3**, **Ext4**, **ReiserFS**), warianty systemu UNIX (**System V**, **BSD**, **MINIX**), systemy Microsoft (**VFAT**, **NTFS, exFAT**), CD-ROM (**ISO9660**) i inne **HPFS** (OS/2), **HFS** (Apple), **FFS** (Amiga).
2. **Sieciowe systemy plików**: **NFS**, **Coda**, **AFS** (Andrew's filesystem), **SMB** (Microsoft Windows), **NCP** (Novell NetWare).
3. **Specjalne systemy plików**: inaczej systemy **wirtualne**, nie korzystają z przestrzeni dyskowej, np. system plików **/proc** dostarcza prostego interfejsu, który pozwala na dostęp do zawartości podstawowych struktur jądra.

Architektura VFS



Ivan Bowman, *Conceptual Architecture of the Linux Kernel*

Wspólny model plików

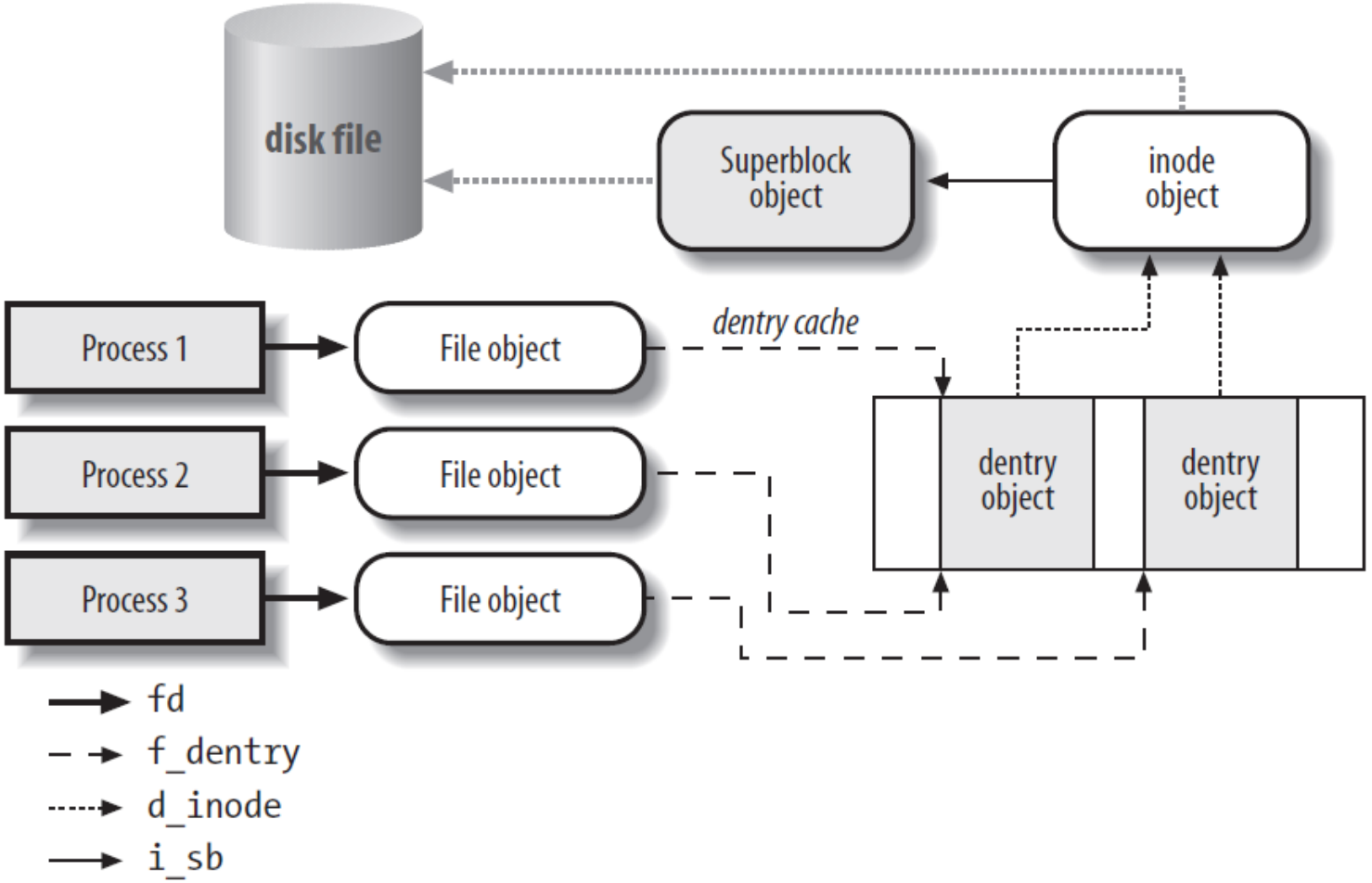
VFS został zaprojektowany na zasadach **obiektowych** i ma dwie składowe:

1. **zbiór definicji** opisujących obiekty; w VFS zdefiniowano cztery podstawowe typy obiektów:
   - **obiekt super bloku** (ang. *super-block object*), przechowujący informacje dotyczące zamontowanego systemu plików,
   - **obiekt i-węzła** (ang. *inode-object*), przechowujący informacje o danym pliku
   - **obiektu pliku** (ang. *file-object*) reprezentujące powiązanie między otwartym plikiem a procesem
   - **obiekt wpisu w katalogu** (ang. *dentry-object*) reprezentujący powiązania pozycji katalogu z odpowiednim plikiem
2. **warstwę oprogramowania** do działań na takich obiektach

Zadaniem VFS jest udostępnianie **i-węzłów**

- identyfikacja i-węzła: (system plików, numer i-węzła)
- VFS definiuje operacje katalogowe na obiekcie i-węzła a nie na obiekcie pliku

Powiązania procesów z obiektami VFS (w pamięci operacyjnej)



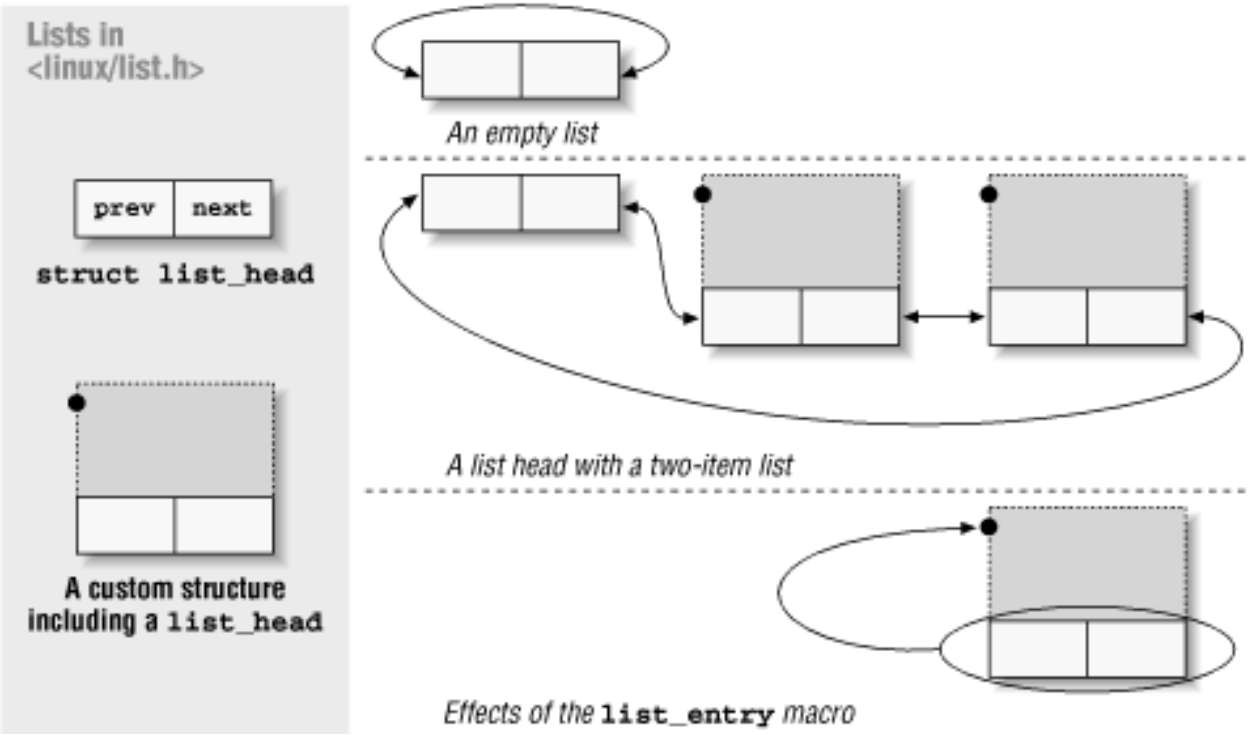
Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*

Relacje pomiędzy obiektami VFS

- Wpis w katalogu a i-węzeł
  - wpis istnieje jedynie w pamięci jądra
  - i-węzeł może znajdować się na dysku, ale jest ładowany do pamięci i tam dostępny (jakiekolwiek zmiany jego zwartości powinny być zapisane z powrotem na dysku)
  - plik (i-węzeł) może mieć wiele zapisów w katalog (np. dowiązań sztywnych)

- Listy (gł. dwukierunkowe) są podstawą wiązania obiektów tego samego typu w VFS
  - ciąg (łańcuch) obiektów tego samego typu dostępnych za pośrednictwem pól obiektów typu **struct list_head**
  - wskazanie na początek (i opcjonalnie koniec) listy za pomocą odpowiedniej zmiennej lub pola obiektu innego typu

Struktura danych **list_head**



Metody obiektów

Z **każdym obiektem** VFS związana jest **tabela operacji** (metod)

- każdy obiekt dostarcza zbioru operacji (w postaci wskaźników na funkcje)
- zwykle metody są **niezależne od typu systemu plików**, ale czasami charakterystyczne tylko dla systemu plików lub pliku

---

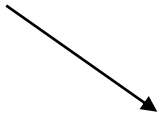
Interfejs warstwy VFS i specyficznego systemu plików

- warstwa VFS wywołuje te funkcje w przypadku konieczności wykonania operacji udostępnianych przez określony moduł systemu plików
- tabela operacji jest wypełniana wtedy, gdy jest ładowany lub inicjowany obiekt VFS i zawiera aktualne funkcje zaimplementowane w modułach systemu plików

Podstawowe wywołania systemowe obsługiwane przez VFS

System Call Name	Description
`mount( ) umount( )`	Mount/Unmount filesystems
`sysfs( )`	Get filesystem information
`statfs( ) fstatfs( ) ustat( )`	Get filesystem statistics
`chroot( )`	Change root directory
`chdir( ) fchdir( ) getcwd( )`	Manipulate current directory
`mkdir( ) rmdir( )`	Create and destroy directories
`getdents( ) readdir( ) link( ) unlink( ) rename( )`	Manipulate directory entries
`readlink( ) symlink( )`	Manipulate soft links
`chown( ) fchown( ) lchown( )`	Modify file owner
`chmod( ) fchmod( ) utime( )`	Modify file attributes
`stat( ) fstat( ) lstat( ) access( )`	Read file status
`open( ) close( ) creat( ) umask( )`	Open and close files
`dup( ) dup2( ) fcntl( )`	Manipulate file descriptors
`select( ) poll( )`	Asynchronous I/O notification
`truncate( ) ftruncate( )`	Change file size
`lseek( ) _llseek( )`	Change file pointer
`read( ) write( ) readv( ) writev( ) sendfile( )`	File I/O operations
`pread( ) pwrite( )`	Seek file and access it
`mmap( ) munmap( )`	File memory mapping
`fdatasync( ) fsync( ) sync( ) msync( )`	Synchronize file data
`flock( )`	Manipulate file lock

niektóre funkcje VFS nie wymagają odwoływania się do funkcji niższego poziomu



Struktury danych VFS - obiekt superbloku

- Wirtualny system plików (VFS) – podstawowe koncepcje
- **Struktury danych VFS**
  - **Obiekty superbloku**
  - Obiekty i-węzła
  - Obiekty pliku
  - Obiekty pozycji w katalogu
- Rejestrowanie typu systemu plików w VFS
- Montowanie systemu plików w VFS
- Przykład działania VFS

Superblok VFS

Superblok jest reprezentowany przez typ **struct super_block** zdefiniowany w pliku **include/linux/fs.h**. Zawiera podstawowe informacje o zamontowanym systemie plików i odpowiada fizycznemu superblokowi dysku.

Istotne pola:

s_list - dwukierunkowa lista wszystkich zamontowanych systemów plików.

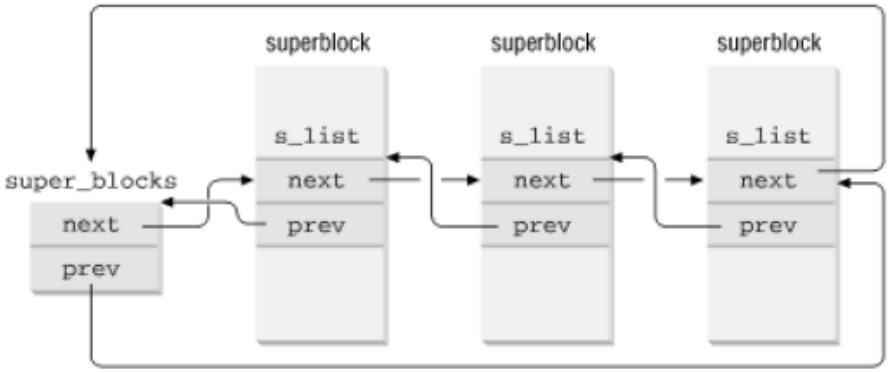
s_dev - urządzenie, na którym znajduje się ten system plików.

s_dirt - flaga ustawiana wtedy, gdy superblok jest modyfikowany i gaszona po zapisaniu superbloku na dysk.

s_dirty - lista zmodyfikowanych i-węzłów na liście **i_list**.

s_files - lista struktur typu **file** umieszczonych na liście **f_list** odpowiadających otwartym plikom tego systemu plików, czyli globalna (w tym systemie plików) tablica otwartych plików.

s_op – metody obiektu



Wskaźniki na pierwszy i ostatni element listy dostępne przez zmienną **super_blocks**

Opis ważniejszych pól

Type	Field	Description
struct list_head	s_list	Pointers for superblock list
kdev_t	s_dev	Device identifier
unsigned long	s_blocksize	Block size in bytes
unsigned char	s_blocksize_bits	Block size in number of bits
unsigned char	s_lock	Lock flag
unsigned char	s_rd_only	Read-only flag
unsigned char	s_dirt	Modified (dirty) flag
struct file_system_type *	s_type	Filesystem type
struct super_operations *	s_op	Superblock methods
struct dquot_operations *	dq_op	Disk quota methods
unsigned long	s_flags	Mount flags
unsigned long	s_magic	Filesystem magic number
unsigned long	s_time	Time of last superblock change
struct dentry *	s_root	Dentry object of mount directory
struct wait_queue *	s_wait	Mount wait queue
struct inode *	s_ibasket	Future development
short int	s_ibasket_count	Future development
short int	s_ibasket_max	Future development
struct list_head	s_dirty	List of modified inodes
union	u	Specific filesystem information

Metody superbloku

`read_inode()` ⟶ **`sb->s_op->read_inode(inode);`**

Zbiór operacji dla danego egzemplarza systemu plików

- **`read_inode( inode)`**: ładuje obiekt typu i-węzeł z dysku
- **`write_inode( inode)`**: aktualizuje obiekt typu i-węzeł na dysku
- **`put_inode( inode)`**: zwalnia dany obiekt typu i-węzeł (niekoniecznie usuwa z pamięci)
- **`delete_inode( inode)`**: usuwa i-węzeł i odpowiadający mu plik z dysku i pamięci
- **`put_super( super)`**: zwalania obiekt typu superblok
- **`write_super( super)`**: zapisuje zmiany na podstawie podanego argumentu
- .. i inne (patrz **struct super_operations** w **include/linux/fs.h**)

Operacje powinny być zależne od typu systemu plików, ustawia się je za pomocą metody **read_super()** obiektu **file_system_type**. Pola funkcji nie obsługiwanych przez system powinny być ustawione na **NULL**.

Metody superbloku - implementacja

```
struct super_operations {
      void (*read_inode) (struct inode *);
      void (*write_inode) (struct inode *);
      void (*put_inode) (struct inode *);
      void (*delete_inode) (struct inode *);
      void (*put_super) (struct super_block *);
      void (*write_super) (struct super_block *);
      int (*statfs) (struct super_block *,
                           struct statfs *, int);
      int (*remount_fs) (struct super_block *,
                           int *, char *);
      void (*clear_inode) (struct inode *);
      ....
   };
```

Dane zależne od typu systemu plików

Typ **union** `u` w **`struct super_block`**

```
#include <linux/minix_fs_sb.h>
#include <linux/ext2_fs_sb.h>
#include <linux/ext3_fs_sb.h>
...
struct super_block {
...
  union {
    struct minix_sb_info minix_sb;
    struct ext2_sb_info ext2_sb;
    struct ext3_sb_info ext3_sb;
    ...
    void *generic_sbp;
  } u;
...
}
```

Struktury danych VFS - obiekty i-węzła

- Wirtualny system plików (VFS) – podstawowe koncepcje
- **Struktury danych VFS**
  - Obiekty superbloku
  - **Obiekty i-węzła**
  - Obiekty pliku
  - Obiekty pozycji w katalogu
- Rejestrowanie typu systemu plików w VFS
- Montowanie systemu plików w VFS
- Przykład działania VFS

Węzły VFS

Struktury danych na poziomie jądra opisujące **rzeczywiste pliki** (lub katalogi)

- Każdy plik/katalog (znajdujący się na dysku) jest reprezentowany przez jeden **unikalny numer i-węzła** oraz zapis i-węzła na dysku
- Numer i-węzła nie zmienia się przez cały okres życia pliku

Aby uzyskać dostęp do pliku należy

- **Zaalokować** w pamięci jądra miejsce na obiekt i-węzła VFS
- **Załadować** zapis i-węzła z dysku (uproszczenie: zakładamy, że system jest typu indeksowego)

Pamięć podręczna i-węzłów

- W celu zapewnienia wysokiej wydajności ostatnio udostępniony (i zwolniony) i-węzeł jest przechowywany w **pamięci podręcznej**

Struktura danych węzła VFS

Obiekty i-węzła tworzone są na bazie struktury **struct inode**, zdefiniowanej w pliku **include/linux/fs.h**

Istotne pola:

- Numer i-węzła i wskaźnik na superblok: **i_ino**, **i_sb**
- Licznik odwołań (ile procesów otworzyło plik): **i_count**
- Informacja o pliku: **i_mode** (typ pliku i prawa dostępu), **i_nlink** (liczba sztywnych dowiązań), **i_uid**, **i_gid**, **i_size** (długość pliku w bajtach), **i_ atime** (czas ostatniego dostępu), **i_mtime** (czas ostatniego zapisu), **i_ctime** (czas ostatniej zmiany i-węzła), **i_blksize** (rozmiar bloku w bajtach), **i_blocks** (liczba bloków pliku)
- Wskaźnik na zbiór metod i-węzła: **i_op**
- Lista i-węzłów: **i_hash** (wskaźnik na tablicę mieszającą), **i_list**, **...**
- Lista wpisów w katalogach dla tego i-węzła: **i_dentry**

Opis ważniejszych pól (1/2)

Type	Field	Description
struct list_head	i_hash	Pointers for the hash list
struct list_head	i_list	Pointers for the inode list
struct list_head	i_dentry	Pointers for the dentry list
unsigned long	i_ino	inode number
unsigned int	i_count	Usage counter
kdev_t	i_dev	Device identifier
umode_t	i_mode	File type and access rights
nlink_t	i_nlink	Number of hard links
uid_t	i_uid	Owner identifier
gid_t	i_gid	Group identifier
kdev_t	i_rdev	Real device identifier
off_t	i_size	File length in bytes
time_t	i_atime	Time of last file access
time_t	i_mtime	Time of last file write
time_t	i_ctime	Time of last inode change
unsigned long	i_blksize	Block size in bytes
unsigned long	i_blocks	Number of blocks of the file
unsigned long	i_version	Version number, automatically incremented after each use

Opis ważniejszych pól (2/2)

unsigned long	i_nrpages	Number of pages containing file data
struct semaphore	i_sem	inode semaphore
struct semaphore	i_atomic_write	inode semaphore for atomic write
struct inode_operations *	i_op	inode operations
struct super_block *	i_sb	Pointer to superblock object
struct wait_queue *	i_wait	inode wait queue
struct file_lock *	i_flock	Pointer to file lock list
struct vm_area_struct *	i_mmap	Pointer to memory regions used to map the file
struct page *	i_pages	Pointer to page descriptor
struct dquot **	i_dquot	inode disk quotas
unsigned long	i_state	inode state flag
unsigned int	i_flags	Filesystem mount flag
unsigned char	i_pipe	True if file is a pipe
unsigned char	i_sock	True if file is a socket
int	i_writecount	Usage counter for writing process
unsigned int	i_attr_flags	File creation flags
__u32	i_generation	Reserved for future development
union	u	Specific filesystem information

Listy węzłów VFS

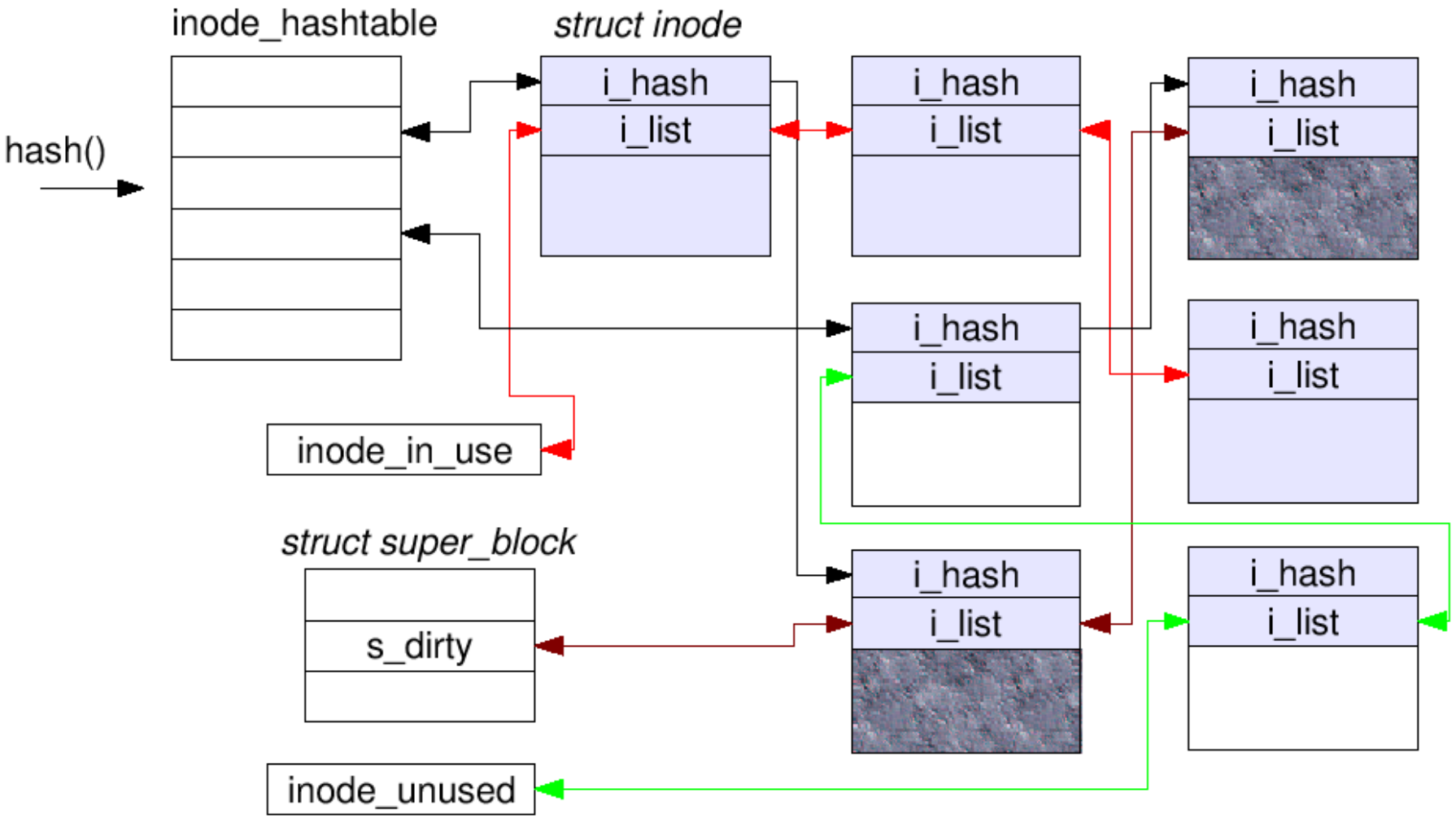
Tablica mieszająca i-węzłów (dla wszystkich i-węzłów w użyciu lub zmodyfikowanych), dostęp do początku listy przez zmienną **`inode_hashtable`**

- umożliwia szybkie poszukiwanie obiektu i-węzła na podstawie numeru i-węzła
- skrót dla każdego obiektu i-węzła obliczany jest na podstawie **i_sb** i **i_ino**
- lista skrótów będących w kolizji łączona jest za pomocą pola **i_hash**

Każdy i-węzeł znajduje się na jednej z trzech list (używanych, nieużywanych, zmodyfikowanych – „brudnych”)

- lista używanych i-węzłów: **i_count > 0** (początek i koniec listy w zmiennej **`inode_in_use`**)
- lista „brudnych” (ang. *dirty*), zmodyfikowanych i-węzłów: **i_count > 0** z ustawionym „brudnym” bitem w polu **i_state** (początek i koniec listy w polu **`s_dirty`** odpowiedniego superbloku)
- lista nie używanych i-węzłów: **i_count = 0** (początek i koniec listy w zmiennej **`inode_unused`**)
- poszczególne listy powiązane są za pomocą pola **`i_list`**
- dodatkowa lista od pola superbloku **`s_inodes`** łączona polami **`i_sb_list`**

Przykład list i-węzłów VFS



Metody i-węzłów

Zbiór zależnych od systemu plików operacji na i-węźle

- **`create()`**: utwórz nowy i-węzeł na dysku (dla nowego pliku)
- **`lookup()`**: przeszukaj katalog w celu znalezienia i-węzła odpowiadającego nazwie pliku
- **`link()`, `unlink()`**: utwórz/usuń dowiązanie twarde
- **`mkdir()`, `rmdir()`**: utwórz/usuń katalog
- **`symlink()`, `mknod()`**: utwórz i-węzeł dla dowiązania symbolicznego i pliku specjalnego
- .. i wiele innych (patrz **struct inode_operations** w pliku **include/linux/fs.h**)

Struktury danych VFS - obiekty pliku

- Wirtualny system plików (VFS) – podstawowe koncepcje
- **Struktury danych VFS**
  - Obiekty superbloku
  - Obiekty i-węzła
  - **Obiekty pliku**
  - Obiekty pozycji w katalogu
- Rejestrowanie typu systemu plików w VFS
- Montowanie systemu plików w VFS
- Przykład działania VFS

Obiekty plików VFS

Struktura danych na poziomie jądra opisująca **współdziałanie procesu z otwartym plikiem** (obiektem i-węzła). Obiekt pliku tworzony jest w oparciu o strukturę `struct file` (w pliku **include/linux/fs.h**)

Istotne pola

- **f_dentry**: obiekt wpisów w katalogach związany z tym plikiem
- **f_op**: wskaźnik na zbiór operacji na pliku
- **f_pos**: bieżący wskaźnik pliku (pozycja/przemieszczenie pliku)
- **f_count**: licznik odwołań do pliku (liczba dołączonych procesów)
- **f_list**: stosowane podczas połączenia tego pliku z jedną z wielu list
- **private_data**: wymagane przez sterownik terminala tty

Opis ważniejszych pól

f_list

Type	Field	Description
struct file *	f_next	Pointer to next file object
struct file **	f_pprev	Pointer to previous file object
struct dentry *	f_dentry	Pointer to associated dentry object
struct file_operations *	f_op	Pointer to file operation table
mode_t	f_mode	Process access mode
loff_t	f_pos	Current file offset (file pointer)
unsigned int	f_count	File object's usage counter
unsigned int	f_flags	Flags specified when opening the file
unsigned long	f_reada	Read-ahead flag
unsigned long	f_ramax	Maximum number of pages to be read-ahead
unsigned long	f_raend	File pointer after last read-ahead
unsigned long	f_ralen	Number of read-ahead bytes
unsigned long	f_rawin	Number of read-ahead pages
struct fown_struct	f_owner	Data for asynchronous I/O via signals
unsigned int	f_uid	User's UID
unsigned int	f_gid	User's GID
int	f_error	Error code for network write operation
unsigned long	f_version	Version number, automatically incremented after each use
void *	private_data	Needed for tty driver

Listy obiektów pliku VFS

- Obiekt plikowy VFS znajduje się na jednej z kilku list (lista obiektów **nie używanych**, lista obiektów „**w użyciu**”, lista **do otwarcia**)
  - Każda lista jest połączona za pomocą pola **`f_list`**
- Każdy superblok przechowuje listę otwartych plików
  - Z tego powodu nie można go zdemontować dopóki pozostanie otwarty przynajmniej jeden plik
  - Nagłówek listy zawarty jest w polu **`s_files`** superbloku (dawniej zmienna **`inuse_filps`**)
- Lista nieużywanych obiektów plików („recycling”)
  - Zmienna **`free_list`** w pliku **fs/file_table.c** (dawniej zmienna **`free_filps`**)
- Lista plików do otwarcia
  - Wtedy, gdy utworzono nowy obiekt plikowy, ale jeszcze go nie otworzono
  - Zmienna **`anon_list`** w pliku **fs/file_table.c**

Operacje na obiektach plikowych

- Zbiór zależnych od systemu plików operacji na pliku
  - **`open()`**: utwórz obiekt plikowy, otwórz plik i połącz go z odpowiadającym mu i-węzłem
  - **`read(), write()`**: czytaj plik, pisz do pliku
  - **`ioctl()`**: wyślij polecenie do odpowiedniego blokowego urządzenia sprzętowego
  - **`mmap()`**: odwzorowuj plik w pamięci w przestrzeni adresowej procesu
  - .. i wiele innych (patrz **struct file_operations** w pliku **include/linux/fs.h**)

Operacje plikowe

```
struct file_operations {
   struct module *owner;
   loff_t(*llseek)(struct file *, loff_t, int);
   ssize_t(*read)(struct file *, char *, size_t, loff_t *);
   ssize_t(*write)(struct file *, const char *, size_t, loff_ t
   *);
   int (*readdir)(struct file *, void *, filldir_t);
   unsigned int(*poll)(struct file *, struct poll_table_struct
   *);
   int (*ioctl)(struct inode *, struct file *, unsigned int,
                                                  unsigned
   long);
   int (*mmap)(struct file *, struct vm_area_struct *);
   int (*open)(struct inode *, struct file *);
   int (*flush)(struct file *);
   int (*relase)(struct inode *, struct file *);
   int (*fsync)(struct file *, struct dentry *, int datasync);
   int (*fasync)(int, struct file *, int);
   int (*lock)(struct file *, int, struct file_ lock *);
   …
```

Struktury danych VFS - obiekty pozycji w katalogu

- Wirtualny system plików (VFS) – podstawowe koncepcje
- **Struktury danych VFS**
  - Obiekty superbloku
  - Obiekty i-węzła
  - Obiekty pliku
  - **Obiekty pozycji w katalogu**
- Rejestrowanie typu systemu plików w VFS
- Montowanie systemu plików w VFS
- Przykład działania VFS

Wpis w katalogu VFS

Struktura danych na poziomie jądra opisująca **wpis w katalogu**:

- Zawiera informacje o **odwzorowaniu nazwy** (pliku, katalogu, itp.) **w numer** i-węzła
- Umożliwia dekodowanie opisu drzewa systemu plików
- Dostarcza mechanizmu wglądu w zawartość pliku w ramach danego systemu plików
- Każdy wpis wskazuje na i-węzeł

Pamięć podręczna wpisów w katalogu (ang. *dentry cache*)

- W celu zapewnienia wysokiej wydajności **ostatnio udostępniony** (i zwolniony) zapis w katalogu jest przechowywany w pamięci podręcznej)

Wpis w katalogu VFS - przykład

Próba dostępu do plik:

`/usr/bin/nano`

spowoduje utworzenie 4 obiektów wpisów katalogowych, po jednym dla każdego komponentu ścieżki:

- **`/`** (katalog, korzeń systemu, punkt montowania)
- **`usr`** (katalog)
- **`bin`** (katalog)
- **`nano`** (plik)

Struktury danych wpisów w katalogach VFS

Obiekt wpisu w katalogu VFS tworzony jest w oparciu o strukturę **`struct dentry`** (w pliku **include/linux/dcache.h**)

Istotne pola:

- wskaźnik na skojarzony i-węzeł: **d_inode**
- katalog rodzicielski: **d_parent**
- lista podkatalogów (jeżeli wpis jest katalogiem): **d_subdirs**
- lista katalogów na tym samym poziomie (jeżeli wpis jest katalogiem): **d_child**
- powiązanie tego obiektu z innymi listami: **d_hash**, **d_lru**
- wskaźnik na zbiór metod zapisów w katalogach: **d_op**
- użycie tego wpisu obiektu: **d_count, d_flags, ...**

Opis ważniejszych pól

Type	Field	Description
int	d_count	Dentry object usage counter
unsigned int	d_flags	Dentry flags
struct inode *	d_inode	Inode associated with filename
struct dentry *	d_parent	Dentry object of parent directory
struct dentry *	d_mounts	For a mount point, the dentry of the root of the mounted filesystem
struct dentry *	d_covers	For the root of a filesystem, the dentry of the mount point
struct list_head	d_hash	Pointers for list in hash table entry
struct list_head	d_lru	Pointers for unused list
struct list_head	d_child	Pointers for the list of dentry objects included in parent directory
struct list_head	d_subdirs	For directories, list of dentry objects of subdirectories
struct list_head	d_alias	List of associated inodes (alias)
struct qstr	d_name	Filename
unsigned long	d_time	Used by d_revalidate method
structdentry_operations*	d_op	Dentry methods
struct super_block *	d_sb	Superblock object of the file
unsigned long	d_reftime	Time when dentry was discarded
void *	d_fsdata	Filesystem-dependent data
unsigned char	d_iname[16]	Space for short filename

Metody zapisów w katalogu

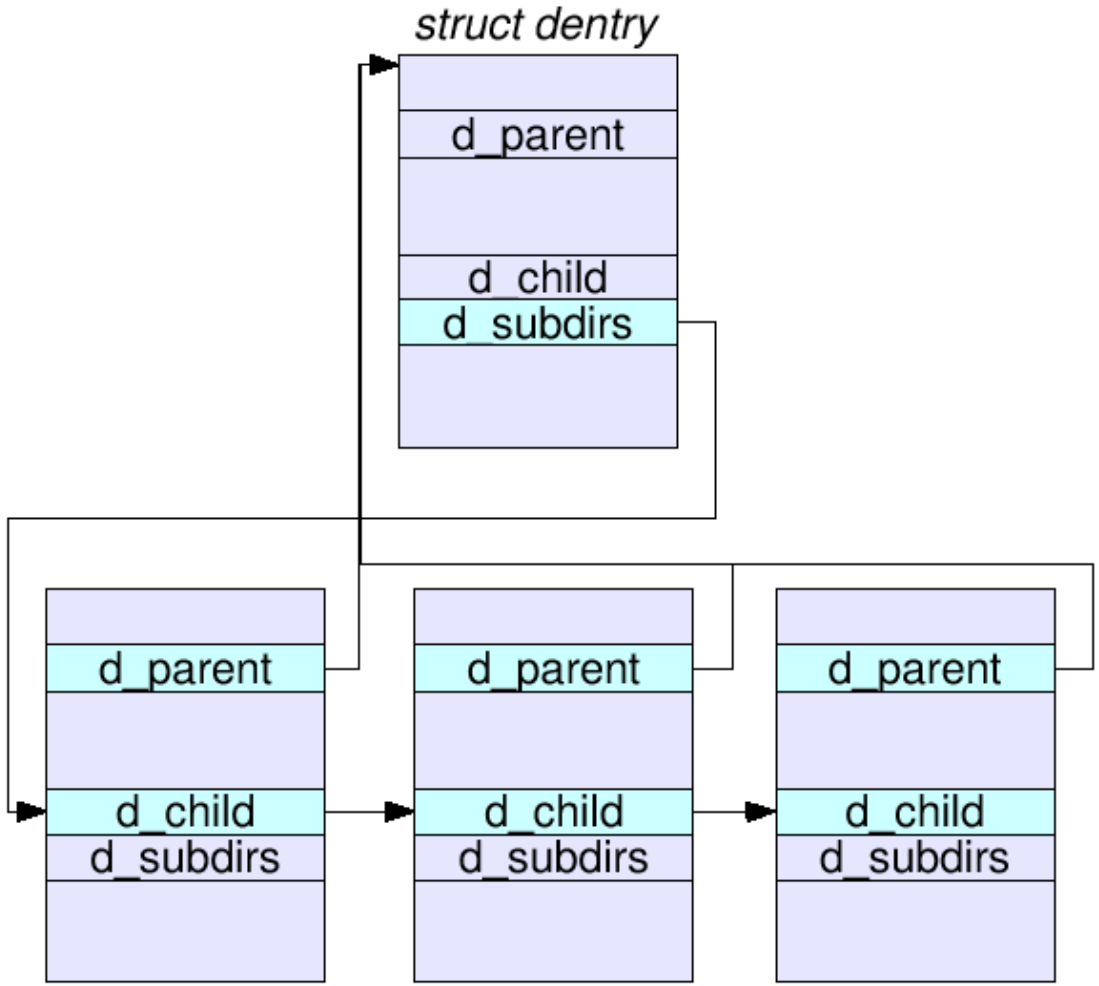
- Zbiór operacji na wpisach w katalogu zależnych od systemu plików
  - **`d_hash()`**: zwraca wartość skrótu dla tego zapisu (argumentem jest **nazwa** wpisu i **obiekt wpisu rodzica**)
  - **`d_compare()`**: porównywanie plików
  - **`d_delete()`**: wywoływana wtedy, gdy **`d_count`** staje się równe zero
  - .. i inne (patrz **struct dentry_operations** w pliku **include/linux/dcache.h**)

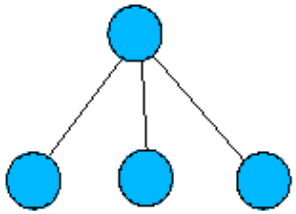
- Funkcje wspólne (niezależne od systemu plików) do obsługi zapisów w katalogach
  - **d_add(), d_alloc(), d_lookup()**, **...** (patrz plik **include/linux/dcache.h**)

Listy wpisów katalogów

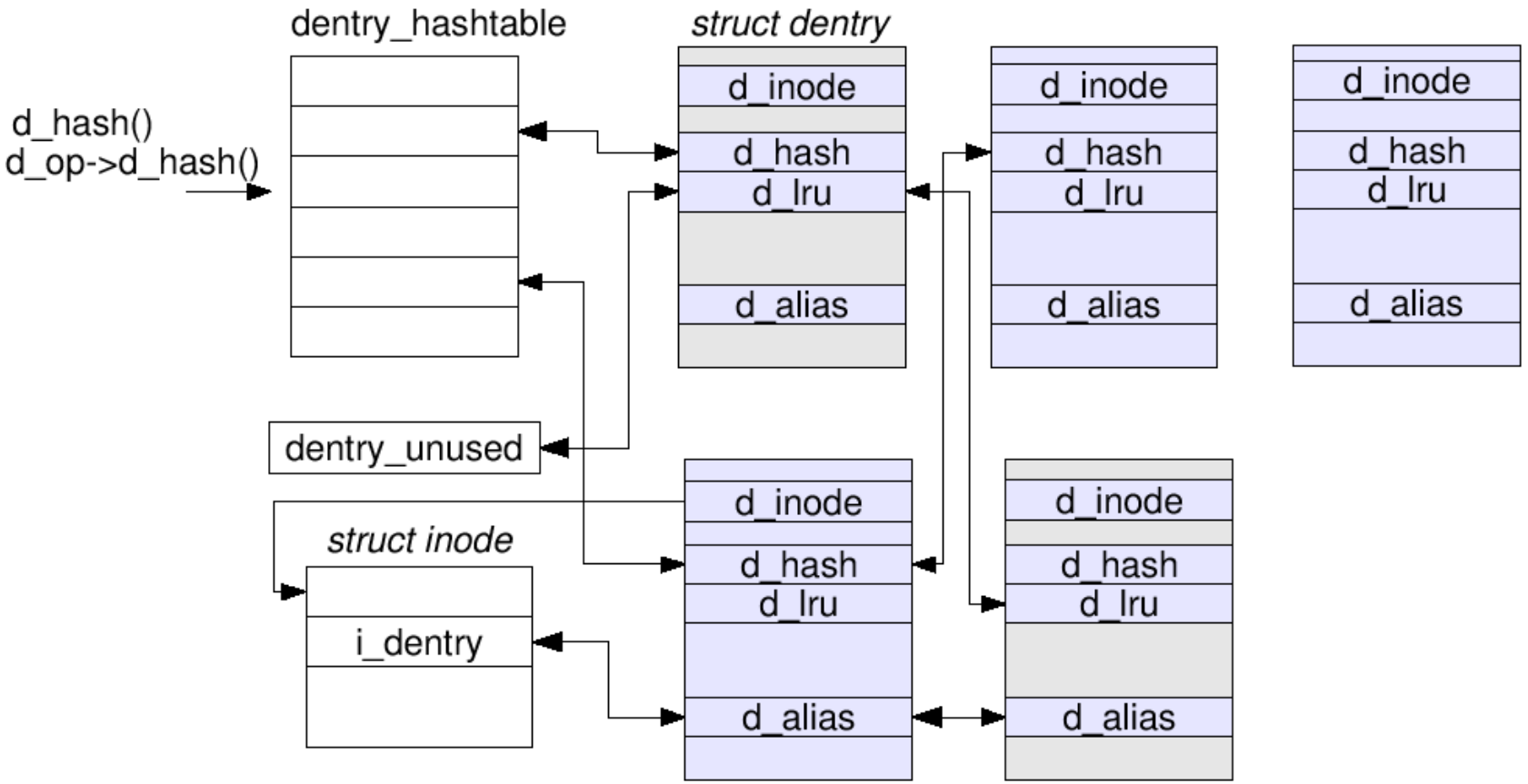
- Opis drzewa: wskaźnik na rodzica i lista potomków
  - Odpowiada opisowi zawartości katalogu
  - Za pomocą pól **d_parent**, **d_subdirs**, **d_child**
- Tabele mieszające wpisów katalogów
  - Szybki dostęp do obiektu opisu katalogu na podstawie znajomości nazwy pliku
  - Lista skrótów kolizyjnych jest łączona za pomocą pola **d_hash**
- Lista nieużywanych (wolnych) wpisów katalogów
  - Dostępna za pomocą pola **d_lru**
- Lista aliasów (te same i-węzły, inne wpisy katalogów)
  - Dostępna za pomocą pola **d_alias**

Listy zapisów w katalogach (postać drzewa)



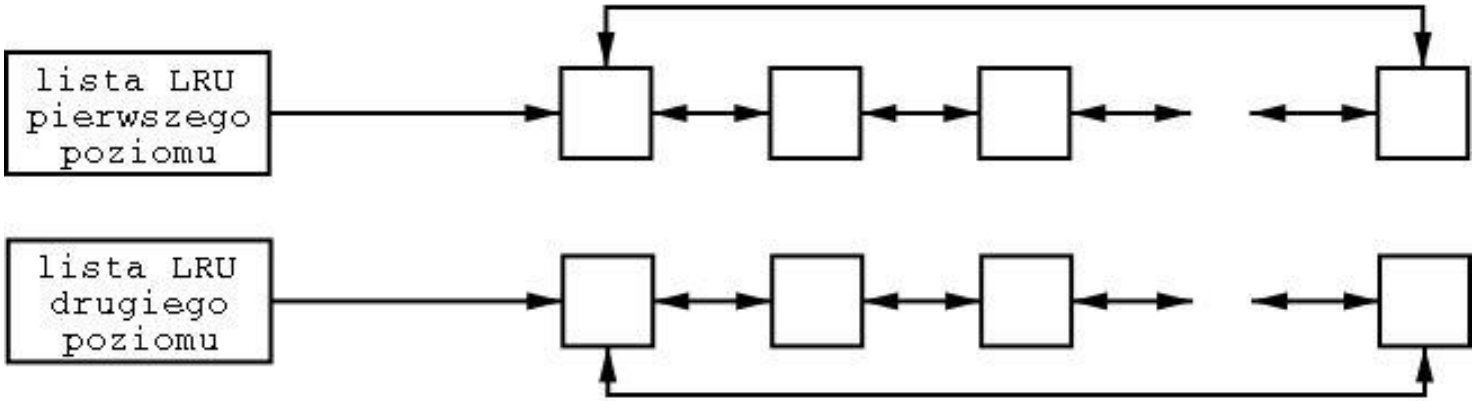


Lista zapisów w katalogach (skrót/wolny/alias)

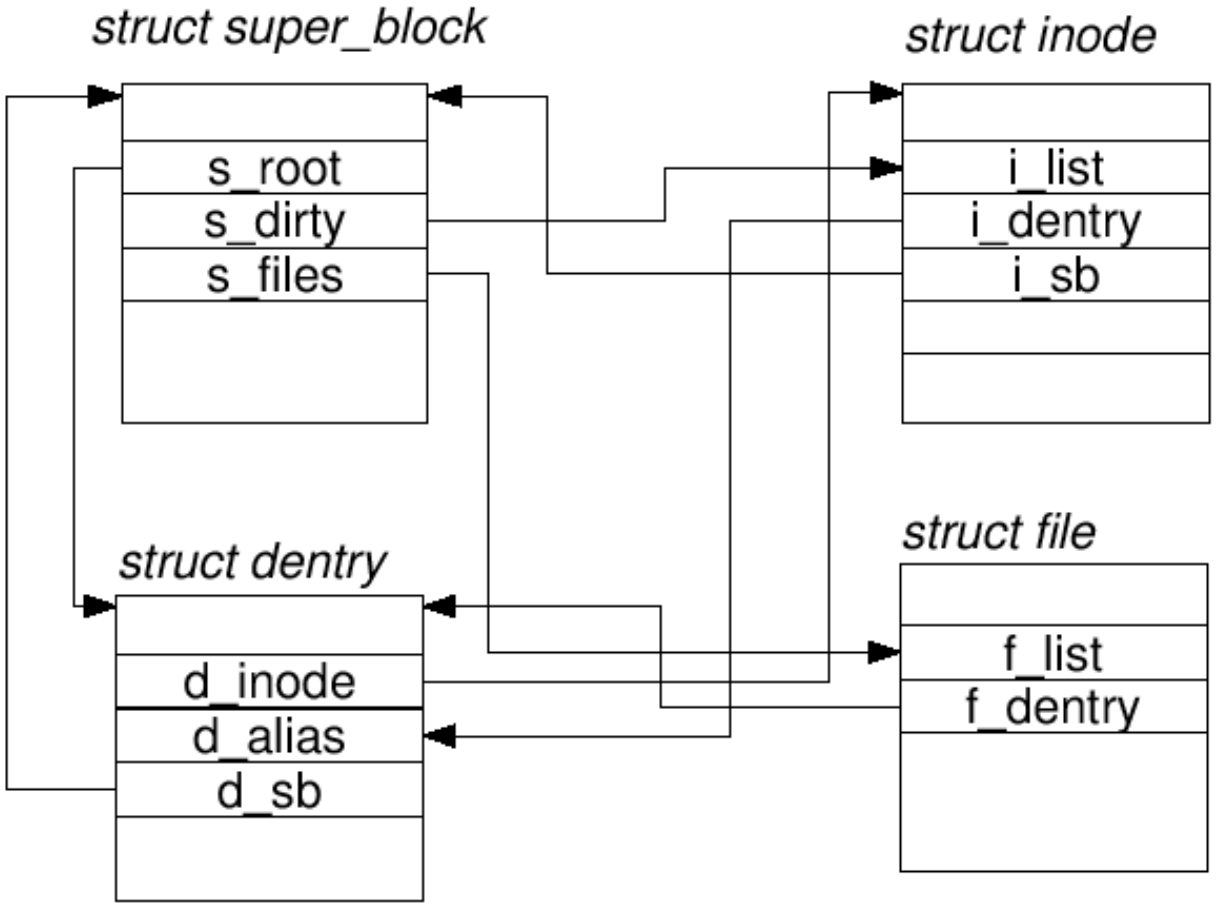


Listy LRU podręcznej pamięci buforowej zapisów w katalogach

- Do listy **pierwszego poziomu** są wstawiane odczytywane z dysku dane katalogowe (np. po wykonaniu polecenia ls).
- Natomiast w liście **drugiego poziomu** przechowywane są pozycje, których używamy częściej (np. nazwy plików, na których wykonaliśmy wc, cat).
- Element migruje z pierwszej listy do drugiej, jeśli odwołaliśmy się do niego (oznacza to, że prawdopodobnie będziemy go także potrzebować w przyszłości)
- W listach LRU elementy najczęściej używane są na końcu.



Podsumowanie list



Dostęp z poziomu struktur procesów (1/2)

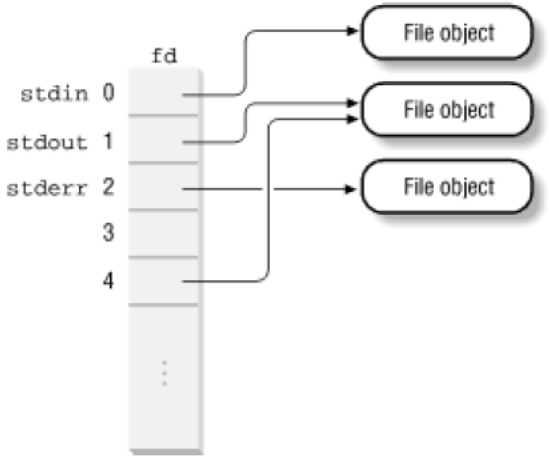
Struktura **`task_struct`** opisuje m.in. związek procesu z plikami:

1. pole **`struct fs_struct *fs`**
   - Zawiera m.in. dowiązanie do opisu bieżącego katalogu procesu (**pwd**) oraz korzenia systemu plików (**root**). Dzięki tym polom proces zna swój kontekst w systemie plików
   - Typ danych zdefiniowany w pliku **include/linux/fs_struct.h**
2. pole **`struct files_struct *files`**
   - Zawiera tablicę indeksowaną liczbami naturalnymi, które odpowiadają deskryptorom otwartych plików. Wartością pozycji tej tablicy jest dowiązanie do globalnej (w ramach systemu plików) tablicy otwartych plików
   - Ma ważne pole: **`struct file ** fd`** .Podczas tworzenia procesu alokuje się dla niego wstępnie **`fd_array`**[NR_OPEN_DEFAULT] na pierwsze 32 otwierane pliki. W razie potrzeby ta tablica jest rozszerzana o kolejne pozycje - na bieżącą tablicę deskryptorów wskazuje pole **`fd`**;
   - Dostęp do otwartego pliku z poziomu procesu: **`t->files->fd[i]`**
   - Typ danych zdefiniowany w pliku **include/linux/sched.h**

Dostęp z poziomu struktur procesów (2/2)

fs_struct *fs

```
struct fs_struct {
    atomic_t count;
    int umask;
    struct dentry * root, * pwd;
};
```



files_struct *files

Type	Field	Description
int	count	Number of processes sharing this table
int	max_fds	Current maximum number of file objects
int	max_fdset	Current maximum number of file descriptors
int	next_fd	Maximum file descriptors ever allocated plus 1
struct file **	fd	Pointer to array of file object pointers
fd_set *	close_on_exec	Pointer to file descriptors to be closed on exec( )
fd_set *	open_fds	Pointer to open file descriptors
fd_set	close_on_exec_init	Initial set of file descriptors to be closed on exec( )
fd_set	open_fds_init	Initial set of file descriptors
struct file *	fd_array[32]	Initial array of file object pointers

maski

Rejestrowanie typu systemu plików w VFS

- Wirtualny system plików (VFS) – podstawowe koncepcje
- Struktury danych VFS
  - Obiekty superbloku
  - Obiekty i-węzła
  - Obiekty pliku
  - Obiekty pozycji w katalogu
- **Rejestrowanie typu systemu plików w VFS**
- Montowanie systemu plików w VFS
- Przykład działania VFS

Typy systemów plików

Zapisy w jądrze związane z implementacją systemu plików:

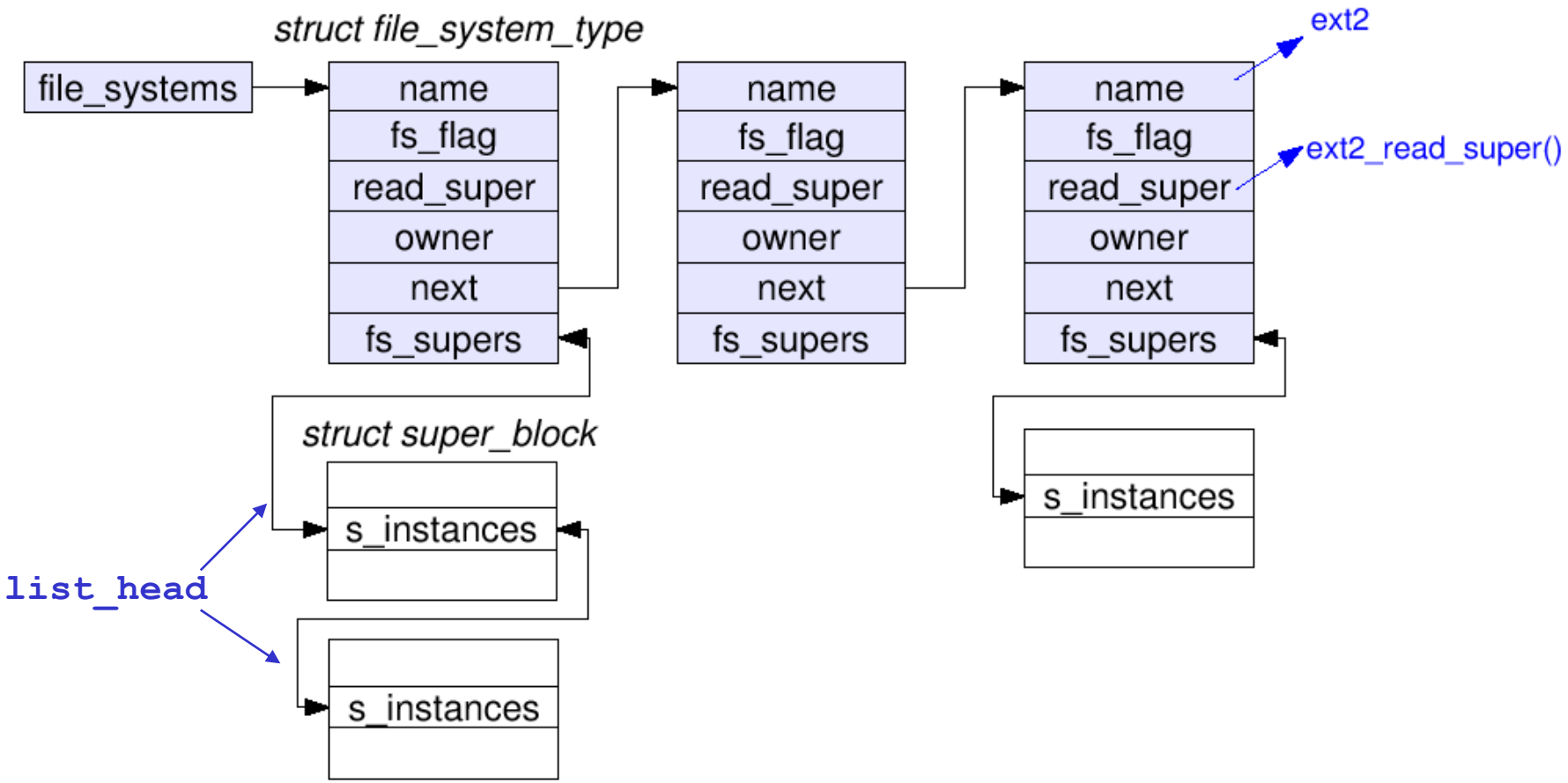
- zawierają listę wbudowanych lub załadowanych modułów systemów plikowych,
- ich typy podane są w strukturze **`struct file_system_type`** (plik nagłówkowy **include/linux/fs.h**)

Podstawowe pola struktury **`struct file_system_type`**:

- **name**: nazwa typu systemu plików, np.“ext2”
- **read_ super**: funkcja odczytująca **superblok**
- **owner**: moduł, który jest implementacją określonego typu systemu plików
- **fs_flags**: określa czy wymagane jest urządzenie rzeczywiste, itp.
- **fs_supers**: lista **super bloków** (dla wszystkich zamontowanych egzemplarzy tego samego typu systemu plików)
- **next**: wskaźnik do następnego elementu listy zarejestrowanych typów systemów

Na pierwszy element listy zarejestrowanych typów systemów plików wskazuje zmienna **`file_systems`**.

Ilustracja listy zarejestrowanych typów systemu plików



Rejestrowanie typu systemu plików

Każdy typu systemu plików **musi być zarejestrowany** przez VFS

- rejestrowanie odbywa się zwykle podczas dynamicznego łączenia modułu za pomocą funkcji **`init_module(...)`**
- w momencie zwolnienia modułu musi być on wyrejestrowany

Aby zarejestrować moduł

- napisz lub wskaż funkcję **`read_super()`** (zależna od typu systemu plików funkcja odczytu superbloku)
- zaalokuj pamięć na obiekt określający typ systemu plików

```
DECLARE_FSTYPE(var, type, read, flags)
DECLARE_FSTYPE_DEV(var, type, read)
```

- wywołaj funkcje rejestracji **`register_filesystem (struct file_system_type *)`**

Przykład rejestrowania systemu plików

Patrz koniec pliku **fs/ext2/super.c**

```
DECLARE_FSTYPE_DEV(ext2_fs_type, "ext2", ext2_read_super);

static int__init init_ext2_fs( void)
{
   return register_file_system( &ext2_fs_type);
}
static int__exit exit_ext2_fs( void)
{
   unregister_file_system( &ext2_fs_type);
}

EXPORT_NO_SYMBOLS;
module_init( init_ext2_fs)
module_exit( exit_ext2_fs)
```

Montowanie systemu plików w VFS

- Wirtualny system plików (VFS) – podstawowe koncepcje
- Struktury danych VFS
  - Obiekty superbloku
  - Obiekty i-węzła
  - Obiekty pliku
  - Obiekty pozycji w katalogu
- Rejestrowanie typu systemu plików w VFS
- **Montowanie systemu plików w VFS**
- Przykład działania VFS

Lista zamontowanych systemów

Wszystkie zamontowane systemy plików są zawarte w liście złożonej z obiektów typu **`struct vfsmount`**. Dostęp do pierwszego elementu listy za pomocą zmiennej **`vfsmntlist`**.

Type	Field	Description
`kdev_t`	`mnt_dev`	Device number
`char *`	`mnt_devname`	Device name
`char *`	`mnt_dirname`	Mount point
`unsigned int`	`mnt_flags`	Device flags
`struct super_block *`	`mnt_sb`	Superblock pointer
`struct quota_mount_options`	`mnt_dquot`	Disk quota mount options
`struct vfsmount *`	`mnt_next`	Pointer to next list element

Funkcje:

- **`add_vfsmnt( )`**
- **`remove_vfsmnt( )`**
- **`lookup_vfsmnt( )`**

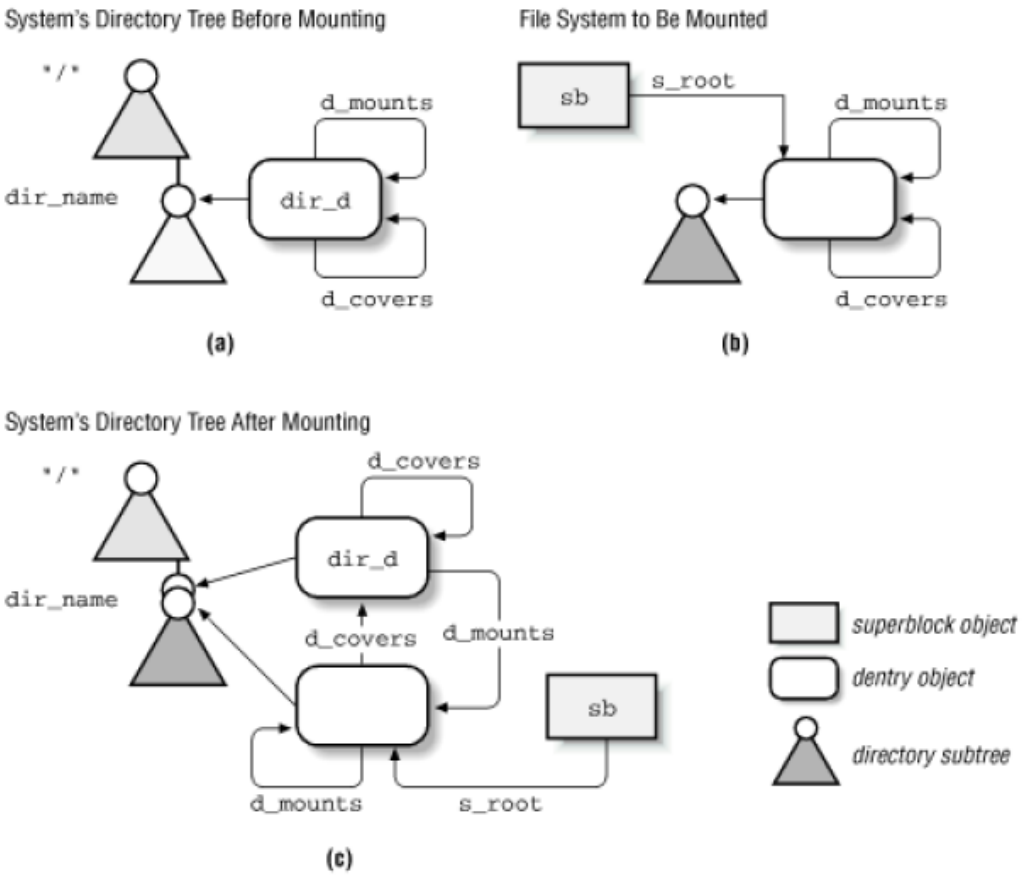
Przykład montowania

Podczas montowania poza listą obiektów **`vfsmount`** modyfikowane są również odpowiednie obiekty **`dentry`**.

`dir_name`
`dir_d`

↓

argumenty fu-cji
`do_mount(…)`

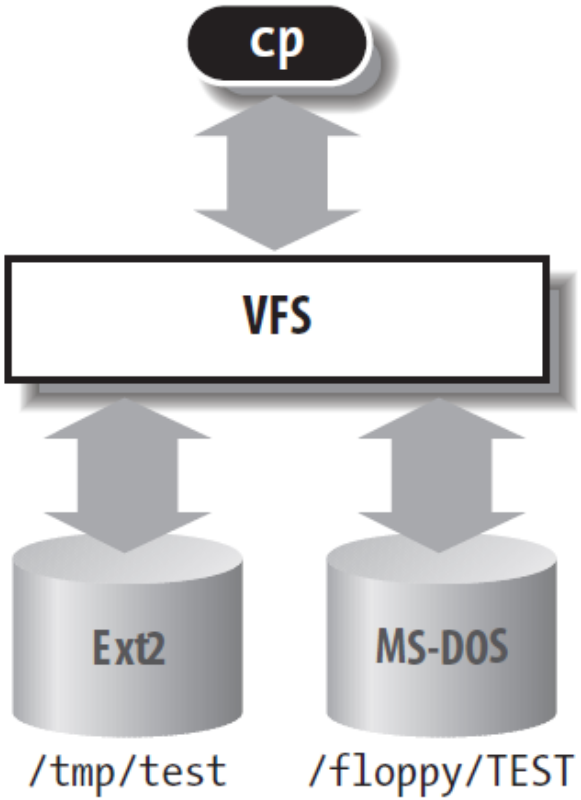


Przykład działania VFS

- Wirtualny system plików (VFS) – podstawowe koncepcje
- Struktury danych VFS
  - Obiekty superbloku
  - Obiekty i-węzła
  - Obiekty pliku
  - Obiekty pozycji w katalogu
- Rejestrowanie typu systemu plików w VFS
- Montowanie systemu plików w VFS
- **Przykład działania VFS**

Przykład działania VFS

W omawianym przykładzie dla przejrzystości **ominięto obsługę błędów i operacje związane z kontrolą dostępu**.

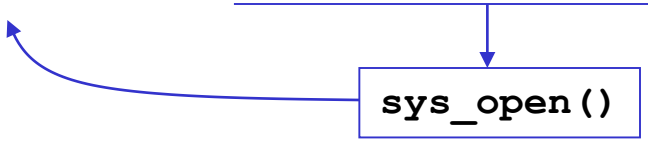



```
$ cp /floppy/TEST /tmp/test
```

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

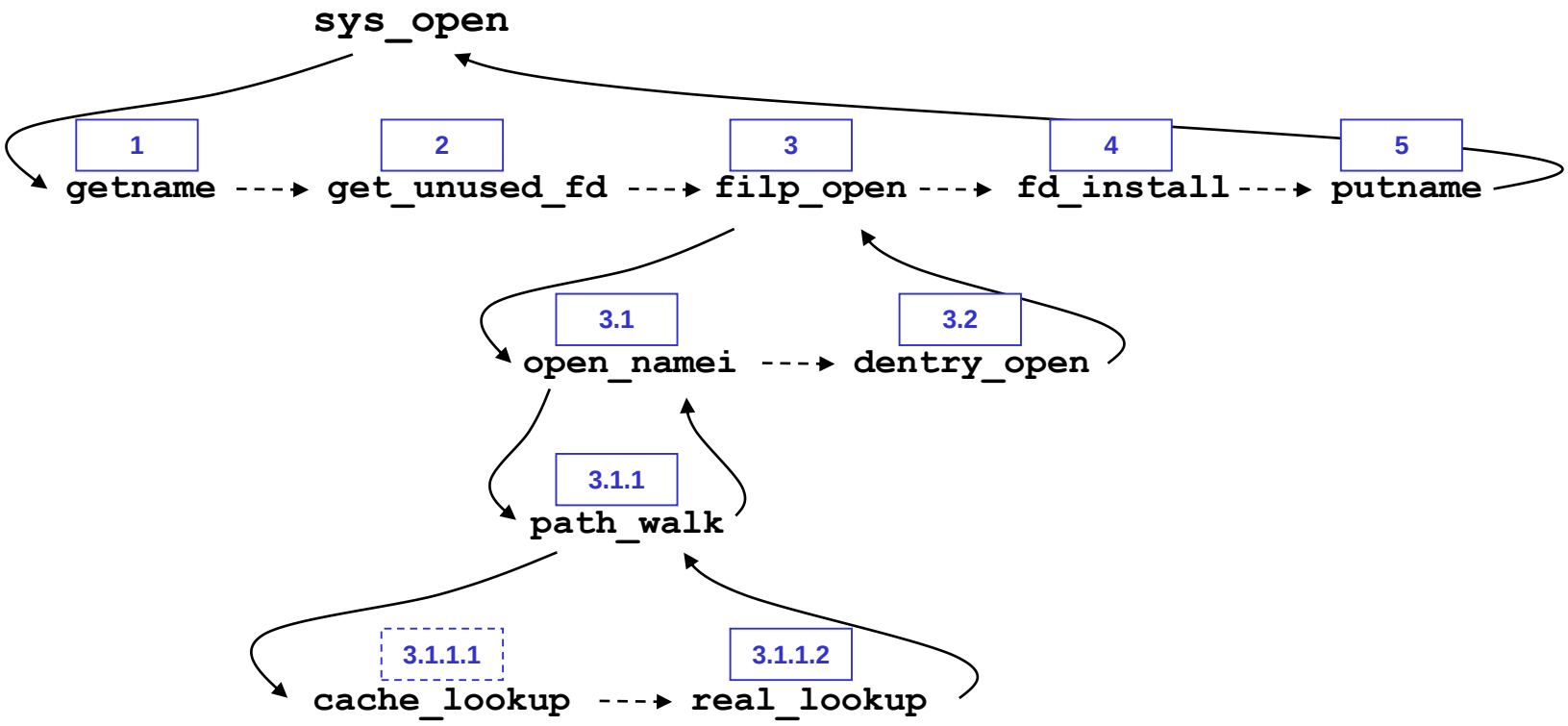
Wywołanie `open()`

```
fd = open( filename, flags, mode)
```



Flag Name	Description
FASYNC	Asynchronous I/O notification via signals
O_APPEND	Write always at end of the file
O_CREAT	Create the file if it does not exist
O_DIRECTORY	Fail if file is not a directory
O_EXCL	With O_CREAT, fail if the file already exists
O_LARGEFILE	Large file (size greater than 2 GB)
O_NDELAY	Same as O_NONBLOCK
O_NOCTTY	Never consider the file as a controlling terminal
O_NOFOLLOW	Do not follow a trailing symbolic link in pathname
O_NONBLOCK	No system calls will block on the file
O_RDONLY	Open for reading
O_RDWR	Open for both reading and writing
O_SYNC	Synchronous write (block until physical write terminates)
O_TRUNC	Truncate the file
O_WRONLY	Open for writing

sys_open() – sekwencja działań



sys_open()

fs/open.c:

```
int sys_open(const char *filename, int flags, int mode) {
        char *tmp = getname(filename);                          1
        int fd = get_unused_fd();                               2
        struct file *f = filp_open(tmp, flags, mode);           3
        fd_install(fd, f);                                      4
        putname(tmp);                                           5
        return fd;
}
```

sys_open()- 1,5

getname 1
putname 5

fs/namei.c:

```
#define __getname()     kmem_cache_alloc(names_cachep, SLAB_KERNEL)
#define putname(name)   kmem_cache_free(names_cachep, (void *)(name))

char *getname(const char *filename) {
        char *tmp = __getname();        /* allocate some memory */
        strncpy_from_user(tmp, filename, PATH_MAX + 1);
        return tmp;
}
```

sys_open() - 2

get_unused_fd 2

fs/open.c:

```
int get_unused_fd(void) {
        struct files_struct *files = current->files;
        int fd = find_next_zero_bit( files->open_fds,
                                     files->max_fdset,
                                     files->next_fd);
        FD_SET(fd, files->open_fds);    /* in use now */
        files->next_fd = fd + 1;
        return fd;
}
```

sys_open() - 4

fd_install 4

include/linux/file.h:

```
void fd_install(unsigned int fd, struct file *file) {
        struct files_struct *files = current->files;
        files->fd[fd] = file;
}
```

sys_open() - 3

filp_open 3

fs/open.c:

```
struct file *filp_open(const char *filename, int flags, int mode) {
        struct nameidata nd;
        open_namei(filename, flags, mode, &nd);          3.1
        return dentry_open(nd.dentry, nd.mnt, flags);    3.2
}
```

include/linux/fs.h:

```
struct nameidata {
        struct dentry *dentry;
        struct vfsmount *mnt;
        struct qstr last;  };
```

sys_open() - 3.1

open_namei 3.1

fs/namei.c:

```
open_namei(const char *pathname, int flag, int mode, struct nameidata *nd) {
        if (!(flag & O_CREAT)) {
                /* The simplest case - just a plain lookup. */
                if (*pathname == '/') {
                        nd->mnt = mntget(current->fs->rootmnt);
                        nd->dentry = dget(current->fs->root);
                } else {
                        nd->mnt = mntget(current->fs->pwdmnt);
                        nd->dentry = dget(current->fs->pwd);
                }
                path_walk(pathname, nd);        3.1.1
                /* Check permissions etc. */
                ...
                return 0;  }
        ...  }
```

sys_open() - 3.1.1

path_walk 3.1.1

fs/namei.c:

```
path_walk(const char *name, struct nameidata *nd) {
        struct dentry *dentry;
        for(;;) {
                struct qstr this;
                this.name = next_part_of(name);
                this.len = length_of(this.name);
                this.hash = hash_fn(this.name);
                /* if . or .. then special, otherwise: */
                dentry = cached_lookup(nd->dentry, &this);          3.1.1.1
                if (!dentry)
                        dentry = real_lookup(nd->dentry, &this);    3.1.1.2
                nd->dentry = dentry;
                if (this_was_the_final_part)
                        return;   }
}
```

sys_open() - 3.1.1.2

real_lookup 3.1.1.2

fs/namei.c:

```
struct dentry *
real_lookup(struct dentry *parent, struct qstr *name, int flags) {
        struct dentry *dentry = d_alloc(parent, name);
        parent->d_inode->i_op->lookup(dir, dentry);
        return dentry;
}
```

odwołanie do funkcji **lookup** konkretnego systemu

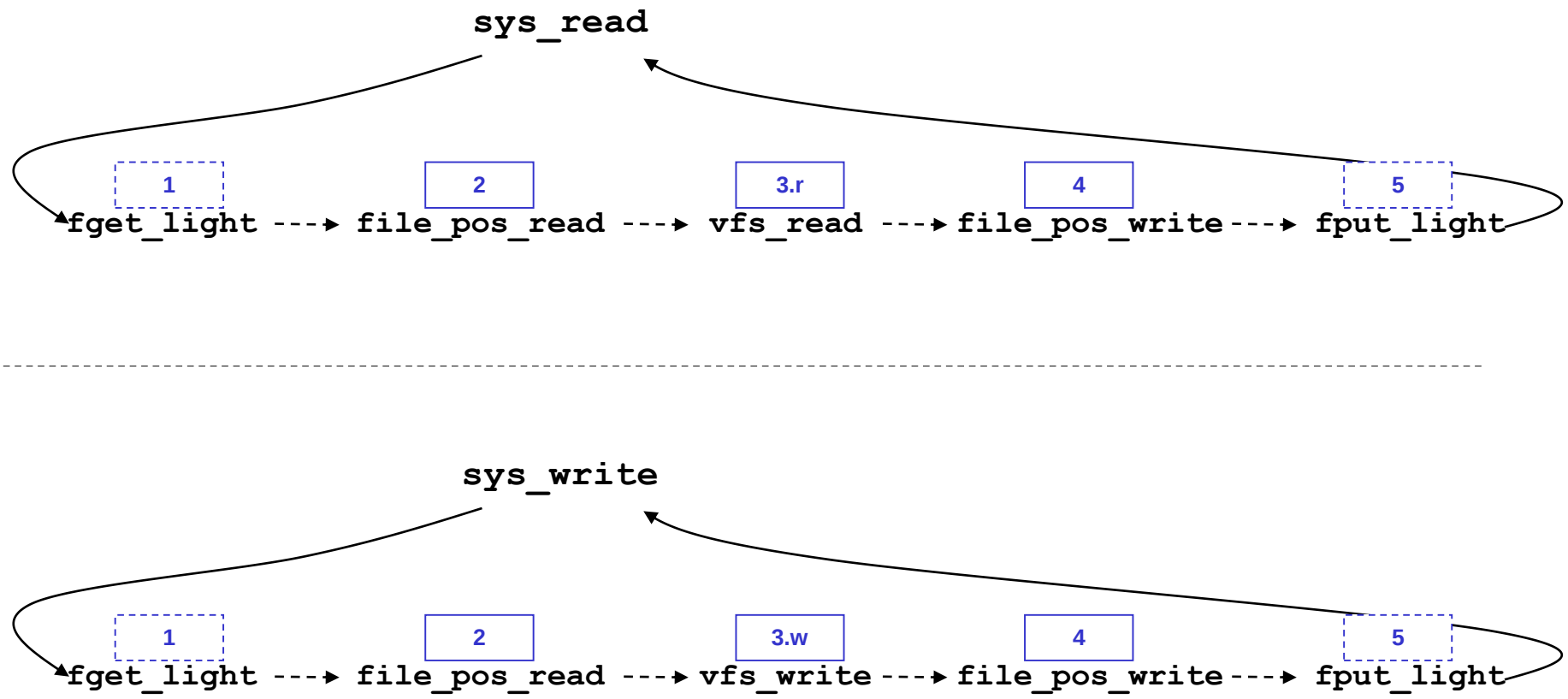
sys_open() - 3.2

dentry_open 3.2

fs/open.c:

```
struct file *
dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags) {
        struct file *f = get_empty_filp();
        f->f_dentry = dentry;
        f->f_vfsmnt = mnt;
        f->f_pos = 0;
        f->f_op = dentry->d_inode->i_fop;
        ...
        return f;
}
```

`sys_read()`, `sys_write()` - sekwencja działań



sys_read()

fs/read_write.c:

```
asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t count){
        struct file *file;
        ssize_t ret = -EBADF;
        int fput_needed;

        file = fget_light(fd, &fput_needed);                    1
        if (file) {
                loff_t pos = file_pos_read(file);               2
                ret = vfs_read(file, buf, count, &pos);         3.r
                file_pos_write(file, pos);                      4
                fput_light(file, fput_needed);                  5
                }
        return ret;
}
```

sys_write()

fs/read_write.c:

```
asmlinkage ssize_t sys_write(unsigned int fd, const char __user * buf, size_t count){
        struct file *file;
        ssize_t ret = -EBADF;
        int fput_needed;

        file = fget_light(fd, &fput_needed);                    1
        if (file) {
                loff_t pos = file_pos_read(file);               2
                ret = vfs_write(file, buf, count, &pos);        3.w
                file_pos_write(file, pos);                      4
                fput_light(file, fput_needed);                  5
                }
        return ret;
}
```

sys_read(), sys_write – 2,4

file_pos_read 2

file_pos_write 4

fs/read_write.c:

```
static inline loff_t file_pos_read(struct file *file)
{
        return file->f_pos;
}

static inline void file_pos_write(struct file *file, loff_t pos)
{
        file->f_pos = pos;
}
```

sys_read() - 3.r

3.r

fs/read_write.c:

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos){
        ssize_t ret;
        ...
        ret = rw_verify_area(READ, file, pos, count);
        if (ret >= 0) {
                count = ret;
                ret = security_file_permission (file, MAY_READ);
                if (!ret) {
                        if (file->f_op->read)
                                ret = file->f_op->read(file, buf, count, pos);
                        else
                                ret = do_sync_read(file, buf, count, pos);
                        ... }
        }
        return ret;
}
```

sys_write() – 3.w

3.w

fs/read_write.c:

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos){
        ssize_t ret;
        ...
        ret = rw_verify_area(WRITE, file, pos, count);
        if (ret >= 0) {
                count = ret;
                ret = security_file_permission (file, MAY_WRITE);
                if (!ret) {
                        if (file->f_op->read)
                                ret = file->f_op->write(file, buf, count, pos);
                        else
                                ret = do_sync_write(file, buf, count, pos);
                        ... }
        }
        return ret;
}
```