

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
- FUSE

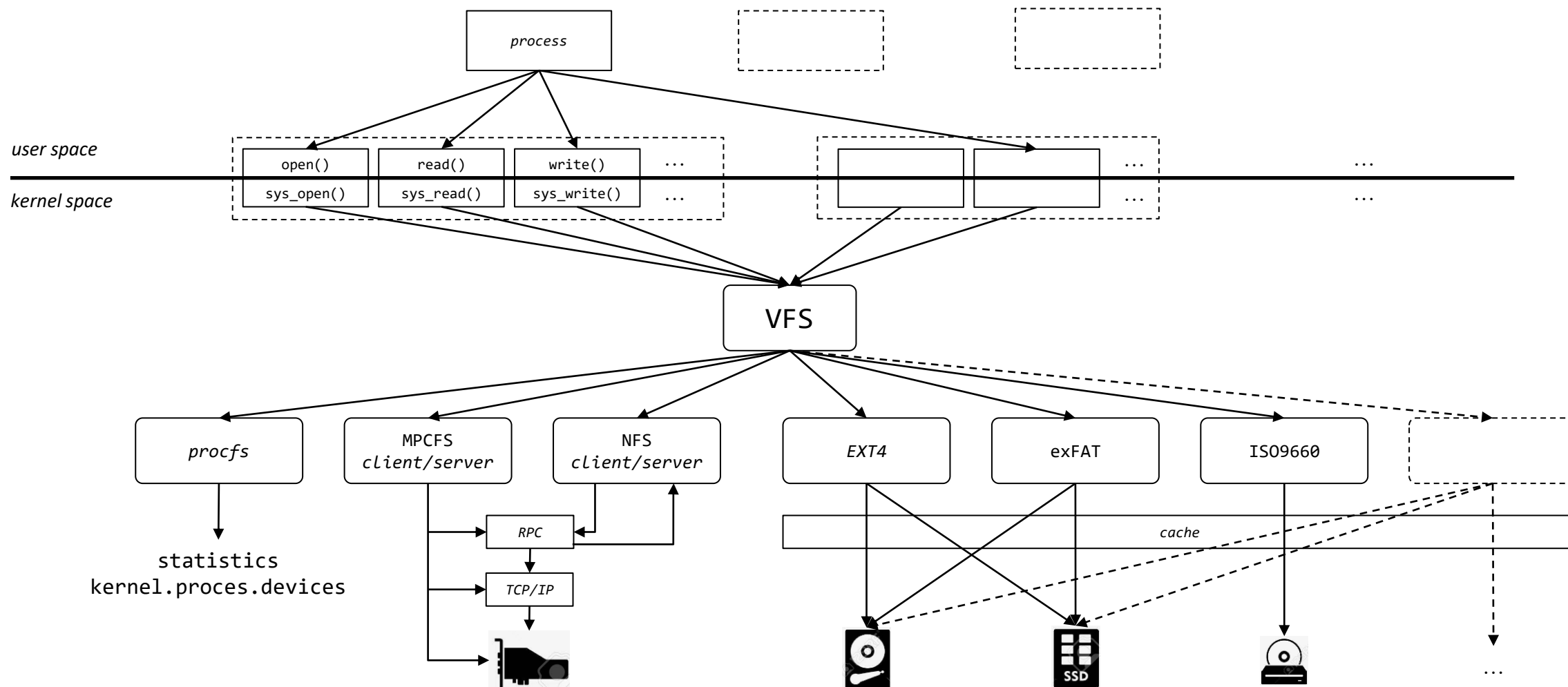
Wirtualny system plików (VFS) – podstawowe koncepcje

- **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
- FUSE

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

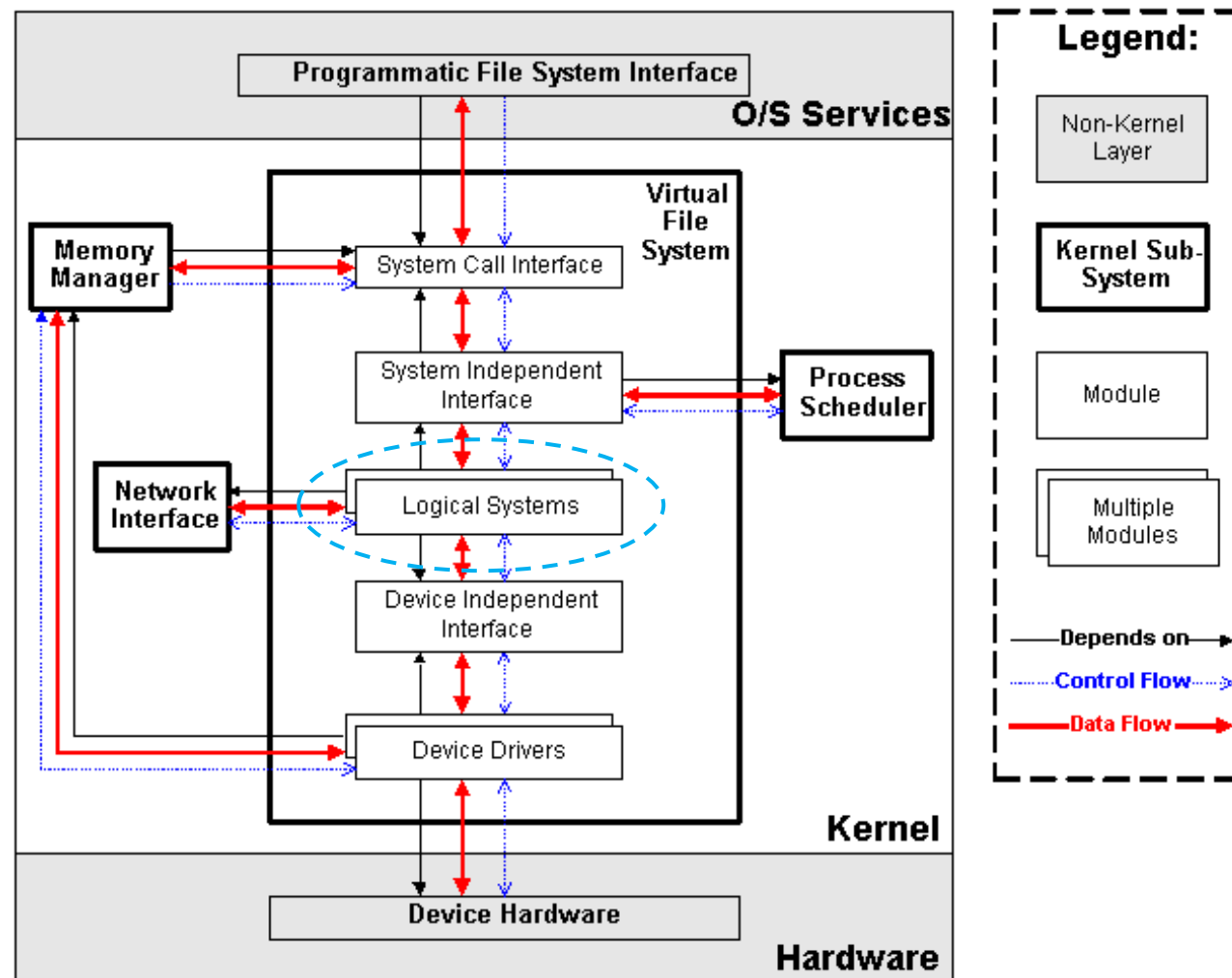


Klasy obsługiwanych systemów plików

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



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:

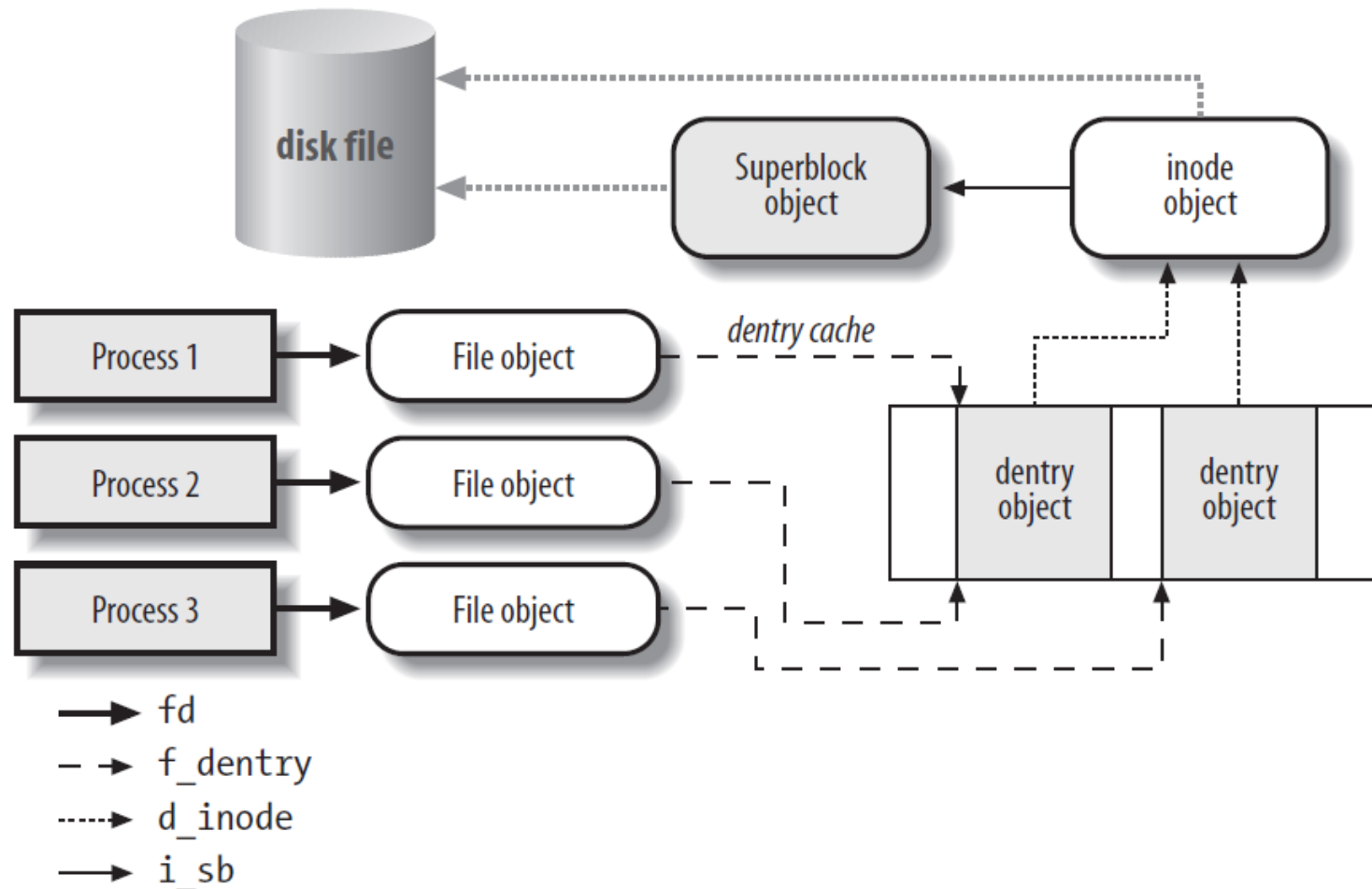
- **obiekt super bloku** (ang. *super-block object*), przechowujący informacje dotyczące zamontowanego systemu plików,
- **obiekt i-węzła** (ang. *inode-object*), przechowujący informacje o danym pliku
- **obiekcie pliku** (ang. *file-object*) reprezentujące powiązanie między otwartym plikiem a procesem
- **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

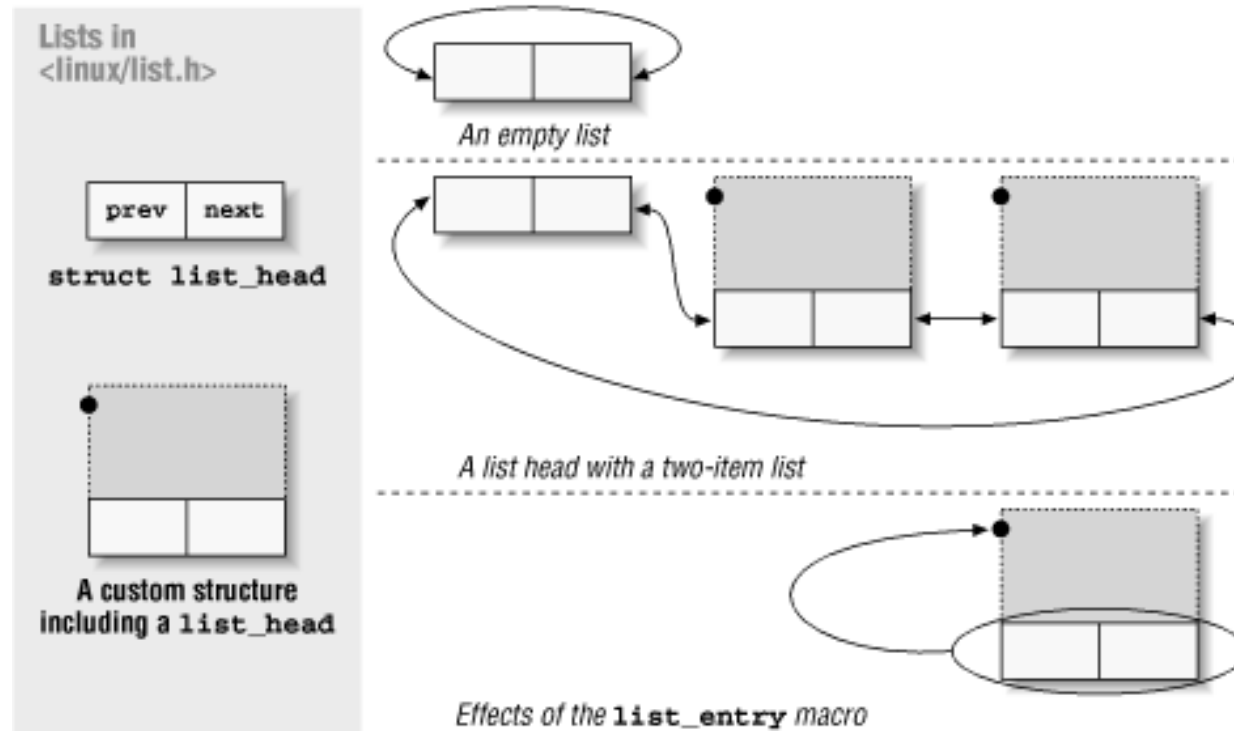
Powiązania procesów z obiektami VFS (w pamięci operacyjnej)



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 (jakiegokolwiek zmiany jego zawartości 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)

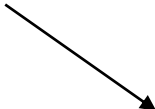
- każdy obiekt dostarcza zbioru operacji (w postaci wskaźników na funkcje)
- zwykle metody są **niezależne od typu systemu plików**, ale czasami charakterystyczne tylko dla systemu plików lub pliku

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

niektóre funkcje VFS nie wymagają odwoływania się do funkcji niższego poziomu



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

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.

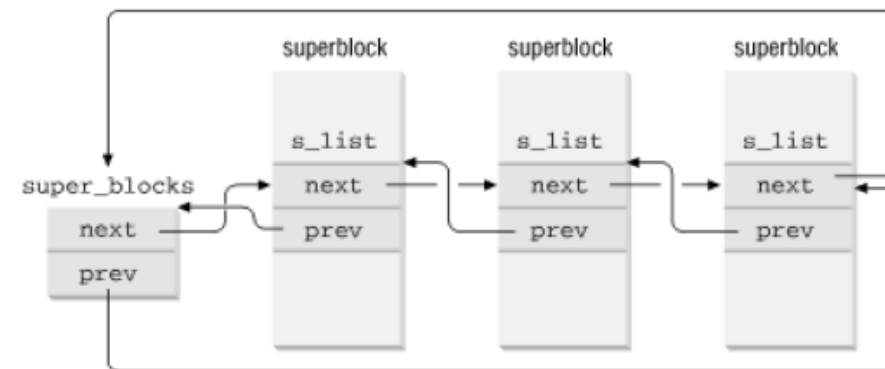
s_op – metody obiektu

s_type – typ systemu plików

s_root – obiekt dentry punktu montowania

s_blocksize – rozmiar bloku w bajtach

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



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)`: zwalnia 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
- FUSE

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**

Listy węzłów VFS

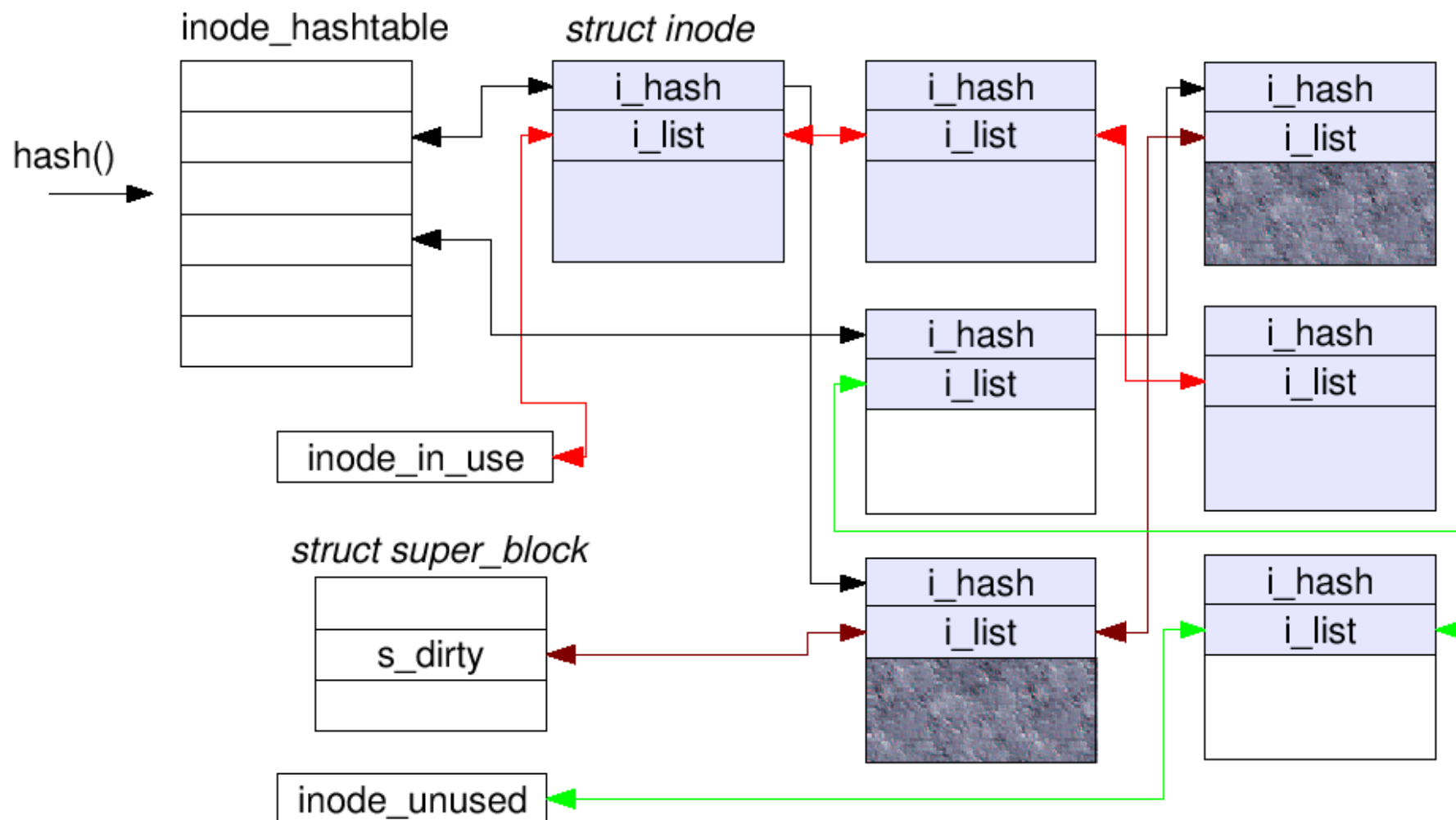
Tablica mieszająca i-węzłów (dla wszystkich i-węzłów w użyciu lub zmodyfikowanych), dostęp do początku listy przez zmienną `inode_hashtable`

- 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
- FUSE

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

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
- FUSE

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**, ...
- Powiązanie punktu montowania i katalogu root: **d_mounts**, **d_covers**

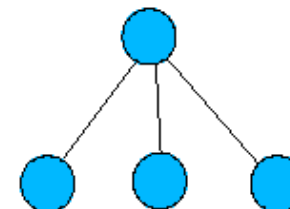
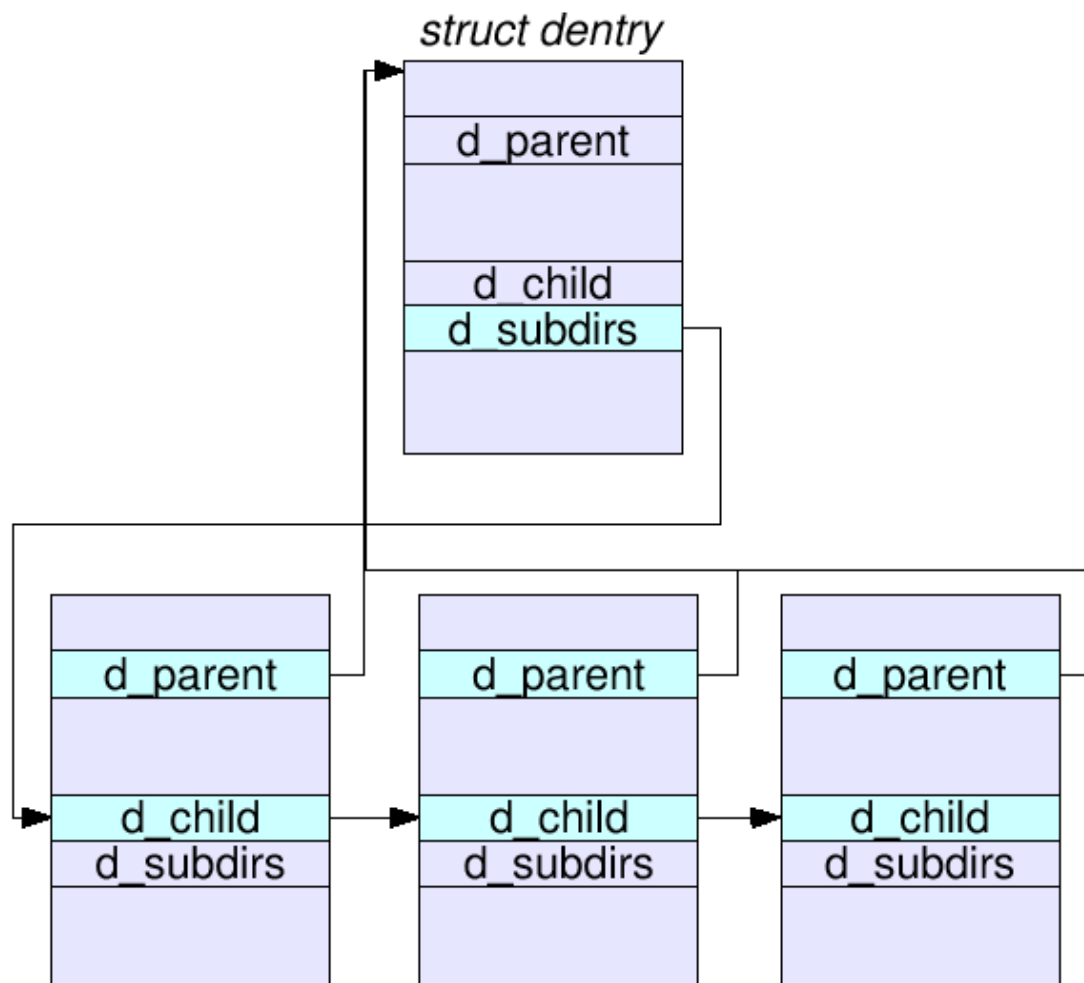
Metody zapisów w katalogu

- 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**)

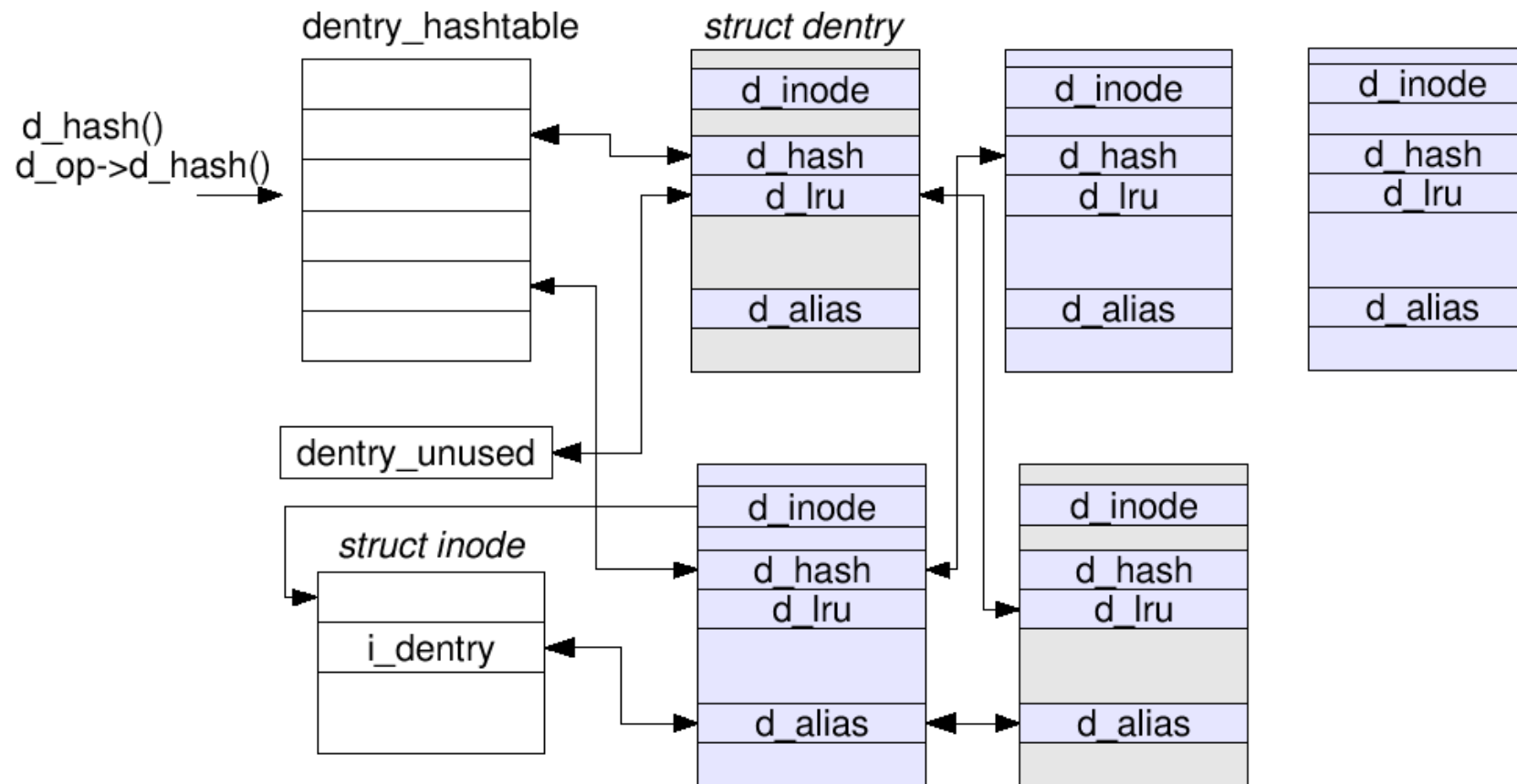
Listy wpisów katalogów

- 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)

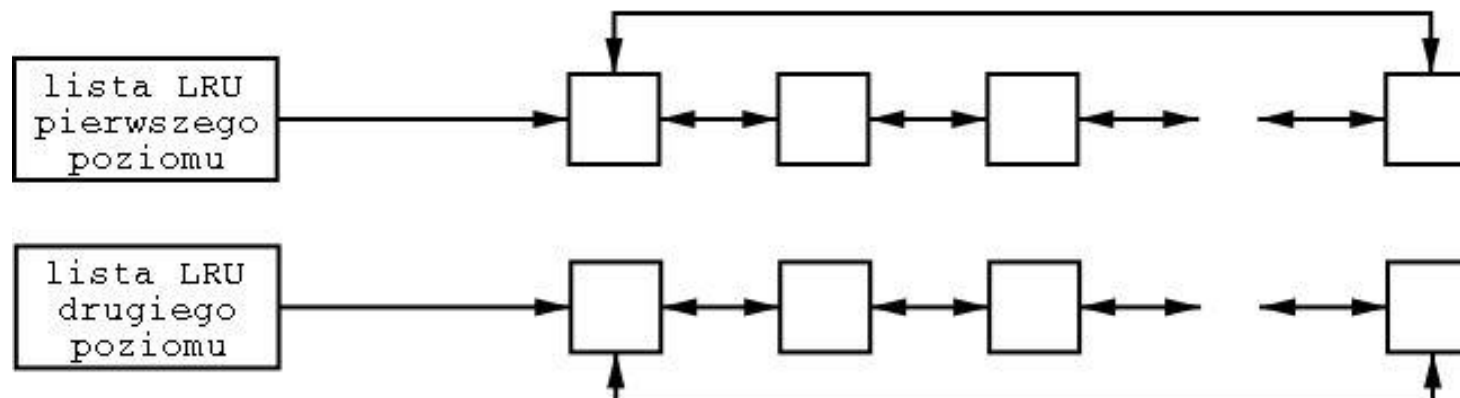


Lista zapisów w katalogach (skrót/wolny/alias)

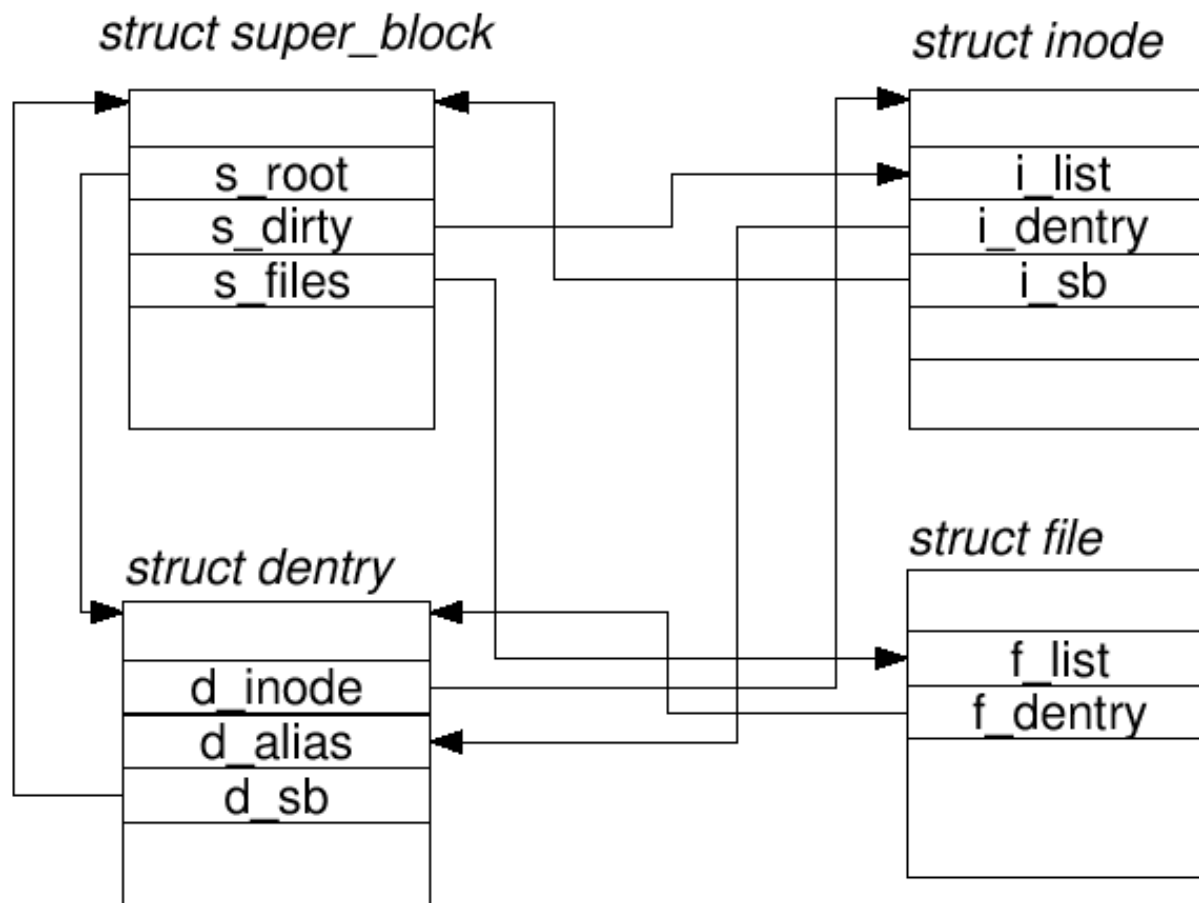


Listy LRU podręcznej pamięci buforowej zapisów w katalogach

- 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`

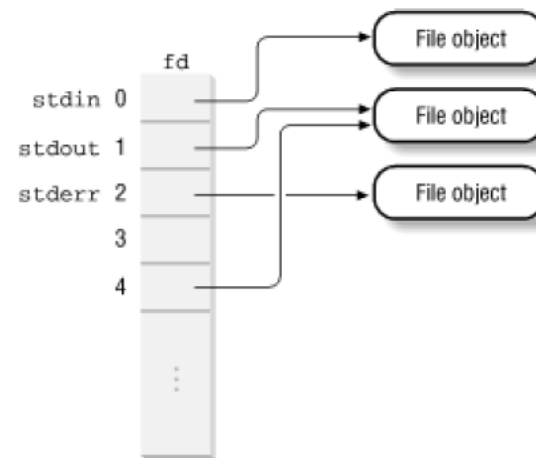
2. pole `struct files_struct *files`

- 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
- FUSE

Typy systemów plików

Zapisy w jądrze związane z implementacją systemu plików:

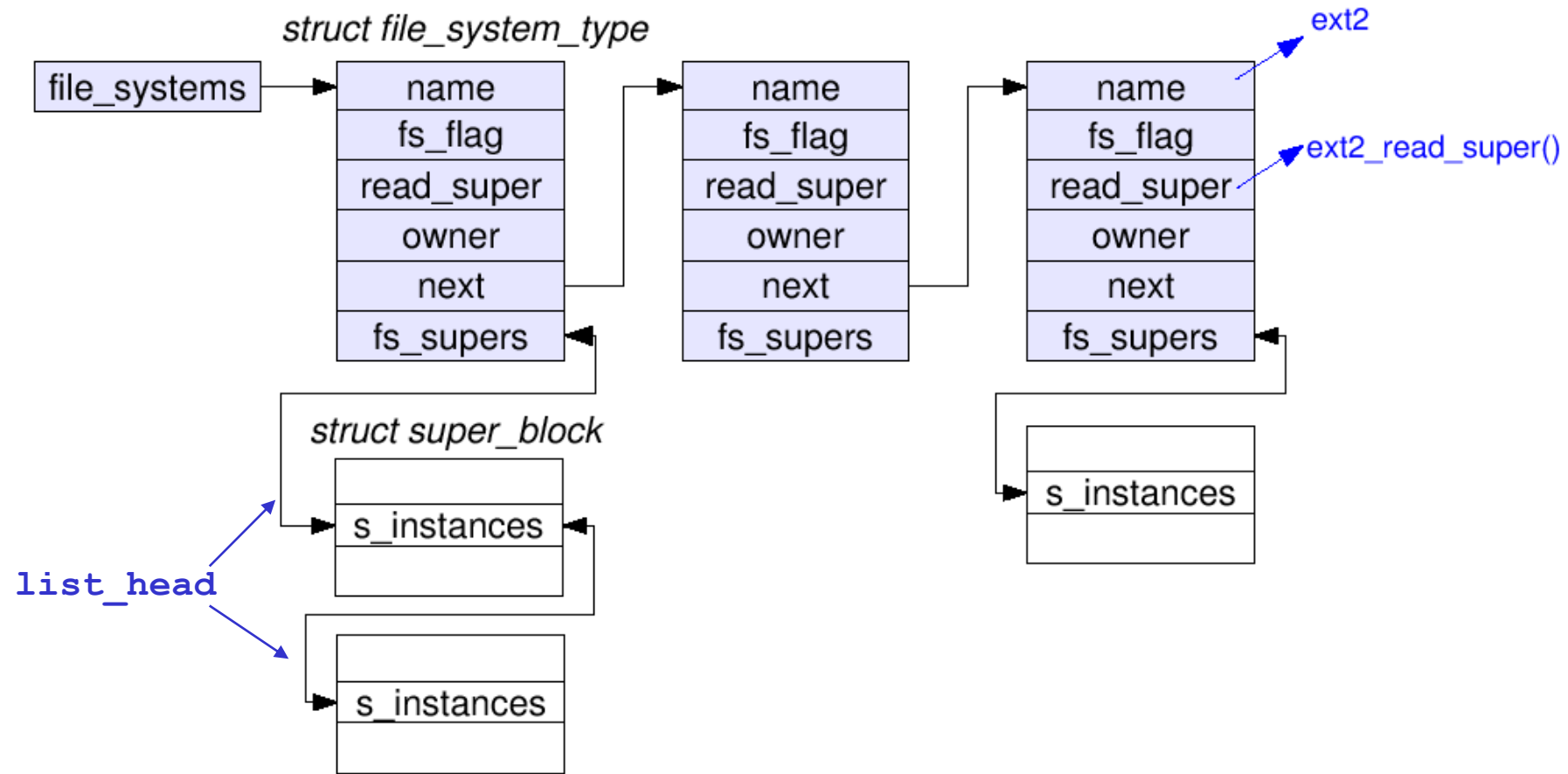
- zawierają listę wbudowanych lub załadowanych modułów systemów plikowych,
- ich typy podane są w strukturze `struct file_system_type` (plik nagłówkowy `include/linux/fs.h`)

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 typ 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 funkcję 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
- FUSE

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

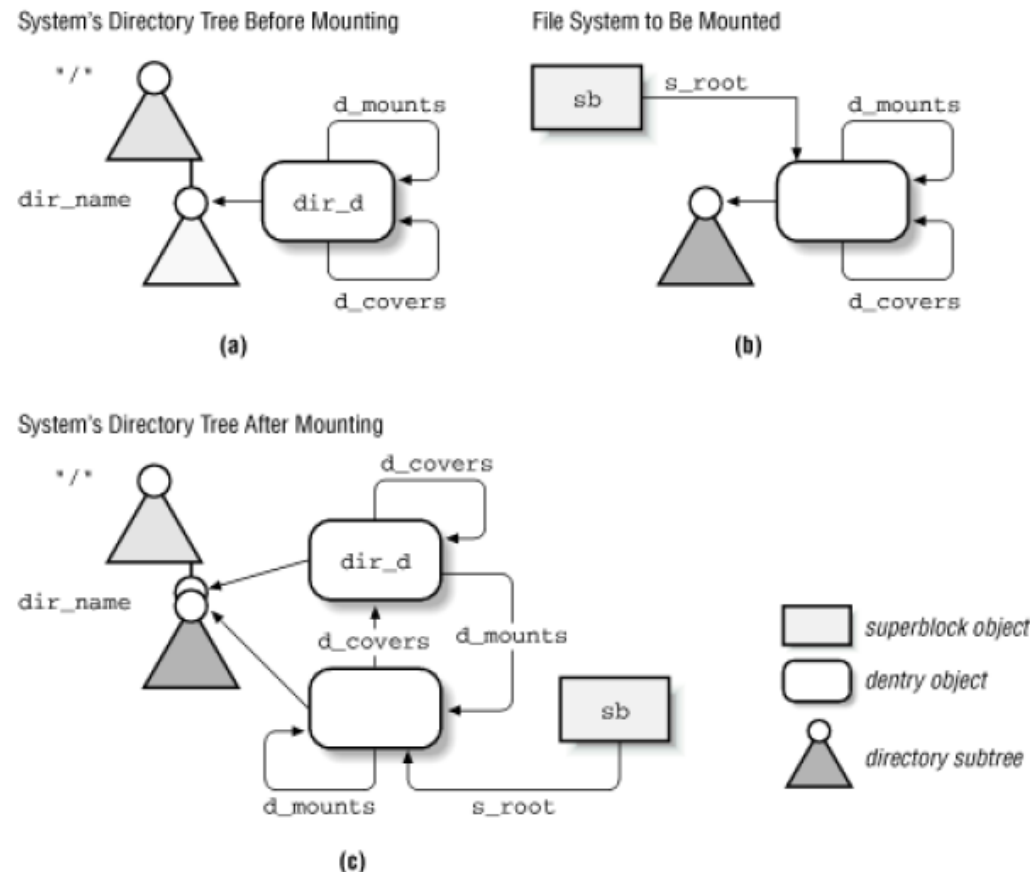
Funkcje:

```
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(...)`

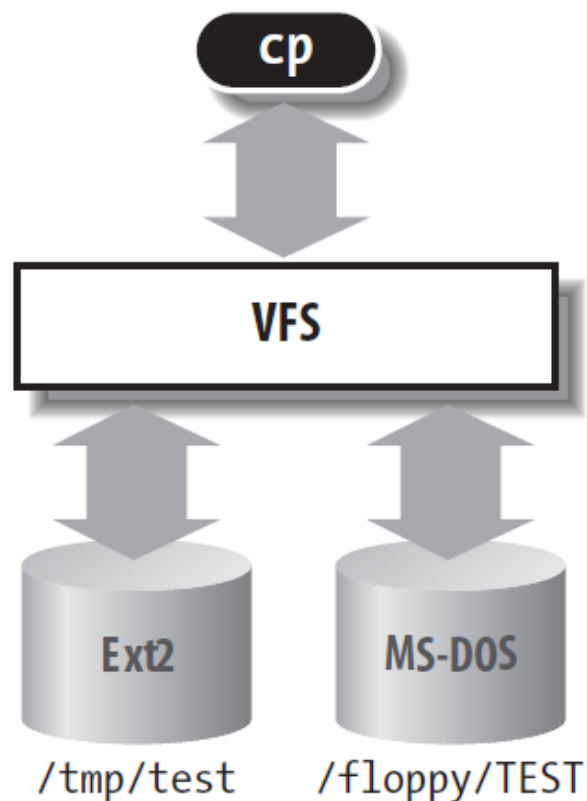


Przykład działania VFS

- 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**
- FUSE

Przykład działania VFS

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

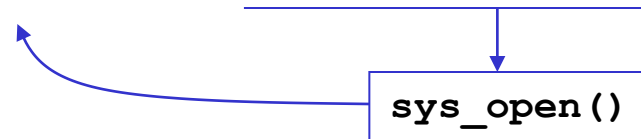


```
$ cp /floppy/TEST /tmp/test
```

```
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);
```

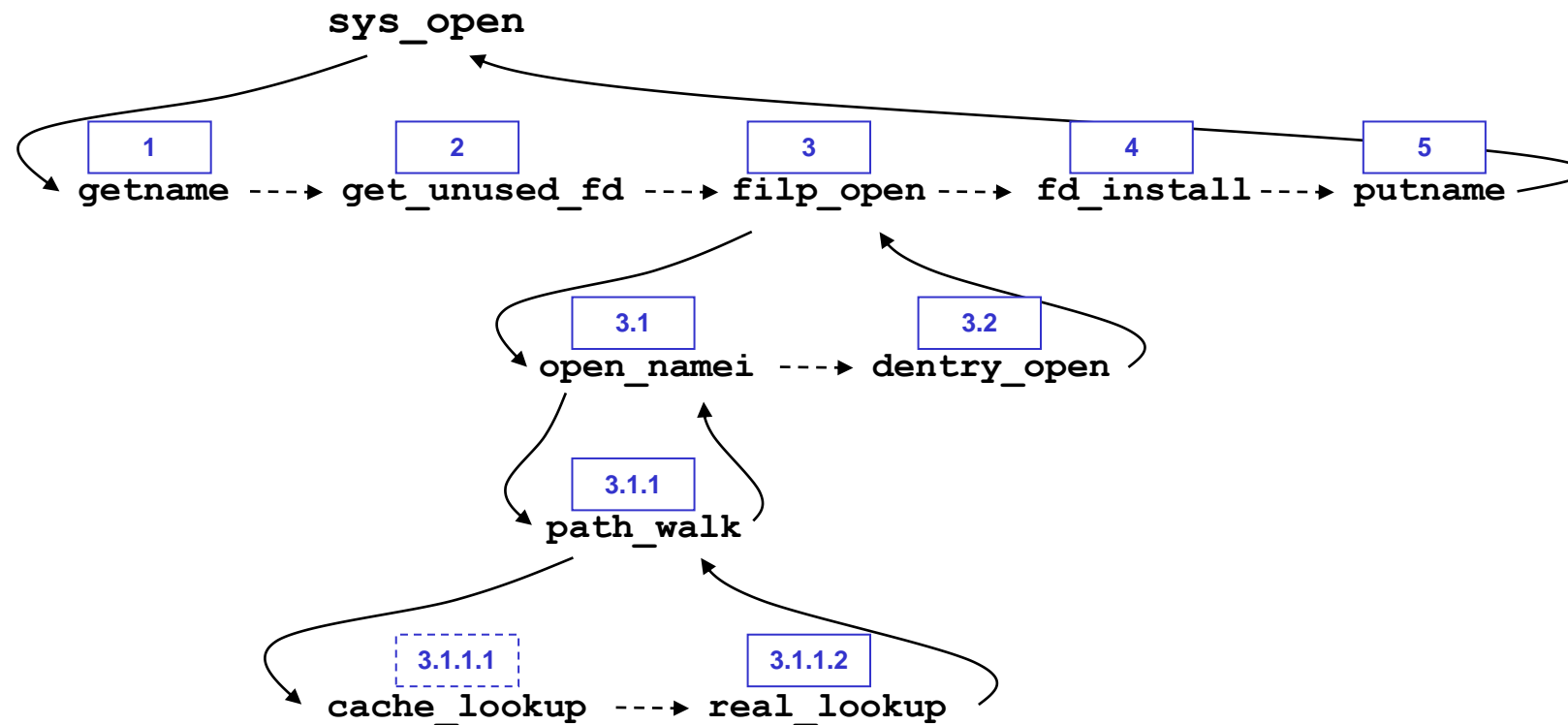
Wywołanie `open()`

```
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);  
    int fd = get_unused_fd();  
    struct file *f = filp_open(tmp, flags, mode);  
    fd_install(fd, f);  
    putname(tmp);  
    return fd;  
}
```

1

2

3

4

5

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);  
    return dentry_open(nd.dentry, nd.mnt, flags);  
}
```

3.13.2

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);
        if (!dentry)
            dentry = real\_lookup(nd->dentry, &this);
        nd->dentry = dentry;
        if (this_was_the_final_part)
            return;
    }
}
```

[3.1.1.1](#)[3.1.1.2](#)

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

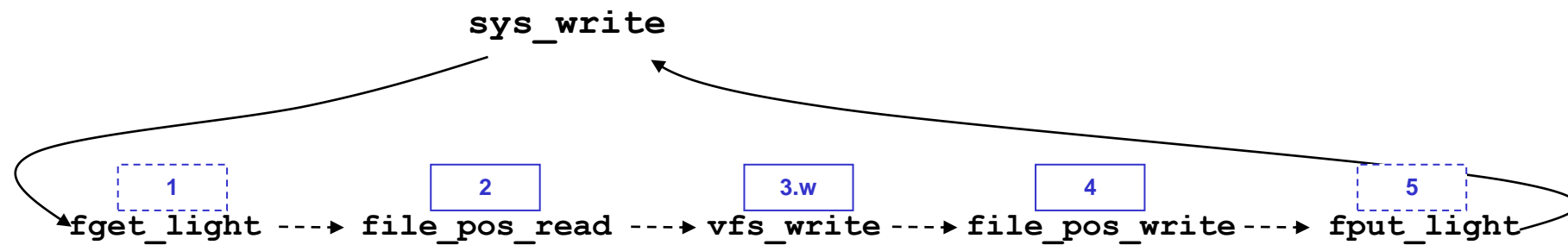
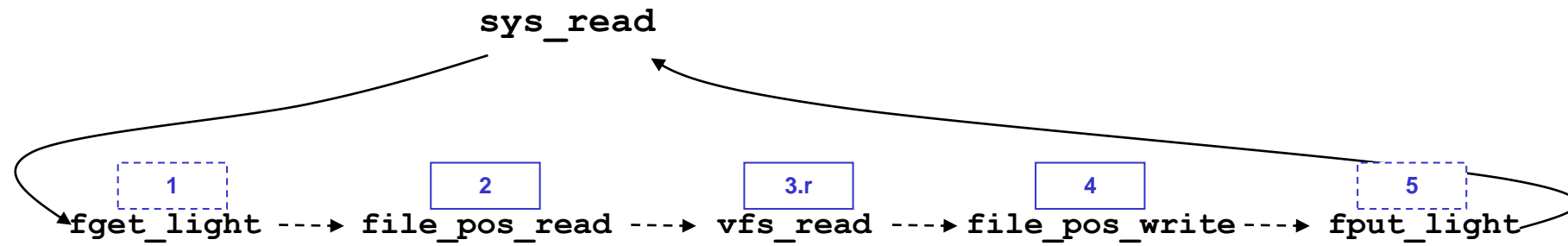
sys_open() - 3.2

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);
    if (file) {
        loff_t pos = file_pos_read(file);
        ret = vfs_read(file, buf, count, &pos);
        file_pos_write(file, pos);
        fput_light(file, fput_needed);
    }
    return ret;
}
```

1

2

3.r

4

5

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);  
    if (file) {  
        loff_t pos = file_pos_read(file);  
        ret = vfs_write(file, buf, count, &pos);  
        file_pos_write(file, pos);  
        fput_light(file, fput_needed);  
    }  
    return ret;  
}
```

1

2

3.w

4

5

`sys_read()` , `sys_write` - 2,4

`file_pos_read`

`file_pos_write`

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;
}
```

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
- **FUSE**

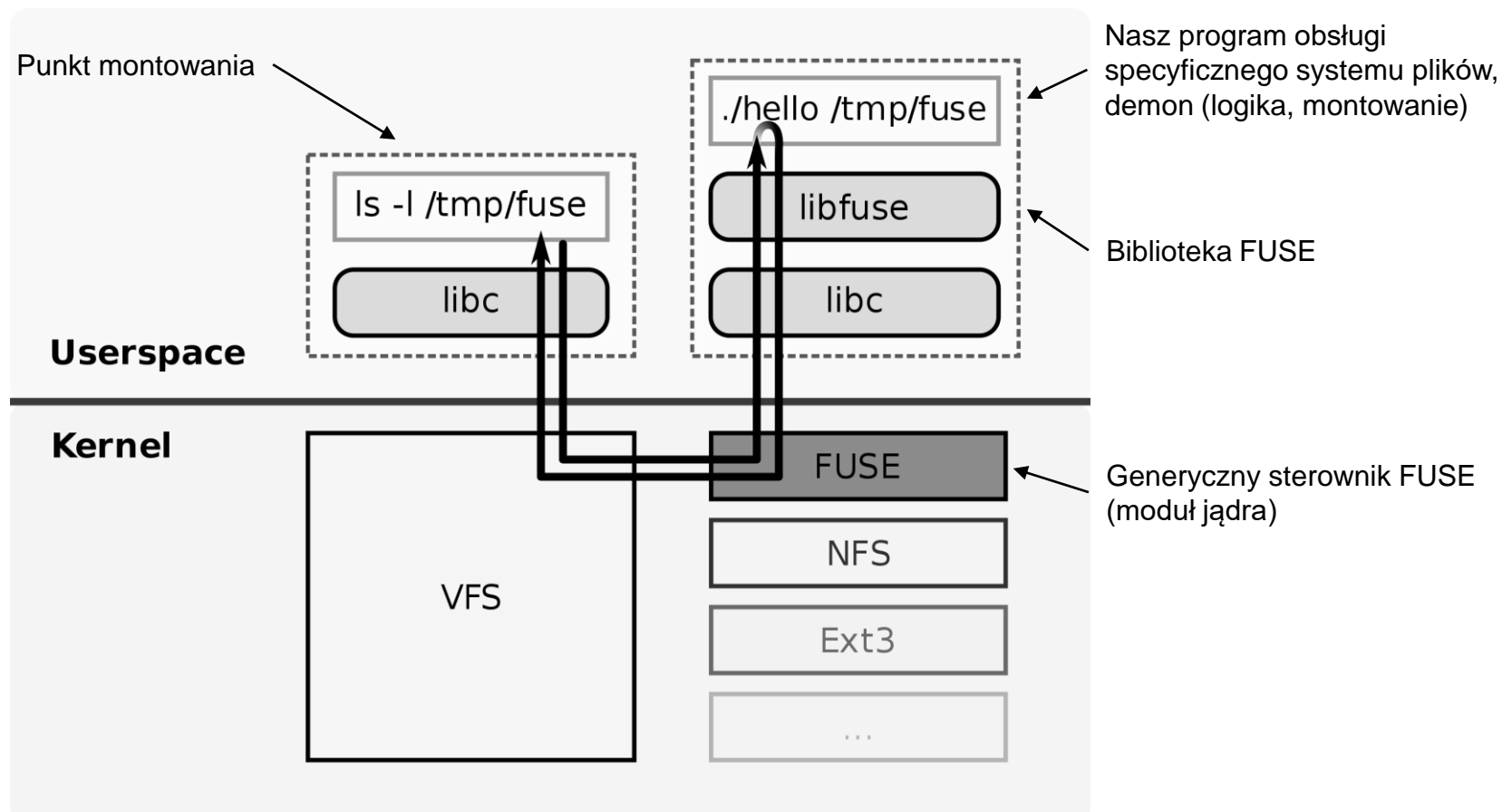
Koncepcja FUSE

- FUSE jest ładowalnym modułem jądra dla systemów operacyjnych klasy Unix, który pozwala nieuprzywilejowanym użytkownikom tworzyć własne systemy plików bez edytowania kodu jądra.
- Osiąga się to poprzez uruchomienie programu/kodu obsługi systemu plików (logiki) w przestrzeni użytkownika, podczas gdy moduł FUSE zapewnia jedynie "most" do rzeczywistych interfejsów jądra.
- FUSE jest szczególnie przydatny do pisania wirtualnych systemów plików. W przeciwieństwie do tradycyjnych systemów plików, które pracują z danymi na pamięci masowej, wirtualne systemy plików nie przechowują danych. Działają one jako widok lub tłumaczenie istniejącego systemu plików lub urządzenia pamięci masowej.
- Łatwa implementacja (uproszczony zestaw poleceń), błędy w implementacji nie wpływają na pracę systemu.

Architektura FUSE

Przeniesienie logiki "sterownika" obsługi systemu plików do przestrzeni użytkownika
(ma to swoje zalety ale i wady)

- Moduł jądra FUSE i biblioteka FUSE komunikują się poprzez specjalny deskryptor pliku, który jest uzyskiwany przez otwarcie pliku **/dev/fuse**.
- Plik ten może być otwierany wielokrotnie, a uzyskany deskryptor pliku jest przekazywany do wywołania systemowego mount, aby dopasować deskryptor do zamontowanego systemu plików.
- Kiedy sterownik jądra FUSE komunikuje się z demonem FUSE, tworzy strukturę **żądania** FUSE. Żądania mają różne typy w zależności od tego, jaką operację kodują.



FUSE: protokół komunikacji jądro - użytkownik

- Tabela zawiera listę wszystkich 43 typów żądań FUSE, pogrupowanych według ich semantyki. Jak widać, większość żądań ma bezpośrednie odwzorowanie na tradycyjne operacje VFS.

Group (#)	Request Types
Special (3)	init, destroy, interrupt
Metadata (14)	lookup, forget, batch_forget, create, unlink, link, rename, rename2, open, release, statfs, fsync, flush, access
Data (2)	read, write
Attributes (2)	getattr, setattr
Extended attributes (4)	setxattr, getxattr, listxattr, removexattr
Symlinks (2)	symlink, readlink
Directory (7)	mkdir, rmdir, opendir, releasedir, readdir, readdirplus, fsyncdir
Locking (3)	getlk, setlk, setlkw
Misc (6)	bmap, fallocate, mknod, ioctl, poll, notify_reply

- Żądanie **init** jest generowane przez jądro, gdy system plików jest montowany. W tym momencie przestrzeń użytkownika i jądro negocjują m.in.: wersję protokołu i zestaw wzajemnie wspieranych możliwości
- Żądanie **destroy** jest wysyłane przez jądro podczas procesu odmontowywania systemu plików. Po otrzymaniu żądania destroy, demon FUSE powinien wykonać wszystkie niezbędne czynności porządkowe. Żadne więcej żądania nie będą przychodzić od jądra dla tej sesji.
- Żądanie **interrupt** jest generowane przez jądro, jeśli żądania, które były wcześniej wygenerowane i przekazane do demona FUSE nie są już potrzebne procesowi (np. gdy proces użytkownika zablokowany na żądanie odczytu jest zakończony/anulowany). Żądania przerwania mają pierwszeństwo przed innymi żądaniami.

Struktura demona FUSE

Dokumentacja m.in.: https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201001/homework/fuse/fuse_doc.html

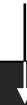
```
#define FUSE_USE_VERSION 26

#include <fuse.h>
#include <string.h>
#include <errno.h>

static const char *filepath = "/file";
static const char *filename = "file";
static const char *symlinkpath = "/link";
static const char *symlinkname = "link";
static const char *filecontent = "I'm the content of the only file
...
static int getattr_callback(const char *path, struct stat *stbuf) {
    ...
}
...
static struct fuse_operations fuse_example_operations = {
    .getattr = getattr_callback,
    .open = open_callback,
    .read = read_callback,
    .readdir = readdir_callback,
    .readlink = readlink_callback,
};

int main(int argc, char *argv[])
{
    return fuse_main(argc, argv, &fuse_example_operations, NULL);
}
```

Punkt montowania



```
$ ./fuse-example -d -s -f /tmp/example/
FUSE library version: 2.9.9
nullpath_ok: 0
nopath: 0
utime_omit_ok: 0
...
```

-d - włącz wyjście debugowania

-f - działaj na pierwszym planie (przydatne przy pracy z debuggerem). UWAGA: Gdy podana jest opcja -f, katalogiem roboczym programu Fuse jest katalog, w którym znajdował się on w chwili uruchomienia. Bez -f, Fuse zmienia katalogi na "/"

-s - uruchomienie jednowątkowe zamiast wielowątkowego (ułatwia debugowanie)

Demon FUSE: przykład 1/3

```
static int getattr_callback(const char *path, struct stat *stbuf) {  
    memset(stbuf, 0, sizeof(struct stat));  
  
    if (strcmp(path, "/") == 0) {  
        stbuf->st_mode = S_IFDIR | 0755;  
        stbuf->st_nlink = 2;  
        return 0;  
    }  
  
    if (strcmp(path, filepath) == 0) {  
        stbuf->st_mode = S_IFREG | 0777;  
        stbuf->st_nlink = 1;  
        stbuf->st_size = strlen(filecontent);  
        return 0;  
    }  
  
    if (strcmp(path, symlinkpath) == 0) {  
        stbuf->st_mode = S_IFLNK | 0777;  
        stbuf->st_nlink = 1;  
        stbuf->st_size = strlen(filename);  
        return 0;  
    }  
  
    return -ENOENT;  
}
```

Wywołanie zwrotne **getattr** jest odpowiedzialne za odczytanie metadanych podanej ścieżki, zawsze wywoływane przed jakąkolwiek operacją wykonaną na systemie plików (dobrze widać to w trybie debuggu).

Demon FUSE: przykład 2/3

```
static int open_callback(const char *path, struct fuse_file_info *fi) {  
    return 0;  
}
```

```
static int readlink_callback(const char *path, char *buf, size_t size) {  
    if (strcmp(path, symlinkpath) == 0) {  
        strncpy(buf, filename, size);  
  
        return 0;  
    }  
}
```

```
return -ENOENT; // ???  
}
```

```
static int readdir_callback(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi) {
```

```
    filler(buf, ".", NULL, 0);  
    filler(buf, "..", NULL, 0);
```

```
    filler(buf, filename, NULL, 0);
```

```
    filler(buf, symlinkname, NULL, 0);
```

```
    return 0;  
}
```

Wywołanie zwrotne **open** jest uruchamiane, gdy system żąda otwarcia pliku. W przykładzie nie mamy prawdziwego pliku, a jedynie jego reprezentację w pamięci, zaimplementujemy to wywołanie zwrotne tylko dlatego, że jest potrzebne do działania FUSE (zawsze zwróci 0, czyli OK).

Zadaniem wywołania zwrotnego **readdir** jest poinformowanie FUSE o dokładnej strukturze katalogu, do którego ma dostęp.

Wywołanie **readlink** używane jest w przypadku dostępu do dowiązania symbolicznego.

Demon FUSE: przykład 3/3

```
static int read_callback(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi) {  
  
    if (strcmp(path, filepath) == 0) {  
        size_t len = strlen(filecontent);  
        if (offset >= len) {  
            return 0;  
        }  
  
        if (offset + size > len) {  
            memcpy(buf, filecontent + offset, len - offset);  
            return len - offset;  
        }  
  
        memcpy(buf, filecontent + offset, size);  
        return size;  
    }  
  
    return -ENOENT;  
}
```

Wywołanie zwrotne **read** jest odpowiedzialne za odczytanie danych z podanego pliku, zwracamy ilość odczytanych bajtów i umieszczamy w **buf** odczytaną zawartość.

Tym razem uruchomiony jako demona

```
$ ./fuse-example /tmp/example/  
$ ls /tmp/example/  
file link  
$ ls -l /tmp/example/  
total 0  
-rwxrwxrwx 1 root root 49 Jan  1  1970 file  
lrwxrwxrwx 1 root root  4 Jan  1  1970 link -> file  
$ cat /tmp/example/file  
I'm the content of the only file available there  
$ cat /tmp/example/link  
I'm the content of the only file available there  
$ readlink /tmp/example/link  
file  
$ fusermount -uz /tmp/example  
$ ls -l /tmp/example/  
total 0
```