Rozdział 4

System plików

Rozdział ten zawiera opis struktur danych oraz funkcji systemowych Linuxa związanych z systemem plików, opis abstrakcji systemu plików (*Virtual File System*), mechanizmów buforowania danych, dostępu do plików, katalogów oraz montowania systemów plików. Termin „system plików" oznacza w kontekście niniejszego rozdziału zbiór wszystkich wyżej wymienionych zagadnień.

Autorami rozdziału są studenci kierunku Informatyka na Uniwersytecie Warszawskim:

Paulina Domagalska: sekcje 4.3.1, 4.3.2
Michał Domagała: sekcja 4.10.4, 4.10.6
Karol Droste: sekcja 4.9
Elżbieta Filipkowska: podrozdział 4.4
Bartłomiej Goldman: podrozdział 4.7
Ewelina Hernik: sekcja 4.5.2
Krzysztof Juchimiuk: sekcje 4.10.1, 4.10.5
Bartosz Kapturkiewicz: sekcje 4.3.3, 4.3.4
Paweł Kwiatkowski: podrozdział 4.11
Jakub Lenart: sekcje 4.10.2, 4.10.3
Bartłomiej Rymkowski: podrozdział 4.7
Jarosław Staniek: podrozdział 4.6[1]
Damian Wójtowicz: sekcje 4.5.1, 4.5.3
Tomasz Zieliński: wstęp, podrozdział 4.1
Michał Wiktor Żmijewski: podrozdział 4.8
Tomasz Dziki, Marcin Żukowski: podrozdział 4.2

[1]oraz redakcja całości

Spis treści

Wstęp

W poprzednich rozdziałach opisane zostały zarządzanie procesami i pamięcią. Choć te części jądra mają największy wpływ na wydajność systemu, przeciętny użytkownik może nawet nie być świadom ich istnienia, jako że nie ma na nie bezpośredniego wpływu. Inaczej dzieje się w przypadku systemu plików. Korzystanie z komputera wiąże się z manipulacją danymi, dlatego każdy dość szybko poznaje wady i zalety rozwiązań przyjętych przez autorów systemu. Głównym zadaniem stawianym przed systemem plików powinno być zapewnienie komfortu pracy i bezpieczeństwa zgromadzonych zasobów. W tym rozdziale opisujemy system plików jako zbiór mechanizmów pozwalających na manipulowanie danymi przechowywanymi w pamięci masowej, nie ograniczamy się tylko do opisu konkretnego formatu zapisu na dysku jakim jest na przykład format **ext2**. Opiszemy w jaki sposób użytkownik może korzystać z zasobów zgromadzonych w pamięci masowej, jak w Linuksie rozwiązano problem różnych formatów zapisu, oraz jak zwiększono szybkość całego środowiska. Przyjrzymy się też bliżej systemowi **ext2** oraz **proc**.

4.1 Miejsce systemu plików w jądrze Linuksa

System plików Linuksa pośredniczy w wymianie danych pomiędzy procesami (użytkownikiem) a pamięcią masową.[2] Udział jądra pozwala na ukrycie specyficznych właściwości sprzętu, zwiększenie szybkości operacji jak i zapewnienie ochrony danych zarówno przed nieautoryzowanym dostępem jaki i uszkodzeniami spowodowanymi awarią sprzętu.[3]

System plików Linuksa może posłużyć za przykład dobrego projektowania, podzielony został on bowiem na cztery kolejne warstwy z których każda powinna odwoływać się do warstwy znajdującej się poniżej, z wyjątkiem ostatniej komunikującej się bezpośrednio ze sprzętem. Taka budowa ułatwia dokonywanie zmian w jądrze zwiększając elastyczność Linuksa rozumianą jako możliwość wprowadzania nowych rozwiązań.

Te kolejne warstwy to:

- Interfejs do systemu plików
- Wirtualny system plików
- Podręczna pamięć buforowa
- Zarządzanie urządzeniami

Pierwszą warstwę tworzą funkcje systemowe, za pomocą których użytkownik, a właściwie jego procesy komunikują się z systemem. Udostępnia ona logiczny obiekt - **plik**, czyli ciąg bajtów opatrzonych nazwą. Zdefiniowany został zbiór operacji dozwolonych na plikach, można więc z nich czytać porcje bajtów, zapisywać w nich dane, przesuwać się w ich obrębie, zmieniać ich nazwę, atrybuty itp.[4] Zyski płynące z takiego jednolitego interfejsu są oczywiste.

Pliki zostały zorganizowane w hierarchiczną strukturę przy pomocy **katalogów**, razem tworzą one drzewo którego liśćmi są pliki, a węzłami katalogi. Jednoznaczna identyfikacja pliku następuje po podaniu ścieżki w drzewie i nazwy pliku.

[2] Na równi z dyskami twardymi obsługiwane są rozwiązania sieciowe, takie jak NFS.

[3] Oczywiście w zakresie na jaki pozwala dana sytuacja.

[4] Dla konkretnego pliku nie wszystkie operacje są jednak dozwolone, nie można na przykład pisać na CDRom, lub wołać lseek dla łącza nienazwanego.

Wirtualny system plików (**VFS** - *Virtual File System* lub *Virtual Filesystem Switch*) będący kolejną warstwą odpowiada za elastyczność Linuksa, objawiającą się w możliwości korzystania z plików w tak różnych formatach jak: FAT, ext2 czy NFS. Z używanym plikiem zostaje skojarzony opisujący go **i-węzeł** (*i-node*). Przechowywane są w nim informacje o właścicielu, liczniku dowiązań, czasach modyfikacji, ale przede wszystkim znajdują się tu wskaźniki do funkcji odpowiedzialnych za operacje związane z fizyczną reprezentacją pliku na nośniku.[5] Istnienie tej warstwy pozwala na tłumaczenie ogólnego interfejsu do plików na właściwe operacje, aby więc umożliwić korzystanie np. z FAT32 wystarcza tylko napisać implementacje odpowiednich funkcji. Inicjalizacja wskaźników na właściwe procedury wiąże się z procesem montowania, podczas którego do istniejącego drzewa katalogów dołączany jest korzeń drzewa plików znajdujący się na przyłączanym nośniku. Proces ten zostanie opisany w kolejnym podrozdziale.

Kolejna warstwa zwiększa wydajność systemu i ukrywa specyfikę urządzeń blokowych[6]. Żądania zapisu i odczytu generowane przez procesy dotyczą różnych porcji danych, jednak urządzenia masowe są przeważnie urządzeniami blokowymi, a więc rzeczywiste operacje zachodzą na z góry ustalonych porcjach danych - **blokach**. Podręczna pamięć buforowa udostępnia pulę **buforów** do których można pisać całe bloki z dysków. Przy okazji udaje się zminimalizować ilość rzeczywistych operacji dyskowych. Po pierwsze jeśli poprzednia operacja czytania dotyczyła mniejszej ilości danych niż rozmiar bloku, to o ile czytanie jest sekwencyjne, kolejna porcja danych prawdopodobnie znajduje się już w pamięci (w buforze do którego czytano poprzednio). Po drugie czytając z dysku kilka kolejnych bloków i umieszczając je w buforach, możemy spowodować, że przy następnym żądaniu w pamięci będą znajdować się już bufory z właściwymi danymi. Po trzecie opóźniając zapis bufora na dysk powodujemy, że kolejna modyfikacja danych będzie dotyczyła tylko obszaru pamięci, a nie urządzenia fizycznego. Ponieważ operacje przebiegające w pamięci operacyjne są wielokrotnie szybsze niż te z udziałem urządzenia fizycznego, podręczna pamięć buforowa zwiększa szybkość sytemu.

Sterowniki urządzeń zajmują się niskopoziomową obsługą żądań pisania i odczytu, zostaną one opisane w następnym rozdziale.

[5]Nie wszystkie funkcje dostępne są poprzez i-węzeł, ale to zostanie dokładniej opisane dalej.
[6]Ta warstwa nie jest wykorzystywana w przypadku np. sieciowego systemu plików.

4.2 Wirtualny system plików

W świecie systemów operacyjnych występuje olbrzymia różnorodnoś systemów plików. Właściwie każdy system operacyjny wprowadza przynajmnie jeden nowy system plików, zazwyczaj całkowicie niekompatybilny z dotychczas istniejącymi. Większość systemów operacyjnych umożliwia współpracę jedynie ze swoimi systemami plików, ewentualnie kilkoma dodatkowymi. Aby uniknąć tego problemu jednym z zamierzeń twórców systemu Linux było umożliwienie łatwej komunikacji z dowolnymi systemami plików. W tym celu wprowadzono mechanizm nazywany **wirtualnym systemem plików** , po części wzorowany na analogiczym mechaniźmie używanym w innych systemach unixowych.

Wirtualny system plików (z ang. **VFS - Virtual File System** lub **Virtual Filesystem Switch**) jest warstwą jądra umożliwiającą programom użytkowym dostęp do danych niezależnie od formatu ich przechowywania. Jest to możliwe dzięki temu, że większość systemów plików udostępnia zbliżony zestaw operacji różniących się często tylko wewnętrzną implementacją. Poprzez udostępnienie programom obsługi poszczególnych systemów plików jednolitego interfejsu VFS umożliwia różnym systemom plików wspólną pracę w ramach tego samego systemu operacyjnego ukrywając przed użytkownikiem ich rzeczywistą strukturę. Sprawia to, że użytkownik nie musi troszczyć się o to z jakim systemem plików aktualnie współpracuje.

Proces użytkowy w momencie, gdy chce uzyskać dostęp do jakiegoś pliku wywołuje jedną ze standardowych funkcji systemowych (np. open, read). Następnie żądanie to jest przekazywane do wirtualnego systemu plików i w zależności od rodzaju systemu plików w jakim jest zapisany dany plik VFS wywołuje funkcję odpowiadającą temu właśnie systemowi plików. Mechanizm ten jest wizualnie zaprezentowany na rysuku 4.1 Wirtualny System Plików.

Jedną z istotnych cech VFS jest fakt, że mimo iż udostępnia on na zewnątrz ujednolicony interfejs, w rzeczywistości wewnętrzna implementacja każdego z systemów plików może być dowolna. Dzięki temu możemy korzystać z bardzo różnorodnych systemów plików, w szczególności takich, które nie korzystają z żadnego fizycznego urządzenia, jak na przykład system plików **proc** . Innym ciekawym systemem plików jest **NFS** - Network File System (z ang. Sieciowy System Plików), umożliwiający dostęp do danych przechowywanych na innych komputerach w identyczny sposób jak w przypadku dysku lokalnego.

4.2.1 Rejestracja systemu plików w jądrze

Pracując w Linuxie możemy używać systemów plików aktualnie zarejestrowanych w systemie. Aby dowiedzieć się, jakie systemy plików są w danej chwili zarejestrowane należy użyć polecenia

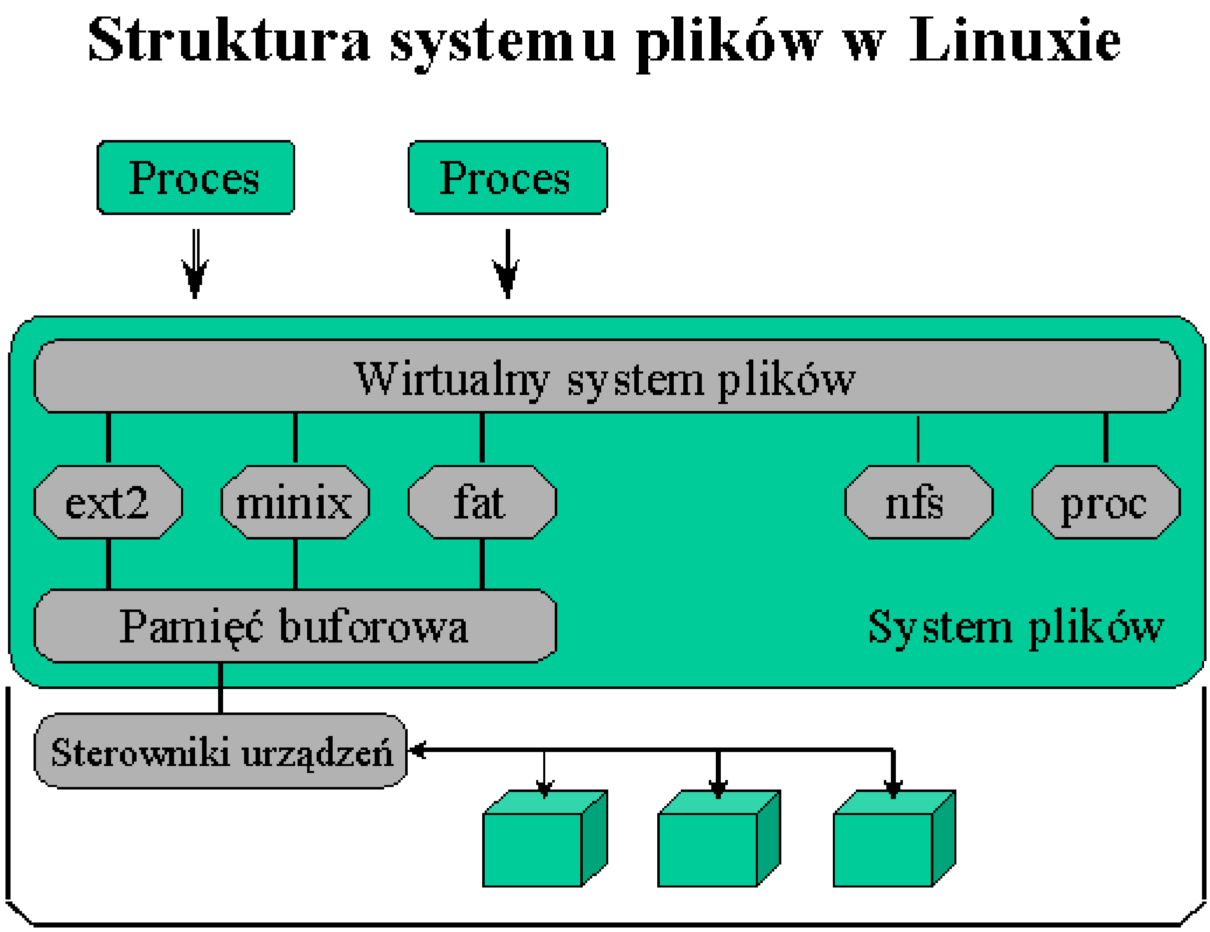
`cat /proc/filesystems`

Aby dodać do jądra obsługę nowego systemu plików należy go zarejestrować używając funkcji zawartej w `fs/super.c`

`int register_filesystem(struct file_system_type * fs)`

Każdemu zarejestrowanemu systemowi plików jest przyporządkowana struktura `file_system_type` zdefiniowana w `include/linux/fs.h` :

```
struct file_system_type {
const char *name;
int fs_flags;
```



Rysunek 4.1: Wirtualny system plików

```
struct super_block *(*read_super) (struct super_block *, void *, int);
struct file_system_type *next;
}
```

name : nazwa systemu plików

flags : flagi (np. `FS_REQUIRES_DEV` -potrzebuje urządzenia, `FS_NO_DCACHE`-nie używana jest pamięć podręczna)

read_super : funkcja wywoływana przy montowaniu systemu plików

next : do wewnętrznego zastosowania przez VFS, powinno być zainicjowane na `NULL`

Efektem wywołania `register_filesystem` jest dodanie podanej struktury do listy zarejestrowanych systemów plików (przechowywanej w jądrze) o początku wskazywanym przez

`static struct file_system_type *file_systems`

Funkcja `read_super` zawarta w `file_system_type` jest inna dla każdego systemu plików.

Wywoływana jest w momencie żądania zamontowania konkretnego systemu plików. Jej zadaniem jest stworzenie **bloku specjalnego** zawierającego informacje na temat jednej instancji systemu plików.

Funkcja jako parametry otrzymuje następujące informacje:

`struct super_block *sb` : struktura bloku specjalnego, Jest ona częściowo zainicjowana przez VFS, pozostałe pola musi wypełnić `read_super`

`void *data` : opcjonalne opcje montowania, standardowo przekazywane jako ciąg znaków ASCII

`int silent` : flaga oznaczająca czy informować o błędach podczas montowania

`read_super` sprawdza czy podane w `sb` urządzenie zawiera system plików odpowiedni dla tej funkcji, jeżeli tak, zwraca wskaźnik do uzupełnionego bloku specjalnego, w przeciwnym wypadku zwraca `NULL`.

4.2.2 Struktura `super_block` i operacje na niej

Blok specjalny (`struct super_block`) zawiera różnorodne informacje na temat zamontowanego systemu plików, m.in. urządzenie obsługujące dany system plików, rodzaj systemu plików, rozmiar bloku, struktury do synchronizacji dostępu, czas dokonanej ostatnio zmiany a także informacje specyficzne dla konkretnej implementacji. Jego pełna struktura jest zdefiniowana w `include/linux/fs.h`.

Najciekawszym elementem `super_block` jest pole `s_op` zawierające kolejny poziom implementacji systemu plików, czyli operacje na bloku specjalnym.

Funkcje dostępne na bloku specjalnym są zawarte w strukturze:

```
struct super_operations {
    void (*read_inode) (struct inode *);
    void (*write_inode) (struct inode *);
    void (*put_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    int (*notify_change) (struct dentry *, struct iattr *);
    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 *);
    void (*umount_begin) (struct super_block *);
}
```

`read_inode` - odczytuje z zamontowanego systemu plików inode o numerze zawartym w `inode->i_ino` (pole to jest zainicjowane przez VFS). Dodatkowo ustawia pole `inode->i_op` na odpowiedni zestaw funkcji operujący na inode.

`write_inode` - zapisuje strukturę inode na urządzenie.

`put_inode` - wywoływane przez VFS gdy inode przestaje być potrzebne. Jeżeli wartość `i_nlink` wyniesie zero, nastąpi fizyczne usunięcie pliku i zwolnienie zajmowanych przez niego bloków

`delete_inode` - usunięcie inode

`notify_change` - informuje, że nastąpiła zmiana w strukturze inode. Przydatne np. przy implementacji NFS, kiedy zmiana inode lokalnego powinna pociągnąć za sobą zmianę na zdalnym systemie

`put_super` - wywoływana gdy VFS chce zwolnić blok specjalny np. przy odmontowywaniu. Dodatkowo jeżeli implementacja obsługi danego systemu plików była załadowana jako moduł, jest zmniejszony licznik jego użyć.

`write_super` - zapisanie bloku specjalnego na dysk. Używana jest m.in. do synchronizowania urządzenia (nie dotyczy systemów plików tylko do odczytu). Również w momencie odmontowywania systemu plików, o ile pole `s_dirt` struktury `super_block` jest ustawione, wywoływana jest ta funkcja.

`statfs` - wywoływana w celu pobrania statystyk systemu plików. Struktura `statfs` zawiera informacje o rozmiarze bloków, liczbie wolnych bloków, liczbie plików itp.

`remount_fs` - wywoływana w momencie przemontowywania systemu plików. Zazwyczaj jej rola sprowadza się do zmiany statusu systemu plików oraz odtworzenia jego spójności.

`clear_inode` - wywoływane przez system plików kiedy struktura `inode` przestaje być potrzebna.

`umount_begin` - wywoływana w momencie gdy chcemy odmontować system plików.

Funkcje te pozwalają na operacje na poszczególnych strukturach inode, na zapisywanie bloku specjalnego oraz na uzyskiwanie informacji na temat systemu plików. To właśnie ta warstwa zajmuje się przekształcaniem konkretnej reprezentacji bloku specjalnego i struktur inode do ujednoliconej postaci w pamięci i na odwrót. Dokładnie rzecz biorąc struktury inode i super_block (albo ich odpowiedniki) nie muszą fizycznie istnieć, implementacja danego systemu plików może tworzyć je w zależności od wewnętrznych potrzeb (np. system plików `proc`).

4.2.3 Struktura `inode` i operacje na niej

Każdemu używanemu aktualnie plikowi w dowolnym systemie plików odpowiada przechowywana w pamięci struktura `inode` . Zawiera ona szczegółowe informacje o tym pliku, m.in. o właścicielu, prawach dostępu, rozmiarze, czasie ostatniej modyfikacji oraz adresy bloków z danymi. Dodatkowo zawiera pole `i_sb` będące wskaźnikiem na odpowiedni blok specjalny, opisany wcześniej, oraz pole `i_op` typu `struct inode_operations` zawierające wskaźnik na zbiór funkcji umożliwiających manipulacje na strukturze inode. Pełna struktura `inode` jest zawarta w pliku `include/linux/fs.h`.

Struktura `inode_operations` zawiera zestaw funkcji umożliwiających operacje na strukturze `inode` w sposób odpowiedni dla danego systemu plików:

```
struct inode_operations {
    struct file_operations * default_file_ops;
    int (*create) (struct inode *,struct dentry *,int);
    struct dentry * (*lookup) (struct inode *,struct dentry *);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,int);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,int,int);
```

```
    int (*rename) (struct inode *, struct dentry *,
            struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char *,int);
    struct dentry * (*follow_link) (struct dentry *, struct dentry *, unsigned int);
    int (*readpage) (struct file *, struct page *);
    int (*writepage) (struct file *, struct page *);
    int (*bmap) (struct inode *,int);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*smap) (struct inode *,int);
    int (*updatepage) (struct file *, struct page *, unsigned long, unsigned int, int);
    int (*revalidate) (struct dentry *); }
}
```

`default_file_ops` - jest to wskaźnik na strukturę typu `file_operations` , która zawiera zbiór operacji dostępnych na otwartych plikach.

`create` - używana przy fizycznym tworzeniu plików. Wyszukuje wolną strukturę inode poprzez `get_empty_inode` , wypełnia ją danymi odpowiednimi dla konkretnego systemu plików, uzupełnia strukturę katalogu.

`lookup` - odnalezienie dla podanego pliku struktury inode. Nazwa pliku jest podana w strukturze `dentry` . Po odnalezieniu jest wywoływana funkcja `d_add()` wstawiająca `inode` do `dentry` . Dodatkowo w strukturze `inode` zwiększany jest licznik odwołań `i_count` . Jeżeli nie odnaleziono odpowiedniej nazwy do struktury `dentry` jest wstawiany `inode` równy NULL. VFS w sposób specjalny traktuje wywołanie tej funkcji dla nazwy "..". W momencie przejścia przez punkt zamontowania systemu plików wywołuje odpowiednią implementację.

`link` - używana przy tworzeniu tzw. sztywnych łączy (hard links). Przed jej wywołaniem sprawdza się czy obydwie struktury `dentry` znajdują się na tym samym urządzeniu i czy proces bieżący ma prawo zapisu do danego katalogu.

`unlink` - używana do usunięcia `inode` . Przed wywołaniem sprawdza się czy bieżący proces ma odpowiednie uprawnienia.

`symlink` - używana przy tworzeniu łącza symbolicznego.

`mkdir` - używana przy tworzeniu podkatalogów. Funkcja po sprawdzeniu możliwości utworzenia podkatalogu w danym miejscu musi przydzielić nową strukturę inode oraz blok na fizycznym nośniku, oraz inicjuje wpisy dla pozycji "."i "..". Podobnie jak w przypadku większości funkcji VFS sprawdza uprawnienia bieżącego procesu.

`rmdir` - używana przy usuwaniu podkatalogów. Funkcja sprawdza czy aktualnie usuwany katalog jest pusty oraz czy nie jest przypadkiem używany przez jakiś proces. Standardowo sprawdzane są prawa dostępu.

`mknod` - używana przy tworzeniu urządzeń blokowych lub znakowych, łącz nazwanych FIFO lub gniazd.

`rename` - używana przy zmianie nazwy pliku.

`readlink` - używana przy odczytywaniu łączy symbolicznych. Zadaniem jej jest umieszczenie w buforze podanym przy wywołaniu ścieżki do pliku, na który wskazuje łącze. Przy przekroczeniu rozmiaru bufora ścieżka jest ucięta do zakresu podanego przy wywołaniu. Funkcja sprawdza czy podana struktura `dentry` zawiera łącze symboliczne, jeżeli nie, zwraca błąd.

`follow_link` - rozwinięcie łącza symbolicznego - funkcja umieści w docelowej strukturze `dentry` pozycję, na którą wskazuje łącze.

`bmap` - używana do zamapowania bloku pliku pod adres w przestrzeni adresowej użytkownika.

`truncate` - skrócenie, ale też wydłużenie pliku do dowolnej(dozwolonej) wielkości. W podanej strukturze `inode` jest już zawarta nowa długość pliku. Funkcja dodatkowo zwalnia bloki nie używane przez plik.

`permission` - sprawdza czy są spełnione podane prawa dostępu. Jako argument może otrzymać kombinacje wartości `MAY_READ` , `MAY_WRITE` lub `MAY_EXEC` .

`smap` - używana przy tworzeniu plików wymiany w systemie plików UMSDOS

4.2.4 Inne operacje na systemach plików

W pliku `fs/super.c` występują też inne funkcje służące do obsługi systemów plików. Przykładowe to:

`int unregister_filesystem(struct file_system_type * fs)`

Funkcja wyrejestrowujaca system plików

`struct file_system_type *get_fs_type(const char *name)`

Funkcja podaje dane na temat systemu plików na podstawie jego nazwy oraz globalnej listy systemów plików `file_systems`

`int get_filesystem_list(char * buf)`

Funkcja wpisuje do bufora buf informacje tekstowe na temat poszczególnych systemów plików(nazwa, flaga `FS_REQUIRES_DEV`), zwraca liczbę elementów.

4.3 System plików ext2

4.3.1 Logiczna struktura dysku.

W systemie plików `ext2` dysk jest podzielony na blok startowy oraz grupy bloków.

1. Blok startowy.

   System plików zaczyna sie od bloku startowego, w którym mogą być przechowywane informacje wykorzystywane przez system operacyjny podczas uruchamiania. Istnieje on niezależnie od tego, czy będzie używany podczas ładowania systemu, czy też nie.

2. Grupy bloków.

   W skład każdej grupy bloków wchodzą:

   - blok specjalny (ext2_super_block)
   - deskryptory grupy
   - bitmapa bloku
   - bitmapa struktury i-węzła
   - tablica struktur i-węzła
   - bloki danych.

   Część danych w bloku specjalnym dotyczy całego systemu plików, co w razie potrzeby może umożliwić jego odtworzenie. Pozostałe informacje dotyczą grupy bloków, do której blok specjalny przynależy, są to m.in.: liczba i-węzłów oraz bloków w grupie, czas ostatniego zapisu.

   W deskryptorach grupy znajduja się wiadomości na temat grup bloków, m.in. liczba wolnych bloków i i-węzłów w grupie, liczba katalogów w grupie (dzięki tej ostatniej informacji katalogi są rozmieszczane po grupach równomiernie).

   Bloki zawierające tablicę i-węzlow zawierają i-węzły. Każdemu plikowi odpowiada dokładnie jeden i-węzeł, mający swój unikatowy (w danym urządzeniu) numer, oraz pola wskazujące na bloki danych, pośrednie bloki danych, podwójnie pośrednie bloki danych i potrójnie pośrednie bloki danych, które wskazują na bloki z danymi pliku lub na bloki w których znajdują się informacje gdzie danych szukać. Poza tym w i-węźle znajduja sie wiadomości o prawach dostępu do pliku.

   W blokach danych zapisane są pliki oraz bloki pośrednie.

4.3.2 Obsługa i-węzłów w pamięci

Przechowywane w pamięci kopie i-węzłów posiadają dodatkowe - w stosunku do i-węzłów na dysku - informacje. Pole `i_lock` określa, czy została założona blokada; pole `i_count` mówi, jaka jest liczba przydzielonych procesom wcieleń plików (licznik odwołań); na podstawie pola `i_dirt` można stwierdzić, czy i-węzeł znajdujący sie w pamięci był modyfikowany. Każdy i-węzeł znajduje się na dwukierunkowej liście cyklicznej, w związku z tym posiada pola `i_next, i_prev` , które wskazujś odpowiednio na jego następnika i poprzednika. Poza tym i-węzły są przechowywane w tablicy mieszającej (dzięki temu zwiększą się dostęp do i-węzła); miejsce w tablicy wyznacza numer urządzenia oraz numer i-węzła. Kolizje obsługiwane są przy pomocy listy dwukierunkowej (pola `i_hash_next`, `i_hash_prev`).

Są dwa stany i-węzła: wolny oraz aktywny. W tablicy mieszającej znajdują się tylko i_węzły aktywne, czyli takie, które są aktualnie wykorzystywane przez jakiś proces. Pozostałe są nazwane wolnymi - mają licznik odwolań równy zero.

Poniższe funkcje slużą do operowania i-węzłami w pamięci:

- `i_get()`
- `i_put()`
- `i_namei()`

Funkcja `i_get()` służy do przydziału struktury i_węzła znajdującej się w pamięci. Odbywa się to według następujących zasad:

1. Korzystając z numeru i-węzła oraz numeru urządzenia jądro przegląda tablicę mieszającą.
2. Jeśli i-węzeł został znaleziony, czyli znajduje się już w pamięci,to zostaje zwiększony jego licznik odwołań o 1.
3. W przeciwnym przypadku sprawdza się czy istnieje już kopia szukanego i-węzła na liście wolnych. Jeśli tak, to jest on przenoszony z listy wolnych do tablicy mieszającej, a jego licznik odwołań zostaje zwiększony o 1. Jeśli nie, to jądro wybiera wolną strukturę z listy, zapisuje w niej nowy numer i-węzła i systemu plików, przenosi do tablicy mieszającej oraz wczytuje odpowiedni i-węzeł z dysku. Gdy lista wolnych jest pusta, wówczas `i_get()` zwraca błąd.

W celu wczytania i-węzła z dysku jądro oblicza numer bloku oraz jego w nim położenie:
numer bloku = [X/Y] + (numer bloku zawierającego początek listy i-węzłow)
położenie w bloku = (X mod Y) * (rozmiar i-węzła na dysku)
Gdzie:
X=numer i-węzła -1;
Y=liczba i-węzłów w bloku;
symbol [x] oznacza cześć całkowitą liczby x.
Powyższe wzory zostały podane za M. Bachem.
Funkcją `i_put()` można zwolnić i-węzeł znajdujżcy się w pamięci:

1. Zmniejszamy licznik odwołań danego i-węzła o 1.
2. Jeśli licznik odwołań wynosi 0, to sprawdzamy czy liczba dowiazań do pliku jest równa 0. Jeśli tak, zwalniane są wszystkie bloki danych związanych z plikiem oraz i-węzeł. Wprzeciwnym przypadku sprawdzamy pole `i_dirt` , w którym zapisana jest informacja, czy i-węzeł byl modyfikowany i jeśli byly zmiany, to zapisujemy i-węzeł na dysk.

Funkcja namei() przekształca nazwę scieżkową pliku na i-węzeł. W tym celu wykorzystuje strukturę katalogu - czyli pliku, w ktorym zapisane są pary: numer i-węzła i nazwa pliku znajdującego się w tym katalogu.

1. Na początku jądro sprawdza, czy nazwa scieżkowa pliku rozpoczyna się od korzenia, czy od symbolu bieżącego katalogu, a następnie funkcja `i_get()` pobiera odpowiednią kopie i-węzeła z pamięci; kopia ta jest przechowywana na zmiennej XXX.

2. Dopóki w nazwie scieżkowej znajdują się jeszcze składowe, pobiera składową, a następnie przegląda zawierający ją katalog (jego i-węzeł mamy na zmiennej), w poszukiwaniu składowej. Jeśli jej nie znajdzie, to koniec. W przeciwnym przypadku pobiera numer jej i-węzła, zwalnia i-węzeł znajdujący się na zmiennej XXX (funkcja `i_put()`), pobiera i-węzeł z pamięci dla składowej (numer dla niej znamy z katalogu), który będzie odtąd pamiętany na zmiennej XXX.

3. Nie ma już więcej składowych w nazwie scieżkowej, a zatem szukany i-węzeł znajduje się na zmiennej XXX.

4.3.3 Obsługa i-węzłów na dysku

Informacje ogólne

W systemie plików `ext2` jeden plik na dysku reprezentowany jest przez jeden **i-węzeł** dyskowy,który składa się częściowo z pól takich jak i-węzeł w pamięci,np.

- typ (plik danych/plik specjalny znakowy/plik specjalny blokowy/katalog/łącze),
- id właściciela,
- rozmiar pliku,
- id grupy,
- ilość dowiązań,czy
- ilość zajmowanych bloków.

do tego dochodzi tablica z adresami bloków dyskowych:

- 12 adresów bloków zawierających dane [7]
- 1 adres bloku zawierającego adresy bloków zaw.dane (tzw.blok pojedynczy pośredni)
- 1 adres bloku zaw.adresy j.w. (tzw.blok podwójny pośredni)
- 1 adres bloku zaw.adresy j.w. (tzw.blok potrójny pośredni)

oraz prawa dostępu do pliku.

Blok to n-bajtowa tablica zapisana na dysku.Jeden blok może zawierać do 256 adresów bloków, zatem w sumie plik może zawierać 16843020 bloków z danymi.

Przypomnieć należy,że struktura logiczna dysku wygląda tak:

Boot sector / grupa 1 / grupa 2 / grupa 3 / ... / grupa i / ...

[7] w systemie UNIX - 10

Gdzie grupa to umieszczone kolejno po sobie:
super_blok / Deskryptory wszystkich grup / Bitmapa zajętości bloków grupy / Bitmapa zajętości i-węzłów grupy / tablica i-węzłów / bloki z danymi...

Bitmapy zajętości to tablice bitów,z których każdy zawiera informację czy dany i-węzeł,bądź blok jest zajęty.Fizycznie kazda grupa zawiera też informacje o całym
systemie plików,co jest przydatne w przypadku jakichkolwiek błędów obsługi dysku.
super_blok zawiera informacje o systemie jak np.liczba wolnych bloków i i-węzłów,oraz pole `s_lock` - semafor binarny służący do blokowania dostępu do danych systemowych,który jest wykorzystywany przez funkcje opisane poniżej. Podział na grupy przyspiesza operacje na dysku,ponieważ pliki z jednego katalogu (tj.i-węzły i bloki) zostają,jeśli jest miejsce,umieszczone w jednej grupie.

Należy dodać,że deskryptor grupy zawiera też następujące informacje:

- nr bloku z bitmapą zajętości bloków grupy
- nr bloku z bitmapą zajętości i-węzłow
- nr pierwszego bloku z tabl.i-węzłów
- liczniki wolnych i-węzłów i bloków.

Tworzenie pliku polega na pobraniu z pamięci i dysku i-węzła i wypełnieniu go. W miarę pisania do tego pliku przypisywane mu są bloki,czyli miejsce na dysku.

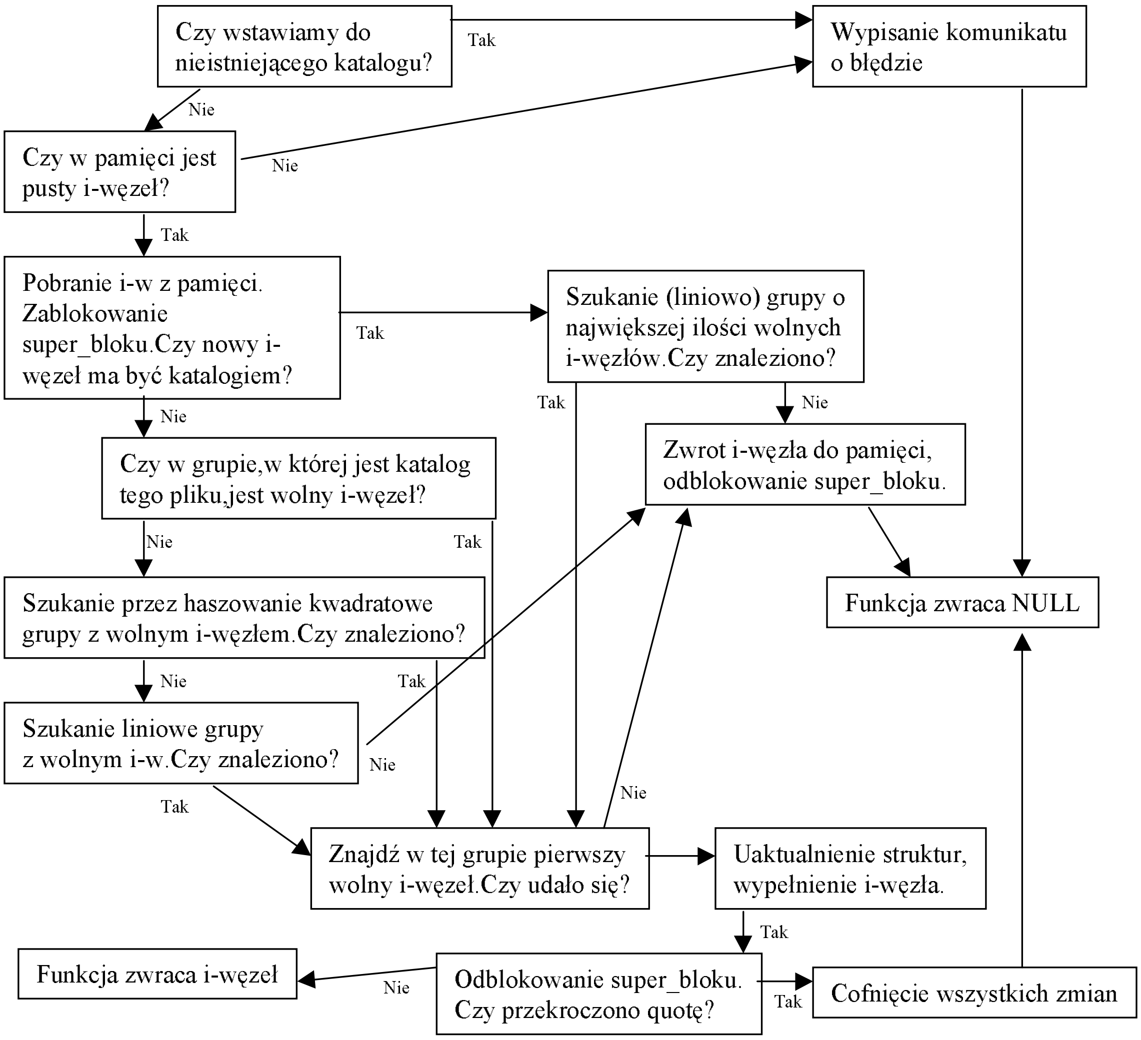
Tu opisane zostaną dwie najważniejsze funkcje zajmujące się przydziałem i zwalnianiem i-węzłów na dysku.Kody źródłowe tych funkcji, tj. `ext2_new_inode` i `ext2_free_inode` a także ich funkcji pomocniczych znajdują się w pliku **fs/ext2/ialloc.c**.

Przydział i-węzła na dysku

Przydziałem i-węzła zajmuje się funkcja `ext2_new_inode`.Pobiera ona informacje:
jaki plik i gdzie (w którym katalogu) ma stworzyć,po czym sprawdza,czy stworzenie pliku jest możliwe.Jeśli tak,tworzony jest nowy plik wybranego typu.Zaletą funkcji jest to, że i-węzeł dla nowego pliku wyszukiwany jest najpierw w grupie,w której jest jego katalog,co później znacznie przyspiesza operacje na dysku.Tu ujawnia się siła podziału na grupy (pliki z jednego katalogu znajdują się bliżej siebie). Jeśli użytkownik,lub jego grupa,ma włączoną quotę (ograniczenia dotyczące ilości i wielkości plików użytkownika/grupy),przed utworzeniem nowego pliku następuje sprawdzenie, czy to utworzenie nie spowoduje przekroczenia ograniczeń.

Zwolnienie i-węzła z dysku

Zwalnianiem i-węzłów zajmuje się funkcja `ext2_free_inode`, znacznie prostsza
od poprzedniej.Najpierw sprawdzane jest,czy można już zwolnić i-węzeł (tj.czy jest on jeszcze wykorzystywany - świadczy o tym m.in.niezerowa ilość dowiązań). Później następuje zwolnienie i-węzła przez zmiany w odpowiednich deskryptorach i mapach.



Rysunek 4.2: Algorytm ext2_new_inode

Algorytm 1 Sprawdzenie quoty

Przypisanie nowemu i-węzłowi adresów dwóch struktur `dquot` zawierających dane o quotach jego użytkownika (I struktura) i jego grupy (II struktura) (realizowane przez funkcję `dquot_initialize()`,patrz **fs/dquot.c**).
wykonuj w pętli
  Dla każdej z tych struktur (tj.dla użytkownika/grupy):
  Zablokowanie struktury
  jeśli ilość i-węzłów użytkownika/grupy + 1 > max.ilość i-węzłów dozwolona dla użytk./gr. **to**
    Odblokowanie struktury,wyjście z błędem
  koniec jeśli
  Dodanie 1 do ilości i-węzłów użytkownika/grupy.
  Odblokowanie struktury. (realizowane przez funkcję `dquot_alloc_inode()`, patrz **fs/dquot.c**).
koniec pętli

4.3.4 Obsługa bloków na dysku

Kody źródłowe funkcji `ext2_new_block` oraz `ext2_free_blocks` a także ich funkcji pomocniczych znajdują się w pliku **fs/ext2/balloc.c**.

Przydział bloku dyskowego

Przydzielaniem bloków dyskowych zajmuje się funkcja `ext2_new_block` Jednym z jej parametrów jest `goal` - numer bloku,który najlepiej byłoby zająć. Jeśli akurat nie można,funkcja szuka wolnych bloków gdzie indziej. Przydzielanie z wyprzedzeniem ma związek z **buforowaniem**,opisanym kilka stron dalej. Zaletą algorytmu przydziału bloków jest fakt,że najpierw szuka się wolnych bloków w podanym miejscu,więc można sensownie rozplanować strukturę logiczną dysku (np.ustawić bloki danego pliku tak,by odległości między nimi były jak najmniejsze, co znacznie przyspiesza korzystanie z dysku). Do pewnej ilości bloków ma dostęp tylko administrator.Umożliwia to ponowne uruchomienie systemu w przypadku,gdyby np.wszystkie bloki przeznaczone dla użytkowników zostały zajęte i system przestał działać prawidłowo.

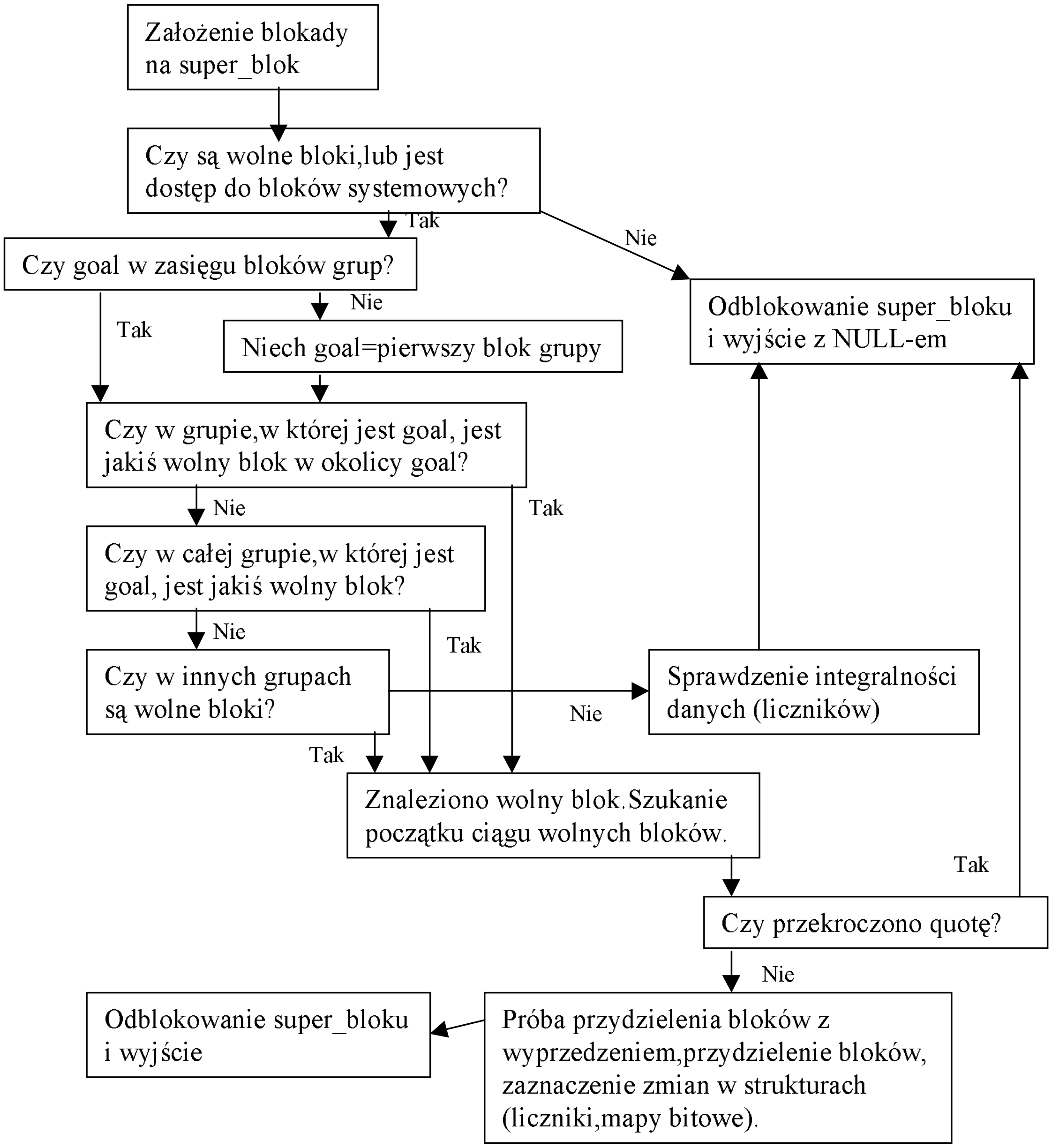
Zwolnienie bloku dyskowego

Zwalnianiem bloków dyskowych zajmuje się funkcja `ext2_free_block`. Na początku sprawdza ona pewne warunki (m.in.czy nie zwalniamy danych systemowych,co mogłoby doprowadzić do katastrofy),po czym zwalnia bloki zmieniając odpowiednie wartości w mapach i deskryptorach.

Algorytm 2 ext2_new_inode

Dane wejściowe: wskaźnik na i-węzeł odpowiadający katalogowi, w którym chcemy utworzyć nowy plik,
typ (`mode`) pliku (dla katalogu funkcja zachowuje się inaczej niż dla innych typów), err - wskaźnik na zmienną zawierającą numer błędu.
Wynik: wskaźnik do i-węzła przypisanego nowemu plikowi,lub NULL (wtedy w `*err` - kod błędu)

jeśli wstawianie do nieistniejącego katalogu **to**
  Błąd.Funkcja zwraca NULL.
koniec jeśli
jeśli w pamięci nie ma pustego i-węzła **to**
  Błąd.Funkcja zwraca NULL.
koniec jeśli
Pobranie pustego i-węzła z pamięci
Założenie blokady na super_blok
jeśli przydzielamy i-węzeł dla katalogu **to**
  Przejście liniowo wszystkich grup w poszukiwaniu tej o największej
  ilości wolnych i-węzłów.
  jeśli nie znaleziono takiej grupy **to**
    Błąd.Odblokowanie super_bloku,zwrot i-węzła do pamięci,funkcja zwraca NULL.
  koniec jeśli
w przeciwnym przypadku
  jeśli w grupie,w której jest katalog nowego pliku,nie ma wolnego i-węzła **to**
    Szukanie przez haszowanie kwadratowe grupy z wolnym i-węzłem
    jeśli Nie znaleziono takiej grupy **to**
      Szukanie liniowe grupy z wolnym i-węzłem
      jeśli Nie znaleziono takiej grupy **to**
        Błąd.Odblokowanie super_bloku,zwrot i-węzła do pamięci,funkcja zwraca NULL.
      koniec jeśli
    koniec jeśli
  koniec jeśli
koniec jeśli
Szukanie pierwszego wolnego i-węzła w znalezionej grupie.
jeśli nie znaleziono takiego i-węzła **to**
  Błąd.Wypisanie komunikatu o uszkodzonym deskryptorze.
  Zwrot i-węzła do pamięci,odblokowanie super_bloku,funkcja zwraca NULL
koniec jeśli
Zaznaczenie,że dokonano zmian w grupie i super_bloku
Wypełnienie i-węzła
Uaktualnienie struktur grupy
Odblokowanie super_bloku
jeśli przekroczono quotę **to**
  Błąd.Cofnięcie wszysztkich operacji,funkcja zwraca NULL.
koniec jeśli



Rysunek 4.3: Algorytm ext2_new_block

Algorytm 3 ext2_free_inode

Dane wejściowe: wskaźnik na i-węzeł
Wynik: nic
  jeśli niemożliwe jest zwolnienie i-węzła (np.i-węzeł nie istnieje,ma dowązania,lub nie istnieje urządzenie,na którym ten i-węzeł powinien się znajdować) **to**
    funkcja kończy działanie.
  koniec jeśli
  Założenie blokady na super_blok.
  Znalezienie numeru grupy i-węzła i jego numeru w tej grupie.
  Zwolnienie i-węzła.
  Zmiana w mapie zajętości i-węzłów w tej grupie.
  Zmiany w deskryptorze grupy (powiększenie ilości wolnych i-węzłów.
  jeśli zwalniany i-węzeł był katalogiem **to**
    Zmniejszenie liczby katalogów w grupie.
  koniec jeśli
  Oznaczenie dokonania zmiany w systemie.
  Odblokowanie super_bloku.

4.4 Podręczna pamięć buforowa

4.4.1 Struktury danych

Pliki przechowywane są na urządzeniach pamięci masowej (dysk, CD-ROM). Jądro zezwala procesom na odczyt danych zapisanych na takim urządzeniu, oraz zapisanie nowych.

Aby poprawić przepustowość systemu, jądro próbuje zminimalizować częstość transmisji danych poprzez utrzymywanie w pamięci głównej puli buforów zwanych **podręczną pamięcią buforową** (ang. *buffer cache*). W bufarach tych przechowywana jest zawartość bloków dyskowych, do których ostatnio odwoływały się procesy. Gdy jądro ma przeczytać dane z dysku, próbuje najpierw czytać z pamięci podręcznej. Jeżeli znajduje się w niej bufor zawierający dane z odpowiedniego bloku dyskowego, nie trzeba ich sprowadzać. W ten sposób udaje się uniknąć operacji wejścia-wyjścia, która byłaby niezbędna, gdyby jądro nie utrzymywało puli wewnętrznych buforów. Jeśli jadnak odpowiedniego bloku nie ma w pamięci podręcznej, jądro czyta je z dysku, oraz zapamiętuje w buforze na wypadek gdyby w niedługim odstępie czasu ten sam, bądź inny proces chciał ponownie odwołać się do tych danych.

Dane w buforze odpowiadają danym z logicznego bloku dyskowego w systemie plików i są identyfikowane za pomocą *numeru urządzenia* i *numeru bloku*. Odwzorowanie to jest tymczasowe, tzn. obowiązuje do momentu, gdy jądro zdecyduje o odwzorowaniu innego bloku z dysku w ten sam bufor.

Jeżeli dane z pewnego bloku dyskowego znajdują się w pamięci podręcznej, to zawiera je conajwyżej jeden bufor, tzn. żaden logiczny blok systemu plików nie może być odwzorowany w więcej niż jeden bufor. Gdyby możliwa była sytuacja, w której dwa różne bufory zawierają dane tego samego bloku dyskowego, jądro miałoby kłopot ze stwierdzeniem, który z tych buforów zawiera aktualne dane, a ostateczna zawartość bloku na dysku zależałaby od tego w jakiej kolejności jądro zapisałoby je na dysk.

Aby jądro sięgając do bloku dyskowego nie musiało przeszukiwać całej puli buforów, organizuje

Algorytm 4 ext2_new_block

Dane wejściowe: wskaźnik na i-węzeł,
`goal`,
ile bloków przydzielić z wyprzedzeniem,
numer pierwszego bloku do przydziały z wyprzedzeniem,
`err` - wskaźnik na zmienną zawierającą numer błędu.
Wynik: Numer przydzielonego bloku,lub zero (wtedy na `*err` - numer błędu)

Założenie blokady na super_blok.
jeśli ilość zarezerwowanych bloków >= ilość wolnych bloków i nie ma dostępu do zasobów administratora **to**
  Odblokowanie super_bloku.Funkcja zwraca 0.
koniec jeśli
jeśli `goal` poza zasięgiem bloków grup **to**
  Niech `goal`=numer pierwszego bloku grupy i-węzła.
koniec jeśli
jeśli w grupie w której jest blok o numerze `goal` nie ma wolnych bloków w jego okolicy **to**
  jeśli w pozostałej części grupy też nie ma wolnego bloku **to**
    jeśli w innych grupach nie ma wolnych bloków **to**
      Sprawdzenie integralności danych (możliwy błąd licznika bloków).
      Odblokowanie super_bloku,funkcja zwraca 0.
    koniec jeśli
  koniec jeśli
koniec jeśli
Znaleziono wolny blok.Szukanie początku ciągu wolnych bloków.
jeśli przekroczono quotę **to**
  funkcja zwraca 0.
koniec jeśli
Jeśli było takie zadanie,próba przydzielenia bloków z wyprzedzeniem (ıprealloc)
Przydzielenie bloku i zaznaczenie zmian w deskryptorach i mapach
zajętości grupy oraz w super_bloku.Odblokowanie super_bloku.

Algorytm 5 ext2_free_blocks

Dane wejściowe: wskaźnik na i-węzeł,
numer bloku początkowego,
ilość bloków
Wynik: nic
Założenie blokady na super_blok.
jeśli próba zwolnienia bloków poza obrębem grup,lub nie ma urządzenia, na którym powinien być ten blok **to**
Błąd - odblokowanie super_bloku i zakończenie działania funkcji.
koniec jeśli
Wczytanie mapy bitowej bloków.
jeśli próba zwolnienia bloków z danymi systemowymi **to**
Błąd - odblokowanie super_bloku i zakończenie działania funkcji.
koniec jeśli
Uaktualnienie liczników i mapy bitowej bloków,zaznaczenie zmian.
Odblokowanie super_bloku.

je w odrębne **kolejki mieszające**. Argumentami funkcji mieszającej są parametry za pomocą których identyfikowany jest bufor, tzn. *numer urządzenia* i *numer bloku*. Bufory w obrębie jednej kolejki mieszającej połączone są w listę cykliczną z podwójnymi dowiązaniami. Jądro utrzymuje tablice kolejek mieszających, której indeksami są wszystkie możliwe wartości funkcji mieszającej. Rozmiar tej tablicy zależy od rozmiaru dostępnej pamięci i jest ustalany przez administratora w czasie generowania systemu.

Funkcja mieszająca powinna w miarę równomiernie rozmieszczać bufory w zbiorze kolejek, oraz ze względów wydajnościowych powinna być prosta. (W opisywanej tu wersji jądra 2.2.12 liczona jest najpierw różnica symetryczna numeru urządzenia i numeru bloku, a następnie liczona jest logiczna koniunkcja bitowa otrzymanej wartości z pewną maską, której w czasie generowania systemu nadawana jest wartość równa liczbie kolejek haszujących).

```
#define _hashfn(dev,block) (((unsigned)(HASHDEV(dev)^block)) & bh_hash_mask)
```

Ponieważ blok dyskowy nie może być jednocześnie odwzorowany w dwa różne bufory, a funkcja mieszająca każdej kombinacji numeru urządzenia i bloku przyporządkowuje jednoznacznie kolejkę mieszającą, to jeżeli tylko blok dyskowy X jest odwzorowany do pewnego bufora X' w pamięci podręcznej, bufor ten może się znajdować tylko w jednej kolejce i może być tylko jeden. Dlatego gdy jądro chce sięgnąć do danych z dysku, oblicza najpierw numer kolejki haszującej, w jakiej może znajdować się bufor zawierający dane odpowiedniego bloku, a następnie przegląda tę kolejkę w nadziei, że blok ten jest już w pamięci głównej. Jeśli okaże się, że na liście mieszającej nie było szukanego bloku, jądro kopiuje zawartość odpowiedniego bloku dyskowego do bufora pamięci podręcznej. Bufor ten jest pobierany z **listy buforów wolnych**, tzn. takich, które nie są skojażone z żadnym blokiem z dysku. Jądro utrzymuje w pamięci dodatkową strukturę - tablicę list buforów wolnych. Jest ona indeksowana rozmiarem bloku, jaki może być odwzorowany w dany bufor (w opisywanej tu wersji jądra - 2.2.12, list tych jest 7; są to kolejno listy buforów o rozmiarze danych równym 0.5, 1, 2, 4, 8, 16, 32 kB). W ramach poszczególnych elementów tej tablicy bufory połączone są w dwukierunkowe listy cykliczne.

Gdy jądro sięga do danych z dysku, a nie znajduje odpowiedniego bloku w kolejce mieszającej, pobiera pierwszy blok z odpowiedniej listy buforów wolnych, czyta do niego dane, oraz wstawia do właściwej kolejki haszującej.

Jądro stara się także co pewien czas dodawać bufory do listy buforów wolnych. W kolejnych punktach opisane będą procesy, które dbają o to, by liczba buforów na tych listach nie spadała na długo poniżej pewnych wartości.

Zwalnianie buforów odbywa się zgodnie ze strategią LRU (ang. *least recently used*), tzn. w pierwszej kolejności zwalniane są bufory najdłużej nieużywane. Aby szybko wybrać taki bufor, jądro przechowuje przydzielone bufory w dodatkowej strukturze - **liście LRU**. Po odwzorowaniu bloku dyskowego do pamięci podręcznej, otrzymany bufor wstawiany jest jednocześnie do dwóch list:

- do odpowiedniej kolejki mieszającej
- do właściwej listy LRU

Przydzielane bufory dostawiane są na koniec kolejki LRU. Gdy teraz jądro zechce zasilić listy buforów wolnych pobiera pierwszy bufor z kolejki LRU usuwając go także z kolejki mieszającej. (W wersji jądra 2.2.12 są trzy listy LRU. Ich opis znajduje się w tabeli 4.1). Bufor w pamięci podręcznej może znajdować się:

- albo jednocześnie na liście LRU i w kolejce mieszającej (o ile bufor reprezentuje pewien blok dyskowy)
- albo na liście buforów wolnych (jeśli nie jest skojarzony z żadnym blokiem dyskowym)

Tablica 4.1: Rodzaje list LRU

Stała	Wartość	Opis kolejki
BUF_CLEAN	0	Lista buforów zawierających aktualne dane
BUF_LOCKED	1	Lista buforów zablokowanych
BUF_DIRTY	2	Lista buforów, których bloki danych mają zawartość niezgodną z odpowiadającymi im blokami dyskowymi

4.4.2 Budowa nagłówka bufora

Bufor składa się z dwóch części, którymi są:

- tablica - zawiera dane z dysku
- nagłówek - identyfikuje bufor; zawiera informacje o jego stanie

Nagłówek bufora (ang. *buffer head*) zawiera pole z **numerem urządzenia**, które jest logicznym numerem systemu plików (jeżeli bufor jest zajęty), lub stałą `B_FREE` (jeśli jest wolny), oraz pole z **numerem bloku**, które specyfikuje numer bloku danych na dysku. Informacje te jednoznacznie identyfikują bufor. Poza tym w nagłówku przechowywane są:

- informacja o stanie bufora (w polu `b_state`), który jest dowolną kombinacją przypadków opisanych w tabeli 4.2
- wskaźnik do tablicy danych
- wskaźnik do poprzedniego i następnego elementu w kolejce mieszającej
- wskaźnik do poprzedniego i następnego elementu na liście buforów wolnych (jeśli bufor jest wolny), lub na liście LRU (wpp)
- wskaźnik do funkcji obsługującej zakończenie operacji wejścia-wyjścia[8]

Niżej podano definicję nagłówka bufora wraz z opisami poszczególnych pól.

```
struct buffer_head {

struct buffer_head * b_next;        /* wskaźnik do następnego bufora
                                       w kolejce mieszającej */
unsigned long b_blocknr;            /* numer bloku */

unsigned long b_size;               /* rozmiar bloku */

kdev_t b_dev;                       /* urządzenie (B_FREE -> bufor wolny) */

kdev_t b_rdev;                      /* urządzenie rzeczywiste */

unsigned long b_rsector;            /* rzeczywista pozycja bufora na dysku */

struct buffer_head * b_this_page;   /* cykliczna lista buforów w obrębie
                                               jednej strony pamięci */
unsigned long b_state;              /* bitmapa statusu bufora */

struct buffer_head * b_next_free;   /* wskaźnik do następnego bufora w dwukierun-
                                               kowej kolejce (buforów wolnych lub LRU)*/
unsigned int b_count;               /* liczba użytkowników bloku */

char * b_data;                      /* wskaźnik do bloku danych */

unsigned int b_list;                /* numer listy, na której bufor się znajduje */

unsigned long b_flushtime;          /* czas, po którym dane z bufora (brudnego)
                                               powinny zostać zapisane */
struct wait_queue * b_wait;         /* kolejka dla procesów oczekujących
                                               na bufor */
struct buffer_head ** b_pprev;      /* wskaźnik do poprzedniego bufora
                                               w kolejce haszującej */
```

[8]Jest to nowe rozwiązanie. W poprzdeniej wersji jądra o tym czy operacja wejścia-wyjścia jeszcze trwa informowała ustawiona przeznaczona do tego celu flaga

```
struct buffer_head * b_prev_free;   /* wskaźnik do poprzedniego bufora
                                            w dwukierunkowej kolejce */
struct buffer_head * b_reqnext;     /* wskaźnik do kolejki żądań */

        /* pola używane do zakończenia obsługi we/wy */

void (*b_end_io)(struct buffer_head *bh, int uptodate);   /* wskaźnik do funkcji
                                                             obsługującej we/wy */
void *b_dev_id;          /* identyfikator urządzenia */
};
```

Tablica 4.2: Flagi statusu bufora

Flaga	Wartość	Znaczenie
BH_Uptodate	0	1 jeśli zawartość bufora jest zgodna z zawartością odpowiedniego bloku na dysku
BH_Dirty	1	1 jeśli bufor zawiera modyfikowane dane
BH_Lock	2	1 jeśli dostęp do bufora jest zablokowany (procesy oczekują w kolejce b_wait)
BH_Req	3	0 jeśli dane w buforze zostały unieważnione
BH_Protected	6	1 jeśli bufor jest chroniony
BH_LowPrio	7	1 jeśli bufor ma niski priorytet

4.4.3 Czytanie bloków dyskowych - funkcje `bread()` i `breada()`

Aby przeczytać blok dyskowy, proces przy użyciu algorytmu `getblk` odszukuje blok w podręcznej pamięci buforowej, lub jeśli go tam nie było przydziela nowy bufor o odpowiednim rozmiarze danych. Jeżeli blok jest w pamięci i zawiera aktualne dane, to jest on przekazywany bez fizycznego czytania danych z dysku. Jeśli, natomiast, bloku nie ma w pamięci głównej, to jądro inicjuje czytanie z dysku do bufora (wywołuje funkcję `ll_rw_block` opisaną w rozdziale poświęconym obsłudze wejścia-wyjścia). Następnie oczekuje na zakończenie obsługi czytanie danych z dysku, oraz sprawdza czy operacja ta powiodła się, tzn. czy bufor zawiera teraz aktualne dane. Jeśli tak jest, przekazuje bufor jako wynik działania algorytmu. W przeciwnym przypadku zwalnia bufor (algorytm `brelse`) i zwraca NULL

Czasem udaje się przewidzieć zapotrzebowanie na pewne bloki dyskowe (np. gdy proces czyta plik sekwencyjnie). W tej sytuacji po przeczytaniu jednego bloku dyskowego zgłaszane jest zapotrzebowanie na kolejny blok. W takich przypadkach jądro sprowadzając do pamięci podręcznej wymagany blok, inicjuje także asynchroniczne czytanie z dysku pewnej liczby (w wersji 2.2.12 maksymalnie 15) kolejnych bloków dyskowych. W tym celu wykonuje algorytm czytania z wyprzedzeniem `breada` (ang. *block read-ahead*).

Jądro sprawdza czy pierwszy blok jest w pamięci buforowej i zawiera aktualne dane. Jeśli tak jest to kończy działanie funkcji nie czytając fizycznie danych z dysku i zwraca bufor jako jej wynik.

Algorytm 6 bread

Dane wejściowe: numer urządzenia, numer bloku, rozmiar danych
Wynik: bufor z odwzorowanym blokiem

```
przydziel bufor na blok (algorytm getblk)
jeśli bufor zawiera aktualne dane to
   zwróć bufor
koniec jeśli
zainicjuj czytanie z dysku do bufora
czekaj na zakończenie operacji wejścia-wyjścia
jeśli bufor zawiera aktualne dane to
   zwróć bufor
koniec jeśli
zwolnij bufor (algorytm brelse)
zwróć NULL
```

W przeciwnym przypadku wywoływany jest podprogram obsługi dysku z poleceniem wczytania bloku.

Następnie liczona jest liczba bloków do czytania asynchronicznego[9]. Po czym jądro w pętli przydziela bufory na kolejne bloki do momentu, aż ich liczba zrówna się z liczbą bloków do wczytania asynchronicznego, lub też natrafi na bufor zawierający aktualne dane. W dalszej kolejności inicjowane jest czytanie z wyprzedzeniem kolejnych bloków dyskowych, na które przydzielono bufory. Następnie, bez oczekiwania na zakończenie opercji wejścia-wyjścia, zwalniane są bufory, do których zainicjowano asynchroniczne czytanie.

Teraz jądro ustawia się w kolejce procesów oczekując zakończenia operacji czytania pierwszego bloku (czytanie synchroniczne), po której zakończeniu sprawdza czy bufor zawiera aktualne dane (dzieje się tak zawsze wtedy, gdy operacja czytania powiedzie się). Jeżeli wszystko się udało, podobnie jak w algorytmie `bread` przekazuje bufor jako wynik, a w przeciwnym przypadku zwraca NULL.

4.4.4 Pisanie bloków dyskowych - demon *bdflush*

Do zapisywania bloków na dysk służy demon **bdflush**.

Funkcja `bdflush` jest zrealizowana w postaci wątku jądra i uruchamiana podczas jego inicjalizacji. Jej zadaniem jest zapisanie pewnej liczby bloków na dysk. Liczba ta jest określona za pomocą parametrów jądra w tabeli 4.3.

Demon `bdflush` może zostać obudzony przy użyciu funkcji `wakeup_bdflush`. Jest on uaktywniany podczas odzyskiwania wolnych buforów (w funkcji `refill_freelist`) oraz przez proces `kupdate`.

Wątek `bdflush` może być konfigurowny w czasie działania przy użyciu funkcji systemowej `sys_bdflush`, tzn. możliwa jest zmiana parametrów wątku opisanych w tablicy 4.3. Jednak parametry te muszą przyjmować wartości z przedziałów opisanych w tablicach `bdflush_min` i `bdflush_max`. Kolejność parametrów w tablicach taka sama jak w tablicy.

[9] W wersji jądra 2.2.12 liczba przeczytanych jednorazowo bloków łącznie z blokiem wczytywanym synchronicznie nie może przekroczyć stałej `NBUF` równej 16

Algorytm 7 breada

Dane wejściowe: numer urządzenia, numer bloku, rozmiar danych, pozycja, od której zaczynamy czytać, rozmiar pliku

Wynik: bufor z odwzorowanymi danymi z logicznego bloku dyskowego opisanego przez parametry wejściawe

```
1: jeśli dane wejściowe są niepoprawne A to
2:   zwróć NULL
3: koniec jeśli
4: przydziel blok na pierwszy (czytany synchronicznie) blok
5: jeśli blok zawiera aktualne dane to
6:   zwróć blok jako wynik
7: w przeciwnym przypadku
8:   zainicjuj czytanie do bloku
9: koniec jeśli
10: policz ile buforów maksymalnie można wczytać z wyprzedzeniem
11: dopóki nie przydzielono buforów na wszystkie bloki, które mają być wczytane wykonuj
12:   przydziel bufor na blok
13:   jeśli bufor zawiera aktualne dane to
14:     zwolnij bufor
15:     wyjdź z pętli
16:   koniec jeśli
17: koniec dopóki
18: jeśli są bloki do wczytania z wyprzedzeniem to
19:   zainicjuj czytanie z wyprzedzeniem do tych bloków
20: koniec jeśli
21: dopóki nie zwolniono wszystkich buforów przydzielonych do czytania asynchronicznego bloków wykonuj
22:   zwolnij kolejny bufor
23: koniec dopóki
24: zaczekaj na zakończenie czytania synchronicznego zainicjowanego w punkcie 8
25: jeśli bufor zawiera aktualne dane to
26:   przekaż bufor jako wynik funkcji
27: koniec jeśli
28: zwolnij bufor (operacja czytania nie powiodła się)
29: zwróć NULL
```

Tablica 4.3: Parametry demona bdflush

Parametr	Domyślna wartość	Znaczenie
`nftact`	40	Procentowa część buforów z zawartocią niezgodną z odpowiednim blokiem na dysku, po przekroczeniu której uaktywniany jest demon `bdflush`
`ndirty`	500	Maksymalna liczba buforów, które demon `bdflush` może zapisać przy jednokrotnym uaktywnienu
`nrefill`	64	Liczba buforów zawierających dane zgodne z odpowiednimi blokami na dysku generowana przy jednokrotnym wywołaniu `refill_freelist`
`nref_dirt`	256	Liczba buforów zawierających dane nieaktualne, po przekroczeniu której uaktywniany jest demon `bdflush` w funkcji `refill_freelist`
`interval`	5*HZ	opóźnienie zapisu w procesie `kupdate`
`age_buffer`	30*HZ	czas, o który opóźniony jest zapis bloku zawierajacego nieaktualne dane
`age_super`	5*HZ	czas, o jaki opóźniony jest zapis bloku specjalnego zawierającego nieaktualne dane
`dummy2`	1884	parametr nie jest używany
`dummy3`	2	parametr nie jest używany

```
int bdflush_min[N_PARAM] = {  0,  10,    5,   25,  0,   1*HZ,   1*HZ, 1, 1};
int bdflush_max[N_PARAM] = {100,5000, 2000, 2000,60*HZ, 600*HZ, 600*HZ, 2047, 5};
```

4.4.5 Scenariusze dostępu - `getblk()` i `brelse()`

Podręczna pamięć buforowa pośredniczy miedzy pamięcią dyskową, a pamięcią procesów użytkowych. Aby odczytać dane z dysku, lub je tam zapisać trzeba posłużyć się buforem, z którego (lub do którego) kopiuje się dane. Do przydzielania buforów używana jest funkcja `getblk` (ang. *get block*).

Gdy funkcja wyższego poziomu obsługująca część transmisji danych potrzebuje bufora, zgłasza zapotrzebowanie podając numer urządzenia, na jakim przechowywane są informacje, po które chce siegnąć proces, oraz numer i rozmiar bloku na tym urządzeniu.

Jądro identyfikuje kolejkę mieszającą, w której może znajdować się bufor zawierający te dane. Następnie przegląda ją kolejno w nadziei, że wymagany blok jest w pamięci podręcznej. Jeśli jądro znajdzie bufor w kolejce mieszającej, przekazuje go funkcji, która zgłosiła na niego zapotrzebowanie.

Może się jednak zdarzyć, że szukany blok nie jest odwzorowany do pamięci podręcznej. Wówczas jądro próbuje pobrać bufor z właściwej dla danego rozmiaru bloku danych listy buforów wolnych. Jeżeli krok ten powiedzie się, jądro inicjuje odpowiednio te pola w nagłówku bufora, za pomocą których identyfikowany jest bufor (nadaje wartości numerowi urządzenia i bloku), przenosi go do list LRU i mieszającej, oraz przekazuje jako wynik do funkcji wyższego rzędu.

Możliwa jest też sytuacja, w której jądru nie udaje się pobrać bufora z listy buforów wolnych, ponieważ jest ona chwilowo pusta. W tym przypadku podejmowana jest próba zasilenia listy buforów wolnych (poprzez wywołanie funkcji `refill_freelist`). Następnie jądro sprawdza czy w kolejce mieszającej znajduje się bufor skojarzony z odpowiednim blokiem z dysku. Jeśli tak jest, to jest on przekazywany jako wynik funkcji. W przeciwnym przypadku, jądro ponownie próbuje przydzielić bufor z listy buforów wolnych.

Mamy zatem trzy[10] możliwe scenariusze działania jądra podczas przydzielania bufora na potrzeby bloku dyskowego:

- blok jest odwzorowany do pamięci podręcznej, jądro odnajduje skojarzony z nim bufor w kolejce mieszającej
- blok nie jest odwzorowany do pamięci podręcznej, jądro przydziela bufor z listy buforów wolnych
- jądro nie odnajduje bloku w kolejce mieszajacej, a lista buforów wolnych jest pusta.

Algorytm 8 getblk

Dane wejściowe: numer urządzenia, numer bloku, rozmiar tablicy danych
Wynik: bufor

1: znajdź w kolejce mieszającej bufor, w który odwzorowany jest szukany blok
2: **jeśli** blok był w pamięci podręcznej **to**
3:    zwróć bufor
4: **koniec jeśli**
5: weź bufor z odpowiedniej dla rozmiaru danych listy buforów wolnych
6: **jeśli** nie było buforów w liście wolnych **to**
7:    przejdź do punktu 11 algorytmu
8: **koniec jeśli**
9: usuń bufor z listy wolnych, zainicjuj odpowiednio pola nagłówka bufora i wstaw do właściwych list LRU i mieszającej
10: zwróć bufor
11: spróbuj dopełnić listę buforów wolnych (algorytm refill_freelist)
12: **jeśli** nie udaje się znaleźć odpowiedniego bloku w kolejce mieszającej **to**
13:    idź do punktu 5
14: **koniec jeśli**
15: idź do punktu 1

Do zwalniania bufora służy funkcja `brelse` (ang. *buffer release*). Funkcja ta nie przenosi bufora na listę buforów wolnych, a jedynie zapisuje z jakim opóźnieniem bufor powinien być zapisany, zmniejsza licznik odwołań (zwiększony przez funkcję `getblk`), oraz o ile bufor nie znajduje się we właściwej liście LRU, odpowiednio go przenosi.

[10]We wcześniejszych wersjach jądra linuksa scenariuszy tych było 4. Wariant, gdy blok znajdował się w kolejce mieszającej rozbity był na dwa przypadki: 1) bufor jest dostępny; 2) bufor nie jest dostępny. W opisywanej tu wersji 2.2.12, jądro przekazuje bufor funkcjom wyższego rzędu nie zważając czy jest on zablokowany, czy też nie. Co więcej opisane wcześniej funkcje czytające dane do bufora także nie zwracają uwagi na ten fakt inicjując operację wejścia-wyjścia. Wywołują funkcję `ll_rw_block`, która blokuje bufor przed rozpoczęciem transmisji danych. Natomiast za ponowne uczynienie bufora dostępnym odpowiada funkcja obsługująca zakończenie operacji wejścia-wyjścia - składowa nagłówka bufora. W pliku źródłowym `buffer.c` znajdują się definicje dwóch takich funkcji. Jedna z nich obsługuje operacje synchroniczne, druga asynchroniczne

Algorytm 9 brelse

Dane wejściowe: nagłówek bufora
Wynik: brak
1: ustaw czas zapisu danych z bufora na dysk
2: przenieś bufor do właściwej listy LRU
3: zmniejsz licznik odwołań do bufora

4.4.6 Strategia odzyskiwania buforów

Do zasilania list buforów wolnych służy funkcja `refill_freelist`. Próbuje ona przydzielić pamięć na bufory spośród wolnej pamięci dostępnej w systemie. Jeżeli próba pobrania wolnej strony powiedzie się, jądro dzieli ją na bufory o odpowiednim rozmiarze danych, a następnie dodaje do list buforów wolnych.

Jeżeli jednak wolnej pamięci w systemie było zbyt mało, funkcja budzi opisanego wcześniej demona `bdflush`.

Algorytm 10 refill_freelist

Dane wejściowe: rozmiar bloku danych
Wynik: brak
1: **jeśli** nie udało się przydzielić pamięci na bufor o podanym rozmiarze **to**
2: obudź demona bdflush
3: **koniec jeśli**

4.4.7 Wady i zalety pamięci podręcznej

Niestety pamięć podręczna ma pewne wady.

- Użytkownik wywołując systemową funkcję zapisu nie wie kiedy naprawdę dane zostaną skopiowane na dysk. Ponieważ stosując zapis opóźniony, jądro odkłada fizyczne zapisanie danych na dysk mogą one zostać utracone w przypadku awarii systemu. W tej sytuacji dane na dysku będą nieaktualne.
- Użycie podręcznej pamięci buforowej wymaga dodatkowego kopiowania danych. Gdy proces użytkowy zgłasza zapotrzebowanie na dane z dysku, są one najpierw kopiowane z dysku do puli bufrów, a dopiero potem do pamięci procesu. Podobnie w drugą stronę. Proces kopiuje dane do jądra, a jądro w dalszej kolejności zapisuje je na dysk.
- Utrzymywanie puli buforów jest dodatkowym narzutem na wolną pamięć w systemie.

Jednakże podręczna pamięć buforowa ma też kilka zalet, które z pewnością zadecydowały o zastosawaniu tego właśnie rozwiązania.

- W przypadku, gdy procesy sięgają często do tych samych danych stosowanie podręcznej puli buforów może istotnie zmniejszyć liczbę transmisji, a przez to poprawić przepustowość systemu i skrócić czas reakcji. Po odwzorowaniu bloku dyskowego do pamięci podręcznej, jądro stara się jak najdłużej przechowywać go w buforach w nadziei, że gdy proces ponownie odwoła się do tego bloku, nie trzeba będzie ponownie odczytywać go z dysku.

- Stosowanie podręcznej pamięci buforowej pomaga w zapewnianiu integralności systemu plików. Jądro dba o to, by procesy nie sięgnęły w tym samym czasie do jednego bloku dyskowego.

- Buforowanie przyczynia się w znacznym stopniu do ujednolicenia różnych systemów plików. Mechanizm ten oddziela funkcje systemowe obsługujące operacje wejścia-wyjścia od warstwy zależnej od systemu plików, dostarczając niezależny od niej interfejs, z którego te właśnie funkcje korzystają. Jądro samo dba o to, by wywołać funkcją chrakterystyczną dla danego urządzenia (systemu plików). Dzięki temu użytkownik posługuje się tymi samymi funkcami obsługi wejścia-wyjścia do transmisji danych z różnych urządzeń.

- Buforowanie pozwala również ukryć przed aplikacjami blokowy charakter operacji dyskowych. Umożliwia łatwe odczytywanie i zapisywanie nawet pojedynczych bajtów bez straty wydajności.

4.5 Obsługa katalogów

W tym podrozdziale zajmiemy się:

- podstawowymi informacjami związanymi z katalogami w systemie plików EXT2
- algorytmami i strukturami danych dotyczącymi obsługi katalogów tworzonymi przez system Linux w pamięci
- algorytmem odpowiedzialnym za tłumaczenie nazwy ścieżkowej na numer i-węzła

4.5.1 Katalogi w systemie plików ext2

Katalogi nadają systemowi plików strukturę hierarchiczną. Na system plików można patrzeć jak na drzewo[11] poetykietowane nazwami plików i katalogów. Taki właśnie sposób organizacji systemu plików jest bardzo wygodny oraz czytelny dla użytkownika i właśnie z tego powodu jest używany w większości współczesnych systemów operacyjnych, także w Linuksie.

W Linuksie katalogi są plikami, ale o specjalnej strukturze. Składają się one z ciągu tzw. **pozycji katalogowych**[12] (patrz *Tablica 4.1* [13]). Każda taka pozycja opisuje dowiązanie do pewnego pliku. W katalogu mogą także istnieć pozycje z numerem i-węzła równym 0. Oznacza to, że pozycja taka została usunięta i może ona zostać wykorzystana podczas dodawania nowego pliku do katalogu.

Tablica 4.4: Pozycja katalogowa

Pole pozycji	Wielkość pozycji	Opis pola
`inode`	4 bajty	jednoznaczny numer i-węzła
`rec_len`	2 bajty	rozmiar pozycji
`name_len`	2 bajty	długość nazwy (1..255 znaków)
`name`	`name_len` bajtów	nazwa

Korzeń drzewa katalogów – główny katalog systemu plików – ma nazwę "/". Plik katalogu nigdy nie jest pusty, gdyż każdy katalog zawiera pliki o nazwach "." i "..", które odpowidają odpowiednio katalogowi bieżącemu i nadrzędnemu w drzewie katalogów. W katalogu będącym korzeniem systemu plików odpowiednie pozycje mają numer i-węzła odpowiadający temu korzeniowi.

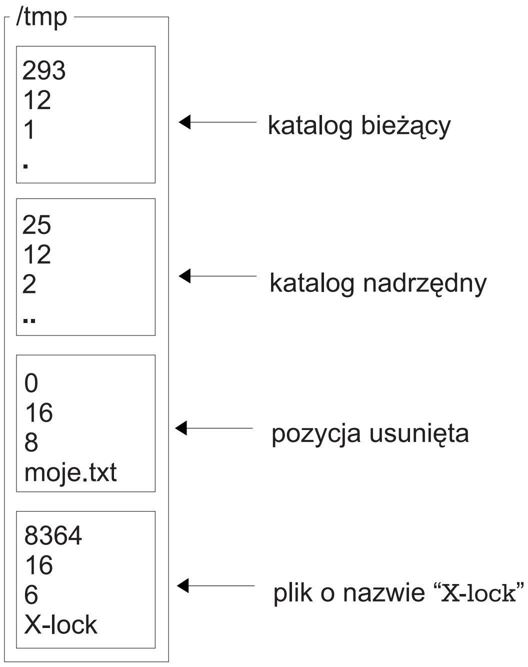
Operacje na katalogach i plikach są takie same. Zwykłe procesy mogą czytać katalogi w taki sam sposób jak zwykłe pliki, lecz tylko jądro może zapisywać katalog, zapewniając w ten sposób jego właściwą strukturę.

Z katalogami, podobnie jak ze zwykłymi plikami, związane są prawa dostępu, lecz mają one trochę inne znaczenie niż w przypadku plików. Odpowiednie prawa wraz z krótkim opisem zostały przedstawione w *Tablicy 4.2* .

[11] Właściwie nie jest to drzewo, gdyż z powodu istnienia dowiązań symbolicznych może ono zawierać cykle

[12] Pozycje katalogowe definiuje struktura `struct ext2_dir_entry` z pliku `include/linux/ext_2_fs.h`

[13] Pozycje katalogowe mogą więc mieć różną wielkość w zależności od długości nazwy, która jest zaokrąglana do wielokrotności 4 bajtów (każdy znak zajmuje 1 bajt).



Rysunek 4.4: Przykładowa zawartość pliku katalogowego `/tmp`

Tablica 4.5: Prawa dostępu dla katalogów

Prawo dostępu	Opis
czytanie	możliwość przeglądania zawartości katalogu
pisanie	tworzenie i usuwanie plików z katalogu
wykonywanie	mażliwość przeszukiwania katalogu

4.5.2 Podręczna pamięć buforowa nazw katalogów

Opis algorytmu

W celu przyspieszenia dostępu do pozycji katalogowych część z nich jest przetrzymywana w pamięci w strukturze `dentry`.

W pamięci znajduje się lista mieszająca `dentry_hashtable` , której elementami są dwukierunkowe listy. Elementami tych list są właśnie pozycje katalogowe.

```
static struct list_head dentry_hashtable[D_HASHSIZE];
```

```
D_HASHSIZE = 1Kb
```

Funkcja mieszająca zależy od:

- struktury `dentry` ojca
- nazwy katalogu
- długości katalogu

Pozycje aktualnie nieużywane (tzn. `d_count` $= 0$) przechowywane są w kolejce LRU `dentry_unused`.

Struktury danych

```
#define DNAME_INLINE_LEN 16

struct dentry {
int d_count;
     /*ilu jest użytkowników*/
unsigned int d_flags;
struct inode * d_inode;
     /*struktura inode naszego katalogu lub NULL*/
struct dentry * d_parent;
     /*nadkatalog - ojciec katalogu*/
struct dentry * d_mounts;
struct dentry * d_covers;
struct list_head d_hash;
     /*lista mieszająca, do której należy ten katalog*/
struct list_head d_lru;
     /*lista pozycji katalogowych nieużywanych występuje tylko przy d_cout = 0*/
struct list_head d_child;
     /*lista podkatalogów ojca naszego katalogu - dzieci naszego ojca*/
struct list_head d_subdirs;
     /*lista dzieci naszego ojca*/
struct list_head d_alias;
     /*aliasy d_inode*/
struct qstr d_name;
     /*dane nazwy katalogu*/
unsigned long d_time;
struct dentry_operations *d_op;
struct super_block * d_sb;
     /*korzeń drzewa, w którym znajduje się nasz katalog*/
unsigned long d_reftime;
     /*czas ostatniego odwołania*/
void * d_fsdata;
unsigned char d_iname[DNAME_INLINE_LEN]
     /*występuje przy krótkich nazwach*/
}

struct sqstr{
const unsigned char* name;
```

```
unsigned int len;
unsigned int hash;
}
     /*zmienna hash w tej strukturze zależy od nazwy i jej długości*/
```

Funkcje

W tym rozdziale opiszę funkcje:

- `d_add` dodaje strukture `dentry` naszego katalogu do listy mieszającej
- `d_delete` usuwa nasz katalog
- `dget` dodaje użytkownika
- `dput` usuwa użytkownika, jeśli jesteśmy ostatni i mamy usunąć strukturę `dentry` naszego katalogu to sprawdzamy również naszych przodków
- `d_lookup` szuka naszego katalogu mając dane: ojca i nazwę

Algorytm 11 d_add

Dane wejściowe: `struct dentry * entry, struct inode * inode`
Wynik: void

1: wyznaczamy kolejkę mieszającą, w której ma być `entry` (na podstawie `d_name.hash` oraz `d_parent`)
2: dodajemy `entry` do wyznaczonej kolejki
3: dodajemy wyznaczoną kolejkę do `d_hash`
4: inicjalizujemy `i_node` przy pomocy `inode`

Algorytm 12 d_delete

Dane wejściowe: `struct dentry * dentry`
Wynik: void

1: **jeśli** jesteśmy ostanim użytkownikiem **to**
2: w miejsce `d_inode` wstawiamy NULL (jeśli będzie potrzebna pamięć wtedy zostanie usunięty)
3: **w przeciwnym przypadku**
4: w miejsce `d_hash` wtawiamy pustą kolejke (wykorzysywane w `dput`)
5: **koniec jeśli**

Algorytm 13 dget

Dane wejściowe: `struct dentry * dentry`
Wynik: `struct dentry *`

1: zwiększamy licznik `d_count`

Algorytm 14 dput

Dane wejściowe: `struct dentry * dentry`
Wynik: void

```
dopóki jesteśmy ostatnim użytkownikiem dentry wykonuj
  jeśli w d_op jest operacja d_delete to
    wykonujemy ją (to nie znaczy że usuwamy dentry!)
  koniec jeśli
  jeśli kolejka d_lru nie jest pusta to
    usuwamy ją
  koniec jeśli
  jeśli kolejka d_hash jest pusta (było wywołane d_delete) to
    zapamiętujemy ojca
    zwalniamy strukturę dentry
    ojciec staje się dentry
  w przeciwnym przypadku
    dodajemy dentry do kolejki dentry_unused
    zapamiętujemy ją na d_lru
    wyjdź z pętli while
  koniec jeśli
koniec dopóki
jeśli nie jesteśmy ostatnimi użytkownikami to
  zmniejszamy licznik d_count
koniec jeśli
```

Algorytm 15 d_lookup

Dane wejściowe: `struct dentry * parent, struct qstr * name`
Wynik: `struct dentry *`

```
znajdujemy listę mieszającą, na której powinien znajdować się nasz katalog
wykonuj w pętli
  w tej pętli przeszukujemy wyznaczoną listę
  jeśli element ma tego samego ojca i taką samą strukturę name to
    wywołujemy funkcję dget ze znalezionym dentry
    zwracamy ten element
  koniec jeśli
koniec pętli
zwracamy NULL
```

4.5.3 Tłumaczenie nazwy ścieżkowej na numer i-węzła

Wstęp Użytkownik systemu wykonując operacje na plikach przy użyciu funkcji systemowych `open`, `chdir`, itp. posługuje się nazwami ścieżkowymi[14] (trudno sobie wyobrazić aby mogło być inaczej). Natomiast jądro systemu wewnętrznie używa numerów i-węzłów odpowiednich plików. Musi istnieć więc pewien mechanizm zajmujący się tłumaczeniem nazw ścieżkowych na numery i-węzłów. Służy do tego celu algorytm namei (patrz *Rysunek 4.1*), który jest zaimplementowany przy pomocy funkcji znajdujących się w źródłach Linuksa w plikach `fs/namei.c` oraz `include/linux/fs.h`.

Rysunek 4.5: Idea działania algorytmu `namei`

Algorytm Schemat działania algorytmu `namei` wydaje się być oczywisty (patrz *Algorytm 1*). Przegląda on nazwę ścieżkową składowa po składowej (oddziela je `"/"`), przekształca każdą składową na numer i-węzła zgodnie z nazwą składowej i przeszukiwanym katalogiem. Jako wynik przekazuje i-węzeł związany z wejściową nazwą ścieżkową.

Przykład Znając już schemat działania algorytmu `namei` rozważmy przykład[15]. Załóżmy, że pewien proces chce odczytać plik `/mnt/ntfs/czytaj.txt`. Nazwa ścieżkowa zaczyna się od "/", więc poszukiwanie rozpoczyna się od korzenia systemu plików, którego i-węzeł staje się i-węzłem bieżącym. Jądro sprawdza czy bieżący i-węzeł odpowiada katalogowi i czy proces posiada prawo do przeszukiwania katalogu. Następnie pobierana jest kolejna składowa napisu (`mnt`) i przeszukiwane są w kolejnych blokach korzenia pozycje katalogowe, aż odnajdzie się pozycję o nazwie `mnt`. Znalazłszy ją, jądro zwalnia i-węzeł korzenia (algorytm `iput`) i przydziela go dla `mnt` (algorytm `iget`) zgodnie z numerem i-węzła pobranym ze znalezionej właśnie pozycji. Ponieważ nie jest to ani punkt montowania, ani dowiązanie symboliczne, więc wykonywany jest kolejny obrót pętli algorytmu. Ten krok jest analogiczny do poprzedniego: sprawdzany jest i-węzeł katalogu `mnt` i odszukany i-węzeł dla nazwy `ntfs`, który staje się bieżącym i-węzłem. Jądro widzi, że odnaleziony i-węzeł odpowiada punktowi montowania (pole `d_mount` jest różne od bieżącegi i-węzła). Korzystając z pola `d_mount` odnajdywany jest wskaźnik do korzenia drzewa katalogów zamontowanego systemu plików i jego i-węzeł staje się i-węzłem bieżącym. Upewniwszy się, że korzeń zamontowanego systemu plików jest katalogiem i że proces posiada odpowiednie prawa, jądro przegląda `ntfs`[16] blok po bloku szukając w strukturze katalogu pozycji odpowiadającej plikowi `czytaj.txt`. Znalazłszy ją, jądro zwalnia i-węzeł korzenia zamontowanego systemu, przydziela i-węzeł pliku `czytaj.txt` i — ponieważ nazwa ścieżkowa się już skończyła — przekazuje ten i-węzeł jako wynik.

Podział algorytmu na funkcje Warto zdawać sobie sprawę, że w pliku `fs/namei` znajduje się kilka odmian algorytmu `namei` przydatnych w odmiennych sytuacjach. Można tam znaleść także wiele funkcji pomocniczych, z których korzystają funkcje implementujące algorytm.

[14]**nazwa ścieżkowa** - ciąg znaków zakończony zerem, podzielony na odrębne części za pomocą ukośnika (/)
[15]Pomijamy istnienie podręcznej pamięci buforowej nazw katalogów.
[16]Właściwie plik katalogowy korzenia zamontowanego tam systemu plików

Algorytm 16 namei

Dane wejściowe: nazwa ścieżkowa
Wynik: numer i-węzła

jeśli nazwa zaczyna się od '/' **to**
  ustaw bieżący i-węzeł na korzeń systemu plików (algorytm `iget`)
w przeciwnym przypadku
  ustaw bieżący i-węzeł na i-węzeł katalogu bieżącego (algorytm `iget`)
koniec jeśli
dopóki nie odcieliśmy jeszcze ostatniego fragmentu nazwy ścieżkowej **wykonuj**
  jeśli bieżący i-węzeł nie odpowiada katalogowi lub prawa dostępu nie są w porządku **to**
    zwróć odpowiedni błąd i zakończ działanie
  koniec jeśli
  odetnij kolejny fragment nazwy do następnego znaku "/"
  znajdź odpowiedni i-węzeł w katalogu, którego i-węzeł jest bieżący (skorzystaj z pamięci buforowej lub przeszukaj liniowo plik katalogowy)
  jeśli nie znaleziono i-węzła **to**
    zwróć odpowiedni błąd i zakończ działanie
  w przeciwnym przypadku
    ustaw bieżący i-węzeł na znaleziony
  koniec jeśli
  jeśli bieżący i-węzeł oznacza punkt zamontowania systemu plików **to**
    podstaw za bieżący i-węzeł – i-węzeł korzenia zamontowanego systemu plików (funkcja `follow_mount`)
  koniec jeśli
  jeśli bieżący i-węzeł oznacza dowiązanie symboliczne **to**
    przejdź przez dowiązanie (funkcja `do_follow_link`)
  koniec jeśli
koniec dopóki
zwróć bieżący i-węzeł

- Algorytm `namei` w standardowej postaci występuje w dwóch wersjach — funkcje `namei()` oraz `lnamei()`. Jedyną różnicą jest to, iż `namei()` dopuszcza dowiązania symboliczne, a `lnamei()` — nie. Są one używane przez proste polecenia np. `chmod`. Wywołania tych funkcji są przez makro (plik `include/linux/fs.h`) zamieniane bezpośrednio na wywołania funkcji `__namei()` z odpowiednimi parametrami.

```
#define namei(pathname)         __namei(pathname, 1)
#define lnamei(pathname)        __namei(pathname, 0)
```

- `struct dentry* __namei(const char *pathname, unsigned int lookup_flags)`

  Zwraca i-węzeł odpowiadający plikowi o podanej nazwie ścieżkowej. Używa do tego celu funkcji `lookup_dentry()`.

- ```
  struct dentry* lookup_dentry(const char *name, struct dentry *base,
                               unsigned int lookup_flags)
  ```

  Rzeczywista realizacja algorytmu `namei` w najprostszej postaci. Funkcja przekształca nazwę ścieżkową na końcowy numer i-węzła.

- `struct dentry* open_namei(const char * pathname, int flag, int mode)`

  Jest to bardziej rozbudowana wersja funkcji `namei()`, wykorzystywana przez funkcję `open()`. Różni się od poprzedniej tym, iż sprawdza większą liczbę warunków (np. dodatkowe prawa dostępu).

- ```
  struct dentry* cached_lookup(struct dentry *parent, struct qstr *name,
                               int flags)
  struct dentry* real_lookup(struct dentry * parent, struct qstr *name,
                             int flags)
  ```

  Powyższe funkcje znajdują i-węzeł odpowiadający pierwszemu elementowi podanej ścieżki, zaczynając od danego i-węzła. Różnią się tym, że pierwsza wykorzystuje podręczną pamięć buforową nazw katalogów, a druga — nie.

Wykorzystanie Sama funkcja `namei()` jest wykorzystywana tylko przez niektóre funkcje systemowe — te w najprostszy sposób korzystające z przekształcania nazw ścieżkowych na numery i-węzłów (np. `chmod`, `cd`). Funkcje bardziej wymagające jak np. `open()` korzystają z własnych wersji algorytmu (`open_namei()`). W nich własnie dokonują się dodatkowe sprawdzenia np. czy plik, do którego się odwołujemy może być czytany, itp.

Uwagi Teraz przedstawię kilka spostrzeżeń dotyczących opisanego algorytmu:

- Warto zauważyć, że realizacja algorytmu `namei` w najprostszej wersji (funkcje: `namei()` oraz `lnamei()`) jest zasadniczo zgodna z opisem podanym przez Bacha w pracy [Bach 95] (*str.* 85-88). Jedyne różnice wynikają stąd, że jądro Linuksa nie wie w jaki sposób system plików przechowuje informacje o plikach i katalogach, więc wywołuje odpowiednie funkcje.

- Porównując implementację algorytmu `namei` zawartą w jądrze Linuksa 2.2.12 z poprzednimi uwagę zwraca to, iż funkcje `dir_namei()`, `__namei()` oraz `follow_link()` zostały połączone w jedną `lookup_dentry()`, która obejmuje wszystkie szczególne przypadki poprzedniego kodu. Zmiana ta nastąpiła w stabilnych wersjach począwszy od jądra 2.2.0, wcześniej była sprawdzana tylko w wersjach testowych serii 2.1.

- Jak już wcześniej zostało wspomniane, do wyszukiwania określonej składowej nazwy ścieżkowej w pliku katalogowym wykorzystuje się podręczną pamięć buforową nazw katalogów lub przeszukiwanie liniowe. O ile korzystanie z pamieci buforowej jest szybkie, o tyle przeglądanie kolejnych pozycji w pliku może wydawać się strasznie wolne, ale i tak jest to jedno z lepszych rozwiązań. Co prawda można by zastosować szybsze metody wyszukiwania, ale byłyby one skomplikowane oraz wymagałyby bardziej złożonej struktury katalogu. Czas działania tych metod dla małych katalogów byłby także dłuższy niż czas wyszukiwania liniowego, co nie jest taką rzadką sytuacją. Poza tym wykorzystuje się dodatkowo buforowanie nazw katalogów co znacznie zmniejsza zamortyzowany koszt czasu działania ciągu wyszukiwań (wywołań funkcji `namei()`). Dodatkowym atutem zastosowanego rozwiązania, którego inne metody nie posiadają, jest jego prostota i małe wymagania pamięciowe[17].

- Istnieje polecenie `namei` obrazujące kolejność przeszukiwanych katalogów. Wypisuje kolejne pliki przez które przechodzi wraz z dodatkowymi informacjami dotyczącymi ich typu: plik regularny, urządenie, katalog, dowiązanie symboliczne, punkt montowania, itd. Może ono być wykorzystywane np. do wykrywania problemów związanych ze zbyt wieloma poziomami dowiązań symbolicznych.

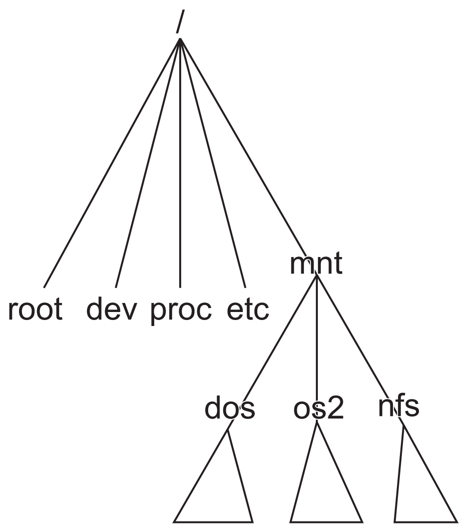
[17]Teraz może nie ma to już takiego znaczenia jak dawniej, kiedy sprzęt komputerowym był dość drogi.

4.6 Montowanie i demontowanie systemów plików

4.6.1 Wstęp

W Linuksie dostęp do systemów plików[18] z punktu widzenia użytkownika jest realizowany jako przeglądanie i modyfikowanie jednego wspólnego **hierarchicznego** drzewo katalogów. Różne systemy plików są identyfikowane jako poddrzewa „zaczepione" w pewnych węzłach tego drzewa. Czynność takiego „zaczepiania" zwana jest **montowaniem** systemów plików.

Węzeł będący korzeniem drzewa zamontowanego systemu nazywamy punktem zamontowania (*mount point*). Po inicjalizacji systemu istnieje zawsze co najmniej jeden zamontowany system plików, tzw. *root file system*. Katalog „/" jest jego punktem montowania.



Rysunek 4.6: Przykład drzewa z zamontowanymi systemami plików

Zaletami montowania systemów plików w trakcie pracy systemu oraz połączenia ich we wspólne drzewo katalogów[19] są:

- prosty dostęp do katalogów montowanych systemów przy pomocy polecenia cd oraz łatwość przechodzenia między różnymi zamontowanymi systemami plików,
- przezroczystość dla oprogramowania użytkowego; nie musi ono wiedzieć czy katalog należy do lokalnego dysku, czy może do odległego komputera np. via *Network File System*

[18] w rozumieniu „fizycznym" np. partycje dysków twardych

[19] w opozycji do statycznego i/lub automatycznego konfigurowania położenia drzew zamontowanych systemów plików, najczęściej wykonywanego przy inicjalizacji systemu (np. system MS-DOS)

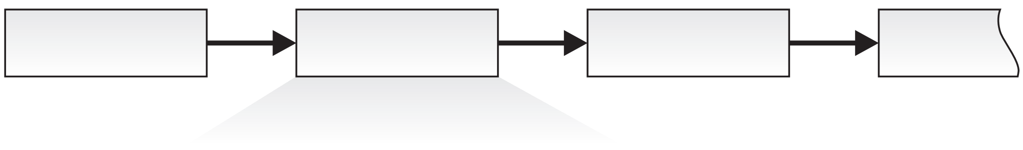
Uwagi:

- współistnienie bardzo wielu różnych typów systemów plików wewnątrz jednego drzewa jest możliwe dzięki stworzeniu abstrakcji systemu plików - **VFS** (*Virtual File System*)
- informacje o zamontowanych systemach plików jądro przechowyje w tablicy **super bloków** (tablica struktur typu `super_block`)
- innym źródłem danych o zamontowanym systemie plików jest i-węzeł odpowiadający jego punktowi zamontowania
- istnieje lista możliwych do zamontowania przez jądro systemów plików, rozszerzalna w trakcie działania systemu przez doładowywanie odpowiednich modułów (np. moduł pozwalający na obsługę dostowego systemu plików - FAT)
- poprzednia zawartość katalogu, w którym został zamontowany system plików przestaje być widoczna i dostępna aż do jego odmontowania; jest tak, gdyż po zamontowaniu w danym katalogu systemu plików zmienił się sposób traktowania i-węzła tego katalogu (patrz punkt: Montowanie)
- montować można również systemy plików **urządzeń wirtualnych**, ciekawym przykładem są: system plików **proc**, **NFS** lub **ramdysk**
- prawo do montowania i odmontowywania systemów plików ma użytkownik-root; inni użytkownicy mogą to robić jedynie w odniesieniu do systemów plików, dla których został ustawiony parametr *users* w pliku `/etc/fstab`
- istnieje możliwość **przemontowania** systemu plików z innymi parametrami (np. z trybu tylko-do-odczytu w tryb ze swobodnym dostępem)[20]

4.6.2 Tablica montowania

Informacje o zamontowanych systemach plików są przechowywane w tzw. **tablicy montowania** (ang.: *Mount Table*). W rzeczywistości jest ona zrealizowana jako lista jednokierunkowa, o początku w zmiennej `vfsmntlist` i elemencie typu `vfsmount.`

`vfsmntlist`



```
struct vfsmount {
```

[20]właśnie taka czynność wykonywana jest przy inicjalizacji Linux'a w odniesieniu do głównego systemu plików (*root*), dzięki czemu zwiększa się bezpieczeństwo danych tam zapisanych, mających istotne znaczenie dla działania systemu

`kdev_t mnt_dev` - nr urządzenia
`char *mnt_devname` - nazwa urządzenia
`char *mnt_dirname` - punkt montowania
`unsigned int mnt_flags` - flagi zamontowania systemu plików
`struct super_block *mnt_sb` - wskaźnik do superbloku zaalokowanego przez system plików
`struct vfsmount *mnt_next` - następny system plików na liście
}

Istnieje również dostęp do informacji na temat:

- ostatnio zamontowanego systemu plików: wskaźnik `vfsmnttail`,
- ostatnio używanego systemu plików: wskaźnik `mru_vfsmnt`

4.6.3 Montowanie - funkcja `mount()`

Do montowania służy polecenie mount, które odpowiada funkcji systemowej `mount()`. Jej argumenty są takie same jak pola struktury vfsmount. Algorytm został przedstawiony dalej.

Typy systemów plików obsługiwanych przez Linuxa, to m.in:

- **minix** - pierwotny system plików używany przez Linuxa, posiadający wiele ograniczeń,
- **ext** - zaprojektowany specjalnie dla Linuxa,
- **ext2** - obecna, ulepszona wersja systemu ext,
- **fat**, **vfat** - systemy plików używane przez odpowiednio: MS-DOS, MS Windows,
- **umsdos** - system plików będący w zasadzie rozszerzeniem funkcjonalności partycji DOSowych (FAT) przez dodanie kontroli praw dostępu do plików podobnej jak w ext2, a w nowszych wersjach nawet tworzenie linków symbolicznych, plików kolejek FIFO itd.
- **ntfs** - (nie mylić z **nfs**), domyślny system plików dla Windows NT,
- **nfs** - sieciowy, rozproszony system plików,
- **proc** - wirtualny system plików udostępniający informacje o systemie,
- **iso9660** - system używany do zapisu plików na dyskach CD

Dla niektórych systemów plików, np. proc, NFS nie istnieją urządzenia blokowe. Listę aktualnie możliwych do zamontowania typów systemów plików udostępnia plik `/proc/filesystems`.

Algorytm 17 Działanie funkcji `mount()`

```
jeśli użytkownik ma prawo do montowania tego systemu plików to
    Pobierz i-węzły: urządzenia i punktu montowania
    Sprawdź, czy punkt montowania jest istniejącym katalogiem (jeśli nie to błąd),
    Przeglądnij listę (file_system_type *file_systems) aktualnie możliwych do zamontowania
    (zarejestrowanych) typów sys. plików w poszukiwaniu żądanego typu
    jeśli nie udało się to
        jądro może się zwrócić do kernel demona kdemon aby załadował moduł do obsługi danego
        typu systemu plików (i zarejestrował ten typ); w przypadku porażki - błąd
    koniec jeśli
    Użyj numeru urządzenia do znalezienia tablicy funkcji obsługi danego urządzenia blokowego,
    Odczytaj (utwórz) super-blok systemu plików czerpiąc informację z urządzenia blokowego
    (jeśli istnieje),
    Dodaj nową pozycję do tablicy montowania,
    Wpisz dowiązanie i-węzła "przykrywanego" katalogu do super-bloku systemu plików,
    Wpisz do i-węzła katalogu montowania w polu inode->i_mount wskaźnik do i-węzła korzenia
    zamontowanego systemu plików.
w przeciwnym przypadku
    błąd
koniec jeśli
```

Uwagi:

Operacja montowania nie powiedzie się, gdy:

- system plików jest już używany (zamontowany),
- katalog będący punktem montowania systemu plików jest już użyty do zamontowania innego sysemu lub jest bieżącym katalogiem dla jakiegoś procesu, W przypadku systemu ext2 utworzenie przez jądro **super-bloku** w pamięci jest proste dzięki dużemu podobieństwu struktur ext2 do struktur Wirtualnego Systemu Plików (VFS); inaczej jest np. dla systemu NFS - przy tworzeniu super-bloku wymagane są bardziej skomplikowane działania, m.in. połączenie się z serwerem NFS.

Przykład polecenia montowania - dla partycji MS-DOS:

```
# mount -t vfat /dev/hda5 /mnt/dos
```

4.6.4 Odmontowywanie - funkcja `umount()`

Do odmontowywania używamy funkcji `umount()`. Po odmontowaniu systemu plików z danego katalogu będącego punktem montowania, katalog ten odzyskuje swą pierwotną zawartość. Parametrem funkcji `umount()` może być ścieżka do urządzenia blokowego, np. `/dev/hda2` lub ścieżka do punktu montowania, np. `/mnt/dos`. Algorytm funkcji `umount()` został przedstawiony poniżej.

Algorytm 18 Działanie funkcji `umount()`

Pobierz i-węzeł pliku będącego parametrem
jeśli należy on do urządzenia blokowego **to**
    pobierz z tablicy super-bloków odpowiedni super-blok systemu plików odpowiadający urządzeniu blokowemu, stamtąd zaś pobierz i-węzeł punktu montowania
w przeciwnym przypadku jeśli należy on do katalogu (punktu montowania) **to**
    pobierz super-blok wskazywany przez jedno z pól i-węzła punktu montowania
w przeciwnym przypadku
    błąd
koniec jeśli
jeśli są jakieś otwarte pliki we wskazanym systemie plików lub jakiś proces ma aktualny katalog ustawiony na katalog tego systemu **to**
    nie można odmontować (sprawdzenie polega na wykryciu obecności w pamięci aktywnych i-węzłów systemu plików)
w przeciwnym przypadku
    Wyczyść pole `i_mount` i-węzła punktu montowania
    Usuń z pamięci i-węzeł punktu montowania oraz i-węzeł korzenia drzewa katalogów odmontowanego systemu
    W razie potrzeby zapisz super-blok odmontowanego systemu
koniec jeśli

4.6.5 Przekraczanie punktów montowania podczas analizy ścieżek plików

Umożliwienie montowania systemów plików wewnątrz drzewa katalogów spowodowało utrudnienia w analizie ścieżek plików - algorytm funkcji `namei()`. Problemem jest tu przekraczanie katalogów będących punktami montowania.

Przykład:

Dla partycji `/dev/hda5` zamontowanej w katalogu `/mnt/dos` muszą być możliwe do wykonania m.in. następujące komendy:

1. `# cd /mnt/dos/programy`
2. `# cd ../..`

Czynności wykonywane dla komendy 1 polegają głównie na wywołaniu funkcji `namei("/mnt/dos")`:

1. Pobranie i-węzła drzewa katalogów
2. Znalezienie pliku katalogu opisanego przez "/" i znalezienie w nim nazwy "mnt"
3. Wyznaczenie numeru i-węzła dla nazwy `"/mnt"` i znalezienie go w pamięci funkcją `iget()`
4. Stwierdzenie, że pole `d_mounts` tego i-węzła jest `<>NULL`, zatem jest to punkt montowania jakiegoś sys. plików

5. Korzystając z pola `d_mounts->d_inode->i_sb->s_root`[21] znajdujemy wskaźnik do korzenia drzewa katalogów systemu plików, który jest podmontowany w "`/mnt`"

6. Znalezienie (ponownie funkcją `iget()`) i-węzła dla katalogu "`/programy`" w drzewie zamontowanego systemu plików, po czym ustalany jest ten i-węzeł jako bieżący katalog procesu

Teraz załóżmy, że wykonaliśmy już komendę 1 (jesteśmy w "/mnt/dos/programy"). Komenda 2 opiera się o wywołanie funkcji `namei("../..")` i działa nieco inaczej, gdyż przekraczamy punkt montowania w przeciwnym kierunku:

1. Po dokonaniu analizy ścieżki "`..`" - tak jak dla poprzednio, znajdujemy i-węzeł odpowiadający korzeniowi zamontowanego systemu plików `/dev/hda5`

2. Wyszukanie w ten sam sposób pozycji "`..`" dla znalezionego i-węzła nie mogło by się udać, gdyż jest to i-węzeł korzenia

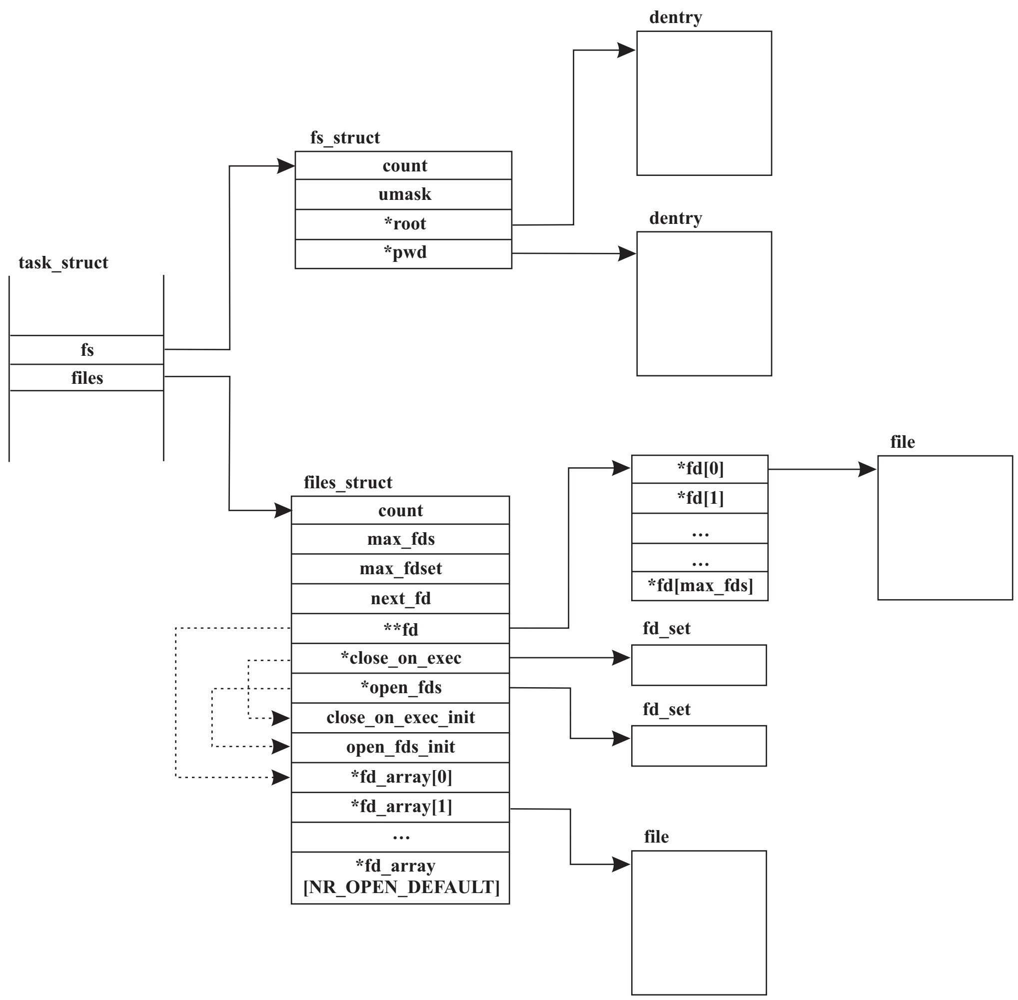
3. Należy sprawdzić, czy i-węzeł korzenia odpowiada korzeniowi zamontowanego systemu plików, w którym aktualnie się poruszamy (`/dev/hda5`). Wystarczy sprawdzić w super-bloku wskazywanym przez ten i-węzeł, czy katalog montowania nie jest korzeniem.

4. Powyższe sprawdzenie daje pomyślny wynik, więc należy pobrać z pola `d_inode->i_sb->s_root` super-bloku i-węzeł punktu zamontowania (`/mnt/dos`), po czym wykonać dla niego operację "..", przechodząc do katalogu `/mnt`.

[21] no cóż, w jądrze 2.0 było to po prostu jedno pole

4.7 Struktury danych procesów związane z systemem plików

W systemie Linux związek procesu z systemem plików opisany jest przez pola `fs` i `files` struktury `task_struct`. Są one odpowiednio typów `fs_struct` i `files_struct`, które to struktury przedstawione są na poniższym rysunku.



Rysunek 4.7: Proces, a systemy plików – struktury danych

Struktura `fs_struct` jest odpowiedzialna za informacje na temat korzenia systemu plików i aktualnego katalogu procesu. Ponadto pamiętana jest domyślna maska praw nadawanych plikom tworzonym przez proces.

W strukturze `files_struct` pamiętane są informacje o deskryptorach plików otwartych przez dany proces. Jest tu również przechowywany zbiór deskryptorów plików, które mają zostać zamknięte w przypadku wywołania przez proces funkcji `exec`.

Obie struktury zdefiniowane są w pliku `include/linux/sched.h` następująco:

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

struct files_struct {
  atomic_t count;
  int max_fds;
  int max_fdset;
  int next_fd;
  struct file ** fd;
  fd_set *close_on_exec;
  fd_set *open_fds;
  fd_set close_on_exec_init;
  fd_set open_fds_init;
  struct file * fd_array[NR_OPEN_DEFAULT];
}
```

4.7.1 Struktura `fs_struct`

Poszczególne pola tej struktury mają następujące znaczenie:

`atomic_t` **count**;

Jest to licznik odwołań do tej struktury przez różne procesy.

`int` **umask**;

Jest to maska praw automatycznie nadawana plikom tworzonym przez dany proces. Na wartość tego pola można wpływać za pośrednictwem wywołania systemowego `umask`.

`struct dentry` ***root**;

Jest to wskaźnik do struktury `struct dentry` opisującej korzeń systemu plików dostępnego dla danego procesu. Wartość tego pola używana jest do określenia bezwzględnych ścieżek dostępu do plików. Katalog ten może być zmieniony tylko przez użytkownika specjalnego poprzez wywołanie funkcji `chroot`. Pole to wykorzystywane jest na przykład przy korzystaniu z Anonimowego FTP.

`struct dentry` ***pwd**;

Jak wyżej. Struktura opisuje aktualny katalog, który wykorzystywany jest do określania względnych ścieżek dostępu do plików. Zmiana tego katalogu następuje poprzez wywołanie funkcji systemowej `chdir`.

Uwaga: Proces potomny może współdzielić zawartość tej struktury ze swoim procesem macierzystym (i ewentualnie innymi procesami) jeśli przy jego tworzeniu przez funkcję `do_fork` (wywołaną na przykład przez clone) była ustawiona flaga CLONE_FS. Wówczas dowolne wywołanie funkcji chroot, chdir czy umask przez któryś z procesów powoduje zmianę odpowiedniej wartości u wszystkich pozostałych.

4.7.2 Struktura files_struct

W jej skład wchodzą następujące pola:

atomic_t **count**;

Jest to licznik odwołań do tej struktury przez różne procesy.

int **max_fds**;

Pole mówiące o liczbie deskryptorów plików, które mogą być otwarte przez proces (rozmiar tablicy fd). Jest ono inicjowane wartością stałej INIT_OPEN_DEFAULT (równej BITS_PER_LONG == 32).

int **max_fdset**;

Rozmiar zbiorów close_on_exec i open_fds. Jest ona zwiększana, kiedy następuje powiększenie tablicy fd do rozmiaru przekraczającego wielkość tych zbiorów. Operacja ta realizowana jest przez funkcję expand_fdset (patrz opis funkcji open).

int **next_fd**;

Jest to numer pierwszego potencjalnie wolnego deskryptora pliku. Jeżeli zamykamy deskryptor (close) o numerze mniejszym od aktualnej wartości tej zmiennej, wówczas przyjmuje ona jego wartość (jest to zatem zawsze najmniejszy wolny deskryptor). Jeśli natomiast otwieramy nowy deskryptor (open), wówczas jej wartość zwiększana jest o 1 i nie musi być to już indeks wolnego deskryptora (jedynie miejsce od którego należy szukać).

struct file **** fd**;

Tablica wskaźników do struktur struct file indeksowana numerami deskryptorów plików otwartych przez dany proces. Za pomocą tych wskaźników możliwy jest dostęp do samego pliku. Początkowo fd jest adresem tablicy fd_array. Gdy proces potrzebuje więcej niż NR_OPEN_DEFAULT deskryptorów tablica ta jest powiększana – za pierwszym razem do 256 pozycji (o ile było ich mniej), następnie do rozmiaru jednej strony (co daje PAGE_SIZE / sizeof(struct file *) pozycji), potem jej rozmiar za każdym razem jest podwajany. Nigdy tablica fd nie zawiera wiecej niż NR_OPEN pozycji (domyślnie stała ta ma wartość 2^{20}). Operacja ta realizowana jest przez funkcję expand_fd_array.

fd_set ***close_on_exec**;

Wskaźnik do zbioru deskryptorów plików, które zostaną zamknięte w momencie wywołania przez proces funkcji systemowej exec. Początkowo jest to adres zbioru close_on_exec_init. Zmieni się to gdy dokonamy powiększenia zbioru, wiąże się to z alokacją pamięci na nowy zbiór. Dotychczasowa wartość zbioru close_on_exec_init pozostaje bez zmian.

fd_set ***open_fds**;

Analogicznie jak wyżej, dotyczy jedynie zbioru wszystkich otwartych przez dany proces deskryptorów. Inicjalnie jest to adres zbioru `open_fds_init`.

`fd_set` **close_on_exec_init;**

Początkowy zbiór deskryptorów plików, które zostaną zamknięte w momencie wywołania przez proces funkcji systemowej `exec`. Zbiór ten reprezentowany jest jako mapa bitowa – ustawiony n-ty bit oznacza, że deskryptor o numerze n należy do zbioru.

`fd_set` **open_fds_init;**

Jak wyżej – deskryptory plików otwartych przez dany proces.

`struct file` *** fd_array[NR_OPEN_DEFAULT];**

Inicjalna tablica wskaźników do struktur `struct file` indeksowana numerami deskryptorów.

Uwaga: Analogicznie jak poprzednia, również i ta struktura może być współdzielona przez więcej niż jeden proces. Wówczas przy tworzeniu procesu potomnego przez funkcję `do_fork` musi być ustawiona flaga `CLONE_FILES`. Powoduje to, że zamknięcie lub otwarcie deskryptora pliku przez jeden z procesów wywołuje analogiczne zmiany w tablicy desryptorów pozostałych.

4.8 Struktura pliku zwykłego

Za sprawą systemu operacyjnego użytkownik postrzega plik jako ciągły strumień bajtów o określonej długości. W rzeczywistości plik nie jest trzymany na dysku w sposób ciągły. Gdyby bowiem tak było, tzn. plik zajmowałby następujące po sobie bloki, dochodziłoby do fragmentacji zewnętrznej oraz ciężko byłoby zwiększyć jego rozmiar.

W systemie ext2 plik jest przechowywany na dysku w wielu nie występujących po sobie blokach. Opis jego rozłożenia na bloki dyskowe oraz dodatkowe informacje związane z nim (tj. właściciel, prawa dostępu, rozmiar itp.) są zapisane w **i-węźle** (`struct ext2_inode`).[22]
Dzięki **VFS** nie jest wymagane, żeby każdy system plików implementował i-węzły.

Strategia ta pozwala na dużą elastyczność systemu plików, lecz zarazem komplikuje sposób lokalizacji danych. Przechowywanie bowiem w i-węźle listy numerów bloków zajmowanych przez plik jest nieefektywne ze względu na obsługę tej listy i zmusza do narzucenia dużych ograniczeń na rozmiar pliku, bądź do wprowadzenia i-węzłów o zmiennej wielkości.

Przyjęte zostało więc inne rozwiązanie. W i-węźle znajduje się tablica o nazwie `i_block` (dla struktury i-węzła na dysku) albo `i_data` (w przypadku struktury i-węzła w pamięci). [23]
W tablicy tej przechowywane są cztery typy adresów bloków z danymi pliku (bloki na dysku są adresowane za pomocą numerów):

- *bezpośredni* - wskazuje na blok z danymi. Liczbę odresów bezpośrednich w tablicy określa stała `EXT2_NDIR_BLOCK` równa 12.
- *pojedynczy pośredni* - jest to adres bloku, który zawiera listę adresów bezpośrednich. Rozmiar bloku wynosi standardowo **1024 B**, a rozmiar wskaźnika do bloku **4 B**, zatem w bloku pośrednim pamiętanych jest 256 adresów bloków.
- *podwójny pośredni* - adres bloku zawierającego listę adresów bloków pojedynczych pośrednich.
- *potrójny pośredni* - jest to adres bloku zawierającego listę bloków podwójnych pośrednich.

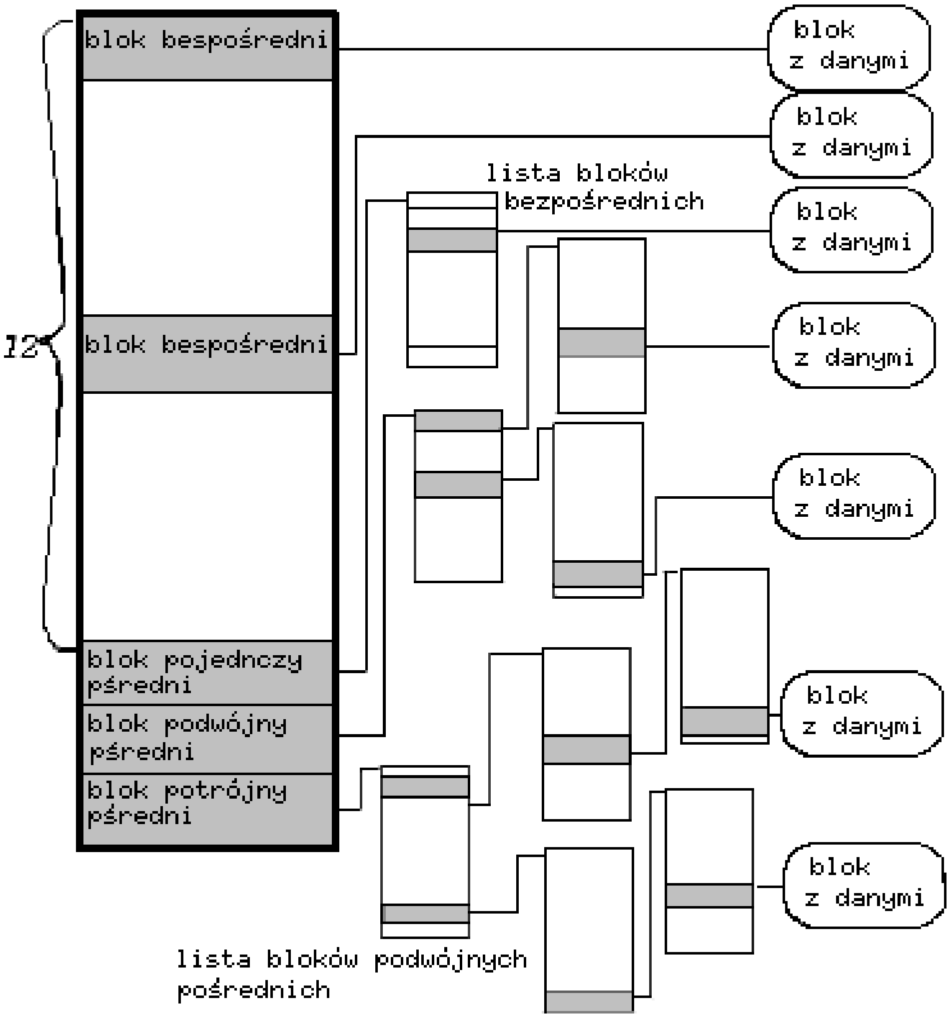
W tablicy znajduje się 12 adresów bezpośrednich i po jednym adresie pojedynczym, podwójnym i potrójnym pośrednim. Jej rozmiar opisuje stała `EXT2_N_BLOCKS` standardowo równa 15. [24]
W ten sposób można zapamiętać plik o rozmiarze **16 GB**. Pliki jednak mogą mieć co najwyżej **4 GB**, bo w i-węźle pole rozmiaru pliku zajmuje 32 bity. Z tego też powodu nie ma bloków poczwórnych pośrednich.

Warto podkreślić, że w wyniku wywołania funkcji **lseek** lub **write** niektóre pozycje w tablicy adresów bloków mogą przyjąć wartość zero. Oznacza to, że na pewnej przestrzeni pliku nie zostało nic zapisane. Na takie bloki nie zużywa się miejsca na dysku.

[22] Opis struktury i-węzła na dysku znajduje się w pliku źródłowym `include/linux/ext2_fs.h`.

[23] Aby przyspieszyć dostęp do pliku, jądro przechowuje w pamięci pewną liczbę kopi i-węzłów dyskowych (`struct ext2_inode`) wraz z dodatkowymi informacjami w strukturach również zwanych i-węzłami (`struct ext2_inode_info`). Opis struktury i-węzła w pamięci znajduje się w pliku źródłowym `include/linux/ext2_fs_i.h`.

[24] Stałe określające wielkość tablicy **i_block** i jej podział zdefiniowane są w pliku include/linux/ext2_fs.h.
`#define EXT2_NDIR_BLOCKS 12` - liczba bloków bezpośrednich;
`#define EXT2_IND_BLOCK EXT2_NDIR_BLOCKS` - pozycja w tablicy bloku pośredniego;
`#define EXT2_DIND_BLOCK (EXT2_IND_BLOCK + 1)` - ... podwójnego pośredniego;
`#define EXT2_TIND_BLOCK (EXT2_DIND_BLOCK + 1)` - ... potrójnego pośredniego;
`#define EXT2_N_BLOCKS (EXT2_TIND_BLOCK + 1)` - liczba wszystkich bloków.



Rysunek 4.8: Tablica i_blok

Do danego bajtu w pliku można się dostać przez logiczny numer bloku, w którym on się znajduje. Za przetłumaczenie numeru logicznego bloku na odpowiedni numer bloku na dysku odpowiedzialna jest funkcja **bmap**.

W ramach tego rozdziału nie została opisana struktura `struct file`, gdyż jest ona odpowiedzialna za reprezentacje pliku z perspektywy procesu użytkownika.

4.9 Dostęp do danych z pliku - algorytm `bmap()`

4.9.1 Wstęp

Algorytm `bmap` służy do przekształcania logicznego adresu bajtu w pliku na adres fizyczny na dysku.

Deklaracja (z include/linux/fs.h): `int bmap( struct inode *inode, int block)`

Argumenty:

`struct inode * inode` wskaźnik do i-węzła pliku

`int block` żądany logiczny number bloku z pliku: `block=logiczny-number-bajtu/rozmiar-bloku`

Funkcja zwraca:

- 0 gdy wystąpił błąd
- fizyczny numer bloku na dysku w przeciwnym wypadku

4.9.2 Implementacja

Funkcja `bmap` jest bardzo prosta: wywołuje funkcję `bmap` dla danego `inode` (o ile taka jest zdefiniowana). Dla przykładu omówię funkcję `ext2_bmap( struct inode * inode, int block)`. Jest ona zadeklarowana w pliku include/linux/ext2_fs.h. Implementacja znajduje się w fs/ext2/inode.c. W tym pliku znajdują się również funkcje/makra pomocnicze:

`inode_bmap(inode, nr)` makro, zwraca pozycję `nr` z tablicy bloków i-węzła. Parametr `nr` musi należeć do przedziału [0, `EXT2_N_BLOCKS-1`]

`int block_bmap(struct buffer_head * bh, int nr )` funkcja, traktując zawarty w `bh` odczytany blok jako tablicę wartości 32-bitowych, zwraca jej pozycję o numerze `nr`. Zwalnia bufor przez wywołanie `brelse(bh)`.

W funkcji zdefiniowane są następujące zmienne pomocnicze:

`int addr_per_block` określa, ile fizycznych adresów bloków mieści się w jednym bloku

`int addr_per_block_bits` jej wartością jest log2(addr_per_block) (patrz fs/ext2/super.c linia 548)

Bloki pliku należą do czterech kategorii. Zaczynając od zera, mamy:

bloków bezpośrednich: `EXT2_NDIR_BLOCKS` (=12)

bloków pośrednich pierwszego poziomu: $2^{(addr\_per\_block\_bits)}$ = `addr_per_block`

bloków pośrednich drugiego poziomu: $2^{(2*addr\_per\_block\_bits)}$ = `addr_per_block`^{2}

bloków pośrednich trzeciego poziomu: $2^{(3*addr\_per\_block\_bits)}$ = `addr_per_block`^{3}

Tak więc w sumie jeden plik może zawierać: `max= 12 + addr_per_block + addr_per_block`^{2} `+ addr_per_block`^{3} bloków.

Algorytm 19 ext2_bmap

Dane wejściowe: struct inode * inode, int block
Wynik: int - nr bloku na dysku

```
jeśli (block < 0) || (block ≥ max) to {czy numer bloku jest prawidłowy}
  return 0;
w przeciwnym przypadku jeśli block < EXT2_NDIR_BLOCKS to {blok bezpośredni}
  return inode( inode, block )
koniec jeśli
block-= addr_per_block; {pomijamy bloki bezpośrednie, wartością zmiennej block staje sie numer bloku pośredniego poziomu I, II lub III}
jeśli block < (1 « (addr_per_block_bits*2) ) to {czy jest to blok pośredni drugiego poziomu}
  i= inode_bmap( inode, EXT2_DIND_BLOCK );
  jeśli i == 0 to
    return 0;
  koniec jeśli
  bl= bread(..., i); {w i jest teraz adres bloku pośredniego poziomu I }
  i= block_bmap( bl, block » addr_per_block_bits ); {w tym bloku znajduje się adres bloku pośredniego poziomu II }
  jeśli i == 0 to
    return 0;
  koniec jeśli
  return block_bmap( bread(..., i), block & (addr_per_block-1) ); { jest to szukany adres }
koniec jeśli
block-= (1 « (addr_per_block_bits*2) ); {jest to blok pośredni poziomu III: block staje się numerem bloku pośredniego poziomu III}
i= inode_bmap( inode, EXT2_TIND_BLOCK ); {czytamy kolejne bloki pośrednie... }
jeśli i == 0 to
  return 0;
koniec jeśli
i= block_bmap( bread(..., i), block » (addr_per_block_bits*2));
jeśli i == 0 to
  return 0;
koniec jeśli
i= block_bmap( bread(..., i), (block » addr_per_block_bits) & (addr_per_block-1));
jeśli i == 0 to
  return 0;
koniec jeśli
return block_bmap( bread(..., i), block & (addr_per_block-1)); {... i mamy wynik}
```

4.9.3 Pseudo-kod algorytmu ext2_bmap

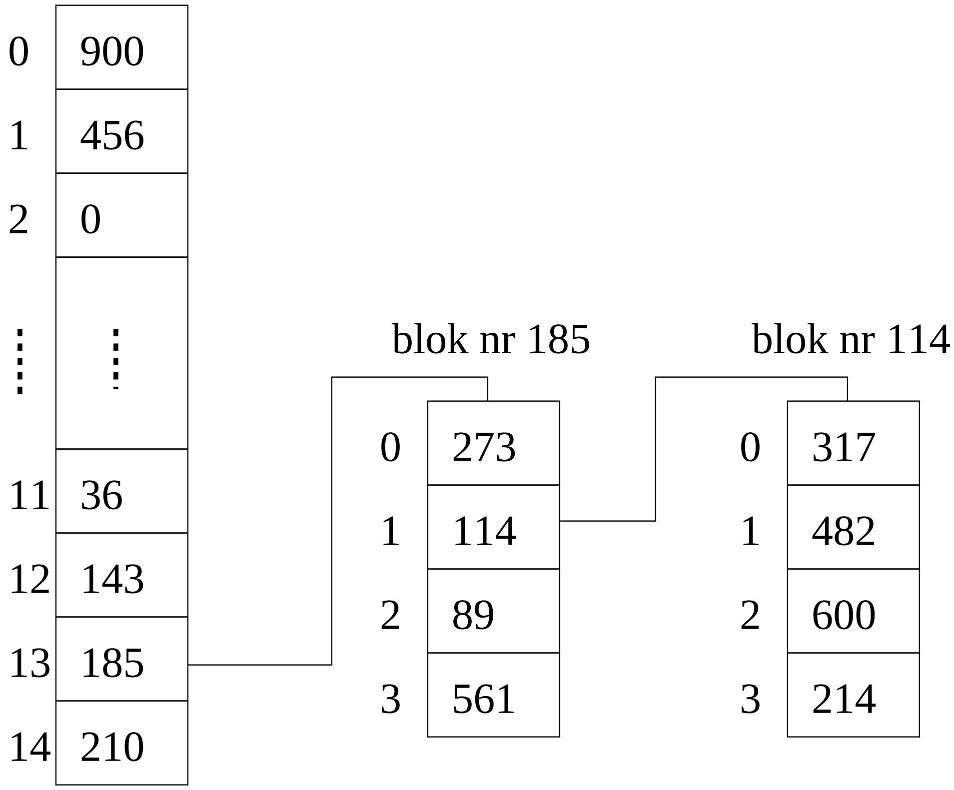
Jak widać, algorytm `ext2_bmap` ze strony 56 ten najszybciej działa dla krótkich plików. Im plik dłuższy, tym więcej trzeba dostępów do dysku, aby znaleźć żądany blok.

4.9.4 Przykład

Załóżmy, ze w bloku mieszczą się 4 adresy. Wtedy mamy:

`addr_per_block`= 4 `addr_per_block_bits`= log2(4) = 2.

Przypuśćmy, ze układ bloków pewnego pliku jest taki, jak na rysunku (4.9). Chcemy odczytać



Rysunek 4.9: Ilustracja przykładu

blok o numerze logicznym 22.

`block`= 22

Ponieważ 22 >= 12=`EXT2_NDIR_BLOCK`, to blok nie należy do bloków bezpośrednich.

`block`= block-12 = 22-12 = 10

10 >= `addr_per_block`=4, więc blok nie należy do bloków jednokrotnie pośrednich.

`block`= `block` - `addr_per_block` = 10 - 4 = 6

`block` = 6 < addr_per_block$\hat{2}$ = 16, więc blok należy do bloków podwójnie pośrednich.

Pobieramy adres bloku podwójnego pośredniego w i-węźle (indeks 13): 185. Wczytujemy ten blok.

Liczymy 6 » `addr_per_block_bits` = 6 » 2 = 2.

Odczytujemy więc drugą pozycję w bloku 185 - jest to 114.

Wczytujemy blok 114.

Numer szukanego bloku znajduje się na pozycji
6 & (`addr_per_block` - 1) = 6 & 3 = 0110b & 0011B = 0010b = 2.

Zatem numer fizyczny bloku o numerze logicznym 22 to 600.

Bibliografia

include/linux/fs.h deklaracja funkcji `bmap`

fs/inode.c definicja funkcji `bmap`

include/linux/ext2_fs.h deklaracja funkcji `ext2_bmap`

fs/ext2/inode.c definicja funkcji `ext2_bmap`, makro `inode_bmap`, funkcja `block_bmap`

4.10 Interfejs systemu plików

4.10.1 Tworzenie i otwieranie pliku

Funkcja `open()` służy do otwierania pliku. Zwraca ona deskryptor pliku, lub też w przypadku niepowodzenia kod błędu. Argumentami funkcji są kolejno:

1. filename
2. flags
3. mode

Pierwszy argument jest nazwą pliku, bądz też określa ścieżkę dostępu do pliku, który chcemy otworzyć.Drugi argument mówi nam jakie prawa dostępu będą związane z plikiem. I tak np. ustawiona flaga (O_RDONLY) pozwoli na otwarcie pliku tylko do czytania, analogicznie (O_WRONLY) daje prawo tylko do pisania, natomiast użycie flagi O_RDWR pozwoli na zarówno czytanie z pliku jak i pisanie do niego.

Argument mode jest używany z flagą OCREAT (wpp. jest ignorowany). Określa prawa dostępu do pliku. Jest modyfikowany przez atrybut umask (maska praw automatycznie nadawana plikom tworzonym przez dany proces) Ostatecznie więc, prawa dostępu będą równe (mode & ũmask). Funkcja open() wywołuje tak naprawdę funkcję sys_open(), przekazując jej swoje argumenty.

W celu wykonania funkcji open() system musi sprawdzić prawa dostępu procesu. W przypadku, gdy proces ma prawo do otwarcia pliku, przy użyciu get_empty_filp(), zajęta zostanie nowa struktura pliku. W strukturze tej ustawiane są odpowiednie wartości do pól f_flags i f_mode, a następnie wywoływana jest funkcja open_namei(). Wywołanie tej funkcji ma na celu przydzielenie struktury inode pliku który ma być otwarty. Jeżeli flaga O_CREATE jest ustawiona, to funkcja open_namei(), oprócz wywołania funkcji lookup_dentry() w celu uzyskania struktury inode pliku, w przypadku gdy wystąpił błąd mówiący, że plik nie istnieje, ale podano jej katalog z odpowiednimi prawami dostępu, tworzy plik wywołując operację create() dla struktur inode. Sprawdzana jest również flaga O_EXCL (używana z flagą OCREATE, jeśli plik istnieje zwróci błąd).

Gdyby okazało się, że plik jest katalogiem, a proces sprawdza prawa zapisu, zostanie zwrócony błąd EISDIRR (otwierano katalog do pisania). Ponadto testowane są następujące prawa: prawo dostępu do urządzeń, tzn. gdy dostęp do urządzeń został zabroniony przez pocję montowania i jeśli plik jest plikiem urządzenia, zwracany jest błąd EACCES(dostęp do pliku zabroniony, lub jeden z katalogów w filename nie zezwala na przeszukiwanie (wykonywanie)) prawo do zapisu np. kiedy plik nie jest plikiem urządzenia , ale system plików przeznaczony jest tylko do odczytu, próba zakończy się błędem EROFS(plik z systemu plików typu read-only próbowano otworzyć do pisania).

Ponadto występują jeszcze następujące kody błędów:

- ETXTBSY (otwierano plik wykonywalny do pisania)
- EFAULT (filename wskazuje na obszar poza przestrzenią adresową procesu)
- ENAMETOOLONG (nazwa filename zbyt długa)
- ENOENT (katalog składowy użyty w ścieżce nie istnieje lub jest nieprawidłowym dowiązaniem)

- ENFILE (próg na maksymalną liczbę otwartych plików w systemie został osiągnięty)
- ENOMEM (niewystarczająca ilość pamięci dostępnej jądru systemu)
- ELOOP (filename zawiera dowiązanie zykliczne tzn., że jego rozwinięcie nadal używa tego dowiązania)
- ENOSPC (brak miejsca na urządzeniu na utworzenie nowego pliku)

4.10.2 Czytanie - funkcja read()

Objaśnienia

Typ pliku oznaczają 4 bity z lewej (ósemkowo) :
001 - `FIFO` 002 - `Character Device` 004 - `Directory` inne - `Regular (zwykły), Block Device, Socket, Link`
Znaczenie bitów słowa dostępu do pliku, zapisanego w i-węźle na polu `MODE`
Znaczenie poszczególnych bitów :

- `setuid` - bit nadawania efektywnego identyfikatora użytkownika (`fsuid`) dla procesu wykonującego plik;
- `setgi` - to samo - dla `fsgid`;
- przyklejania - gdy 1 to w pamięci zostaje kopia pliku po jego wykonaniu

Funkcje systemowe ogólnie można podzielic na :

- funkcje obslugujące istniejące pliki
- służące do tworzenia nowych plików
- obsługujące i-węzeł
- pozwalające przemieszczać się w systemie plików
- rozszerzające drzewo systemu plików i zmieniające strukturę hierarchii systemu plików

Funkcja systemowa `read()` słóży do obsługi istniejącego pliku, stanowi ona przeciwieństwo funkcji `write()`. Powoduje ona wczytanie odpowiedniej ilosci bajtów danych z otwartego pliku do odpowiedniej struktury danych użytkownika. Czytanie rozpoczyna się od bieżącej pozycji wskaźnika w pliku; po zakończeniu czytania, ów wskaźnik będzie zwiększony o liczbę przeczytanych bajtów. Jeśli dane nie znajdują się już w wewnętrznym buforze jądra, to proces musi zaczekać na pobranie ich z dysku. Zazwyczaj jądro systemu probuje przyspieszyć dzialanie, zwracając uwage na sposób dostępu do pliku, sugerując sekwencyjne wczytywanie kolejnych bloków dyskowych i w konsekwencji wczytując dane z wyprzedzeniem. Jeśli system nie jest zbyt przeciążony i dane mogą przez pewien czas przebywać w buforze, a operacje czytania przebiegają sekwencyjnie, to wczytywanie z wyprzedzeniem jest całkiem skuteczne.

Opis funkcji `read()`

- służy do obsługi istniejącego pliku
- wczytuje odpowiednią ilość bajtów danych z otwartego pliku do odpowiedniej struktury danych użytkownika
- zwiększa wskaźnik o liczbę przeczytanych bajtów
- sekwencyjnie wczytuje kolejne bloki dyskowe i w konsekwencji wczytuje dane z wyprzedzeniem

Opis funkcji `read()` **dla wszystkich systemów plików**

`DEFINICJA: int read(unsigned int fd, char *buf, unsigned int count);`
WYNIK: liczba wczytanych bajtów; 0 w przypadku napotkania na koniec pliku; -1 w przypadku błędu;

BLAD: `errno` = `EDEADLOCK`(jeżeli na plik lub rekordy jest założona blokada) `EFAULT`(w przestrzeni wirtualnej nie ma zaalokowanego bufora użytkownika); `EINVAL`(jeżeli nie możemy wywolać funkcji **read()** dla konkretnego systemu plików) `EBADF`(niepoprawny deskryptor pliku)

fd jest deskryptorem czytanego pliku **buf** jest adresem struktury danych użytkownika do którego wczytujemy dane **count** jest to liczba bajtów, które chce wczytać użytkownik

- pobieramy i-węzeł korzystając z deskryptora pliku użytkownika
- sprawdzamy w tablicy plików prawo do czytania dla pliku
- sprawdzamy, czy możemy wywołać funkcje read() dla konkretnego systemu plików
- w i-węźle pliku jest wskaźnik do wszystkich blokad założonych na ten plik. Proces, który chce odczytać dane z pewnego obszaru nie może zostać wstrzymany przez żadną blokadę obowiązkową, obejmującą ten obszar, jeśli jest jej właścicielem. W przeciwnym przypadku zostaje uśpiony
- upewniamy się czy struktura danych użytkownika jest zaalokowana w jego przestrzeni ardesowej
- dla systemu plików `EXT2` wywołujemy funkcję `generic_file_read()`

Opis funkcji `generic_file_read()`

`DEFINICJA: ssize_t generic_file_read(struct file *filp, char *buf, size_t count, loff_t *ppos)` `WYNIK:` liczba wczytanych bajtów 0 w przypadku napotkania na koniec pliku -1 w przypadku błędu

`BLAD: errno = EFAULT`(gdy jest założona blokada na plik)

Parametry **buf** i **count** są takie same jak w wywołaniu `read()` `ppos` jest aresem pozycji w pliku `filp` pozycja w tablicy plików odpowiadająca danemu plikowi

Jeśli proces sekwencyjnie czyta dane z pliku, jądro zleca, by kolejne dane z pliku znajdującego się na dysku wczytywać do pamięci, zanim jeszcze proces zażąda odczytu odpowiednich danych. Zatem jądro przewiduje, do których danych z pliku bedzie chciał się odwolać proces po to, aby żądania odczytu danych przez proces jak najczęściej były zaspokajane jedynie przez odwołanie się do pamięci operacyjnej. Przyspiesza to czytanie danych przez proces w przypadku, gdy system nie jest zbyt przeciążony i dane mogą przez pewien czas pozostawać w pamięci operacyjnej. Pozycja na globalnej liscie plików odpowiadająca danemu deskryptorowi przechowuje informacje związane z czytaniem z wyprzedzeniem dla danego pliku. Z odpowiednich pól jądro może się dowiedzieć, gdzie zakończylo się ostatnie czytanie z wyprzedzeniem. Pola te są w strukturze `struct file`:

- `f_ramax` - wartość ta zmienia się dynamicznie podczas wykonywania sie funkcji i dodatkowo musi być nie mniejsza niż `MIN_READAHEAD`, oraz nie większa niż `MAX_READAHEAD`. Pole to oznacza ile maksymalnie może zostać wczytanych bajtów z wyprzedzeniem
- `f_rawin` - tu trzymana jest ilość danych ostatnio wczytanych z wyprzedzeniem

Funkcja generic_file_read()

- jeżeli pozycja, od której proces chce rozpocząć czytanie lezy wewnątrz obszaru, który poprzednio był wczytany z wyprzedzeniem, to jądro stwierdza, że plik jest czytany sekwencyjnie i probuje czytania z wyprzedzeniem. W przeciwnym przypadku, w odpowiedniej pozycji na liście plików zaznacza, że czytanie z wyprzedzeniem nie odbędzie się.
- jeśli rozmiar danych do wczytania z wyprzedzeniem jest mniejszy niż pół ramki to jądro stwierdza, że jest za malo i nie dochodzi do czytania z wyprzedzeniem.
- az do zakonczenia czytania, w pętli wykonuja się poniższe czynności
- jądro szuka w pamięci operacyjnej ramki, zawierającej odpowiednie dane z danego pliku. Jeśli takiej nie ma (nawet jeśli plik był czytany z wyprzedzeniem, odpowiednia ramka mogła ulec modyfikacji, lub zostać wymieciona z pamięci), jądro rezerwuje wolną ramke, nastepnie zleca wypelnienie jej danymi z odpowiadającego jej bloku na dysku.
- jeśli dana ramka jest wypelniona aktualnymi danymi, badz jej zawartosc jest wlasnie aktualizowana, jądro inicjuje czytanie z wyprzedzeniem za pomocą funkcji `generic_file_readahead()`. Czeka na zakonczenie wszelkich operacji wejścia/wyjścia i zostanie ona odblokowana. Gdyby odblokowana ramka zawierała nieaktualne dane - jądro synchronicznie wypełnia ją danymi z odpowiadającego jej bloku na dysku. Następnie odbywa się kopiowanie z odpowiedniego fragmentu ramki do przestrzeni adresowej procesu.
- wywoływana jest funkcja aktualizująca i-węzeł danego pliku.

Uwagi

1. Funkcja systemowa `read()` nie może opóźniać wczytywania danych, tak jak to robi funkcja `write()` w przypadku zapisu.
2. Jeśli dane nie znajdują się w wewnętrznym buforze jądra, to proces musi zaczekać na pobranie ich z dysku.
3. W przypadku plików specjalnych oraz łączy komunikacyjnych powrót z funkcji `read()` jest natychmiastowy, jeśli plik byl otworzony z flagą `O_NDELAY` i nie ma danych do przesłania.

4. Jeśli system nie jest zbyt przeciążony i dane mogą przez pewien czas pozostawać w buforze, to w przypadku czytania sekwencyjnego, zastosowane wczytywanie z wyprzedzeniem jest skuteczne i w znaczny sposób poprawia wydajność systemu.

4.10.3 Pisanie - funkcja `write()`

Pomocne struktory danych

- `struct file:`
  - `mode_t f_mode` - tryb otwarcia pliku (pisanie/czytanie) - to nie jest ten sam tryb, co w i-wezle (patrz objasnienia w funkcji `read()`)
  - `loff_t f_pos` - pozycja w pliku
- `struct file *f_next, *f_prev` - wskaznik do poprzedniego i nastepnego pliku w tablicy plikow, ktora wcale nie jest tablica, ale lista dwukierunkowa. Jadro trzyma wskaznik do pierwszego pliku. Maksymalny rozmiar tej listy to NR_FILE = maksymalna liczba plikow otwartych w systemie
- `unsigned short f_flags` - czy plik ma szczegolne wlasciwosci (tylko do nadpisywania, niezmienialny, do pisania synchronicznego lub inne) - dotyczy konkretnego otwarcia pliku, a nie wspolnego dla wszystkich otwarc i-wezla
- `struct inode *f_inode` - i-wezel odpowiadajacy plikowi

Wprowadzenie

Funkcja `write()` sluzy do pisania do otwartego pliku (w tym rowniez gniazda, gdyz gniazdo jest typem pliku).Podczas zlecania zapisu do pliku, jadro jedynie informuje uzytkownika, ze zadanie zostalo wykonane, lecz tak naprawde jadro zaznacza w i-wezle danego pliku, ze nastapila zmiana. Zapis na dysk nastepuje dopiero podczas zwalniania i-wezla. Plusem tego rozwiazania jest mala ilosc transmisji na dysk, minusem zas ryzyko utracenia wprowadzonych zmian w przypadku awarii sprzetu.

Opis funkcji `write()` dla wszystkich systemow plikow

`DEFINICJA: int write(unsigned int fd, const char *buf, size_t count);`

WYNIK: liczba zapisanych bajtow; -1 w przypadku bledu;

BLAD: `errno` = `EINVAL` (jezeli nie mozemy wywolac funkcji **write()** dla konkreatnego systemu plikow) `EBADF` (niepoprawny deskryptor pliku)

fd jest deskryptorem czytanego pliku **buf** jest adresem struktory danych uzytkownika z ktorej zapisujemy dane **count** jest to liczba bajtow, ktore chce zapisac uzytkownik

- zanajduje pozycje na globalnej liscie plikow, odpowiadajaca deskryptorowi **fd**; pobiera i-wezel pliku.
- sprawdza prawo do pisania do pliku.
- sprawdza czy na plik nie jest zalozona blokada.

- sprawdza, czy mozy czytac z adresu ***buf**.
- opuszcza semafor na i-wezel.
- dla systemu `ext2` wykonujemy funkcje `ext2_file_write()`.
- podnosi semafor z i-wezla.

Opis funkcji `ext2_file_write()`

`DEFINICJA: ssize_t ext2_file_write(struct file *filp, const char *buf, size_t count, loff_t *ppos)` `WYNIK:` liczba zapisanych bajtow -1 w przypadku bledu

`BLAD: errno = ENOMEM`(blad przy tworzeniu ramki) `EINVAL`(gdy i-wezel nie istnieje, lub gdy chcemy zapisac ujemna liczbe bajtow, lub gdy plik nie jest plikiem regilarnym) `ENOSPC`(gdy plik nalezy do systemu plikow otwartego tylko do odczytu) `EFBIG`(gdy zapisanie wszystkich bajtow przekroczy dopuszczala dlugosc pliku)

- sprawdzane jest, czy plik nie nalezy do systemy plikow otwartych tylko do odczytu.
- usuwany jest bit `suid` (patrz **„Objasnienia" dla** `read()`).
- w zaleznosci od trybu otwarcia pliku, jadro ustala miejsce, od ktorego zacznie pisac.
- dla kazdego bloku dyskowego wykonuja sie nastepujace czynnosci.
  - jadro wyznacza logiczny numer bloku dyskowego, zawierajacego miejsce, od ktorego nalezy rozpoczac zapis.
  - dane nie sa zapisywane bezposrednio na dysk; jadro umieszcza je w wolnych buforach. Jesli zapis danych z bufora na dysk ulegnie opoznieniu, bufor bedzie zawieral prawidlowe dane, ktore powinny sie znajdowac w odpowiadajacym mu bloku. Jadro szuka wiec w podrecznej pamieci buforowej bufora, ktory odpowiada wyznaczonemu buforowi. Jesli pamiec buforowa nie zawiera odpowiedniego bufora, przyporzadkowywany jest nowy wolny bufor. Zaden proces nie moze uzyc znalezionego bufora, dopoki biezacy proces go uzywa.
  - gdyby jadro usilowalo zapisac na dysk jedynie fragment bloku zniszczyloby zawartosc tej czesci bloku, ktora powinna pozostac nie zmieniona. Zatem, jadro odwoluje sie do naglowka odpowiedniego bufora, by sprawdzic, czy zawiera on dane, ktore powinny znajdowac sie na dysku. W przypadku, gdy dane w buforze sa niewlasciwe, jadro synchronicznie wczytuje do niego zawartosc odpowiedniego bloku dyskowego.
  - wyznaczony fragment danych (do zapisu) jadro kopiuje w odpowiednie miejsce danego bufora.
  - proces otwierajacy plik mogl nie zazadac wstrzymania do czasu zapisu danych, jadro zaznacza wtedy, ze proces nie korzysta juz z danego bufora. Dane sa przechowywane w podrecznej pamieci buforowej, na wypadek gdyby wkrotce mialy zostac ponownie wczytane lub modyfikowane. Tak oszczedza sie liczbe operacji dyskowych.

  - jesli proces musi byc wstrzymany do czsu zakonczenia zapisu, jadro wstawia dany bufor do podrecznej tablicy buforow; jesli ta zostanie wypelniona, jadro synchronicznie zapisuje zawartosc kazdego zawartego w niej bufora do odpowiedniego bloku dyskowego. Gdy po wyjsciu z wyzej opisanej petli lokalna tablica buforow nie jest pusta, jadro synchronicznie je zapisze na dysk.
  - jesli jadro nie rozwazylo jeszcze wszystkich danych przekazanych przez proces, zapisuje na dysk kolejny ich fragment.

- uaktualniane sa pola w i-wezle mowiace o rozmiarze pliku, jego biezacej pozycji i czasie ostatniej modyfikacji.
- i-wezel zaznaczany jest jako brudny(czyli zmieniony).

Uwagi

1. Pisanie z opoznieniem po 16 blokow w funkcji `ext2_file_write()`
2. Liczba otwartych plikow nie moze byc wieksza niz NR_FILE oraz liczba plikow otwartych przez jeden proces nie moze byc wieksza niz NR_OPEN

4.10.4 Obsługa łączy nienazwanych

Wprowadzenie

łącza nienazwane są wygodnym mechanizmem komunikacji między procesami spokrewnionymi. Wygodnym dlatego, że system - zamiast programisty - martwi się o zapewnienie synchronizacji.

Jak system przechowuje łącze?

łącza nienazwane przechowywane są w i-węźle z ustawionym polem `i_pipe` na 1. W polu `u` ustawione są parametry łącza oraz bufor na dane. Format parametrów jest nastepujący:

```
struct pipe_inode_info {
struct wait_queue * wait; /* kolejka procesów zablokowanych na łączu */
char * base; /* bufor danych */
unsigned int start;  /* indeks początku danych w buforze */
unsigned int lock;     /* blokada */
unsigned int rd_openers;  /* nieużywane dla łączy nienazwanych */
unsigned int wr_openers;/* nieużywane dla łączy nienazwanych */
unsigned int readers;     /* liczba procesów które mogą czytać */
unsigned int writers;     /* liczba procesów które mogą pisać */
};
```

Bufor danych jest kolejką zaimplementowaną w tablicy. W starszych wersjach przechowywano jeszcze ilość danych w buforze. Obecnie jest to pamiętane w i-węźle. Bufor ma pojemność 4kilobajtów.

Jak używamy łączy nienazwanych?

Jak sama nazwa wskazuje łącza nie maja nazw. Odwołuje się do nich przez deskryptory. Ponieważ jedynym sposobem na przekazanie deskryptora do innego procesu jest wykonanie funkcji `fork()` i stworzenie potomka, który odziedziczy tablicę deskryptorów ojca, łącza nienazwane mogą byc używane jedynie przez procesy spokrewnione.

Aby komunikować się poprzez łącza nienazwane należy:

1. Utworzyć łącze
2. Stworzyć potomka
3. Zamknąć zbędne deskryptory (nie obligatoryjne, ale rozsądne)
4. Nadawca zamyka deskryptor do czytania
5. Odbiorca zamyka deskryptor do pisania

Jak widać z tego schematu, łącze jest jednokierunkowe (tzn. należy je używać "tylko w jedna strone"). Jeżeli potrzebujemy komunikacji dwustronnej potrzebujemy dwóch łączy.

Tworzenie łącza - funkcja `int pipe(int fd[2])`

Aby utworzyć łącze wywołujemy `pipe()`, jako parametr przekazujemy dwuelementową tablicę typu int. Jeżeli funkcja się powiedzie, zwrócone zostanie zero, a tablica wypełniona numerami deskryptorów.

1. fd[0] deskryptor do czytania z łącza
2. fd[1] deskryptor do pisania do łącza

Jeżeli funkcja się nie powiedzie zwracane jest -1 a na zmiennej `errno` **przekazywane są następujące kody błędu:**

1. EMFILE brak wolnych deskryptorów
2. EFAULT niewłaściwy argument fd

Łącza zamyka się zwyczajnie, robiąc close() na deskryptorze. Łącza zamyka się "na zawsze", do raz zamkniętego deskryptora nie ma już powrotu.

Czytanie z łącza - funkcja `int read(int fd, char* buf, size_t count)`

Czytamy z łącza zwykłą funkcją `read()` ze zwykłym znaczeniem parametrów. Informacja przeczytana opuszcza łącze. W zależności od wypełnienia łącza (ściślej bufora łącza) i parametru count zachowuje się ona w sposob nastepujący:

- **proces chce czytać mniej niż jest w buforze:** proces przeczyta tyle bajtów ile zażądał

- **proces chce czytać wiecej niż jest w buforze i bufor jest niepusty:** proces przeczyta tyle bajtów ile jest w buforze
- **bufor jest pusty i `read` wywołano z flagą O_NONBLOCK:** read() zwroci -1, `errno` ustawione na EAGAIN
- **bufor jest pusty i `read` wywołano bez flagi O_NONBLOCK:** proces jest usypiany do czasu gdy w buforze coś się znajdzie **Wyjątek!** Jeżeli nie ma już procesu, ktory miałby otwarty deskryptor pisania do łącza oraz bufor jest pusty, to bez względu na flagę O_NONBLOCK zwracane jest zero.

Pisanie do łącza - funkcja `int write(int fd, char* buf, size_t count)`

Piszemy do łącza zwykłą funkcją `write()` ze zwykłym znaczeniem parametrów. W zależności od wypełnienia bufora zachowuje się ona w sposob następujacy:

- **proces chce zapisać mniej niż jest wolnego miejsca w buforze:** proces zapisze tyle bajtów ile chce
- **proces chce zapisać wiecej niż jest wolnego miejsca w buforze oraz read wywołano z flagą O_NONBLOCK:** proces zapisze tyle bajtow ile może
- **proces chce zapisać wiecej niż jest wolnego miejsca w buforze oraz read wywołano bez flagi O_NONBLOCK:** proces zapisze tyle ile może i zaśnie do czasu, gdy w łączu zwolni się miejsce. **UWAGA!** Jeżeli kilka procesow bedzie w takiej sytuacji (zapchane łącze) możliwa jest sytuacja że komunikaty się wymieszają

Jeżeli nie ma procesu który mogłby czytać z łącza, próba zapisu nie powiedzie się. W takiej sytuacji do procesu zostanie wyslany sygnał SIGPIPE, zwrócona wartość -1 oraz errno ustawione na EPIPE.

4.10.5 Zamykanie pliku - funkcja `close()`

Funcja służy do zwalniania deskryptora pliku, dzięki czemu daje możliwość przydzielenia go innemu plikowi. Najpierw sprawdzane jest czy otwarto plik o deskryptorze podanym jako argument funkcji. Jeśli tak, to zmiennej lokalnej przypisywany jest wskaźnik do struktury pliku, wpp. NULL. Następnie wywoływana jest funkcja filp_close(), która sprawdza czy nikt nie odwołuje się już do pliku. Jeżeli deskryptor jest ostatnim, który odnosi się do danej pozycji w tablicy plików, wówczas pozycja ta jest zwalniana. Zdejmowane są wszelkie blokady założone przez proces na plik. Funkcja zwraca 0 w przypadku sukcesu, oraz kod błędu gdy się nie powiedzie.

4.10.6 Blokowanie plików

Czym jest blokowanie plików?

Blokowanie pliku polega na ograniczeniu innym procesom dostępu do tego pliku. Na przykład, jeśli proces chce coś zapisać do pliku, to chciałby zabronić innym procesom dostępu do niego na czas zapisywania.

Terminologia

Wiele angielskich terminów można przetłumaczyć na słowo **blokada**. Aby uniknać nieporozumień, przyjmijmy:

1. Blokadę na pliku, czyli ograniczenie dostępu do pliku (ang. *lock*) będziemy nazywać **blokadą**
2. Blokadę procesu, czyli jego uśpienie (ang. *wait*) będziemy nazywać **wstrzymywaniem**
3. Blokadę systemu, czyli wzajemną blokadę procesów (ang. *deadlock*) będziemy nazywać **zakleszczeniem** lub **zastojem**.

Ogólny schemat stosowania blokad:

1. Załóż blokadę na plik
2. Pracuj z plikiem
3. Zdejmij blokadę

Kiedy stosuje się blokady obowiązkowe, punkt pierwszy i trzeci wykonuje się niejawnie. Wiecej informacji o tym ponizej.

Blokady dzielone (ang. *shared*) i wyłączne (ang. *exlusive*).

Terminy te najlepiej objaśnią przykłady:

- Kiedy chcemy pisać do pliku, używamy **blokad wyłącznych**, aby zabronić innym procesom dostępu do pliku.
- Kiedy chcemy czytać z pliku, możemy pozwolić innym procesom na czytanie i używamy **blokad dzielonych.**
- Jesli założona jest blokada wyłączna, wówczas nie powiedzie się próba założenia żadnej innej blokady.
- Jesli założona jest blokada dzielona, wówczas można załozyc drugą blokadę dzieloną, natomiast nie można założyć blokady wyłącznej.

Blokowanie wstrzymujace i niewstrzymujace

Blokowanie **wstrzymujace** oznacza, ze jeśli nie powiedzie sie próba założenia blokady, proces zostanie zawieszony aż do momentu, gdy będzie mógł założyć blokadę.

Blokowanie **niewstrzymujace** oznacza, ze w jeśli nie powiedzie sie próba założenia blokady, proces zacznie wykonywać kolejną instrukcję.

Przy blokowaniu wstrzymujacym istnieje niebezpieczeństwo zakleszczenia. Nazywamy tak sytuacje, gdy dwa procesy czekają na siebie nawzajem. Nie grozi to w przypadku blokowania niewstrzymujacego. Zazwyczaj jednak używa sie sposobu grożącego zakleszczeniem.

W Linuksie istnieja dwa standardy blokownia:

1. Standard BSD

2. Standard POSIX

Standard BSD

W tym standardzie używa sie funkcji `int flock(int fd, int operation)`, gdzie:

1. fd jest deskryptorem pliku, na ktorym operujemy,

2. operation opisuje co chcemy zrobić

Funkcja zwraca 0 w przypadku sukcesu i -1 w przypadku porażki.

Dostępne sa następujące operacje:

1. LOCK_SH założenie blokady dzielonej

2. LOCK_EX założenie blokady włąacznej

3. LOCK_UN zdjęcie blokady

Domyślnie zakładanie blokady jest operacją wstrzymującą. Blokowanie niewstrzymujące uzyskujemy dołaczając flagę LOCK_NB, np. flock(fd, LOCK_SH|LOCK_NB).

Uwaga: Blokady BSD nie działają na sieciowym systemie plików (NFS). W odróżnieniu od standardu POSIX, nie ma żadnego zabezpieczenia przed zakleszczaniem.

Standard POSIX

W tym standardzie blokowanie jest bardziej skomplikowane. Przede wszystkim blokujemy nie całe pliki, lecz wybrane kawałki. Dlatego nie wystarczy podać tylko operację, ktora chcemy wykonać. Aby założyć blokadę, trzeba wypełnić następującą strukturę:

```
struct flock {
short l_type;     /* typ blokady */
short l_whence; /* tryb obliczania przesunięcia w rekordzie
(jak w lseek)*/
off_t l_start;    /* początek obszaru blokowanego */
off_t l_len;      /* długość obszaru blokowanego */
pid_t l_pid;      /* właściciel blokady  */
};
```

Istnieja nastepujace typy blokady

1. F_RDLCK Blokada odczytu (dzielona),
2. F_WRLCK Blokada zapisu (wyłączna),
3. F_UNLCK Zdjęcie blokady

Do blokowania uzywa sie funkcji `nt fcntl(int fd, int cmd, struct flock * lock)`, gdzie:

1. fd - deskryptor pliku który chcemy blokować,
2. cmd - komenda
3. lock - wskaźnik do zmiennej typu struct flock, dokładnie opisujący blokadę.

Funkcja zwraca 0 w przypadku powodzenia i -1 w przeciwnym przypadku.

Korzystamy z następujacych komend:

1. F_SETLK niewstrzymująca próba założenia blokady
2. F_SETLKW wstrzymująca próba założenia blokady.
3. F_GETLK sprawdzenie czy założenie blokady doprowadzi do zakleszczenia.

Funkcja `fcntl` ma jeszcze inne komendy, ale dla potrzeb tego rozdziału zostanie to pominięte.

Uwaga: Standard POSIX zapobiega zakleszczeniom

Poniżej opisany jest algorytm, ktory bada niebezpieczeństwo zastoju. Jeśli proces chce założyć blokadę wstrzymujacą i powinien zostać uśpiony, system sprawdza, czy jest to bezpieczne.

Dane wejściowe:

- my_task - wskaźnik do procesu, ktory chcemy uspić
- blocked_task - wskaźnik do procesu, ktory zabrania założenia blokady (blocked_task założył blokadę wcześniej)

Na wyjściu otrzymujemy:

- 1 w przypadku zakleszczenia
- 0 w p.p.

```
etykieta next_task:

  if (my_task==blocked_task)
     return 1;
  for (wszystkie blokady)
      {
```

```
        if (blocked_task uśpiony na kolejce tej blokady)
           {
           if (wlasciciel blokady==my_task)
              return 0;
           blocked_task=właściciel blokady;
           goto next_task;
           }
        }
    return 0;
```

Co robi system po wykryciu niebezpieczenstwa zakleszczenia?

- Jeżeli użyto komendy F_SETLKW, to funkcja blokująca zwraca -1, a errno jest ustawione E_DEADLOCK.
- Jezeli użyto komendy F_GETLK, czyli sprawdzenia ewentualności zastoju, to struktura flock zostaje wypełniona informacjami o blokadzie uniemożliwiającej zablokowanie pliku.
- Dla komendy F_SETLK nie ma groźby zakleszczenia, bo jest to blokowanie niewstrzymujace.

Pozostaje zapytać, co zrobi funkcja fcntl wywołana z komendą F_GETLK jeśli nie będzie groźby zastoju. Otoż wtedy pole l_type zmiennej lock zostanie ustawione na F_UNLCK.

- **Blokady POSIX działają na NFS.**
- **Blokady POSIX i BSD są względem siebie całkowicie niezależne! (ang.** ***oblivious*****).**

Blokady zalecane i obowiązkowe

Opisane wyżej blokady są **zalecane** (ang. *advisory*), co oznacza że ich stosowanie jest dobrowolne. Jeżeli jakiś proces nie chce stosować się do ceremoniału zakładania i zwalniania blokad, nikt go nie zmusi. Ponadto blokady zalecane funkcjonują jedynie pomiędzy procesami je stosującymi. Oznacza to, że proces-prostak może dostać się do zablokowanego pliku, a także uda się założyć blokadę na plik aktualnie używany przez prostaka.

Kiedy używamy blokad **obowiązkowych** (ang. *mandatory*), wszystkie procesy muszą je stosować. Toteż jeżeli jeden proces założy blokadę obowiązkową, wszystkie inne muszą się podporządkować. Ponadto, każde wywołanie funkcji systemowych takich jak open, read,write itp. powoduje sprawdzenie istniejących blokad oraz założenie własnych. Na przykład próba odczytania pierwszych dziesięciu bajtów pliku f spowoduje blokadę odczytu na pierwszych 10 bajtach f.

Blokady obowiązkowe są cechą systemu plikow. Sa włączane i wyłączane podczas montowania. Standardowo blokady obowiązkowe sa wyłączone.

Blokady obowiązkowe sa zawsze typu POSIX

4.11 System plików proc

4.11.1 Wstęp

Wśród wielu różnych systemów plików jak na przykład opisany wcześniej ext2 znajduje się **system plików proc**. Ponieważ korzysta on ze standardowego **interfejsu vfs**, dla użytkownika systemu wygląda on normalnie (można poruszać się po katalogach, czytać pliki, ...). Jednak pod różnymi względami różni się on od innych systemów plików. Nie istnieje żadne urządzenie (nośnik danych), na którym byłyby przechowywane pliki proc-a. Jest to system plików, którego struktury i sposób działania w pełni zostały zaimplementowane w jądrze linuxa.

4.11.2 Zalety

Standardowo w każdej dystrybucji linuxa system plików proc można znaleźć zamontowany w katalogu `/proc`. Jest od bardzo przydatny zarówno dla programów współdziałających z jądrem, jak i dla zwykłych użytkowników linuxa. Proc udostępnia w swoich plikach wiele parametrów systemu operacyjnego, dane o sprzęcie i aktualnie działających procesach. Szczególnie ta ostatnia własność jest bardzo przydatna. Można w ten sposób sprawdzić zawartość pamięci procesu, deskryptory plików, z których proces korzysta, aktualny katalog procesu itd. Poprzez pliki udostępniane są niektóre zmienne jądra, które można modyfikować w trakcie jego działania.

4.11.3 Implementacja

Źródła

Implementacja system plików proc znajduje się w podkatalogu `fs/proc` źródeł linuxa, a odpowiedni plik nagłówkowy opisujący używane struktury to `include/linux/proc_fs.h`

Rejestracja proc-a

Aby system plików był rozpoznawany przez jądro trzeba go zarejestrować. Procedura `sys_setup` inicjująca systemy plików wywoluje procedurę `init_proc_fs` inicjującą proc-a, która między innymi rejestruje go procedurą rejestrującą `register_filesystem`. Zarejestrowany system plików od tej chwili może być montowany.

Montowanie proc-a

W trakcie ładowania systemu operacyjnego montowane są systemy plików wskazane w pliku `/etc/fstab`. Standardowo dokonuje się to podczas wykonywania skryptu bootującego `/sbin/init.d/boot`. Zrobiono jednak wyjątek dla proc. Nie jest ważne, czy znajduję się on w `/etc/fstab`, czy nie; nie jest brany pod uwagę. Montuje się go oddzielnie w tym samym skrypcie przed zamontowaniem pozostałych systemów plików z `/etc/fstab`. Jako proste ćwiczenie można spróbować zamontować sobie "drugiego"proc-a:

1. `mkdir /proc2`
2. `mount -n -t proc proc /proc2`

Budowa wewnętrzna proc-a

Podstawową strukturą systemu plików proc jest `proc_dir_entry`

```
struct proc_dir_entry {
    /* Numer i-węzła
    unsigned short low_ino;
    /* Długośc nazwy pliku */
    unsigned short namelen;
    /* Nazwa pliku */
    const char *name;
    /* Atrybuty pliku (czy jest katalogiem itp.) */
    mode_t mode;
    /* Liczba plików w katalogu (dla katalogów) */
    nlink_t nlink;
    /* Identyfikator właściciela pliku */
    uid_t uid;
    /* Identyfikator grupy właściciela pliku */
    gid_t gid;
    /* Długość pliku */
    unsigned long size;
    /* Wskaźnik do struktury opisującej dostępne operacje na pliku */
    struct inode_operations *ops;
    /* Funkcja wywoływana przy odczycie pliku */
    int (*get_info)(char*,char**,off_t,int,int);
    /* Funkcja ustawiająca w i-węźle odpowiednie atrybuty dla pliku */
    void (*fill_inode)(struct inode*);
    /* Wskaźnik do następnego pliku w katalogu */
    struct proc_dir_entry *next;
    /* Wskaźnik do katalogu nadrzędnego pliku */
    struct proc_dir_entry *parent;
    /* Wskaźnik do pierwszego pliku w katalogu (dla katalogów) */
    struct proc_dir_entry *subdir;
    /* Dane */
    void *data;

    /* Pola dodane w wersji 2.2 jądra */

    /* Wskaźniki do funkcji realizujących różne zadania
    związane z plikiem lub katalogiem */
    int (*read_proc)(char*,char**,off_t,int,int*,void*);
    int (*write_proc)(struct file*,const char*,unsigned long,void*);
    int (*readlink_proc)(struct proc_dir_entry*,char*);
    /* Licznik użyć */
    unsigned int count;
    /* Flaga usunięcia */
```

```
    int deleted;
}
```

Ze struktur takich jest zbudowane całe **drzewo plików i katalogów proc-a**. Każda z nich opisuje jakiś plik lub katalog. Wyjątkiem są katalogi procesów. Wszystkie one korzystają z tej samej gałęzi drzewa struktur `proc_dir_entry`. Poszczególne katalogi procesów są rozróżniane na poziomie funkcji dostępu do katalogów. To znaczy jeżeli czytany jest katalog zawierający (jedno) `proc_dir_entry` katalogu procesu, to odpowiednia funkcja `read_dir` interfejsu vfs zinterpretuje to dla każdego aktualnie działającego procesu z osobna. Podobnie gdy chcemy wejść do katalogu procesu, to odpowiednia funkcja `lookup` wyszuka żądany proces i zwróci jego sztuczny i-węzeł. Dla innych plików i katalogów jest normalnie. Są one wykrywane tylko wtedy, gdy posiadają swoją strukturę `proc_dir_entry`.

Drzewo struktur `proc_dir_entry` plików i katalogów jest budowane w wielu miejscach kodu. Zasadniczo każdy podkatalog jest tworzony w innej procedurze; przykładowo katalog główny proc w procedurze `proc_root_init`, a katalog procesu w procedurze `proc_base_init`. Drzewo jest budowane za pomocą funkcji `proc_register` i `proc_register_dynamic`. Obie umożliwia dołączenie podrzędnego węzła drzewa do nadrzędnego (katalogu), z tym, że druga pozwala utworzyć dynamicznie nowy numer i-węzła dla podrzędnego węzła (pole `low_ino` struktury `proc_dir_entry`). Pierwsza zakłada, że oba węzły zostały już zdefiniowane wcześniej (patrz rysunek). Dodatkowo są różne inne pomocnicze funkcje takie jak `create_proc_entry` lub `remove_proc_entry`.

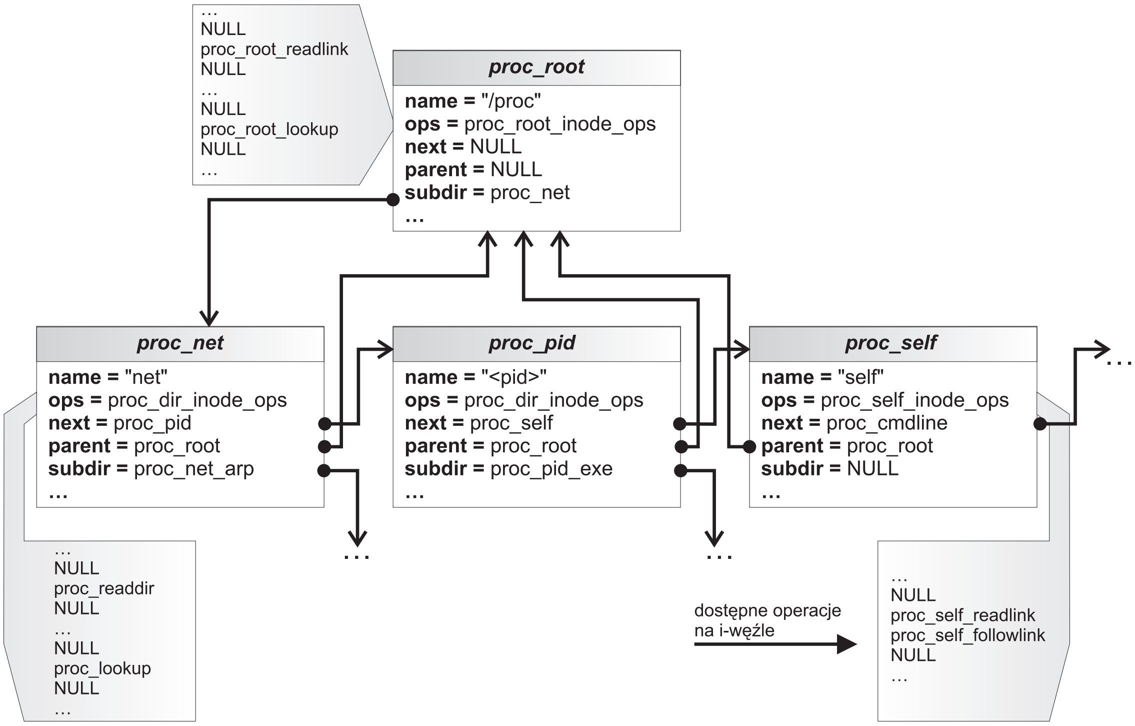
Interfejs vfs proc-a w swoich funkcjach korzysta właśnie z tych struktur. Na ich podstawie identyfikuje pliki i katalogi proc-a (numery i-węzłów), generuje i-węzły (wyznacza odpowiednie nazwy, tryby i prawa dostępu). Wskaźnik `get_info` w strukturze `proc_dir_entry` domyślnie ma wskazywać na funkcję zwracającą zawartość odpowiedniego pliku lub katalogu. Często wskaźnik ten jest pusty. Może to oznaczać, że zwracanie zawartości jest obsługiwane na poziomie funkcji interfejsu (tzn. interfejs sam wyznacza odpowienią funkcję nie odwołując się do `proc_dir_entry`). W pliku `fs/proc/array.c` zdefiniowane zostały operacje na wirtualnym i-węźle `array`, które opisują domyślne zachowanie dla plików proc-a. W zalezności od numeru i-węzła wywoływana jest odpowiednia funkcja zwracająca zawartość pliku specyficzna dla niego. W tym samym pliku znajduje się wiele z nich. Mogą one stanowić przykład dla programistów, którzy chcieliby napisać własne pliki proc.

Interfejs vfs proc-a

Proc jak każdy inny system plików musi obsługiwać różne funkcje interfejsu vfs, poprzez które jądro może się z nim komunikować. Funkcje te to zarówno funkcje ogólne dotyczące systemu plików jako całości, jak i funkcje dotyczące poszczególnych plików.

Podstawowa struktura opisująca system plików `super_block` jest zwracana przez funkcję `proc_read_super`. Inną ważną funkcją struktury **super_operations** dotyczącą całego proc-a jest `proc_read_inode`, która zwraca odpowiedni i-węzeł pliku lub katalogu na podstawie numeru i-węzła.

Poszczególne pliki udostępniają zróżnicowane interfejsy. Standardowo zaimplementowane są tylko niektóre funkcje; dla katalogów funkcja odczytu katalogu `readdir` i funkcja wchodzenia do katalogu `lookup`; dla plików funkcja odczytu `read`; dla linków funkcje czytania linku `readlink` i podążania za linkiem `follow_link`. Dla niektórych plików jest dodatkowo funkcja zapisu pliku `write` lub inne. Definicje interfejsów plików i katalogów są umieszczone w strukturach typu



Rysunek 4.10: Fragment drzewa struktur proc_dir_entry

file_operations i **inode_operations** w różnych miejscach źródeł proc-a. Razem z nimi znajdują się implementacje odpowiednich funkcji interfejsu. Szczególnie ciekawe są funkcje opisujące dostęp do pamięci procesu zaimplementowane w pliku `fs/proc/mem.c`

4.11.4 Opis wybranych plikow i katalogow proc-a

Poniższa lista opisuje jedynie wybrane pliki i katalogi proc-a. Opisanie dokładnie wszystkich zajęłoby za dużo miejsca i wykracza poza zakres tego podrozdziału. Dokładniejsze informacje na temat tych plików i innych można znaleźć w man-ie do proc-a lub w dokumentacji jądra linuxa w pliku `proc.txt`

- `proc/cmdline` - informacje o trybie uruchomienia systemu
- `proc/cpuinfo` - informacje o procesorze
- `proc/devices` - lista numerow urzadzen
- `proc/dma` - lista zarejestrowanych kanalow DMA
- `proc/filesystems` - lista dostepnych systemow plikow
- `proc/interrupts` - lista przerwan wraz z liczbami wywolan

- `proc/ioports` - lista zarejestrowanych regionow portow I/O
- `proc/kcore` - pamiec fizyczna systemu (format pliku core)
- `proc/kmsg` - jezeli nie pracuje syslog, to lista wiadomosci jadra
- `proc/ksyms` - lista definicji symboli modulow
- `proc/meminfo` - informacje o pamieci
- `proc/modules` - lista modulow zaladowanych do systemu
- `proc/mounts` - lista zamontowanych systemow plikow
- `proc/net/` - katalog z roznymi plikami dotyczacymi sieci
- `proc/net/route` - lista interfejsow sieciowych
- `proc/pci` - lista znalezionych urzadzen PCI
- `proc/scsi/` - katalog z roznymi plikami dotyczacymi urzadzen SCSI
- `proc/self/` - link do katalogu procesu odwolujacego sie do proc
- `proc/stat` - statystyki jadra systemu
- `proc/sys/` - katalog z roznymi plikami dotyczacymi systemu
- `proc/sys/fs/` - katalog z roznymi zmiennymi systemu plikow
- `proc/sys/kernel/` - katalog z roznymi zmiennymi jadra
- `proc/sys/net/` - katalog z roznymi zmiennymi dotyczacymi sieci
- `proc/sys/vm/` - katalog z roznymi zmiennymi dotyczacymi pamieci wirt.
- `proc/uptime` - czas dzialania systemu
- `proc/version` - informacje o wersji jadra
- `proc/<pid>/` - katalog z informacjami o procesie o numerze <pid>
- `proc/<pid>/cmdline` - linia wywolania (komendy) procesu
- `proc/<pid>/cwd` - link do biezacego katalogu procesu
- `proc/<pid>/environ` - zmienne srodowiskowe procesu
- `proc/<pid>/exe` - link do wykonywanego kodu procesu
- `proc/<pid>/fd` - katalog uzywanych deskryptorow plikow
- `proc/<pid>/maps` - lista obecnie zmapowanych regionow pamieci procesu
- `proc/<pid>/mem` - pamiec procesu
- `proc/<pid>/root` - link do katalogu glownego dla procesu

4.11.5 Zmiany proc-a w jądrze 2.2 w stosunku do jądra 2.0

- dostosowanie do struktury `dentry`, która się pojawiła w jądrze 2.2
- przebudowa sposobu (miejsce i kolejność) budowania drzewa proc-a
- całkowite przerzucenie algorytmu wyznaczania funkcji zwracającej zawartość pliku z funkcji `proc_read_inode` do funkcji `array_read` interfejsu standardowo przypisywanego prostym plikom.
- dodanie różnych funkcji pomocnych przy budowaniu drzewa: `create_proc_entry`, `remove_proc_entry`
- dodanie nowych plikow/katalogów np. `mca, hardware, stram, partitions`
- dodanie w niektórych miejscach implementacji funkcji interfejsów plików
- zdefiniowanie innego alternatywnego do `array` interfejsu vfs `generic` dla pliku proc
- dodanie zabezpieczen zwiazanych ze zmiana biezacego katalogu glownego poprzez `chroot`, przez co zwiekszone zostalo bezpieczenstwo przy montowaniu proc-a w takich przypadkach.
- dodana obsluga `openpromfs` Sun-a

4.12 Zadania

1. Czytanie i-węzła[25]

   Mamy system plików EXT2, w którym przyjmujemy, że każda grupa bloków składa się (kolejno) z:[26]

   - nagłówka, przechowującego: deskryptor grupy, bitową mapę zajętości i-węzłów (wypełnia 1 blok) oraz bitową mapę zajętości bloków (wypełnia 1 blok); nagłówek ma łączny rozmiar `GROUP_HDR_SIZE` = 128 KB,
   - tablicy i-węzłów,
   - właściwych bloków zawierających dane, każdy po `BLOCK_SIZE` = 4 KB

   Zakładamy, że jeden i-węzeł w zajmuje `INODE_SIZE` = 256 bajtów.

   Zaszło zdarzenie: funkcja `iget()` nie znalazła i-węzła `INODE_NR` = **164546** w buforach i-węzłów, trzeba więc wczytać go z dysku (funkcją `ext2_read_inode()`). Aby to było możliwe, należy wyznaczyć: bezwzględny numer bloku `BLOCK_NR`, zawierającego szukany i-węzeł z tablicy i-węzłów oraz przesunięcie w tym bloku (`INODE_OFFSET`), gdzie i-węzeł się znajduje.

   Zadanie polega na wyliczeniu liczb `BLOCK_NR` oraz `INODE_OFFSET` dla i-węzła `INODE_NR`.

   Uwaga: pomijamy fakt zajmowania bloków przez *Master Boot Record* itp. uznając, że mamy do czynienia wyłącznie z zapisanymi kolejno grupami, więc zerowy blok należy do pierwszej grupy, a ostatni do ostatniej.

2. Rozważmy następujący prosty system plików: rozmiar bloków na dysku to 16 bajtów, numery bloków zajmują 2 bajty, pliki opisywane są przez i-węzły które zawierają 6 numerów bloków, w tym 3 numery bloków bezpośrednich. Jeżeli dostęp do pliku jest sekwencyjny stosowane jest czytanie bloków z wyprzedzeniem, czytane są zawsze dodatkowe 3 bloki, chyba że nie pozwala na to rozmiar pliku. Poniżej znajduje się zawartość dwóch i-węzłów, oraz pięciu bloków dyskowych. Dla podanej sekwencji żądań odczytu podaj ilość wczytanych bloków do pamięci, oraz liczbę oczekiwań na zakończenie operacji wejścia wyjścia Przyjmij że i-węzły znajdują się już w pamięci, a wypełnione bufory będą dostępne przez cały analizowany czas, zaś odstęp pomiędzy kolejnymi żądaniami odczytu jest wystarczający do zakończenia czytania asynchronicznego.[27]

   Żądania czytania: (i-węzeł, Pozycja w pliku, Ilość bajtów)

   (15, 8, 8); (15, 16, 8); (31, 0, 16); (15, 56, 8); (15, 64, 8); (31, 16, 16); (31, 32, 8); (15, 184, 8); (15, 192, 16); (15, 208, 8);

3. Istnienie warstwy VFS umożliwia między innymi dostęp w Linuksie do wielu systemów plików (FAT, NFS). W systemie korzysta się ze wskaźników do funkcji implementujących właściwe rozwiązania. Wymień struktury w których zawarte są wskaźniki do funkcji związanych z VFS (podaj po jednym przykładzie funkcji). Czy wiesz jaka filozofia kryje się za takim rozmieszczeniem.[28]

[25] Autor: Jarosław Staniek
[26] podane parametry nie koniecznie muszą się zgadzać z rzeczywistością :)
[27] Autor: Tomasz Zieliński
[28] Autor: Tomasz Zieliński

Tablica 4.6: Zawartość i-węzłów oraz bloków

i-węzeł 15	i-węzeł 31	blok 19	blok 30	blok 68	blok 70	blok 86
...	...	101	68	92	27	80
...	...	102	86	100	28	81
16	40	111	–	104	29	83
17	90	112	–	105	6	84
18	91	113	–	106	123	–
19	–	200	–	107	124	–
30	–	201	–	70	–	–
–	–	202	–	71	–	–

4. W systemie działają dwa procesy (P1 i P2), które odwołują się do danych z tego samego urządzenia. Proces P1 tylko je odczytuje, natomiast P2 każdorazowo modyfikuje dane. Pi-j oznacza, że proces i-ty odwołuje się do j-tego bloku danego systemu plików. Dana jest następująca sekwencja odwołań:
   P1-4 P2-17 P1-8 P2-4 P1-6 P2-9 P1-9 P2-6 P1-17 P2-3 P1-5 P2-5
   Zakładamy, że początkowo w pamięci podręcznej nie było żadnego bloku z tego urządzenia, oraz że lista buforów wolnych jest na tyle wypełniona, aby nie trzeba było jej dopełniać.
   Ile operacji wejścia- wyjścia wykona jądro na potrzeby tych procesów?[29]

5. Podczas czytania sekwencyjnego pliku, jądro wykonuje algorytm odczytu asynchronicznego. W wersji 2.2.12 może przeczytać maksymalnie NBUF=16 bloków (łącznie z blokiem podanym jako parametr - tym, od którego rozpoczynamy czytanie i który czytany jest synchronicznie).
   Zakładamy, że w pamięci znajdują się bloki:
   3 5 11 14 24 27 48 60 75
   tego pliku (plik składa się ze 100 bloków) oraz, że zawierają one aktualne dane. Ile bloków przeczyta jądro wykonując algorytm czytania z wyprzedzeniem breada
   a) bloku nr 11
   b) bloku nr 70
   Czy odpowiedź uległaby zmianie, gdyby dane w pamięci nie były aktualne?[30]

6. Opisz działanie algorytmu `namei` dla nazwy ścieżkowej `../etc/fstab`. Katalogiem bieżącym dla procesu jest `/home/`. Wszystkie katalogi ze ścieżki znajdują się na tym samym systemie plików. [31]

7. Oblicz średni czas działania algorytmu `namei` dla ścieżki `/home/inf/adam/ważne/list.txt` wiedząc, że:

   - nazwy katalogów `/home/` i `/home/inf/` znajdują się w podręcznej pamięci buforowej nazw katalogów
   - pozycja o nazwie `adam` jest 48 pozycją w pliku katalogowym `/home/inf`

[29] Autor: Elżbieta Filipkowska
[30] Autor: Elżbieta Filipkowska
[31] Autor: Damian Wójtowicz

- pozycja o nazwie `ważne` jest 8
- pozycja o nazwie `list.txt` jest 56
- dwa pierwsze bloki dyskowe zajmowane przez plik katalogowy `/home/inf/adam` znajdują się w pamięci
- średnia wielkość pozycji katalogowej wynosi $2^5 b$
- wielkość bloku dyskowego wynosi $1kb$
- średni czas wczytania bloku dyskowego (łącznie z odnalezieniem odpowiedniej ścieżki na dysku) wynosi $10^{-3}s$
- czas sprawdzenia czy pozycja jest w pamięci buforowej nazw katalogów wynosi średnio $5\mu s$
- średni czas przeszukania całego bloku (znajdującego się w pamięci) w poszukiwaniu pozycji o danej nazwie wynosi $200\mu s$

Pomijamy czas sprawdzania czy mamy doczynienia z punktem montowania lub dowiązaniem symbolicznym. Wszystkie katalogi ze ścieżki znajdują się w tym samym systemie plików.

Uwaga! Podane wartości liczbowe nie są do końca zgodne z rzeczywistością, zostały dobrane tak aby uprościć obliczenia. [32]

8. Stworzono katalog `/home/ola/zadania` i utworzono w nim pliki o nazwach: `mailstat`, `userinfo`, `sysalter`, `fmap.c`, `fmap.h`, `testfmap.c`, `wyniki.txt`, `dokumentacja.txt`. Nastepnie usunięto z niego pliki `testfmap.c` oraz `wyniki.txt`. To były jedyne operacje na katalogu.

   Oblicz wielkość pliku katalogowego powstałego dla powyższego katalogu (chodzi to o ilość danych w bajtach). Ile w rzeczywistości on zajmuje? [33]

9. Jaka będzie zawartość zbioru, na który wskazuje pole `open_fds` struktury `files_struct` po wykonaniu poniższego ciągu wywołań funkcji `open()` i `close()` przy założeniu, że na początku otwarte były jedynie deskryptory 0,1 i 2 oraz, że wszystkie wywołania zakończyły się poprawnie (`n x open(...)` oznacza n-krotne wywołanie funkcji `open()` z pewnymi parametrami):

   `6 x open(...), close(1), 2 x open(...), close(6), close(3), open(...)` ? [34]

10. Uszereguj wywołania funkcji `open()` w zadaniu nr 1 pod względem długości wykonania (zajmowania czasu procesora) ? [35]

11. Chcemy utworzyć i zamontować nowy własny system plików, który strukturę drzewa, pliki i inne potrzebne informacje przechowuje w pewien sposób w pliku w katalogu głównym Linuxa. Co w tym celu musimy zrobić (zaimplementować lub zmienić w jądrze)? Zakładamy, że funkcjonalność tego systemu plików jest minimalna (chodzenie po katalogach, czytanie i zapisywanie plików).[36]

---

[32] Autor: Damian Wójtowicz
[33] Autor: Damian Wójtowicz
[34] Autorzy: Bartłomiej Goldman, Bartłomiej Rymkowski
[35] Autorzy: Bartłomiej Goldman, Bartłomiej Rymkowski
[36] Autor: Paweł Kwiatkowski

12. Gdyby w jednym bloku mieściło się 274 i-węzłów, a i-węzeł numer 411 znajdowałby się w 17 bloku na 1360 pozycji, to jaki byłby rozmiar i-węzła na dysku? Jaki jest numer bloku, od którego rozpoczyna się lista i-węzłów?[37]

13. Podaj maksymalny rozmiar pliku, który można byłoby zapamiętać gdyby:
    a) nie było potrójnego bloku pośredniego;
    b) byłby dodatkowo blok poczwórny pośredni.

    Zakładamy, że przypadku b) w i-węźle pole rozmiaru pliku będzie odpowiednio duże.[38]

14. Gdzie w tablicy danego i-węzła znajduje się blok o numerze logicznym 530?
    Zakładamy standardowe wielkości stałych.[39]

15. W systemie mamy łącze nienazwane L oraz trzy procesy spokrewnione P0, P1 i P2 mające dostęp do L. Poniżej zapisany jest harmonogram działania procesów, symbolicznie uporządkowany. Określ po każdym kroku stan łącza oraz procesów. Zakładamy, ze P0 ma otwarty deskryptor do pisania a P1 i P2 do czytania. Kolejność jest liczona symbolicznie godzinami.

    Przykład:

    00.00 Proces P2 próbuje czytać z L (L puste) 1 kilobajt danych bez flagi O_NONBLOCK

    Odp: 00.00;L puste, P2 zawieszone.

    Harmonogram:

    00.00 Proces P0 próbuje pisać do łącza 3 kb.

    01.00 Proces P0 próbuje pisać do łącza 3 kb z flaga O_NONBLOCK.

    02.00 Proces P0 próbuje pisać do łącza 2 kb.

    03.00 Proces P1 czyta z L 2 kb.

    04.00 P2 czyta z L 4kb.

    05.00 P1 czyta z L 1kb.

    06.00 P2 czyta z L 1kb z flagą O_NONBLOCK.

    07.00 P0 pisze do L 2 kb.

    08.00 P1 czyta z L 1kb.

    09.00 P0 zamyka deskryptor L.

    10.00 P1 czyta z L 1kb.

16. Zastanowić się, dlaczego w standardzie BSD nie wykrywa się zakleszczeń.[40]

17. Komunikacja między procesami za pomocą `FIFO`. Opisać schemat komunikowania się dwóch procesów. Jeden pisze do `FIFO`, drugi czyta z tego samego `FIFO`.[41]

[37] Autor: Paulina Domagalska
[38] Autor: Michał Wiktor Żmijewski
[39] Autor: Michał Wiktor Żmijewski
[40] Autor: Michał Domagała
[41] Autorzy : Jakub Lenart, Krzysztof Juchimiuk

18. Zadanie dotyczy algorymu bmap służącego do przekształcania logicznego adresu bajtu w pliku na adres fizyczny na dysku. Mając dany numer logiczny bloku np. 34, znaleźć jego pozycję na dysku.[42]

19. Załóżmy istnienie hipotetycznego systemu plików,w którym plik składa się z:
dziesięciu bloków adresowanych bezpośrednio
pewnej ilości bloków z adresami w dwóch blokach pojedynczych pośrednich
pewnej ilości bloków,do których można się dostać przez dwa bloki podwójne pośrednie
pewnej ilości bloków,do której można się dostać przez jeden blok potrójny pośredni.
Niech blok zawiera 1024 bajty,lub 128 adresów bloków. Ile wynieść może maksymalny rozmiar logiczny pliku? [43]

20. W ext2_new_inode sprawdzenie quoty odbywa się po odblokowaniu super_bloku. Czy czymś to grozi? Czy swarzałaby jakieś zagrożenie sytuacja,w której to sprawdzenie następuje przed odblokowaniem super_bloku?[44]

21. Funkcje obsługi bloków wykorzystują trzy różne funkcje obsługi sytuacji nietypowych/błędnych: `ext2_warning`, `ext2_error` i `ext2_panic`, co określa trzy poziomy błędów, jakie może zwrócić funkcja. Dlaczego dokonano takiego podziału?[45]

22. Wyobraźmy sobie system plików SCROOGE o następujących właściwościach:

    - pliki są tylko do odczytu
    - nie ma drzewiastej struktury katalogów - wszystkie pliki są w głównym katalogu
    - dane są zapisane na urządzeniu fizycznym podzielonym na dwa obszary
        - obszar 1 zawiera dane dotyczące plików na urządzeniu: numer, nazwa, wielkość, pozycja na urządzeniu (w bajtach)
        - obszar 2 zawiera dane plików, fizycznie jest podzielony na sektory wielkości 512 bajtów, jednak każdy plik moze się zaczynać w dowolnym miejscu sektora (brak wewnętrznej fragmentacji)
    - sterownik urządzenia umożliwia wczytywanie w dowolny sposób danych z 1 obszaru, z 2-go potrafi zwrócić sektor 512 bajtowy o podanym numerze.

    Dla takiego systemu plików należy wymyśleć w jaki sposób powinny działać funkcje:
    `file_operations->read`
    `inode_operations->lookup`
    `read_super`

    Jeżeli uważasz, że są potrzebne dodatkowe informacje w `struct inode` lub `struct super_block` wymień je.[46]

23. Porównaj działanie funkcji `read_super` dla systemów plików proc i ext2. Uzasadnij występujące różnice. Na przykładzie różnic wskaż podstawowe zalety wirtualnego systemu plików.[47]

[42] Autorzy : Jakub Lenart, Krzysztof Juchimiuk
[43] Autor: Bartosz Kapturkiewicz
[44] Autor: Bartosz Kapturkiewicz
[45] Autor: Bartosz Kapturkiewicz
[46] Autor: Tomasz Dziki
[47] Autor: Marcin Żukowski

4.13 Rozwiązania zadań

1. Czytanie i-węzła

   Wiedząc, że bitmapa zajętości i-węzłów zajmuje 1 blok, mamy, że maksymalna liczba i-węzłów w grupie to: `INODES_PER_GROUP = 1*BLOCK_SIZE*8 = 4096 * 8 = 16384`. Analogicznie, dla 4KB bitmapy zajętości bloków, mamy liczbę bloków dla danych w grupie: `DATA_PER_GROUP` = 16384. W 1 grupie można więc zapisać 16384*4096 = 128 MB danych.

   W sumie 1 grupa zajmuje:

   `BLOCKS_PER_GROUP`

   `= GROUP_HDR_SIZE / BLOCK_SIZE + INODES_PER_GROUP*INODE_SIZE`

   `/ BLOCK_SIZE + DATA_PER_GROUP`

   `= 128KB/4096 + 16384*256/4096 + 16384`

   = $2^{17}/2^{12} + 2^{14}*2^{8}/2^{12} + 2^{14}$

   `= 17440` bloków.

   Mając rozmiar grupy, wyznaczamy numer grupy, w której leży szukany i-węzeł, pamiętając, że i-węzły numeruje się od 1:

   `GROUP_NR = (INODE_NR-1) / INODES_PER_GROUP = 164545/16384 = 10` (licząc od 1: 11)

   Przesunięcie w bajtach w tablicy i-węzłów do miejsca, w którym jest zapisany szukany i-węzeł:

   `INODES_TAB_OFFSET = ((INODE_NR-1) mod INODES_PER_GROUP) * INODE_SIZE`

   `= (164545 mod 16384) * 256 = 705 * 256`

   Ostatecznie:

   Numer bloku, w którym zapisano szukany i-węzeł:

   `BLOCK_NR = BLOCKS_PER_GROUP*GROUP_NR + INODES_TAB_OFFSET/BLOCK_SIZE`

   `= 17440*10 + 705*256/4096 = 174400 + 705/16 =` **174444**

   Przesunięcie bajtowe w tym bloku:

   `INODE_OFFSET = INODES_TAB_OFFSET mod BLOCK_SIZE = 705*256 mod 4096 =` **256**

2. Przeczytanych zostanie 17 bloków, trzeba będzie czekać na zokończenie 10 operacji we-wy. W rozwiązaniu pomocna jest tabela (C-oznacza oczekiwanie):

Żądanie	Pozycja po	Blok	Wczytane bloki	Komentarz
(15, 8, 8)	16	16	16	C; Po prostu czytamy blok
(15, 16, 8)	24	17	17,18,19,20	C; Sekwencyjne, czytamy z wyprzedzeniem
(31, 0, 16)	16	40	40	C;
(15, 56, 8)	64	101	101	C; Blok pojedynczo pośredni był już w pamięci (19)
(15, 64, 8)	72	102	102,103,104,105	C; Sekwencyjne
(31, 16, 16)	32	90	90,91	C; Sekwencyjne, ale czytamy tylko 2 bloki

				ponieważ koniec pliku
(31, 32, 8)	40	91	–	Już jest
(15, 184, 8)	192	92	30,68,92	Cx3; Czytamy blok podwójnie i pojedynczo pośredni, aż trzy operacje
(15, 192, 16)	208	100	100	C; Sekwencyjne, ale nie czytamy kolejnych bo blok 101 już jest w pamięci
(15, 208, 8)	216	104	–	Już jest

3. Oto przykładowe funkcje wraz ze strukturami w których się znajdują:

   - Funkcja `read_super`; `struct file_system_type`.
   - Funkcje `read_inode, write_inode, write_super`; `struct super_operations`.
   - Funkcje `create, mkdir, rename`; `struct inode_operations`.
   - Funkcje `read, write`; `struct file_operations`.

   Uwaga, `struktury super_operations, inode_operations` oraz `file_operations` są wskazywane przez odpowiednie pola struktur `super_block, inode i file`; wymienienie tych struktur również jest poprawne (logicznie bowiem są one ich częścią).

   Starano się tak rozmieścić wskaźniki by dostęp do funkcji odbywał się poprzez te struktury które w wyniku jej działania zostają zmodyfikowane (pamiętaj o uwadze).

4. Pierwsze 3 odwołania wymagają przeczytania danych, następnie proces P2 sięga do bloku 4, który już jest w pamieci i zawiera aktualne dane. Kolejne 2 odwołania (bloki 6 i 9) także wymagają kopiowania danych z urządzenia. Następnie proces P1 chce czytać z bloku nr 9, blok ten jest w pamięci, ale zawiera modyfikowane dane i dlatego należy je przedtem zapisać. W dalszej kolejnośći proces P2 odwołuje się do danych, które są w pamięci i są aktualne. Proces P1 zapisuje blok 17 (podobnie jak wcześniej blok nr 9). Następnie P2 czyta z urządzenia blok nr 3 (nie było go w pamięci), P1 czyta blok nr 5 (z tego samego powodu). Na koniec proces P2 odwołuje się do bloku, którego nie musi fizycznie odczytywać z urządzenia, bo jest on już w pamięci i zawiera aktualne dane. Łącznie 9 operacji wejścia-wyjścia.

5. a) Jądro nie czyta fizycznie żdnych danych. Pobiera pierwszy blok z pamięci podręcznej i kończy działanie algorytmy breada.
   b) Jądro wczytuje synchronicznie pierwszy blok (nr 70), oraz inicjuje asynchroniczne czytanie czterech kolejnych bloków (do bloku 75, który jest już w pamięci i zawiera aktualne dane). Jądro zaprzestaje czytania bloków gdy natrafi na blok zawierajacy aktualne dane. Gdy zatem dane w pamieci podrecznej nie były aktualne algorytm ten nie zakończyłby się tak szybko. Jądro wczytałoby tyle bloków, ile może maksymalnie.

6. Nazwa ścieżkowa zaczyna się od `..`, więc przeszukiwanie rozpoczyna się od katalogu bieżącego, czyli `/home/`, którego i-węzeł staje się bieżącym i-węzłem.Sprawdza się czy bieżący i-węzeł odpowiada katalogowi oraz czy proces posiada prawo do przeszukiwania go. Pobierana jest pierwsza składowa napisu (`..`) i przeszukiwane są kolejne bloki pozycje w pliku bieżącego katalogu, aż odnajdzie się pozycje o nazwie `..` (powinna ona znajdować się jednej

z dwóch pierwszych pozycji). Jądro zwalnia bieżący i-węzeł i przydziela go znalezionej właśnie pozycji. Następnie znów sprawdza się rodzaj pliku i prawa (jak wyżej). Analogicznie jak przy pierwszym obrocie pętli pobiera się kolejną składową nazwy (`etc`) i przeszukuje bloki pliku katalogowego w poszukiwaniu pozycji o nazwie `etc`. Kiedy pozycja zostanie znaleziona podmienia się bieżący i-węzeł. Kolejny, i już ostatni, obrót pętli algorytmu jest analogiczny do poprzednich i w jego wyniku bieżącym i-węzłem został i-węzła dla pliku `/etc/fstab`. Ponieważ przetworzono już całą nazwę ścieżkową, więc jako wynik zwrócony zostaje numer ostatnio znalezionego i-węzła.

7. Najpierw zauważmy, że średnio w bloku dyskowym znajdują się $\frac{2^{10}b}{2^5b} = 32$ pozycje katalogowe. Teraz możemy już przejść do właściwych obliczeń.

$$[(5)+(5)+(5+200+\frac{1}{2}\cdot 200)+(5+1000+\frac{1}{4}\cdot 200)+(5+1000+200+1000+\frac{3}{4}\cdot 200)]\mu s = 3725\mu s$$

8. Ilość plików w katalogu (łącznie z usuniętymi) wynosi 10, ponieważ w każdym katalogu znajdują się zawsze pliki o nazwach `.` i `..`. Długości nazw plików (po zaokrągleniu do 4n bajtów) wynoszą odpowiednio (pliki `.`, `..` a następnie wg. kolejności ich wyliczenia w treści): 4, 4, 8, 8, 8, 8, 8, 12, 12, 16. Stąd wielkość pliku katalogowego wyności 168 bajtów, gdyż:

$$10 \cdot (4+2+2) + 2 \cdot 4 + 5 \cdot 8 + 2 \cdot 12 + 16 = 168.$$

W rzeczywistości plik ten zajmuje 1024 bajty, ponieważ jest mu przydzielany cały blok dyskowy.

9. Ustawione będą bity 0..5 oraz 7..9.

10. Najdłużej wykonywać się będzie ósme wywołanie funkcji `open()`, ponieważ sprawdzane będą kolejno deskryptory o numerach 2..9. Wszystkie inne wywołania tej funkcji wykonywać się będą tak samo długo - w każdym przypadku `next_fd` wskazuje wolny deskryptor.

11. Należy wykonać odpowiednie rzeczy:

    (a) Pierwszą czynnością jest stworzenie interfejsu do systemu plików. W tym celu należy zaimplementować funkcje:

    - `read_super` - operacja odczytu superbloku dla systemu plików
    - `super_operations` - dla superbloku operacje opisujące w jaki sposób odczytywać (`read_inode`) i zapisywać (`write_inode`) i-węzły w naszym pliku
    - `inode_operations` - dla i-węzłów operacje jak oglądać zawartość katalogu (`lookup`), jak tworzyć (`mkdir`) i usuwać (`rmdir`) katalogi
    - `file_operations` - dla plików operacje jak czytać (`read`) i zapisywać (`write`) pliki oraz jak czytać katalogi (`readdir`)

    Podczas implementacji tych funkcji szczególnie należy się zastanowić nad odpowiednim przekształcaniem danych z naszego pliku na linuksowe i-węzły, w szczególności nad tym jakie będą ich unikalne numery.

(b) Następnie tworzymy strukturę `file_system_type`, która opisuje nasz system plików (jak system może się do niego dostać)

(c) Używając tej struktury procedurą `register_filesystem` rejestrujemy system plików w systemie

(d) Montujemy system plików procedurą `mount`

12. Rozmiar i-węzła: 10 Numer bloku: 16

13. a) $1024b * (12 + (1024b/4b) + (1024b/4b)^2) = 64,26MB$;
b) $1024b * (12 + (1024b/4b) + (1024b/4b)^2) + (1024b/4b)^3 + (1024b/4b)^4) = 4112GB$.

14. W bloku podwójnie pośrednim, a dokładnie:
$530 > 12$ - zatem nie w bezpośrednich $wzgl = 530 - 12 = 518$,
$518 > \frac{1024b}{4b}$ - zatem nie w pośrednim pojedynczym $wzgl = wzgl - 256 = 262$,
$262 < (\frac{1024b}{4b})^2$ - zatem znajduje się w bloku podwójnym pośrednim,
$262 > \frac{1024b}{4b)}$ - nie na pierwszej pozycji listy bloków pojedynczych pośrednich $wgzl = wzgl - 256 = 6$,
$6 < \frac{1024b}{4b}$ - na drugiej pozycji w tej liscie, i na szóstej w liście bloków bezpośrednich.

(podwójny pośredni -> lista pośrednich pojedynczych (2 pozycja) -> lista bezpośrednich (6 pozycja)).

15. 00.00 L : 3kb

01.00 L: 4kb, to czego nie zapisał PO zostało utracone

02.00 L: 4kb, P0 zawieszony

03.00 L:2kb

03.05 L:4kb, P0 zapisał 2kb i jest odwieszony

04.00 L:0kb

05.00 L:0kb, P1 zawieszony

06.00 j.w.

07.00 L:2kb, P1 zawieszony

07.05 L:1kb, P1 odwieszony

08.00 L:0kb

10.00 L:0kb, P1 nie zawieszony choć, czytał wstrzymująco, ponieważ brak procesu piszącego do L.

16. W porównaniu ze standardem POSIX, funkcja blokująca flock jest uboga. Zakładane blokady nie są tak rozbudowane. W szczególności przechowywana jest tylko informacja o stanie plików, a nie jest pamiętana informacja o właścicielach blokad. Zakleszczenie jest stanem procesów, a w standardzie BSD procesy są "anonimowe". Dlatego sprawdzanie zastoju jest niemożliwe.

17. `FIFO` ma dwa deskryptory. Proces czytajacy otwiera jeden deskryptor `FIFO` do czytania, drugi do pisania. Następnie ten, który pisze, czeka, aż w będzie miejsce dla jego przesyłki, zaś ten, który czyta, czeka, aż będzie co czytać.

18. Jak wiadomo struktura inode pliku zawiera adresy bloków bezpośrednich, pośrednich pierwszego poziomu (takich które wskazują na bloki z adresami do bloków bezpośrednich), pośrednich drugiego poziomu (wskazują na bloki pośrednie pierwszego poziomu), pośrednich trzeciego poziomu. Najpierw szukamy tego adresu pośrod adresów bloków bezpośrednich.Jeśli jest większy niż liczba adresów bezpośrednich to odejmujemy od niego tę liczbę i szukamy go pośród adresów pośrednich pierwszego poziomu. Jeżeli nie znajdziemy go tutaj, to należy zajrzeć do adresów pośrednich drugiego rodzaju. Itd.

19. $128^3 + 2 * 128^2$+2*128+10=2 181 310 464 bajty = ok. 2 gb.

20. Na pierwsze pytanie odpowiedź brzmi : nie, ponieważ:

    Po pierwsze,sprawdzenie quoty polega na sprawdzeniu, czy po dodaniu jednego i-węzła do dotychczasowej ilości i-węzłów użytkownika/grupy nastąpi przekroczenie quoty,a nie na sprawdzeniu już po utworzeniu pliku,czy nie spowodował on przekroczenia. Uniemożliwia to sytuację,w której np.wiele procesów po pobraniu i wypełnieniu i-węzłów stwierdza,że należy je oddać,bo ilość i-węzłów użytkownika/grupy plus ilość nowych plików wykracza poza zakres,podczas gdy na kilka z nich miejsca by wystarczyło.

    Po drugie,sprawdzenie to nie może być wykonywane przez kilka procesów jednocześnie,zatem nie może dojść do sytuacji,w której quota zostaje przekroczona z powodu przeczytania przez procesy nieaktualnych danych dotyczących ilości i-węzłów użytkownika/grupy.

    Na drugie pytanie odpowiedź brzmi : tak, gdyż istniałaby możliwość blokady, ponieważ proces blokujący super_blok blokuje później inną rzecz,w tym przypadku możliwość sprawdzenia quoty.Można wyobrazić sobie funkcję,która przy zablokowanej możliwości sprawdzania quoty blokuje super_blok (ich przeplecione działanie mogłoby spowodować zawieszenie systemu,bo nikt nie mógłby operować na danych systemowych,zabezpieczonych blokowaniem tego super_bloku).

21. Ponieważ każdy błąd ma jakiś wpływ na funkcjonowanie systemu. Ostrzeżenie (`ext2_warning` wysyłane jest, gdy np.ustawiamy jakiś bit na wartość, którą on już ma. Nie ma to wpływu na działanie systemu,ale oznacza,że ćoś tu nie gra". Błąd (`ext2_error` ma miejsce np. gdy funkcja nie znajduje wolnego bloku podczas gdy licznik wolnych bloków ma wartość większą od zera.Oznacza to błąd w systemie, który może utrudniać korzystanie z zasobów, ale możliwy do naprawienia bez utraty stabilności systemu.

    Panika `ext2_panic` ma miejsce np. gdy przydziela się bloki zajęte przez dane systemowe - oznacza to niestabilność systemu.

22. W `struct inode` w polu `union{...}` dodajemy `struct scrooge_inode_info` zawierający dla danego i_node pozycje na dysku liczoną w bajtach(nie w sektorach).

    Funkcję `file_operations->read` można rozpatrywać w dwóch przypadkach: odczytywany inode (`file->dentry->d_inode`) ma numer 1 - oznacza to że czytamy z katalogu głównego - w zależnosci od `file->f_pos` podajemy po kolei w standardowy sposób strukturę katalogu zawierającą nazwy plików, rozmiary itp. Korzystamy z odwołania do danych zapisanych na urządzeniu w pierwszym obszarze. wpp. oznacza to, że czytamy plik, zaglądamy do `file->dentry->d_inode->scrooge_inode_info`, z zawartych tam informacji czytamy numer pierwszego bajtu dla danego pliku, dodajemy do niego `file->f_pos` i na podstawie wyniku wczytujemy poprzez sterownik odpowiedni sektor z obszaru 2 (numer_sektora = wynik/512) i pobieramy odpowiedni bajt (wynik mod 512). Przy przekroczeniu granicy sektora wczytujemy kolejny sektor. Jeżeli `f_pos` przekroczy zawarty w `file->dentry->d_inode->i_size` rozmiar pliku kończymy read i zwracamy liczbę przeczytanych bajtów.

Funkcja `inode_operations->lookup` sprawdza czy podana w parametrze `dentry` nazwa pliku występuje jako nazwa pliku w 1 obszarze, jeżeli tak, standardowo inicjuje strukturę inode zapisując w niej dodatkowo numer pierwszego bajtu w obszarze 2 dla danego pliku.

Funkcja `read_super` sprawdza czy podane jako parametr urządzenie jest typu SCROOGE i czy jest montowane w trybie read-only, jeżeli tak to inicjuje standardowo `struct super_block`, tworząc jednocześnie inode zawierający główny katalog zamontowanego urządzenia przechowywujący dane o plikach, z numerem ustawionym na 1.

23. - **proc**
  Funkcja `read_super` dla systemu proc jest dosyć prosta. W związku z tym że system proc nie korzysta z żadnego fizycznego urządzenia wystarczy wszystkie potrzebne parametry dla `struct super_block` (`s_blocksize,s_blocksizebits, s_magic, s_op`) ustawić na stałe. Następnie jest wykonywana próba zainicjowania struktury inode dla katalogu głównego. Na koniec inicjuje struktury `dentry` dla katalogu głównego. W przypadku niepowodzenia funkcja zwraca NULL.
    - **ext2**
  W systemie ext2 funkcja `read_super` ma dużo bardziej skomplikowane zadanie. Musi wykonać następujące kroki
      - pobranie z fizycznego urządzenia rozmiaru bloku
      - sprawdzenie poprawności podanych opcji montowania
      - pobranie z dysku struktury `super_block`
      - sprawdzenie czy typem partycji jest EXT2
      - sprawdzenie czy dane na dysku są zgodne z parametrami
      - próba odczytania deskryptorów grup i sprawdzenie czy spełniają odpowiednie warunki (np. czy liczba inode na grupę nie przekracza dozwolonego zakresu).
      - zainicjowanie struktury `dentry` dla katalogu głównego.

Jak widać VFS umożliwia współpracę z wieloma systemami plików różniącymi się:

- potrzebą istnienia fizycznego urządzenia
- sposobem wewnętrznego zapisu danych
- sposobem dostępu (np. poprzez urządzenie fizyczne - EXT2, sieć - NFS, bezpośrednio z parametrów systemowych - PROC)

Literatura

[Rochkind 93] Rochkind M. J.: *Programowanie w systemie Unix dla zaawansowanych.* Warszawa, WNT 1993.

[Bach 95] Bach M. J.: *Budowa systemu operacyjnego UNIX.* Warszawa, WNT 1995.