System plików

Tomasz Gajewski
Maciej Grzonkowski
Jacek Jankowski
Marta Karbowa
Eryk Makowski

7 stycznia 1999

1 System plików – wprowadzenie

Na samym wstępie chcę wyjaśnić jedną kwestię dotycząca nazewnictwa w tym rozdziale. Systemy plików takie jak *fat* czy *extfs* będę nazywał *formatem danych*, a terminu *system plików* będę używał w odniesieniu do całości kodu i algorytmów dotyczących plików.

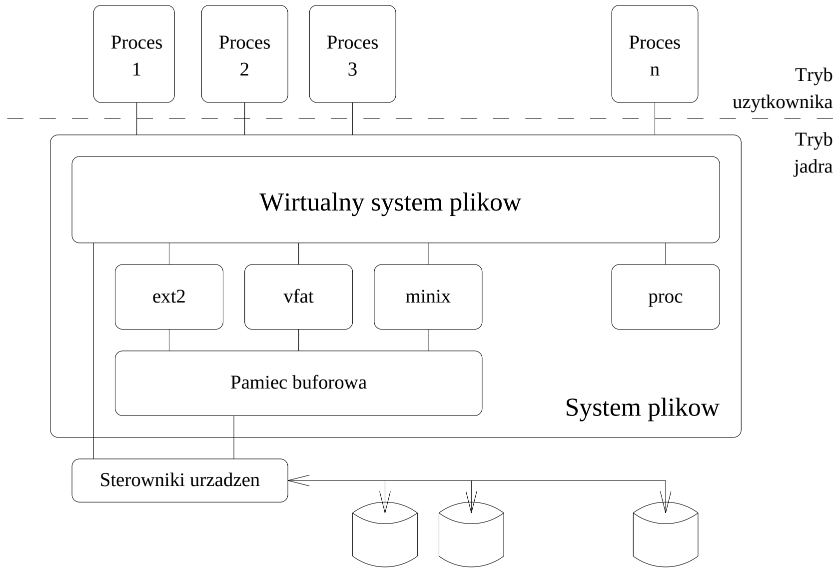
Czym jest system plików i do czego służy, każdy mniej więcej wie, więc nie będziemy się tym zajmować. Natomiast rzeczą, o której warto wspomnieć we wstępie są cechy, którymi powinien się cechować DOBRY system plików. Najważniejszymi z nich są:

- Wysoka wydajność – algorytmy.
- Możliwie wysoka odporność na uszkodzenia sprzętu – minimalizacja ewentualnych strat (w zasadzie dotyczy *formatu danych*).
- Jednolity interfejs dostępu dla procesów (niezależny od *formatu danych* na urządzeniach).
- Możliwość rozbudowywania go o nowe cechy (np. nowe *formaty danych*).

Właśnie głównie na powyżej wymienione cechy będziemy zwracać uwagę opisując rozwiązania przyjęte w Linuxie w implementacji systemu plików.

2 System plików od strony jądra – ogólny schemat

Najważniejszym pojęciem dotyczącym systemu plików Linuxa jest pojęcie *wirtualnego systemu plików.* Jest to jakby puszka, która ukrywa przed procesami format struktur przechowujących pliki. Dostęp do zawartości puszki odbywa się za pośrednictwem funkcji systemowych (np. *open, read, write, chmod*).



Rysunek 1: Ogólna struktura implementacji systemu plików w Linuxie

Linux wiele ze swej popularności zawdzięcza temu, że umie obsługiwać wiele różnych formatów plików (np. *ext2fs, minix, fat, vfat, hpfs*). Wirtualny system plików przy dostępie do danych w różnych formatach używa zunifikowanego, dość ogólnego interfejsu, dzięki czemu dodanie obsługi nowego formatu danych jest stosunkowo proste.

Taka koncepcja umożliwia także implementowanie obsługi formatów danych, które nie są zapisane na fizycznym urządzeniu lub są dostępne za pośrednictwem sieci. Przykładami są *proc, nfs, smbfs*.

Aby zapobiec wielokrotnemu odczytywaniu i zapisywaniu tych samych danych na dysku, zaimplementowano system buforowania. Korzystają z niego części systemu plików implementujące poszczególne formaty danych (ale tylko tych skojarzonych z fizycznym urządzeniem – nie dotyczy to np. formatu *proc*)

Ogólny schemat budowy systemu plików w Linuxie obrazuje *rysunek 1*.

3 Bloki i fragmenty w *ext2fs*

Pisząc o *ext2fs* należy wprowadzić dwa pojęcia *bloku*: *blok logiczny* i *blok fizyczny*.

- *blokiem fizycznym* nazywamy porcję danych o wielkości takiej jakimi przesyłane są dane do i z urządzenia.

- *blokiem logicznym* nazywamy bloki złożone z kilku położonych jeden po drugim *bloków fizycznych* (w Linuxie "kilka" jest potęgą dwójki).

Dostęp do konkretnej części pliku składa się z dwóch faz. Pierwszą jest znalezienie numerów bloków logicznych w których jest zapisany poszukiwany kawałek pliku (ta faza będzie dokładniej opisana w następnych rozdziałach). Jest ona wykonywana na wyższym poziomie niż warstwa druga, w czasie której jest dokonywane przeliczenie numerów bloków logicznych na numery bloków fizycznych.

Większość plików oczywiście nie ma rozmiaru równego wielokrotności wielkości bloku logicznego, więc ostatni blok każdego pliku nie jest zapełniony w całości. Dla każdego pliku marnowane byłoby średnio pół bloku. Może to stanowić dosyć dużą stratę, więc w *ext2fs* dodano jeszcze możliwość korzystania z *fragmentów*.

Fragment jest blokiem o wielkości wielokrotności rozmiaru bloku fizycznego, ale nie większego od rozmiaru bloku logicznego (standardowo w Linuxie rozmiar fragmentu jest równy rozmiarowi bloku logicznego, więc po prostu fragmenty nie są wykorzystywane). Przy użyciu fragmentów plik jest ciągiem bloków, po których występuje krótka sekwencja fragmentów. Kiedy plik się powiększa końcowe fragmenty są łączone w blok.

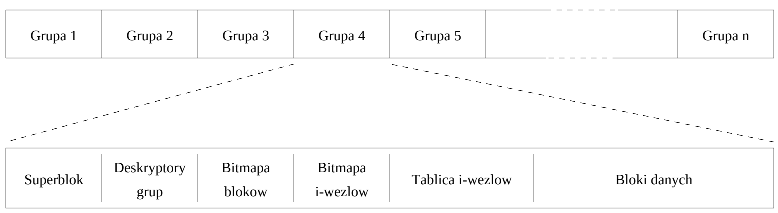
4 Logiczna struktura dysku w *ext2fs*

Ext2fs dzieli dysk na części zwane *grupami*. Każda grupa zawiera kopię krytycznych dla dysku danych. Oprócz tego podział dysku na grupy umożliwia zwiększenie efektywności poprzez użycie wydajnych algorytmów alokacji.

Każda grupa składa się z następujących części (*rysunek 2*):

- *Superblok* – zawiera kluczowe informacje o dysku i formacie danych (jak np. liczby i-węzłów i bloków na dysku, liczby wolnych i-węzłów i bloków). Superbloki we wszystkich grupach mają tę samą zawartość, aby zmniejszyć prawdopodobieństwo zniszczenia danych w nich zapisanych.

- *Deskryptory grup* – podobnie jak w przypadku superbloków ich zawartość jest skopiowana do wszystkich grup. Zawierają one informacje o miejscach, w których zaczynają sie pozostałe części grup i o liczbie wolnych i-węzłów i bloków w grupach.



Rysunek 2: Logiczna struktura dysku w *ext2fs*

- *Bitmapa bloków* – tablica bitów zawierająca informacje o zajętości bloków należących do grupy (ta i następne części nie są powielane wielokrotnie). Ważne jest to, że każdy bit odpowiada najmniejszej jednostce na dysku, która może być zapisana bądź nie czyli, fragmentowi. W przypadku aktualnej implementacji *ext2fs* w Linuxie rozmiary fragmentów i bloków są równe.
- *Bitmapa i-węzłów* – analogiczna jak powyższa bitmapa bloków tylko dotycząca i-węzłów w grupie.
- *Tablica i-węzłów* – obszar na dysku zawierający i-węzły z danej grupy.
- *Bloki danych* – nazwa mówi sama za siebie.

5 Struktura zwykłego pliku

Jednym z zadań realizowanych przez system Linux jest udostępnianie użytkownikowi abstrakcji pliku. W codziennej pracy najczęściej korzysta się z plików przechowywanych na dysku twardym. Plik stanowi pewną spójną całość – uporządkowany ciąg danych. Użytkownik może czytać dane sekwencyjnie, bądź w dowolny inny sposób, posługując się *logicznymi adresami* danych.

Adres logiczny – widziany przez użytkownika nie stanowi bezpośrednio *adresu fizycznego* – określającego położenie danej na dysku. Opis "rozłożenia" pliku na bloki dyskowe jest zapisany w i-węźle, oraz w przypadku dłuższych plików – w strukturach pomocniczych zwanych blokami pośrednimi.

Odczytanie określonej danej z pliku wymaga identyfikacji pliku (numer i-węzła) oraz znajomości adresu danej względem początku pliku. Danymi wejściowymi są zatem numer i-węzła pliku i logiczny adres bloku. Logiczny adres bloku oblicza się dzieląc logiczny adres bajtu przez rozmiar bloku. Na wyjściu chcemy otrzymać fizyczny adres bloku dyskowego. Algorytm zaimplementowany w systemie Linux działa w następujący sposób:

1. Sprawdzenie poprawności danych wejściowych
2. Jeśli blok jest bezpośredni[1], to zwracamy adres bloku pobrany z i-węzła
3. Jeśli blok jest adresowany pośrednio[2] najpierw pobierany jest z i-węzła

[1] Blok bezpośredni to taki blok, którego adres dyskowy jest zapisany w i-węźle pliku. Jest to kilka pierwszych bloków pliku. Ilość bloków bezpośrednich określa stała `EXT2_NDIR_BLOCKS` – standardowo równa 12. Rozmiar bloku to stała `BLOCK_SIZE` – standardowo równa 1024. Zatem wielkość przestrzeni dyskowej dostępna w adresowaniu bezpośrednim w obrębie jednego pliku to 12KB.

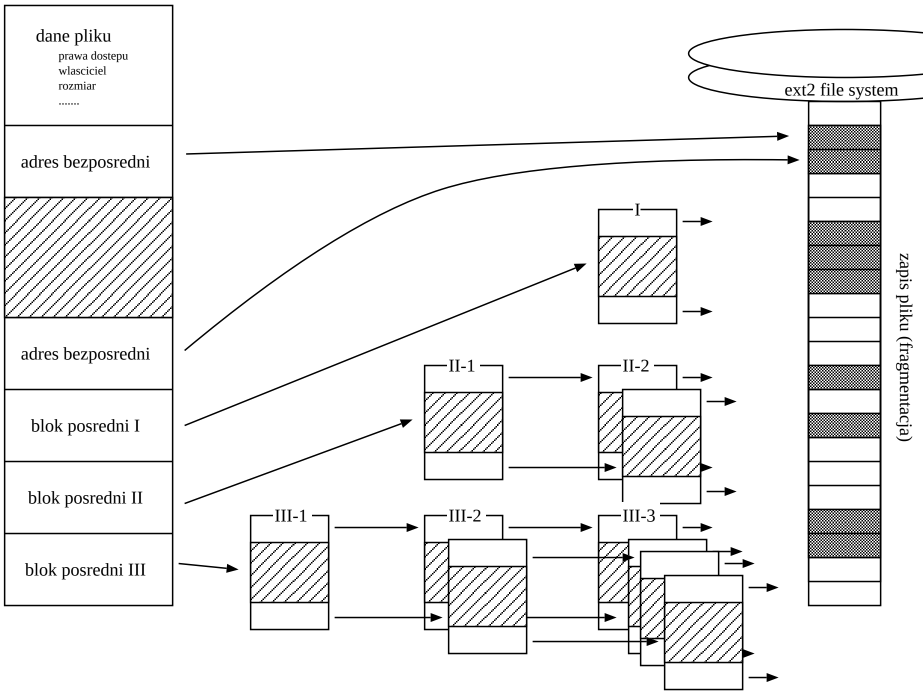
[2] Adresowanie pośrednie w tym przypadku polega na tym, że zamiast adresu bloku dyskowego należącego do pliku przechowywany jest adres tablicy zawierającej adresy bloków pliku. W systemie ext2 stososowane są cztery poziomy pośredniości – od bezpośredniego, do potrójnie pośredniego.

adres dyskowy bloku zawierającego adresy bloków pliku[3] (*oznaczony na rysunku symbolem I*), następnie jest on wczytywany do pamięci, z niego pobiera się adres bloku na dysku.

4. Jeśli blok jest adresowany podwójnie pośrednio wczytywany jest adres bloku zawierającego adresy pośrednie pierwszego stopnia (*oznaczony na rysunku symbolem II-1*), Następnie wykonywane są te same czynności co w punkcie 3 na danych zawartych we wczytanym bloku.

5. Jeśli blok jest adresowany w sposób podwójnie pośredni, to najpierw jest wczytywany blok podwójnie pośredni (*oznaczony symbolem III-1*, następnie na danych tego bloku wykonuje się czynności analogiczne jak w punkcie 4.

Sposób dostępu do bloków obrazuje *rysunek 3*



Rysunek 3: Odwzorowanie adresów logicznych pliku na fizyczne

[3]Na każdym poziomie pośredniości tablica zawierająca adresy bloków wypełnia cały blok. Zawiera zatem 32 adresów (`BLOCK_SIZE/sizeof(__u32)`). Istnieje zatem możliwość zaadresowania ponad 32GB. Faktycznie rozmiar pliku ograniczony jest przez rozmiar pola w i-węźle pliku przechowującego rozmiar pliku i wynosi 4GB.

6 Rzut oka na podręczną pamięć buforową

System plików to interfejs pomiędzy aplikacjami a danymi składowanymi na nośnikach (dyskach). Z punktu widzenia aplikacji dane te składowane są w postaci plików, czyli ciągów bajtów. W rzeczywistości dyski są urządzeniami blokowymi - podstawową jednostką operacji dyskowej jest blok, czyli ciąg bajtów o ustalonym dla danego urządzenia rozmiarze. Ta właściwość nadała kształt systemowi podręcznej pamięci buforowej.

Pamięć ta składa się z puli buforów - obiektów odwzorowujących w pamięć główną zawartość bloków dyskowych. Odwzorowanie takie pozwala na znaczne przyspieszenie operacji, gdyż wykonywane są one na danych znajdujących się w pamięci głównej - znacznie szybszej niż pamięć dyskowa. Można by na tym zakończyć ten temat, gdyby w pamięci głównej mieściły się kopie wszystkich bloków z urządzenia. Tak oczywiście nie jest, buforować można tylko pewną ilość bloków, korzystając z określonej puli buforów - powstaje zatem całkiem ciekawe zagadnienie administrowania tą pulą (decydowania, które bloki mają być odwzorowane w pamięci), którym zajmiemy się w następnych rozdziałach. Głównym założeniem dotyczącym opisywanej pamięci podręcznej jest to, iż dany blok dyskowy może być odwzorowany w co najwyżej jeden bufor w danej chwili (co nie oznacza, że w następnej chwili czasu nie może być on odwzorowany w zupełnie inny bufor).

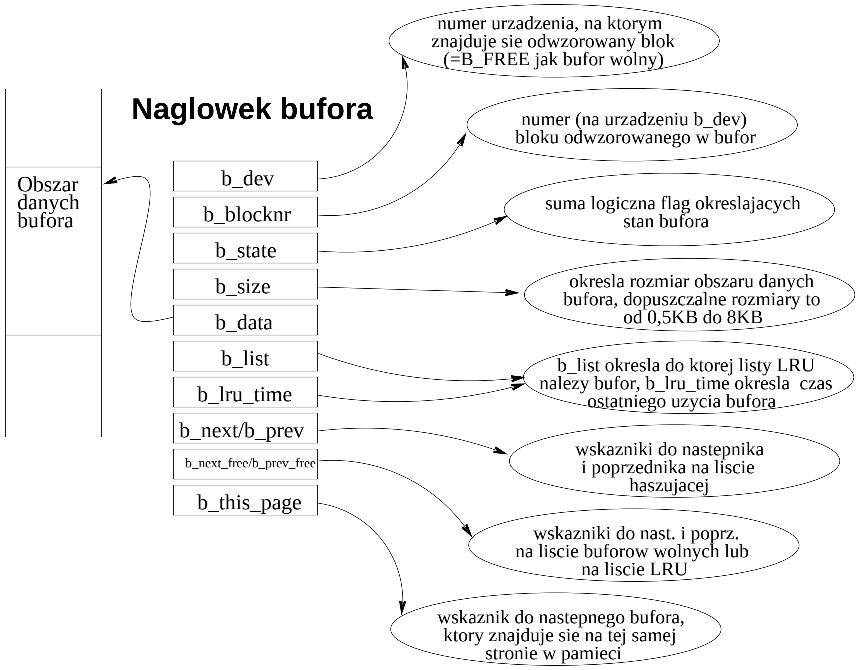
7 Wady i zalety buforowania

Zacznijmy od zalet. Zaliczyć do nich należy to, że buforowanie przyczynia się w znacznym stopniu do ujednolicenia różnych systemów plików. Mechanizm buforowania oddziela nas od warstwy zależnej od systemu plików, dostarczając niezależny od tej warstwy interfejs, z którego korzystają systemowe funkcje wyższego rzędu. Kolejną zaletą pamięci buforowej jest znaczne przyspieszenie operacji dyskowych, na co składa się między innymi czytanie bloków z wyprzedzeniem oraz eliminowanie niepotrzebnych odwołań do dysku. Buforowanie pozwala rownież ukryć przed aplikacjami blokowy charakter operacji dyskowych - pozwala na łatwe odczytywanie i zapisywanie nawet pojedynczych bajtów bez straty wydajności.

Wady ... brak o ile mamy *UPS* - zlecenie operacji zapisu nie oznacza fizycznego zapisania danych na dysk, istotne informacje mogą utknąć w pamięci buforowej nawet na parę sekund.

8 Podstawowa jednostka - bufor

Bufor jest obiektem odwzorowującym w pamięć główną pojedynczy blok dyskowy. Składa się on z nagłówka bufora (rys. 4) oraz obszaru pamięci o rozmiarze równym rozmiarowi bloku dyskowego służącemu do przechowywania danych z bloku. Ponieważ rozmiar bloku dyskowego może być różny na różnych urządzeniach, zatem i bufory wchodzące w skład podręcznej pamięci buforowej muszą



Rysunek 4: Struktura nagłówka bufora (wybrane pola)

być różnych rozmiarów. Nagłówek bufora jest strukturą zawierającą oprócz wskaźnika do obszaru danych (pole *b_data*) szereg pól określających jego stan. Do tej grupy należą pola określające numer bloku odwzorowanego aktualnie w ten bufor (pole *b_blocknr*), numer urządzenia na którym ten blok się znajduje (pole *b_dev*, to pole ma wartość *B_FREE* jeżeli bufor jest aktualnie wolny), rozmiar bufora (pole *b_size*) oraz pole *b_state* będące logiczną sumą flag, z których niektóre to: *BH_Uptodate* (bufor zawiera dane z dysku), *BH_Dirty* (zawartość bufora trzeba zapisać na dysk, ponieważ została zmodyfikowana), *BH_Lock* (bufor zablokowany - trwa na nim operacja wejścia/wyjścia). Do struktury nagłówka bufora należą ponadto pola pozwalające na łączenie buforów w różne struktury danych (pozwalające na wydajne administrowanie pulą buforów). Pola te zostaną opisane w rozdziale następnym przy okazji omawiania używanych struktur.

9 Struktury danych

Buforowanie bloków ma przyspieszyć wykonywanie operacji dyskowych. Oczywistym jest, że czym większa jest pula buforów, tym większe prawdopodobieństwo, że odwołując się do bloku dyskowego odnajdziemy go w pamięci - co znacznie przyspieszy wykonanie operacji. Chcielibyśmy zatem, aby ilość pamięci przeznaczona na bufory dostosowana była do chwilowej ilości wolnej pamięci. Chcielibyśmy ponadto, aby bloki, do których są częste odwołania, z dużym prowdopodobieństwem znajdowały się w pamięci. Aby system pamięci buforowej spełniał powyższe wymagania, utrzymywane są rozmaite struktury buforów:

- tablica haszująca - utrzymywana jest tablica kolejek liniowych dwukierunkowych. Tablica ma rozmiar zależny od wielkości pamięci. Każdy bufor, który zawiera dane z dysku znajduje się w jednej z tych kolejek. Funkcja haszująca wyznaczająca listę dla danego bufora jest bardzo prosta - numer listy (indeks w tablicy) to numer bloku (powstały z numeru urządzenia oraz numeru bloku na tym urządzeniu) modulo rozmiar tablicy haszującej. Aby otrzymać bufor zawierający dane z konkretnego bloku dyskowego, lub dowiedzieć się, że takiego bufora nie ma, wystarczy przejrzeć listę o numerze wyznaczonym przez funkcje mieszającą. Lista będzie tym krótsza im większy będzie rozmiar tablicy, lecz większa tablica zajmuje na stałe więcej pamięci - dlatego właśnie rozmiar tablicy mieszającej uzależniony jest od wielkości pamięci głównej. Bufory łączone są w opisywane listy za pomocą pól *b_next* i *b_prev* nagłówków buforów.

- lista buforów wolnych - dla każdego z dopuszczalnych rozmiarów bufora trzymana jest dwukierunkowa, cykliczna lista buforów wolnych o tym rozmiarze. Dzięki takiej organizacji łatwo można sięgnąć po wolny bufor dla bloku o danej wielkości. Do stworzenia list wykorzystuje się pola *b_next_free* i *b_prev_free* nagłówków buforów.

- listy *LRU* - mając do dyspozycji ograniczoną pulę buforów często trzeba podejmować decyzję, które bufory należy zwolnić aby zapisać w nich dane z aktualnie potrzebnego bloku. W Linuxie wykorzystuje się w tym celu strategię "Najdłużej nieużywany" (*Last Recently Used*). Każdy bufor w swoim nagłówku ma zapisaną informację o czasie ostatniego użycia (pole *b_lru_time*). Aby szybko wybierać bufory najdłużej nieużywane utrzymuje się listy *LRU*. Bufory podzielono na kategorie (patrz *Tablica 1*), dla każdej z nich jest osobna lista *LRU*. Listy te są dwukierunkowe, cykliczne i są zbudowane w oparciu o pola *b_next_free* i *b_prev_free* nagłówka bufora (wykorzystuje się te same pola co w liście buforów wolnych, ponieważ na listach *LRU* znajdują się tylko bufory zajęte). Pierwszy bufor na liście to najdłużej nieużywany w danej kategorii. Pole nagłówka *b_list* określa (o ile bufor jest zajęty), do której kategorii (listy) bufor należy.

Tablica 1: Kategorie buforów

BUF_CLEAN	bufory nie będące w poniższych kategoriach
BUF_LOCKED	bufory, na których trwa operacja wejścia/wyjścia
BUF_LOCKED1	j.w, ale bufory zawierające i-węzły lub super bloki
BUF_DIRTY	bufory przeznaczone do zapisu (zmodyfikowane)

- listy buforów na stronie - jak już wspomniałem, na każdy bufor składa się nagłówek oraz obszar danych. Obie te części znajdują się w pamięci, która podlega stronicowaniu. Pamięć na nagłówki jest alokowana i zwalniana po sztuce. Zupełnie inaczej jest w przypadku obszarów danych. Pamięć na te obszary alokowana jest całymi stronami - odpowiednia funkcja pobiera stronę i kolejne jej fragmenty przydziela kolejnym buforom. Chcąc również zwalniać tę pamięć stronami potrzebujemy struktury dającej łatwy dostęp do wszystkich buforów, których obszar danych znajduje się na tej samej stronie pamięci. Taką strukturę tworzy pole *b_this_page* nagłówka bufora, które jest wskaźnikiem do kolejnego bufora na tej samej stronie.

- lista wolnych nagłówków - obszar danych bufora ma określoną wielkość dostosowaną do jednego z dopuszczalnych rozmiarów bloku dyskowego. Oczywiście nie da się go użyć do buforowania bloków innego rozmiaru (zapisywanie bloków mniejszych jest nieefektywne), dlatego często zwalnia się stronę buforów jednego rozmiaru, aby zająć ją buforami innego rozmiaru. Z nagłówkami sprawa jest prostsza - pasują do bloków każdego rozmiaru, dlatego rzadko zwalnia się pamięć, którą zajmują. Konktretnie - kiedy likwidowany jest bufor, pamięć obszaru danych jest zwalniana, a nagłówek trafia do listy wolnych nagłówków (chyba że lista ta jest zbyt długa, wtedy pamięć, którą zajmuje jest również zwalniana). Funkcja tworząca nowe bufory nagłówek pobiera z tej właśnie listy (chyba, że lista jest pusta; wtedy tworzy jeden nowy nagłówek).

10 Algorytmy obsługi pamięci buforowej

Funkcje odczytu i zapisu bloku dyskowego przy zastosowaniu pamięci buforowej mają nastepującą postać:

- Odczyt bloku dyskowego - sprawdź, czy jest bufor zawierający dane z żądanego bloku - jak jest, to przekaż go jako wynik wpp., odczytaj blok do wolnego bufora i zwróć go

- Zapis bloku dyskowego - sprawdź, czy jest bufor zawierający dane z żądanego bloku (jak nie ma, to odczytaj blok do wolnego bufora), zmodyfikuj bufor i zaznacz go do zapisu

Widać, że dla realizacji tych funkcji kluczowymi algorytmami są:

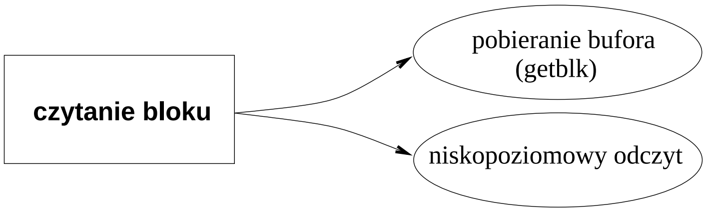
- algorytm szybkiego znajdowania bufora zawierającego dane z danego bloku lub stwierdzania, że takiego bufora w pamięci nie ma,

- algorytm znajdowania wolnego bufora (ciekawy, kiedy lista wolnych jest pusta),

- algorytmy administrowania rozmiarem pamięci buforowej (alokowania i zwalniania pamięci na obszary danych buforów i ich nagłówki).

W dalszej części omówię funkcje realizujące wymienione algorytmy.

10.1 Algorytm czytania bloku (*bread*)



Rysunek 5: Funkcje wywoływane przez *bread* (*breada*)

- pobieramy bufor przy użyciu opisanego niżej algorytmu (*getblk*),
- jeżeli otrzymany bufor zawiera żądany blok (jest *uptodate*), to oddajemy go jako wynik,
- wpp. inicjujemy niskopoziomowy odczyt bloku do otrzymanego pustego bufora,
- czekamy na zakończenie operacji, sprawdzamy, czy się powiodło (czy bufor zawiera blok dyskowy), zwracamy wypełniony bufor lub informację o błędzie.

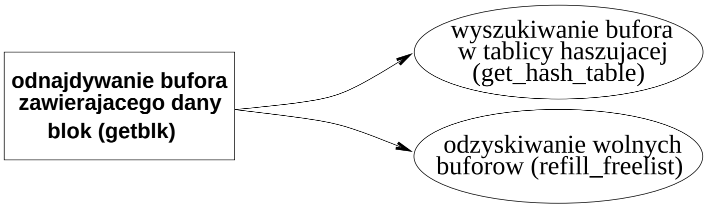
10.2 Algorytm czytania bloków z wyprzedzeniem (*breada*)

Celem stawianym funkcji breada jest odczytanie żądanego bloku do bufora oraz zainicjowanie czytania następujacych po nim bloków. Taka operacja pozwoli na przyspieszenie realizacji żądań dostępu do bloków usytuowanych za blokiem odczytywanym - czyli bloków, których z dużym prawdopodobieństwem dotyczyć będą kolejne żądania. Oto schemat algorytmu:

- pobieramy bufor przy użyciu opisanego niżej algorytmu (*getblk*),
- jeżeli otrzymany bufor zawiera żądany blok to oddajemy go jako wynik,
- wpp. inicjujemy niskopoziomowy odczyt bloku do otrzymanego, pustego bufora,
- spośród bloków - znajdujących się na dysku za odczytywanym - wybieramy te, które nie są odwzorowane w pamięć buforową (bierzemy pod uwagę tylko określoną ilość bloków),
- inicjujemy niskopoziomowy odczyt wyznaczonych bloków,
- czekamy na zakończenie odczytu żądanego bloku; pozostałe bloki czytane są asynchronicznie,
- sprawdzamy, czy się powiodło i oddajemy w wyniku wypełniony bufor lub informację o błędzie.

10.3 Algorytm znajdowania bufora zawierającego określony blok (*getblk*)

Funkcja ma wyszukać w tablicy haszującej bufor zawierający dany blok. W zależności od powodzenia tej operacji funkcja ma zwrócić bądź znaleziony bufor, bądź bufor wolny właściwego rozmiaru. Bufor jest wstawiany (przesuwany) na koniec listy *LRU*.



Rysunek 6: Funkcje wywoływane przez *getblk*

- wyszukaj w tablicy haszującej bufor zawierający dany blok (jeżeli bufor jest zablokowany - trwa na nim operacja wejścia/wyjścia, to zaśnij w związanej z nim kolejce),
- jeżeli bufor został odnaleziony, to oddaj go jako wynik funkcji,
- wpp. jeżeli na liście wolnych właściwego rozmiaru jest jakiś bufor, to oddaj go jako wynik,
- wpp. wywołaj funkcje odzyskującą wolne bufory (*refill_freelist*),
- powtórz algorytm (w czasie odzyskiwania mogliśmy usnąć, więc żądany blok mógł się pojawić w pamięci - dlatego od nowa szukamy go w tablicy haszującej).

10.4 Algorytmy administrowania rozmiarem pamięci buforowej

- tworzenie strony buforów (*grow_buffers*),
- tworzenie nowego nagłówka bufora (*get_more_buffer_heads*),
- zwalnianie strony buforów (*try_to_free_buffer*),
- zwalnianie nagłówka bufora (*put_unused_buffer_head*);

Bufory tworzone są całymi stronami. Odpowiednia funkcja pobiera wolną stronę pamięci, dopóki jest miejsce na tej stronie pobiera wolny nagłówek bufora (*get_unused_buffer_head*) i definiuje jego obszar danych jako kolejny fragment zajętej strony. Funkcja dająca wolny nagłówek w razie potrzeby (jeżeli lista wolnych nagłówków jest pusta) najpierw tworzy go alokując potrzebną pamięć i wstawiając go do listy nieużywanych nagłówków (*get_more_buffer_heads*). Aby pamięć przeznaczona na bufory nie rozrastała się zbytnio, stosuje się następującą technikę: podczas odzyskiwania wolnych buforów (*refill_freelist*) danego rozmiaru próbujemy zwalniać pamięć zajmowaną przez nieużywane bufory innych rozmiarów. W tym celu wywołujemy funkcje *try_to_free_buffer*, która działa w następujący sposób:

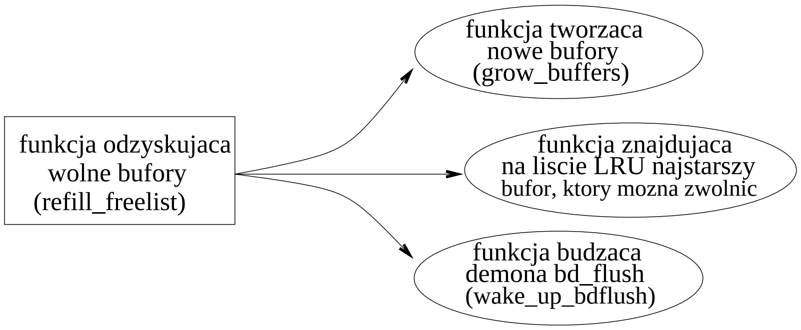
- sprawdzamy, czy wszystkie bufory na stronie są nieużywane,
- jeżeli tak, to wyjmujemy je ze wszystkich struktur, nagłówki odkładamy (funkcja *put_unused_buffer_head*) i zwalniamy stronę pamięci.

Wspomniana funkcja *put_unused_buffer_head* wstawia nagłówek do listy wolnych, chyba że lista ta jest zbyt długa - wtedy zwalniamy pamięć zajmowaną przez nagłówek (oszczędność pamięci).

10.5 Algorytm odzyskiwania wolnych buforów (*refill_freelist*)

Gdy przychodzi żądanie odczytu bloku dyskowego, wyszukiwany jest w pamięci bufor zawierający dane z tego bloku; jeżeli taki bufor nie zostanie odnaleziony, pobierany jest bufor wolny. W tym punkcie zajmiemy się sytuacją, kiedy pobranie wolnego bufora jest nietrywialne (lista buforów wolnych jest pusta).

W opisanej sytuacji wykorzystuje się funkcję *refill_freelist*, której zadaniem jest dostarczenie wolnych buforów o żądanej wielkości - niejako ubocznym działaniem funkcji jest zwalnianie pamięci zajmowanej przez nieużywane bufory innych rozmiarów niż żądany. Lektura źródeł funkcji dostarcza refleksji, że zastosowany algorytm jest wynikiem długotrwałych doświadczeń mających na celu zminimalizowanie kosztu odzyskiwania buforów. Pierw próbujemy najtańszego rozwiązania, potem stopniowo - jeżeli dotychczasowe działania nie przyniosły skutku - próbujemy coraz kosztowniejszych technik. Istotne jest również to, że dla zwiększenia efektywności przy każdym wywołaniu funkcji próbujemy



Rysunek 7: Funkcje wywoływane przez *refill_free_list*

odzyskać pewną ilość buforów (zmienna *needed*) o żądanym rozmiarze, mimo iż chwilowo potrzebny jest tylko jeden. Oto schemat algorytmu:

- jeżeli jest dużo wolnej pamięci (dwa razy więcej niż minimalna ilość wolnych stron), to próbujemy stworzyć nowe bufory potrzebnego rozmiaru (*grow_buffers*),

- jeżeli uzyskaliśmy mniej buforów niż chcieliśmy (potrzebujemy *needed* buforów), to próbujemy odzyskiwać zajęte bufory zgodnie ze strategią *Last Recently Used* korzystając z opisanych wcześniej list *LRU*. Dla każdej listy *LRU* (patrz pkt 4) znajdujemy kandydata do zwolnienia (robi to funkcja biorąca pierwszy bufor z danej listy - najdłużej nieużywany - który się do zwolnienia nadaje, czyli jest właściwego rozmiaru, żaden proces na niego nie czeka, nie trwa na nim operacja wejścia/wyjścia, ani nie jest zmodyfikowany - dodatkowo funkcja ta dla każdego przeglądanego elementu, który nie jest żądanego rozmiaru wywołuje opisaną w poprzednim punkcie funkcje *try_to_free_buffer*, próbując zredukować obszar pamięci zajmowany przez bufory - zwiększając tym samym szanse na późniejsze zajęcie tego obszaru buforami żądanego rozmiaru). Spośród kandydatów wybieramy bufor najdłużej nieużywany, jest on zwalniany i przenoszony do listy wolnych. Tę technikę stosuje się aż do otrzymania satysfakcjonującej ilości wolnych buforów żądanego rozmiaru (co kończy działanie funkcji) lub do stwierdzenia braku kandydatów. W drugim przypadku przechodzimy dalej,

- próbujemy stworzyć nowe bufory potrzebnego rozmiaru stawiając łagodniejszy warunek co do ilości wolnej pamięci, jeśli i to nie da wystarczającej ilości buforów, to przechodzimy do następnego punktu,

- jeżeli zmodyfikowanych buforów jest dużo to budzimy demona *bd_flush* (opisany jest dokładniej w następnym rozdziale), który zapisuje na dysk zmodyfikowane bufory. Otrzymujemy w ten sposób potencjalnie więcej buforów nadających się do zwolnienia, zatem wracamy dwa punkty wyżej - do odzyskiwania buforów na podstawie list *LRU* (dodatkowo zmniejszamy swoje zapotrzebowanie na ilość wolnych buforów). Jeżeli zmodyfikowanych buforów było zbyt mało aby budzić demona, przechodzimy do kolejnego punktu,

  wiem, że dużo to pojęcie względne - tu dużo, to procentowo więcej niż pewna stała

- jeżeli tylko mamy jakieś wolne bufory w żądanym rozmiarze, to kończymy działanie,

  cóż za desperacja

- kolejny raz próbujemy *grow_buffers*, oraz o ile tylko są jakieś zmodyfikowane bufory, budzimy *bd_flush'a*, jeżeli znowu nic to *grow_buffers* - tym razem z większym priorytetem,

- jeszcze bardziej zmniejszamy ambicje (zapotrzebowanie na bufory) i zaczynamy zabawę od początku (konkretnie od punktu drugiego).

11 Demon *bd_flush*

Jak wspomniałem we wstępie do poprzedniego rozdziału, zapis bloku sprowadza się do znalezienia w pamięci bufora zawierającego dane z tego bloku (w razie konieczności należy zająć wolny blok i wypełnić go danymi z dysku) oraz zmodyfikowania zawartości bufora i zaznaczenia tego faktu w nagłówku bufora. Potrzebny jest zatem mechanizm okresowego zapisywania zawartości takich buforów na dysk. W *Linuxie* taki mechanizm realizuje demon (wątek jądra) *bd_flush*. Działa on w nieskończonej pętli, za każdym jej obrotem wykonując następujący algorytm:

- zapisuj na dysk bufory zmodyfikowane, zliczając ile już zapisałeś,
- jeżeli ilość zapisanych buforów osiągnęła pewną stałą (parametr działania *bd_flush'a*), to jeżeli w systemie jest dużo brudnych (zmodyfikowanych) buforów, to zacznij od początku, wpp zaśnij.

Demon *bd_flush* jest budzony okresowo (standardowo co 5 sekund) oraz wtedy, kiedy ilość buforów zmodyfikowanych przekroczy pewien limit (jest to parametr pracy demona). Można go także obudzić (tak jak np. w funkcji *refill_freelist*) używając funkcji *wake_up_bdflush*.

12 Inicjacja podręcznej pamięci buforowej

Inicjacją pamięci buforowej zajmuje się funkcja *buffer_init* wywoływana w czasie startu systemu *Linux*. Funkcja ta podejmuje nastepujące działania:

- alokuje pamięć na tablicę mieszającą,
- tworzy bufory (*grow_buffers*) o rozmiarze równym rozmiarowi bloku na urządzeniu, z którego system jest uruchamiany.

13 I-węzły

Z każdym plikiem w Linuxie skojarzony jest *i-węzeł*. Jest to struktura zawierająca niezbędne dla systemu dane o pliku. Częściowo temat i-węzłów został już poruszony w rozdziale dotyczącym struktury pliku. W tym przedstawiamy tabelę wymieniającą główne pola struktury i-węzła (*Tablica 2*), oraz algorytmy związane z alokacją i zwalnianiem i-węzłów w pamięci (i-węzły nie są modyfikowane bezpośrednio na dysku) i na dysku.

Aby zapewnić szybki dostęp do i-węzłów są one umieszczone w pamięci w tablicy haszującej zawierającej 512 pozycji[4]. Standardowo w pamięci przechowywanych jest maksymalnie 3072 i-węzłów[5].

[4]stała NR_IHASH
[5]stała NR_INODE

Tablica 2: Najważniejsze pola struktury i-węzła

typ pliku
uid właściciela pliku
rozmiar
licznik dowiązań
czasy ostatniego dostępu i modyfikacji
prawa dostępu

Każdy element w tablicy haszującej i-węzłów zawiera informację o tym, czy dana kolejka w tablicy jest zablokowana, czy nie. Uniemożliwia to skasowanie pliku procesowi, jeśli jest możliwe, że inny proces czeka na pobranie i-węzła z dysku (wtedy proces dostałby i-węzeł, którego nie ma). Ważne jest to, że blokada jest zakładana tylko w przypadku, gdy proces może się zawiesić w oczekiwaniu na urządzenie dyskowe. Gdy nie może się zawiesić na kolejce, to może modyfikować wszystkie struktury bez obaw.

13.1 Algorytm *iget* – pobranie i-węzła z pamięci

Algorytm *iget* odpowiada za dostarczenie procesowi kopii i-węzła i w gruncie rzeczy jest bardzo prosty. Najpierw jest sprawdzane czy i-węzeł jest w tablicy haszującej. Jeśli tak to po aktualizacji liczników otwarcia pliku i wolnych i-węzłów zwróć go. W przeciwnym przypadku pobieramy wolny i-węzeł (*get_empty_inode*) i wczytujemy do niego dane z dysku (*read_inode*).

Implementacja się komplikuje przez to, że trzeba sprawdzać wiele warunków, aby utrzymać integralność systemu plików (szczególnie gdy wykonywana operacja może spowodować zawieszenie się w oczekiwaniu na urządzenie).

13.2 Algorytm *iput* – zwolnienie i-węzła w pamięci

Algorytm zwalniania kopii i-węzła z pamięci tak jak i poprzedni jest stosunkowo prosty i również jest obwarowany zabezpieczeniami nie dopuszczającymi do kłopotów.

Najpierw jest zmniejszana liczba otwarć pliku i jeśli nadal jest ona większa od zera to algorytm kończy się. W przeciwnym przypadku sprawdza następujące przypadki i odpowiednio na nie reaguje:

- i-węzeł był typu *pipe* – zwolnij związaną z nim stronę pamięci.
- i-węzeł w buforze miał ustawiony atrybut *dirty* – zapisz go na dysk.
- i-węzeł podlega ograniczeniom *quoty* – odpowiednio zaktualizować.

13.3 Algorytm *ialloc* – alokowanie i-węzła na dysku

Algorytm alokowania i-węzła na dysku działa dwojako w zależności od tego czy nowy i-węzeł będzie katalogiem, czy też nie.

Pierwszym krokiem jest znalezienie grupy, w której zaalokujemy i-węzeł. Ze względów optymalizacyjnych jest to bardzo ważny krok.

- W przypadku katalogu wybieramy spośród grup, które mają liczbę wolnych i-węzłów większą od średniej, tę, w której jest najwięcej wolnych bloków danych.
- W pozostałych przypadkach algorytm jest trochę bardziej złożony i składa się z kilku faz.
  - próbuje zaalokować i-węzeł w tej samej grupie, w której znajduje się blok nadrzędnego katalogu; jeśli się nie udaje to
  - szuka grupy zawierającej wolny i-węzeł w każdym kroku przeskakując k grup, gdzie k jest kolejną potęgą 2 (algorytm *quadratic hash*); jeśli się nie udaje to
  - przeszukuje grupy po kolei (i jest to ostateczna metoda).

Pozostała część algorytmu nie zależy od przeznaczenia i-węzła. Najpierw jest znajdowany liniowo (korzystając z bitmapy i-węzłów) pierwszy wolny i-węzeł, a potem są inicjowane odpowiednie struktury (*struct inode*) oraz modyfikowane wartości w superbloku i deskryptorach grup.

Warto zauważyć, iż w związku z tym, że algorytm odczytuje i modyfikuje dane dotyczące aktualnego stanu systemu plików, wymaga wyłącznego dostępu do tych danych (zapewnia to funkcja *lock_super*).

13.4 Algorytm *ifree* – zwalnianie i-węzła na dysku

Algorytm nie jest zbyt wyrafinowany. Po sprawdzeniu, że można zwolnić i-węzeł i zapewnieniu sobie wyłączności na operacje na superbloku (*lock_super*) modyfikuje odpowiednio struktury systemu plików (wyzerowanie odpowiedniego bitu w bitmapie i-węzłów, zwiększenie liczby wolnych i-węzłów, ew. zmniejszenie liczby zajętych katalogów, wyczyszczenie i-węzła).

14 Katalogi w systemie Linux

14.1 Katalogi w systemie `ext2`

W systemie `ext2` katalogi są zapisywane na dysku jak zwykłe pliki. Katalog różni się od zwykłego pliku tym, że w polu typu pliku jest oznaczony jako katalog. Zawartość pliku – katalogu to lista rekordów. Strukturę rekordu przedstawia *Tablica 3*.

Tablica 3: Pozycja katalogowa

numer i-węzła
długość pozycji katalogowej
długość nazwy
nazwa

Długość nazwy jest ograniczona do 255 znaków[6]. Dłuższe nazwy mogą być obcinane, albo traktowane jako błędne[7].

W katalogu znajdują się pozycje o nazwach '..' i '.' zawierające dowiązania odpowiednio do katalogu bieżącego i macierzystego. W katalogu będącym korzeniem te dwie pozycje mają numer i-węzła bieżącego katalogu.

Usuwanie pozycji z katalogu polega na nadaniu i-węzłowi numeru 0. Taka pozycja może zostać powtórnie wykorzystana. Podczas tworzenia nowego pliku w danym katalogu nowa pozycja jest dodawana na pierwszej pozycji o numerze i-węzła równym 0 i zawierającej wystarczająco dużo miejsca. Jeśli taki rekord nie zostanie znaleziony, nowa pozycja jest dodawana na końcu.

Prawa dostępu dla katalogów mają nieco inne znaczenie niż w przypadku zwykłych plików. Zostały one opisane w *Tablicy 4*.

Tablica 4: Prawa dostępu dla katalogów

czytanie	możliwość przeglądania zawartości katalogu
pisanie	tworzenie i usuwanie plików
wykonywanie	możliwość przeszukiwania katalogu (jeśli znamy nazwę pliku, możemy go odczytać nie posiadając prawa czytania do katalogu – wystarczy prawo czytania pliku)

Należy tu podkreślić, że procesy mogą jedynie odczytywać katalogi, pisać do katalogu może tylko jądro systemu.

14.2 Pamięć podręczna katalogów

Aby przyśpieszyć dostęp do pozycji katalogowych, część z nich przechowywana jest w pamięci RAM. Dotyczy to takich pozycji, których długość nazwy nie przekracza 15 znaków[8]. Mechanizm ten jest częścią wirtualnego systemu plików, zatem działa dla wszystkich używanych przez Linux systemów plików.

Pozycje pamiętane są w specjalnej strukturze, będącej połączeniem dwóch podwójnie wiązanych cyklicznych kolejek LRU i tablicy mieszającej, której ele-

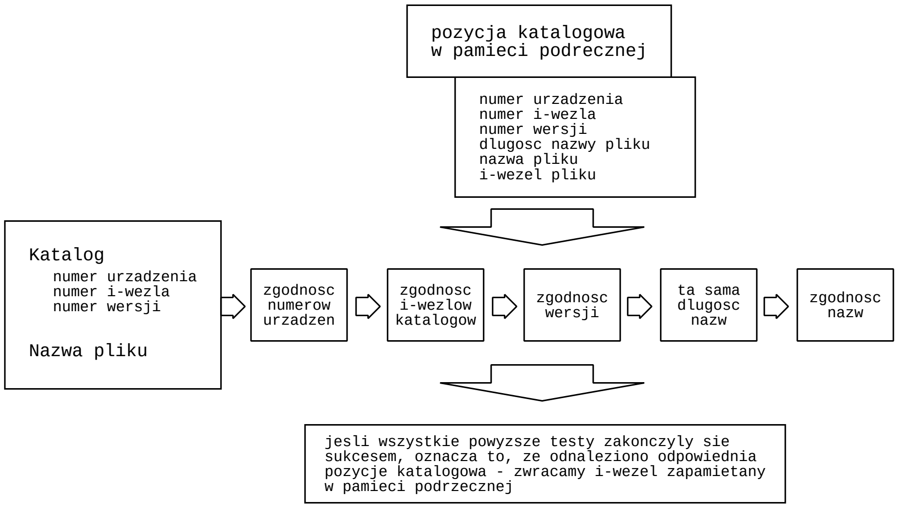
[6] Stała `EXT2_NAME_LEN`
[7] W zależności od flagi `NO_TRUNCATE`
[8] stała `DCACHE_NAME_LEN`

mentami są listy cykliczne, podwójnie wiązane. Ma to zapewniać szybki dostęp do danych na temat pozycji katalogowych (stąd użycie tablicy mieszającej) a jednocześnie umożliwiać kontrolę nad dodawaniem i usuwaniem pozycji (tym "zajmują się" kolejki).

Tablica 5: Informacje przechowywane w pamięci podręcznej katalogów

numer urządzenia	numer urządzenia na którym znajduje się plik
numer i-węzła katalogu	
numer i-węzła pliku	
numer wersji katalogu	przy zmianie zawartości katalogu zmienia się jego numer wersji
długość nazwy	
nazwa	

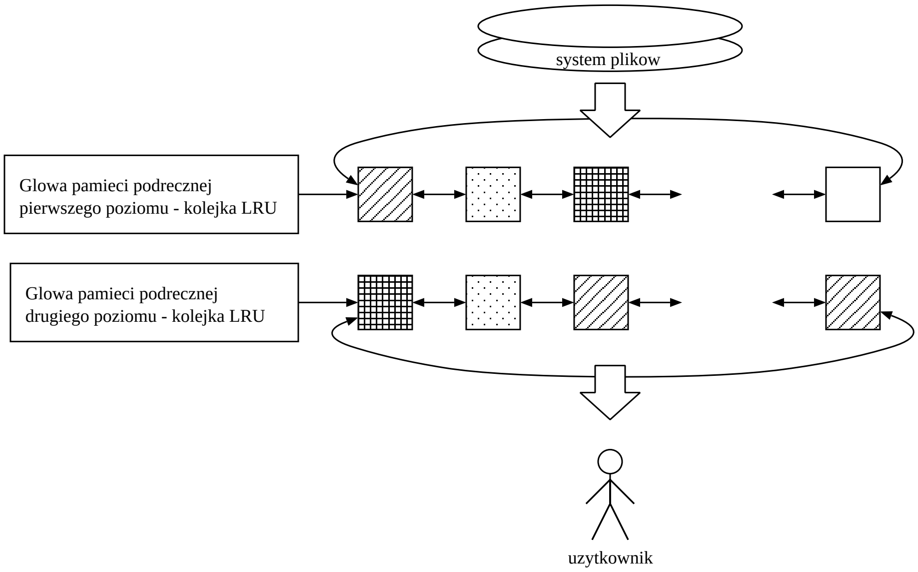
Ogólny schemat działania pamięci podręcznej katalogów przedstawiony jest na *rysunku 8*.



Rysunek 8: Ogólny schemat działania pamięci podręcznej katalogów

Pozycje katalogowe zorganizowane są w dwie kolejki LRU. Każda z kolejek zawiera 128 elementów[9]. Przy inicjalizacji kolejek jest do nich wstawiana odpowiednia ilość fałszywych elementów. Idea działania tych kolejek polega na tym, aby w pierwszej kolejce były przechowywane wszystkie odczytywane dane katalogowe (np. w wyniku komendy `ls`), natomiast do drugiej te dane, które są używane (np. nazwa pliku odczytanego poleceniem `cat`). Operacje, które

[9]stała `DCACHE_SIZE`



Rysunek 9: Dwupoziomowa pamięć podręczna katalogów

tylko odczytują katalogi i często dotyczą plików, których później nie chcemy używać zasilały pierwszą kolejkę. Pozycje z pierwszej kolejki "awansują" do drugiej dopiero, gdy zostaną użyte w jakiejś funkcji związanej z użyciem pliku (odczytanie, wykonanie, itp.)[10].

Przy dodawaniu elementów do kolejek nie wolno przekroczyć maksymalnego przewidzianego dla nich rozmiaru. Musi być zatem możliwość usuwania elementów z kolejki. W tym celu kolejki są zorganizowane w listę LRU. Każdy dostęp (a więc użycie, albo powtórne odczytanie np. za pomocą polecenia `ls`, itp) do pozycji katalogowej powoduje jej przesunięcie na koniec kolejki. Na początku kolejki znajduje się zawsze najdłużej nieużywany element i on jest w razie potrzeby usuwany.

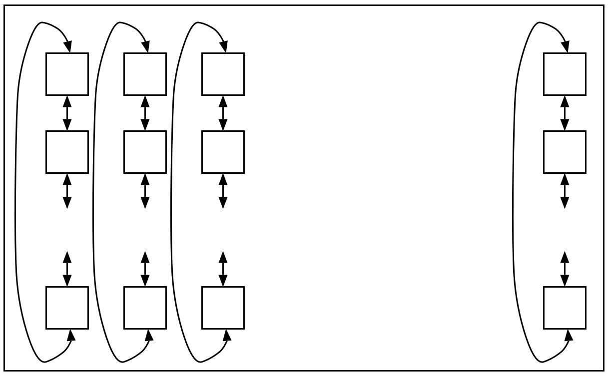
Pozycje zorganizowane są w 32-elementowej[11] tablicy mieszającej. Każdy element tablicy to cykliczna, podwójnie wiązana kolejka pozycji katalogowych[12] W sumie przechowuje się maksymalnie 256 pozycji[13]. Zatem w jednej kolejce

[10] Stwierdzenie "awansują" jest o tyle nieścisłe, że pozycja nie jest przenoszona z pierwszej kolejki do drugiej. Tworzony jest drugi egzemplarz danych na temat tej samej pozycji katalogowej i to on jest wstawiany do drugiej kolejki. Musimy mieć pewność, że poprzedni egzemplarz danych zostanie z czasem wyrzucony z pierwszej kolejki. Tak się stanie ponieważ oba egzemplarze mają ten sam numer w tablicy mieszającej. Nowy egzemplarz jest wstawiany na wcześniejszą pozycję w liście będącej elementem tablicy mieszającej. Wiadomo zatem, że to nowy egzemplarz będzie wyszukiwany, i przesuwany na koniec swojej kolejki LRU. Taka "udziwniona" implementacja pozwala zachować niezmiennik stałej długości obu kolejek.

[11] stała `DCACHE_HASH_QUEUES`

[12] Na rysunku dwupoziomowej pamięci podręcznej pozycje należące do tej samej kolejki w tablicy mieszającej zostały zaznaczone tym samym wzorem.

[13] 2∗ stała `DCACHE_SIZE`



Rysunek 10: Pamięć podręczna katalogów – Tablica mieszająca

przechowuje się średnio 8 pozycji[14]. Po inicjalizacji pamięci kolejki są puste. W trakcie działania systemu stopniowo się wypełniają, aż osiągną rozmiar maksymalny. Ilość elementów w pamięci podręcznej danych katalogowych nie zmniejsza się.

Podczas sprawdzania pamięci podręcznej odpowiednia kolejka tablicy mieszającej przeglądana jest liniowo.

Te dwie struktury (tablica mieszająca i podwójna lista LRU) utrzymywane są jednocześnie tak, że w każdej pozycji znajdują się wskaźniki do poprzedniego i następnego elementu w kolejce należącej do tablicy mieszającej oraz wskaźniki do poprzedniego i następnego elementu w odpowiedniej liście LRU.

15 Przekształcanie nazwy ścieżkowej w i-węzeł

Wewnętrzną reprezentacją pliku w systemie Linux, jak dowiedzieliśmy się w poprzednich rozdziałach, są tzw. i-węzły. Dla użytkownika naturalne jest jednak odwoływanie się do plików za pośrednictwem systememu katalogów i nazw tak, jak to się dzieje w przypadku komend `open`, `exec`, i wielu innych. Aby przekształcić ścieżkę opisującą dostęp do pliku system musi wykonać dwie czynności ściśle ze sobą powiązane: "przejść" przez odpowiednie katalogi i znaleźć i-węzel pliku. Wywołując polecenie

```
namei pathname
```

można przekonać się, jak przebiega "przechodzenie" drzewa dla zadanej ścieżki dostępu.

[14]Funkcja mieszająca zależy od: numeru i-węzła katalogu, długości nazwy pozycji, nazwy pozycji, oraz numeru urządzenia. Jej odchylenie standardowe mieszaj"w moim systemie jest równe około 1

Ścieżka dostępu może zaczynać się po prostu od nazwy – wtedy poszukiwanie pliku rozpoczyna się od bieżącego katalogu procesu, bądź od znaku '/' – wówczas poszukiwanie rozpoczyna się od korzenia. Katalog bieżący musi być pamiętany w u-obszarze procesu[15]. Zaskakujące jest to, że w u-obszarze procesu pamiętany jest również wskaźnik do i-węzła korzenia systemu plików[16]. Okazuje się, że jest to potrzebne, gdyż proces z uprawnieniami `root` może wywołać funkcję `chroot` zmieniającą korzeń dla danego procesu.

Algorytm znajdowania i-węzła jest bardzo prosty. W istocie polega przejściu po ścieżce dostępu od lewej do prawej, co doprowadza do lokalizacji szukanego pliku (dostajemy i-węzeł). Działanie algorytmu podsumowują tablice *Algorytm 1*, *Algorytm 2* i *Algorytm 3*.

Algorytm 1 funkcja `namei`

Warunek początkowy: Ścieżka dostępu do pliku
Warunek końcowy: I-węzeł z założoną blokadą odpowiadający plikowi identyfikowanemu daną ścieżką

1: Znajdź i-węzeł ostatniego katalogu ścieżki (funkcja `dir_namei`)
2: Znajdź pozycję odpowiadającą plikowi w znalezionym katalogu (funkcja `lookup`)
3: Jeśli znaleziony plik jest dowiązaniem symbolicznym – obsłuż je (funkcja `follow_link`)

Podczas działania algorytmu wielokrotnie wykonywane jest przeszukiwanie katalogu w poszukiwaniu pozycji o zadanej nazwie. Warto zwrócić uwagę na sposób w realizacji tego zadania. Tutaj zajmiemy się tylko algorytmem stosowanym w systemie `ext2`[17]. Jest on zaimplementowany w funkcji `lookup`. Został on w systemie Linux poprawiony w stosunku do algorytmu opisywanego przez Bacha przez wprowadzenie tablicy mieszającej pełniącej rolę pamięci podręcznej, która została opisana w poprzednim rozdziale. W tej pamięci są przechowywane są dane identyfikujące pliki o nazwach krótszych niż 15 znaków[18]) (patrz *Tablica 5*).

Przedstawiony tu algorytm dotyczy wersji `namei`, która jest używana przez najprostsze komendy. Dla funkcji `open` jest na przykład zaimplementowany bardziej skomplikowany algorytm biorący pod uwagę tryb otwierania pliku, itp.

16 Obsługa map bitowych zajętości bloków

Każda grupa zawiera mapę bitową zajętości plików. Mapy bitowe są przechowywane w pamięci podręcznej (osobnej dla każdego zamontowanego sys-

---

[15] w polu `fs` – jest to struktura zawierająca pole `pwd` będące wskaźnikiem do odpowiedniego i-węzła

[16] pole `fs->root`

[17] Dla każdego systemu plików (`FAT`, `ext`, itp) jest implementowany specyficzny dla niego algorytm przeszukiwania katalogów.

[18] stała `DCACHE_NAME_LEN`

Algorytm 2 funkcja `dir_namei`

Warunek początkowy: Nazwa ścieżki dostępu.
Warunek końcowy: Znaleziony i-węzeł.

```
jeśli ścieżka zaczyna się od '/' to
   roboczy i-węzeł := korzeń systemu plików
w przeciwnym przypadku
   roboczy i-węzeł := bieżący katalog
koniec jeśli
dopóki bieżąca składowa ścieżki nie jest pusta wykonuj
   Zwiększ ilość dowiązań (icount) do roboczego i-węzła
   Znajdź pozycję w katalogu odpowiadającą bieżącej składowej ścieżki
   (funkcja lookup)
   jeśli znaleziony plik to dowiązanie symboliczne to
      obsłuż dowiązanie symboliczne (funkcja follow_link)
   koniec jeśli
koniec dopóki
```

Algorytm 3 funkcja `lookup`

Warunek początkowy: I-węzeł – katalog i nazwa poszukiwanego pliku
Warunek końcowy: I-węzeł znalezionego pliku

```
jeśli nazwa to '..' to
   jeśli bieżący katalog jest korzeniem to
      zwróć katalog bieżący.
   w przeciwnym przypadku jeśli bieżący katalog jest korzeniem pod-
   montowanego systemu plików to
      zwróć katalog do którego jest zamontowany bieżący system plików.
   koniec jeśli
koniec jeśli
jeśli bieżący katalog jest poprawny i proces ma prawo jego wykonywania
to
   szukaj pozycji w katalogu w pamięci podręcznej za pomocą funkcji
   dcache_lookup.
   jeśli znaleziono pozycję to
      zwróć znalezioną pozycję.
   w przeciwnym przypadku
      znajdź pozycję za pomocą funkcji find_entry.
      dodaj pozycję do tablicy mieszającej.
      zwróć znalezioną pozycję.
   koniec jeśli
w przeciwnym przypadku
   błąd
koniec jeśli
```

temu plików). Rozmiar tej pamięci został określony na 8 map[19]. Jeśli system plików zawiera więcej niż 8 grup, mapy organizowane są w kolejkę LRU (w razie potrzeby wczytania nowej mapy, najdłużej nieużywana jest usuwana z pamięci).

Przy ładowaniu map bitowych istnieją dwa "skróty". Po pierwsze sprawdzamy, czy żądanie nie dotyczy tej samej mapy bitowej, której dotyczyło poprzednie. Wówczas od razu zwracamy odpowiednią mapę.

Specjalnie obsługiwany jest również przypadek małych systemów plików. Jeśli wszystkie mapy mieszczą się w strukturze bloku identyfikacyjnego. to są one przechowywane w pamięci w tablicy (nie jest to kolejka LRU). Możemy zwrócić wymaganą mapę w czasie stałym

17 Przydział bloków dyskowych

Przydział nowego bloku dyskowego dla pliku odbywa się na podstawie i-węzła pliku oraz tzw. celu. Cel to blok, który byłby najlepszy dla pliku. Na przykład jeśli powiększamy plik, który zajmuje na dysku 10 spójnych bloków, chcielibyśmy zaalokować kolejny – 11 bezpośrednio po końcu pliku. Taka strategia ma na celu minimalizowanie fragmentacji plików, co bezpośrednio przekłada się na szybkość dostępu do danych (ilość ruchów głowicy, ale również minimalizacja zapotrzebowania na dodatkowe struktury danych, które Linux przechowuje w pamięci i musi przeglądać podczas dostępu do plików[20]).

Algorytm przydzielania bloków próbuje znaleźć jak najbliższy w stosunku do celu blok. Poszukiwanie odbywa się w następującej kolejności (jeśli znajdziemy blok natychmiast się go zwraca – dalsze możliwości nie są brane pod uwagę).

1. Blok określony celem.
2. Blok znajdujący się wśród kilku najbliższych (wprzód) bloków należących do tej samej grupy. Ilość przeszukiwanych bloków n jest losowa ($n > 64$ i $64|n$)
3. Blok znajdujący się na początku wolnego obszaru co najmniej 8 bloków[21]. Przeszukiwanie rozpoczyna się od bloku na którym zakończono poprzednie przeszukiwanie (nie od początku grupy).
4. Dowolny wolny blok w grupie, do której należy cel.
5. Dowolny wolny blok w dowolnej grupie.

[19] stała `EXT2_MAX_GROUP_LOADED`

[20] Gdy dbamy o to aby pliki były jak najbardziej spójne mamy szanse zmniejszyć zapotrzebowanie na deskryptory grup i mapy bitowe zajętości bloków.

[21] Sprawdza się, czy jest wyzerowany bajt w mapie bitowej zajętości bloków. Widać więc, że gdy wolny obszar nie wypadnie w mapie dokładnie na granicy bajtów, nie zostanie znaleziony. Tak naprawdę gwarancję znalezienia mamy dopiero dla ciągów 15-blokowych i dłuższych.

W punktach 3-5, przydzielany jest zawsze pierwszy blok ze spójnego zbioru bloków, który został znaleziony. Jeśli wymagana jest prealokacja bloków[22], alokowane jest w miarę możliwości następnych 7 bloków[23] zajmujących spójny obszar i należących do tej samej grupy.

18 Otwieranie pliku

Funkcja systemowa *open()* służy do otwierania pliku. Jej wywołanie jest pierwszym krokiem jaki musi wykonać proces, by uzyskać dostęp do danych w pliku. Składnia wywołania jest następująca:

fd = *open*(**nazwa_sciežkowa**, **znaczniki**, **prawa_dostępu**).

Plik możemy otworzyć w różnych celach. Możemy na przykład chcieć pisać do pliku lub czytać z niego. Na cel otwierania pliku wskazują **znaczniki**. W **znacznikach** zawarta jest również informacja o sposobie, w jaki plik ma być otworzony - możemy między innymi zażyczyć sobie, żeby nasz proces nie był wstrzymywany przy wykonywaniu operacji na pliku, jeśli operacja ta miałaby być zablokowana w oczekiwaniu na jakieś zdarzenie.

Jest również możliwość otwarcia pliku w celu jego stworzenia - funkcja *create()* nie jest niczym innym niz wywołaniem funkcji *open()* z odpowiednimi **znacznikami**. Właśnie w przypadku, gdy chcemy otworzyć plik, funkcja *open()* korzysta ze swojego trzeciego parametru - zależeć od niego będą prawa dostępu do tworzonego pliku.

Jeśli funkcja *open()* się powiedzie, jako jej wynik przekazywany jest deskryptor otwieranego pliku. Deskryptor ten jest używany przez takie operacje na danym pliku jak czytanie, pisanie czy zamykanie.

18.1 Opis działania funkcji *open()* :

- Jądro rezerwuje wolną pozycję w lokalnej tablicy deskryptorów plików procesu. Jeśli funkcja *open()* się powiedzie, indeks tej pozycji będzie zwróconym przez funkcję deskryptorem pliku **fd**. Każde dwa otwarte przez proces pliki (nawet jeśli są tym samym plikiem) mają rożne deskryptory. Tablica deskryptorów ma ustalony rozmiar.

- Jądro znajduje nieużywaną pozycję (nazwijmy ją **f**) na globalnej liście plików, na którą, po udanym zakończeniu funkcji *open()*, wskazywać będzie pozycja w tablicy deskryptorów plików odpowiadająca **fd**. Dla danego pliku, za każdym razem gdy jest on otwierany, przydzielana jest nowa pozycja na globalnej liście plików. Wszystkie te pozycje wskazują na ten sam i_węzeł na globalnej liście i_węzłów w pamięci. Obie globalne listy mają ograniczony rozmiar. Rysunek 11pokazuje zależność pomiędzy listami i_węzłów i plików oraz lokalnymi tablicami deskryptorów procesów.

[22] Prealokacja to wstępne rezerwowanie większej ilości bloków w korzystnym (zbliżonym do celu) miejscu po to, aby w razie późniejszego zapotrzebowania na nowe bloki dla danego pliku można było czerpać z puli prealokowanych bloków.

[23] Stała zaszyta na sztywno w kodzie.

- Wywoływana jest funkcja *open_namei()*, z której dostajemy i_węzeł odpowiadający otwieranemu plikowi. Pola struktury danego i_węzła są już odpowiednio zainicjalizowane.

- W przypadku, gdy zwykły plik ma być otworzony do pisania, jądro musi sprawdzić, czy żaden z procesów nie odwzorował jakiegoś fragmentu danego pliku w swoją przestrzeń adresową, zabraniając jednocześnie pisania na nim.

- Jądro inicjalizuje pola **f**. Ustawiana jest na przykład bieżąca pozycja w pliku (czyli miejsce, od którego rozpocznie się czytanie - przyjęto, iż miejscem tym jest początek pliku). Jednym z pól każdego i_węzła jest wskaźnik do struktury, która zawiera wskaźniki do operacji charakterystycznych dla systemu plików oraz typu pliku, z którymi i_węzeł jest związany. Oprócz wskaźników do operacji jakie mogą być przeprowadzane na i_węźle, jest również wskaźnik do struktury zawierającej operacje na pliku - właśnie na podstawie tego pola jest inicjalizowane odpowiednie pole **f**. Jedna z takich operacji na pliku jest tu wywoływana - jest to funkcja *open()*.

Opis funkcji *open_namei()* umieszczamy osobno, aby opis algorytmu funkcji *open()* był czytelniejszy.

Funkcja *open_namei()* znajduje i_węzeł pliku na podstawie jego nazwy ścieżkowej. Funkcja nie tylko zapewnia dostęp do odpowiednich danych, lecz także sprawdza, czy proces, który chce otworzyć dany plik w określony przez siebie sposób ma prawo to uczynić.

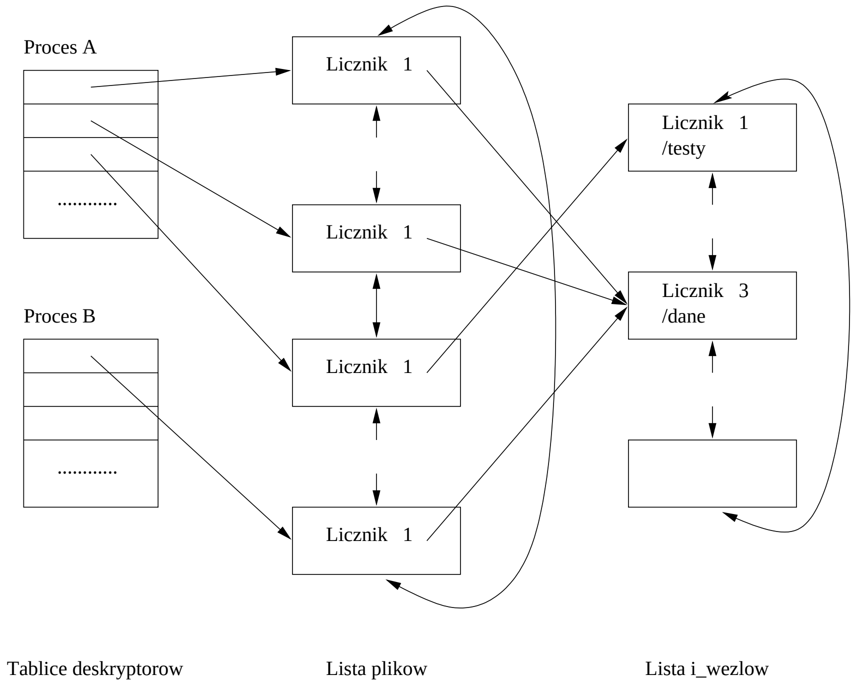
18.2 Opis działania funkcji *open_namei()* :

- Jądro znajduje i_węzeł katalogu, który zawiera dany plik oraz nazwę tego pliku (czyli ciąg znaków występujący po ostatnim znaku '/' w nazwie ścieżkowej pliku). Używa w tym celu funkcji *dir_namei()*. Zatem w przypadku, gdy plik jest katalogiem jego i_węzeł został już znaleziony. Pisać do katalogu może jedynie jądro systemu.

- Jądro stara się odnaleźć i_węzeł danego pliku, wywołując funkcję *lookup()*.

- Funkcja *open_namei()* może odpowiadać za utworzenie pliku. Wówczas jeśli i_węzeł danego pliku nie istnieje, to jest on tworzony za pomocą operacji na i_węźle zdefiniowanej dla katalogu, który ma zawierać dany plik - dzięki temu struktury danych, które zależą od konkretnego systemu plików i typu pliku mogą być odpowiednio zainicjalizowane.

- Znaleziony i_węzeł może być i_węzłem pliku, który jest dowiązaniem symbolicznym do innego pliku. Jądro musi odnaleźć i_węzeł, na który wskazuje dane dowiązanie. Znaleziony i_węzeł będzie wynikiem funkcji *open_namei()*. I_węzeł ten zawiera informacje o typie otwieranego pliku, a także wskaźniki do operacji specyficznych dla danego typu pliku.



Rysunek 11: Pokazuje zależność pomiędzy globalnymi listami i_węzlów i plików oraz lokalnymi tablicami deskryptorów dwóch procesów. Niech procesy nazywają się odpowiednio A i B. Zakładamy, że proces A dwukrotnie otworzył plik "/dane" a następnie plik "/testy". Proces B otworzył plik "/dane". Wymienione pliki są otworzone jedynie przez procesy A i B.

- Jądro wykonuje szereg testów, by sprawdzić, czy proces, który chce otworzyć dany plik w określony przez siebie sposób ma prawo to uczynić. W tym celu wywoływana jest między innymi funkcja *permission()*, która odwołuje się do odpowiednich pól danego i_węzła. Jeśli otwierany plik wymaga specyficznego sprawdzenia praw dostępu do niego, zamiast funkcji *permission()* wywoływana jest operacja o tej samej nazwie zdefiniowana dla i_węzła danego pliku.

- Dosyć złożony jest sposób sprawdzenia, czy przy otwieraniu pliku jego rozmiar może byc zmniejszony do 0. Najpierw jądro kontroluje, czy żaden z procesów nie odwzorował jakiegoś fragmentu danego pliku w swoją przestrzeń adresową, zabraniając jednocześnie dokonywania na nim zmian. Następnie sprawdza, czy na danym pliku nie została założona blokada obowiązkowa. Proces nie będzie mógł otworzyć pliku z założoną blokadą obowiązkową, której nie jest właścicielem. Jeśli testy zakończyły się pomyślnie w i_węźle zapisywane są zmiany związane z rozmiarem pliku, a sam plik jest skracany.

19 Zamykanie pliku

Proces zamyka otwarty plik, gdy przestaje z niego korzystać. Proces nie musi jednak zamykać pliku przed zakończeniem swojego działania. W momencie, gdy proces się kończy, jądro sprawdza, które spośród jego deskryptorów są wciąż aktywne i zamyka odpowiadające im pliki.

Składnia wywołania funkcji systemowej *close()* jest następująca:

close(**fd**),

gdzie fd jest deskryptorem pliku, który chcemy zamknąć.

19.1 Opis działania funkcji *close()* :

- Jądro sprawdza, czy pozycja w lokalnej tablicy deskryptorów plików procesu odpowiadająca deskryptorowi **fd** wskazuje na pewien plik na globalnej liście plików. Jądro zaznacza, że odpowiednia pozycja w tablicy deskryptorów jest wolna.

- Jądro sprawdza, czy na dany plik zostały założone jakieś blokady, odwołując się do odpowiedniego pola w i_węźle pliku. Jeśli na pliku zostały założone blokady typu POSIX, jądro zdejmuje wszystkie blokady założone na plik przez proces, który chce dany plik zamknąć. Jeśli natomiast na pliku założono blokady typu FLOCK, jądro zdejmuje tylko te blokady, które dotyczą odpowiedniego wcielenia pliku (odpowiedniej pozycji na liście plików) i to pod warunkiem, że dla żadnego procesu nie istnieje pozycja w jego tablicy deskryptorów, która wskazywałaby na dane wcielenie pliku.

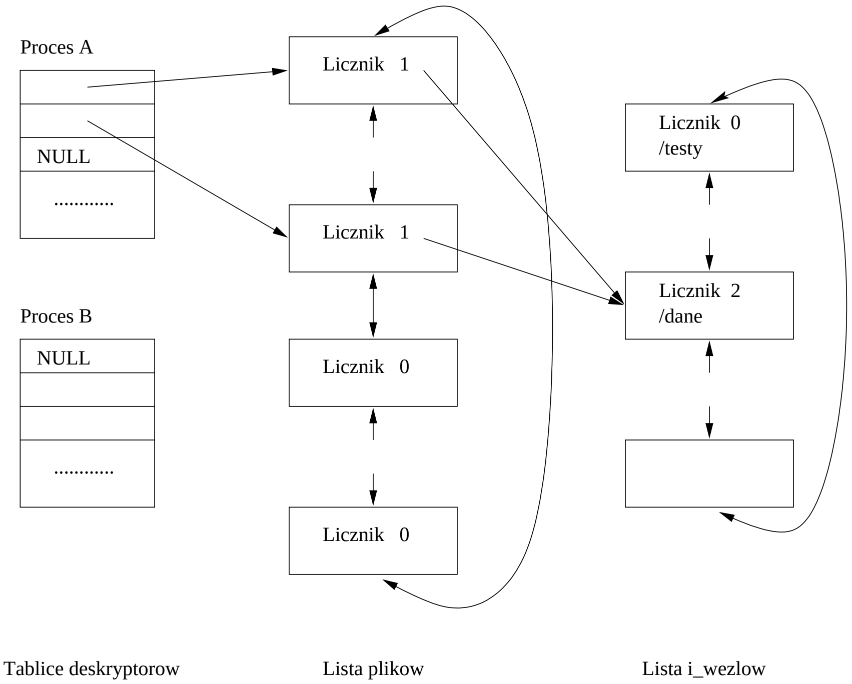
- W pozycji na liście plików, na którą wskazywał deskryptor **fd** zmniejszany jest licznik odwołań do niej. Licznik ten może być większy niż 1 w wyniku wywołania funkcji *dup()* i *fork()*.

- Może się zdarzyć, że dla żadnego procesu nie istnieje deskryptor związany z daną pozycją na liście plików. Wówczas pozycja ta jest zwalniana, po uprzednim sprawdzeniu, czy nie należy wykonać pewnych dodatkowych operacji związanych z typem zwalnianego pliku (jądro odwołuje się do odpowiedniej operacji na pliku, zdefiniowanej dla danej pozycji na liście plików). Zwolnienia pozycji na liście plików dokonuje się poprzez zaznaczenie, że pozycja ta nie wskazuje na żaden i_węzeł. Następnie za pomocą funkcji *i_put()* jądro zwalnia i_węzeł w pamięci przydzielony danemu plikowi w funkcji *open()*.

20 Czytanie z pliku

Funkcja systemowa *read()* służy do wczytania danych z otwartego pliku. Składnia wywołania funkcji systemowej *read()* jest następująca:

l = *read* (**fd**, **bufor**, **liczba**), gdzie



Rysunek 12: Pokazuje odpowiednie pozycje struktur danych przedstawionych na poprzednim rysunku w przypadku, gdy proces A zamknął plik "/testy" a proces B plik "/dane".

fd jest deskryptorem pliku, **bufor** jest adresem struktury danych w procesie użytkownika, która będzie zawierała wczytane dane po poprawnym zakończeniu wywołania, **liczba** to liczba bajtów, jaką chce wczytać użytkownik, a **l** jest liczbą faktycznie wczytanych bajtów.

Funkcja *read()* rozpoczyna czytanie od bieżącej pozycji w pliku, a po zakończeniu czytania aktualizuje bieżącą pozycję zapisaną w strukturze na globalnej liście plików. Zatem kolejne operacje czytania z pliku będą dostarczały dane sekwencyjnie.

20.1 Opis działania funkcji *read()* :

- Jądro znajduje pozycję na globalnej liście plików, na którą wskazuje pozycja w lokalnej tablicy deskryptorów plików procesu odpowiadająca deskryptorowi **fd**. Ze znalezionej pozycji pobiera i_węzeł pliku. Zależność pomiędzy wymienionymi wyżej strukturami jest opisana w rozdziale dotyczącym funkcji systemowej *open()*.
- Jądro sprawdza, czy proces ma prawo odczytać dany plik - w tym celu odwołuje się do odpowiedniej pozycji na globalnej liście plików.

- I_węzeł pliku zawiera wskaźnik do listy wszystkich blokad założonych na dany plik. Jeśli na danym pliku są założone blokady obowiązkowe, jądro musi przejrzeć listę blokad danego pliku, by sprawdzić, czy proces może z niego czytać. Jeśli jakaś blokada obowiązkowa uniemożliwia odczyt pliku, jądro usiłuje uśpić proces, o ile tylko operacje na pliku, który chcemy odczytać mogą być blokowane. Zanim proces zostanie uśpiony, jądro sprawdza, czy jego uśpienie nie spowoduje zastoju. Gdy proces się budzi, jądro sprawdza, czy podczas snu procesu z pliku nie zdjęto wszystkich blokad obowiązkowych. Jeśli na pliku są założone blokady obowiązkowe, jądro ponownie przegląda całą ich listę. Proces, który chce odczytać dane z pewnego obszaru pliku nie może zostać wstrzymany przez żadną blokadę obowiązkową obejmującą ten obszar w przypadku, gdy jest jej właścicielem. Proces zostanie natomiast zablokowany, jeśli chce odczytać obszar pliku, na którym założył obowiązkową blokadę proces, który chce do niego pisać.
- Jądro sprawdza, czy może zapisać odpowiednią liczbę danych pod adres odpowiadający zmiennej **bufor**.
- Jądro wywołuje specyficzną dla systemu plików i typu pliku operację *read()* zdefiniowaną dla pozycji na globalnej liście plików, odpowiadającej danemu deskryptorowi pliku. Dla przykładu omówimy tu operację *read()* zdefiniowaną dla zwykłego pliku w systemie plików EXT2 - *generic_file_read()*.

20.2 Opis działania funkcji *generic_file_read()*:

20.2.1 Wstęp:

Jeśli proces sekwencyjnie czyta dane z pliku, jądro zleca, by kolejne dane z pliku znajdujące się na dysku wczytywać do pamięci, zanim jeszcze proces zażąda odczytu odpowiednich danych. Zatem jądro przewiduje, do których danych z pliku będzie chciał się odwołać proces po to, aby żądania odczytu danych przez proces jak najczęściej były zaspokajane jedynie przez odwołanie się do pamięci operacyjnej. Przyspiesza to oczywiście 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 liście plików odpowiadająca danemu deskryptorowi pliku przechowuje informacje związane z czytaniem danego pliku z wyprzedzeniem. Z odpowiednich pól struktury na liście plików jądro może się więc dowiedzieć między innymi, gdzie zostało zakończone czytanie z wyprzedzeniem, ile danych zostało wówczas wczytanych oraz ile danych maksymalnie może być wczytanych w ten sposób.

20.2.2 Opis algorytmu:

- Jeśli jądro stwierdzi, że pozycja, od której proces chce rozpocząć czytanie danego pliku znajduje się wewnątrz obszaru, który poprzednio był wczytany z wyprzedzeniem przyjmuje, że plik jest czytany sekwencyjnie, czyli, że powinno być zainicjowane czytanie z wyprzedzeniem danego pliku. W przeciwnym przypadku, jądro zaznacza w odpowiedniej pozycji na liście plików, że czytanie z wyprzedzeniem się nie odbędzie.

- Jądro sprawdza, ile danych może być wczytanych z wyprzedzeniem. Rozmiar danych może się okazać zbyt mały, by czytanie z wyprzedzeniem warto było inicjować. Rozmiar danych do wczytania musi należeć do ustalonego przedziału.

- Następnie jądro w pętli wykonuje opisane poniżej czynności, aż do zakończenia czytania.

- Jądro szuka w pamięci operacyjnej ramki, która zawiera odpowiednie dane z naszego pliku. Może się zdarzyć, że jądro nie znajdzie w pamięci operacyjnej takiej ramki - nawet jeśli plik był czytany z wyprzedzeniem, odpowiednia ramka mogła zostać usunięta z pamięci. Jądro rezerwuje wówczas wolną ramkę w pamięci i wstawia ją do odpowiednich struktur przechowujących ramki. Następnie zleca wypełnienie ramki danymi z odpowiadającego jej bloku na dysku.

- Jeśli dana ramka jest wypełniona aktualnymi danymi, bądź jej zawartość jest właśnie aktualizowana, jądro inicjuje wczytanie odpowiedniej liczby stron z wyprzedzeniem wywołując funkcję *generic_file_readahead()*. Następnie jądro czeka, aż na danej ramce zostaną zakończone wszystkie zainicjowane wcześniej operacje wejścia-wyjścia i zostanie ona odblokowana. Jeśli okaże się, że odblokowana ramka zawiera nieaktualne dane - jądro synchronicznie wypełnia ją danymi z odpowiadającego jej bloku na dysku. Teraz pozostaje jeszcze skopiować dane z odpowiedniego fragmentu ramki do przestrzeni adresowej procesu.

21 Pisanie do pliku

Funkcja systemowa *write()* służy do zapisania danych w otwartym pliku. Składnia wywołania funkcji systemowej *write()* jest następująca:

l = write (**fd**, **bufor**, **liczba**), gdzie

fd jest deskryptorem pliku, **bufor** jest adresem struktury danych w procesie użytkownika, której zawartość należy zapisać na dysku, **liczba** to liczba bajtów, którą chce zapisać użytkownik, a **l** jest liczbą rzeczywiście zapisanych bajtów.

21.1 Opis działania funkcji *write()* :

- Jądro znajduje pozycję na globalnej liście plików odpowiadającą deskryptorowi **fd**. Ze znalezionej pozycji pobiera i_węzeł pliku. Zależność pomiędzy wymienionymi wyżej strukturami jest opisana w rozdziale dotyczącym funkcji systemowej *open()*.

- Jądro sprawdza, czy proces ma prawo zapisać dane do pliku - w tym celu odwołuje się do odpowiedniej pozycji na globalnej liście plików.
- Podobnie jak dla funkcji *read()* jądro musi teraz sprawdzić, czy któraś z obowiązkowych blokad założonych na dany plik nie będzie kolidować z operacją, jaka ma być na nim przeprowadzona - w tym przypadku chodzi oczywiście o zapis danych do pliku. Proces, który chce zapisać dane do pewnego obszaru pliku, nie może zostać wstrzymany przez żadną blokadę obowiązkową obejmującą ten obszar w przypadku, gdy jest jej właścicielem. Dany proces zostanie natomiast wstrzymany przez jakąkolwiek blokadę obowiązkową obejmującą odpowiedni obszar, jeśli jej właścicielem jest inny proces. Jądro będzie kontynuować wykonanie funkcji *write()*, gdy wszystkie blokady zabraniające procesowi pisać do pliku zostaną zdjęte.
- Jądro sprawdza, czy może odczytać odpowiednią liczbę danych spod adresu odpowiadającego zmiennej **bufor**.
- Jądro wywołuje specyficzną dla systemu plików i typu pliku operację *write()* zdefiniowaną dla pozycji na globalnej liście plików odpowiadającej danemu deskryptorowi pliku. Dla przykładu omówimy tu operację *write()* zdefiniowaną dla zwykłego pliku w systemie plików EXT2 - *ext2_file_write()*. Ponieważ podczas wykonywania operacji *write()* zawartość i_węzła danego pliku może ulec zmianie, na czas wykonania tej operacji jądro zakłada blokadę na i_węzeł. Gdyby na przykład kilka procesów jednocześnie przydzielało różne bloki na dysku na te same adresy logiczne, i_węzeł pliku zostałby zniszczony.

21.2 Opis działania funkcji *ext2_file_write()*:

- Jądro odwołuje się do odpowiedniej pozycji na liście plików, aby ustalić miejsce, od którego należy rozpocząć zapis danych do pliku. Miejscem tym może być bieżąca pozycja w pliku lub koniec pliku.
- Proces może oczywiście zapisać do pliku ograniczoną liczbę danych, ponieważ rozmiar pliku jest ograniczony.
- Jądro w pętli wykonuje opisane poniżej czynnosci, aż do zakończenia zapisu. Pojedynczy obrót pętli jest związany z zapisem na dysku jednego bloku dyskowego.
- Jądro wyznacza logiczny numer bloku dyskowego zawierającego miejsce, od którego należy rozpocząć zapis danych do pliku.
- Jądro nie zapisuje danych bezpośrednio na dysk, lecz najpierw umieszcza je w pamięci buforowej, by potem przepisać zawartość bufora na dysk. Jeśli zapis danych z bufora na dysk ulegnie opóźnieniu, bufor będzie zawierał prawidłowe dane, które powinny się znajdować w odpowiadającym mu bloku. Jądro szuka więc w podręcznej pamięci buforowej bufora, który opowiada wyznaczonemu blokowi. Jeśli pamięć buforowa nie zawiera

odpowiedniego bufora, do wyznaczonego bloku jądro przyporządkowuje nowy wolny bufor. Żaden proces nie będzie mógł użyć znalezionego bufora, dopóki bieżący proces będzie wykorzystywał ten bufor.

- Gdyby jądro usiłowało zapisać na dysk jedynie fragment bloku zniszczyłoby zawartość tej części bloku, która powinna pozostać nie zmieniona. Zatem jeśli jądro musi zapisać na dysku tylko część bloku, odwołuje się najpierw do nagłówka odpowiedniego bufora, by sprawdzić czy zawiera on dane, które powinny znajdować się na dysku. W przypadku gdy dane w buforze są niewłaściwe, jądro synchronicznie wczytuje do niego zawartość odpowiedniego bloku dyskowego.

- Jądro kopiuje w odpowiednie miejsce danego bufora wyznaczony fragment danych, które proces chce zapisać do pliku.

- Następnie jądro wykonuje czynności związane z zapisem zawartości danego bufora na dysk - jak się przekonamy nie musi to jednak oznaczać, że odpowiednie dane są niezwłocznie zapisywane na dysk.

- Proces otwierając plik mógł nie zażądać, by przy wykonywaniu operacji zapisu danych do pliku był wstrzymywany do czasu jej zakończenia. Jądro zaznacza wtedy tylko, że proces nie korzysta już z danego bufora. Zanim dane zawarte w buforze zostaną zapisane na dysk, są przez pewien czas przechowywane w podręcznej pamięci buforowej na wypadek, gdyby wkrótce miały zostać odczytane lub zmodyfikowane przez jakiś proces. W ten sposób jądro stara się ograniczać liczbę operacji dyskowych.

- W przypadku, gdy przy wykonywaniu operacji zapisu danych do pliku proces musi być wstrzymywany do czasu jej zakończenia, jądro wstawia dany bufor do podręcznej tablicy przechowującej bufory. Jeśli tablica zostanie wypełniona, jądro synchronicznie zapisuje zawartość każdego zawartego w niej bufora do odpowiedniego bloku dyskowego. W przypadku, gdy po wyjściu z opisanej powyżej pętli lokalna tablica buforów nie będzie pusta, jądro synchronicznie zapisze na dysku każdy jej element.

- Jeśli jądro nie rozważyło jeszcze wszystkich danych przekazanych przez proces, zostają podjęte działania związane z zapisem na dysk kolejnego ich fragmentu.

- W wyniku zapisywania danych do pliku rozmiar pliku zapisany w odpowiadającym mu i_węźle, może oczywiście ulec zmianie. Zmienić może się również bieżąca pozycja w pliku przechowywana w odpowiedniej strukturze z globalnej listy plików.

22 Blokowanie plików i rekordów w Linuxie.

W systemie wieloprocesowym takim jak Linux, w wyniku współbieżnego działania wielu procesów, może dojść do sytuacji, w której kilka procesów będzie jednocześnie zapisywać informacje do pliku z danymi, bądź w trakcie czytania zawartości pliku przez jeden proces inny będzie modyfikował dane w nim zawarte. Łatwo wyobrazić sobie skutki takiego działania. Aby utrudnić powstanie niespójności danych w Linuxie (podobnie jak w innych systemach rodzaju Unix) wprowadzono możliwość zakładania blokad na cały plik bądź jego cześć.

Wyróżnia się dwa rodzaje blokad:

- blokady zalecane (z ang. *advisory*)
- blokady obowiązujące (z ang. *mandatory*)

Blokady zalecane występują w dwóch standardach: - *POSIX* - umożliwiające zablokowanie zarówno części pliku, jak i całości oraz - *BSD* - blokujące cały plik. Uwaga: W Linuxie nie wolno zakładać na jednym pliku blokad różnych standardów, bo może to doprowadzić do zastoju (DEADLOCK). Blokowanie plików z użyciem mechanizmu blokad zalecanych jest skuteczne tylko w przypadku, gdy wszystkie procesy stosują się do nastepujacych zasad: - przed rozpoczęciem operacji na pliku proces zakłada blokadę - po zakończeniu pracy z plikiem proces zdejmuje blokadę. Niestety blokady zalecane nie zabezpieczają przed zapisem / odczytem danych z obszaru objętego blokadą, tzn. proces który chce wykonać operacje na pliku jeśli nie będzie próbował założyć blokady to nie zostanie powstrzymany przed wykonaniem tej operacji.

Blokady obowiązujące wprowadzono aby utrudnić tego typu działanie. Proces nie może przeprowadzić operacji na pliku z danymi jeśli jest na nim blokada obowiązująca, aczkolwiek blokady obowiązujące nie sprawdzają się we wszystkich przypadkach (niekiedy mogą być nawet szkodliwe np. mogą zawiesić pracę servera NFS w oczekiwaniu na zwolnienie blokady z pliku) oraz można je obejść. Ich celem bylo uniemożliwienie wprowadzenia niespójności w plikach w wypadku błędów w programach, niestety nie realizują w pełni tego zadania. Dodatkowo w przypadku błędnie działającego programu, nie można zmodyfikować pliku dopóki ten program nie zakończy swojego działania. Powinno się ich używać jedynie na lokalnych systemach plików. Włączenie blokad obowiązujących jest w Linuxie opcjonalne (domyślnie sa wyłączone). Blokady obowiązujące występują tylko w standardzie POSIX, przed założeniem blokady trzeba dodatkowo wyzerować bit group_execute (prawa dostępu do pliku) oraz ustawić bit group_id (SGDI) flagi pliku. Blokowanie obowiązujące nie działa w przypadku plików mapowanych funkcją mmap() z flagą MAP_SHARED.

Jak te mechanizmy są zrealizowane w Linuxie ?

1. blokady w standardzie POSIX za pomocą funkcji systemowej - *fcntl()*
2. blokady w standardzie BSD za pomocą funkcji systemowej - *flock().*

Blokady obu standardów są implementowane w oparciu o te same struktury danych w jądrze systemu (związane z i-węzłem pliku).

22.1 Wywołanie funkcji *fcntl*

Funkcja fcntl() może być wywołana w celu sprawdzenia czy istnieją jakieś blokady konfliktujące z proponowaną. Blokady typu POSIX mogą się przecinać jeśli tylko zostały założone w tym samym celu (np. do pisania lub do czytania). Blokady mogą być zakładane zarówno na cały plik jak i na jego część. Są dwa rodzaje wywołań tej funkcji w celu założenia blokady, jedno przy niemożności założenia blokady zawiesza proces(proces jest usypiany na kolejce związanej z blokadą), natomiast drugie powoduje zwrócenie odpowiedniej informacji o braku możliwości założenia blokady. Kiedy blokada zostaje zdjęta wszystkie procesy uśpione budzą się i próbują jeszcze raz założyć blokadę. W przypadku kiedy proces ma być uśpiony jest wywoływany dodatkowo algorytm sprawdzający możliwość powstania zastoju. Blokady typu POSIX są związane z konkretnym procesem, tylko właściciel może zdjąc blokadę.

Tablica 6: Semantyka blokad fcntl

Isniejące blokady	Zał. blokada czytania	Zał. blokada pisania
Brak	Możliwa	Możliwa
Kilka blokad czytania	Możliwa	Nie dozwolona
Jedna blokada pisania	Nie dozwolona	Nie dozwolona

22.2 Wywołanie funkcji *flock* w BSD

Implementacja blokad jest dużo prostsza niż w przypadku fcntl(), bo nie trzeba sprawdzać przecięć z istniejącymi blokadami. Semantyka podobna do POSIX tzn. może być jednocześnie kilka blokad typu dzielonego i tylko jedna wyłączna, najczęściej pierwszego typu używa się przy czytaniu z pliku, a drugiego przy pisaniu. Blokada obejmuje cały plik. Jest związana ze wskaźnikiem do pliku, w związku z tym może być przekazywana za pomocą fork () do procesów potomnych i nie można ustalić kto jest właścicielem blokady (dlatego nie można mieszać blokad różnych standardów bo nie wiadomo jakie procesy wzięły by udział w zastoju). Blokada istnieje dopóki plik nie zostaje zamknięty.

23 Funkcje systemowe systemu pliku.

Zamieszczony tu został opis niektórych funkcji systemowych systemu plików systemu Linux.

23.1 Funkcja Mknod()

Funkcja ta służy do tworzenia plików różnych typów: m.in łącz nazwanych, plików specjalnych urządzeń, plików zwyczajnych. Możemy określić tryby dostępu do pliku.

Wywoływanie: mknod (ścieżka dostępu, tryby dostępu, nr. urządzenia);

Algorytm:

- Poza tworzeniem kolejek FIFO tylko superużytkownik ma prawo wywoływania tej funkcji. Tworzony plik nie może być katalogiem, ponieważ do tworzenia katalogów służy odrębna funkcja mkdir().
- Sprawdzane jest czy istnieje taki typ plików specjalnych w systemie.
- Pobierany jest i-węzeł katalogu macierzystego dla tego pliku (wynikający ze ścieżki dostępu).
- Jeśli katalog ma prawa do zapisu i jeśli w systemie plików istnieje odpowiednia funkcja mknod to:
- zwiększa licznik dowiązań do tego katalogu,
- blokuje go i wywołuje funkcję z odpowiedniego systemu plików
- po czym odblokowuje katalog i zwalnia i-węzeł z nim związany.
- W zależności od systemu plików nadawane są tworzonemu plikowi uprawnienia wprowadzone przy wywołaniu funkcji.

23.2 Funkcja link.

Funkcja służy do utworzenia nowego dowiązania do pliku. Tworzony jest tzw. *hard link*. Nowe dowiązanie jest dodawane do katalogu, ktory jest wyspecyfikowany w nowej ścieżce. Proces wołający musi mieć prawa do zapisu w tym katalogu. Dowiązanie określone przez drugi parametr nie może istnieć. Oba dowiązania będą nierozróżnialne. Dowiązywanie do katalogu jest dozwolone jedynie procesom superużytkownika (mogą powstać pętle w grafie katalogów) Funkcja ta nie służy do zmiany istniejącego dowiązania.

Wywołanie :
link (stara_nazwa, nowa_nazwa)
Algorytm:

- Pobierany jest i-węzeł pliku, do którego będzie tworzone dowiązanie,
- kopiuje go za pomocą getname() do wolnej przestrzeni jadra
- pobiera i-węzeł pliku nowego dowiązania oraz nazwę z katalogiem - funkcja dir_namei()
- sprawdza czy można pisać w katalogu oraz czy można zmodyfikować jego i-węzeł
- sprawdza czy oba katalogi są w ramach jednego urządzenia
- jeśli istnieje funkcja odpowiednia dla danego systemu plików to: po zablokowaniu na czas przeprowadzenia operacji katalogu wykonuje operacje odpowiadającą danemu systemowi plików.
- zwalnia zarezerwowane obszary pamięci jądra.

23.3 Funkcja unlink().

Usuwa dowiązanie do pliku, zmniejsza również licznik dowiązań do danego pliku. Jeśli w wyniku tego licznik osiągnął wartość zero to plik jest usuwany z katalogu, bloki dyskowe przez niego zajmowane są zwalniane, a także zwalniany jest jego i-węzęł.

Wywołanie:
unlink (nazwa_pliku)
Algorytm:

- Pobierany jest i-węzęł pliku
- kopiuje go za pomocą getname() do wolnej przestrzeni jądra
- pobiera i-węzęł katalogu macierzystego tego pliku
- sprawdza możliwości wprowadzania zmian w katalogu jak i w jego i-węźle
- jeśli dostępna jest funkcja *unlink* dla danego systemu plików to ją wykonuje
- oczywiście zablokowuje katalog na czas wykonywania operacji
- funkcja ta w zależności od rodzaju systemu plików wykonuje odpowiednie operacje na i-węźle i strukturach związanych z plikami.
- zwalnia pamieć przydzieloną w przestrzeni jądra

23.4 Funkcja chmod()

Realizuje zmianę trybu dostępu do pliku. Nie można za jej pomocą zmienić rodzaju pliku. Proces wołający musi mieć obowiązujące identyfikator równy identyfikatorowi właściciela pliku lub być procesem superużytkownika. Funkcja modyfikuje i_węzeł pliku.

23.5 Funkcja chown()

Zmienia właściciela pliku. Może być wywołana jedynie przez superużytkownika. Może zmienić grupę, do której należy plik. Praktycznie może byc wołane tylko przez superużytkownika. Podobnie jak chmod() zmienia i_węzeł pliku.

23.6 Funkcja stat()

Funkcja ta służy do pobierania statusu pliku przechowywanego w jego i-węzłe. Przekazuje takie informacje jak typ pliku, ident. właściciela pliku, prawa dostępu, rozmiar pliku, liczbę dowiązań, numer i-węzła i czasy dostępów do pliku. Działanie tej funkcji sprowadza się jedynie do przepisania zawartości i-węzła do bufora.

Wywołując podajemy nazwę pliku i bufor, do którego mają zostać zapisane informacje o pliku.

23.7 Funkcja dup ()

Tworzy duplikat istniejącego deskryptora pliku i przekazuje numer nowego deskryptora związanego z tym samym plikiem zwykłym lub specjalnym. Przy czym przekazany numer jest najniższym numerem wolnego deskryptora.

Wywołanie z parametrem będącym deskryptorem do duplikacji.

Algorytm jest prościutki:

- Sprawdza poprawność deskryptora pliku
- znajduje pierwszą wolną pozycję w tablicy deskryptorów plików i kopiuje dany deskryptor do tej pozycji.
- Zwiększa ilość dowiązań do pliku wskazywanego przez deskryptor
- Zwraca znaleziony deskryptor

23.8 Funkcja systemowa mount().

W Linuxie różne systemy plików mogą być połączone w jedną spójną strukturę - hierarchiczne drzewo katalogów. Uzyskujemy to dzięki montowaniu systemu plików do istniejacej struktury katalogów. Nowy system plików może zostać dołączony do dowolnego katalogu. Katalog ten od tej pory jest znany w systemie jako punkt zamontowania (mount point) i jego zawartość jest zakrywana przez korzeń drzewa katalogów wmontowanego systemu plików.

Zanim uzyskamy dostęp do plików np. na dyskietce, musimy ją zamontować.

Wszystkie zamontowane do tej pory systemy plików są reprezentowane przez strukturę super_block, która jest przechowywana w statycznej tablicy.

Proces wywołujący tę funkcję może zostać wstrzymany przez sterownik urządzenia ze względu na czas konieczny do dostępu do urządzenia.

Jądro w Linuxie przechowuje listę (ang. mount table) zamontowanych systemów plików. Trzymane są również dwa wskaźniki do elementów tej listy:

Wskaźnik na ostatnio zamontowany system plików i na ostatnio używany system plików, czyli na przykład na przemontowywany z nowymi flagami.

Dodatkowo jądro przechowuje listę typów systemów plików, ktore można w danym momencie zamontować. Może ona się dynamicznie zmieniać jeśli mamy skompilowane moduły do obsługi odpowiednich typów systemów plików oraz działający demon kerneld (jak działa jest opisane w treści algorytmu mount()) lub wykonamy polecenie insmod.

Systemy plików mogą być związane z urządzeniem fizycznym bądź wirtualnym.

Sposób wywołania funkcji: *mount(nazwa_urzadznia, punkt_zamonowania, typ_systemu_plikow)*

Algorytm:

Prawo do montowania ma jedynie superuser. Chyba, że zostały one w specjalny sposób nadane użytkownikowi.

Jeśli chcemy powtórnie zamontować system plików to:

- najpierw sprawdzane są zgodności odpowiednich parametrów (czy jest już zamontowany system plików, a jeśli tak, to czy z właściwymi opcjami i w zależności od tego co chcemy zrobić czy jest to w tej chwili możliwe (np. nie można przemontować na READ_ONLY file systemu z otwartymi plikami do zapisu itp.)
- później po przemontowaniu są zmieniane odpowiednie wartości flag w tablicy z superblokami (superblok danego systemu plików) i w tablicy montowania

jeśli montujemy nowy system plików:

- Sprawdza, czy dla tego systemu plików jest zdefiniowany moduł obsługi
- jeśli nie ma takiego modulu to próbuje go załadować za pomocą demona kerneld
- pobiera i-węzęł punktu zamontowania i jeśli dotyczy on katalogu, na którym nie ma jeszcze zamontowanego żadnego innego systemu plików to inicjalizuje superbloki montowanego systemu oraz głównego systemu plików
- po czym dodaje nową pozycję do listy zamontowania,
- do superbloku dowiązuje i_node przykrytego katalogu oraz
- do katalogu dowiązuje i_node katalogu korzenia montowanego systemu plików.
- Dodatkowo odbywają się jeszcze operacje związane ze zwalnianiem i-węzła i strony z pamięci, ale zostały pominięte w tym opisie.

23.9 Funkcja umount()

Działanie tej funkcji jest odwrotne do funkcji mount. Odmontowuje system plików z drzewa katalogów, jednoczesnie przykryty katalog odzyskuje swoją zawartość, po tym jak przestaje być korzeniem zamontowanego systemu plików. Można wywołać tę funkcję na dwa różne sposoby: - z podaną nazwą urządzenia do odmontowania - z podaną nazwą katalogu do zwolnienia (punktu zamontowania)

Algorytm:

- Pobierany jest i-węzęł, który musi być punktem zamontowania
- Pobierany jest i-węzęł korzenia systemu plików (odmontowywanego)
- W przypadku odmontowywania korzenia głównego systemu plików, mamy jedynie możliwość przemontowania go na READ_ONLY i wymuszenia synchronizacji
- jeśli odmontowywany system plików nie jest głównym, pobieramy superblok

- sprawdzamy, czy przypadkiem ktoś nie korzysta z tego systemu plików,
- jeśli nie, to synchronizuje bufory zajmowane przez superblok danego systemu plików,
- usuwa odpowiednią strukturę z listy zamontowanych systemów,
- na koniec wywołuje synchronizację urządzenia