System plików w systemie Linux 2.0.32

7 stycznia 1999

Magdalena Urbańska (mu171729@students.mimuw.edu.pl)
Tomasz Osman (to167297@students.mimuw.edu.pl)
Łukasz Kowalik (lk167281@students.mimuw.edu.pl)
Maciej Kurowski (mk167286@rainbow.mimuw.edu.pl)
Jan Palczewski (jp167299@rainbow.mimuw.edu.pl)

Spis rzeczy

1 Podręczna pula buforów w systemie Linux 2.0.32

autor: Maciej Kurowski

1.1 Wstęp

Podstawowym problemem na jaki napotykamy pracując z komputerem jest powolność operacji dyskowych. Pamięć masowa komputera odpowiada za archiwizację i przechowywanie trwałych danych, jest więc zwykle wielokrotnie większa niż pamięć operacyjna. Przy zachowaniu rozsądnych proprorcji cenowych pomiędzy podzespołami komputera możemy oczekiwać, że będzie ona zawsze wielokrotnie wolniejsza... .

Wszystkim co może zrobić system operacyjny by przyśpieszyć kontakt z dyskiem jest możliwie rzadkie dokonywanie operacji dyskowych. Taka jest właśnie idea buforowania. Program użytkownika w krótkim przedziale czasowym korzysta tylko z niewielkiego wycinka danych dyskowych (kilka - kilkanaście plików) możemy zatem przechowywać te dane w szybkiej pamięci operacyjnej i przy odwołaniach do nich tylko "udawać", że odwołanie jest do dysku. Bufory w pamięci operacyjnej odwzorowywane są różnowartościowo w zbiór bloków [1]urządzeń blokowych[2], z którymi pracuje komputer.

Dla wygody użytkownika cały mechanizm powinien być przezroczysty, tzn sposób odwoływania się do dowolnych danych na dysku lub w buforze powinien być jednolity. Ponadto na systemie operacyjnym spoczywaja decyza jakie dane dołączać do pamięci buforowej a jakie nie są już potrzebne.

1.2 Linux

Mechanizm buforowania w Linuxie jest zrealizowany na wysokim poziomie abstrakcji. Tych samych procedur używa się wobec wszystkich urządzeń blokowych. Podstawowym zastosowaniem modułu jest obsługa dysku więc pozwolę sobie dalej używac określenia plik zamiast: obiekt związany z urządzeniem blokowym. Mimo takiej konwencji zwracam uwagę, że prezentowane mechanizmy odnoszą się do wszystkich urządzeń blokowych.

Odwołanie do pliku możemy podzielić na następujące fazy

1. wysoki poziom: wyszukiwanie odpowiedniej i-węzła, sprawdzanie uprawnień, ..., tłumaczenie zlecenia dostępu do pliku na odwolania do poszczególnych bloków dyskowych.
2. buforowanie: sprawdzanie dostępnośći danego bloku dyskowego w puli buforów, ewentualnie zgłoszenie żądania transmisji dyskowej
3. niski poziom: szeregowanie żądań transmisji dyskowych, odwołanie do sterowników poszczególnych urządzeń

Tematem tego rozdziału jest rozwinięcie punktu drugiego tego schematu. Przedstawię wykorzystane struktury danych, i algorytmy stosowane podczas buforowania.

1.3 Nagłówek bufora

1.3.1 Struktura pamięci przydzielonej na nagłówki i bufory

Pamięć przeznaczoną na przechowywanie danych o buforach można podzielić na dwie kategorie:

- nagłówki buforów (`buffer_head`).

---

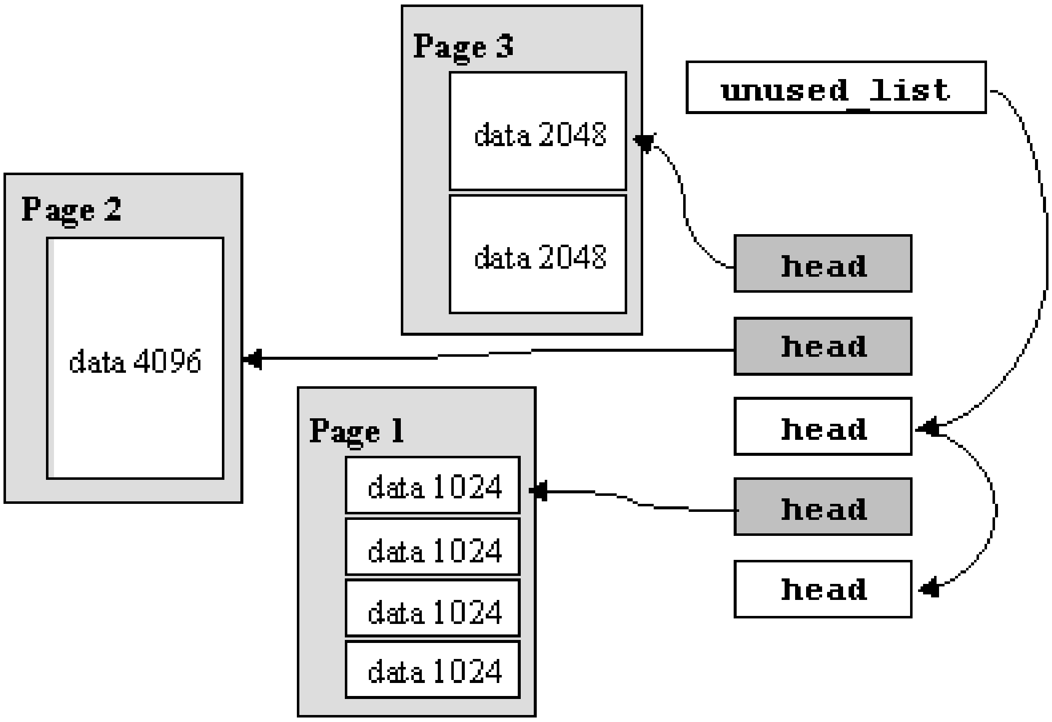
[1] Blok dyskowy: identyfikujemy go parę liczb nr urządzenia blokowego, nr bloku na urządzeniu.

[2] Urządzenia blokowe: identyfikujemy je parą liczb major - typ urządzenia, minor - nrurządzenia danego typu. W praktyce te dwie liczby kodujemy jedną typu k_dev (patrz rozdział i/o).

- strony z buforami (obszarami na buforowane dane). Każda taka strona dzielona jest na bufory - obszary równej wielkości

Nagłówki buforów są obiektami niezależnymi od konkretnych buforowanych danych. Każdy łączony jest wielokrotnie z różnymi buforami ponadto istnieje możliwość, że w danym momencie nagłówek nie jest połączony z żadnym buforem i czeka na ponowne użycie poza omawianymi dalej wyjątkami należy wtedy do listy wolnych nagłówków buforów - `unused_list`.

Struktura przydziału pamięci na nagłówki i bufory



Rzuca się w oczy pewne ograniczenie związane z taką organizacją: rozmiar ramki pamięci musi być wielokrotnością rozmiaru bloku urządzenia. W szczególności nie można używać urządzeń o blokach większych niż strony pamięci. Standardowo strona pamięci ma 4096 bajtów a bloki są rozmiarów 512, 1024, 2048 lub 4096 bajtów (rozmiar 8172 jest nie używany).Dodatkowo mamy tu do czynienia z fragmentacją wewnętrzną pamięci, gdyż warunkiem koniecznym do zwolnienia strony jest odblokowanie wszystkich buforów izakończenie na nich operacji wejścia/wyjścia.

1.3.2 Stan bufora

Podczas działania systemu bufor może znajdować się w różnych stanach co wpływa znacząco na jego funkcjonalność. System zapisuje stan bufora w nagłówku przez ustawienie odpowiednich znaczników:

pole b_dev

`BF_FREE` nagłówek wolny nie związany z buforem

bity pola b_state:

`BH_Uptodate` dane w buforze pokrywają się z danymi na dysku

`BH_Dirty` dane w buforze różnią się od danych na dysku, bufor przeznaczony do zapisu, bufory w takim stanie będę nazywał brudnymi.

`BH_Lock` na buforze trwa operacja wejśćia/wyjścia

`BH_Req` bufor jest unieważniony, transmisja dyskowa się nie powiodła

`BH_Protected` bufor zawiera specjelne dane i nie może być zwolniony (używany np przy obsłudze ram-dysku)

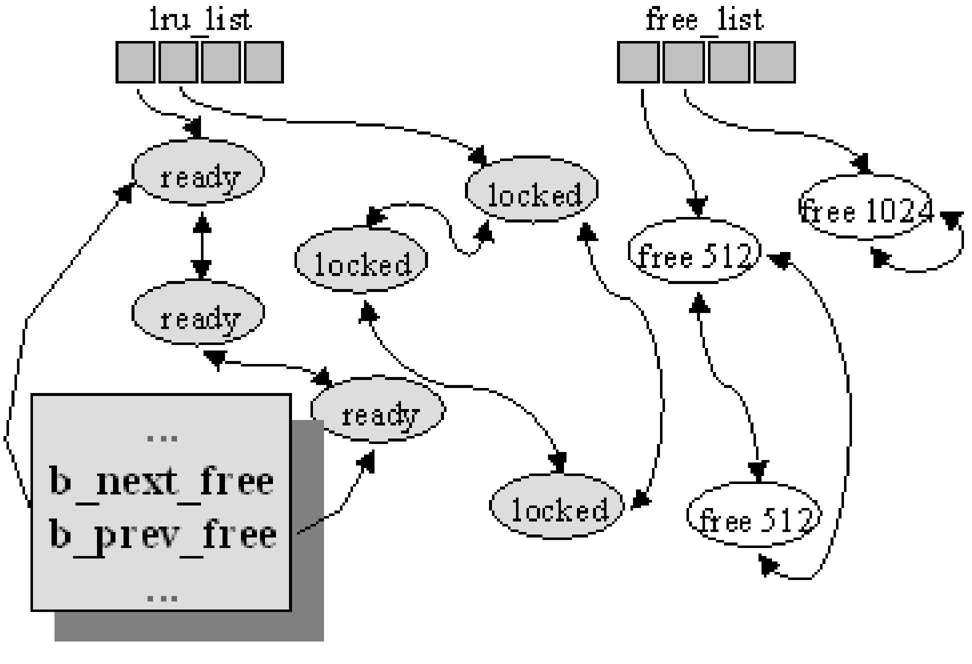
`BH_FreeOnIO` nagłówek bufora ma zostać zwolniony po zakończeniu operacji wejścia/wyjścia

Aby ułatwić niektóre operacje na buforach wymagających szybkiego wyszukania bufora w odpowiednim stanie, nagłówki buforów o podobnej funkcjonalności połączone są w cykliczne listy LRU (least recently used). Bufory na listach posortowane są ze względu na czas ostatnich odwiedzin. Liczba list LRU maleje wraz z kolejnymi wersjami jądra systemu Linux. W opisywanej wersji 2.0.32 mamy:

- `BUF_LOCKED` lista nagłówków buforów na których trwa operacja wejscia/wyjścia (stan `BH_Locked`)
- `BUF_LOCKED1` lista nagłówków buforów specjalnych, zawierającymi np inody, superbloki itp
- `BUF_DIRTY` lista nagłówków buforów brudnych (w stanie `BH_Dirty`)
- `BUF_CLEAN` pozostałe nagłówki skojarzone z buforami

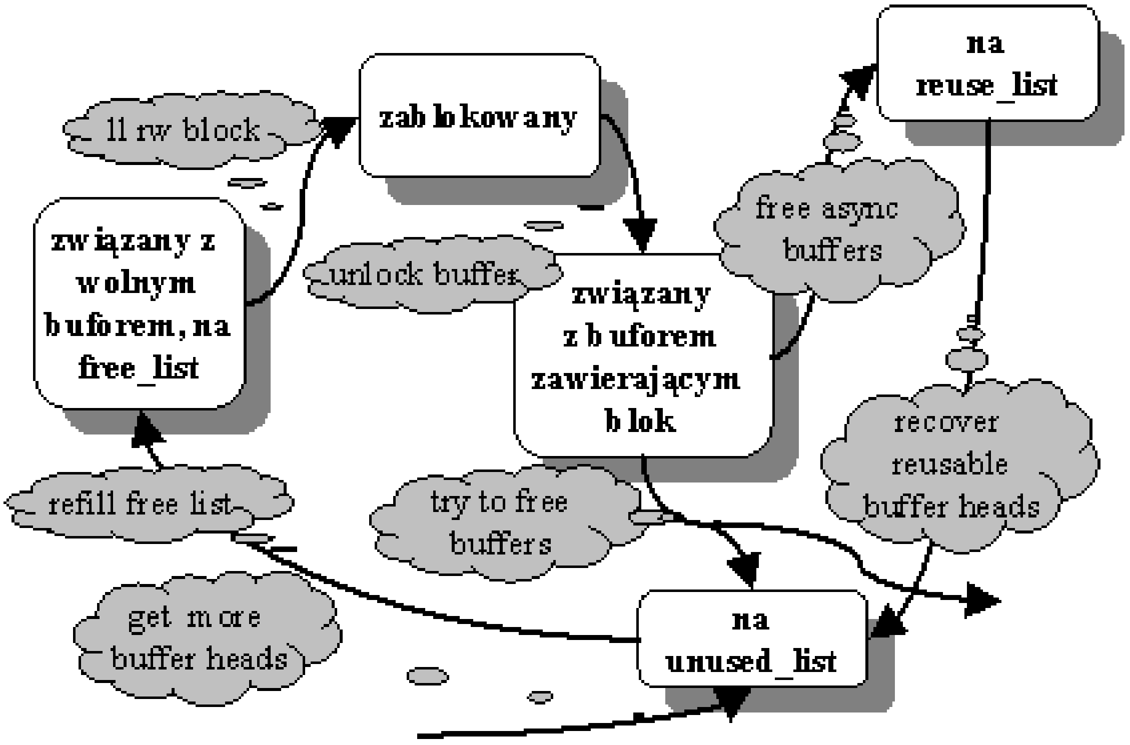
Nagłówki skojażone z wolnymi buforami znajdują się na jednej z 4 list wolnych buforów. Na każdej liście znajdują się bufory równych rozmiarów. Oczywiście nagłówek nie może znajdować się jednocześnie na liście wolnych buforów i liście LRU.

Listy LRU i wolnych buforów



Konsekwentna ogranizacja list LRU jest w praktyce trudna do realizacji. Wyobraźmy sobie sytuację, w której nadchodzi przerwanie końca transmisji dyskowej. Obsługując przerwanie zmieniamy stan bufora z `BH_Locked` na `BH_Uptodate`. Nie możemy jednak przenieść bufora do odpowiedniej mu listy LRU, gdyż jej integralność mołaby zostać zaburzona (być może gdy nadeszło przerwanie była właśnie w trakcie modyfikacji). Autorzy systemu pogodzili się z konieczną niekonsekwencją. Powyższa niezgodność zostanie usunięta przy okazji przez jedną z procedur przeglądających listy LRU. Przedstawiony poniżej diagram przedstawia graf zmian stanu bufora. Poszczególne przejścia w grafie są wyjaśnione dokładnie w kolejnych rozdziałach

Schemat ELH nagłówka bufora

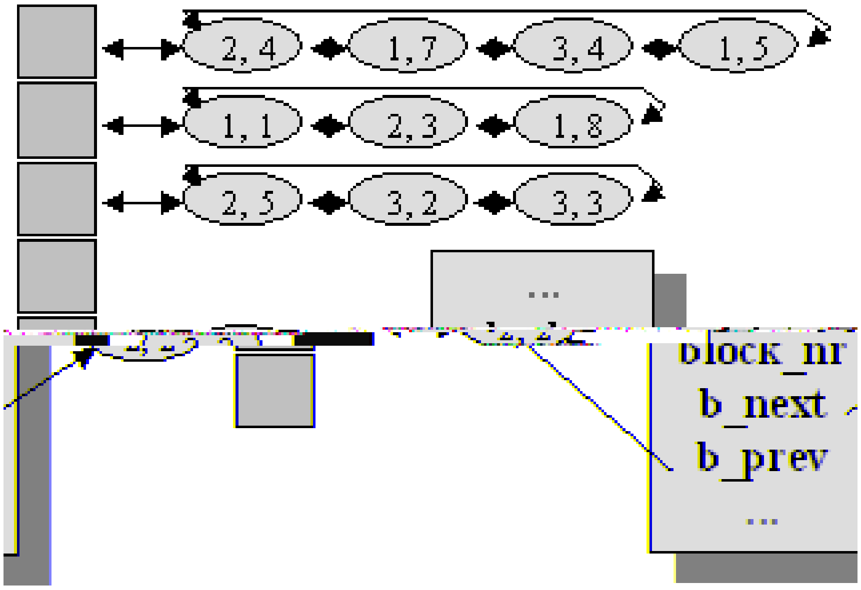


1.4 Algorytmy

1.4.1 Wyszukiwanie bufora get_blk

Podczas każdego odwołania użytkownika do bloku dyskowego wymagane jest sprawdzenie, czy jest on zbuforowany w pamięci. Może się bowiem zdarzyć, że zapisana na dysku wersja bloku nie jest aktualna. Wyszukiwanie bufora odpowiadającego danemu blokowi jest często używaną operacją i musi być możliwie jak najszybsze. Wykorzystywana jest do tego celu tablica haszująca, gdzie kluczem jest numer urządzenia i numer bloku. Kolizje w tablicy rozwiązywane są metodą list cyklicznych. Mamy więc tablicę dwukierunkowych list cyklicznych, na każdej liście znajdują się nagłówki buforów o równej wartości funkcji haszującej.

Tablica haszująca



Schemat algorytmu get_blk:

- obliczamy wartość funkcji haszującej i wybieramy dobrą listę (`get_hash_table`)
- przeglądamy listę w poszukiwaniu bufora zawierającego dany blok
- Intuicja nakazuje aby usuwać bufory dawno nie używane więc zaznaczamy bufor jako odwiedzony przesuwając go na koniec jego listy LRU i stęplując aktualnym czasem (pole `b_lru_time`)

W wypadku nie znalezienia bloku w pamięci buforowej tworzymy pusty bufor o podanym rozmiarze. Od tej pory będzie on związany z danym blokiem. Operacja tworzenia bufora odbywa się w krokach:

- wybieramy pusty nagłówek bufora z listy nieużywanych nagłówków
- wybieramy pusty bufor z listy pustych buforów o odpowiednim rozmiarze
- wypełniamy nagłówek bufora (nr bloku, nr urządzenia, rozmiar bloku, wskaźnik do bufora)
- wstawiamy nagłówek bufora do struktur danych opisujących zajęte bufory: tablicy haszującej, listy LRU, listy buforów na stronie pamięci

W opisie pominąłem komplikacje związane z obsługą sytuacji wyjątkowych gdy brakuje wolnych nagłówków buforów lub buforów odpowiedniej wielkości. Omówię je w następnym punkcie poswięconym odnawianiu zasobów. Przeanalizujmy prosty przykład: dla tablicy haszującej wielkości 512 i 1024 buforów w pamięci średnia długość listy wynosi 2, zatem aby stwierdzić czy dany blok jest zbuforowany potrzeba sprawdzić tylko dwa nagłówki buforów. W porównaniu do czasu trwania transmisji dyskowej jest to czas pomijalny.

1.4.2 Odnawianie zasobów

Jak wspomniałem we wstępie jednym z wymagań jakie stawiamy modułowi zarządzającemu buforami jest dynamiczne zarządzanie pamięcią. Termin ten oznacza w praktyce tyle, że system potrafi dostosować liczbę zaallokowanych buforów do faktycznych potrzeb użytkownika. Nie dopuszczamy zarówno sytuacji, w której zaallokowana jest duża liczba buforów a wykorzystujemy tylko niewielki ich procent jak i brak wolnych buforów pomimo wystarczającej ilości pamięci operacyjnej. Procedura pozyskująca bufory danego rozmiaru (refill_freelist) może być czasochłonna ale podczas jednego wywołania tworzy wiele buforów co gwarantuje nam, że nie jest wywoływana często.

Zródła pozyskiwania nowych buforów

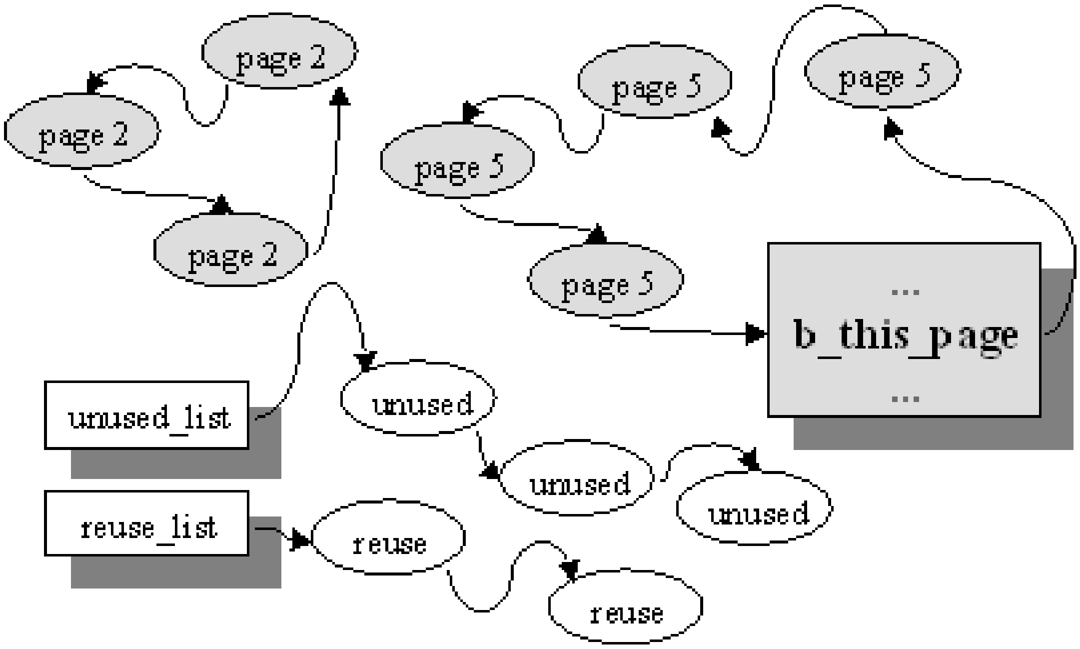
- przydział nowych stron pamięci z niskim priorytetem. Za priorytet przymuje się stałą oznaczającą oczekiwaną liczbę wolnych stron po zakończeniu algorytmu. Im ta liczba jest mniejsza tym większy jest priorytet operacji.
- zwalnianie zajętych buforów zgodnie z algorytmem LRU tzn najdawniej używane bufory zwalniane są jako pierwsze. Pod uwagę bierzemy jedynie bufory danego rozmiaru. Oczywiście nie wolno nam zwalniać buforów do których trwa transmisja dyskowa ani buforów brudnych bez uprzedniego ich zapisania. Dokładnie: z każdej z list LRU wybieramy kandydata do zwolnienia. Zwalniamy najdawniej odwiedzany bufor spośród kandydatów sprawdzamy stemple czasowe pole `b_lru_time`.
- przydział nowych stron pamięci z wysokim priorytetem
- zapisanie brudnych buforów na dysk

W trakcie przydziału nowych stron na bufory może zdarzyć się, sytuacja w której zabraknie wolnych naglówków buforów. Są one tworzone przez procedurę `get_more_buffer_heads` która przydziela pamięć i umieszcza na liście `unused_list` skąd pobierają je odpowiednie procedury.

1.4.3 Zwalnianie zasobów

W celu zapewnienia równowagi pomiędzy liczbą buforów różnych rozmiarów w Linuxie 2.0.32 stosuje się najprostsze środki. Zrezygnowano z rozbudowanych algorytmów postarzania buforów stosowanych w starszych Linuxach. Podczas przeglądania list LRU w wyborze kandydata do zwolnienia (`refill_freelist`, `find_candidate`) napotykane bufory innych rozmiarów traktujemy jako niepotrzebne i staramy się je zwolnić. Operacja kończy się powodzeniem, jeżeli wszystkie bufory na stronie pamięci mogą zostać zwolnione (nie są brudne ani zablokowane). Aby możliwe było przeglądanie buforów znajdujących się na stronie pamięci utrzymywana jest dodatkowa struktura. Nagłówki buforów znajdujących się na jednej stronie powiązane są w jednokierunkową listę cykliczną.

Listy nagłówków buforów, które znajdują się na jednej stronie pamięci



- przeglądamy listę LRU poczynając od najstarszych buforów w poszukiwaniu bufora, który można zwolnić (`find_candidate`)
- napotykamy bufor o innym rozmiarze
- przeglądamy wszystkie bufory dzielące z nim stronę pamięci
- jeśli wszystkie można zwolnić, zwalniamy całą stronę

Okazuje się, że taki algorytm jest wystarczająco dobrą heurystyką, ponieważ wraz z liczbą buforów danego rozmiaru rośnie prawdopodobieństwo, że napotkamy stronę z takimi buforami, którą uda się zwolnić. Odzyskane nagłówki zwolnionych buforów, które nie są już związane z buforami umieszczamy na liście `unused_list` a w wypadku jej przepełnienia zwalniamy. Utrzymywanie listy wolnych nagłówków zwalnia nas z konieczności każdorazowego przydziału pamięci przy tworzeniu nowego bufora. Istnieje również inny scenariusz zwalniania bufora. Realizowany jest w takich przypadkach, w ktorych nie chcemy, aby wczytywane dane były buforowane (na przykład podczas odczytywania strony pamieci wirtualnej z dysku). System działa wtedy zgodnie z algorytmem:

- przydzielamy stronę pamięci na którą wczytamy dane
- tworzymy "sztuczne" bufory na pobranej stronie. Ich nagłówki nie są wstawiane, do żadnej z opisywanych struktur danych
- nagłówki buforów mają ustawiony znacznik `BF_FreeOnIO` oznaczający konieczność zwolnienia nagłówków buforów zaraz po końcu transmisji
- inicjowana jest transmisja a dysku
- odbieramy przerwanie końca transmisji dyskowej. Wolne naglówki buforów nie mogą zostać umieszczone na liście buforów wolnych `unused_list` aby nie zaburzyć jej integralności. Umiesczamy je na specjalnej liście `reuse_list`.

- przenoszenie nagłówków z listy `reuse_list` na `unused_list` odbywa się poza obsługą przerwania (`recover_reusable_buffer_heads`). Przeglądanie listy reuse_list odbywa się w bezpieczny sposób tzn tak, że ewentualne dopisanie do niej nowych buforów w jego trakcie nie zaburzy jej integralności

1.4.4 Zapis brudnych buforów

Do obsługi puli buforów nie wystarczy prosty interfejs. Zarządzanie nią opiera się na heurystykach. System zlicza bufory znajdujące się w różnych stanach i dba o zachowanie odpowiednich proporcji (np stosunku pomiędzy np liczbą buforów brudnych i pozostałych) - przy niekorzystnym ich układzie niektóre operacje mogłyby stać się kosztowne czasowo. Zapisem brudnych buforów na dysk zajmuje się proces `bd_flush`. Jest on budzony, gdy liczba brudnych buforów osiągnie odpowiedni poziom (zdefinowany w tablicy `bdf_prm`).

- sprawdzamy nagłówki z listy nagłówków buforów brudnych
- dla wszystkich buforów, których czas żądania zapisu (`b_flushtime`) jest starszy niż czas aktualny inicjalizujemy transmisję danych na dysk

System dostarcza również mechanizmów do bardziej selektywnej synchronizacji. Użytkownik musi mieć możliwość wymuszenia jej gdy chce np wyjąć dyskietkę ze stacji dysków. Istnieje możliwość zapisu na dysk wszystkich buforów związanych z danym urządzeniem (`sync_buffers`). Przeglądane są wtedy wszystkie listy LRU.

1.4.5 Wczytywanie bloku

Pobieranie bloku jest podstawową procedurą interfejsu udostępnianego przez moduł buforujący innym modułom. Korzysta z procedury wyszukiwania bloku w pamięci. Przedstawię schemat funkcji bread:

- dany jest numer urządzenia blokowego i numer bloku na tym urządzeniu
- pobieramy nagłówek bufora odpowiadający danemu blokowi, jeśli odpowiedniego bufora nie ma w pamięci zostanie stworzony nowy nagłówek i pusty bufor
- jeśli nagłówek jest nowo stworzony inicjujemy transmisję z dysku do bufora
- czekamy na jej zakończenie transmisji
- transmisja mogła się nie powieść więc ponownie sprawdzamy aktualność danych w buforze

Funkcja breada ma identyczne zadanie jak bread różni się tylko zasiegiem. Jej działanie oparte jest na obserwacji, że podczas pracy z plikiem często ma miejsce sytuacja w której odczytujemy kolejne bloki pliku. Funkcja stara się przewidzieć liczbę kolejnych bloków jakiej użyje uzytkownik i odczytuje je z wyprzedzeniem zyskując w ten sposób na czasie. Wnioskowanie odbywa się na podstawie dodatkowych parametrów pozycji w pliku i jego rozmiaru. Przyjęto prostą heurystykę: wczytujemy kolejnych 16 bloków pliku o ile nie przekracza to jego rozmiaru.

1.4.6 Zakończenie pracy z blokiem

Po zakończeniu pracy z blokiem konieczne jest wykonanie na buforze czynności administracyjnych. W takiej sytuacji system oczekuje od użytkownika wykonania operacji brelse. Prawdopodobnie możliwe jest takie zaprojektowanie tej operacji by była wykonywana automatycznie na wszystkich odpowiednio starych buforach ale taka heurystyka nie przybliża dostatecznie dobrze zbioru aktualnie używanych bloków. Niektóre z nich są używane przez cały czas działania systemu, z innych zaś korzystamy tulko podczas jednorazowego odczytu więc czas ostatniego użycia nie jest tu dobrym kryterium.

Schemat procedury brelse:

- czekamy aż bufor będzie dostępny (odblokowany)
- o ile bufor jest brudny ustawiamy czas żądania zapisu (`b_flushtime`). Transmisja dyskowa zostanie zainicjalizowana podczas pierwszej operacji zapisu buforów późniejszej niż ustawiony żądany czas. Opóźnienie zapisu jest celowe. Podczas częstej modyfikacji jednego bloku unikamy niepotrzebnego zapisu na dysk stanów pośrednich. Zapis na dysk będzie z dużym prawdopodobieństwem dotczył jego wersji ostatecznej
- sprawdzamy, czy bufor znajduje się na poprawnej liście LRU i jeżeli nie wstawiamy go na właściwą
- jeżeli liczba brudnych buforów jest dostatecznie duża budzimy proces zapisujący bloki na dysk.

1.5 Podsumowanie

Buforowanie jest ważnym systemu operacyjnego i jak wynika z testów drastycznie zwieksza komfort pracy z komputerem. Szybko rosnące parametry pamięci operacyjnej i dysków spawiają, że moduł buforujący poddawany jest wraz z kolejnymi wersjami Linuxa ustawicznym zmianom. Zwiększanie ilości dostępnej pamięci umożliwia przechowywanie dużej liczby buforów. Skłania to do zaniechania złożonych algorytmów selekcji bloków przechowywanych w pamięci. Rezygnuje się stopniowo np ze mechanizmów postarzania buforów. Wydaje mi się, że jedną z pozostałości historycznych są również listy LRU. Ich realizacja jest niekonsekwentna i korzyści wypływające z ich użycia nie usprawiedliwiają narzutu związanego z ich obsługą.

2 Wewnętrzna reprezentacja plików w Linuxie

autorka: Magdalena Urbańska

2.1 Wstęp

Linux obsługuje wiele różnych systemów plików m.in.: ext, ext2, msdos, minix, vfat, iso9660. Dostęp do plików realizowany jest poprzez uniwersalny interfejs, czyli **VFS - VIRTUAL FILE SYSTEM**. VFS udostępnia jednolite funkcje na zewnątrz, a następnie sam wywołuje ich odpowiedniki dla właściwego systemu plików. Ponadto VFS przechowuje własne uniwersalne struktury danych oraz dostarcza pewne standardowe funkcje niezaimplementowane dla poszczególnych systemów (zazwyczaj żaden system nie ma własnej funkcji *lseek()*). Dzięki istnieniu VFS szczegóły budowy konkretnych systemów plików mogą być ukryte przed użytkownikami.

W Linuxie systemy plików są powiązane w hierarchiczną strukturę, z kolei system plików sam stanowi hierarchiczną strukturę, do której należą pliki różnych typów. Mogą to być:

- plik zwykły
- katalog
- urządzenie blokowe
- urządzenie znakowe
- łącze nazwane (FIFO)
- dowiązanie symboliczne (symlink)
- gniazdo (socket)

Plik jest widziany przez użytkownika jako strumień bajtów. System musi jednak przechowywać pewne dodatkowe informacje z nim związane np. właściciela, prawa dostępu, rozmiar itp. Służy do tego struktura **i-węzła** (ang. inode - index node). W Linuxie mamy do czynienia z dwoma rodzajami i-węzłów: na dysku (bądź innym nośniku, który będę umownie nazywać dyskiem) i w pamięci. I-węzły w pamięci są kopiami i-węzłów dyskowych uzupełnionymi o pewne potrzebne dane. Dzięki nim dostęp do plików jest szybszy. (W zasadzie nie jest wymagane, żeby każdy system plików implementował i-węzeł dyskowy w tradycyjnej Unixowej postaci, plik może być reprezentowany przez zupełnie inne struktury, ale z punktu widzenia VFS nie ma to żadnego znaczenia, gdyż komunikuje się on z wszystkimi systemami poprzez ujednolicony interfejs.)

2.2 Reprezentacja systemu plików w jądrze

Fizyczna struktura poszczególnych systemów plików może wyglądać inaczej, jednakże ich reprezentacja w jądrze jest jednolita.

Jak wygląda tworzenie tej reprezentacji dla nowo podłączanego systemu plików w Linuxie?

Najpierw system musi zostać zarejestrowany. Dokonuje się to poprzez funkcję *register_file_system*(&(struct file_system_type)) Argumentem tej funkcji jest struktura zawierająca :

- nazwę systemu plików np. ext2
- informację, czy system wymaga urządzenia (0 lub 1)
- wskaźnik do funkcji *read_super* np. *ext2_read_super*

(Struktury file_system_type są powiązane w jądrze w listę jednokierunkową)
Funkcja *read_super* tworzy interfejs między danym systemem plików a VFS. Dzięki niej VFS uzyskuje podstawowe informacje o systemie.
Żeby można było dostępować do poszczególnych plików, po zarejestrowaniu systemu należy go jeszcze zamontować (funkcja *mount()* lub *mount_root()*).
W czasie montowania w pamięci zostaje utworzony i-węzeł korzenia systemu. Każdy zamontowany system plików jest w jądrze reprezentowany przez strukturę super_block. Struktury te są przechowywane w tablicy super_blocks o rozmiarze NR_SUPER (w tej wersji Linuxa równym 64).

Struktura super_block (blok identyfikacyjny w pamięci)

Struktura super_block zostaje podczas montowania zainicjowana poprzez wywołanie funkcji *read_super* dla danego systemu.
Zawiera ona:

- numer logicznego urządzenia danego systemu plików
- rozmiar bloku w systemie
- prawa dostępu
- flagę informującą, czy struktura została zmodyfikowana
- czas ostatniej modyfikacji
- identyfikator systemu plików
- wskaźnik do i-węzła będącego punktem zamontowania systemu (mount point)
- wskaźnik do i-węzła będącego korzeniem systemu (root inode)

- typ systemu (wskaźnik do struktury file_system_type)
- struktury zapewniające synchronizację dostępu
- informację specyficzną dla danego systemu np. ext2_sb_info

Poza powyższymi struktura super_block zawiera jeszcze jedno istotne pole, jest to wskaźnik do struktury **super_operations**. Stanowi ona wektor funkcji umożliwia-jących dostęp do systemu plików. Dostarczone funkcje są wykorzystywane przez VFS. Pozwalają one przechodzić między specyficzną dla systemu reprezentacją i-węzla i bloku identyfikacyjnego a ich ogólną postacią w pamięci.
Super_operations zawiera wskaźniki do następujących funkcji:

- *read_inode(inode)* - odczytywanie i kopiowanie do pamięci i-węzła dyskowego
- *write_inode(inode)* - zapisywanie i-węzła na dysk
- *put_inode(inode)* - usunięcie fizyczne i-węzła i zwolnienie jego bloków dyskowych, o ile liczba dowiązań jest zerowa
- *write_super(sb)* - zapisanie na dysk informacji o strukturze super_block
- *put_super(sb)* - zwolnienie struktury super_block (wywoływane przy odmontowaniu systemu)
- *notify_change(inode,iattr)* - przekazanie w strukturze iattr informacji o zmia-nach dokonanych w i-węźle przez funkcje systemowe
- *statfs* - pobranie informacji o systemie (np. ilość wolnych bloków, rozmiar bloku itp.)
- *remount_fs* - zmiana statusu systemu

Jeżeli jakaś funkcja nie jest dla danego systemu plików zaimplementowana, w odpowiednim miejscu znajduje się po prostu NULL.

2.2.1 Struktura pliku

W Linuxie z plikiem związanych jest kilka struktur danych. Po pierwsze, jak już to zostało powiedziane we wstępie, plikowi odpowiada dokładnie jeden i-węzeł na dysku. I-węzeł stanowi podstawowe źródło informacji o pliku: jego właścicielu, typie, rozmiarze, rozmieszczeniu jego zawartości na dysku (lub innym nośniku). Aby przyspieszyć dostęp do plików, jądro przechowuje w pamięci pewną pulę kopii i-węzłów dyskowych w strukturach również zwanych i-węzłami. Ponadto istnieje struktura file, która reprezentuje plik z perspektywy procesu (użytkownika).

2.2.2 Plik z perspektywy procesu

Można się zastanawiać, jaki jest cel wprowadzania dodatkowej struktury reprezentującej plik. Chodzi o umożliwienie wykonywania operacji na pliku wielu proce-som jednocześnie. Istnienie struktury file rozwiązuje problem sychronizacji, poz-wala realizować dostęp dzielony.
Najważniejsze pola struktury file:

- tryb otwarcia pliku oraz pewne flagi ustawiane przy wywołaniu *open()* dla danego pliku np. O_APPEND, O_NDELAY (niektóre z tych flag mogą być modyfikowane funkcją *fcntl()*)
- pozycja, od której rozpocznie się następna operacja wejścia/wyjścia
- licznik otwarć, czyli liczba wystąpień wskaźnika do danej struktury w tablicach deskryptorów procesów
- wskaźnik do odpowiedniego i-węzła w pamięci

- wskaźnik do struktury file_operations

Proces otwierając plik otrzymuje dla niego deskryptor. W tablicy indeksowanej deskryptorami przechowuje on wskaźniki do odpowiednich struktur typu file w pamięci. (Rozmiar tej tablicy wynosi NR_OPEN = 256). Z tego wynika, że z jednym plikiem może być związanych kilka struktur file. Również do danej struktury może odnosić się wiele deskryptorów danego procesu, a ponieważ deskryptory otwartych plików są dziedziczone przez procesy-dzieci, wskaźnik do tej samej struktury może posiadać wiele procesów na raz. Licznik otwarć jest przy otwieraniu pliku ustawiany na 1. Przy każdym kopiowaniu deskryptora (wywołanie funkcji systemowych *dup, dup2* lub *fork*) jest on zwiększany o 1, zaś zmniejszany przy każdym zamknięciu pliku (wywołanie *close, _exit, exec*). Struktura zostaje zwolniona, gdy licznik otwarć wyzeruje się. Struktury file są powiązane w cykliczną listę dwukierunkową file_table, do której dostępujemy poprzez wskaźnik first_file. Rozmiar listy jest ograniczony przez stałą NR_FILES = 1024. Struktura **file_operations** zawiera wskaźniki do funkcji (charakterystycznych dla odpowiedniego systemu plików i typu pliku) pozwalających operować na pliku.
Są to m.in.:

- *lseek* - zmiana pozycji w pliku
- *read* - odczytanie pewnej ilości bajtów z pliku
- *write* - zapisanie pewnej ilości bajtów do pliku
- *readdir* - odczytanie kolejnej pozycji w katalogu
- *open* - otwieranie pliku
- *release* - zwalnianie struktury file
- *mmap* - mapowanie części pliku do przestrzeni adresowej użytkownika

2.2.3 I-węzeł w pamięci i na dysku

Struktura inode w pamięci jest jednakowa dla wszystkich plików, bez względu na system plików, do którego należą.
Najważniejsze pola:

- logiczny numer urządzenia, na którym znajduje się plik
- numer i-węzła na dysku
- prawa dostępu i typ pliku (możliwe typy: zwykły, katalog, łacze nazwane, dowiązanie symboliczne (symlink), urządzenie blokowe, urządzenie znakowe; 0 oznacza i-węzeł wolny)

  prawa dostępu dotyczą czytania, pisania i wykonywania dla właściciela, grupy właściciela i pozostałych użytkowników
- identyfikator właściciela i grupy właściciela
- liczba dowiązań (różnych nazw tego pliku - hard links)
- rozmiar w bajtach (o 1 większy od najwyższego numeru bajtu danych w pliku)
- ilość bloków dyskowych zajmowanych przez plik
- rozmiar bloku
- czas utworzenia, ostatniego dostępu i ostatniej modyfikacji pliku
- licznik odwołań do i-węzła (0 - gdy wolny) - liczba aktywnych wcieleń pliku

- flaga informująca, czy i-węzeł w pamięci różni się od dyskowego
- liczba uprawionych do zapisu w pliku
- wskaźnik do odpowiedniej struktury super_block
- wskaźnik do struktury inode_operations
- wskaźnik do i-węzła korzenia zamontowanego systemu plików, o ile i-węzeł jest punktem zamontowania
- struktury zapewniające synchronizację dostępu do i-węzła (czy jest założona blokada na plik, kolejka czekających na jej zdjęcie, semafor itd.)
- informacja charakterystyczna dla danego systemu plików np. ext2_inode_info - m.in. tablica numerów bloków dyskowych przechowujących zawartość pliku - i_data
- wskaźniki do innych i-węzłów w pamięci

I-węzeł dyskowy może mieć w pamięci **tylko jedną** kopię. Maksymalna ilość kopii wynosi 3072 (stała NR_INODE). Liczbę kopii znajdujących się aktualnie w pamięci podaje zmienna nr_inodes, zaś i-węzłów wolnych zmienna - nr_free_inodes. Nowe i-węzły generuje funkcja *grow_inodes* wywoływana zawsze, gdy mniej niż $\frac{1}{4}$ i-węzłów jest wolnych (i, naturalnie, ich liczba nie przekracza masymalnej
dopuszczalnej ilości).
Wszystkie i-węzły w pamięci są powiązane w dwukierunkową listę cykliczną. Dostęp do pierwszego wolnego i-węzła jest realizowany poprzez wskaźnik first_inode. Wolne i-węzły wstawia się zawsze na początek listy. Poruszanie się po liście umożliwiają wskaźniki: i_next, i_prev. Dla zwiększenia efektywności dostępu
i-węzły przechowuje się także w tablicy haszującej hash_table[NR_HASH]
(NR_HASH=512), w której kolizje są rozwiązywane za pomocą dwukierunkowych list cyklicznych. Funkcja haszująca ma postać:

h(dev,i) = (dev ^ i) / NR_HASH ,
gdzie dev - numer urządzenia, i - numer i-węzła na dysku.

Do poruszania się po tablicy haszującej służą wskaźniki hash_next i hash_prev.
Struktura inode_operations zawiera wskaźniki do pozostałych funkcji służacych do pracy z plikami danego systemu, które nie znalazły się w strukturze file_operations. Przyczyną takiego podziału jest fakt, że część funkcji musi modyfikować strukturę file np. *read* zmienia pozycję w pliku.
Niektóre funkcje z inode_operations:

- *create* - utworzenie zwykłego pliku
- *link* - utworzenie dowiązania do pliku
- *unlink* - usunięcie dowiązania
- *lookup* - znalezienie i-węzła odpowiadającego plikowi o podanej nazwie w konkretnym katalogu
- *symlink* - utworzenie symbolicznego dowiązania do pliku
- *mkdir* - utworzenie i-węzła katalogu
- *rmdir* - usunięcie i-węzła katalogu
- *mknod* - utwozenie nowego i-węzła specjalnego typu
- *rename* - zmiana nazwy pliku

- *truncate* - zmiana długości pliku
- *bmap* - przekształcenie logicznego numeru bloku na adres fizyczny

Reprezentacja i-węzła na dysku ściśle zależy od konkretnego systemu plików. W systemie EXT2 jest to struktura ext2_inode. Zawiera ona większość informacji znajdujących się w strukturze inode, czyli typ pliku, prawa dostępu, rozmiar, liczba dowiązań, itd. oraz dane właściwe dla danego systemu. Adresy bloków dyskowych składających się na plik są przechowywane w tablicy i_block.

2.2.4 Przechowywanie zawartości zwykłego pliku na dysku w EXT2

Jest oczywiste, że plik nie może zajmować ciągłego obszaru na dysku. Prowadziłoby to do dużej fragmentacji zewnętrznej i podnosiło znacznie koszt zwiększania rozmiaru pliku. Stąd pomysł rozproszenia bloków tworzących plik. Numery bloków można by przechowywać w liście liniowej. Obsługa takiej listy jest jednak mało efektywna, a ponadto ten sposób pamiętania bloków wymuszałby bądź wprowadzenie zmiennego rozmiaru i-węzła bądź dopuszczenie bardzo dużego ograniczenia górnego na rozmiar pliku. W Linuxie problem ten rozwiązano za pomocą adresowania pośredniego. W systemie EXT2 tablice numerów bloków mają rozmiar 15 (stała EXT2_N_BLOCKS). Z tego pierwszych 12 numerów wskazuje na bloki bezpośrednie - zawierające dane. Na kolejnej pozycji znajduje się numer bloku pojedynczego pośredniego - zawiera on numery bloków bezpośrednich. Na blok podwójny pośredni składają się numery bloków pojedynczych pośrednich, a na potrójny pośredni - podwójnych pośrednich.
Warto zauważyć, że niektóre pozycje w tablicy numerów mogą być równe 0.
Oznacza to, że odpowiadające im bloki logiczne nie zawierają danych. Nie zajmują one miejsca na dysku. (Taka sytuacja może powstać w wyniku wywołania funkcji systemowych *lseeek()* lub *write()*).
Przy minimalnym rozmiarze bloku w EXT2 wynoszącym 1024 bajty (numer bloku zajmuje 32 bity) metoda ta pozwala zaadresować ponad 16GB (w praktyce jednak, ponieważ rozmiar pliku jest przechowywany na 32 bitach, rozmiar pliku jest i tak ograniczony do 4GB - 2^{32}).

Przez procesy plik i tak jest widziany jako ciągły strumień bajtów zaczynający się od adresu zero. Odwołanie do bajtu następuje poprzez logiczny numer bloku, w którym on się znajduje. Zadaniem funkcji *bmap* jest przetłumaczenie tego numeru na odpowiedni numer bloku na dysku.

2.3 Algorytmy związane z i-węzłami

2.3.1 Pobieranie i-węzła - iget

Algorytm `iget` przydziela i-węzeł w pamięci.
Parametry:

- wskaźnik do odpowiedniej struktury super_block
- numer i-węzła na dysku
- czy i-węzeł jest punktem zamontowania

Schemat działania:

1. Szukaj i-węzła w tablicy haszującej. Jeśli został znaleziony, skocz do 2, wpp. skocz do 3.
2. I-węzeł znajduje się w pamięci:
   - Jeżeli i-węzeł jest właśnie przygotowywany do ponownego użycia (*clear_inode*), zaśnij, a po przebudzeniu skocz do 1.
   - W przeciwnym razie:
     - zwiększ licznik odwołań (jeżeli jest on równy 1, zmniejsz liczbę wolnych)

    - jeżeli i-węzeł jest zablokowany, czekaj na zdjęcie blokady
    - jeżeli i-węzeł zmienił się, skocz do 1.
  - Jeżeli i-węzeł jest punktem zamontowania, zwolnij go i pobierz i-węzeł korzenia zamontowanego systemu.
  - Zwróć i-węzeł

3. I-węzła nie ma w pamięci:
   - Spróbuj pobrać wolny i-węzeł (*get_ empty_ inode*) tzn. poszukaj na liście wolnego. Jeżeli nie ma wolnych, ale nie przekroczyliśmy limitu, spróbuj zaalokować nową stronę w pamięci i stworzyć wolne i-węzły.
     - Jest wolny i-węzeł
       wypełnij odpowiednio strukturę inode
       umieść i-węzeł na końcu listy, przed pierwszym wolnym
       wstaw i-węzeł do tablicy haszującej
       odczytaj i-węzeł z dysku (funkcja *read_ inode*)
       zwróć i-węzeł
     - Nie ma wolnego i-węzła
       zwróć NULL

2.3.2 Zwalnianie i-węzła - iput

Algorytm `iput` zwalnia i-węzeł w pamięci.
Parametry:

- wskaźnik do i-węzła w pamięci

Schemat działania:

1. jeżeli i-węzeł jest zablokowany, czekaj na zdjęcie blokady
2. Zmniejsz licznik odwołań
3. Jeżeli licznik odwołań jest większy niż 1, wyjdź
4. Wpp. wykonaj *put_ inode*
   jeśli nie ma dowiązań do pliku, zostanie on
   usunięty fizycznie - w tej sytuacji wyjdź.
5. Jeżeli i-węzeł w pamięci i dyskowy różnią się,
   zapisz i-węzeł na dysk (*write_ inode*)
6. Zwiększ liczbę wolnych i wyjdź.

2.3.3 Zamiana logicznego numeru bloku na adres fizyczny - bmap

Algorytm `bmap` przekształca logiczny numer bloku na adres fizyczny.
Parametry:

- wskaźnik na i-węzeł w pamięci
- logiczny numer bloku

Schemat działania:
Określamy poziom pośredniości. Logiczny numer bloku służy jako rodzaj indeksu w tablicy numerów bloków. Jeżeli numer jest mniejszy niż ilość bloków bezpośrednich, jesteśmy na poziomie bezpośrednim, wpp. jeżeli numer jest mniejszy niż (ilość bloków bezpośrednich + liczba_adresów_w_bloku_pośrednim), jesteśmy na poziomie pojedynczym pośrednim itd.

1. Blok znajduje się na poziomie bezpośrednim:
    - zwróć odpowiedni numer bloku fizycznego (czyli numer bloku, który znajduje się w tablicy numerów bloków na pozycji równej logicznemu numerowi bloku)
2. Blok znajduje się na poziomie pojedynczym pośrednim:
    - pobierz numer fizyczny bloku pośredniego
    - odczytaj blok o tym numerze (algorytm `bread`)
    - odczytaj odpowiedni numer bloku bezpośredniego
    - zwolnij bufor z odczytanym blokiem (algorytm `brelse`)
    - zwróć odczytany numer
3. Blok znajduje się na poziomie podwójnym pośrednim:
    - pobierz numer fizyczny bloku podwójnego pośredniego
    - odczytaj blok o tym numerze (algorytm `bread`)
    - odczytaj odpowiedni numer bloku pojedynczego pośredniego
    - zwolnij bufor z odczytanym blokiem (algorytm `brelse`)
    - odczytaj blok o tym numerze (algorytm `bread`)
    - odczytaj odpowiedni numer bloku bezpośredniego
    - zwolnij bufor z czytania (algorytm `brelse`)
    - zwróć odczytany numer
4. dla poziomu potrójnego pośredniego analogicznie o jedno zagłębienie więcej

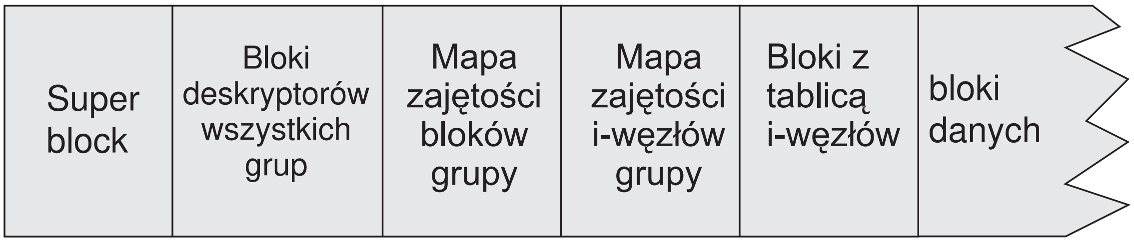
3 System plików EXT2 w Systemie Linux wersja 2.0.32

Łukasz Kowalik, l.kowalik@students.edu.pl

3.1 Struktura systemu plików EXT2 na dysku

Współczesne dyski twarde podzielone są stałej długości fragmenty : bloki dyskowe. Operacje transmisji dotyczą zawsze całych bloków a nie poszczególnych bajtów. Rozmiar bloku w EXT2 określany podczas formatowania dysku jest potęgą dwójki z przedziału od 1024 do 4096 bajtów (liczby te są określone przez stałe EXT2_MIN_SIZE i EXT2_MAX_SIZE w pliku nagłówkowym ext2_fs.h). Charakterystyczną cechą EXT2 jest fakt, iż bloki dyskowe połączone są w większe jednostki zwane grupami. Każda grupa stanowi spójny ciąg bloków, których liczba jest taka sama we wszystkich grupach (z wyjątkiem być może ostatniej). Idea ta zobrazowana jest na rysunku1:

Wszystkie grupy bloków dyskowych mają identyczną budowę pokazaną schematycznie na rysunku 2:



Na początku znajduje się blok identyfikacyjny (inaczej: super block).

Blok identyfikacyjny opisuje system plików. Zawiera on informacje ogólne o filesystemie takie jak wielkość bloku, liczba wszystkich i-węzłów, liczba wolnych i-węzłów, liczba wszystkich bloków, liczba wolnych bloków, wielkość grupy bloków, liczba grup, czasy ostatniego zapisu superbloku, montowania systemu plików i wiele innych. Podczas inicjalizowania systemu plików Linux tworzy w pamięci podobną, choć rozszerzoną o wiele nowych pól strukturę i wypełnia ją zgodnie z informacjami z dysku.

Za blokiem identyfikacyjnym znajdują się bloki deskryptorów wszystkich grup (ich liczba jest zależna od liczby grup w systemie plików). [3]

Kolejny blok zajęty jest przez mapę bitową zajętości bloków grupy (każdemu bitowi odpowiada 1 blok). W dalszej kolejności mamy blok z bitmapą zajętości i-węzłów grupy oraz pewna liczba (stała dla każdej grupy, zależna od wielkości bloku (!) - o tym zaraz) bloków zawierających tablicę i-węzłów grupy. W tym miejscu kończą się bloki używane przez system i przestrzeń do końca grupy wypełniona jest blokami danych (użytkownika). Zauważmy w tym miejscu, że ponieważ mapa bitowa zajętości bloków zajmuje zawsze dokładnie 1 blok, rozmiar bloku dyskowego implikuje zawsze konkretny rozmiar grupy (liczba grup zależy jeszcze od wielkości dysku). Dla bloku o

---

[3] Uwaga ! Informacje o tym, że pierwsze bloki każdej z grup zawierają redundantne kopie superbloku i deskryptorów zawarte są w projekcie Linux, LabLinuxie i Kernel Hacker's Guide. Uruchamiając program /sbin/dumpe2fs można się przekonać, że bloki te oznaczone są jako systemowe i nie są zajęte przez bitmapy zajętości ani tablicę i-węzłów (struktura wszystkich grup jest taka sama). Przeglądając pliki źródłowe w katalogu fs/ext2 można się jednak przekonać, że bloki te są buforowane a bufory powiązane są tylko z blokami w pierwszej grupie. W przypadku jakichkolwiek zmian informacji zawartych w SB i deskryptorach grup są one zapisywane więc tylko w grupie nr 1. Informacje te mogą jednak zostać łatwo odtworzone na podstawie innych bloków (map bitowych zajętości bloków i i-węzłów). Najprawdopodobniej więc bloki te naprawdę zawierają opisane struktury, tyle że tylko niektóre zawarte w nich wartości są wiarygodne (ale tylko one są naprawdę ważne w przypadku awarii)

rozmiarze 1024 będziemy mieli więc 1024 * 8 * 1024 = 8 MB jako rozmiar grupy, z drugiej strony dla 4096 bajtów w bloku otrzymujemy grupę wielkości 128 MB. Na mapę zajętości i-węzłów przeznaczono także 1 blok. Jest to rozsądne i zupełnie wystarczające: w sytuacji krańcowej możemy mieć tyle i-węzłów ile jest bloków danych w grupie, podobnie w całym systemie plików: i-węzłów może być tyle ile wszystkich bloków z danymi. Uniemożliwia to więc sytuację, gdy mamy miejsce na nowe pliki, ale skończyły się już i-węzły. Tak jest oczywiście tylko wtedy, gdy ustalono maksymalną liczbę dostępnych i-węzłów w grupach. W rzeczywistości najczęściej liczba ta jest mniejsza (opisywana bitmapa jest wykorzystywana tylko częściowo), choć także na poziomie praktycznie wykluczającym brak wolnych i-węzłów.

Wyjaśnimy teraz pojęcie deskryptora grupy, które pojawiło się już wcześniej. Do każdej grupy przypisany jest jej deskryptor, zawierający kilka podstawowych informacji na jej temat, tzn:

- numer bloku z bitmapą zajętości bloków
- numer bloku z bitmapą zajętości i-węzłów
- numer pierwszego bloku z tablicą i-węzłów
- liczba wolnych i-węzłów
- liczba wolnych bloków
- liczba katalogów w grupie

Opisana powyżej struktura systemu plików na dysku składająca się z wielu grup bloków dyskowych niesie ze sobą pewne korzyści. Po pierwsze, redundantne kopie bloku identyfikacyjnego i deskryptorów grup zwiększają bezpieczeństwo systemu. W przypadku awarii fragmentu dysku istnieje szansa odtworzenia niektórych zasobów. Ponadto, jak zobaczymy niebawem, system dba o to, żeby bloki z jednego pliku, a nawet bloki plików z jednego katalogu leżały w tej samej grupie, oraz o równomierne wypełnianie grup. Struktura grup bloków umożliwia więc optymalizację prędkości pracy z dyskiem i równomierne zużycie sektorów na dysku.

3.2 Niektóre struktury danych systemu EXT2 tworzone w pamięci operacyjnej

W celu przyspieszenia operacji na dysku niektóre bloki systemowe są wczytywane przez system do buforów w pamięci operacyjnej (używane są standardowe algorytmy bread i brelse). Należą do nich np. blok identyfikacyjny i bloki deskryptorów grup. Do buforów załadowana jest także pewna liczba (maksymalnie MAX_GROUP_LOADED=8) bitmap zajętości bloków i i-węzłów. Jeśli liczba grup jest większa od powyższej stałej, bloki bitmap są wczytywane i usuwane z pamięci zgodnie z algorytmem LRU.

3.3 Niektóre algorytmy specyficzne dla systemu plików EXT2

3.3.1 Przydział i-węzłów na dysku (ialloc.c).

Przyjrzyjmy się algorytmowi przydziału i-węzłów. Jest on wykonywany podczas tworzenia nowych plików i katalogów.

```
nazwa funkcji: ext2_new_inode
wejście: i-węzeł katalogu, rodzaj i-węzła (katalog lub nie katalog)
wyjście: nowy i-węzeł
algorytm:
{
pobierz nowy i-węzeł z puli wolnych i-węzłów w pamięci
zablokuj super blok
```

```
if (tworzymy i-węzeł dla katalogu)
{
    g := grupa, w której jest najwięcej wolnych i-węzłów
}
else /* tworzymy i-węzeł dla pliku */
{
    g := grupa, w której jest katalog zawierający tworzony plik
    if (w grupie g nie ma wolnych i-węzłów)
    {
        szukamy grupy z wolnym i-węzłem po kolei wśród grup g+1, g+2, g+4, g+8, ...
            itd. (algorytm quadratic-hash)
        jeśli to nie dało rezultatu szukamy sekwencyjnie po kolei we wszystkich grupach, ale zaczynając od grupy katalogu.
        if (znaleźliśmy grupę z wolnym i-węzłem)
            g := znaleziona grupa
        else
            odblokuj SB, zwolnij i-węzeł w pamięci (iput), zwróć błąd
    }
}
wczytaj bitmapę zajętości i-węzłów w grupie g i ustaw pierwszy wolny bit
uaktualnij odpowiednie liczniki (wolnych i-węzłów, katalogów) w SB i deskryptorach
inicjalizuj pola nowego i-węzła
odblokuj SB
}
```

Do wniosków płynących z analizy powyższego algorytmu przejdziemy później, po przedstawieniu algorytmu alokacji bloków na dysku. Na razie zwróćmy jedynie uwagę na fakt blokowania i odblokowywania bloku identyfikacyjnego. Realizujące te operacje funkcje lock_super i unlock_super implementują w rzeczywistości semafor binarny; procesy muszą czekać, dopóki SB zostanie zwolniony. W istocie, operacja przydziału nowego i-węzła na dysku jest sekcją krytyczną. Nie możemy dopuścić do sytuacji, gdy dwóm działającym współbieżnie procesom przydzielono dla różnych plików ten sam i-węzeł. Semafor ten używany jest we wszystkich operacjach odczytujących bądź zapisujących bloki systemowe.

3.3.2 Algorytm zwalniania i-węzłów na dysku (ialloc.c)

```
nazwa funkcji: ext2_free_inode
wejście: i-węzeł do zwolnienia
wyjście: brak
algorytm:
{
    sprawdź poprawność argumentów
    blokuj SB;
    wylicz nr grupy zawierającej i-węzeł i wczytaj jej bitmapę zajętości i-węzłów
    wyzeruj odpowiedni bit
    uaktualnij liczniki w SB i deskryptorze grupy
    zwolnij pamięć zajmowaną przez i-węzeł
    odblokuj SB
}
```

Powyższa funkcja wywoływana jest gdy plik lub katalog zostanie usunięty (liczba dowiązań spadnie do 0). Jak widać, jej działanie test trywialne i nie wymaga wyjaśnienia. Mniej oczywiste są tu jedynie niektóre warunki sprawdzane podczas kontroli poprawności argumentów. Sprawdzane jest między innymi, czy nikt oprócz bieżącego procesu nie używa i-węzła (w przeciwnym przypadku błąd) oraz czy liczba dowiązań do i-węzła jest zerowa.

3.3.3 Algorytm przydziału bloków dyskowych (balloc.c)

Podczas zapisywania danych do pliku system musi alokować nowe bloki dyskowe. Jest to realizowane przez funkcję ...

```
ext2_new_block
wejście: i-węzeł pliku, goal = numer bloku, który chcielibyśmy zaalokować
wyjście: ew. informacja o błędzie, liczba prealokowanych bloków, nr pierwszego takiego bloku
algorytm:
{
        blokujemy SB
        if (nie ma wolnych bloków)
                if (użytkownik ma uprawnienia nadzorcy systemu (superuser))
                        if (nie ma wolnych bloków wśród bloków zarezerwowanych)
                                zwróć błąd
                else zwróć błąd;
        g := grupa zawierająca blok goal
        if (goal jest zajęty)
{
                próbujemy szukać wolnych bloków BLISKO za blokiem goal;
                if (nie udało się)
                        szukamy do końca grupy spójnego ciągu 8 bloków
                        if (nie znaleźliśmy) szukamy wolnego bloku (zaczynając od goal+1)
        }
        if (w grupie g nie ma wolnych bloków)
{
                szukamy grupy z wolnym blokiem, ale po kolei, poczynając od g+1.
                szukamy w znalezionej grupie spójnego ciągu 8 bloków
                if (nie znaleźliśmy) szukamy jakiegokolwiek wolnego bloku
        }
        wywołaj funkcję sprawdzającą quotę (w razie potrzeby wychodzimy z błędem)
        ustaw odpowiedni bit w mapie bitowej.
        if (podczas kompilowania jądra zdefiniowano taką możliwość)
                próbuj prealokować kolejnych 7 bloków, za każdym razem sprawdź quotę
        uaktualnij liczniki w SB i deskryptorze grupy
        powiąż przydzielony blok dyskowy z buforem (algorytm bread)
        odblokuj SB
}
```

Jak widać funkcja jest dość skomplikowana. W tym miejscu wyjaśnijmy jedynie kilka niezrozumiałych być może haseł z tekstu algorytmu.

W systemie plików EXT2 istnieje pewna stała (określona w bloku identyfikacyjnym i ustalana przy formatowaniu dysku) wyznaczająca liczbę bloków zarezerwowanych dla systemu. Jeżeli liczba wolnych bloków na dysku spadnie poniżej tej stałej, nowe bloki mogą być przedzielane jedynie procesom użytkowników z uprawnieniami root-a.

Prealokacja bloków dyskowych jest cechą systemu którą można włączyć lub wyłączyć przed kompilacją jądra. Prealokacja ma na celu zwiększenie prędkości zapisu do większych plików na dysku. Wypełniając żądanie przydziału nowego bloku, system, niejako przewidując przyszłość próbuje przydzielić kilka kolejnych. Rozwiązanie to ma również inną zaletę, w przypadku działania wielu współbieżnych procesów piszących na dysk ("odbierając" sobie nawzajem bloki na dysku powodują większe *poszatkowanie* plików).

3.3.4 Algorytm zwalniania bloków dyskowych (w balloc.c)

```
funkcja: ext2_free_blocks
wejście: i-węzeł pliku zawierającego zwalniane bloki, nr pierwszego bloku, liczba bloków
wyjście: brak
algorytm:
{
    if (nie wszystkie zwalniane bloki leżą w 1 grupie) zwróć błąd
    blokuj SB
    dla każdego kolejnego bloku:
    {
        zerujemy odpowiedni bit
        uaktualniamy dane w i-węźle
        uaktualniamy liczniki w SB i deskryptorze grupy
    }
    odblokuj SB
}
```

Ta funkcja nie wymaga chyba zbyt wiele wyjaśnień. Jak widać służy do zwalniania spójnego ciągu bloków w obrębie jednej grupy.

3.3.5 Wnioski

Z analizy przedstawionych w punktach 3.1 - 3.4 algorytmów płyną ważne wnioski na temat strategii działania systemu plików EXT2. Widać wyraźnie, że system plików dba o równomierne rozmieszczenie i-węzłów w grupach. Cel ten osiągany jest dzięki odpowiedniemu wyborowi grupy dla i-węzła katalogu. Oglądając funkcje ext2_file_write i ext2_getblk można się przekonać jak ustawiany jest parametr goal żądanego bloku dyskowego przy alokacji nowego bloku. Otóż w przypadku dopisywania danych na końcu pliku przekazywany jest następny blok za końcem pliku, natomiast w przypadku tworzenia nowego pliku przekazuje się pierwszy blok danych grupy zawierającej i-węzeł pliku. Algorytmy new_inode i new_block, wspólnymi siłami dbają więc o równomierne rozmieszczenie danych na dysku. Widzimy również, iż EXT2 stara się przechowywać całe katalogi (i-węzły i dane) w możliwie małej liczbie grup. Jak to było widoczne w obu algorytmach, w przypadku braku wolnych i-węzłów lub bloków w grupie odpowiednio katalogu lub pliku numery kolejnych grup do sprawdzenia zależą od numeru tej "pierwszej". Dzięki temu właśnie pliki należące do jednego katalogu jak również bloki jednego pliku nie są rozrzucone po wszystkich grupach.

Widzimy teraz dokładnie, że podział dysku na grupy bloków nie jest jedynie zbiorem iluzorycznych granic w przestrzeni dyskowej. Użyte algorytmy powodują dużą niezależność danych zapisanych w poszczególnych grupach. Równocześnie są jednak elastyczne nie dopuszczając do jakiegoś nowego rodzaju fragmentacji. Takowy mógłby wystąpić gdyby ustalono np. iż bloki jednego pliku MUSZĄ znaleźć się w jednej grupie. Mogłyby więc wystąpić sytuacje, gdzie jest miejsce na zapisanie pliku na dysku, ale system zgłasza błąd. Taka wersja systemu plików opisana jest w książce Bacha (czyli pewnie była używana w starszych UNIXach)

3.4 Katalogi w Linuxie

3.4.1 Wstęp

Katalogi nakładają na system plików strukturę hierarchiczną. Na system plików można patrzeć jak na drzewo poetykietowane nazwami plików i katalogów, z tym że węzłami wewnętrznymi mogą być tylko katalogi. Taki sposób organizacji systemu plików jest bardzo wygodny i czytelny dla użytkownika i dlatego używany jest w większości współczesnych systemów operacyjnych, także w Linuxie.

Dla ścisłości powiedzmy sobie jeszcze, że wspomniane "drzewo" w linuxie, z powodu istnienia dowiązań symbolicznych może zawierać cykle (przestaje wtedy tak naprawdę być drzewem).

3.4.2 Struktura katalogów na dysku w systemie plików EXT2

W Linuxie katalogi są plikami o specjalnej strukturze. Każdy plik katalogu zawiera ciąg tzw. pozycji. Każda pozycja opisuje dowiązanie do pewnego pliku znajdującego się w katalogu. Pozycja w systemie plików EXT2 zawiera takie informacje jak:

- jednoznaczny w danym systemie plików numer i-węzła
- długość pozycji w bajtach
- długość nazwy
- nazwa (długość do 255 znaków)

Pozycje mogą więc mieć różną długość zależną od nazwy dowiązania. W pliku tym mogą istnieć pozycje z numerem i-węzła równym 0. Powstają one po usunięciu pozycji z katalogu. W takie miejsca mogą być wstawiane nowo tworzone pozycje.

3.4.3 Algorytmy i struktury danych dotyczące obsługi katalogów tworzone przez system Linux w pamięci (plik fs/dcache.c)

Dla zwiększenia sprawności operacji dyskowych system Linux przechowuje część pozycji katalogowych w pamięci. Zaszczytu tego mogą dostąpić jedynie pozycje o nazwach nie dłuższych niż 15 znaków (stała DCACHE_NAME_LEN). To dziwne założenie zostało wprowadzone podobnież w celu oszczędności pamięci.

Pozycja w pamięci różni się znacznie od tej na dysku zawiera bowiem dodatkowe informacje. Pierwszą z nich jest numer urządzenia (dysku, partycji) na którym znajduje się system plików zawierający daną pozycję (dopiero mając numer i-węzła i urządzenia zawierającego plik możemy go odnaleźć na dysku). Znajduje się tam także numer i-węzła katalogu zawierającego daną pozycję i jego wersja. Podczas wyszukiwania pozycji w pamięci sprawdzane są następujące pola jednoznacznie identyfikujące pozycję: nazwa, numer urządzenia, numer i-węzła katalogu i jego wersja. Wersja sprawdzana jest w celu uniknięcia odczytania nieaktualnych danych.

W danej chwili w pamięci może być przechowywana tylko pewna niewielka liczba pozycji (2*DCACHE_SIZE=2*128=256). Aby przyspieszyć ich wyszukiwanie w pamięci pozycje zorganizowane są w listy, do których dostęp umożliwia tablica haszująca. Liczba tych list (rozmiar tablicy haszującej) wynosi 32 (D_CACHE_HASH_QUEUES).

Wyszukiwanie w pamięci pozycji realizowane jest więc według następującego schematu:

- za pomocą funkcji haszującej na podstawie numeru urządznia, i-węzła katalogu i nazwy określ pozycję w tablicy list;
- przeszukaj wybraną listę porównując 4 wymienione nieco wyżej pola.

Niezależnie od struktury haszującej, pozycje katalogowe zorganizowane są w kolejki LRU. Tak, właśnie tak: kolejki, a nie kolejkę. Użyto bowiem dwóch poziomów buforowania (dwóch kolejek LRU). W kolejce poziomu I lądują nowo dodawane pozycje (których wcześniej nie było w pamięci). W kolejce poziomu II mają natomiast znajdować się pozycje używane co najmniej dwukrotnie.

Jeśli więc znajdziemy pozycję w pamięci i znajduje się ona w kolejce poziomu I przenosimy ją na poziom II (usuwając w razie potrzeby najdawniej używaną pozycję z poziomu II - jak to w kolejce LRU bywa). Intencja zawarta w tym mechanizmie polega na tym, że na poziomie pierwszym buforowania pojawiać mogą się pozycje "przypadkowe", natomiast na poziomie II znajdują się pozycje, których naprawdę (często) używamy. Linus tłumaczy wprowadzenie tego algorytmu chęcią wyeliminowania sytuacji, gdy np. wywołanie funkcji readdir() (czytającej być może ogromną liczbę pozycji, z których wykorzystamy tylko kilka) powoduje wyrzucenie z pamięci pozycji potrzebnych często.

3.4.4 Zarys algorytmu namei (w: namei.c)

Użytkownik systemu wykonując operacje na plikach posługuje się nazwami ścieżkowymi, natomiast system plików i-węzłami. Musi istnieć więc jakiś mechanizm tłumaczenia jednych na drugie. Jest on realizowany przez funkcję namei.

Schemat działania namei jest oczywisty:

```
algorytm namei:
{
        if (nazwa zaczyna się od '/')
 ustaw bieżący i-węzeł na korzeń systemu plików
        else
ustaw bieżący i-węzeł na i-węzeł bieżącego katalogu.
        do {
                odetnij kolejny fragment nazwy do nastepnego znaku '/'
                znajdź odpowiedni i-węzeł w katalogu (którego i-węzeł jest bieżący)
                ustaw bieżący i-węzeł na ten znaleziony
                sprawdź czy bieżący i-węzeł oznacza dowiązanie symboliczne. Jeśli tak jest
                        wykonaj funkcję follow_link;
                sprawdź, czy nie przechodzimy przez punkt zamontowania systemu plików...
        } until (odcięliśmy ostatni fragment, nie było już znaku '/')
        zwróć bieżący i-węzeł
}
```

Warto wiedzieć, iż w pliku namei.c znajduje się kilka odmian algorytmu namei przydatnych w odmiennych sytuacjach, np. dir_namei zwraca i-węzeł pliku i jego katalogu (przydatny w funkcjach systemowych modyfikujących plik albo jego i-węzeł. Jest też open_namei wywoływana przez funkcję open (sprawdzane są prawa dostępu).

4 Funkcje systemowe systemu plików (część 1).

autor: Tomasz Osman

4.1 Wstęp .

Funkcje z ,których korzysta użytkownik są elementami bibliotek. Odwołują się one do funkcji udostępnianych przez system poprzez wywołanie odpowiedniego przerwania z numerem funkcji .Tak też funkcjom fopen , open z bibliotek odpowiada funkcja sys_open , procedurom fprintf , write odpowiada funkcja sys_write , fscanf odwołuje się do sys_read , fclose do sys_close itd. Funkcje systemowe działają na zasadzie zleceń ,które jądro udostępnia użytkownikom gwarantując za pomocą tego wąskiego interfejsu bezpieczeństwo systemu i jednolity sposób odwoływania się do plików niezależnie od systemu plików.

4.2 Ogólny schemat działania funkcji systemowych systemu plików.

Realizacja ogólnego interfejsu procedur dla różnych systemów (a także typów plików) polega na podzieleniu naszego zadania na część ogólną wykonywaną dla wszystkich typów plików (np. sprawdzenie praw dostępu , zaznaczenie modyfikacji pliku itp. - w tej części korzystamy z głównych struktur opisujących plik jakimi są i-węzły i tablica plików -struktur "file") oraz część specjalną wykonywaną dla danego rodzaju pliku , odpowiednia funkcja którą wykonujemy znajduje się w tablicy funkcji (charakterystycznej dla danego rodzaju plików) i prowadzi do niej wskaźnik z i-węzła pliku (póżniej wskaźnik do tablicy kopiowany jest do pozycji f_op struktury "file" - w trakcie wykonywania open). Wskaźnik do tablicy funkcji w i-węźle inicjowany jest w momencie tworzenia danego i-węzła (dokładniej w trakcie tworzenia kopii w pamięci operacyjnej) Wtedy to zależnie od typu pliku i systemu plików do którego należy plik podstawiany jest odpowiedni wskaźnik (np. w przypadku gdy jest to urządzenie znajdujemy odpowiednią tablicę funkcji w tablicy rozdzielczej - patrz opis podsystemu wejście - wyjście).

Przykładowa tablica funkcji (dokładniej struktura file_operations) dla plików regularnych z systemu ext2 wygląda następująco:

```
NULL   // standardowa funkcjia lseek pasuje dla systemu ext2
generic_file_read  //  czytanie
ext2_file_write // pisanie
NULL
NULL
ext2_ioctl
generic_file_mmap
NULL  // nie wymagamy specjalnych procedur otwarcia
ext2_release_file // wywoływana w przypadku zwolnienia pozycji
       // z tablicy plików
ext2_sync_file
NULL,
NULL,
NULL
```

Jeśli wskaźnik do funkcji jest NULL to znaczy ,że albo system ext2 akceptuje standardową obsługę tego zadania albo w ogóle nie podtrzymuje realizacji danego zadania .

4.3 Główne struktury danych systemu plików .

Główne trzy elementy opisujące plik to tablica deskryptorów plików - charakterystyczna (osobna) dla każdego procesu . Pozycje z tablicy deskryptorów wskazują na pozycje tablicy plików (wspólne dla wszystkich procesów) , a te wskazują na i-węzły . Każda pozycja z tablicy plików (struktura "file") posiada informację o ilości dowiązań prowadzących z tablic deskryptorów .Licznik ten jest zwiększany gdy wywołujemy funkcje dup (kopiowanie deskryptorów) i w przypadku tworzenia nowego procesu . Za pomocą tego mechanizmu możemy realizować komunikacje między procesami . Podobny licznik zawiera każdy i-węzeł przyczym zlicza on wszystkie odwołania do danego i-węzła (nie koniecznie pochodzących z tablicy plików) . W strukturze "file" zapisane są wszystkie dane dotyczące kontekstu otwarcia pliku (czyli tryb otwarcia , pozycja pliku , dostępne operacje itp.) , natomiast w i-węźle trzymamy globalną informację o danym pliku (więcej informacji na ten temat znajdziesz szanowny czytelniku w rozdziale dotyczącym i-węzłów).

4.4 Otwieranie pliku

`open(const char * pathname, int flags, int mode)`

4.4.1 Wstęp.

Funkcja ta jest pierwszą czynnością jaką musi wykonać proces by uzyskać dostęp do pliku . Zmieniając struktury danych zapisywana jest informacja ,że ktoś bedzie korzystał z pliku , przygotowujemy mechanizmy transmisji . Wynikiem funkcji jest deskryptor , który identyfikuje plik (w jaki sposób - patrz glówne struktury danych).Ilość otwartych plików przez jeden proces jest ograniczona: rozmiarem tablicy (listy) plików (NR_FILE)- standardowo 1024 , rozmiarem tablicy deskryptorów NR_OPEN -standardowo 256 . Można zmniejszyć ten limit - ustawiając ograniczenie na prywatnej zmiennej procesu. Ze względu na to ,że występują rożnorodne typy plików (pliki specjalne - urządzenia , pliki katalogi ,...) procedura otwarcia zależy od tego co otwieramy (jak to jest realizowane - patrz ogólny schemat działania funkcji systemowych) i jak otwieramy (co wymagamy od pliku).

4.4.2 Sposoby otwarcia pliku

Niektóre flagi są zdefiniowane tylko dla pewnych rodzajów plików. W niektórych przypadkach flagi są ignorowane (np. obcięcie łacza) lub zgłaszany jest bład (np. brak odpowiednich uprawnień).

- O_RDONLY - otwarcie tylko do czytania
- O_WRONLY - otwarcie tylko do pisania
- O_RDWR - czytanie i pisanie
- O_CREATE -jeśli plik nie istnieje będzie stworzony , jeśli istnieje i występuje flaga O_EXCL to błąd
- O_TRUNC - obcięcie pliku ,O_APPEND - przed każdym pisaniem wskaźnik pozycji pliku ustawiany jest na koniec
- O_NONBLOCK ,O_NONDELAY - wykonywane operacje będą odbywały się bez blokowania (zawieszania) procesu
- O_SYNC - synchroniczny zapis oznaczający ,że po każdym zapisie dane fizycznie znajduja się na dysku

4.4.3 Algorytm

Pobranie nieużywanego deskryptora (o najmniejszym numerze). Pobranie wolnego miejsca w tablicy (liście) plików , jeśli wszystkie są zajęte i jest ich mniej niż NR_FILE to alokujemy miejsce na nowe pozycje "file" (pobieramy stronę pamięci - którą całą przeznaczamy na struktury "file") w przeciwnym wypadku błąd .

Następnie kopjujemy flagi otwarcia pliku do struktury "file" (tu jest trzymana informacja o sposobie otwarcia pliku dla pózniejszych odwołań do pliku). Następnie szukamy i-węzła katalogu dla naszego pliku i nazwy pliku w tym katalogu (ostatnia składowa ścieżki */nazwa).Procedure tą realizuje jedna z funkcji typu *_namei (zamieniającą scieżke na i-węzeł) algorytm zajmuje dany i-węzeł (funkcja i_get) zwiekszając odpowiedni licznik odwołań do i-węzła (patrz wewnętrzna reprezentacja plików i-węzły).

Dalsze postępowanie zależne jest od tego czy występuje flaga O_CREATE. Jeżeli tak to opuszczamy semafor na i-węźle katalogu by inne procesy nie ingerowały w strukture i-węzła w trakcie naszych zmian . Sprawdzamy czy istnieje już szukany plik (algorytm lookup) -jak to się dzieje ,że algorytm lookup potrafi odpowiedzieć na to pytanie niezależnie od rodzaju systemu plików .Oczywiście (patrz ogólny schemat działania funkcji systemowych) wygląda to tak ,że algorytm najpierw sprawdza czy i-węzeł jest i-węzłem katalogu czy mamy odpowiednie prawa do katalogu (prawa wykonywania) ,po wykonaniu tej części ogólnej wykonywana jest część charakterystyczna dla danego systemu plików .

Tak więc jeśli dany plik już istnieje i ustawiona jest flaga O_EXCL to zwracany jest błąd w przeciwnym razie pobierany jest i-węzeł naszego pliku . Jeśli natomiast plik nie istnieje to sprawdzamy czy możemy utworzyć plik w katalogu : sprawdzenie praw zapisu i wykonywania (patrz sprawdzanie praw dostępu), sprawdzenie czy istnieje dowiązanie do funkcji tworzącej plik w katalogu (inna dla różnych systemów).Wywołujemy tą funkcję ,w tym momencie posiadamy i-węzeł interesującego nas pliku. Przy tworzeniu nowego pliku nadajemy uprawnienia dla wszystkich użytkowników wykorzystując zmienną mode zmodyfikowaną o maskę (nadawanie pewnych praw może być zabronione dla pewnych procesów - maska ta znajduje się na prywatnej zmiennej procesu current->...).Stworzenie nowego pliku wymaga pobrania wolnego i-węzła (algorytm i-alloc) zapisanie go i zlecenie zapisu tego i-węzła na dysk (wszystko to odbywa sie poprzez podręczną pamięć buforową zlecamy zapis bloku - odpowiadającego i-węzłowi),zapis ten musi być synchroniczny w przypadku zapisu niesynchronicznego dane z i-węzła katalogu mogłyby się znaleźć wcześniej na dysku i w przypadku awarii katalog mógłby zawierać dowiązanie do losowego i-węzła . W naszym przypadku gdy awaria wystąpi po zapisie i-węzła nowego pliku i przed zapisem i-węzła katalogu system wprawdzie jest rozspujniony ale łatwo to naprawić przy sprawdzaniu systemu plików (algorytm fsck). Jeśli flaga O_CREATE nie była ustawiona to pobierany jest i-węzeł naszego pliku (algorytm lookup).

Następnie wykonujemy szereg czynności mających na celu ustalenie zgodności żądań (flag) do możliwości (prawa procesu) i rodzaju pliku. Tak więc nie można otworzyć katalogu do zapisu (zapisywanie zawartości w sposób niekontrolowany mógłby wyrządzić szkody w systemie plików), sprawdzane są prawa (patrz sprawdzanie praw) , jeśli plik jest gniazdem , łączem , urządzenie kasowana jest flaga obcięcia pliku .Jeżeli plik jest przeznaczony do dopisywania (informacja o tym znajduje się w i-węźle w polu i_mode) to można go otworzyć w trybie pisania jedynie z flagą O_APPEND (do dopisywania) ,flaga obcięcia pliku jest niedozwolona .Obcięcie pliku jest także zabronione jeśli są założone pewne blokady na plik .

Jeśli powyższe czynności zakończyły się pomyślnym pobraniem i-węzła to następna część odpowiada poprawianiu struktur opisujących dostęp do otwartego pliku . Tak też modyfikowane są pola pozycji z tablicy plików (struktura "file"). Czyli: ustawienie dowiązanie do i-węzła , zainicjowanie pozycji w pliku na 0 , zainicjowanie znacznika czytania z wyprzedzeniem na 0 ,ustalenie skopiowanie z i-węzła zestaw możliwych operacji na i-węźle

.Wreszcie następuje wywołanie charakterystycznej funkcji otwierającej plik dla danego typu pliku (jeśli istnieje) .Procedura ta może inicjalizować pewne struktury odpowiadające za transmisje danych (w przypadku łączy) lub podmieniać tablice możliwych operacji jak to się dzieje w przypadku urządzeń.

Końcowym etapem funkcji jest wpisanie dowiązania do pozycji z tablicy plików w pozycji z tablicy deskryptorów (deskryptor wskazuje na strtukture "file").

4.5 Pisanie do pliku

```
write(int fd , char * buf , int count);
```

4.5.1 Wstęp

Funkcja zapisuje pewną ilość (prubuje zapisać ilość równą count) bajtów do pliku z bufora podanej na zmiennej buf . Plik identyfikowany jest przez deskryptor pliku . Funkcja zwraca ilość zapisanych bajtów.

4.5.2 Algorytm

Pobranie pozycji w tablicy plików (deskryptor wskazuje na nią) i pobranie odpowiedniego i-węzła (jedno z pól struktury "file" wskazuje na niego). Sprawdzenie czy na część pliku na którą chcemy pisać są założone blokady jeśli tak to błąd .Sprawdzenie prawa do czytania z podanego obszaru pamięci użytkownika - jeśli nie sprawdzilibyśmy tego teraz to przy próbie czytania z odpowiedniego miejsca w pamięci wygenerowany zostałby błąd i standardowa obsługa błędu zabiłaby proces , tak więc nasza funkcja gwarantuje zakończenie pomyślne (w tym sensie) zapisu .Sprawdzamy czy plik otwarty jest do zapisu - jeśli nie błąd.

Następnie sprawdzamy czy jest zdefiniowana specjalna procedura zapisu dla tego typu pliku i wywołujemy tą procedurę , przedtem jednak opuszczamy semafor gwarantujac ,że nikt nie zmodyfikuje zawartości pliku (jak i zawartości i-węzła) w trakcie pisania do pliku . Następnie kasujemy bit setuid i setgid z pola i_mode i-węzła pliku. Bity te oznaczają ,że efektywnym właścicielem procesu (efektywny identyfikator grupy) w przypadku gdy chcemy wykonać ten plik staje się właściciel pliku.Skasowanie tych bitów ma na celu zapobiegnięcie sytuacji , że urzyczamy swoich praw innym procesom (w ramach wykonywania naszego pliku) ,przyczym zawartość pliku uległa zmianie i bity były ustawione w starym kontekście zawartości pliku .Jeśli chcemy by bity były na owo ustawione musi to zrobić właściciel pliku.

Dalsza część funkcji zależy od tego do jakiego typu pliku zapisujemy dane. Dalej będziemy opisywać funkcje ext2_file_write (funkcje zapisu dla systemu ext2). Upewniamy się ,że system pliku nie jest przeznaczony tylko do odczytu , informacja o tym znajduje się w strukturze super_block do ,której prowadzi dowiązanie z i-węzła (super blok znajduje się w odpowiednim miejscu na dysku i gdy podmontowujemy system plików pobierany jest on do pamięci patrz system plików - ext2).

Następnie sprawdzamy czy plik jest regularny. Ustawiamy bieżącą pozycję pliku w zależności od flagi O_APPEND . Obliczamy logiczny numer bloku w tym pliku (i przesunięcie w pliku) - korzystamy tu z informacji o rozmiarze bloku zawartym w strukturze super_block. Dalej w pętli dopóki mamy coś do zapisu i rozmiar pliku nie przekroczył 2 GB zapisujemy kolejne bloki . W tym celu pobieramy kolejny bufor odpowiadający naszemu blokowi , specjalne pole w i-węźle mówi nam o urządzeniu odpwiadającym dyskowi gdzie znajduje się nasz plik. Wykorzystywana jest tu także funkcja bmap , która zamienia (mapuje) logiczny numer bloku na fizyczny (patrz opis systemu ext2).Jeśli danemu blokowi logicznemu nie odpowiada blok fizyczny pobieramy wolny blok (lub kilka bloków jeśli musimy je zainicjować przy adresowaniu pośrednim) i przypisujemy blok do naszego i-węzła (modyfikacja i-węzła -zaznaczamy i-węzeł jako brudny)) Pobranie wolnego bloku wymaga zmiany struktury "listy" wolnych bloków (patrz opis systemu plików ext2) dbamy tutaj by zmiany były synchronicznie zapisywane na dysk , by nie dopuścić do sytuacji ,że blok jest używany przez plik i znajduje się na liście wolnych bloków .

Jeśli nie będziemy zapisywać całego bufora (może sie to zdarzyć przy zapisie pierwszego lub ostatniego bloku) to najpierw musimy pobrać zawartość bufora (oczywiście jeżeli bufor nie jest aktualny).Tak więc zlecamy wczytanie bloku i czekamy na koniec transmisji na buforze , zostaniemy obudzeni przez procedure obsługi przerwania końca zapisu bloku (patrz podsystem wejścia - wyjścia).Następnie kopjujemy dane z bufora użytkownika do bufora pliku .W tym momencie uaktualniamy kopie strony .Otóż należy sobie zdać sprawę z tego ,że oprócz buforowania danych z pliku w buforach systemowych dane te są także przetrzymywane w ”cache’u” stron (strony odpowiadają pewnym częściom pliku) tak też w przypadku zmiany zawartości pliku odpowiednie strony muszą zostać uaktualnione (by uzyskać więcej informacji - patrz opis funkcji read). Jeśli zapis nie jest synchroniczny to porzucamy bufory (mechanizm buforowania w pewnym momencie zadba o wysłanie zawartości na dysk ,być może uczyni to z pewnym opóźnieniem .Należy mieć więc w świadomości ,że w przypadku awarii nie wszystkie dane znajdą się na dysku możemy temu zaradzić często opróżniając bufory systemowe za pomocą funkcji sync()). Jeśli zapis jest synchroniczny wpisujemy bufor do lokalnej tablicy buforów . W momencie gdy tablica się zapełni zapisujemy dane na dysk . Tablica standardowo zawiera 16 pozycji .

Na końcu pętli uaktualniamy zmienne: pozycje w buforze użytkownika , ilość zapisanych danych . Po wyjściu z pętli uaktualniamy dane dotyczące pliku: rozmiar pliku jeśli uległ zwiększeniu , ostatni czas modyfikacji , nowa pozycja w pliku ,zaznaczenie ,że i-węzeł jest brudny (różni się od wersji dyskowej). Po zakończeniu transmisji podnosimy semafor - udostępniając plik innym.

4.6 Zmiana bieżącego katalogu

`chdir(const char * pathname );`

4.6.1 Wstęp

Każdy proces posiada w swoim kontekście wykonywania przydzielony plik zwany dalej bieżącym plikiem . Proces w momencie tworzenia (algorytm fork) dziedziczy taki plik po ojcu.Względem tego pliku analizowana jest nazwa plików ,chyba ,że proces zażyczy sobie inaczej (np . piszac / na początku ścieżki).Proces trzyma dowiązanie do i-węzła na zmiennej prywatnej (current->...). Mechanizm "listowy" przetrzymywania i-węzłów gwarantuje ,że nasz i-węzeł nie zostanie zwolniony dopóki jest on bieżącym plikiem naszego procesu (gwarantuje to licznik dowiązań w i-węźle).

4.6.2 Algorytm

Najpierw za pomocą procedury namei (patrz system plików ext2) analizowana jest nazwa ścieżkowa i pobierany jest interesujący nas i-węzeł. Jeśli i-węzeł nie jest katalogiem (informację o tym pobieramy z pola i_mode pobranego i-węzła) to błąd . Następnie sprawdzamy prawa wykonywania dla pliku (patrz sprawdzanie praw). Jeśli wszystko zakończyło się pomyślnie zwalniany jest i-węzeł bieżącego pliku (algorytm i_put : zmniejsza licznik odwołań i jeśli spada on do zera i-węzeł oznaczany jest jako wolny (można go ponownie użyć), jeśli to konieczne zapisywany jest na dysk . Sprawdzana jest także liczba dowiązań do pliku i jeśli spadła do zera (żaden katalog go nie zawiera) i-węzeł jest fizycznie zwalniany (algorytm ifree) ,i zwalniane są wszystkie bloki dyskowe naszego pliku (algorytm free) - dokładniejszy opis patrz wewnętrzna reprezentacja plików - i-węzły). Po zwolnieniu i-węzła bieżącego pliku ustawiamy dowiązanie do nowego i-węzła na zmiennej prywtnej procesu (current->fs->pwd).

4.7 Zmiana aktualnego korzenia plików

`chroot(const char * pathname);`

4.7.1 Wstęp

Każdemu procesowi przydzielony jest plik - główny korzeń systemu plików. Podobnie jak bieżący plik proces dziedziczy korzeń od przodka .Względem korzenia rozpatrywane są nazwy ścieżkowe jeśli nazwa zaczyna się od /... Wskaźnik do odpowiedniego i-węzła znajduje się na zmiennej prywatnej procesu (current->fs->root).

4.7.2 Algorytm

Analizując nazwę ścieżkową pobieramy interesujący nas i-węzeł (namei). Następnie sprawdzamy czy plik jest katalogiem w przeciwnym wypadku zwalniamy pobrany i-węzeł a funkcja kończy się błędem .Jeśli wszystko jest w porządku sprawdzamy czy proces ma status "superusera" - co oznacza ,że identyfikator efektywnego właściciela procesu musi być równy 0 czyli taki sam jak identyfikator nadzorcy systemu.Otóż oprócz właściciela procesu , proces ma efektywnego właściciela procesu , który w momencie wykonywania pliku może być zmieniony na właściciela pliku w zależności od tego czy jest ustawiony bit setuid w polu i_mode i-węzła pliku . Czyli status "superusera" oznacza ,że właścicielem pliku był nadzorca lub proces wykonuje specyficzny plik nadzorcy . Jeśli wszystko w porządku zwalniamy stary i-węzeł korzenia plików i zapisujemy dowiązanie do nowego i-węzła korzenia na zmiennej current->fs->root .

4.8 Sprawdzanie praw dostępu do pliku

`permission(struct inode *, int mask);`

4.8.1 Wstęp

Funkcja ta istnieje by zapewnić bezpieczeństwo systemu plików. Sprawdza prawa dostępu wykorzystując dane zawarte w i-węźle i dane opisujce status procesu.Nie jest to funkcja systemowa udostępniana dla użytkownika ,lecz używana jest bardzo często przez inne funkcje.

4.8.2 Opis działania

Jeżeli jest zdefiniowana specjalna procedura postępowania dla danego typu pliku to jest wywoływana - tutaj omówię funkcję charakterystyczną dla systemu plików ext2 . Niemożliwe jest uzyskanie dostępu w trybie zapisu do pliku z systemu plików tylko do odczytu . Wszystkie prawa posiada proces o statusie "superusera" (porównaj chroot) .Przy sprawdzaniu efektywnego właściciela procesu używana jest zmienna current->fsuid , która oznacza efektywnego właściciela procesu w kontekście systemu plików - oznacza to w zasadzie to samo co current->euid (efektywny właściciel procesu) przyczym może być dodatkowo zmieniona na życzenie procesu (stąd jej wyodrębnienie).

Następnie jeśli (current->fsuid) jest rózne od identyfikatora nadzorcy (0) kwalifikujemy proces do odpowiedniej grupy : właściciel pliku , należy do tej samej grupy co właściciel pliku , inni . Po zakwalifikowaniu do odpowiedniej grupy pobieramy prawa dla tej grupy z pola i_mode i-węzła pliku (ostatnie trzy trójki bitów to prawa odczytu ,zapisu , wykonywania dla właściciela , grupy i innych) i porównujemy te prawa z żądaniami (zmienna mask) i na tej podstawie decydujemy czy proces uzyskuje dostęp do pliku.

4.9 Zmiana bieżącej pozycji pliku

`lseek(int fd ,int offset , int origin);`

4.9.1 Opis działania

Pierwszą rzeczą jest sprawdzenie poprawności parametrów gdyż pobranie ich od użytkownika nie gwarantuje nam tego - tak też numer deskryptora nie może być zbyt dyży (ograniczony stałą NR_OPEN) , deskryptor musi wskazywać na pozycje z tablicy plików ,a pozycja z tablicy plików musi wskazywać na i-węzeł pliku .Argument origin musi posiadać jedną z trzech wartości .

Następnie sprawdzamy czy jest zdefiniowana specjalna funkcja dla danego typu pliku .Jest to spowodowane tym ,że rózne typy plików mają różną semantykę działania tej funkcji (np .łacze akceptuje tylko przesunięcie wprzód i nie można przesunąć pozycji poza rozmiar pliku) ,pozatym nawet dla plików tego samego typu w zależności od rodzaju systemu plików mamy różne funkcje .Jeśli nie ma zdefiniowanej charakterystycznej funkcji to wykonywana jest standardowa procedura obsługi (korzysta z niej także system ext2). W zależności od zmiennej origin przesuwamy pozycję względem początku pliku , względem bieżącej pozycji lub końca pliku .O wielkości przesunięcia decyduje zmienna offset (może być ujemna) Jeśli nowa pozycja mialaby mieć wartość ujemną zwracany jest błąd .Jeśli nowa pozycja jest rożna od starej aktualizujemy pozycje (zmienna f_pos w strukturze "file"). Zerujemy także wskaźnik czytania z wyprzedzeniem z której korzysta funkcja read (patrz opis funkcji read).

4.10 Zamknięcie pliku

`close(int fd);`

4.10.1 Wstęp

Funkcja close wywoływana jest gdy kończymy korzystać z pliku . Jeśli proces nie zadbał o zamknięcie wszystkich deskryptorów przed końcem swego działania to system czyni to za niego.

4.10.2 Algorytm

Oczywiście najpierw musimy sprawdzić poprawność użytego numeru deskryptora (mógł on zostać przekazany bezpośrednio od użytkownika). Jeśli wszystko w porządku (numer wskazuje na pozycje w tablicy deskryptorów , a deskryptor wskazuje na pozycje w tablicy plików) deskryptor oznaczany jest jako wolny (wskazuje na NULL).

Następnie usuwane są wszystkie blokady założone przez proces użytkownika na plik.Poczym zmniejszany jest licznik odwołań w strukturze "file" jeśli licznik spada do zera struktura "file" jest zwalniana (oznaczana jako gotowa do ponownego użycia bez fizycznegozwalniania strony pamięci , którą zajmuje) , zwalniamy także odpowiadający i-węzeł (algorytm i_put patrz wewnętrzna reprezentacja plików - i-węzły).Wywoływana jest także funkcja informująca o zwolnieniu struktury "file" (jeśli jest zdefiniowana dla danego typu pliku) . Ma to znaczenie na przykład w przypadku urządzeń ,albowiem do odmontowania modułu musimy mieć pewność ,że nikt nie korzysta z urządzenia , sam licznik odwołań w i-węźle nie wystarcza gdyż zlicza on wszystkie odwołania do pliku ,a nas interesują tylko te pochodzące z otwarcia pliku . Pozatym może być dużo urządzeń tego samego typu (drugorzędny numer urządzenia - patrz podsystem wejścia- wyjścia) i odmontowanie modułu wymaga informacji ,że żadne z nich nie jest używane.

4.11 Czytanie z pliku

`read(int fd ,char * buf , int count);`

4.11.1 Wstęp

Funkcja wczytuje daną ilość bajtów z pliku identyfikowanego przez deskryptor.

4.11.2 Algorytm

Po wykonaniu części ogólnej ,która polega na pobraniu wskaźników do struktur opisujących plik (patrz opis głównych struktur danych), sprawdzeniu czy plik jest otwarty do czytania (pole f_mode w strukturz "file") , sprawdzeniu czy na plik założone są blokady i zweryfikowaniu czy obszar pamięci podany przez użytkownika należy do procesu przechodzimy do wykonania funkcji typowej dla danego rodzaju pliku . Dalej omówimy funkcje read dla plików regularnych systemu ext2.

Procedura realizuje czytanie z wyprzedzeniem zlecając wczytanie kolejnych części pliku jeszcze przy wczytywaniu wcześniejszych . Jeśli kolejno chcemy odwołać się do następnego obszaru mamy już go pod ręką . Oprócz systemowego mechanizmu buforowaniu przechowujemy w pamięci strony zawierające dane z wczytywanego pliku (strony pełnią funkcję "cache'u" tak samo jak bufory - przy czym są znacznie większe). W strukturz "file" trzymane są dane informujące o kontekście ostatniego czytania z wyprzedzeniem , mianowicie chodzi o zapamiętanie informacji czy ostatnio realizowano czytanie z wyprzedzeniem , ilość wczytanych danych i maxymalny rozmiar danych ,który można wczytać z wyprzedzeniem .

Procedura w pętli wczytuje kolejne strony pamięci kopiując ją (być może tylko część) do przestrzeni użytkownika.Opiszemy nasz algorytm na zasadzie przejść między stanami .

Sprawdzamy czy posiadamy w pamięci interesującą nas ramkę pamięci (być może wczytaną już wcześniej) w zależności od tego idziemy do stanu znaleziono ramkę lub nie znaleziono ramki .

- Stan znaleziono ramkę:

Jeśli nasza ramka jest aktualna lub założona jest na nią blokada co oznacza ,że realizowana jest transmisja z dysku to zlecamy wczytanie następnych stron w przeciwnym razie ramka jest nieaktualna i przechodzimy do etykiety ramka bład - zlecanie wczytania następnych ramek w tym przypadku nie ma sensu bo i tak za chwile będziemy wczytywać ramkę jeszcze raz i tą decyzję możemy podjąć wtedy . Następnie (wracając do poprzedniego przypadku) czekamy na ramkę i w zależności od tego czy transmisja zakończyła się pomyślnie (ramka aktualna) przechodzimy do stanu sukces lub ramka błąd .

- Stan sukces:

Kopiujemy dane do bufora użytkownika i aktualizujemy zmienne lokalne (ilość przeczytanych zmiennych itp.) W zależności od ilości bajtów pozostałej do przeczytania przechodzimy do stanu koniec pętli albo idziemy na początek pętli .

- Stan nie znaleziono ramki:

Pobieramy od systemy nową ramkę (możliwe ,że mamy nie wykorzystaną pod ręką). Umieszczamy ją w odpowiedniej kolejce haszującej . Zlecamy wczytanie do niej danych i przechodzimy do stanu znaleziono ramkę .

- Stan ramka błąd:

Próbujemy ponownie wczytać dane do ramki .Jeśli powtórnie nie powiedzie się to wychodzimy z pętli ,w przeciwnym wypadku idziemy do stanu sukces.

- Stan koniec pętli:

Uaktualniamy pozycje w pliku i czas modyfikacji pliku. Zwracamy ilość wczytanych bajtów .

5 Funkcje systemowe systemu plików (część 2).

autor: Jan Palczewski

5.1 Kopiowanie deskryptorów - funkcje `dup` i `dup2`

Funkcje `dup` i `dup2` służą do duplikowania deskryptorów plików. Zduplikowany deskryptor wskazuje na tą samą strukturę w tablicy plików, jest drugą nazwą dla wejściowego deskryptora. I niczym więcej. Wszystkie informacje przechowywane są w tablicy plików, zaś deskryptor jest tylko rodzajem indeksu do niej. Składnia wywołania obu funkcji jest następująca:

```
dup( deskryptor );
int dup2( stary_deskryptor, nowy_deskryptor );
```

Funkcja *dup* znajduje pierwszą wolną (tzn. o najmniejszym numerze) pozycję w tablicy deskryptorów i kopiuje na nią zawartość pozycji `deskryptor`. Zwiększa o jeden licznik dowiązań we wskazywanej strukturze z tablicy plików. Działanie *dup2* jest podobne, jednak przy wywołaniu proces określa, na którą pozycję chce zdublować `stary_deskryptor`. Pozycja ta może być jednak zajęta. Wywoływana jest wtedy funkcja *close*, zamykająca wskazany deskryptor. Należy odróżnić zamykanie deskryptora od zamykania pliku. Funkcja systemowa *close* zmniejsza licznik dowiązań w strukturze tablicy plików wskazywanej przez podany jej deskryptor i dopiero gdy licznik ten osiągnie wartość zerową, odbywa się zamknięcie pliku.

Z powyższego opisu mogłoby wynikać, że duplikowanie deskryptorów nie ma sensu. Jest jednak ono często wykorzystywane, szczególnie w interpretatorach poleceń, gdzie przy jego pomocy dokonuje się przekierowania wejścia i wyjścia programów oraz tworzenia rurek (*pipe'ów*). Standardowo programy używają trzech deskryptorów 0,1,2, odpowiadających standardowym potokom wejściowemu, wyjściowemu i błędów. Aby wykonać przekierowanie `ls > ala`, interpretator poleceń powinien przed wykonaniem funkcji *exec* spowodować, by dane zapisywane do deskryptora 1 zapamiętywane były w pliku `ala`. Można to wykonać poniższym fragmentem kodu:

```
if( ( fd = open( "ala", O_RDONLY || O_CREAT ) ) == -1 ) błąd();
if( dup2( fd, 1 ) == -1 ) błąd();
// Deskryptor fd jest już niepotrzebny.
if( close( fd ) == -1 ) błąd();
```

W starszych wersjach systemu, gdzie nie było funkcji *dup2*, wykonywało się to korzystając z *dup*:

```
if( ( fd = open( "ala", O_RDONLY || O_CREAT ) ) == -1 ) błąd();
// Zakładamy, że deskryptor 0 jest używany. W normalnych warunkach nie
// powinien być zamknięty.
if( close( 1 ) == -1 ) błąd();
// Przydzielając najniższy wolny deskryptor funkcja dup użyje 1.
if( dup( fd ) == -1 ) błąd();
// Deskryptor fd jest już niepotrzebny.
if( close( fd ) == -1 ) błąd();
```

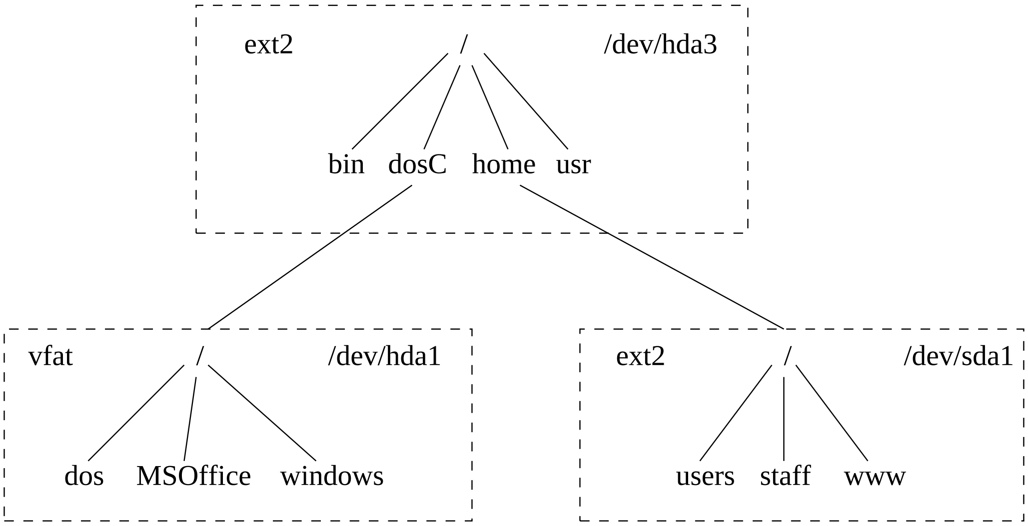
5.2 Montowanie systemów plików

5.2.1 Wstęp

W systemie plików Linux'a nie ma wielu niezależnych drzew katalogów (jak w DOS'ie). Istnieje jedno drzewo katalogów, którego części mogą fizycznie znajdować się na różnych dyskach podłączonych do różnych kontrolerów, a nawet na innych komputerach, z którymi system porozumiewa się poprzez sieć (Network File System). Z punktu widzenia użytkownika całość ta jest jednym systemem plików, tak jakby rezydowała na jednej partycji dysku.

Na rys. 1 znajduje się przykładowe drzewo katalogów systemu plików Linux'a. Współistnieją w nim obok siebie fizycznie zupełnie różne systemy plików – zarówno system ext2, jak i vfat.

Dzieje się tak dzięki stworzeniu abstrakcji systemu plików VFS (Virtual File System), która została omówiona we wcześniejszych rozdziałach.



Rysunek 1: Przykładowe drzewo katalogów

Główny system plików, znajdujący się na urządzeniu blokowym /dev/hda3, został wczytany i zainstalowany przez jądro w momencie startu systemu. Pozostałe zostały zamontowane później. Oto jak mogły wyglądać komendy wykonujące te operacje:

```
# mount -t vfat /dev/hda1 /dosC
# mount -t ext2 /dev/sda1 /home
```

Operacja zamontowania systemu plików wymaga podania zarówno katalogu, w którym zostanie zamontowany system plików (ścieżką do jego korzenia staje się katalog zamontowania - na rysunku to `/home` i `/dosC`, zaś wszystkie podkatalogi montowanego systemu plików stają się podkatalogami katalogu zamontowania), jak i typu systemu plików (**vfat**, **ext2** i in.). Zwykle także podaje się scieżkę do urządzenia blokowego, na którym znajduje się system plików. Są jednak od tego odstępstwa, np. **procfs** czy **NFS**, które urządzenia blokowego nie używają.

5.2.2 Montowanie systemów plików

Do montowania systemu plików służy funkcja systemowa *mount*. Komenda **mount**, przytoczona powyżej, korzysta właśnie z tej funkcji, aby wykonać swoje zadanie. Mimo, że program mount dokonuje pewnego sprawdzenia parametrów, to dopiero po przekazaniu sterowania do jądra odbywa się pełna weryfikacja, czy możliwe jest wykonanie postawionego zadania. Na samym początku jądro sprawdza, czy proces wywołujący funkcję systemową *mount* ma uprawnienia administratora. Procesom użytkowników nie wolno wykonywać funkcji, które mogą w takim stopniu zachwiać integralność systemu. Następnie VFS przegląda listę znanych systemów plików w poszukiwaniu struktury *file_system_type* odpowiadającej podanej nazwie systemu plików (w powyższym przykładzie to **ext2** i **vfat**). Struktura ta zawiera m.in. wskaźnik do procedury, która jest wołana przez VFS w celu wczytania superblock'u i zainicjalizowania struktur danych przy montowaniu systemu plików. Jeśli poszukiwanie zakończyło się fiaskiem, to jeszcze pozostaje jeden ratunek. Gdy aktywna jest opcja *kernel daemon'a*, to jądro zwraca się do **kerneld** z żądaniem wczytania modułu do obsługi potrzebnego systemu plików. Jeśli i to się nie powiedzie, to *mount* zwraca błąd.

Jądro identyfikuje urządzenie blokowe zawierające system plików do zamontowania (dla systemów plików nie wymagających urządzenia blokowego ten etap jest pomijany – **procfs**, **NFS**). Sprawdza także, czy ten system plików nie jest już wykorzystywany. Pobiera i-węzeł katalogu zamontowania, upewniając się, że nie jest on już punktem zamontowania, ani nie jest wykorzystywany przez żaden inny proces. Zatem katalog zamontowania nie może być katalogiem bieżącym żadnego procesu, ale nic nie przeszkadza temu, aby jeden z jego podkatalogów był. Wtedy dokonanie montowania spowoduje pewne perturbacje w działalności procesu, ale nie zdestabilizuje systemu.

Następnie VFS tworzy nową strukturę *super_block*, zawierającą m.in. i-węzeł korzenia katalogów montowanego systemu plików, i wywołuje właściwą procedurę inicjalizującą, do której wskaźnik zapisany jest w *file_system_type*. Wszystkie struktury *super_block* VFS'u są przechowywane w tablicy *super_blocks* i tam też muszą się znaleźć informacje o nowo wczytanym systemie plików. Przyjrzyjmy się dokładniej działaniu procedury inicjującej *super_block*. Wczytuje ona dane z urządzenia blokowego starając się wydobyć niezbędne informacje dla jądra. O ile dla systemu **ext2** jest to stosunkowo łatwe ze względu na podobieństwo jego struktur danych do VFS'u, to w przypadku **vfat'u** wymaga już bardziej skomplikowanych działań. Inaczej jeszcze odbywa się gromadzenie tych danych dla systemów plików, które nie rezydują na urządzeniach blokowych. W przypadku **NFS** zamiast odwołania do dysku procedura inicjalizacyjna łączy się poprzez sieć do serwera NFS i od niego dostaje potrzebne informacje.

Każdy zamontowany system plików jest opisany przez strukturę *vfsmount*, która przechowuje nazwę katalogu zamontowania, numer i nazwę urządzenia blokowego oraz wskaźnik do właściwego *super_block'u*. Struktury *vfsmount* połączone są w listę. Zatem następnym etapem montowania jest stworzenie takiej struktury i wstawienie jej na listę. Pozostało jeszcze zainicjowanie pól, które umożliwią późniejsze przekraczanie punktów montowania w alogorytmie *namei*. W strukturze *super_block* znajdują się dwa pola: *s_covered* i *s_mounted*. Pierwsze z nich wskazuje na i-węzeł katalogu zamontowania, zaś drugie na i-węzeł korzenia katalogów montowanego systemu plików. Oba te i-węzły będą stale w pamięci do czasu odmontowania systemu plików. W i-węźle katalogu montowania znajduje się pole *i_mount*, które wskazuje na i-węzeł *s_mounted* (w innych przypadkach pole *i_mount* ma wartość `NULL`). Mimo, że na dysku nie ma miejsca na zapisanie powyższych informacji, to ponieważ opisane struktury rezydują w pamięci przez cały okres użytkowania systemu plików, to nie ma obawy o ich stracenie i tym samym uniemożliwienie poruszania się po drzewie katalogów.

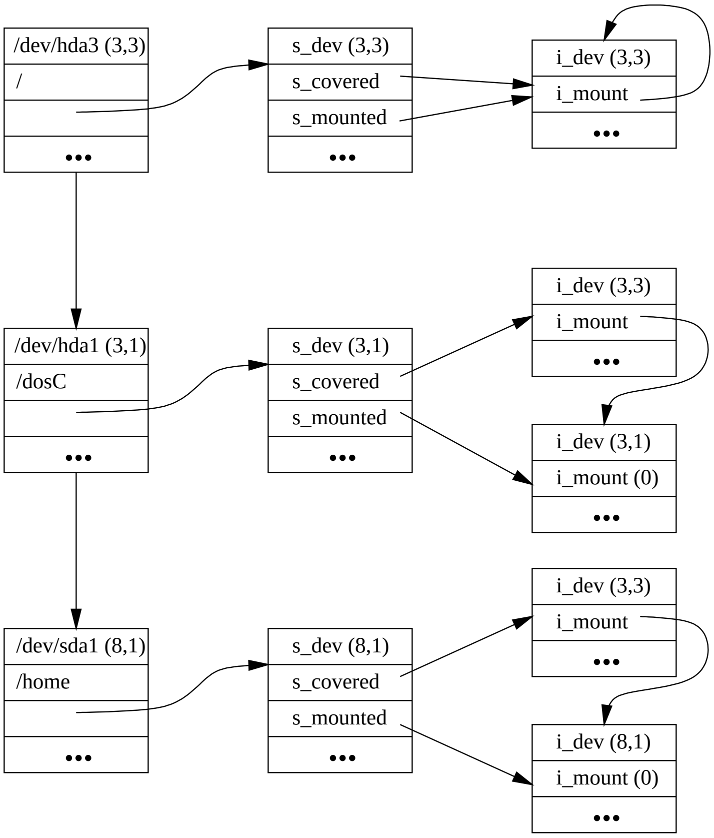
Na rys. 2 znajdują się struktury jądra przechowujące dane o drzewie katalogów zobrazowanym na rys. 1. Punktem zamontowania dla systemu plików `/dev/sda1` jest katalog `/home`, który znajduje się na `/dev/hda3`. W i-węźle `/home`, który rezyduje w pamięci, pole *i_mount* wskazuje na i-węzeł korzenia katalogów `/dev/sda1`. Nic nie stoi na przeszkodzie, by w katalogu `/home/users` zamontować kolejny system plików, lecz nie uda się to już dla `/home`, ani dla `/dosC`, gdyż są one wykorzystywane jako punkty montowania.

5.2.3 Przekraczanie punktów zamontowania podczas analizy nazw ścieżkowych

Wprowadzenie możliwości montowania systemów plików wprowadziło pewne utrudnienia w analizie nazw ścieżkowych algorytmem *namei*. Mianowicie, problemy stwarzają punkty zamontowania. Poniższe przykłady pokażą, jak jądro Linux'a radzi sobie z nimi. Odwołajmy się znów do rys. 1. Przypuśćmy, że użytkownik wykonuje następujące komendy:

```
# cd /home/users
# cd ../..
```

Prześledźmy wykonanie pierwszej z nich. Główną jej częścią jest pobranie i-węzła katalogu `/home/users`. Wywoływana jest zatem funkcja `namei("/home/users")`. Pobiera ona i-węzeł korzenia drzewa katalogów, wczytuje plik opisywany przez ten i-węzeł szukając pozycji odpowiadającej nazwie `home`. Znaleziona pozycja zawiera numer i-węzła `home`. Wywołuje zatem *iget*, która, zgodnie z tym co zostało powiedziane w poprzednim rozdziale, znajduje szukany i-węzeł w pamięci. Stwierdza, że pole *i_mount* ma wartość różną od `NULL`. Jest to punkt zamontowania. Nie może więc zwrócić tego i-węzła, gdyż z logicznego punktu widzenia on nie istnieje. Przecież plik `/home`



Rysunek 2: Struktury danych

znajduje się już w systemie plików `/dev/sda1`. Z pomocą przychodzi pole *i_ mount*, które zawiera wskaźnik do i-węzła korzenia drzewa katalogów zamontowanego systemu plików. W efekcie funkcja *iget* zwraca ten i-węzeł. Jądro rozpoczyna poszukiwanie pozycji `users` już na nowym systemie plików. Znajduje jej i-węzeł i ustala jako katalog bieżący procesu. Widać stąd, że przechodzenie z nadrzędnego do zamontowanego systemu plików wymaga tylko niewielkiej modyfikacji algorytmu *iget*.

Zupełnie inaczej wygląda wykonanie drugiej komendy. Tutaj przekraczamy punkt zamontowania w przeciwnym kierunku. Katalogiem bieżącym jest `/home/users`. Funkcja *namei* wywoływana jest z parametrem `"../.."`. Rozpoczyna ona zatem analizę nazwy ścieżkowej od katalogu bieżącego. Wczytuje go do pamięci szukając pozycji `..`. Znajduje ją i pobiera jej i-węzeł. Jest to i-węzeł korzenia katalogów systemu plików `/dev/sda1`. Wyszukanie w analogiczny sposób pozycji `..` w tym katalogu nie przyniosłoby oczekiwanych rezultatów (na przykład w systemie plików **`ext2`** pozycja ta wskazywałaby na tenże sam katalog). Jądro musi zatem obsłużyć tą sytuację w inny sposób. Sprawdza zatem, czy bieżący i-węzeł nie jest korzeniem katalogów w zamontowanym systemie plików. Każdy i-węzeł rezydujący w pamięci posiada wskaźnik do *super_ block'u* opisującego jego system plików, zaś w *super_ block'u* pamiętany jest i-węzeł korzenia katalogów. Wystarczy więc sprawdzić, czy zapamiętany i bieżący i-węzeł, to ten sam obiekt. W omawianym przypadku tak właśnie jest. Jądro pobiera i-węzeł punktu zamontowania, który znajduje się w *super_ block'u* w polu *s_ covered* i kontynuuje opisany już wyżej algorytm, czyli przeszukuje katalog, by znaleźć numer i-węzła katalogu `..`. Ta operacja jest już poprawna, gdyż wykonuje się na właściwym systemie plików (na `/dev/hda3`). Widać zatem, że przekraczanie punktu zamontowania przy przejściu od zamontowanego do nadrzędnego systemu plików wymaga niewielkiej modyfikacji samej funkcji *namei*.

5.2.4 Odmontowywanie systemów plików - `umount`

W wielu momentach działania systemu operacyjnego może zajść konieczność odmontowania któregoś z zamontowanych systemów plików. Zawsze wykonywana jest ta operacja w czasie zamykania systemu. Oczywiście wolno ją wykonywać tylko administratorowi. Odmontowanie systemu plików `/dev/hda1` z rys. 1 może być wykonane na dwa sposoby:

```
# umount /dev/hda1
# umount /dosC
```

Program **umount**, podobnie jak **mount**, dokonuje wstępnego sprawdzenia parametrów i wywołuje funkcję systemową *umount*. Odpowiednie jej wywołania będą miały postać: `umount("/dev/hda1")` oraz `umount("/dosC")`. Zauważmy, że za pierwszym razem podana jest ścieżka do urządzenia blokowego, zaś za drugim – punkt zamontowania, nie specyfikując, jaki rodzaj informacji został przekazany. Jak zatem jądro sobie radzi? Otóż funkcja *umount* pobiera i-węzeł pliku, na który wskazuje jej parametr. Jeśli jest to i-węzeł urządzenia blokowego, to wiadomo, że został przekazany parametr pierwszego typu. W przeciwnym przypadku musi być to punkt zamontowania. Narzuca się od razu pytanie: czy znajomość tylko punktu zamontowania albo tylko urządzenia blokowego wystarczy, by jednoznacznie zidentyfikować system plików? Okazuje się, że tak. Jeśli znamy urządzenie blokowe, to możemy pobrać dla niego *super_block* przeszukując tablicę *super_blocks*. Gdy zaś mamy i-węzeł punktu zamontowania, to jedno z jego pól wskazuje na *super_block* systemu plików, na którym się znajduje. Nie jest to jednak cecha właściwa tylko i-węzłowi punktu zamontowania. Z punktu widzenia identyfikacji systemu plików można byłoby podać ścieżkę do dowolnego pliku znajdującego się na tym systemie plików, lecz sprzyjałoby to popełnianiu błędów przez administratora. Dlatego funkcja *umount* sprawdza, czy podany i-węzeł jest punktem zamontowania (w analogiczny sposób jak w algorytmie *namei*) i jeśli nim nie jest, to zwraca błąd.

Bardzo ważne jest założenie, jaki przyjeli twórcy Linux'a. Nie można odmontować systemu plików, na którym odbywają się jakieś operacje dyskowe, są otwarte pliki itp. Gdyby nie to, to odmontowanie systemu plików byłoby podobne do wyjęcia dyskietki, bez zwracania uwagi, czy w danym momencie dokonuje się na niej zapisu. A jak wiadomo kończy się to błędnymi, żeby nie powiedzieć bzdurnymi, danymi, które w efekcie znajdą sie na nośniku. Z tego powodu przed rozpoczęciem operacji odmontowania, jądro sprawdza, czy w pamięci nie ma żadnego aktywnego i-węzła należącego do odmontowywanego systemu plików oczywiście pamiętając, że i-węzeł korzenia drzewa katalogów może mieć licznik odwołań równy dwa (dowiązanie do niego jest zarówno w *super_block'u*, jak i w i-węźle katalogu zamontowania). Jasne jest zatem, dlaczego nie można odmontować systemu plików, jeśli katalog bieżący jakiegoś procesu się na nim znajduje. Nie da się także odmontować systemu plików, jeśli znajduje się na nim punkt zamontowania.

Następnie jądro przywraca wygląd drzewa katalogów sprzed operacji montowania. Wystarczy tylko wyczyścić pole *i_mount* w i-węźle punktu zamontowania. Zwalniane są i-węzeł punktu zamontowania i i-węzeł korzenia drzewa katalogów odmontowywanego systemu plików (*iput*); były one pobrane w czasie wykonywania *mount* i od tamtego czasu rezydowały w pamięci. Ponadto w razie potrzeby zapisywany jest na dysk (urządzenie blokowe) *super_block*. Usuwana jest z listy *vfsmnt* pozycja odpowiadająca odmontowywanemu systemowi plików. Na samym końcu, kiedy już żaden proces nie ma dostępu do systemu plików, zapisywane są wszystkie zmodyfikowane bloki należące do urządzenia blokowego, na którym ten system plików się znajdował.

5.3 Tworzenie i usuwanie dowiązań do plików

Jądro udostępnia dla procesów dwie funkcje:

```
link("ścieżka_istniejącego_dowiązania","ścieżka_nowego_dowiązania");
unlink("ścieżka_dowiązania");
```

Pierwsza z nich służy do tworzenia dowiązań, zaś druga do ich usuwania. Dowiązanie to nazwa pliku w drzewie katalogów. Posiada ona zawsze numer skojarzonego i-węzła (musi się on znajdować na tym samym systemie plików). Do jednego i-węzła może być wiele dowiązań. Ich liczba przechowywana jest w specjalnym polu, zwanym *licznikiem dowiązań*. Przykładem może być pakiet **gcc**, który w katalogu `/usr/bin` tworzy dwie pozycje `c++` oraz `g++`, które wskazują na ten sam i-węzeł. Są to dwie rownorzędne nazwy tego samego programu.

Funckja *unlink* usuwa wskazane dowiązanie. Wykonanie `unlink("/usr/bin/c++")` spowoduje, że w katalogu `/usr/bin` nie będzie pozycji `c++`, lecz kompilator C++ będzie nadal dostępny pod nazwą `/usr/bin/g++`.

Przejdźmy do dokładniejszego opisu algorytmów. W systemie Linux, ze względu na ogromne różnice pomiędzy obsługiwanymi systemami plików, warstwa ogólna *VFS'u* wykonuje niewiele operacji związanych z opisywanymi funkcjami. Gros zadań wykonywanych jest przez specjalizowane procedury poszczególnych systemów plików. Postanowiłem więc opisać dokładniej implementację włąściwą dla systemu plików **ext2**, który jest macierzystym systemem plików Linux'a. Istnieje dla niego dodatkowe ograniczenie. Nie można tworzyć i usuwać dowiązań do katalogów.

link("nazwa_istniejącego_pliku","nazwa_nowego_pliku")

Jądro pobiera i-węzeł pierwszego parametru (`stary`) wykonując algorytm *namei*. Pobiera także i-węzeł katalogu, w którym ma stworzyć nowe dowiązanie (`nowy`). Sprawdza prawa dostępu do obydwu i-węzłów, weryfikując, czy znajdują się one na jednym systemie plików. Oczywiście tworzenie nowej pozycji w katalogu wiąże się z prawem pisania w tym katalogu. Następnie jądro opuszcza semafor na i-węźle `nowy`, tym samym zapewniając sobie wyłączność manipulowania katalogiem. Semafor ten jest używany ilekroć dochodzi do modyfikacji zawartości katalogu (dodania nowej pozycji, skasowania istniejącej), a zatem w czasie wykonania takich fukcji jak `unlink`, `creat`, czy `mknod`. W tym miejscu sterowanie oddawane jest do funkcji właściwej dla obsługiwanego systemu plików. Zgodnie z przyjętym założeniem wywoływana jest fukcja *ext2_link*. Przegląda ona katalog `nowy` sprawdzając, czy nie nie ma w nim już pozycji `nazwa_nowego_pliku`. Jeśli znalazła ją, to zwraca błąd. W przeciwnym przypadku tworzy nową pozycję i zaznacza ją do jak najszybszego zapisu (nie jest to jednak zapis natychmiastowy, ale znacznie wcześniejszy niż dla bloków normalnych plików). Następnie zwiększa licznik dowiązań do i-węzła `stary`, zaznacza go jako zmodyfikowany i wykonuje algorytm *iput*. Nie oznacza to jednak, że i-węzeł zostanie szybko zapisany. Może być on przecież używany przez inne procesy lub nawet przez bieżący proces, który może mieć otwarty plik `stary` (tym samym używać tego i-węzła). W tym miejscu kończy się wykonanie funkcji *ext2_link* i sterowanie powraca do funkcji ogólnej warstwy *VFS'u*. W niej podnoszony jest semafor i-węzła `nowy`, tym samym informując, że modyfikacje katalogu się zakończyły. I-węzeł ten jest następnie zwalniany (*iput*).

unlink("nazwa_pliku")

Jądro pobiera i-węzeł katalogu, w którym znajduje się plik wskazany ścieżką `nazwa_pliku` (*namei*) i sprawdza prawa dostępu procesu: czy może on wykonywać (czyli pobierać i-węzły pozycji w katalogu) i pisać w katalogu. Następnie, podobnie jak w funkcji *link*, opuszcza semafor na pobranym i-węźle i oddaje sterowanie do funkcji *ext2_unlink*. Funkcja pobiera i-węzeł `nazwa_pliku` (*namei*) i sprawdza, czy nie jest on katalogiem (do usuwania katalogów służy funkcja *rmdir*). Zmniejsza jego licznik dowiązań i usuwa z katalogu odpowiadającą mu pozycję, zaznaczając ją do jak najszybszego zapisu. Dopiero później zwalnia i-węzeł samego pliku. Nie oznacza to jednak, że plik ten przestaje w tym momencie fizycznie istnieć na dysku. Jego bloki i pozycja w tablicy i-węzłów zostają zwolnione dopiero po tym, jak licznik odwołań i-węzła osiągnie wartość zerową. Na tym kończy się działanie *ext2_unlink*. Funkcja ogólnej warstwy *VFS'u* podnosi jeszcze semafor na i-węźle katalogu, w którym znajdował się plik `nazwa_pliku`, zwalnia ten i-węzeł (*iput*) i kończy swoje działanie.

W lepszym zrozumieniu, kiedy zwalniane są bloki pliku, może pomóc poniższy przykład. Jest to sposób na stworzenie idealnego pliku tymczasowego, który po zakończeniu procesu zawsze przestanie istnieć.

```
int twórz_tymczasowy( const char * nazwa ) {
```

```
    int fd;

    if( ( fd = open( nazwa, O_CREAT || O_RDWR ) ) == -1 ) return -1;
    if( ( unlink( nazwa ) == -1 ) return -1;
    return fd;
}
```

Plik utworzony powyższą funkcją będzie istniał dotąd, aż proces, który go utworzył, nie zamknie go lub się nie zakończy (wtedy system sam zamknie wszystkie otwarte pliki). Nie będzie miał jednak żadnego dowiązania w drzewie katalogów zabezpieczając tym samym przed dostępem przez inne procesy.

5.4 Tworzenie plików specjalnych

Funkcja systemowa *mknod* tworzy pliki urządzeń (znakowe i blokowe), łącza nazwane i puste pliki zwykłe (lecz nie katalogi, gdyż do tego służy oddzielna fukcja *mkdir*). Przydziala ona zarówno nowy i-węzeł, jak i dowiązanie do niego. Składnia jej wywołania jest następująca:

`mknod("nazwa_pliku",typ_i_prawa,urządzenie);`

Parametr `nazwa_pliku` określa nazwę dowiązania w drzewie katalogów do tworzonego i-węzła, `typ_i_prawa` – rodzaj i-węzła i prawa dostępu do niego, zaś pole `urządzenie` jest używane tylko w przypadku tworzenia pliku specjalnego urządzenia i specyfikuje jego numer główny i drugorzędny. Jądro zapewnia, że proces o prawach innych niż administratora może tworzyć wyłącznie łącza nazwane (opisane dalej).

Algorytm funkcji *mknod* jest bardzo prosty. Tutaj, podobnie jak w przypadku tworzenia i usuwania dowiązań, gros pracy odbywa się w wyspecjalizowanych procedurach odpowiedniego systemu plików. Przedstawiam więc implementację dla systemu plików **ext2**.

W pierwszej fazie *VFS* sprawdza, czy proces ma prawo do wykonania żądania. Następnie pobiera i-węzeł katalogu, w którym ma zostać stworzone dowiązanie i zakłada na niego blokadę. Oddaje sterowanie do funkcji *ext2_mknod*, która sprawdziwszy, że w plik `nazwa_pliku` jeszcze nie istnieje w drzewie katalogów, żąda przydzielenia nowego i-węzła (*ext2_new_inode*). Wypełnia odpowiednio jego pola i dodaje pozycję do katalogu zaznaczając ją do jak najszybszego zapisu. Zwalnia następnie nowoutworzony i-węzeł (*iput*), który także zostaje zaznaczony do szybkiego zapisu. Dzieje się tak, gdyż licznik odwołań spada do zera (nikt inny nie może jeszcze go używać, bo dopiero co powstał) i *iput* ustawia i-węzeł, jako zmodyfikowany, do zapisu. Sterowanie powraca do ogólnej warstwy *VFS'u*, która zdejmuje założoną wcześniej blokadę i zwalnia i-węzeł katalogu.

5.5 Łącza nazwane i nienazwane

5.5.1 Co to jest?

Łącze jest rodzajem kolejki prostej (FIFO) o ograniczonej wielkości. Porządek obsługi danych jest zgodny z modelem FIFO, jednak ze względów implementacyjnych wprowadzono ograniczenie na wielkość bufora, w którym przechowywane są dane. Wymusiło to oczywiście zmianę semantyki funkcji piszących do kolejki – czasami nie mogą one zapisać danych z powodu braku wolnego miejsca.

Łącza nazwane różnią się od nienazwanych tylko sposobem, w jaki procesy zaczynają z nich korzystać. Łącze nazwane identyfikowane jest przez plik specjalny (tworzony przez funkcję *mknod*). Proces chcący korzystać z takiego łącza otwiera plik wykonując na nim operację *open*, tą samą, co w przypadku zwykłych plików. Dostęp do łącza jest ustalany tak samo, jak w przypadku innych plików, przy wykorzystaniu pól określających właściciela i prawa do czytania i pisania. Z takiego łącza korzystać mogą procesy, które nie są ze sobą spokrewnione, co, jak okaże się wkrótce, nie jest możliwe dla łącz nienazwanych.

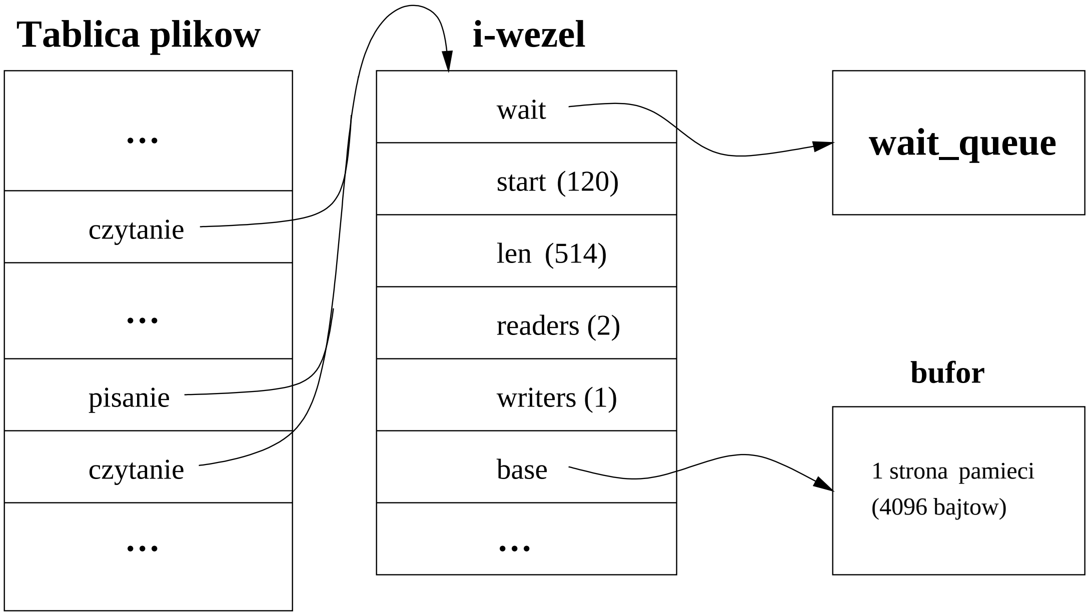
Łącza nienazwane tworzone są przez funkcję systemową *pipe*, która przekazuje procesowi wołającemu dwa deskryptory: do czytania i do pisania. Jest to jedyny sposób na odwołanie się do tego łącza, zatem korzystać mogą z niego proces, który je utworzył, i jego potomkowie.

Na czym polega siła łącz. Otóż czytanie i pisanie do nich jest wykonywane przez te same funkcje systemowe, co w przypadku plików. Można zatem traktować łącza jako rodzaj pliku i używać do ich obsługi standardowych bibliotek. Interpretatory poleceń korzystają z łączy, by umożliwić tworzenie rurek, czyli łączyć wyjście jednego programu z wejściem drugiego – powstaje potok przetwarzający dane. Wydaje się, że jest to jedno z podstawowych zastosowań łączy nienazwanych. Łącza nazwane mogą natomiast być używane do zepewniania komunikacji pomiędzy procesami (korzysta z nich program Midnight Commander, by zapewnić sobie kontakt z interpretatorem poleceń). Poniższy przykład pokazuje, jak można użyć obu typów łącz do zbudowania zamkniętego łańcucha procesów (w *bash'u*):

```
> mknod polaczenie p
> ogniwo < polaczenie | ogniwo | ogniwo | ... | ogniwo > polaczenie
```

5.5.2 Struktury danych

Na rys. 3 przedstawione są struktury danych jądra wykorzystywane do obsługi łącza. Łącze ma swój i-węzeł, do którego odwołują się pozycje z tablicy plików. Taka organizacja umożliwia dostęp do łącza tak, jakby było zwykłym plikiem.



Rysunek 3: Struktury danych

Przyjrzyjmy się niektórym polom i-węzła łącza. Mamy tam wskaźnik do bufora kolejki (*base*), który jest jedną pełną stroną pamięci (4096 bajtów), alokowaną w momencie tworzenia i-węzła w pamięci (czyli w czasie wykonania *pipe* lub pierwszego otwarcie łącza nazwanego). Bufor ten jest zwalniany w czasie usuwania i-węzła z pamięci, gdy jego licznik odwołań spadnie do zera. W przypadku łącza nazwanego może to oznaczać utratę pewnych danych, które nie zostały przeczytane, a mogłyby być w przyszłości, lecz łącze służy do komunikacji bezpośredniej, niemalże w czasie rzeczywistym, więc takie zachowanie jest do przyjęcia. Pokazuje ono też, że istnienie pliku łącza w drzewie katalogów nie oznacza, że samo łącze tam się znajduje. Jest to tylko identyfikator łącza, które w całości implementowane jest w pamięci operacyjnej.

Pole *wait* jest wskaźnikiem do kolejki procesów oczekujących na zapisanie lub odczyt danych z łącza. Nazwa kolejka jest trochę myląca, gdyż jest to zbiór procesów, które są budzone wszystkie na raz, bez możliwości ich szeregowania. W dalszej części opisu łączy wyjaśnione będzie, kiedy dokładnie procesy korzystają z tej kolejki.

Bardzo ważnymi polami są *readers* i *writers*, które przechowują liczbę czytelników i pisarzy danego łącza, czyli liczbę odpowiednich dowiązań z tablicy plików. Są wykorzystywane przez procedury obsługi łącza do określenia jego stanu. Przykładowo, nie ma sensu czekać na nowe dane w łączu, jeśli liczba pisarzy, którzy mogliby je wprowadzić, jest zerowa.

5.5.3 Operacje blokujące i nieblokujące

Łącza zaimplementowane w jądrze Linux'a dają możliwość wyboru pomiędzy dwoma trybami obsługi sytuacji wyjątkowych: trybem blokującym i trybem nieblokującym. Przypomnijmy, że często występującymi zdarzeniami są brak miejsca w buforze do zapisania danych oraz brak danych do odczytania. W tej sytuacji jądro może zareagować dwojako. Albo zawiesić proces, by oczekiwał na zaistnienie warunków umożliwiających wykonanie żądanej operacji, albo zwrócić błąd. Pierwsze z tych rozwiązań stosuje się dla łącz blokujących, zaś drugie dla nieblokujących.

5.5.4 Tworzenie łącza

Zostało wcześniej wspomniane, iż są dwa sposoby rozpoczęcia korzystania z łącza. W przypadku łączy nazwanych jest to otwarcie pliku łącza funkcją systemową *open*, zaś łącza nienazwane tworzy się przy pomocy *pipe*. Zajmijmy się tą drugą funkcją. Jej implementacja jest bardzo prosta. Jądro żąda przydzielenia pustego i-węzła w pamięci, inicjuje jego pola i alokuje jedną stronę pamięci na bufor. Następnie przydziela sobie dwie wolne pozycje z tablicy plików. Jedną z tych pozycji inicjuje, jako plik tylko do czytania, zaś drugą – tylko do pisania. Znajduje dwa wolne deskryptory plików w tablicy deskryptorów procesu i ustawia je, aby wskazywały na zainicjowane wcześniej elementy tablicy plików. Na tym kończy działanie, zwracając procesowi dwa przydzielone deskryptory.

Alogrytm otwarcia łącza nazwanego jest prawie identyczny z algorytmem otwierania zwykłego pliku. Funckja systemowa *open* odczytuje i-węzeł łącza (*iget*). Jeśli i-węzeł ten nie ma jeszcze zaalokowanego bufora, to przydziela na niego jedną stronę pamięci. Następnie postępuje zależnie od otrzymanych parametrów. Może otwierać łącze do pisania, do czytania albo na raz do czytania i pisania. Dodatkowo jeszcze można zmienić jej reakcję określając, czy operacja ma być blokująca, czy nie. Rozważmy najpierw działanie z opcją blokowania. Wtedy algorytm otwarcia tylko do czytania w przypadku braku procesów piszących usypia bieżący proces w oczekiwaniu na pojawienie się pisarzy. Analogicznie, proces otwierający tylko do pisania oczekuje na pojawienie się procesów czytelników. W przypadku, gdy żądamy otwarcia zarówno do czytania, jak i do pisania, blokada nigdy nie nastąpi. W najgorszym wypadku proces sam będzie przesyłał informacje do siebie. Zauważmy, że przy takiej implementacji łącz nazwanych nie jest możliwe zasymulowanie działania funkcji *pipe* bez stosowania sztuczek. Otwierając łącze do czytania proces nie może już go otworzyć do pisania, gdyż pierwsza z tych operacji spowoduje jego uśpienie. Z tego i podobnych powodów Linux umożliwia nieblokujące otwieranie łącz nazwanych. Funkcja *open* kończy swoje działanie bezzwłocznie nawet, gdy brakuje pary do otwarcia łącza (łącze zostaje otwarte, gdy żądanie dotyczy czytania, gdy zaś dotyczy pisania, to zwracany jest błąd).

5.5.5 Funkcje czytanie i pisania do łącza

Powyżej została przekazana już większość informacji dotyczących funkcji czytania i pisania do łącza. Przypomnijmy, że funkcje te są identyczne, jak dla normalnych plików umożliwiając traktowania łączy jako standardowego wejścia czy wyjścia. Ponadto możliwe są modyfikacje działania tych funkcji poprzez użycie opcji nieblokowania. Przyjrzyjmy się dokładniej, kiedy w czasie wykonywania żądania odczytu danych z łącza może nastąpić uśpienie procesu i jakie inne sytuacje wyjątkowe mogą się wydarzyć.

Funkcja *read* odczytuje z łącza tyle danych, ile się w nim znajduje, lecz nie więcej niż zostało ustalone przy jej wywołaniu. Może się jednak zdarzyć, że bufor łącza jest pusty. Jeśli nie ma żadnych pisarzy, to jądro informuje o tym proces zwracając wartość 0. Jeśli jednak są jacyś pisarze, to dalsze działanie zależy od aktywności opcji nieblokowania. Jeśli jest wyłączona, to proces zostaje uśpiony (w kolejce *wait*) w oczekiwaniu na pojawienie się danych w buforze; ze względu na implementację kolejki procesów uśpionych nie można zakładać, że kolejność dostępu do danych będzie taka sama, jak kolejność zgłoszeń. Jeśli opcja nieblokowania jest włączona, to w takiej sytuacji jądro zwraca błąd `EAGAIN` informując, żeby proces spróbował ponownie później.

Funkcja *write* zapisuje do łącza podaną ilość danych. Pierwszą operacją jest sprawdzenie, czy jakiś proces będzie mógł te dane przeczytać, czyli innymi słowy czy istnieją jacyś czytelnicy. Jeśli ich nie ma, to wywoływany jest sygnał `SIGPIPE` i zwracany błąd. Gdy czytelnicy istnieją, to jądro próbuje umieścić dostarczone dane w buforze łącza. Dane wielkości nie przekraczającej wielkości łącza (czyli 4096 bajtów) muszą zostać zapisane atomowo. Niestety algorytm zapewniania tego warunku (w przypadku pisania blokującego) nie gwarantuje, że nie dojdzie do zagłodzenia. Proces czeka w pętli (zasypiając na wspomnianej kolejce *wait*) dotąd, aż w łączu nie zrobi się dostatecznie wiele miejsca, by mógł zapisać swoje dane. W tym czasie inne procesy o mniej wygórowanych żądaniach dostają dostęp do łącza pisząc do niego i czytając. Gdy operacja pisania jest nieblokująca, to po stwierdzeniu, że w buforze nie ma dość miejsca zwracany jest błąd `EAGAIN`. Gdy jednak żądanie zapisu dotyczy bufora większego niż 4096 bajtów, to działanie znacznie zależy od stanu opcji nieblokowania. Gdy jest ona włączona, jądro zapisuje tyle danych, ile się da, do bufora łącza (być może nie uda mu się nic zapisać) i informuje wywołujący proces o tym, ile zapisał. Gdy zaś operacja pisania jest blokująca, to jądro w pętli zapisuje tyle danych, ile jest miejsca w buforze, po czym usypia (na kolejce *wait*) czekając na możliwość zapisania kolejnej porcji. Nie jest jednak zapewniona spójność danych. Mogą występować przeploty wielu żądań zapisu, pomieszanie danych z wielu buforów.

5.5.6 Zamykanie łącza

Zamykanie łącza jest podobne do zamykania pliku. Występują jednak dwie bardzo ważne różnice. Po pierwsze jądro odpowiednio zmienia wartości pól *readers* i *writers* w i-węźle. Po drugie, gdy wartość licznika odwołań i-węzła spadnie do zera, to zwalniana jest pamięć bufora nawet, jeśli znajdowały się w nim jakieś dane.

Bibliografia

[1] Maurice J. Bach: Budowa systemu operacyjnego UNIX

[2] Zródła jądra systemu Linux 2.0.32

[3] LabLinux

[4] Projekt Linux

[5] Linux Kernel Internals

[6] Linux Kernel Hacker's Guide