Rozdział 1

Getblk

Grzegorz Kowalski <gk181054@students.mimuw.edu.pl>
Jarosław Zimon <jz181073@students.mimuw.edu.pl>

1.1 Bardzo krótki wstęp do pamięci buforowej

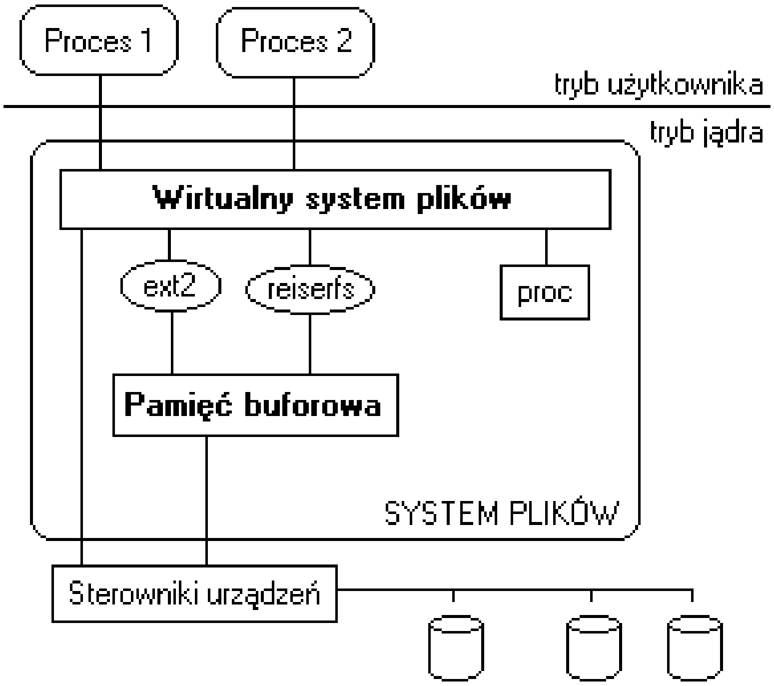
W większości systemów komputerowych elementem krytycznym dla szybkości wykonywania się procesu są operacje wejścia/wyjścia. Jest to spowodowane relatywnie długim czasem dostępu do urządzeń pamięci masowej w porównaniu do czasu odwołania się do jakiegoś adresu w pamięci operacyjnej. Aby usprawnić korzystanie z danych zapisanych w plikach systemy operacyjne (w szczególności także Linux) rezerwują pewien obszar pamięci jądra na bufory. Bufory to specjalne struktury, do których wczytywane są bloki danych z urządzeń blokowych. Dzięki ich stosowaniu średni czas odczytu (zapisu) danych z (do) pliku ulega znacznemu skróceniu, ponieważ w trakcie realizacji żądania sprowadzenia bloku z dysku do pamięci operacyjnej jądro często znajduje go w jakimś buforze i unika czasochłonnej operacji wejścia/wyjścia. Dodatkową zaletą jest też to, że użytkownik nie musi się przejmować faktem, że rzeczywiste operacje na dysku są operacjami blokowymi (może odwoływać się do dowolnego bajtu na dysku, a system tłumaczy to na odwołanie do pewnego bajtu w buforze). Odwołanie do dowolnego pliku można podzielić na trzy fazy (por. Rys. 1.1)

1. wysokopoziomowe algorytmy związane z wyszukaniem odpowiedniego i-węzła, sprawdzeniem uprawnień dostępu do pliku i konwersją parametrów żądania na odwołania do odpowiednich bloków dyskowych
2. algorytmy pamięci buforowej, czyli wyszukiwanie podanego bloku w puli buforów, ewentualne zainicjowanie transmisji dyskowej
3. niskopoziomowe funkcje szeregowania żądań transmisji z urządzeń blokowych, odwołania do sterowników urządzeń

Aby efektywnie i bezpiecznie zarządzać buforami (brak blokad i głodzenia procesów czekających na wczytanie bloku), jądro systemu operacyjnego korzysta z kilku funkcji, które, realizując odpowiednie algortymy, prowadzą szeroko pojętą administrację pamięcią buforową. Jednym z takich algorytmów jest getblk.

1.2 Miejsce buforów w systemie operacyjnym

Jak widać na rysunku 1.1 wirtualny system plików odwołuje się do urządzeń przez operacje na buforach.



Rysunek 1.1: Miejsce pamięci buforowej w systemie plików

1.3 Getblk - do czego służy

Funkcja `getblk()` jest wywoływana przy próbach odczytu i zapisu bloku dyskowego. Jej zadaniem jest odnalezienie w pamięci bufora, w który odwzorowany jest podany blok, a także dbałość o utrzymanie podstawowej własności pamięci buforowej, którą jest istnienie co najwyżej jednego bufora zawierającego dany blok dyskowy. Operacja wyszukiwania bufora może być wykonywana bardzo często, dlatego jej efektywna implementacja jest bardzo istotna dla wydajności systemu. Strukturą, w której przechowywane są nagłówki buforów z danymi jest tablica haszująca. Miejsce, jakie należy w niej przydzielić buforowi jest obliczane przez opisaną w punkcie 1.10 funkcję haszującą na podstawie numeru logicznego urządzenia blokowego oraz numeru bloku w jego obrębie. Rozwiązywanie kolizji realizowane jest przez listy cykliczne. W jednej takiej liście znajdują się nagłówki buforów odpowiadające blokom, dla których funkcja haszująca przyjmuje tę samą wartość. Należy podkreślić, że bufor zwracany przez funkcję `getblk()` (a dokładniej zawartość jego pola z danymi) nie musi być aktualnym obrazem tego, co znajduje się na dysku. Getblk 'martwi się' tylko o zwrócenie odpowiedniej struktury danych, natomiast weryfikacja zawartości bufora odbywa się już w funkcji wywołującej. Na przykład `getblk()` może zwrócić pusty bufor w przypadku, gdy żądany blok nie został odnaleziony w pamięci buforowej. Wtedy funkcja wołająca (np. `bread()`) zleca operację transmisji tego bloku z dysku do otrzymanego bufora.

1.4 Uwagi dotyczące nazewnictwa

Ponieważ funkcje wchodzące w skład modułu obsługi pamięci buforowej operują na blokach danych, terminem często pojawiającym się przy ich opisie jest 'urządzenie blokowe'. Dla własnej wygody oraz przez wzgląd na czytelnika przy pisaniu tego materiału często używaliśmy krótszego zamiennika 'dysk', mając na myśli dowolne urządzenie tego rodzaju.

1.5 Wykorzystywane struktury danych

Bufor składa się z dwóch części: tablicy zawierającej dane bloku dyskowego i nagłówka identyfikującego bufor. Nagłówek bufora jest strukturą `struct buffer_head`.

1.5.1 Opis pól struktury buffer_head

- numer logiczny urządzenia, do którego należy reprezentowany blok; stała B_FREE oznacza, że bufor jest wolny

  ```
  kdev_t b_dev;
  ```

- położenie logiczne bloku na danym urządzeniu logicznym

  ```
  unsigned long b_blocknr;
  ```

- stan bufora, zbiór wartości BH_* opisany dalej

  ```
  unsigned long b_state;
  ```

- czas, kiedy należy zapisać bufor na dysk; zero gdy zapis na dysk nie jest potrzebny

  ```
  unsigned long b_flushtime;
  ```

- rozmiar bufora - Linux dopuszcza bufory różnych rozmiarów

  ```
  unsigned long b_size;
  ```

- liczbia procesów korzystających z bufora

  ```
  unsigned int b_count;
  ```

- dowiązania do struktur, w których są przechowywane bufory - dwie kolejki dwukierunkowe

  ```
  struct buffer_head * b_next;
  struct buffer_head ** b_pprev;

  struct buffer_head * b_next_free;
  struct buffer_head * b_prev_free;
  ```

- numer listy `lru_list`, jedna ze stałych BUF_*

  ```
  unsigned int b_list;
  ```

- cykliczna kolejka jednokierunkowa buforów, które przechowują dane na tej samej stronie pamięci

  ```
  struct buffer_head * b_this_page;
  ```

- wskaźnik, pod którym znajdują się dane bloku; nagłówek nie musi mieć przydzielonej pamięci na dane, jeśli nie jest używany

  ```
  char * b_data;
  ```

- kolejka procesów oczekujących na odblokowanie tego bufora

  ```
  struct wait_queue * b_wait;
  ```

- funkcja zakończenia operacji we/wy na buforze; w szczególności wtedy następuje odblokowanie bufora

  ```
  void (*b_end_io)(struct buffer_head *bh, int uptodate);
  ```

1.5.2 Opis znaczenia stałych stosowanych w nagłówku bufora

Pole `b_state` nagłówka bufora zawiera zbiór bitów, z których każdy może być ustawiony lub zgaszony:

- `BH_Uptodate` - bufor zawiera aktualne dane z dysku; inicjalnie po przydzielaniu nowego bufora na blok jest zgaszony; po zakończeniu operacji odczytu danych z dysku aż do zwolnienia bufora jest ustawiony
- `BH_Dirty` - jeśli bit jest ustawiony, to zawartość bufora trzeba zapisać na dysk, ponieważ dane bloku zostały zmienione; przy kierowaniu bufora do kolejki zleceń zapisu na dysk bit jest gaszony
- `BH_Lock` - trwa właśnie operacja wejścia/wyjście na buforze więc nie można korzystać z danych bufora, lecz należy zasnąć w oczekiwaniu na zakończenie tej operacji

Pozostałe bity `BH_Req` i `BH_Protected` nie są przez algorytm używane.
Pole `b_list` nagłówka bufora oznacza numer kolejki `lru_list`, w której przechowywany jest nagłówek i może zawierać następujące wartości:

- `BUF_LOCKED` - bufor był 'ostatnio w stanie`BH_Lock`
- `BUF_DIRTY` - bufor był ostatnio' w stanie`BH_Dirty`
- `BUF_CLEAN` - wszystkie pozostałe nagłowki znajdujące się w tablicy haszującej , których nie ma na powyższych listach

Pisząc 'ostatnio' mamy na myśli stan, który istniał przy wykonywaniu funkcji `refile_buffer()`, kiedy to bufor jest wyjmowany z dotychczasowej kolejki i wstawiany do tej, która wynika z pola `b_state` nagłówka bufora. Dodatkowo w `lru_list` zyskujemy uporządkowanie po czasach ostatniego dostępu do bufora.

1.5.3 Zmienne statyczne przechowujące nagłówki buforów

Najważniejszą strukturą do przechowywania nagłówków buforów jest tablica haszująca.

```
static struct buffer_head ** hash_table = NULL;
```

W tej tablicy przechowywane są nagłówki buforów, które zostały przydzielone funkcją `getblk()`. Ponieważ każdy taki nagłówek bufora jest powiązany z dokładnie jednym blokiem urządzenia blokowego, to dla procesów odwołania do bufora są odwołaniami do danego bloku. I-ta pozycja w tablicy haszującej to początek listy dwukierunkowej, w której znajdują się nagłówki buforów, które po aplikacji funkcji haszującej do informacji o logicznym położeniu bloku dają "`i`". Oczywistą sprawą jest, że większy rozmiar tablicy `hash_table` powoduje skrócenie list przechowywanych na pozycji i-tej w tej tablicy. Jednak ustalenie rozmiaru i przydzielanie pamięci na tablicę haszującą następuje raz w funkcji `buffer_init()` przy uruchamianiu systemu. Ważne jest więc, żeby przewidzieć możliwie optymalny rozmiar tej tablicy. Rozmiar tablicy jest zależny m.in. od wielkości pamięci operacyjnej.

W funkcji `buffer_init()` inicjalizowane są też zmienne, których używa funkcja haszująca, tak by wyliczyć wartość w odpowiednim zakresie

```
static unsigned int bh_hash_mask = 0;
```

oraz uzyskać statystycznie możliwie dobry rozkład (równe długości list w tablicy haszującej).

```
static unsigned int bh_hash_shift = 0;
```

Długość list w tablicy haszującej jest istotna z punktu widzenia wydajności, ponieważ każde żądanie dostępu do bloku (pośrednio przez wywołanie funkcji systemowej np. `write()` i `read()`) powoduje przeszukanie jednej z list (o numerze zwróconym przez funkcję haszującą) w celu sprawdzenia, czy blok jest już przechowywany w pamięci, czyli czy nagłówek bufora przechowujący ten blok znajduje się na liście. Należy zaznaczyć, że ani nagłówki buforów, ani strony pamięci z danymi bloków nie są wymieniane na dysk. Te pierwsze ponieważ wymiana mogłaby wymagać nagłówków buforów do samego zapisu strony z nagłówkami buforów, a drugie nie muszą być wymieniane, ponieważ już znajdują się gdzieś na dysku.

W tym miejscu warto wspomnieć, że operacja swapowania również korzysta z buforów. Jeśli na stronie powinna być wykonana operacja wejścia/wyjścia, to pobierane są nieużywane nagłówki buforów (z listy `unused_list`) i przydzielana jest pamięć na dane bufora właśnie na tej stronie. Następnie zlecane są asynchroniczne operacje wejścia/wyjścia. Nagłówki buforów zawsze muszą istnieć dla celów wymiany, w tym celu na liście `unused_list` utrzymywany jest zbiór nagłówków, które nie są przydzielane dla procesów pobierających nagłówek w trybie synchronicznym; liczba tych nagłówków wynosi `NR_RESERVED`. Aby nie powstał zastój zarezerwowana jest odpowiednia liczba nagłówków pozwalająca wysłać 2 strony na dysk. Brak nagłówków zarezerwowanych dla celów wymiany oznacza, że właśnie jest wykonywana operacja wejścia/wyjścia na stronach i za chwilę nagłówki zostaną zwrócone do puli.

```
#define MAX_BUF_PER_PAGE (PAGE_SIZE / 512)
#define NR_RESERVED (2*MAX_BUF_PER_PAGE)
```

Ponieważ przydzielanie nowych bloków powoduje zajmowanie pamięci, to system stara się usuwać z pamięci bloki już niepotrzebne w chwili, gdy brakuje wolnej pamięci operacyjnej. W tym celu utrzymuje się listy `lru_list`, które przetrzymują nagłówki buforów w takiej kolejności, że ostatnio używane są na początku listy.

```
#define NR_LIST 3
static struct buffer_head * lru_list[NR_LIST] = {NULL, };
```

W `lru_list` oddzielnie są trzymane nagłówki buforów, co do których nastąpiło zlecenie zapisu, wymagające zapisu oraz pozostałe z `hash_table`.

Nagłówki buforów dla celów wymiany nie są przechowywane w `hash_table` ani w `lru_list` - po zakończeniu operacji wejścia/wyjścia są one zwracane na listę `reuse_list`, a stamtąd na `unused_list`.

Usunięcie bufora z kolejek w `lru_list` i `hash_table` odbywa się w czasie stałym, ponieważ mamy do dyspozycji listę dwukierunkową. Dostęp do najczęściej używanych buforów w tablicy haszującej jest szybki, ponieważ funkcja `refile_buffer`, wykonywana po każdym dostępie do bufora (przy zwalnianiu bufora funkcją `brelse()`), przenosi go na początek listy w taj tablicy.

Pamięć na nagłówki buforów przydzielana jest dynamicznie, maksymalna liczba nagłówków buforów nie używanych (na `unused_list`) wynosi `MAX_UNUSED_BUFFERS`. Jeśli jest ich więcej, to pamięć na nie jest zwalniana .

```
#define MAX_UNUSED_BUFFERS NR_RESERVED+20
```

Funkcja `getblk()` pobiera nagłówki buforów z już zaalokowaną pamięcią na dane bloku, sięgając przy tym do tablicy list wolnych buforów.

```
static struct buffer_head * free_list[NR_SIZES] = {NULL, };
```

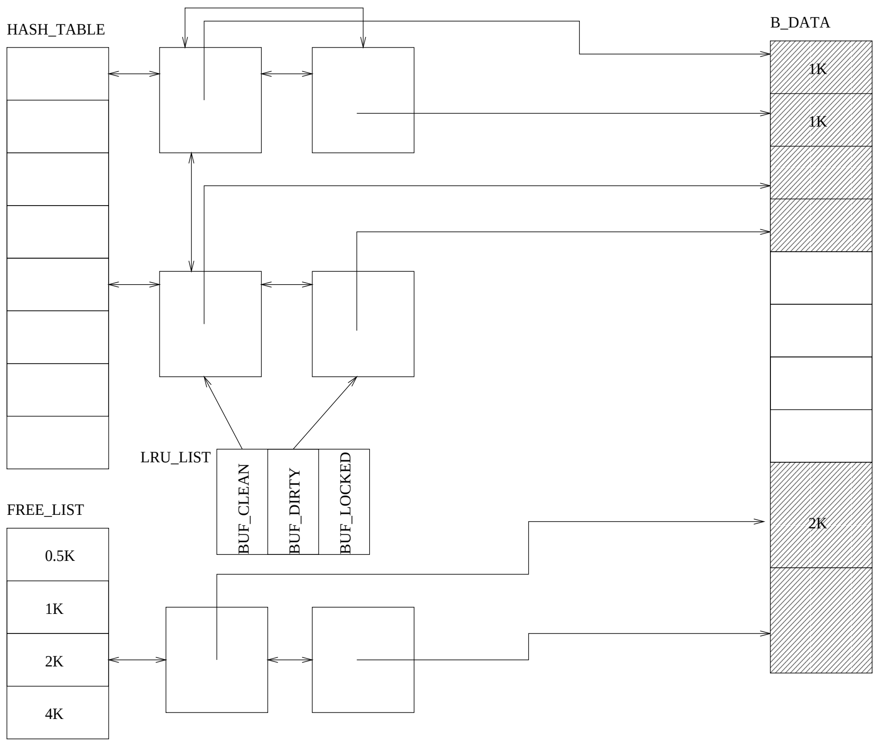
Na pozycji i-tej w tablicy wolnych buforów znajdują się nagłówki buforów z zaalokowanym miejscem na dane bufora równym 2^{9+i}. Różnych rozmiarów buforów jest `NR_SIZES`.

```
#define NR_SIZES 7
```

Dopuszczalnych jest siedem rozmiarów bufora, $2^9 <= 2^k == b\_size <= 2^{15}$, lecz standardowo rozmiar bufora jest równy rozmiarowi bloku dyskowego `BLOCK_SIZE` (1K). Dla kompatybilności z rożnymi systemami plików dopuszcza się inne rozmiary. Z `free_list` pobierany jest zawsze pierwszy nagłówek na liście.

Konwersja z rozmiaru bufora w bajtach na indeks w tablicy `free_list` następuje za pomocą poniższego makra:

```
#define BUFSIZE_INDEX(X) ((int) buffersize_index[(X)>>9])
```



Rysunek 1.2:

Każdy bufor znajduje się albo na liście w kolejce haszującej (wiązanie listy następuje przez pola `b_next` i `b_pprev`) oraz jednocześnie na odpowiedniej liście w `lru_list` (wiązanie przez pola `b_next_free` i `b_prev_free`), (wtedy nagłówek bufora ma zainicjowany numer systemu plików i numer logiczny bloku) lub na odpowiedniej liście w `free_list` (wiązanie listy następuje przez pola `b_next_free` i `b_prev_free`) (wtedy `b_dev == B_FREE`).

W przypadku potrzeby zwolnienia strony pamięci zwalniane są te strony, na których są nie zajęte bufory. W szczególności bufor nie jest zajęty, gdy `b_count == 0` i nie są potrzebne operacje dyskowe na buforze.

```
#define BUFFER_BUSY_BITS ((1<<BH_Dirty) | (1<<BH_Lock) | (1<<BH_Protected))
#define buffer_busy(bh) ((bh)->b_count || ((bh)->b_state & BUFFER_BUSY_BITS))
```

Istnieją dwie listy zawierające same nagłówki buforów (wiązanie listy następuje przez pola `b_next_free`):

```
static struct buffer_head * unused_list = NULL;
static struct buffer_head * reuse_list = NULL;
```

Gdy zabraknie nagłówka bufora dla procesu, oczekuje on na zwolnienie zasobów w kolejce `buffer_wait`.

```
static struct wait_queue * buffer_wait = NULL;
```

Liczba elementów w każdej liście jest zliczana m.in. na potrzeby demona `bd_flush` .

1.6 Zapis algorytmu getblk w abstrakcyjnym języku wysokiego poziomu

Algorytm 1 getblk

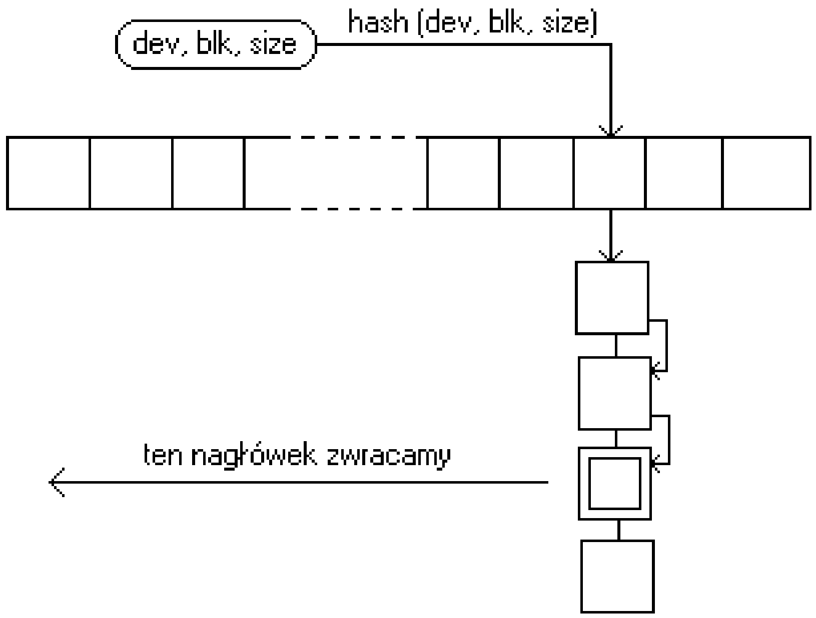
Dane wejściowe: numer urządzenia blokowego, numer bloku, rozmiar bloku
Wynik: bufor dla tego bloku

```
repeat:
buf_head:=pobierz_bufor_z_tablicy_haszującej()
jeśli buf_head jest nagłówkiem istniejącego bufora to
   zwiększ liczbę użytkownikow bufora o 1
   jeśli bufor nie jest przeznaczony do opóźnionego zapisu to
      czas_zapisu:=0
   koniec jeśli
   zwróć bufor
koniec jeśli
getfree:
buf_head:=weź_wolny_nagłówek_bufora_z_listy_wolnych_odpowiedniego_rozmiaru()
jeśli buf_head=NULL to
   skocz do refill:
koniec jeśli
usuń_nagłówek_bufora_z_listy(buf_head)
zainicjalizuj_pola_nagłówka_bufora_odpowiednimi_wartościami()
b_state:=0 (w szczególności znaczy to, że bufor nie jest UPTODATE)
wstaw_do_kolejek(buf_head)
zwróć bufor
refill:
stwórz_wolne_bufory_odpowiedniego_rozmiaru()
jeśli nie ma bufora w tablicy haszującej to
   skocz do getfree:
koniec jeśli
skocz do repeat:
```

1.7 Możliwe scenariusze dostępu do bufora

1.7.1 Jądro znajduje bufor w kolejce mieszającej

Po obliczeniu numeru kolejki mieszającej, w której może znajdować się bufor zawierający podany blok jądro przegląda listę w sposób sekwencyjny. Po znalezieniu nagłówka bufora, którego pola numeru logicznego urządzenia, numeru bloku oraz jego wielkości zgadzają się z zadanymi parametrami jest on od razu zwracany. Może to być bufor oznaczony do opóźnionego zapisu, albo nawet zablokowany, ale, tak jak zostało to zaznaczone wcześniej, sprawdzenie tego faktu nie należy już do algorytmu getblk. Zwracamy uwagę, że w tym (i oczywiście nie jedynym) miejscu Linux 2.2.17 różni się od tego, co można przeczytać w książce Bacha ([Bach]). Ten scenariusz dostępu ilustruje rysunek 1.3

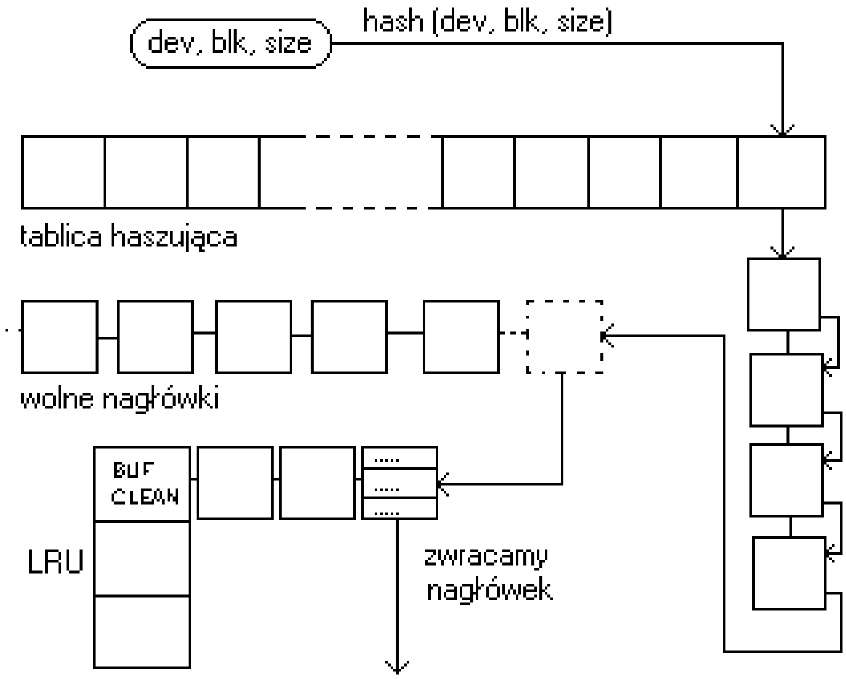


Rysunek 1.3: Bufor zostaje znaleziony w odpowiedniej kolejce haszującej. Dokładny opis w sekcji 1.7.1

1.7.2 Nie ma bufora w kolejce mieszającej, jest przynajmniej jeden bufor w puli wolnych

Jeżeli okaże się, że danego bloku nie ma w kolejce mieszającej, to należy utworzyć nowy bufor, w którym będziemy ten blok przechowywać. Istotne jest to, że w tym przypadku jesteśmy pewni, że blok nie jest odwzorowany w żaden inny bufor, ponieważ jedynym miejsce, gdzie mógłby się on znajdować jest kolejka, która właśnie została przeszukana. Kolejne kroki, które algorytm teraz wykonuje, to:

- pobranie wolnego nagłówka bufora z listy wolnych nagłówków odpowiedniego rozmiaru
- inicjalizacja pól nagłówka, czyli ustawienie numeru urządzenia i bloku, ustawienie wartości `b_count` (liczba użytkowników) na 1, `b_list` (lista LRU) na `BUF_CLEAN`, `b_flushtime` na 0



Rysunek 1.4: Danemu blokowi nie odpowiada żaden bufor w systemie, więc zostaje utworzony i zainicjalizowany nowy bufor. Dokładny opis w sekcji 1.7.2

(czyli bufor nie jest przeznaczony do zapisu), `b_dev_id` na `NULL` i przypisanie pod `b_end_io` wskaźnika na procedurę `end_buffer_io_sync()`, która jest wywoływana, gdy nadejdzie sygnał zakończenia transmisji danych z dysku do bufora (funkcja ta między innymi odblokowuje bufor)

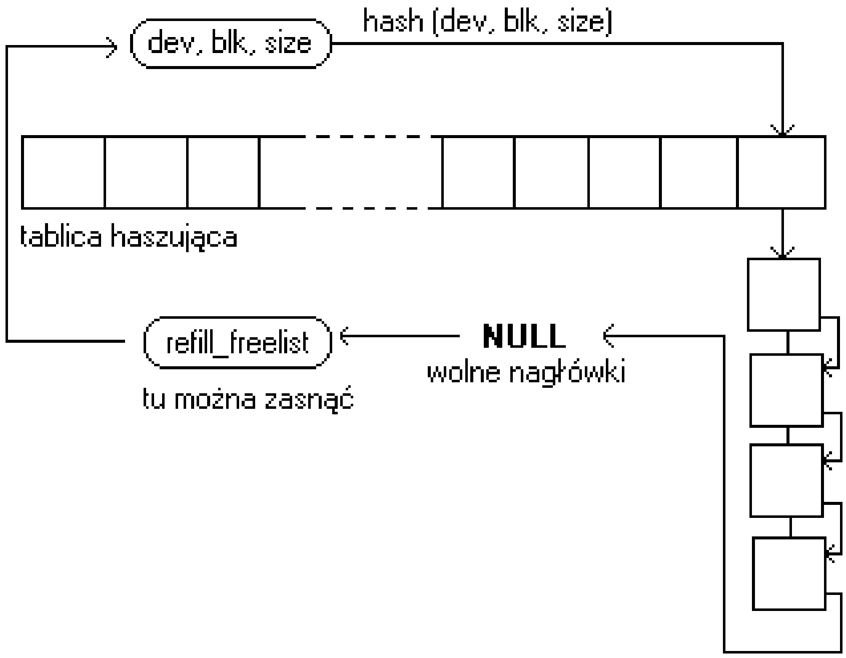
- ustawienie pola `b_state` na 0, co, między innymi, oznacza, że dane w buforze nie odpowiadają zawartości bloku na dysku
- wywołanie procedury `insert_into_queues(bh)`, która wstawia nagłówek bufora na koniec odpowiedniej listy LRU i zwiększa o 1 licznik dotyczący tej listy, po czym umieszcza go w kolejce haszującej o wyliczonym numerze uaktualniając liczbę wszystkich buforów w tablicy haszującej
- zwrócenie nagłówka bufora

Ten scenariusz dostępu ilustruje rysunek 1.4

1.7.3 Nie ma bufora w kolejce mieszającej i nie ma żadnych wolnych buforów

Może się też tak zdarzyć, że algorytm nie znajdzie bufora w tablicy mieszającej, a gdy zechce utworzyć nowy bufor natknie się na brak wolnych nagłówków. W takiej sytuacji wywołuje funkcję `refill_freelist()`[1], której zadaniem jest dostarczenie nowych buforów dla potrzeb procesów działających w systemie. Ponieważ w trakcie wywołania tej funkcji możliwe było zaśnięcie na pewien

[1]Funkcja ta nie będzie tu dokładnie opisana, bo jest to tematem innej pracy w obrębie projektu 'Algorytmy + Struktury Danych = Linux'. W punkcie 1.9 można znaleźć opis jej argumentów.



Rysunek 1.5: Oczywiście nie chodzi tu o nieskończoną pętlę. Po wywołaniu funkcji `refill_freelist()` należy jeszcze raz sprawdzić tablicę mieszającą. Jeżeli bufor tam będzie, to mamy sytuację jak na Rys. 1.3. W przeciwnym przypadku patrzymy, czy są wolne nagłówki. Jeżeli tak, to jest jak na Rys. 1.4, jeżeli nie, to od nowa wywołujemy `refill_freelist()`. Dokładny opis w sekcji 1.7.3

czas, to po jej zakończeniu jeszcze raz przeprowadzane jest wyszukiwanie bufora w kolejkach haszujących, bo być może w tym czasie jakiś inny proces zdobył wolny bufor, który został przeznaczony na ten sam blok. W tym przypadku getblk nie może pobrać nowego, wolnego bufora, bo musi być zachowany niezmiennik, że jednemu blokowi odpowiada co najwyżej jeden bufor w systemie. W przypadku, gdy `refill_freelist()` nie zdobędzie żadnych nowych buforów getblk wywołuje ją po raz kolejny. Ten scenariusz dostępu ilustruje rysunek 1.5

1.8 Kod źródłowy funkcji getblk (dotyczy jądra 2.2.17)

```
struct buffer_head * getblk(kdev_t dev, int block, int size)
{
struct buffer_head * bh;
int isize;

repeat:
bh = get_hash_table(dev, block, size);
if (bh) {
if (!buffer_dirty(bh)) {
bh->b_flushtime = 0;
}
return bh;
```

```
}

isize = BUFSIZE_INDEX(size);
get_free:
bh = free_list[isize];
if (!bh)
goto refill;
remove_from_free_list(bh);

/* OK, FINALLY we know that this buffer is the only one of its kind,
 * and that it's unused (b_count=0), unlocked, and clean.
 */
init_buffer(bh, dev, block, end_buffer_io_sync, NULL);
bh->b_state=0;
insert_into_queues(bh);
return bh;

/*
 * If we block while refilling the free list, somebody may
 * create the buffer first ... search the hashes again.
 */
refill:
refill_freelist(size);
if (!find_buffer(dev,block,size))
goto get_free;
goto repeat;
}
```

1.9 Interfejs funkcji wywoływanych przez getblk

- `struct buffer_head * get_hash_table(kdev_t dev, int block, int size)`

  Funkcja, jeśli znajdzie bufor w `hash_table`, to zwiększa liczbą korzystających `b_count` i zwraca bufor; jeśli nie ma poszukiwanego bufora zwraca `NULL`

- `struct buffer_head * find_buffer(kdev_t dev, int block, int size)`

  Funkcja przeszukuje listę z odpowiedniej pozycji w tablicy haszującej w poszukiwaniu nagłówka bufora; jeśli nie ma poszukiwanego bufora zwraca `NULL`

- `void init_buffer(struct buffer_head *bh, kdev_t dev, int block, bh_end_io_t *handler, void *dev_id)`

  Funkcja inicjuje nowy nagłówek bufora na standardowe wartości

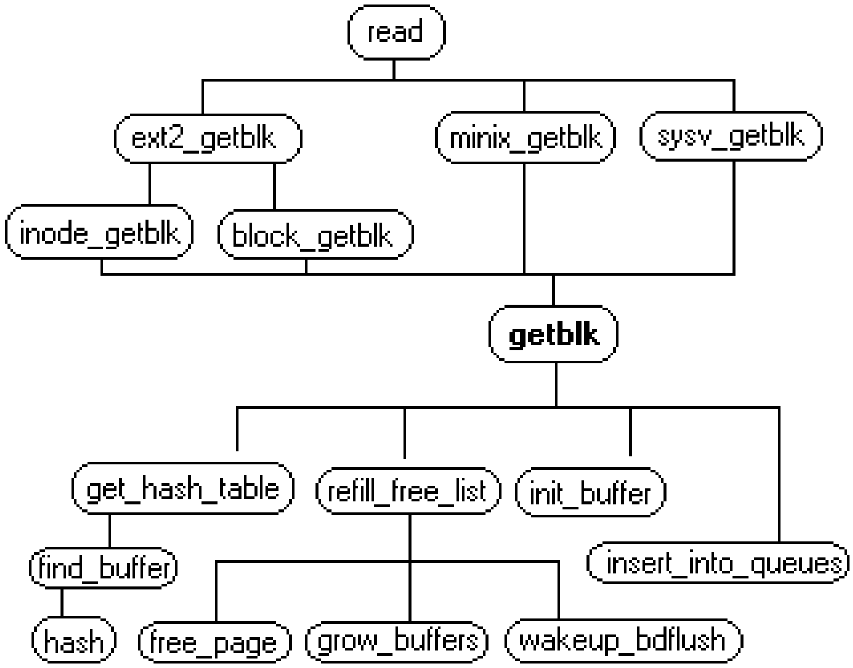
- `static void insert_into_queues(struct buffer_head * bh)`

  Funkcja wkłada nagłówek bufora na początek kolejek w `hash_table` (do kolejki o numerze zwracanym przez funkcję haszującą) oraz `lru_list` (do kolejki o numerze `b_list`)

- `static void refill_freelist(int size)`

  Funkcja zapełnia listę `free_list` wolnymi buforami; w przypadku braku zasobów budzi `bd_flush`

- `#define __get_free_page(gfp_mask) __get_free_pages((gfp_mask),0)`



Rysunek 1.6: Funkcja umieszczona na rysunku wyżej wywołuje funkcje znajdujące się pod nią

Funkcja przydziela stronę pamięci dla buforów (strona pamięci z flagą `GFP_BUFFER`) i zwraca jej adres

- `static int grow_buffers(int size)`

  Funkcja zwiększa ilość wolnych buforów rozmiaru size, o ile da się przydzielić stronę na dane bloków; rozmiary bufora nie mogą przekroczyć rozmiaru strony.

- `static struct buffer_head * create_buffers(unsigned long page, unsigned long size, int async)`

  Funkcja przydziela pamięć na bloki dyskowe rozmiaru `size` na podanej stronie pamięci, korzystając z nieużywanych nagłówków buforów. Nagłówki są łączone wskaźnikiem `b_this_page`, inicjowane są też inne pola nagłówka wskazujące na to, że jest on nie używany i bufor jest czysty

- `int brw_page(int rw, struct page *page, kdev_t dev, int b[], int size, int bmap)`

  Funkcja do wymiany stron pamięci na dysk, używa ona nagłówków buforów jednak nie wstawia ich do żadnych struktur, lecz zleca asynchroniczny odczyt/zapis . Funkcja ta jest wywoływana np. przy błędzie braku strony. Bufory mogą nie zostać przydzielone dla zwykłych procesów z powodu braku nagłówków. Dla celów stronicowania brak wolnych nagłówków oznacza, że trwają inne operacje na nagłówkach z puli zarezerwowanych, i należy na nie zaczekać .

1.10 Implementacja funkcji haszującej

Funkcja mieszająca jest makrodefinicją następującej treści:

```
#define _hashfn(dev,block) \
((((dev)<<(bh_hash_shift - 6)) ^ ((dev)<<(bh_hash_shift - 9))) ^ \
 (((block)<<(bh_hash_shift - 6)) ^ ((block) >> 13) ^ ((block) << (bh_hash_shift - 12))))
#define hash(dev,block) hash_table[_hashfn(dev,block) & bh_hash_mask]
```

Wartości zmiennych `bh_hash_shift` i `bh_hash_mask` wyznaczane są w funkcji `buffer_init (unsigned long memory_size)`, której zadaniem jest zaalokowanie pamięci na kolejki haszujące i zainicjalizowanie listy wolnych buforów. Funkcja ta jest wywoływana przez funkcję `start_kernel()` (init/main.c) przy uruchamianiu systemu. Aktualna wartość parametru `memory_size` przekazana `buffer_init()` jest ustalana jako (`memory_end - memory_start`), gdzie dwie ostatnie wielkości są obliczane przez funkcję `setup_arch()`, również wywoływaną przez procedurę `start_kernel()` jescze przed `buffer_init()`. Oto fragment kodu funcji `buffer_init()`, gdzie obliczane są wartości `bh_hash_shift` i `bh_hash_mask`:

```
...

int order;
unsigned int nr_hash;

/* we need to guess at the right sort of size for a buffer cache.
   the heuristic from working with large databases and getting
   fsync times (ext2) manageable, is the following */

memory_size >>= 20;
for (order = 0; (1UL << order) < memory_size; order++);

/* try to allocate something until we get it or we're asking
           for something that is really too small */

do {
unsigned long tmp;

nr_hash = (1UL << order) * PAGE_SIZE /
    sizeof(struct buffer_head *);
bh_hash_mask = (nr_hash - 1);

tmp = nr_hash;
bh_hash_shift = 0;
while((tmp >>= 1UL) != 0UL)
bh_hash_shift++;

hash_table = (struct buffer_head **)
    __get_free_pages(GFP_ATOMIC, order);
} while (hash_table == NULL && --order >= 0);

...
```

Jak widać niejawne parametry funkcji haszującej (`bh_hash_shift` i `bh_hash_mask`) zależą od rozmiaru nagłówka bufora, a także od rozmiaru pamięci operacyjnej maszyny, na której pracuje Linux. Postać funkcji mieszającej oraz sposób inicjalizacji parametrów `bh_hash_shift` i `bh_hash_mask` zostały opracowane przez Dave'a M. Autor twierdzi, że heurystyka ta oparta jest na doświadczeniach zdobytych przy pracy z dużymi bazami danych i systemem plików ext2.

1.11 Getblk - ocena efektywności i bezpieczeństwa

Porównajmy szybkość reakcji systemu na żądanie sprowadzenia do pamięci bloku dyskowego w przypadku, gdy nie ma mechanizmu buforowania oraz gdy system plików dysponuje pewną pulą buforów, gdzie algorytmem wyszukującym jest getblk. Załóżmy, że w pamięci przechowujemy 2048 buforów w tablicy haszującej długości 512. W pierwszym przypadku proces, który zgłosił zapotrzebowanie na blok będzie wstrzymany na czas rzędu 10 milisekund (przy dzisiejszych parametrach twardych dysków), albo jeszcze dłużej, jeżeli urządzenie blokowe, z którego jądro wczytuje blok charakteryzuje się gorszymi parametrami technicznymi. Przy drugim scenariuszu dostępu należy obliczyć wartość funkcji haszującej (czas praktycznie zaniedbywalny) oraz przejrzeć listę przywiązaną pod wyliczonym indeksem tablicy, co w oczekiwanym przypadku powinno zająć 4 * czas dostępu do komórki pamięci, czyli około 40 nanosekund (dotyczy aktualnych możliwości przeciętnych układów pamięci SDRAM). Widać stąd, że sprawnie zorganizowany mechanizm buforowania wydatnie podnosi szybkość działania systemu.

Jeżeli chodzi o wprowadzanie modyfikacji mających polepszyć wydajność algorytmu getblk, to po analizie kodu źródłowego i porównaniu go z opisami zamieszczonymi w pracach studentów naszego wydziału z kilku poprzednich lat można zauważyć tendencję do zmniejszania liczby kolejek LRU (teraz są tylko trzy) oraz stosowania bardziej wyszukanych algorytmów dla funkcji mieszającej. Aby uzyskać jak największą szybkość działania funkcji operujących na listach nagłówków buforów, struktura `buf_head` została zaprojektowana w przemyślany sposób. Otóż zbiór pól wykorzystywanych przez funkcje wyszukujące w kolejkach haszujących (np. `find_buffer()`) zajmuje pierwszych 16 bajtów, a pola wykorzystywane przy przechodzeniu list LRU (`sync_buffers()`, `refill_freelist()`) kolejne 16 bajtów. Uzasadnieniem takiej organizacji nagłówka bufora jest fakt, że długość linii pamięci cache w procesorach 32-bitowych ma właśnie 16 bajtów.

Co do bezpieczeństwa, to należy zwrócić uwagę na fakt, że teoretycznie może zaistnieć sytuacja, kiedy system jest _bardzo obciążony i po wywołaniu funkcji `refill_free_list()` nie dostaniemy żadnych nowych wolnych buforów. W takiej sytuacji reakcją algorytmu getblk jest ponowne wywołanie funkcji `refill_free_list()`, co w szczególnie 'złośliwym' przypadku znowu zakończy się fiaskiem. Ze względu na konstrukcję algorytmu mamy tu prostą drogę do nieskończonego ciągu wywołań. Prawdopodobieństwo wystąpienia takiego scenariusza zdarzeń w praktyce jest jednak zaniedbywalnie małe, ze względu na istnienie odpowiednich algorytmów dynamicznego zarządzania pamięcią, których zadaniem jest ochrona systemu przed takimi problemami (zainteresowanego czytelnika odsyłamy do prac traktujących o zarządzaniu pamięcią).

1.12 Podsumowanie

Co należy zapamiętać o algorytmie getblk:

- jest jednym z podstawowych algorytmów wykorzystywanych w Linuksie do obsługi pamięci buforowej
- odpowiada za wyszukanie i zwrócenie do wołającej funkcji bufora zawierającego dany blok (jeżeli taki jest), lub zwrócenie pustego bufora, do którego blok będzie wczytany (jeżeli nie znajdze bufora w kolejce haszującej)
- zapewnia utrzymanie niezmiennika, który mówi, że dany blok znajduje się w co najwyżej jednym buforze
- procedury czytania i pisania pliku zwykle korzystają z jego usług (tak jest w ext2, minix, fat, hfs (Macintosh))
- ze względu na poprzedni punkt efektywność implementacji funkcji getblk() wpływa na całościową wydajność systemu

- w kolejnych edycjach jądra Linuxa algorytm oraz wykorzystywane przez niego struktury danych podlegają dosyć częstym modyfikacjom (w 2.2.17 jest już inaczej niż np. w 2.2.12)
- jest potencjalnie nieskończony (choć w praktyce się to nie zdarza)

1.13 Pliki źródłowe

include/linux/fs.h
fs/buffer.c

Literatura

[Bach] Bach M. : *Budowa systemu operacyjnego Unix.* Warszawa, WNT 1995.

[Lab Linux] Studenci Wydziału Matematyki, Informatyki i Mechaniki UW: *Projekt Lab Linux* Warszawa, 1997/98

[Linux Prezentacje] Studenci Wydziału Matematyki, Informatyki i Mechaniki UW: *Projekt Linux Prezentacje* Warszawa, 1998/99