Obsługa podręcznej pamięci buforowej stron

Tomasz Szymko

17 grudnia 2001

Spis treści

1 Wstęp **3**

2 Wprowadzenie **3**

3 Realizacja podręcznej pamięci buforowej stron **3**
- 3.1 Struktura podręcznej pamięci buforowej stron . . . 3
- 3.2 Opis podstawowych operacji na buforach stron . . . 4
  - 3.2.1 Opis funkcji inicjalizującej system podręcznej pamięci buforowej stron . 4
  - 3.2.2 Opis funkcji wyszukujących strony w buforze stron . . . 6
  - 3.2.3 Opis dodawania stron do bufora stron . . . 8
  - 3.2.4 Opis usuwania stron z bufora stron . . . 9

4 Opis struktur danych **9**
- 4.1 Struktury i makra używane do obsługi tablicy mieszającej . . . 9
- 4.2 Struktura nagłówka ramki - opis skrócony . . . 9
- 4.3 Pozostałe struktury danych . . . 11

A Zadania **12**
- A.1 Zadanie 1. . . . 12

B Rozwiązania zadań **13**
- B.1 Rozwiązanie zadania 1. . . . 13

C Struktura address_space - opis skrócony **13**

1 Wstęp

Niniejsza praca omawia sposób realizacji obsługi podręcznej pamięci buforowej stron przez jądro Linuxa w wersji[1] 2.4.7. Szczególny nacisk położony jest na implementację tegoż jądra przeznaczoną na procesory architektury i386 i jeśli nie zaznaczono inaczej to wszystkie szczegóły techniczne dotyczą właśnie i386. W szczególności niektóre tu podane fakty mogą być nieprawdziwe dla innych architektur sprzętowych.

2 Wprowadzenie

Jak powszechnie wiadomo, jedną z bardziej krytycznych (czasochłonnych) operacji w systemie komputerowym jest czytanie danych ze stosunkowo wolnych pamięci masowych[2]. Dlatego system operacyjny, w celu zwiększenia ogólnej wydajności systemu, stara się optymalizować dostęp do pamięci masowych. Aby usprawnić czytanie (pisanie) z dysków, jądro systemu utrzymuje pulę buforów, w których trzyma ostatnio używane pliki. Dzięki temu, kolejne operacje na danym pliku są wykonywane na jego obrazie umieszczonym w pamięci[3]. Zabieg ten daje nam dwie zasadnicze korzyści:
– przyśpiesza pracę systemu, bo kolejne odwołania do pliku będą przekierowywane do jego obrazu w pamięci operacyjnej.
– zmniejsza zużycie pamięci, bo przy kolejnych próbach wczytywania pliku, proces będzie przekierowywany do już wczytanego obrazu pliku.

3 Realizacja podręcznej pamięci buforowej stron

3.1 Struktura podręcznej pamięci buforowej stron

W Linuxie bufor stron jest strukturą inicjalizowaną i utrzymywaną przez jądro systemu. Bufor ten jest zbiorem ramek trzymanych w listach, do których dowiązania pamiętane są w tablicy mieszającej (ang. hash table) `page_hash_table`. W systemie, bufor stron jest początkowo pusty, a potem rozrasta się w miarę potrzeb, dopóki nie zabraknie pamięci. Gdy skończy się pamięć, wówczas uruchamiany jest mechanizm czyszczenia pamięci, który prawie zawsze zaczyna zwalnianie pamięci od redukcji podręcznej pamięci buforowej stron. Reasumując, w systemie jest tylko jeden bufor stron i nie ma limitu górnego na ilość buforowanych stron.

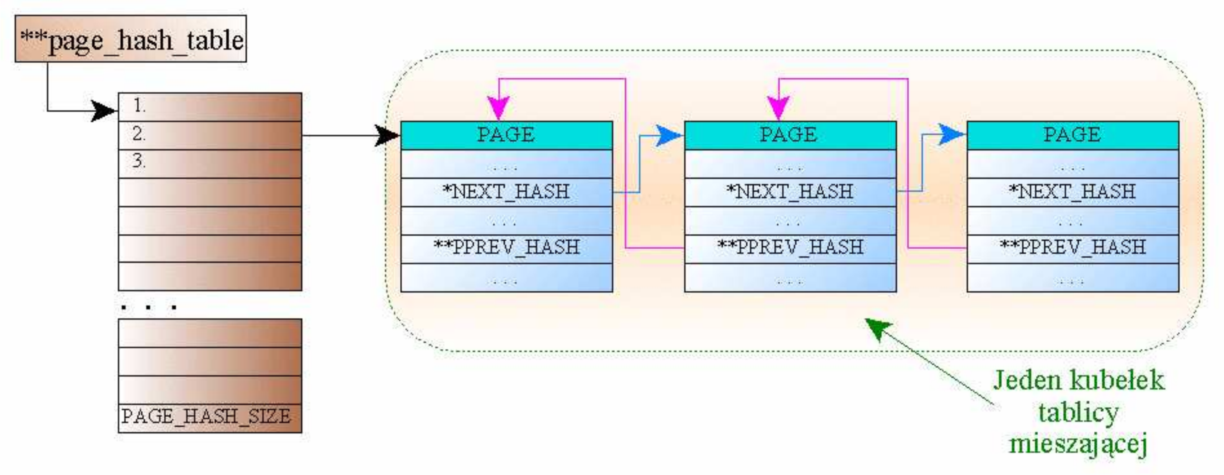
Aby szybko stwierdzić czy dana strona[4], jest już buforowana, bufor stron został zaimplementowany jako tablica mieszająca z rozwiązywaniem kolizji metodą łańcuchową. Tablica mieszająca ma rozmiar zależny od rozmiaru pamięci operacyjnej komputera, a pamiętane są w niej

[1] W innych wersjach omawiane zagadnienia mogą być zrealizowane podobnie, albo zupełnie inaczej!

[2] Masowe pamięci w pełni elektroniczne lub optyczne… to będzie coś!

[3] Jądro systemu dba przy tym, aby dane trzymane w plikach i pamięci były spójne.

[4] Przypominam, że ramka to najmniejszy obszar pamięci fizycznej, a strona to najmniejszy obszar pamięci wirtualnej. Rozmiar ramki jest zazwyczaj równy rozmiarowi strony.



Rysunek 1: Struktura podręcznej pamięci stron.

wskaźniki na listy ramek wchodzących w skład bufora. Dalej taką pojedyńczą listę ramek będę nazywał „kubełkiem” tablicy mieszjącej.

3.2 Opis podstawowych operacji na buforach stron

Teraz przejdę do omówienia podstawowych funkcji operujących na buforze stron. Zacznę od opisu funkcji inicjalizującej bufor. Potem opiszę wyszukiwanie stron, a na koniec będzie parę słów o dodawaniu i usuwaniu stron z bufora.

3.2.1 Opis funkcji inicjalizującej system podręcznej pamięci buforowej stron

Do inicjalizacji buforów stron służy funkcja page_cache_init. Oto ona wraz z moimi komentarzami:

```
void __init page_cache_init(unsigned long mempages)
/**
 * Bufor stron jest inicjalizowany przy starcie systemu.
 * Tak więc ta funkcja jest wołana tylko raz w funkcji:
 * start_kernel() z parametrem num_physpages, przy czym
 * num_physpages = ilość pamięci w systemie wyrażona w ramkach.
 */
{
  unsigned long htable_size, order;

  htable_size = mempages;
  htable_size *= sizeof(struct page *);
```

```
/* na i386 jest to równoważne: htable_size *= 16; */

for(order = 0; (PAGE_SIZE << order) < htable_size; order++)
  ;
/**
 * pętla ustawia order tak żeby htable_size mieścił się w 2^order
 * stron. Order przyjmie następujące wartości w zależności od
 * rozmiaru pamięci:
 *       pamięć       | order | rozmiar tablicy page_hash_table
 *   4MB --   8MB-1B  |  3    |        2048    kubełków
 *   8MB --  16MB-1B  |  4    |        4096    -- // --
 *  16MB --  32MB-1B  |  5    |        8192
 *  32MB --  64MB-1B  |  6    |       16384
 *  64MB -- 128MB-1B  |  7    |       32768
 * 128MB -- 256MB-1B  |  8    |       65536
 * 256MB -- 512MB-1B  |  9    |      131072
 * 512MB --   4GB     |  9 bo order jest poźniej (przy przydzie-
 *                    | laniu pamięci) przycinany tak, żeby nie
 *                    | przekraczał stałej MAX_ORDER równej 10
 *
 *    Uzależnienie wielkości tablicy page_hash_table od pamięci
 * ma na celu, z jednej strony oszczędność pamięci, a z drugiej
 * poprawienie właściwości funkcji mieszającej.
 */
do {
  unsigned long tmp =(PAGE_SIZE << order) / sizeof(struct page *);
  /**
   * tmp jest liczbą wskaźników jakie się pomieszczą
   * w (PAGE_SIZE << order) bajtach pamięci.
   *
   * Poniższe operacje wyznaczają logarytm dwójkowy z tmp.
   * Jak się potem okaże, tak uzyskana wartość page_hash_bits
   * będzie podstawą do wyznaczenia liczby pozycji przechowy-
   * wanych w tablicy page_hash_table. Patrz do punktu o
   * strukturach danych.
   */
  page_hash_bits = 0;
  while((tmp >>= 1UL) != 0UL)
    page_hash_bits++;

  page_hash_table = (struct page **)
                  __get_free_pages(GFP_ATOMIC, order);
```

```
    /**
     * __get_free_pages zwraca NULL jeśli order >= MAX_ORDER,
     * lub gdy nie udało się przydzielić pamięci, wpp. zwraca
     * wskaźnik do pierwszej z 2^order ramek pamięci
     * przydzielonych na tablicę page_hash_table.
     * Przydzielone ramki są powiązane w listę dwukierunkową.
     */

  } while(page_hash_table == NULL && --order > 0);

  /**
   * w printk jest mały BŁĄD, bo przez wzgląd na warunek
   * ostatniej pętli powinno być (order + 1)
   */
  printk(
    "Page-cache hash table entries: %d (order: %ld, %ld bytes)\n"
    ,(1 << page_hash_bits), order, (PAGE_SIZE << order));
  /**
   * Tak więc w tablicy page_hash_table będziemy mieć:
   * (1 << page_hash_bits) pozycji!
   */

  if (!page_hash_table)
    panic("Failed to allocate page hash table\n");
  memset((void *)page_hash_table, 0,
          PAGE_HASH_SIZE * sizeof(struct page *));
}
```

3.2.2 Opis funkcji wyszukujących strony w buforze stron

Do szybkiego rozsyłania oraz wyszukiwania danych w kubełkach tablicy mieszającej służy specjalna funkcja mieszająca: page_hash(). Funkcja ta jest zaimplementowana w postaci makra, które wraz z „wywoływaną” w nim funkcją, wyznaczającą przesunięcie w tablicy mieszającej, przedstawiają się tak:

```
inline unsigned long
_page_hashfn(struct address_space * mapping, unsigned long index)
{
#define i (((unsigned long) mapping)/  \
           (sizeof(struct inode) & ~ (sizeof(struct inode) - 1)))
#define s(x) ((x)+((x)>>PAGE_HASH_BITS)) \
        return s(i+index) & (PAGE_HASH_SIZE-1);

#undef i
#undef s
}

#define page_hash(mapping,index) \
                  (page_hash_table+_page_hashfn(mapping,index))
```

Zagmatwane, czyż nie?! Jednak, jeśli przyjąć, że wartością wyrażenia:
(sizeof(struct inode) & ˜ (sizeof(struct inode) - 1))
jest pewna stała (nazwijmy ją: stała), to treść powyższych makr, możnaby zapisać w natępującym pseudokodzie:

```
page_hash(mapping, index) {
    long wsk; /* wskaźnik na kubełek odpowiadający parze: (mapping, index) */
    wsk = (mapping/stała + index);
    wsk = wsk + (wsk div (ilość elementów w tablicy mieszającej));
    wsk = wsk mod (ilość elementów w tablicy mieszającej);
    wsk = page_hash_table + wsk;
    return wsk;
}
```

Jak widać funkcja mieszająca operuje na trzech zmiennych, których znaczenie jest następujące:

mapping - jest wskaźnikiem na strukturę opisującą przestrzeń adresową pliku[5]. Ponieważ z każdym plikiem jest związana jedna taka struktura, więc dla danego pliku wskaźnik mapping będzie go jednoznacznie identyfikował.

index - jest przesunięciem (offsetem) w pliku pokazywanym przez mapping.

page_hash_table - tu używane jako adres początku tablicy mieszającej.

[5]Nie jest to do końca prawdą, ale dla potrzeb tego paragrafu, takie uogólnienie wystarczy. Po szczegóły odsyłam do prezentacji poświęconej systemom plików.

Teraz gdy już wiemy jak wygląda funkcja mieszająca, to możemy zająć się opisem wyszukiwania stron w buforze. Przebiega ono zgodnie z poniższym algorytmem:

1. Wyznacz wartość funkcji mieszającej, czyli wskaźnik na odpowiedni kubełek.

2. Mając mapping i index, sprawdź czy w wyznaczonym kubełku jest zamapowana szukana strona. Strona jest zamapowana, gdy na liście kubełka jest ramka, w nagłówku której odpowiednie pola są równe:
   (page->mapping == mapping)
   (page->index == index)

3. Gdy znalazłeś szukaną ramkę, to zwróć do niej wskaźnik, w przeciwnym przypadku zwróć NULL.

Funkcja mieszająca jest zdefiniowana w pliku pagemap.h. Funkcje realizujące wyszukiwanie strony w buforze stron są zdefiniowane w pliku filemap.c. W szczególności warto zwrócić uwagę na:
__find_page_nolock - szuka strony w kubełku
__find_page_simple - uproszczone szukanie strony w kubełku (bez ustawiania bitu odniesienia do strony)

3.2.3 Opis dodawania stron do bufora stron

Ponieważ w funkcjach wstawiających stronę do bufora stron nie ma specjalnych rewelacji, to ograniczę się do opisu algorytmu dodawania nowej strony. Ciekawskich odsyłam do pliku filemap.c. W szczególności warto zajrzeć do funkcji:
page_cache_read - wczytanie strony do bufora
add_page_to_hash_queue - dodanie strony do kubełka
add_page_to_inode_queue - dodanie strony do przestrzeni adresowej pliku
Wszystkie inne funkcje dodające stronę do bufora, służą do obsłużenia różnych przypadków ustawienia flag i ochrony, ale zasadniczo do wykonania właściwej roboty korzystają z trzech powyższych funkcji.

Ogólnie rzecz biorąc, strona jest dodawana do bufora stron zawsze, gdy funkcja obsługująca wyjątek braku strony (ang. page fault), nie znajdzie tej strony w buforze stron. Bez zagłębiania się w szczegóły techniczne, można powiedzieć, że dodawanie strony do bufora wygląda tak:
— tworzona jest nowa ramka, do której wczytywane są dane z pliku.
— ramka ta dołączana jest do odpowiedniego kubełka tablicy mieszającej.
— ponadto wstawia się tę ramkę do listy „czystych” stron związanej z przestrzenią adresową danego pliku.

3.2.4 Opis usuwania stron z bufora stron

Strony są usuwane z bufora stron dopiero, gdy w systemie zacznie brakować pamięci. Wywoływane są wówczas dwie funkcje:
remove_page_from_hash_queue - usuwa stronę z kubełka
remove_page_from_inode_queue - usuwa stronę z przestrzeni adresowej właściciela
a następnie zwalniana jest ramka pamięci zajmowana przez tę stronę. I to już wszystko.

4 Opis struktur danych

4.1 Struktury i makra używane do obsługi tablicy mieszającej

atomic_t page_cache_size
liczba stron jaka aktualnie jest w buforze stron

struct page **page_hash_table
wskaźnik na tablicę mieszjącą

unsigned int page_hash_bits
logarytm z liczby pozycji w tablicy mieszającej

#define PAGE_HASH_BITS (page_hash_bits)
jak wyżej

#define PAGE_HASH_SIZE (1 « PAGE_HASH_BITS)
ilość pozycji w tablicy mieszającej

4.2 Struktura nagłówka ramki - opis skrócony

Bardziej szczegółowy opis powinien być w rozdziale poświęconym obsłudze tablicy ramek. Każda ramka w systemie Linux jest skojarzona ze strukturą page. Struktura przechowuje informacje o ramce pamięci fizycznej. Jest zdefiniowana w pliku mm.h.

```
typedef struct page {
    struct list_head        list;
    struct address_space    *mapping;
    unsigned long           index;
    struct page             *next_hash;
    atomic_t                count;
    unsigned long           flags;
    struct list_head        lru;
    unsigned long           age;
    wait_queue_head_t       wait;
    struct page             **pprev_hash;
```

```
    struct buffer_head      *buffers;
    void                    *virtual;
    struct zone_struct      *zone;
} mem_map_t;
```

Znaczenie pól:

list	1. jeżeli ramka jest wolna, to przez to pole jest dołączona do dwukierunkowej listy wolnych ramek 2. jeżeli jest to ramka z mapowania i-węzła, to przez to pole jest dołączona do dwukierunkowej listy wczytanych stron tego i-węzła
*mapping	wskaźnik (zwrotny) na strukturę opisującą przestrzeń adresową (np. przestrzeń inode'a) do której jest przypisana ta ramka
index	offset w mapowanym pliku, od którego zaczyna się obszar mapowany w tej ramce. Offset ten nie musi być wielokrotnością rozmiaru ramki (PAGE_SIZE)
*next_hash	wskaźnik na następną ramkę w kubełku tablicy mieszającej
count	zawiera liczbę odniesień do ramki (0 ramka nie była używana)
flags	flagi, szczegóły w rozdziale poświęconym tablicy ramek
lru	lista do realizacji algorytmu LRU na ramkach
age	licznik wieku ramki
wait	kolejka procesów czekających na odblokowanie ramki (np. na zakończenie operacji I/O na ramce)
**pprev_hash	wskaźnik na poprzednią ramkę w kubełku tablicy mieszającej
*buffers	jeżeli ramka jest wykorzystywana na bufory dyskowe, to jest to wskaźnik na listę nagłówków buforów umieszczonych na tej ramce, w przeciwnym przypadku jest tu NULL
*virtual	adres wirtualny
*zone	strefa pamięci, w której znajduje się ramka. Pamięć fizyczna PC jest dzielona na 3 strefy. W każdej strefie pamięć jest zarządzana osobno. Szczegóły powinny być w rozdziale o organizacji pamięci fizycznej i w pliku mmzone.h.

4.3 Pozostałe struktury danych

Opis vm_area_struct jest w rozdziale poświęconym obsłudze pamięci wirtualnej.

Opis address_space powinien się znaleźć w prezentacji poświęconej systemom plików. Skrócony opis jest w dodatku C.

A Zadania

A.1 Zadanie 1.

Załóżmy, że proces P1 mapuje plik o rozmiarze 100 stron, przy pomocy pojedynczago obszaru VMA. Początkowo w buforze stron, nie ma stron związanych z tym VMA.

Przy pobieraniu strony do bufora jest rozpoczynane wczytywanie stron z wyprzedzeniem o ile VMA do którego ona należy ma ustawioną flagę VM_SEQ_READ (a nasz ma). Załóż, że dla naszego pliku oprócz żądanej strony zawsze jest wczytywanych 10 następnych stron.

Proces odwołuje się do następujących offsetów w pliku:
2001, 51842, 4999, 101010, 35353.
Podaj ile stron pliku zostanie zbuforowane i ile razy. Określ zakresy offsetów pliku wczytane do bufora stron. Ile minimalnie, w omawianym przypadku, potrzeba odwołań do pliku, aby wszystkie jego strony zostały zbuforowane?

B Rozwiązania zadań

B.1 Rozwiązanie zadania 1.

Strony do bufora stron są wczytywane tylko raz, w buforze nie ma duplikatów! Zbuforowane zostaną 34 strony z zakresów:
0 - 94207, 98304 - 143359.

Potrzeba minimum 10 odwołań najlepiej do stron:
0, 11, 22, 33, 44, 55, 66, 77, 88, 99.

C Struktura address_space - opis skrócony

Struktura address_space z grubsza rzecz biorąc opisuje przestrzeń adresową przydzieloną inodowi w pamięci, to jest zbiór jego VMA i ramek.

```
struct address_space{
    struct list_head    clean_pages; /*lista czystych ramek należących do przestrzeni*/
    struct list_head    dirty_pages; /*lista brudnych ramek należących do przestrzeni*/
    struct list_head    locked_pages; /*lista zablokowanych ramek należących do przestrzeni*/
    unsigned long    nrpages; /*liczba ramek przestrzeni*/
    struct address_space_operations    *a_ops; /*funkcje do operowania na przestrzeni*/
    struct inode    *host; /*wskaźnik na inode będący właścicielem tej przestrzeni*/
    struct vm_area_struct    *i_mmap; /*wskaźnik na prywatny obszar vma*/
    struct vm_area_struct    *i_mmap_shared; /*wskaźnik na dzielony obszar vma*/
    spinlock_t    i_shared_lock; /*blokada wirująca*/
    int    gfp_mask; /*flagi*/
};
```

Bibliografia

[1] A. Silberschatz, P. B. Galvin: *Podstawy systemów operacyjnych*, WNT 2000

[2] Kody źródłowe jądra Linuxa w wersji 2.4.7.

[3] Projekty studentów MIMUW z lat poprzednich