Podręczna pamięć buforowa stron (jądro 2.2.17)
Artur Bil
Bazyli Blicharski
20 grudzień 2000

1 Wstęp

System Linux jest jednym z tych systemów, którego projektanci postanowili w pewien dość ciekawy sposób rozwiązać problem zapotrzebowania procesów na pamięć operacyjną (której to ilość, jak z doświadczenia wiadomo, prawie zawsze jest niewystarczająca) oraz ochrony pamięci poszczególnych procesów. Powodów ogromnego zapotrzebowania procesów na pamięć jest wiele, do głównych zaś należą: duży popyt na wszelkiego rodzaju pamięciożerne interfejsy graficzne oraz coraz to bardziej rozbudowane programy użytkowe, tworzone za pomocą specyficznych narządzi wizualnych, przyśpieszających proces tworzenia oprogramowania kosztem jego wielkości. Wieloużytkownikowość oraz wieloprocesowość zaś nakłada na system sprawowanie kontroli nad zasobami pamięci poszczególnych procesów ale także ochronę tychże obszarów przed innymi procesami. Stosowanym w systemie Linux mechanizmem rozwiązującym powyższe problemy jest mechanizm stronicowania. System ten zapewnia ochronę danych na poziomie poszczególnych procesów - żaden proces ńie widziśtron przydzielonych innemu procesowi (wyjątkiem jest pamięć dzielona) oraz dostęp do znacznie większej niż fizycznie osiągalnej ilości pamięci. Obsługa dostępu do tej pamięci (zasymulowanej na przestrzeniach pamięci masowych - zazwyczaj na dyskach twardych) nazywa się stronicowaniem na żądanie. Polega ono na usuwaniu chwilowo niepotrzebnych stron z pamięci fizycznej do pamięci pomocniczej. Odpowiednim "żąglowaniemśtronami pomiędzy pamięcią RAM a dyskiem zajmuje się demon kswapd, który co jakiś czas uruchamia się w tle i wedle odpowiedniego algorytmu usuwa nieużywane strony z pamięci (strategia ta nazywa się LFU - najrzadziej używane najpierw). Jego działanie stosuje się do trzech rodzajów stron:

- prywatnych stron procesu
- stron pamięci dzielonej
- stron będących obrazami plików

Szczególnie będzie nas interesować ten ostatni rodzaj stron, jako że z nim ściśle jest związany mechanizm obsługi podręcznej pamięci buforowej stron (to właśnie ta pamięć przechowuje strony mapowanych plików, używanych przez procesy).

2 Przestrzeń adresowa procesu

Przestrzeń adresowa każdego procesu jest podzielona na cześci, zwanymi obszarami pamięci wirtualnej (VMA - Virtual Memory Areas). Każdy z takich obszarów może mieć ustawione różne atrybuty ochrony, mechanizmy obsługi błędów oraz różne sposoby usuwania z pamięci fizycznej (dane procesu na przykład usuwane są z pamięci w inny sposób niż fragmenty plików) i jest zdefiniowany przez strukturę vm_area_struct zdefiniowaną w pliku include/linux/mm.h i opisaną poniżej:

```
struct vm_area_struct {
```

```
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct * vm_next;
    pgprot_t vm_page_prot;
    unsigned short vm_flags;
    short vm_avl_height;
    struct vm_area_struct * vm_avl_left;
    struct vm_area_struct * vm_avl_right;
    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;
    struct vm_operations_struct * vm_ops;
    unsigned long vm_offset;
    struct file * vm_file;
    unsigned long vm_pte;
};
```

Najważniejsze pola tej struktury to:

- **vm_mm** wskaźnik do całej pamięci wirtualnej procesu
- **vm_start** przesunięcie początku danego obszaru w przestrzeni adresowej danego procesu
- **vm_end** przesunięcie końca danego obszaru w przestrzeni adresowej danego procesu
- **vm_next** wskaźnik na następny obszar (wykorzystywane przy mapowaniu plikow
- **vm_page_prot** atrybuty ochrony danego obszaru, może przyjmować wartości:
  - PAGE_READONLY
  - PAGE_COPY
  - PAGE_SHARED
- **vm_flags** określa właściciwości danego obszaru, takie jak: prawa do wykonania, odczytu, zapisu, możliwości zmiany praw dostępu itp. Określają one również czy w obszarze tym znajduje się kod wykonywalny czy też dane procesu
- **vm_ops** struktura przechowująca wskaźniki do następujących procedur obsługi:
  - **open** używane przy dołączaniu danego obszaru do przestrzeni adresowej procesu
  - **close** używane przy usuwaniu obszaru z przestrzeni adresowej procesu
  - **nopage** używana przy obsłudze błędu braku strony (najistotniejsza procedura mechanizmu obsługi podręcznej pamięci buforowej stron)
  - **unmap** używana przy ódmapowywaniu"danych z pamięci procesu
  - **swapout** używana przy usuwaniu pamięci pomocniczej
  - **swapin** używana przy powrotnym wczytywaniu strony z urządzenia wymiany
- **vm_file** przechowuje wskaźnik do zamapowanego pliku
- **vm_offset** przesunięcie w tym pliku

3 Podręczna pamięć buforowa stron

Jak już wspomniałem zadaniem podręcznej pamięci buforowej stron jest przechowywanie w pamięci operacyjnej najczęściej (lub ostatnio) używanych stron mapowanych plików, przez co uzyskuje się znaczne przyśpieszenie w dostępie do plików znajdujących się na dysku. W momencie gdy dana strona jest potrzebna zostaje ona wczytywana do podręcznej pamięci buforowej stron, aby pozostać tam w razie gdyby była potrzebna po raz kolejny temu samemu lub innemu procesowi. Dzięki temu uzyskuje się znaczne przyśpieszenie (ten sam plik nie jest ładowany z dysku wielokrotnie tylko jeden raz, później przy kolejnej próbie dostępu jest już pobierany z odpowiedniego miejsca w pamięci) i redukcję ilości powtarzających się danych (tych samych stron wczytywanych z pliku) znajdujących się w pamięci. Podręczna pamięć buforowa składa z tablicy haszującej page_hash_table, która zawiera wskaźniki do struktur mem_map_t. Są to struktury opisujące poszczególne ramki pamięci, ich definicja znajduje się w include/linux/mm.h, zaś sama struktura wygląda tak:

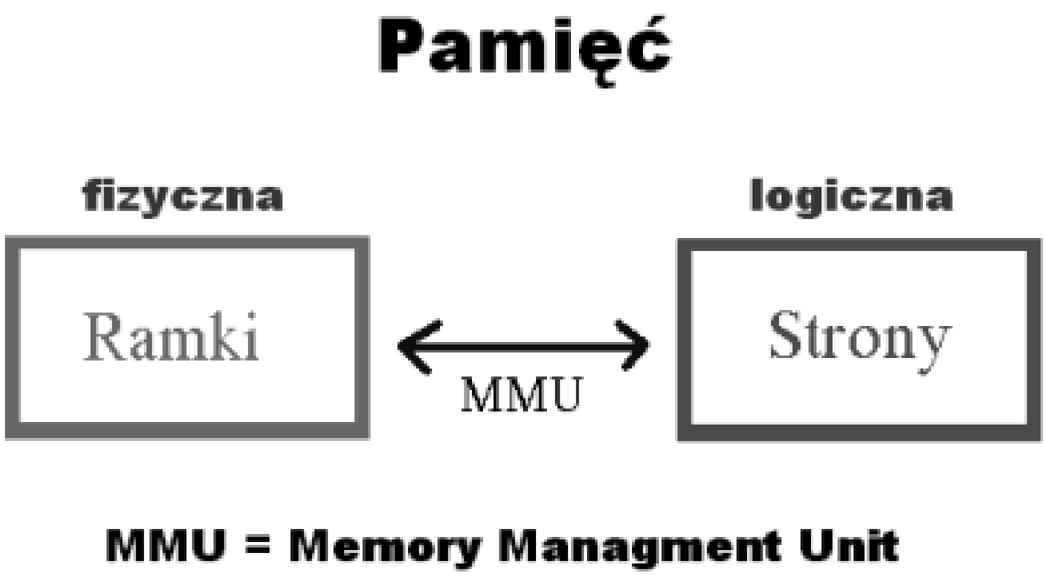
```
struct vm_area_struct {
   struct mm_struct * vm_mm;
   unsigned long vm_start;
   unsigned long vm_end;
   struct vm_area_struct * vm_next;
   pgprot_t vm_page_prot;
   unsigned short vm_flags;
   short vm_avl_height;
   struct vm_area_struct * vm_avl_left;
   struct vm_area_struct * vm_avl_right;
   struct vm_area_struct *vm_next_share;
   struct vm_area_struct **vm_pprev_share;
   struct vm_operations_struct * vm_ops;
   unsigned long vm_offset;
   struct file * vm_file;
   unsigned long vm_pte;
}
```

Dla przypomnienia dodam, że ramka jest jednostka opisującą jednostkę fizyczną pamięci (w przeciwieństwie do strony) (obrazek nr 1):

Struktury mem_map_t powiązane są ze sobą w listy dwukierunkowe za pomocą pól next_hash, prev_hash, zaś każdy indeks w tablicy haszującej **page_hash_table** zawiera wskaźnik do osobnej listy takich struktur. Aby jednoznacznie zidentyfikować mapowaną stronę wystarczy, że posiadamy i-węzeł (i-node) danego pliku (każdy plik w systemie Linux jest dany przez jeden unikatowy i-węzeł, przez co nie można napotkać kolizji) oraz przesunięcie w nim (offset). Na podstawie tych parametrów za pomocą odpowiedniej funkcji (**find_page**) wyliczana jest pozycja stron danego pliku w tablicy haszującej **page_hash_table**. Sama tablica **page_hash_table** jest zadeklarowana w pliku **mm/filemap.c** w następujący sposób:

```
struct page **page_hash_table
```

zaś inicjalizowana jest za pomocą funkcji **page_cache_init** (również **mm/filemap.c**). Pozostaje jeszcze sprawa wielkości tej tablicy (od tego oraz postaci funkcji haszującej zależy jak szybki



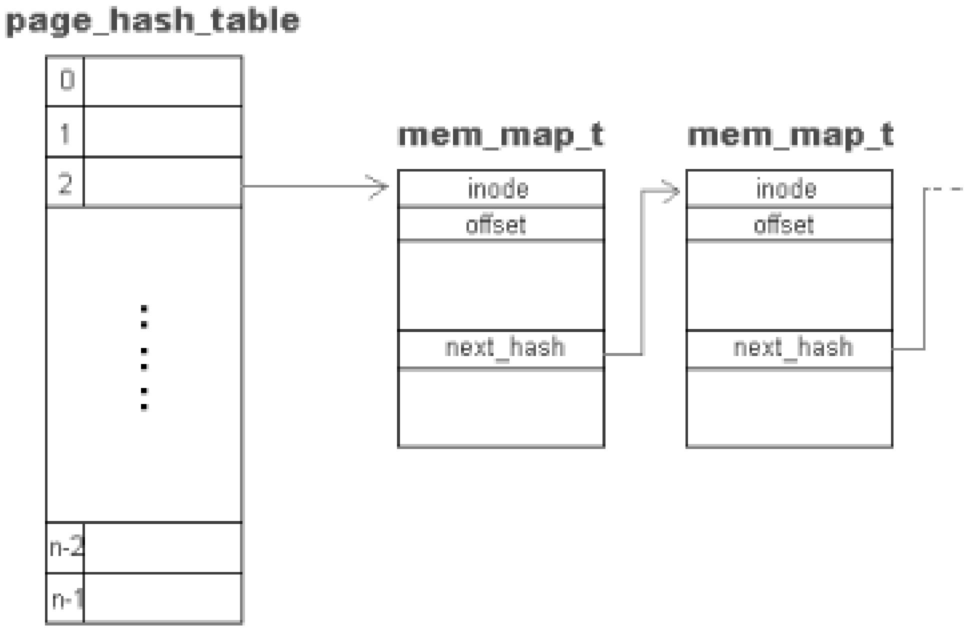
Rysunek 1: Pamięć fizyczna i logiczna

będzie dostęp do odpowiednich stron w podręcznej pamięci buforowej stron). Analizując poprzednie wersje jądra systemu Linux łatwo zauważyć, że wielkość tej tablicy się zmieniała. Aktualnie (**wersja 2.2.17**) nie jest ona określona stałą (jak to miało miejsce dotychczas - np. w **wersji 2.2.12** jej wielkość to 4096 elementów tablicy **page_hash_table**) i zależy tylko i wyłącznie od wielkości pamięci operacyjnej komputera. O tym ile indeksów będzie posiadała tablica page_hash_table decyduje właśnie funkcja **page_cache_init**, która też alokuje pamięć dla tej struktury. Poniższy rysunek przedstawia schematyczny wygląd tablicy **page_hash_table** (obrazek nr 2):

Funkcja **find_page** znajdująca się w **mm/filemap.h** potrafi na podstawie podanego i-węzła oraz przesunięcia w pliku obliczyć indeks do odpowiedniego wiersza tablicy **page_hash_table** po czym wskazać poszukiwaną stronę na liście (lub jej brak). Schemat obsługi podręcznej pamięci buforowej stron jest opisany następującym algorytmem (obrazek nr 2):

Algorytm ten jest wykorzystywany przez najważniejszą funkcję obsługi podręcznej pamięci buforowej stron, a mianowicie przez **filemap_nopage** znajdującej się w **mm/filemap.c**. Parametrem jej uruchomienia jest obszar pamięci typu **vm_area_struct**, adres dla którego nastąpił błąd braku strony (na jego podstawie jest wyliczany offset w pliku, dla którego trzeba szukać stron w pamięci podręcznej) oraz zmienna logiczna **no_share**, która przyjmuje wartość true, gdy poszukiwana strona ma być załadowana na wyłączność procesu, który uruchomił tą funkcję (w takim przypadku zawsze jest robiona prywatna kopia strony). Poniżej znajduje się kod funkcji **filemap_nopage** wraz z odpowiednimi komentarzami.

```
static unsigned long filemap_nopage(struct vm_area_struct * area,
unsigned long address, int no_share)
{
```



Rysunek 2: Tablica haszująca **page_hash_table**

```
struct file * file = area->vm_file;
struct dentry * dentry = file->f_dentry;
struct inode * inode = dentry->d_inode;
unsigned long offset, reada, i;
struct page * page, **hash;
unsigned long old_page, new_page;

new_page = 0;
/* wyliczany jest offset w pliku  */
offset = (address & PAGE_MASK) - area->vm_start + area->vm_offset;
if (offset >= inode->i_size && (area->vm_flags & VM_SHARED) && area->vm_mm == current->mm)
goto no_page;  /*  adres poza plikiem!  */
/*  znajdujemy adres odpowiedniego indeksu w tablicy haszujacej */
hash = page_hash(inode, offset);
/*  ...i szukamy, czy potrzebnej strony w niej nie ma  */
page = __find_page(inode, offset, *hash);
if (!page)
goto no_cached_page; /*  nie znaleziono strony  */
```

```
found_page:
/*  znaleziono szukana strone  */
if (no_share && !new_page) {
/*  parametr no_share - alokujemy nowa strone  */
new_page = page_cache_alloc();
if (!new_page)
goto release_and_oom;
}

/*  sprawdzamy czy strona nie ma zalozonej blokady  */
if (PageLocked(page))
goto page_locked_wait;
/*  oraz czy nie jest przedawniona  */
if (!PageUptodate(page))
goto page_read_error;

success:
old_page = page_address(page);
if (!no_share) {
/*  nie potrzebujemy strony na wyłączność, wiec używamy tej znalezionej  */
if (new_page)
page_cache_free(new_page);

flush_page_to_ram(old_page);
return old_page;
}

/* wykonujemy kopię strony - potrzebujemy jej na wyłączność */
copy_page(new_page, old_page);
/* na niektórych platformach potrzebujemy przerzucić stronę do pamięci, jeśli
     jest w cache podręcznym procesora (w procesorach i386 jest to inst. pusta)  */
flush_page_to_ram(new_page);
page_cache_release(page);
return new_page;

no_cached_page:
/*  próbujemy czytać cały klaster (heurystyka - działa szybciej) */
reada   = offset;
reada >>= PAGE_CACHE_SHIFT + page_cluster;
reada <<= PAGE_CACHE_SHIFT + page_cluster;

for (i = 1 << page_cluster; i > 0; --i, reada += PAGE_CACHE_SIZE)
new_page = try_to_read_ahead(file, reada, new_page);

if (!new_page)
```

```
new_page = page_cache_alloc();
if (!new_page)
goto oom;

/*  proces mógł pospać podczas tej operacji, więc jeszcze raz sprawdzamy obecność
     szukanej strony w page cache */
page = find_page(inode, offset);
if (page)
goto found_page;

/*  nie znaleźliśmy więc zaalokowaną stronę wrzucamy do page cache  */
page = page_cache_entry(new_page);
new_page = 0;
add_to_page_cache(page, inode, offset, hash);

if (inode->i_op->readpage(file, page) != 0)
goto failure;

goto found_page;

page_locked_wait:
/*  oczekujemy na stronę  */
__wait_on_page(page);
if (PageUptodate(page))
goto success;

page_read_error:
/*  tu wchodzimy w sytuacji, gdy strona była przedawniona  */
if (inode->i_op->readpage(file, page) != 0)
goto failure;
wait_on_page(page);
if (PageError(page))
goto failure;
if (PageUptodate(page))
goto success;

failure:
/*  nie udało się, zwalniamy pamięć  */
page_cache_release(page);
if (new_page)
page_cache_free(new_page);
no_page:
return 0;

release_and_oom:
page_cache_release(page);
```

```
oom:
return -1;
}
```

Jak widać procedura ta w pełni opiera się na wspomnianym wcześniej algorytmie. Ciekawym jest sposób w jaki czyta ona stronę do podręcznej pamięci buforowej stron, w przypadku gdy jej tam nie ma - próbuje czytać "ile się da", ale nie więcej niż jeden klaster. Jest to tak zorganizowane, gdyż analitycy systemów obliczyli i doświadczenie pokazało, że w momencie gdy proces odwołuje się do strony pewnego pliku to jest ogromne prawdopodobieństwo, że zaraz poprosi o następną (dzieje się tak w szczególności w przypadku czytania plików wykonywalnych, które oczywiście wykonują się sekwencyjnie). Dlatego czytanie z wyprzedzeniem po kilka stron naraz jest rozwiązaniem praktycznym i efektywnym. Pozostaje jeszcze pytanie kiedy strony są usuwane z podręcznej pamięci buforowej stron? Dzieje się to w momencie, gdy w systemie pojawia się zapotrzebowanie na pamięć (został na przykład uruchomiony nowy, wymagający dużej ilości pamięci proces) - wtedy według odpowiedniej strategii z pamięci podręcznej wyrzucane są kolejne strony: najpierw te, o które nie ubiega się już żaden proces (czyli wszystkie procesy, które wcześniej źgłaszały się"po tę stronę umarły), po czym kolejne takie, które mają najmniejszą liczbę procesów o nie się ubiegających. Jednym słowem podręczna pamięć buforowa stron rozrasta aż do momentu, gdy nie zabraknie pamięci. Poniżej znajduje się opis najważniejszych funkcji wykorzystywanych przez procedurę filemap_nopage:

- **page_hash** funkcja haszująca, która dla konkretnego i-węzła oraz offsetu w pliku znajduje w page_hash_table adres listy, w której powinna się zawierać odpowiednia strona
- **find_page** funkcja znajdująca w odpowiedniej liście stronę zamapowanego pliku, na podstawie danego i-węzła, offsetu oraz indeksu w tablicy haszującej (w postaci adresu)
- **page_cache_alloc** funkcja alokująca pamięć dla nowej strony w podręcznej pamięci buforowej stron
- **PageLocked** funkcja sprawdzająca czy jakiś proces nie nałożył na tę stronę blokady
- **PageUpToDate** funkcja dokonująca sprawdzenia, czy podana w parametrze strona nie uległa przedawnieniu (poprzez np. nadpisanie pliku przez jakiś proces)
- **flush_page_to_ram** ta procedura przerzuca strone do pamieci, jeśli ta się tam nie znajduje. Ma ona zastosowanie w niektórych typach procesorów, które udostępniają własne cache-oweanie
- **copy_page** procedura kopiująca stronę
- **page_cache_release** funkcja zwalniająca pamięc przydzieloną stronie
- **try_to_read_ahead** funkcja próbująca czytać z wyprzedzeniem z pliku
- **add_to_page_cache** funkcja, której zadaniem jest dodanie zaalokowanej już strony do page_hash_table

Mechanizm podręcznej pamięci buforowej stron jest jak widać mechanizmem bardzo wydatnym i znacznie zwiększającym prędkość pracy systemu. Dzięki odpowiedniej gospodarce pamięcią (rozrastanie się podręcznej pamięcy buforowej stron dopóty dopóki nie pojawi się zapotrzebowanie na zajmowaną pamięć) jest metodą może dość dziwną, jednakże bardzo skuteczną. Podobnie się ma sprawa z czytaniem z pliku z wyprzedzeniem - zastosowanie tej heurystyki powoduje ono również znaczną poprawę osiągnięć.

Bibliografia

[1] opracowania tematów dotyczących pamięci w systemie Linux z lat poprzednich, znajdujące się na witrynie internetowej http://rainbow.mimuw.edu.pl/so

[2] kody zródłowe jądra linuxa w wersji 2.2.17 (w szczegolnosci pliki include/linux/pagemap.h i mm/filemap.c)

[3] inne zasoby internetowe

[4] Maurice J. Bach - "Budowa systemu operacyjnego UNIX"

Rysunek 3: Algorytm obsługi podręcznej pamięci buforowej stron