Obsługa tablicy ramek

Karol Bieńkowski
(kb181029@zodiac.mimuw.edu.pl)

20 grudnia 2000

Temat nr 25.

Opracowanie zawiera opis struktur używanych do przechowywania informacji o ramkamch oraz procedur do ich obsługi. Dotyczy zarówno zarządzania wolnymi ramkami jak i ramkami wykorzystywanymi na potrzeby systemu lub użytkowników.

1 Wprowadzenie

Linux używa systemu zarządzania pamięcią zwanego stonicowaniem. Dokładniejszy opis tego systemu znajduje się w rozdziale poświęconym tablicy stron, tu należy się jednak krótkie przypomnienie. Każdy proces w systemie dysponuje własną, wirtualną przestrzenią adresową. Rozmiar tej przestrzeni zależy od architektury procesora, a nie od ilości fizycznie zainstalowanej pamięci RAM. Tu opisuję architekturę i386, gdzie rozmiar pamięci wirtualnej wynosi 4 GB.

Z punktu widzenia procesu pamięć to ciągły ciągły zbiór bajtów adresowanych kolejnymi liczbami od 0 do 4 GB - 1. Jednak system operacyjny dzieli tę pamięć (wirtualną!) na **strony**, czyli kawałki o ustalonej wielkości. Wielkość strony, zależna od architektury, wynosi 4 kB dla i386. Konsekwencją podziału pamięci wirtualnej na strony jest podział pamięci fizycznej na **ramki**. Ramka jest to cząstka pamięci operacyjnej wielkości strony, czyli 4 kB. Tak jak cała pamięć procesu jest podzielona na strony, tak cała pamięć operacyjna jest podzielona na ramki.

Proces odwołując się do pamięci używa adresów wirtualnych. System operacyjny, korzystając ze wsparcia sprzętowego, tłumaczy te adresy na adresy fizyczne. Do konwersji jest używana tablica stron, oddzielna dla każdego procesu. W tej tablicy indeksowanej numerami stron znajdują się numery ramek. Numer ramki określa jednoznacznie adres fizyczny - ramki zapełniają kolejno całą pamięć operacyjną, ramka o numerze 0 zaczyna się od adresu 0. Ilość ramek w systemie jest łatwo wyznaczyć - to wielkość pamięci dzielona przez rozmiar strony. Informacja o ramkach jest trzymana globalnie, każdej ramce odpowiada w systemie jedna struktura ją opisująca. W przypadku stron sytuacja jest inna - każdy proces ma struktury opisujące jego strony.

Uwaga terminologiczna: polskie nazewnictwo wprowadza rozróżnienie między strony i ramki, jednak w tekstach angielskich używa się terminu *page* dla obu pojęć, *frame* ma trochę inne znaczenie.

Strukturą opisującą pojedynczą ramkę jest `struct page` zdefiniowana w pliku `include/linux/mm.h` w sposób następujący:

```
typedef struct page {
    /* these must be first (free area handling) */
    struct page *next;
    struct page *prev;
    struct inode *inode;
```

```
    unsigned long offset;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
        /* atomic flags, some possibly updated asynchronously */
    struct wait_queue *wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
} mem_map_t;
```

Można więc też zamiast określenia `struct page` używać typu `mem_map_t`. Pola tej struktury mają różne znaczenie w zależności od tego jak ramka jest wykorzystywana. Prosty przykład: jeżeli ramka jest wolna to `page->next` pokazuje na następną wolną ramkę, a jeżeli ramka jest używana do mapowania pliku to `page->next` oznacza następną ramkę związaną z danym plikiem. Oczywiście wykorzystanie każdej ramki może zmieniać się w trakcie działania systemu. W dalszych punktach zostaną opisane różne zastosowania ramek, i odpowiadające im zastosowania pól struktury `page`.

Pola `page->count` i `page->flags` mają zawsze to samo znaczenie, niezależnie od sposobu wykorzystania ramki. Licznik `page->count` jest używany do wskazania liczby odniesień do ramki, czyli liczby jej użytkowników. Jeżeli `page->count` == `0` to ramka jest wolna, a `page->count` == `1` oznacza, że ramka jest używana w dokładnie jednym celu. `Pole page->flags` na pojedynczych bitach przechowuje flagi opisujące stan ramki.

2 Tablica wszystkich ramek mem_map

W systemie istnieje globalna tablica przechowująca struktury `page` - jest to `mem_map`. Tablica ta to ciąg struktur `page`, a nie wskaźników do nich. Dlaczego? Każdej ramce odpowiada dokładnie jedna struktura `page`, więc wielkość tablicy jest określona i niezmienna. Zatem można zrezygnować z narzutu związanego z przechowywaniem wskaźników. Pamięć na tablicę `mem_map` jest przydzielana w momencie startu systemu, a nie w trakcie kompilacji jądra. Powód jest prosty - w trakcie kompilacji nie wiadomo jaką pamięcią fizyczną będzie dysponował komputer. Po dołożeniu dodatkowych megabajtów RAM-u nie ma konieczności rekompilacji jądra.

Numer ramki to indeks pod jakim ramka występuje w tablicy `mem_map`. Ramka zerowa - `mem_map[0]` zaczyna się od adresu `0`, a kończy na `4k - 1`. Przy odwołaniach do tablicy `mem_map` najczęściej jest używana konstrukcja `mem_mam[MAP_NR(vaddr)]`, gdzie `MAP_NR` to makro dające numer ramki odpowiadający adresowi wirtualnemu `vaddr` w kontekście jądra. Drugi sposób dostania się do struktury `page` to odwołanie do `mem_map + map_nr`, gdzie `map_nr` to żądany numer ramki. Szczegóły w punkcie opisującym makra (patrz p. 9).

2.1 Deklaracja, definicja i inicjalizacja mem_map

Tablica `mem_map` jest zadeklarowana w pliku `include/linux/mm.h`:

```
extern mem_map_t * mem_map;
```

Jej definicja (czyli przydział miejsca w pamięci) następuje w pliku `mm/memory.c`:

```
mem_map_t * mem_map = NULL;
```

Jak widać ta definicja wcale nie przydziela miejsca na tablicę, służy tylko do oszukania kompilatora podczas kompilacji jądra. Prawdziwy przydział pamięci następuje w funkcji `free_area_init` podczas startu systemu. Dzieje się to w kolejno wywoływanych funkcjach inicjalizujących:

- free_area_init (mm/paging.c), wołana przez paging_init (arch/i386/mm/init.c) - przydziela pamięć dla tablicy mem_map. Mechanizm przydzielania pamięci na struktury w trakcie startu systemu jest następujący: kolejno wywoływane funkcje (np. sched_init, time_init) dostają jako parametry początek i koniec dostępnej pamięci (start_mem i end_mem), tworzą swoje struktury na początku pamięci i zwracają początek przesunięty o tyle ile zajęły. Tak też powstaje mem_map:

```
mem_map = (mem_map_t *) LONG_ALIGN(start_mem);
p = mem_map + MAP_NR(end_mem);
start_mem = LONG_ALIGN((unsigned long) p);
memset(mem_map, 0, start_mem - (unsigned long) mem_map);
```

Gdzie: p jest zmienną pomocniczą typu mem_map_t, a makro LONG_ALIGN zaokrągla w górę do wielokrotności 4. Zaraz potem każdej stronie ustawia się flagi PG_reserved i PG_DMA, a page->count się zeruje.

- mem_init (mm/init.c) - czyści się bit PG_reserved ramkom, które nie są wykorzystywane przez jądro. Ramkom zarezerwowanym wpisuje się page->count = 1, żeby zaznaczyć, że są używane. Pozostałe mają page->count == 0 ustawione przez free_area_init. Zeruje się bit PG_DMA dla ramek, które nie mogą być objęte mechanizmem DMA, dla i386 są to adresy powyżej 0x1000000 == 16 MB.

- free_initmem - zwalnia, tzn. czyści bit PG_reserved, dodatkowym ramkom używanym przez jądro w czasie inicjalizacji. Wyświetla przy tym znany napis: *Freeing unused kernel memory: Xk freed.*

3 Lista wolnych ramek free_area

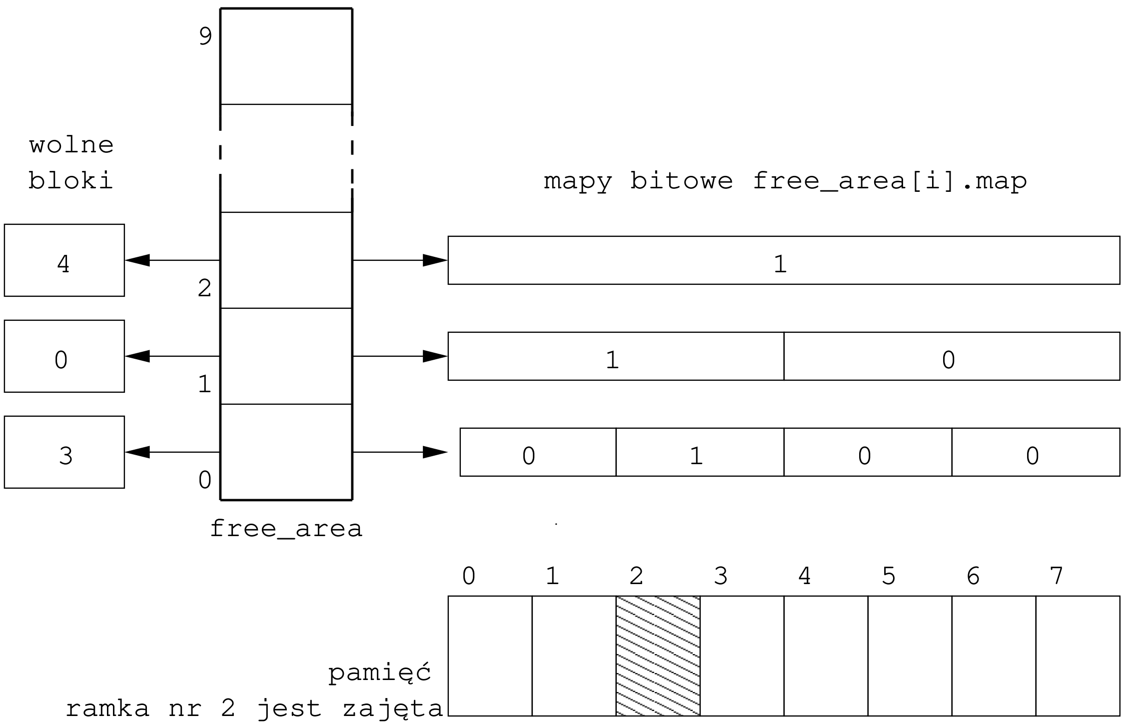
Jednym z głównych zadań jądra jest przydzielanie procesom pamięci. Linux do przydziału ramek używa **strategii buddy**. Dokładny opis tej strategii znajduje się w rozdziałach poświęconych przydziałowi i zwalnianiu bloków pamięci. Tutaj przedstawię jedynie jej zarys, wymagany do zrozumienia opisywanych struktur danych. Jednostką przydziału pamięci jest **blok** składający się z 2 ^ i ramek. Wolne bloki są przechowywane w tablicy list free_area. Każda lista zawiera bloki tej samej wielkości, na i-tej liście znajdują się bloki rozmiaru 2 ^ i ramek. Oto definicja głowy listy przechowującej bloki (mm/page_alloc.c):

```
/* The start of this MUST match the start of "struct page" */
struct free_area_struct {
    struct page *next;
    struct page *prev;
    unsigned int *map;
    unsigned long count;
};
```

Wskaźniki free_area_struct->next i free_area_struct->prev tworzą listę dwukierunkową, której elementami są struktury page opisujące ramki. Na liście znajdują się tylko ramki rozpoczynające blok, np. dla bloku [4,5,6,7] na liście jest ramka 4. Do tworzenia listy używane są wskaźniki page->next i page->prev ze struktury page, stąd wymaganie by struktury page i free_area_struct miały takie same początki. Wymaganie to jest wyszczególnione w komentarzach towarzyszących definicjom obu struktur.

W polu free_area_struct->count przechowuje się liczność listy.

Mapa bitowa free_area_struct->map zawiera informacje niezbędne do łączenia bloku z jego **bliźniakiem**. W algorytmie buddy każdy blok ma swojego bliźniaka, takiego samego rozmiaru jak on. Bliźniaki powstają z podziału na połowy większego bloku, więc są sąsiadami. Przykłady:



Rysunek 1: Wolne ramki

bliźniakiem dla [1] jest [0], bliźniakiem dla [0,1] jest [2,3]. Każdy bit w `free_area_struct->map` zawiera informację o dwóch bliźniakach: jest równy `1` gdy dokładnie jeden z bliźniaków jest wolny, w przeciwnym razie `0` (patrz rysunek 1). W każdej mapie bitowej pierwszy bit odpowiada za pierwsze dwa bloki, drugi bit za trzeci i czwarty blok, itd. Ogólnie bit `(map_nr » (1 + order))` odpowiada za blok rozpoczynający się od ramki o numerze `map_nr`, gdzie 2 ˆ `order` to rozmiar bloku.

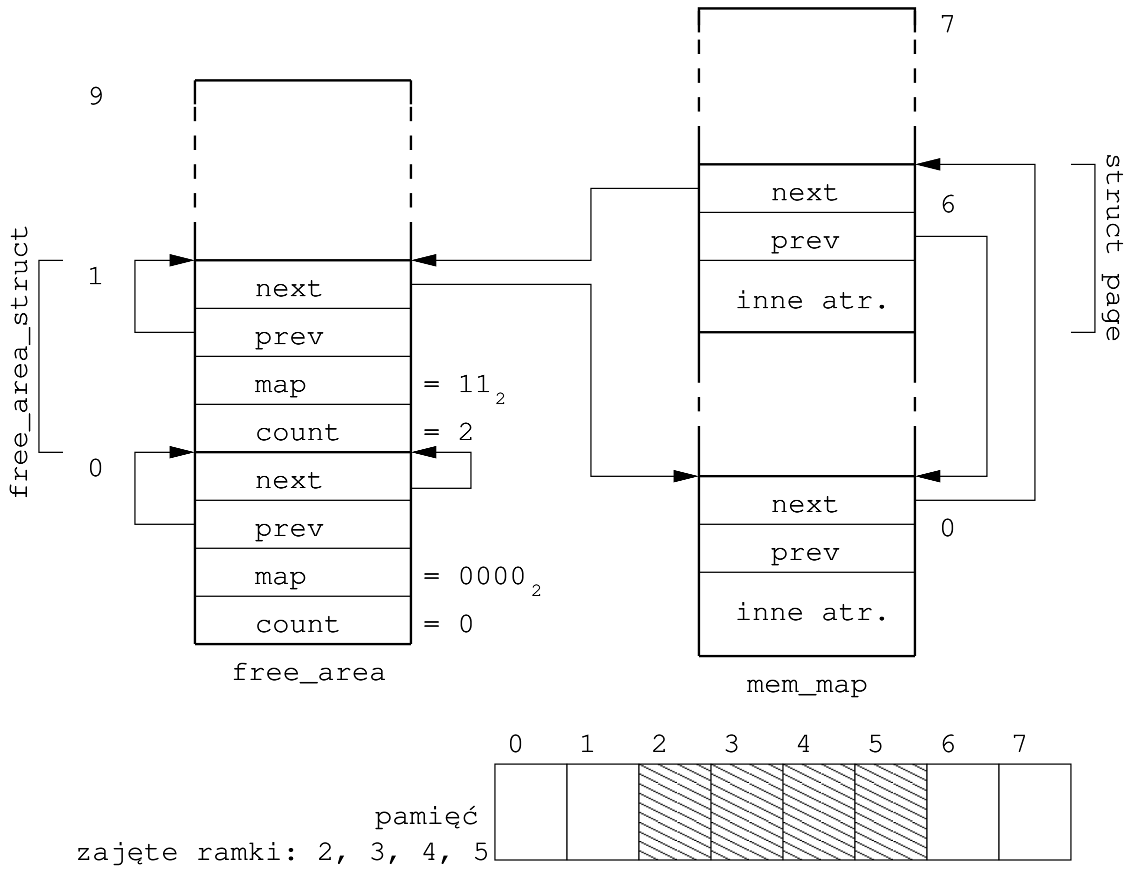
Wielkość mapy bitowej zależy od wielkości bloków. Liczba bitów w i-tej mapie jest równa połowie liczby bloków danej wielkości 2 ˆ i, w szczególności mapa zerowa (bloki to ramki) ma dwa razy mniej bitów niż jest ramek w pamięci. Zerowa mapa bitowa jest największa, każda następna maleje o połowę.

Listy bloków są zebrane w tablicy `free_area`:

```
static struct free_area_struct free_area[NR_MEM_TYPES][NR_MEM_LISTS];
```

Wolna pamięć jest dzielona na dwa rodzaje, ramki każdego rodzaju są trzymane oddzielnie. Rozróżnienie ma związek z mechanizmem DMA - w tablicy `free_area[1]` są trzymane ramki, które mogą zostać użyte przez bufory mechanizmu DMA, pozostałe w `free_area[0]`. Podział nie dotyczy wszystkich architektur, ale jest ważny w przypadku i386, gdzie dla mechanizmu DMA są dostępne tylko obszary pamięci poniżej 16 MB. W tym przypadku ramki z tablicy `free_area[1]` są cenniejsze. Tak więc `NR_MEM_TYPES` wynosi 2 (nie-DMA i DMA).

`NR_MEM_LISTS` oznacza liczbę dozwolonych wielkości bloków. Dla architektury i386 `NR_MEM_-LISTS` jest równe `10`, zatem bloki mają rozmiary od jednej do 2 ˆ 9 == 512 ramek (2 MB).



Rysunek 2: Szczegółowy rysunek struktury free_area oraz jej powiązanie z mem_map

3.1 Operacje na liście wolnych bloków

Do operacji na listach wolnych bloków są zdefiniowane następujące funkcje:

```
static inline void init_mem_queue(struct free_area_struct * head);
static inline add_mem_queue(struct free_area_struct * head, struct page * entry);
static inline remove_mem_queue(struct page * entry);
```

oraz makro:

```
#define memory_head(x) ((struct page *)(x))
```

Rola ww. funkcji to manipulacja wskaźnikami, tak by zachować odpowiednie powiązanie między elementami listy (patrz też rysunek 2). Nowe elementy są wstawiane na początek listy. Wskaźnik `prev` pierwszego elementu i `next` ostatniego wskazują na `memory_head(głowa listy na której się znajdują)`. Funkcja `add_mem_queue` poza zmianą wskaźników zwiększa `head->count`, a `remove_mem_queue` tylko przestawia wskaźniki. Zatem `count` trzeba zmniejszyć ręcznie.

3.2 Inicjalizacja

Tablica `free_area` jest alokowana statycznie w trakcie kompilacji jądra, gdyż jej rozmiar jest znany. Jednak rozmiary map bitowych `free_area_struct->map` zależą od ilości pamięci fizycznej, więc są tworzone i zerowane dopiero przy starcie systemu. Dzieje się to w funkcji `free_area_init` (z `mm/paging_init.c`), gdzie są również inicjowane kolejki wolnych obszarów (`init_mem_queue`). Szczegóły nt. ustalania wielkości pola `map` znajduje się w dodatku na końcu rozdziału (p. 10).

4 W ramce jest mapowany plik

Z mapowaniem pliku w ramce mamy do czynienia wtedy, gdy część pliku jest umieszczona w pamięci. Większość plików nie mieści się w całości w jednej ramce, poza tym zwykle nie ma potrzeby sprowadzania do pamięci całego pliku. Zatem pojedyncza ramka zawiera tylko kawałek pliku i trzeba pamiętać, który to kawałek. Do tego służą pola struktury `page`. `Page->inode` wskazuje, który plik jest mapowany, podając wskaźnik do jego **iwęzła** (ang. *inode*), a `page->offset` miejsce w pliku, z którego zostały pobrane dane. Poza tym wykorzystywane są `page->next` i `page->prev` - są wskaźnikami do listy dwukierunkowej zawierającej wszystkie ramki używane do mapowania pojedynczego pliku. Oprócz list ramek dla poszczególnych plików istnieje w systemie globalna tablica haszująca `page_hash_table`. Kluczem do niej jest para `(inode, offset)`, więc pozwala na znalezienie ramki z zadaną częścią pliku. W każdej z pozycji tablicy `page_hash_table` może znajdować się lista ramek - gdy są ramki o tej samej wartości funkcji haszującej. Wtedy wskaźniki `page->next_hash` i `*page->pprev_hash` pokazują na odpowiednio następną i poprzednią ramkę znajdującą się na tej liście. Za każdym razem jak ramka zostanie odnaleziona przez tablicę haszującą `page_hash_table` to zostanie jej ustawiony bit `PG_referenced` oraz jest zwiększany licznik `page->count`, aby zliczać odniesienia do ramki. Te informacje są używane przez funkcję wymiatającą ramki do pliku wymiany (`shrink_mmap`).

Licznik `page->count` wskazuje na liczbę przyłączeń (ang. *attaches*) pliku, plus 1 jeżeli na ramce są bufory.

4.1 Wymiana ramek

Szczególnym przypadkiem mapowania jest mapowanie pliku wymiany. Oznacza to, że obraz ramki znajduje się w pliku wymiany. W tym przypadku ramka ma ustawioną flagę `PG_swap_cache`, `page->offset` wskazuje pozycję w pliku wymiany a `page->inode` pokazuje na "wirtualny" iwezęł pliku wymiany (`swapper_inode`).

4.2 Operacje na listach

Następujące funkcje (zdefiniowane w `include/linux/pagemap.h`) są używane do dodawania i usuwania ramki z listy ramek skojarzonych z iwęzłem oraz z tablicy haszującej `page_hash_table`: `add_page_to_inode_queue`, `remove_page_from_inode_queue`, `add_page_to_hash_queue`, `remove_page_from_hash_queue`.

5 Ramki na bufory

Jeżeli ramka jest wykorzystywana na bufory (dyskowe) to `page->buffers` pokazuje na listę nagłówków buforów umieszczonych na tej ramce. Bufory są powiązane w listę przy użyciu wskaźników `buffer_head->b_this_page`.

Jeżeli na ramce nie ma buforów to `page->buffers` jest równe `NULL`.

6 Ramki na pamięć jądra

Ramki mogą być wykorzystywane na struktury jądra. Ta pamięć jest zdobywana przez jądro funkcją `kmallloc`. Szczegóły znajdują się w rozdziale poświęconym tej funkcji. Ramka należąca do tego rodzaju pamięci ma ustawioną flagę `PG_Slab`. Wskaźnik `page->next` wskazuje na **cache** (`kmem_cache_t`) do którego należy ramka. Dzięki temu można znaleźć *cache* znając tylko adres wirtualny obiektu w pamięci. Jest to wykorzystywane w funkcji `kfree`. Patrz zadanie 1.

7 Ramki używane przez procesy

Powyższe punkty dotyczyły ramek zarządzanych przez system: wolnych ramek, ramek używanych do mapowania plików i ramek na bufory. Ramki mogą też być wykorzystywane przez zwykłe procesy do przechowywania swoich danych. Są one przydzielane procedurą `__get_free_pages`. W tym przypadku `page->count` musi wynosić przynajmniej `1`, co oznacza, że ramka nie jest wolna. Ustawianie licznika na `1` odbywa się w funkcji `__get_free_pages`. Pozostałe pola struktury `page` nie są używane, ale jest zalecane by miały wartość `0` lub `NULL`.

8 Operacje wejścia-wyjścia na ramkach

Na ramkach są wykonywane operacje wejścia-wyjścia. Jest wiele okazji ku temu:

- jeżeli w ramce jest mapowany plik to trzeba go odczytać z dysku,
- jeżeli tenże plik został zmodyfikowany, to trzeba zmienioną wersję zapisać na dysk,
- jeżeli ramka procesu ma zostać "wyswapowana", a nie ma jej w pliku wymiany albo jest tam jej nieaktualna wersja to trzeba ją zapisać.

Operacje dyskowe są długotrwałe, więc nie można sobie pozwolić na blokowanie jądra na cały czas ich trwania. Nie można także pozwolić, żeby ramka była używana w trakcie przesyłania danych. W tym celu ustawia się flagę `PG_locked` tuż przed operacją we/wy, a zeruje się ją po zakończeniu operacji. Tak więc w jej trakcie transferu ramka nie jest dostępna i każdy proces, który chce z niej korzystać usypia i dopisuje się do kolejki `page->wait`. Po skończonej operacji ustawia się także flagę `PG_uptodate` oznaczającą, że zawartość strony jest aktualna lub `PG_error`, gdy nastąpił błąd we/wy.

9 Dodatek A. Stałe i makra dostępu do tablicy mem_map

Przy dostępie do tablicy ramek (`mem_map`) oraz przy innych czynnościach wymagającego bezpośredniego dostępu do pamięci w kodzie jądra używa się makr. Warto sobie je przyswoić, ponieważ są używane często, ich znajomość ułatwia zrozumienie kodu Linuxa.

- `PAGE_SHIFT` - definiuje wielkość strony/ramki jako 2 ^ `PAGE_SHIFT`. Jest zdefiniowane niezależnie od `PAGE_SIZE`, gdyż wiele operacji w jądrze jest wykonywanych bezpośrednio na bitach, np. `(nr » PAGE_SHIFT)`. Dla architektury i386 wynosi `12`;
- `PAGE_SIZE` - wielkość strony/ramki, zdefiniowane jako 2 ^ `PAGE_SHIFT`, czyli 4k dla i386;
- `PAGE_MASK` - maska bitowa, ma jedynki na pozycjach adresu odpowiadających numerowi strony, a zera na ostatnich 12-bitach odpowiadających offsetowi. Używana np. do zerowania offsetu: `vaddr =& PAGE_MASK`;

- PAGE_OFFSET - Jądro jest tak kompilowane, że jego adresy wirtualne zaczynają się od adresu PAGE_OFFSET, który jest standardowo równy 0xC0000000 = 3 GB. PAGE_OFFSET jest używany do przeliczania adresów wirtualnych na rzeczywiste - od adresu wirtualnego wystarczy odjąć PAGE_OFFSET, aby otrzymać adres fizyczny. Ustawienia PAGE_OFFSET dokonuje się przy kompilacji jądra (opcja dostępna w make menuconfig).

  Granica PAGE_OFFSET dzieli pamięć wirtualną na dwie części: pamięć poniżej PAGE_OFFSET dostępną w trybie użytkownika i powyżej PAGE_OFFSET dostępną tylko w trybie jądra. Ta druga musi być wystarczająco duża, by jądro miało dostęp do całej pamięci fizycznej. Przecież jądro to tak naprawdę część procesu, więc przy dostępie do pamięci używa adresów wirtualnych. Przykład: adres fizyczny 0x1234 jest osiągalny przez jądro pod adresem wirtualnym 0xC0001234.

  Tak więc standardowe ustawienie PAGE_OFFSET na 3 GB umożliwia wykorzystanie tylko do 1 GB RAM-u, gdyż przestrzeń adresowa jądra ma wielkość tylko 1 GB (adresy od 3 do 4 GB). W systemach mających więcej pamięci PAGE_OFFSET ustawia się na 2 GB lub nawet 1 GB.

- MAP_NR(vaddr) - podaje numer ramki, w której znajduje się dany adres wirtualny, przy założeniu, że vaddr dotyczy pamięci jądra. MAP_NR odejmuje od adresu PAGE_OFFSET, aby uzyskać adres fizyczny i przesuwa wynik o PAGE_SHIFT bitów w prawo, żeby pozbyć się bitów określających przesunięcie w ramce. Warto zauważyć iż prostota tej operacji jest możliwa dzięki temu, że rozmiar strony/ramki jest potęgą dwójki. Gdyby tak nie było to makro MAP_NR wymagałoby dzieleń;

- end_mem - zmienna określająca maksymalny dostępny adres wirtualny;

- max_mapnr = num_physpages = MAP_NR(end_mem) - określa ilość ramek w systemie (przepisane z arch/i386/mm/init.c).

10 Dodatek B. Określanie wielkości bitmap w free_area

Oto jak są ustalane rozmiary map bitowych dla wolnych obszarów podczas startu systemu w funkcji free_area_init z mm/page_alloc.h. Tworzymy bitmapę dla bloków rozmiaru 2 ˆ i ramek:

```
bitmap_size = (end_mem - PAGE_OFFSET) >> (PAGE_SHIFT + i);
bitmap_size = (bitmap_size + 7) >> 3;
bitmap_size = LONG_ALIGN(bitmap_size);
```

Gdzie: `#define LONG_ALIGN(x) (((x)+(sizeof(long))-1)& ((sizeof(long))-1))` - podaje pierwszą wielokrotność sizeof(long) == 4, większą lub równą x.

Pierwsza insrukcja liczy liczbę bloków wielkości 2 ˆ i w pamięci, druga dzieli wynik przez 8 (tyle bitów ma bajt) a trzecia zaokrągla do wielokrotności słowa (4 bajty). Widać więc, że na każdą mapę i-tą bitową jest przeznaczane tyle bitów ile jest bloków wielkości 2 ˆ i. To o dwa razy za dużo! Algorytm buddy wykorzystuje tylko połowę bitów z każdej mapy, gdyż jeden bit odpowiada za dwa bloki. Można więc pierwszą instrukcję zastąpić instrukcją:

```
bitmap_size = (end_mem - PAGE_OFFSET) >> (PAGE_SHIFT + i + 1);
```

żeby uzyskać dodatkowe 8 kB pamięci RAM (przy założeniu, że w systemie jest 128 MB). Patrz też zadanie 3.

11 Dodatek C. Opis wszystkich flag ramek

W tym punkcie są zebrane wszystkie flagi służące do opisu ramek. Większość z nich została opisana wyżej, więc tu znajduje się jedynie któtki opis i odnośnik do właściwego punktu w tekście.

- `PG_locked` - na ramce trwa operacja dyskowa (p. 8);
- `PG_error` - operacja dyskowa zakończyła się błędem (p. 8);
- `PG_referenced` - ustawiana przy odniesieniu do ramki przez tablicę `page_hash_table` (p. 4);
- `PG_dirty` - nie jest używana;
- `PG_uptodate` - ustawiana po operacji wejścia/wyjścia (p. 8);
- `PG_free_after`, `PG_decr_after`, `PG_swap_unlock_after` - są używane w funkcjach obsługi buforów dyskowych. Są w nich zapisane informacje o funkcjach, które należy wykonać po zakończeniu operacji dyskowej na ramce. I tak: `PG_decr_after` powoduje zmniejszenie licznika `nr_async_pages`, zliczającego asynchroniczne operacje we/wy; `PG_swap_unlock_after` - wywołanie `swap_after_unlock_page(page->offset)`; `PG_free_after` - wywołanie `__free_page(page)`. Flaga `PG_swap_unlock_after` jest ustawiana na czas operacji dyskowych na stronach z ustawioną flagą `PG_swap_cache`. Flagi te są sprawdzane w funkcji `after_unlock_page` z `fs/buffer.c`;
- `PG_DMA` - ustawiana, kiedy ramka może być objęta mechanizmem DMA (p. 3);
- `PS_Slab` - jest ustawiona, wtedy kiedy na ramce jest zaalokowana tafla (ang. *slab*) (p. 6);
- `PG_swap_cache` - oznacza, że strona jest w pamięci podręcznej wymiany *(ang. swap cache)*. Flaga jest ustawiana na czas trwania operacji we/wy na pliku wymiany;
- `PG_skip` - jest używana na procesorach sparc/sparc64 do pomijania niektórych obszarów pamięci;
- `PG_reserved` - oznacza, że do danej ramki nie wolno się odnosić, fizycznie może jej nawet nie być.

11.1 Operacje na flagach

Flagi to makra bezparametrowe. Wartość makra to numer bitu na którym jest zapisana wartość. Oto przykładowa definicja flagi:

```
#define PG_swap_cache           10
```

Do operacji na flagach zdefiniowane są makra o nazwach odpowiadającym nazwom flag i przyjmujących jeden parametr - wskaźnik do struktury `page`. Przykład:

```
#define PageSwapCache(page)
          (test_bit(PG_swap_cache, &(page)->flags))
#define PageSetSwapCache(page)
          (set_bit(PG_swap_cache, &(page)->flags))
#define PageTestandSetSwapCache(page)
          (test_and_set_bit(PG_swap_cache, &(page)->flags))
#define PageClearSwapCache(page)
```

```
          (clear_bit(PG_swap_cache, &(page)->flags))
#define PageTestandClearSwapCache(page)
          (test_and_clear_bit(PG_swap_cache, &(page)->flags))
```

Makra dla innych flag są zdefiniowane w analogiczny sposób, więc ich przytaczanie mija się z celem. Nie wszystkie operacje są wykonywane za pomocą dodatkowych makr, np. do czyszczenia bitu `PG_referenced` używa się konstrukcji `page->flags &=  (1 « PG_referenced)`. Po prostu te makra, które byłyby używane w kodzie tylko kilka razy nie są zdefiniowane. Zmniejsza to lekko przejrzystość kodu, ale przyspiesza kompilację.

12 Zadania

1. W której ramce znajduje się adres wirtualny `0xC000FF00`? Rozmiar ramki to 4 kB, a przestrzeń wirtualna jądra zaczyna się od 3 GB. Jak dla danego adresu wirtualnego wyznaczyć wskaźnik do *cache'u* w którym się znajduje? Patrz tez w p. 6.
2. Opisz jak będzie wyglądać struktura opisująca wolne ramki, jeżeli w systemie jest 16 ramek z czego zajęte są: 0, 3 - 5, 13 - 15. Jakie będą wolne bloki, jak będą wyglądać mapy bitowe `free_area->map`?
3. Ile pamięci można zaoszczędzić, gdy poprawi się przydział pamięci na mapy bitowe `free_-area->map`, tak jak zostało to opisane w p. 3? Załóż, że w systemie jest 128 MB RAM.

13 Rozwiązania zadań

1. Adres jest w `(0xC00FF000 - 0xC0000000) / 0x1000 == 0xF == 15` ramce. Adres *cache'u* to `mem_map[MAP_NR(addr)].next`.
2. Kolejne listy wolnych bloków `free_area[i]`: [0] -> 1 -> 2 -> 12, [1] -> 6, [2] -> 8, pozostałe listy są puste. Koleje mapy bitowe `free_area[i]->.map`: `11000010, 0101, 00, 0`, pozostałe nie istnieją.
3. Przed poprawką największa mapa bitowa ma rozmiar: `(128 * 1024 * 1024) / (1 * 4 * 1024) / 8 == 4096` bajtów, a najmniejsza `(128 * 1024 * 1024) / (512 * 4 * 1024) / 8 == 8` bajtów. Tak więc suma ich rozmiarów to `4096 + 2048 + ...  + 16 + 8` bajtów. Po zmniejszeniu każdej z bitmap o połowę ich rozmiary będą wynosić od `2048` do `4` bajtów, więc różnica to `4096 - 4 == 4092`. Uwzględniając to, że istnieją dwa rodzaje pamięci (DMA i nie-DMA) mamy ostatecznie: `2 * 4092 == 8184` bajty.

Bibliografia

[Rusling 99] David A. Rusling: *The Linux Kernel.* Cześć Linux Documentation Project (`http://www.linuxdoc.org`), wersja 0.8-3, 1996-1999

[Linux Podręcznik] Studenci Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego: *Linux Podręcznik.* Opracowana na zajęciach z Systemów Operacyjnych, 1999-2000, (`http://rainbow.mimuw.edu.pl/SO`)

[Linux Prezentacje] Studenci Wydziału Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego: *Projekt Linux Prezentacje.* Opracowane na zajęciach z Systemów Operacyjnych, 1998-1999, (`http://rainbow.mimuw.edu.pl/SO`)