Rozdział 1

Zwalnianie ramek

Dzieło Szymona Siwka, stworzone w dwudziestym wieku w Babicach, opisuje mechanizm zwalniania ramek (pamięci fizycznej) w systemie Linux (wersja jądra 2.2.17)

1.1 Wstęp

W linuksie pamięć fizyczna podzielona jest na ramki. Z uwagi na to, że do pewnych celów (na przykład na bufory) potrzebne są spójne bloki pamięci fizycznej, konieczne jest sensowne zarządzanie przydziałem ramek. Sensowne - czyli szybkie i minimalizujące fragmentację pamięci. Osiągnięte jest to dzięki używaniu algorytmu bliźniaków. Zgodnie z tą metodą przydzielane są bloki składające się z 2^k ramek. Jeśli nie ma bloku odpowiedniej wielkości, to dzielimy większy na dwie połowy (to są właśnie te bliźniaki), aż uzyskamy blok właściwej wielkości. Dla każdego bloku jednoznacznie określony jest jego bliźniaczy blok. Zwalnianie bloku wygląda tak:

Algorytm 1 zwalnianie bloku ramek

dopóki wielkość(blok) $<$ maksymalna_wielkość_bloku **wykonuj**
    jeśli bliźniak(blok)$\in$lista_wolnych_bloków **to**
        lista_wolnych_bloków:= lista_wolnych_bloków$\setminus\{$bliźniak(blok)$\}$
        blok:=blok $\cup$ bliźniak(blok)
    koniec jeśli
koniec dopóki
lista_wolnych_bloków:=lista_wolnych_bloków$\setminus\{$bliźniak(blok)$\}$

Tak wygląda idea, a teraz omówię jak to jest realizowane w omawianej wersji jądra.

1.2 Opis używanych struktur danych

Stan ramki opisany jest przez strukturę typu `page`:

```
typedef struct page {
    struct page *next;
```

```
    struct page *prev;
    struct inode *inode;
    unsigned long offset;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct wait_queue *wait;
    struct page **pprev_hash;
    struct buffer_head * buffers;
};
```

Opiszę pola używane podczas zwalniania ramek:

- `count` - ilość procesów "widzących" daną ramkę.
- `flags` - bity tego pola opisują różne własności ramki (w opisywanej wersji jądra używanych jest 13 bitów). Znaczenie poszczególnych bitów będę objaśniał na bieżąco.
- `next, prev` - wskazują pierwszą ramkę następnego (poprzedniego) wolnego bloku ramek identycznej wielkości jak blok, do którego należy opisywana ramka.

Informacja o wolnych blokach zawarta jest w tablicy `free_area`. Tablica ta jest zdefiniowana następująco:

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

Typy pamięci są dwa:

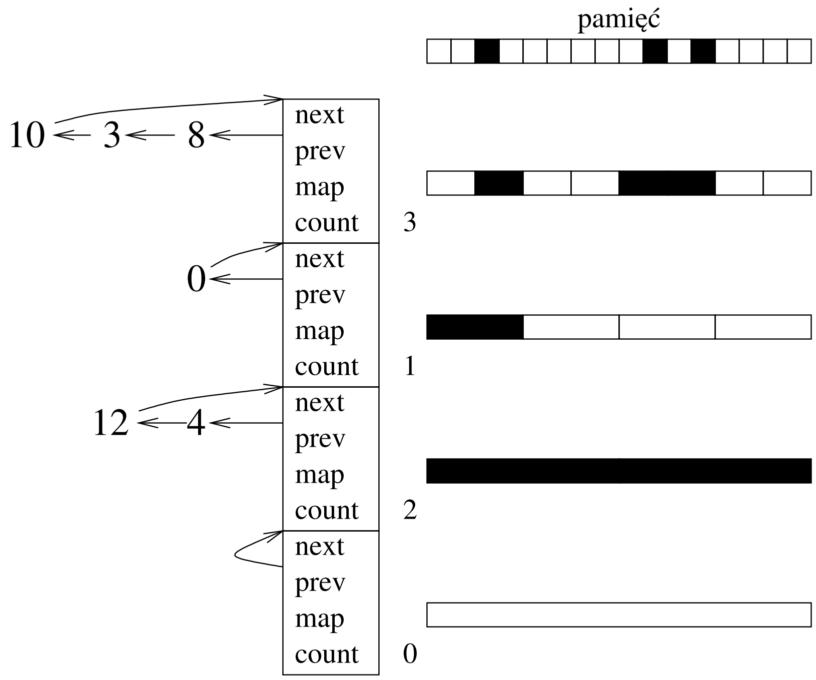
- dostępny dla operacji DMA
- nie dostępny

NR_MEM_LISTS dla większości architektur jest równe 10, dla jednego z procesorów musi być 12, gdyż potrzebuje dużych bloków pamięci na bufory. Oto definicja elementów tablicy:

```
struct free_area_struct {
    struct page *next;
    struct page *prev;
    unsigned int *map;
    unsigned long count;
};
```

Pierwsze pola rekordów `free_area_struct` i `page` pokrywają się i nie jest ot przypadkowe - zmienne typu `free_area_struct` "podszywają się" pod zmienne typu `page`.

i-ty element tablicy jest dwukierunkową listą cykliczną wolnych bloków pamięci wielkości 2^i ramek, tzn. `next` wskazuje na pierwszą ramkę z pierwszego bloku z tej listy, a `prev` na pierwszą ramkę ostatniego bloku. Powstaje pytanie, co się dzieje gdy nie ma bloków dla pewnego i; wówczas stosuje się trik taki, że `next` i `prev` wskazują na strukturę, której są polami, czyli i-ty element tablicy `free_area`. Pole `count` mówi ile jest wolnych bloków odpowiedniej wielkości. Pole `map` jest mapą bitową; j-ty bit równy jest 1 wtedy, gdy wolny jest dokładnie jeden z dwóch bloków wielkości 2^j, o numerach $2j$ i $2j+1$. Przykład mamy na rysunku 1.1. Dla uproszczenia zmieniłem listę dwukierunkową na jednokierunkową. Na rysunku kwadrat odpowiada jednej ramce; kwadrat zaciemniony, to ramka zajęta. W mapach bitowych `map` pole zaczernione oznacza bit ustawiony na jeden.



Rysunek 1.1: Przykładowa postać tablicy `free_area`

1.3 Opis funkcji

Do zwalniania ramek przewidziano następującą funkcję:

```
void free_pages(unsigned long addr, unsigned long order)
{
    unsigned long map_nr = MAP_NR(addr);

    if (map_nr < max_mapnr)
        __free_pages(mem_map + map_nr, order);
}
```

Zwalniany jest blok kolejnych 2^{order} ramek, zawierający fizyczny adres odpowiadający adresowi `addr`, który to adres jest adresem z wirtualnej pamięci jądra. Do zwalniania pojedynczej ramki mamy zdefiniowane makro `free_page(addr)`.

MAP_NR to makro liczące, w której ramce jest dany adres. W instrukcji `if` sprawdzane jest, czy zwalniana ramka jest w pamięci fizycznej[1]. Jeśli tak, to wywoływana jest funkcja prymitywniejsza. Jej pierwszy argument jest opisem zwalnianej ramki. Zmienna `mem_map` jest tablicą (elementy są typu `page`) opisującą wszystkie ramki.

Oto kod funkcji `__free_pages` :

```
void __free_pages(struct page *page, unsigned long order)
{
    if (!PageReserved(page) && atomic_dec_and_test(&page->count)) {
        if (PageSwapCache(page))
            panic ("Freeing swap cache page");
        page->flags &= ~(1 << PG_referenced);
        free_pages_ok(page - mem_map, order, PageDMA(page) ? 1 : 0);
        return;
    }
}
```

Najpierw sprawdzamy czy ramka nie jest zarezerwowana[2] (`PageReserved`) oraz czy inny proces nie korzysta z ramki (`page->count`). Następnie staramy się uniknąć zwalniania ramek należących do cache pamięci wymiany (jak widać próba zwolnienia takiej ramki jest oznaką poważnego błędu). Jeśli testy wypadły pomyślnie, zerujemy flagę `PG_referenced` (korzystamy z niej przy sprawdzaniu, czy możemy stronę wymieść z pamięci fizycznej) i wywołujemy najbardziej podstawową funkcję zwalniania ramek `free_pages_ok` z dodatkowym parametrem mówiącym, czy na zwalnianej ramce może być wykonywana operacja DMA.

I nadszedł czas na gwóźdź programu - funkcja `free_pages_ok` implementująca algorytm bliźniaków:

```
static inline void free_pages_ok(unsigned long map_nr, unsigned long order, unsigned type)
{
    struct free_area_struct *area = free_area[type] + order;
    unsigned long index = map_nr >> (1 + order);
    unsigned long mask = (~0UL) << order;
    unsigned long flags;

    spin_lock_irqsave(&page_alloc_lock, flags);

#define list(x) (mem_map+(x))

    map_nr &= mask;
    nr_free_pages -= mask;
    while (mask + (1 << (NR_MEM_LISTS-1))) {
```

[1] warunek nie będzie spełniony, jeśli podamy adres w pamięci karty graficznej

[2] to znaczy, że do tej ramki nie mogą następować żadne odwołania

```
        if (!test_and_change_bit(index, area->map))
            break;
        area->count--;
        remove_mem_queue(list(map_nr ^ -mask));
        mask <<= 1;
        area++;
        index >>= 1;
        map_nr &= mask;
    }
    add_mem_queue(area, list(map_nr));

#undef list

    spin_unlock_irqrestore(&page_alloc_lock, flags);
}
```

Mam nadzieję, że powyższy kod jest mniej czytelny od pseudo-kodu podanego na wstępie.

Najpierw blokujemy przerwania zapewniając sobie niepodzielność operacji. Wyrażenie `n & mask` daje nam numer pierwszej ramki z bloku zawierającego n-tą ramkę.

Wyrażenie `n ^ -mask` jest równe numerowi bliźniaka n-tej ramki.

Zmienna `index` jest pozycją badanej pary bliźniaków w tablicy `map`.

Wolny blok wstawiamy jako pierwszy element list (zatem w czasie stałym), usuwanie bloku z listy jest jeszcze łatwiejsze.

1.4 Przykład

Przykład będzie odnosił się do sytuacji pokazanej na rysunku 1.1. Zwalniamy blok wielkości 1 zawierający ramkę o numerze 2.

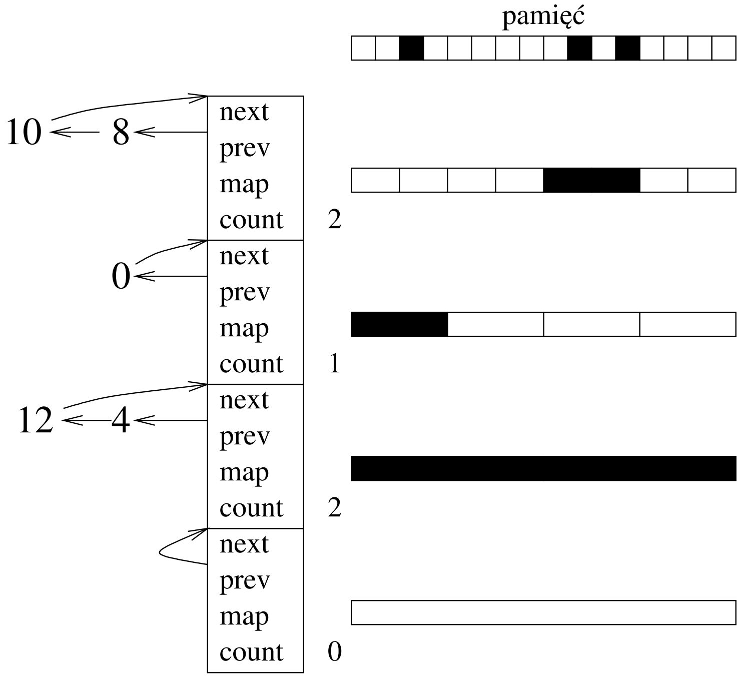
Sytuacje po kolejnych przebiegach pętli przedstawiona jest na rysunkach 1.2, 1.3, 1.4, 1.5.

1.5 Źródła

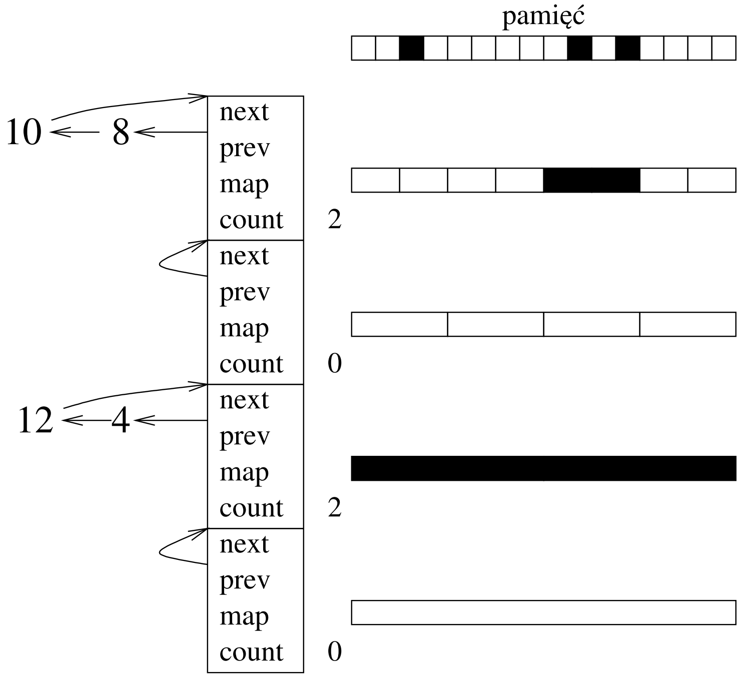
Podstawowym plikiem źródłowym jest `mm/page_alloc.c`.

Struktura `page` zdefiniowana jest w pliku `include/linux/mm.h`

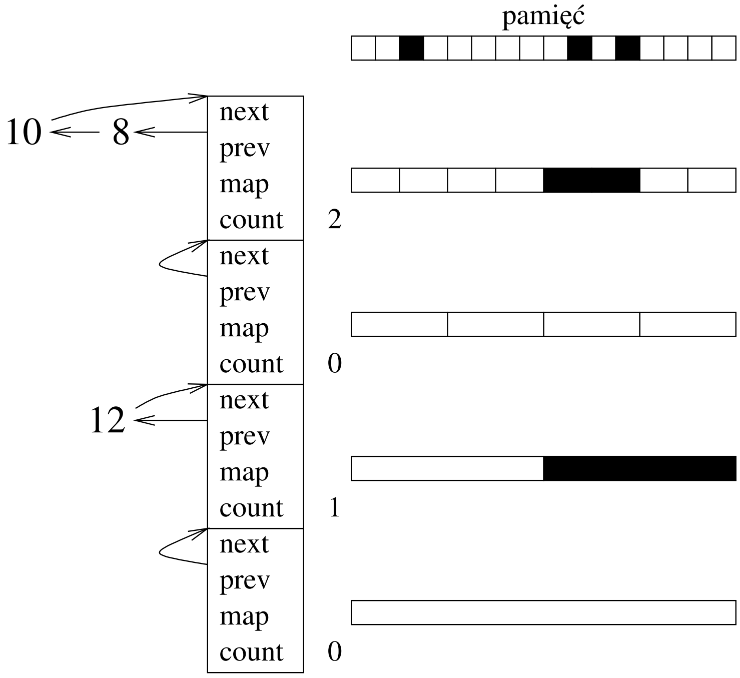
Istotne są również pliki (zależne od architektury) `include/asm/page.h` oraz `include/asm/page_offset.h`.



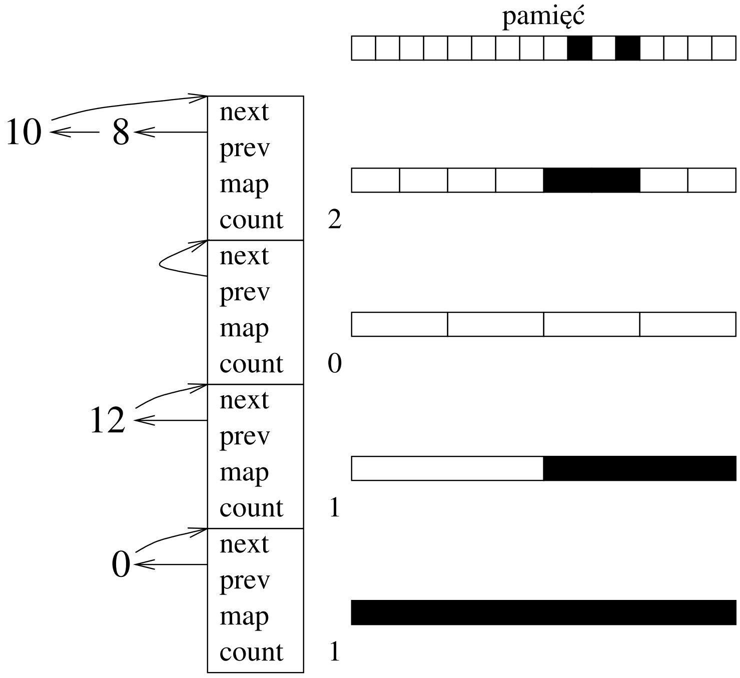
Rysunek 1.2: Po pierwszym przebiegu pętli



Rysunek 1.3: Po drugim przebiegu pętli



Rysunek 1.4: Po trzecim przebiegu pętli



Rysunek 1.5: Ostateczny efekt