Rozdział 1

Algorytm przydziału stron *get_free_pages()*

Autorzy:
Jacek Fiok
Agata Strojnowska

Rozdział ten zawiera opis algorytmu przydziału spójnych bloków pamięci wielkości 2^i stron, wykorzystywany przez funkcje typu *alloc*.

System Linux realizuje algorytm Buddy (bliźniaków). Jest to w miarę skuteczna metoda utrzymania na niskim poziomie stopnia fragmentacji pamięci. Główną zaletą algorytmu jest jego prostota i wykorzystywanie naturalnych metod mnożenia i dzielenia przez 2 [1]. Pozwala to szybko dzielić i łączyć wolne bloki pamięci.

[1] mowa oczywiście o przesuwaniu bitów.

1.0.1 Używane struktury danych

Informacje o stronach w pamięci przechowuje struktura *page*. Początek tej struktury musi być taki, jak pokazano - jest to wykorzystywane przy konwersji wskaźników.

```
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;
       struct wait_queue *wait;
       struct page **pprev_hash;
       struct buffer_head * buffers;
} mem_map_t;

/* Page flag bit values */
#define PG_locked                0
#define PG_error                 1
#define PG_referenced            2
#define PG_dirty                 3
#define PG_uptodate              4
#define PG_free_after            5
#define PG_decr_after            6
#define PG_swap_unlock_after     7
#define PG_DMA                   8
#define PG_Slab                  9
#define PG_swap_cache           10
#define PG_skip                 11
#define PG_reserved             31
```

Struktura *free_area_struct* przechowuje informacje o wolnych blokach pamięci jednego *typu* i jednakowej *wielkości*. Pełni funkcję strażnika listy tych bloków.

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

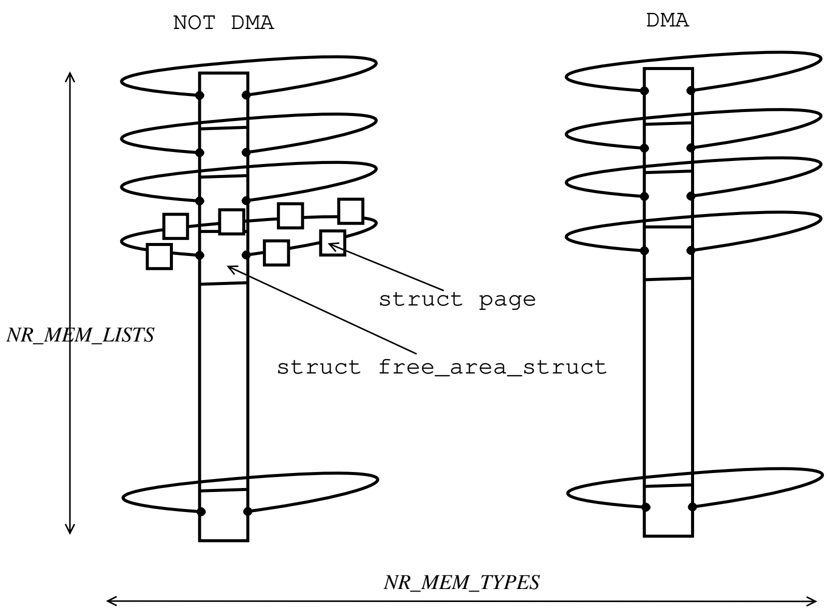
free_area jest tablicą cyklicznych dwukierunkowych list ze strażnikiem typu *free_area_struct* i elementami typu *page* (czyli *mem_map_t*).

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

- NR_MEM_LISTS wyznacza maksymalną wielkość bloku, który można przydzielić ($2^{NR\_MEM\_LISTS-1}$).
- NR_MEM_TYPES oznacza:
  0 — zwykłe bloki pamięci;
  1 — bloki pamięci obsługujące transmisje DMA.

Jak widać w poniższej deklaracji, Linux definiuje stałą NR_MEM_LISTS jednakowo dla wszystkich architektur poza AP-1000. Dodatkowo warto zauważyć, że nastąpiła tu zmiana w implementacji w stosunku do poprzednich wersji systemu - bloki DMA traktowane są całkowicie niezależnie.

```
#if CONFIG_AP1000
/* the AP+ needs to allocate 8MB contiguous, aligned chunks of ram
   for the ring buffers */
#define NR_MEM_LISTS 12
#else
#define NR_MEM_LISTS 10
#endif
#define NR_MEM_TYPES 2          /* GFP_DMA vs not for now. */
```



Rysunek 1.1: Schemat poglądowy używanych struktur danych.

Funkcja *get_free_pages()* używa kilku ciekawych makr:

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

#define MARK_USED(index, order, area) \
        change_bit((index) >> (1+(order)), (area)->map)
```

Pozostałych, dłuższych makr nie będziemy cytować tu w całości, poprzestając na prezentacji realizowanego przez nie algorytmu.

1.0.2 Zajmowanie bloków pamięci

Cała strategia przydziału zawarta jest w funkcji __get_free_pages() z pliku mm/page_alloc.c (uwaga: zmiana w stosunku do wersji jądra 2.2.12 - brak pliku .h).

```
unsigned long __get_free_pages(int gfp_mask, unsigned long order)
```

Funkcja stara się zająć blok wielkości 2^{order} ramek. Jeśli uda sie znaleźć taki blok to zwraca jego adres, wpp. zwraca 0. Utrzymuje również strukturę free_area, co umożliwia efektywne znajdowanie nowych bloków pamięci do zajęcia oraz łączenie zwalnianych.

Używane flagi w gfp_mask:
__GFP_HIGH — dozwolone zejście poniżej wysokiego poziomu wolnych ramek
__GFP_MED — dozwolone zejście poniżej średniego poziomu wolnych ramek
__GFP_DMA — bloki pamięci obsługującej transmisje DMA
__GFP_WAIT — tu używana jedynie do debugowania.

Algorytm 1 __get_free_pages

Dane wejściowe: gfp_mask — dodatkowe wymagania, m.in.: czy pamięć ma być typu DMA; order — wielkość bloku ($= 2^{order}$)

Wynik: adres przydzielonego bloku lub 0 w przypadku niepowodzenia

jeśli order >= NR_MEM_LISTS **to**
    zwróć 0, żądany blok jest za duży
koniec jeśli
sprawdź, czy można w tej chwili alokować pamięć (np. czy nie zostały przekroczone limity MIN/LOW/HIGH_MEMORY); być może trzeba będzie poczekać na odpowiedniejszą chwilę; jeśli kompletna katastrofa to zwróć 0. Dokładniejszy opis poniżej.
ustaw lock
znajdź najmniejszy wolny blok pamięci nie mniejszy od żądanego (przeszukaj strukturę free_area[typ][i] dla i=order..NUM_LIST-1)
jeśli nie ma żadnego bloku o rozmiarze $2^k >= 2^{order}$ **to**
    zwróć 0
koniec jeśli
B := odnaleziony blok
usuń B z free_area[typ][k]
zmniejsz licznik dostępnych stron (*nr_free_pages*)
zamień na przeciwny bit bloku B w free_area[typ][k]->map
dopóki B jest za duży (k > order) **wykonuj**
    podziel B na pół (dwóch bliźniaków)
    jedną z połówek wstaw do kolejki free_area[typ][k-1]
    zamień na przeciwny bit wstawianego bloku we free_area[typ][k-1]->map
    B := druga połowa
koniec dopóki
w page_struct bloku B zaznacz, ze strona jest używana
zdejmij lock
zwróć adres bloku B

Algorytm 2 __get_free_pages: pre-paging

```
jeśli jest ustawiona flaga PF_MEMALLOC to
   przejdź do alokowania pamięci
koniec jeśli
jeśli free_before_allocate > 0 to
   ustaw flagę PF_MEMALLOC
   spróbuj zwolnić pamięć (try_to_free_pages(gfp_mask))
   zdejmij flagę PF_MEMALLOC
koniec jeśli
jeśli liczba stron > średni limit to
   przejdź do alokowania pamięci
koniec jeśli
obudź demona kswapd
jeśli liczba stron > minimalny limit to
   przejdź do alokowania pamięci
koniec jeśli
ustaw flagę PF_MEMALLOC
zwiększ free_before_allocate
spróbuj zwolnić pamięć (try_to_free_pages(gfp_mask)) i zapamiętaj wynik
(na zmiennej freed)
zmniejsz free_before_allocate
zdejmij flagę PF_MEMALLOC
jeśli w tym czasie ktoś inny zwolnił dużo pamięci: liczba stron > maksymalny
limit to
   przejdź do alokowania pamięci
koniec jeśli
jeśli nie udało się nic zwolnić i gfp_mask nie miała ustawionych ani
__GFP_MED ani __GFP_HIGH to
   zwróć 0, nie ma czego alokować
w przeciwnym przypadku
   przejdź do alokowania pamięci
koniec jeśli
```

Jeśli zażądano bloku pamięci DMA (ustawiony bit __GFP_DMA w gfp_mask), to funkcja przeszuka tylko bloki typu DMA (struktura free_pages[1]). W przeciwnym przypadku wyszukiwanie rozpocznie się od zwyklych bloków (free_pages[0]), a w przypadku niepowodzenia przeszukane zostaną rownież bloki DMA.

Istnienie „mapy bitowej” (pole *map* we *free_area_struct*) tłumaczy się poprzez wykorzystywanie jej przy zwalnianiu pamięci (funkcje *free_pages()* i *free_pages_ok()*). Jest to temat oddzielnego referatu.

1.0.3 Uwagi i ciekawostki

1. W źródłach znaleźć możemy m.in. takie oto komentarze:
   ”stupid algorithm”,
   ”somewhat less stupid version of algorithm” (mmap.c),
   ”You really aren't expected to understand this” (page_alloc.c).

2. Ciekawe rozwiązania:
   mask += mask (page_alloc.c, linia 335).

3. Warto przejrzeć wynik funkcji show_free_areas(). Wynik m.in. tej funkcji można wywołać poprzez mechanizm magicSysRq: będąc na konsoli tekstowej należy nacisnąć Alt+PrintScreen+M.
   Opcja magicSysRq znajduje się w sekcji „kernel hacking” konfiguracji jądra.
   Na marginesie: wypróbuj Alt+PrintScreen+[inne literki].

4. Skąd wiemy, ile pamięci trzeba zwolnić?
   – jest to parametr dla *free_pages()* –> *__free_pages()*, zatem wniosek: odpowiedzialne za to są mechanizmy wyższego poziomu.

5. W systemie linux cały czas działa „dobry duszek” [2] – byt, który dba o pewną pulę wolnych ramek, widziany jako proces **kswapd**. Jest to zagadnienie pokrewne dla poruszonych w niniejszej pracy mechanizmów *pre-paging*.

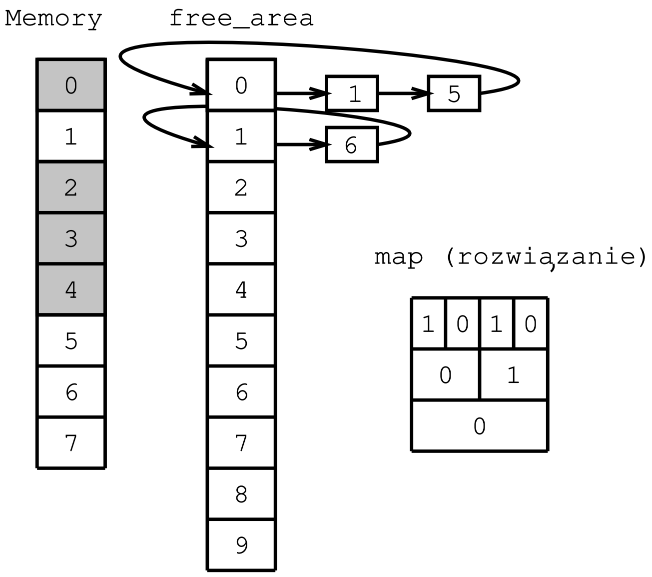
1.1 Ćwiczenia

1. Mając dane: obraz pamięci oraz strukturę free_area - ustal mapę bitową (patrz rys. 1.2).

2. Zaalokuj po kolei bloki wielkości 4, 1 i 2 ramek dla stanu pamięci podanego na rys. 1.3.

3. Czy ma znaczenie dla samego algorytmu, czy pamięć jest alokowana od góry, czy od dołu?

[2]zapożyczenie ze slajdów z poprzednich lat.



Rysunek 1.2: Przykładowy stan pamięci.

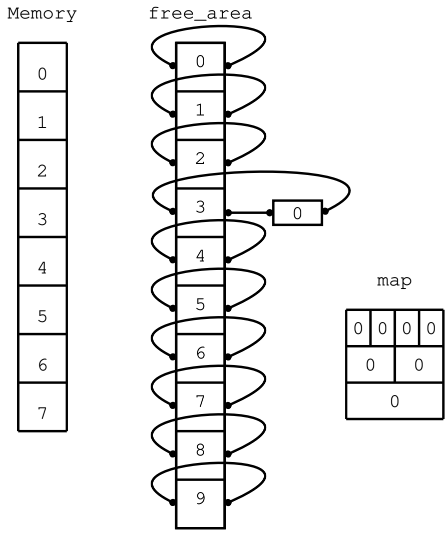
Ciekawostka: jak to w życiu bywa, między rozważaniami teoretycznymi a praktyką występują pewne drobne rozbieżności. I tak np. w popularnej architekturze płyt głównych Intela dla procesorów Pentium **i430TX** pamięć podręczna drugiego poziomu znajdująca się w płycie potrafi „objąć" jedynie **pierwsze** 64MB pamięci głównej. Informacje na ten temat, choć niezbyt szczegółowe, znaleźć możemy w odpowiednim pliku dokumentacji jądra (Documentation/memory.txt).

4. Czy zajmując bloki można doprowadzić do dowolnego układu zajętych i wolnych ramek w pamięci? A jeśli jeszcze zwalniamy? Jak bardzo można sfragmentować pamięć w najgorszym przypadku?

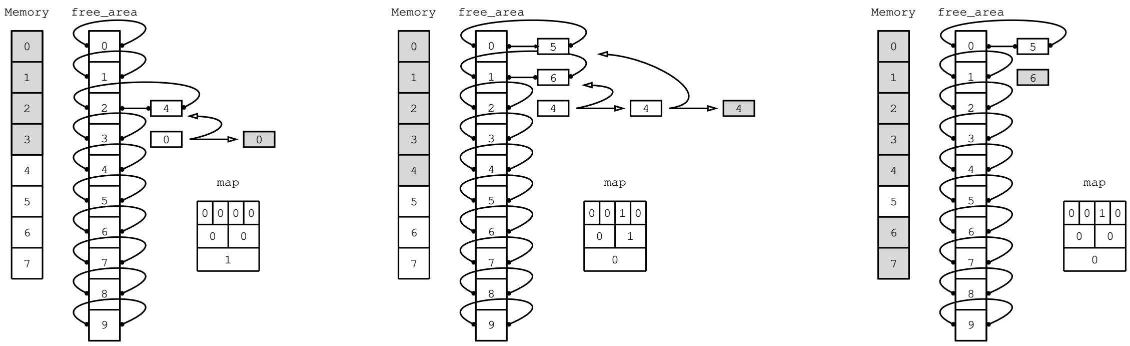
1.2 Źródła wiedzy

Źródła jądra systemu: **[/usr/src/linux/]mm/page-alloc.c**.
Dodatkowo pliki **mm/mmap.c** i **include/linux/mm.h**.

Przygotowując tę pracę korzystaliśmy również z programu indeksującego źródła systemu: **lxr.linux.no/ident/** oraz dostępnych materiałów przygotowanych przez studentów MIM UW z poprzednich lat (**rainbow.mimuw.edu.pl/SO/**).



Rysunek 1.3: Stan pamięci przed przydzieleniem nowego bloku.



Rysunek 1.4: Przydzielono bloki wielkości 4, 1, 2.