Rozdział 1

Przydział pamięci dla jądra linuksa w wersji 2.2.17 (algorytm kmalloc)

BARTŁOMIEJ BÓBSKI & PRZEMYSŁAW ŁUPIŃSKI

1.1 Ogólna charakterystyka (idea)

1.1.1 Wstęp

Przydział pamięci dla jądra jest zupełnie inny w jądrze 2.2.17 niż w poprzednich jego wersjach o numerach 2.0.x i poprzedzających je.

Alokacja i zwalnianie obiektów (tak będę nazywał pewne spójne obszary pamięci fizycznej) są zarówno najważniejszymi jak i najczęściej wykonywanymi operacjami w jądrze. Z tego powodu jest potrzebny szybki mechanizm ich wykonywania.

Podstawową jednostką, organizującą ten proces jest alokator pamięci z buforowaniem obiektów (ang. *Object-Caching Kernel Memory Allocator*).

Object-Caching - jest techniką zarządzania stworzonymi obiektami, **Allocator** - jest mechanizmem korzystającym z tej techniki.

Raz stworzony obiekt pomiędzy kolejnymi użyciami nie jest niszczony (chyba że zabraknie pamięci fizycznej) - nie trzeba za każdym razem go inicjalizować i niszczyć. W wielu przypadkach koszt tworzenia takiego obiektu (tu także odpowiednich struktur danych tu opisujących dany obiekt) jest dużo większy niż koszt jego przydziału dla jądra.

Alokator pracuje w trybie wielowątkowym (przydziela pamięć dla współbieżnych/równoległych procesów jądra), dlatego zastosowane są w nim semafory synchronizujące operacje na strukturach utrzymujących stan pamięci i zarządzających nią.

1.1.2 Wygląd struktur danych opisujących stan pamięci

Alokator jest narzędziem do zarządzania skonstruowanymi obiektami i strukturami danych opisującymi je. Wielkość obiektów jakie można przydzielać jest z góry ustalona tzn. często przydzielana pamięć nie odpowiada dokładnie żądaniu - jest od niej większa (fragmentacja wewnętrzna).

- Cache

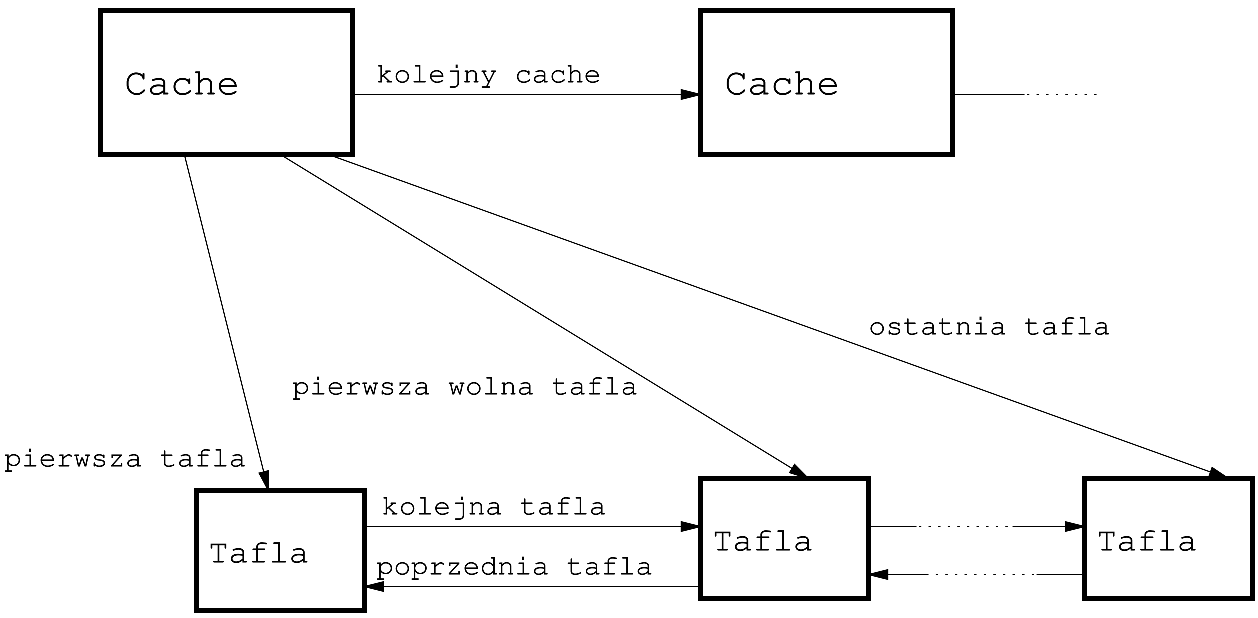
  Logiczna struktura zawierająca stan wolnej i przydzielonej pamięci. Cache'e są tworzone przy starcie systemu. Każdy z nich odpowiada za obiekty o konkretnej wielkości. Dostępne wielkości obiektów to 32 B, 64B, .. , 131072 B - tylko takie obiekty można przydzielać. Obiekty są inicjowane tylko raz, przy konstrukcji. Jednak sam cache nie wskazuje na fizyczny obiekt, cache'e zawierają listę tafli (ang. *slab*). Tafle na liście są uporządkowane w kolejności: zapełnione (wszystkie opisywane przez nie obiekty są w danym momencie używane), częściowo zapełnione, puste.

- Tafle (slab)

  Fizycznie jest to jedna lub kilka stron pamięci logicznie podzielonych na obiekty tego samego rozmiaru. W zależności od rozmiaru obiektów tafle pokazują na konkretne obiekty w pamięci fizycznej lub na bufory (opisane dokładnie w podpunkcie organizacja buforów).

- bufor (opcjonalnie)

  Wskazują na konkretny obiekt w pamięci fizycznej. Opcjonalnie, ponieważ w zależności od wielkości obiektów mogą istnieć lub nie (opis w podpunkcie organizacja buforów).



Rysunek 1.1: Struktury danych

Pozostała część struktur danych opisujących obiekty jest na rysunku w podpunkcie organizacja buforów.

1.1.3 Schemat działania (przydzielania i zwalniania obiektów)

Alokacja obiektu:

```
if(są obiekty w cache'u)
   weź jeden (nie potrzebna konstrukcja)
else
   alokuj pamięć;
   skonstruuj obiekty;
```

Ważne jest że w klauzuli else nie jest tworzony jeden obiekt tylko cała tafla obiektów (fragmentacja zewnętrzna) jest to powód wyodrębnienia tafli pomiędzy obiektami, a cache'm. Tu jednocześnie może się odbywać przydział obiektów z różnych cache'y (tzn. obiektów o różnej wielkości).
Zwalnianie obiektu:

```
zwróć go do cache'u (nie potrzeba niszczenia)
```

Zadanie zwrotu pamięci:

```
Weź tafle z samymi wolnymi obiektami;
zniszcz ją; (odłącz daną tafle od reszty cache'a)
zwolnij pamięć fizyczną zawieraną przez nią; (przez obiekty które opisywała)
```

Tu także jest niszczona cała tafla obiektów, a nie konkretny, pojedynczy obiekt.

1.1.4 Zalety

Szybkość działania (uniknięcie częstej inicjalizacji i destrukcji obiektów), algorytm przydziału obiektu rzadko tworzy nowe obiekty (ponieważ na raz jest tworzona cała tafla obiektów).

1.2 Szczegółowa charakterystyka (implementacja)

1.2.1 Szczegółowy opis struktur danych i funkcji operujących na nich

Struktury :

- kmem_cache_s

  Struktura implementuje stan cache'u.

```
struct kmem_cache_s {
  kmem_slab_t   *c_freep    - pierwsza wolna tafla, tafla w której są jeszcze wolne
                              obiekty;
  unsigned long  c_flags    - stałe flagi (informują gdzie znajdują się bufory
                              opisujące obiekty);
  unsigned long  c_offset   - omówiona w podpunkcie (organizacja buforów);
  unsigned long  c_num      - ilość obiektów w tafli;
  unsigned long  c_magic    - do wykrywania czy są jeszcze jakieś tafle, czy należy
                              utworzyć nową;
  kmem_slab_t   *c_firstp   - wskaźnik do pierwszej tafli;
  kmem_slab_t   *c_lastp    - wskaźnik do ostatniej tafli;
  spinlock_t     c_spinlock - semafor;
```

```
  unsigned long  c_growing  - informuje czy można w danym momencie zwalniać jakąś
                              tafle z cache'a;
  unsigned long  c_dflags   - flagi ustawiane dynamicznie;
  unsigned long  c_gfporder - 2^c_gfporder  - liczba stron przypadających na jedną taflę;
  void (*c_ctor)(void *, kmem_cache_t *, unsigned long) - konstruktor;
  void (*c_dtor)(void *, kmem_cache_t *, unsigned long) - destruktor;
  unsigned long  c_align
  size_t         c_colour
  size_t         c_colour_next
  kmem_cache_t  *c_index_cachep - powyższe cztery wartości pozwalają w odpowiednim
                                  miejscu w pamięci fizycznej umieścić taflę;
  const char    *c_name         - nazwa cache'a;
  struct kmem_cache_s  *c_nextp - wskaźnik do następnego cache'a;

#if     SLAB_STATS  jeżeli chcemy utrzymywać statystyki dotyczące danego cache'a
  unsigned       long c_num_active - liczba aktywnych tafli;
  unsigned       c_num_allocations - liczba posiadanych tafli;
  unsigned       long c__high_mark - maksymalna liczba aktywnych tafli jaka była
                                     do tej pory;
#endif  /* SLAB_STATS */
};
```

- kmem_cache_t

```
typedef struct kmem_cache_s kmem_cache_t;
```

- kmem_slab_s

Strutura zarządzania taflą. Jeśli alokator zarząda powiększenia cache'a jest dodawana cała nowa tafla, jeśli system zarząda pamięci jest zwracana cała tafla (nie pojedyncze bufory). Zawiera jedną lub więcej stron pamięci podzieloną na takiej samej wielkości bufory (obiekty). Na podstawie jej atrybutów wiemy czy opisuje jakieś wolne obiekty, a jeśli tak to gdzie one się znajdują. Może zawierać informacje dotyczące już konkretnych obiektów w pamięci fizycznej lub informacje na temat buforów je pisujących (podpunkt organizacja buforów)

```
typedef struct kmem_slab_s {
  struct kmem_bufctl_s  *s_freep   - pierwszy wolny bufor w danej tafli;
  struct kmem_bufctl_s   s_index   - lista buforów;
  unsigned long          s_magic   - czy dana tafla ma jakieś wolne obiekty;
  unsigned long          s_inuse   - liczba zajętych buforów (obiektów);
  struct kmem_slab_s    *s_nextp   - następna tafla;
  struct kmem_slab_s    *s_prevp   - poprzednia tafla;
  void                  *s_mem     - adres pierwszego obiektu w pamięci fizycznej
                                     dokładnie omówiona w podpunkcie organizacja buforów;
  unsigned long          s_offset:SLAB_OFFSET_BITS
                         s_dma:1;
} kmem_slab_t;
```

- kmem_bufctl_s

  Struktura implementująca bufory. Jest używana do łączenia obiektów wewnątrz tafli, identyfikuje do jakiej tafli należy dany przez nią opisywany obiekt oraz posiada adres dołączonego obiektu.

```
typedef struct kmem_bufctl_s {
        union {
                struct kmem_bufctl_s   *buf_nextp - następny bufor;
                kmem_slab_t            *buf_slabp - tafla dla bufora;
                void *                  buf_objp  - adres obiektu;
        } u;
} kmem_bufctl_t;
```

- cache_sizes

  Struktura zawierająca opis i pojedynczego cache'a. Dzięki jej jądro wie jaki cache odpowiada jakiej wielkości obiektom i na tej podstawie wybiera cache do alokacji obiektu.

```
typedef struct cache_sizes {
        size_t           cs_size       - rozmiar danego cache'a
        kmem_cache_t    *cs_cachep     - cache który opisuje
} cache_sizes_t;
```

- cache_sizes_t

  Struktura zawierająca opis wszystkich cache'y, przy starcie systemu jest wypełniana przez funkcję `kmem_cache_sizes_init`. Dla danego rozmiaru są tworzone cache'e ale fizyczna alokacja obiektów (tafli obiektów) następuje dopiero w momencie zgłoszenia zapotrzebowania przez jądro na pierwszy obiekt.

```
static cache_sizes_t cache_sizes{[]}{[]} = {
#if     PAGE_SIZE == 4096  - dyrektywa dla kompilatora
        {  32,          NULL},
#endif
        {  64,          NULL},
        { 128,          NULL},
             ......
        {131072,        NULL},
        {0,             NULL}
};
```

Funkcje: (opisuję tylko ważniejsze funkcje)

- tworzące cache:

  `kmem_cache_create` - parametrami są nazwa, rozmiar, flagi, konstruktor, destruktor.

Funkcja tworzy pojedynczy cache. Zwraca wskaźnik do utworzonego cache'u lub NULL jak wystąpi jakiś błąd. Ustawia atrybuty struktury `kmem_cache_t`.

`kmem_cache_sizes_init` - inicjuje strukturę `kmem_sizes_t` wołając funkcję `kmem_cache_create` dla każdego z rozmiarów cache'y.

- operujące na nim (z tych funkcji korzysta algorytm kmalloc):
  - na obiektach już będących w cache'u:

    `kmem_cache_alloc` - zwraca `__kmem_cache_alloc`.

    `static inline void* __kmem_cache_alloc` - zwraca wskaźnik do wolnego obiektu w żądanym cache'u.

    `static inline kmem_slab_t kmem_cache_serch_dma` - zwraca taflę, która przechowuje obiekty odpowiednie dla DMA.

    `static inline size_t kmem_cache_cal_waste` - zwraca ilość straconych bajtów dla danej tafli.
  - na obiektach, które są aktualnie dokładane do cache'u:

    `static int kmem_cache_grow` - dodaje nową tafle do cache'a (tafla z zaalokowaną pamięcią na obiekty).

    `static inline void* kmem_getpages` - pobiera strony pamięci fizycznej dla tafli.

    `static inline void* kmem_freepages` - zwalnia strony pamięci fizycznej.

    `static inline void kmem_cache_init_objs` - inicjuje obiekty w tafli ustawiając odpowiednie wartości tafli np. `s_index` i `s_freep` w zależności jakiej wielkości obiekty są w tafli (zobacz podpunkt organizacja buforów).

    `static inline kmem_slab_t kmem_cache_slabmgmt` - dla danej tafli alokuje pamięć na końcu przestrzeni pamięci fizycznej którą opisuje.

    `kmem_slab_unlink` - usuwa z list tafli daną tafle.

    `kmem_slab_link_end` - dodaje tafle na koniec listy tafli.

    `kmem_slab_link_free` - dodaje tafle przed pierwszą wolną tafle w liście tafli.

1.2.2 Organizacja buforów

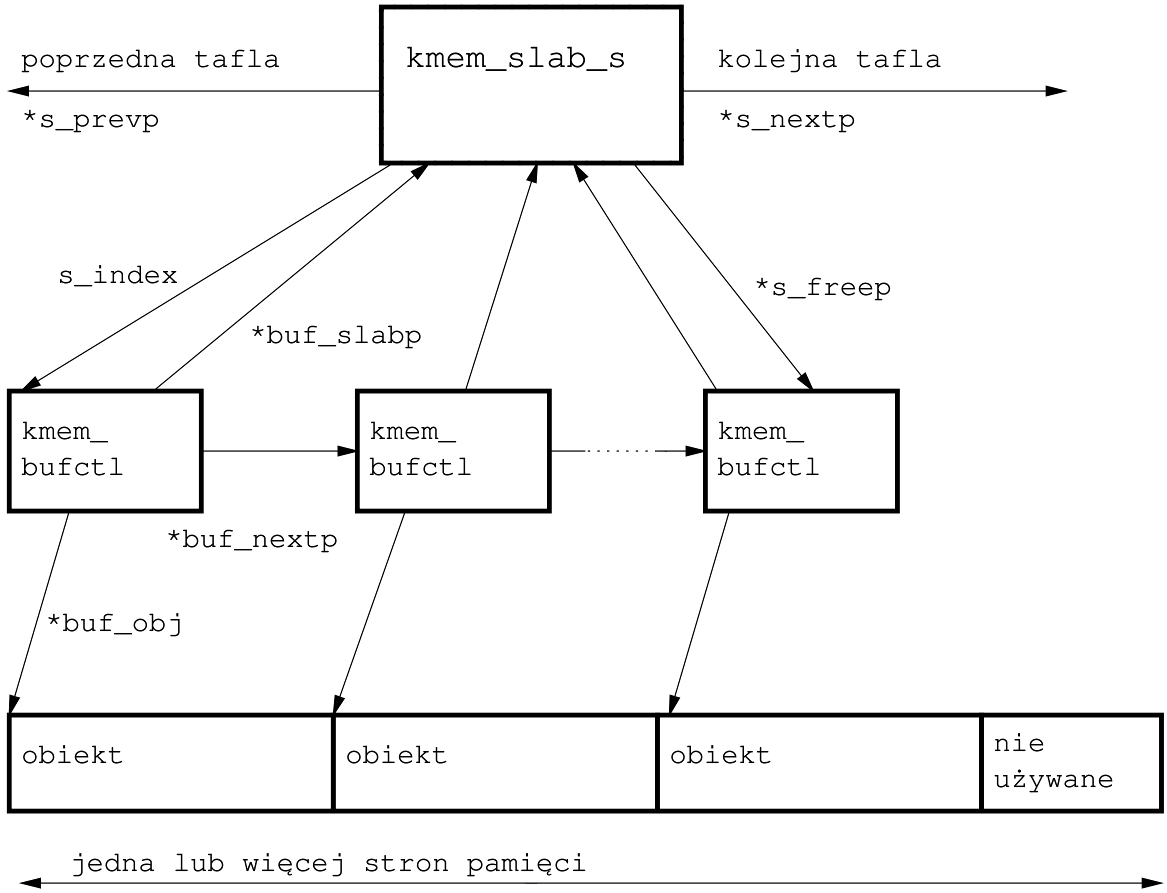
Bufory zostały wprowadzone w celu jednoznacznej identyfikacji obiektów. Dzięki nim mamy łatwy dostęp do konkretnych obiektów. Pomimo tej zalety często staje się nie opłacalne przechowywanie buforów np. gdy rozmiar samej struktury bufora jest podobny do rozmiaru obiektu jaki opisuje. To skłoniło to zastosowania następującego podziału (został on ustalony na podstawie przeprowadzonych testów):

- *Dla obiektów większych od 1/8 strony*

  Każdy z obiektów jest zarządzany przez strukturę `kmem_bufctl_s`

  Wyliczanie adresu obiektu:

  `c_offset` (cache) informuje nas w tym przypadku jaka jest odległość w pamięci pomiędzy strukturą `kmem_bufctl_s`, a obiektem który opisuje. Tu wystarczy zbadać pole `buf_objp` (bufora opisującego dany obiekt) aby dowiedzieć się gdzie znajduje się obiekt w pamięci fizycznej.



Rysunek 1.2: Organizacja buforów większych od 1/8 strony

- *Dla obiektów mniejszych od 1/8 strony*

  Tu nie ma buforów, każdy obiekt jest dostępny z tafli.

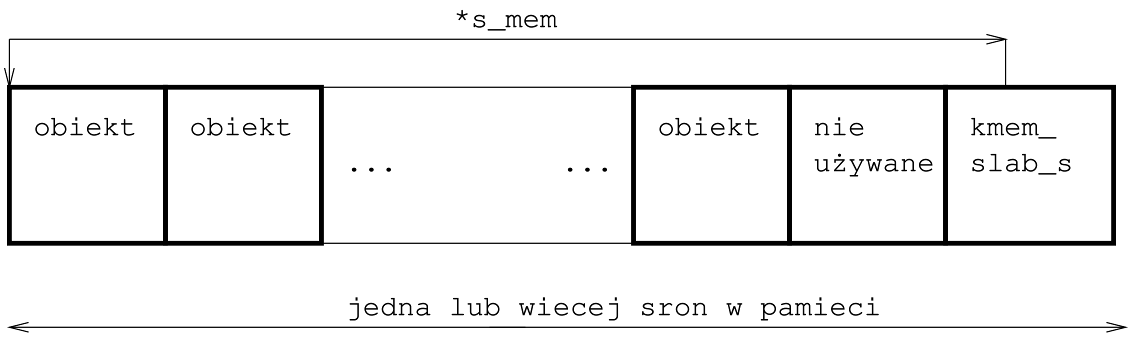
  Wyliczanie adresu obiektu:

  `c_offset` (cache) jest to dystans pomiędzy obiektami. `*s_mem` (tafla) wskazuje na pierwszy obiekt, a następnie aby wyliczyć adres jakiegoś obiektu `*s_mem` + który `*c_offset`, gdzie który = `s_freep - s_index` (tafla). Który - jest pierwszym wolnym obiektem w danej tafli. Pola `s_freep` i `s_index` muszą być modyfikowane przy każdej alokacji i zwalnianiu obiektu.

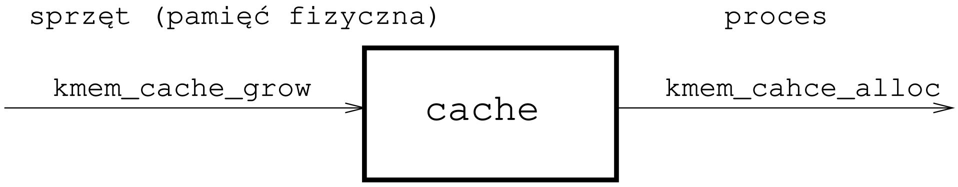
1.2.3 Interfejs

Interfejs możemy podzielić na dwie składowe:

- Interfejs od strony jądra



Rysunek 1.3: Organizacja buforów mniejszych od 1/8 strony



Rysunek 1.4: interfejs

Często alokacja i zwalnianie obiektów będzie się odbywać tylko na tym poziomie tzn. ilość dostępnych (już utworzonych) obiektów będzie w pełni wystarczać. Wymiana obiektów będzie następować tylko pomiędzy jądrem, a cache'm.

- Interfejs od strony sprzętu

  Czasem się jednak zdarzy, że zabraknie obiektów wtedy trzeba utworzyć nowe. Wymiana obiektów pomiędzy pamięcią a cache'm.

Cache "pośredniczy" pomiędzy pamięcią fizyczną, a jądrem.

1.2.4 System

W pliku */proc/slabinfo* są zawarte informacje na temat obecnego stanu wykorzystania cache'y w naszym systemie.

1.2.5 Algorytm KMALLOC

Algorytm ten służy do alokacji pamięci w jądrze. Parametr `size` w funkcji:

```
void {*}
kmalloc(size\_t size, int flags)
```

określa rozmiar potrzebnej pamięci, zaś `flags` to flagi, np.: `SLAB_DMA` - czy dany obszar ma służyć do transmisji DMA, `SLAB_ATOMIC` - do badania czy funkcja kmalloc powiodła się. Przydział pamięci przebiega następująco:

1. W funkcji kmalloc - wyszukaj cache o rozmiarze równym lub minimalnie większym od żądanego.
2. Wołaj funkcję `__kmem_cache_alloc` dając jej jako parametr znaleziony cache.
3. Opuść semafor
4. Jeśli nie ma wolnych tafli to dodaj nową taflę
5. Oblicz wskaźnik do wolnego obiektu w tafli i zaktualizuj dane w strukturach
6. Podnieś semafor
7. Zwróć obliczony wskaźnik

Teraz bardziej szczegółowo:

Algorytm 1 kmalloc

Dane wejściowe: `size` - rozmiar potrzebnej pamięci, `flags` - flagi
Wynik: adres wolnego obiektu (obszaru pamięci)

1: Przeszukuj iteracyjnie tablicę `cache_sizes`, aż znajdziesz cache o rozmiarze równym lub minimalnie większym od żądanego rozmiaru `size`
2: `cachep` = znaleziony cache
3: Zablokuj przerwania - opuść semafor `cachep->cspin_lock` poprzez funkcję `spin_lock_irqsave`
4: **jeśli** nie ma wolnych tafli (`cachep->c_freep->s_magic != SLAB_MAGIC_ALLOC`) **to**
5: idź do (26)
6: **koniec jeśli**
7: **jeśli** wymagana jest pamięć DMA (`flags & SLAB_DMA`) **to**
8: idź do (21)
9: **koniec jeśli**
10: `cachep->c_freep->s_inuse++`
11: `bufp = cachep->c_freep->s_freep` /* szukany obiekt */
12: `cachep->c_freep->s_freep = bufp->buf_nextp`
13: **jeśli** w tafli nie ma już wolnego obiektu - bufora (`!cachep->c_freep->s_freep`) **to**
14: idź do (19).
15: **koniec jeśli**
16: Odblokuj przerwania - podnieś semafor `cachep->cspin_lock` poprzez funkcję `spin_unlock_irqrestore`
17: oblicz wskaźnik do znalezionego wolnego obiektu `bufp`.
18: Zakończ się zwracając obliczony wskaźnik
19: Zaktualizuj wskaźnik do następnej tafli (`cachep->c_freep = cachep->c_freep->s_nextp`)
20: Idź do (16).
21: /* wyszukuje taflę DMA */
22: Przeszukuj wszystkie tafle, aż znajdziesz taflę z obiektami w pamięci DMA
23: **jeśli** znaleziono taflę z obiektami w pamięci DMA (`cachep->c_freep->s.dma`) **to**
24: Idź do (4)
25: **koniec jeśli**
26: /* alokuje nową taflę */
27: Podnieś semafor `cachep->cspin_lock`.
28: **jeśli** uda się dodać nową taflę z wolnymi obiektami poprzez funkcję `kmem_cache_grow(cachep, flags)` **to**
29: opuść semafor `cachep->cspin_lock`
30: idź do (4)
31: **w przeciwnym przypadku**
32: zakończ się zwracając `NULL`
33: **koniec jeśli**

Literatura

[1] Źródła linuksa - *linux/mm/slab.c*