Laboratorium 2: „Dynamiczny przydział pamięci w przestrzeni jądra Linuksa” (jedne zajęcia)

dr inż. Arkadiusz Chrobot

8 marca 2024

Spis treści

Wprowadzenie

Instrukcja zawiera informacje o sposobie korzystania z pamięci dynamicznej w jądrze systemu Linux. Rozdział 1 zawiera ogólne informacje o mechanizmach dynamicznego zarządzania pamięcią w jądrze Linuksa, które krótko nazywamy alokatorami (ang. *allocators*). Kolejny rozdział (2) zawiera opis i sposób użycia interfejsu alokatora strefowego. Następny rozdział (3) zawiera informacje na temat interfejsu alokatora plastrowego oraz przykłady korzystania z tego mechanizmu. Rozdział 4 opisuje sposoby przydziału/zwalniania pamięci, które choć powiązane są z obydwoma alokatorami, to nie dają się jednoznacznie zakwalifikować jako część jednego z nich. Instrukcję kończy rozdział z zadaniami do samodzielnego wykonania.

1. Alokatory pamięci

Jądro systemu Linux posiada alokatory, które umożliwiają dynamiczne zarządzanie zarówno jego pamięcią, jak i pamięcią należącą do procesów użytkownika. Najważniejsze z nich to alokator strefowy i alokator plastrowy.

Pierwszy z nich jest mechanizmem niskopoziomowym, przydzielającym pamięć w oparciu o algorytm bliźniaków (ang. *buddy system*), który ewidencjonuje obszary fizycznie ciągłe obszary pamięci i przydziela ją porcjami, których wielkość wyraża się potęgą liczby dwa pomnożoną przez wielkość strony. Zatem podstawową jednostką pamięci dla algorytmu bliźniaków jest strona. Nazwa alokatora wynika z tego, że pamięć fizyczna w Linuksie może być podzielona na strefy, dla których ten alokator utrzymuje osobne ewidencje obszarów wolnych. Podział na strefy jest zależny od platformy sprzętowej. Przykładowo, w 32-bitowych komputerach klasy PC pamięć jest typowo dzielona na trzy strefy:

1. przydziałów DMA, czyli obszaru gdzie mogą być tworzone bufory dla urządzeń peryferyjnych wykorzystujących transmisję DMA,
2. zwykłych (ang. *normal*) przydziałów, czyli obszaru z którego przydzielana jest pamięć na typowe potrzeby jądra i procesów użytkownika,
3. przydziałów wysokich (ang. *highmem*), czyli obszaru, który nie jest bezpośrednio adresowany przez jądro.

W przypadku komputera Raspberry Pi 2 istnieje tylko jedna strefa - strefa przydziałów normalnych.

Jądro systemu operacyjnego musi często przydzielać i zwalniać pamięć na różne, wykorzystywane przez siebie struktury danych. Typowe rozwiązania stosowane do tych operacji są zbyt nieekonomiczne i mało wydajne. W Linuksie zastosowano zatem alokator plastrowy[1] (ang. *slab allocator)*, który tworzy pamięci podręczne[2], które stanowią swoiste magazyny struktur danych stosowanych przez jądro. Alokator ten korzysta z usług alokatora strefowego celem utworzenia obszarów pamięci wypełnionych strukturami

[1] W polskiej literaturze można spotkać też określenie „alokator płytowy".
[2] Proszę nie mylić ich ze sprzętowymi pamięciami podręcznymi procesora.

danego typu, np. deskryptorami procesów, pamięci lub buforami dla podsystemu sieciowego. Te obszary nazywane są plastrami i razem tworzą pamięć podręczą[3]. Są one tworzone zanim inne podsystemy jądra zażądają przydzielenia choćby pojedynczej struktury potrzebnej do ich działania. Dzięki temu utworzenie nowej struktury polega na przekazaniu wskaźnika do gotowej struktury umieszczonej w plastrze podsystemowi, który jej zażądał. Zwolnienie zaś polega na oznaczeniu tej struktury jako dostępnej dla kolejnych przydziałów. Dodatkowo alokator plastrowy traktuje wszystkie struktury jako obiekty i umożliwia ich inicjację za pomocą specjalnej funkcji, która pełni funkcję konstruktora[4]. Istnieje też specjalny rodzaj pamięci podręcznych, które nazwane zostały pulami pamięci (ang. *memory pool*). Zapewniają one, że zbiór wolnych struktur się nie wyczerpie i przydział nowej struktury zawsze będzie możliwy. Używane są one w przydziałach krytycznych, czyli takich, które nigdy nie mogą zawieść.

2. Alokator strefowy

W jądrze Linuksa istnieje pięć makr i funkcji zdefiniowanych w pliku nagłówkowym `linux/gfp.h`, które stanowią interfejs alokatora strefowego, tj. umożliwiają przydział i zwalnianie obszarów pamięci za jego pośrednictwem.

`unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)` - Funkcja ta przydziela obszar pamięci o wielkości 2^{order} stron i zwraca adres jego początku. Jeśli przydział się nie powiedzie, to zwraca ona zero. Parametr `order` to wykładnik potęgi określającej liczbę przydzielanych stron, np. aby przydzielić jedną stronę trzeba przez ten parametr przekazać liczbę `0`. Przez pierwszy parametr przekazywany jest tzw. znacznik typu. Określa on jakie czynności mogą być podjęte w trakcie wykonywania przydziału i z której strefy zostanie on dokonany. W ramach tej instrukcji będzie używany znacznik dla zwykłych przydziałów pamięci jądra, czyli `GFP_KERNEL`

`__get_free_page(gfp_mask)` - Makro, które pozwala przydzielić tylko jedną stronę. Jako argument przyjmuje ono znacznik typu, a zwraca adres strony lub zero, jeśli przydział się nie powiedzie.

`unsigned long get_zeroed_page(gfp_t gfp_mask)` - Funkcja ta przydziela pojedynczą stronę, której zawartość jest wyzerowana. Służy ona do realizacji przydziałów dla przestrzeni użytkownika. Wartością przez nią zwracaną jest adres strony lub zero, jeśli przydział się nie powiedzie.

`struct page *alloc_pages(gfp_t gfp_mask, unsigned int order)` - funkcja ta jest funkcją `inline` i przyjmuje te same argumenty co `__get_free_pages()`, ale zamiast adresu przydzielonego obszaru zwraca adres struktury `struct page`. Każda taka struktura jest związana z pojedynczą ramką w pamięci fizycznej komputera. Aby uzyskać adres strony w niej rezydującej należy użyć makra `page_address`, które będzie opisane później.

`alloc_page(gfp_mask)` - przyjmuje ten sam argument co `__get_free_page`, ale zwraca adres struktury `struct page` zamiast, bezpośrednio, adresu strony.

W pamięci fizycznej jądra Linuksa mogą istnieć strefy zawierające strony bez stałego adresu wirtualnego. Te strony rezydują jednak w ramkach, a z każdą ramką związana jest struktura `struct page`. Mając wskaźnik na taką strukturę można się odwołać do strony zawartej w związanej z nią ramce i pozyskać jej adres przy użyciu makra `page_address()`. Jest ono zdefiniowana w pliku nagłówkowym `linux/mm.h` i przyjmuje jako argument adres struktury `struct page`, a zwraca adres strony. Z `alloc_pages()` i `alloc_page` związane są także makra `IS_ERR` oraz `PTR_ERR`. Oba jako argument przyjmują adresy zwrócone przez wymienioną funkcję lub wymienione makro. Pierwsze sprawdza, czy te adresy nie sygnalizują niepowodzenia przydziału pamięci. Jeśli tak, to zwraca wartość różna od zera, a jeśli nie, to zero. Drugie makro dla błędnego adresu pozwala określić kod wyjątku, który pojawił się podczas próby przydziału pamięci.

Pamięć przydzieloną przez wyżej opisane makra i funkcje można zwolnić za pomocą:

`free_page(addr)` - zwalnia pamięć przydzieloną za pomocą `__get_free_page()` lub `get_zeroed_page()`. Przez parametr `addr` przekazywany jest adres strony do zwolnienia.

[3] Plastry określane są też mianem płyt.

[4] Wcześniejsze wersje jądra umożliwiały także zdefiniowanie destruktora, ale programiści nie korzystali z tej możliwości i ostatecznie ją zlikwidowano.

`void free_pages(unsigned long addr, unsigned int order)` - zwalnia pamięć przydzieloną przez `__get_free_pages()`. Jako pierwszy argument wywołania przyjmuje adres zwalnianego obszaru, a jako drugi argument przyjmuje wykładnik potęgi dwójki określającej liczbę stron do zwolnienia.

`__free_page(page)` - zwalnia pamięć przydzieloną przez `alloc_page`. Jako argument przyjmuje adres struktury `struct page`.

`void __free_pages(struct page *page, unsigned int order)` - zwalnia pamięć przydzieloną przez `alloc_pages()`. Pierwszym argumentem jej wywołania jest adres struktury `struct page`, a drugi jest taki sam, jak w przypadku `free_pages()`.

Listing 1 zawiera kod modułu demonstrującego użycie alokatora strefowego. Pamięć przydzielana jest w funkcji inicjującej moduł, a zwalniana w funkcji sprzątającej.

Listing 1: Użycie alokatora strefowego

```
#include<linux/module.h>
#include<linux/gfp.h>
#include<linux/mm.h>

#define ORDER 2

static struct page *page_pointer, *pages_pointer;
static unsigned long int single_page_address, zeroed_page_address,
                    multiple_pages_address;


static int __init mallocmod_init(void)
{
        pages_pointer = alloc_pages(GFP_KERNEL,ORDER);
        if(IS_ERR(pages_pointer)) {
                pr_alert("Error allocating %u pages: %ld!\n",1<<ORDER,
                                PTR_ERR(pages_pointer));
                return -ENOMEM;
        }
        pr_notice("Pages address: %p\n",page_address(pages_pointer));

        page_pointer = alloc_page(GFP_KERNEL);
        if(IS_ERR(page_pointer)) {
                pr_alert("Error allocating page: %ld!\n",PTR_ERR(page_pointer));
                return -ENOMEM;
        }
        pr_notice("Pages address: %p\n",page_address(page_pointer));

        multiple_pages_address = __get_free_pages(GFP_KERNEL,ORDER);
        if(!multiple_pages_address) {
                pr_alert("Error allocating %u pages!\n",1<<ORDER);
                return -ENOMEM;
        }
        pr_notice("Pages address: %lx\n",multiple_pages_address);

        single_page_address = __get_free_page(GFP_KERNEL);
        if(!single_page_address) {
                pr_alert("Error allocating page!\n");
                return -ENOMEM;
        }
        pr_notice("Page address: %lx\n",single_page_address);
```

```
42
43          zeroed_page_address = get_zeroed_page(GFP_KERNEL);
44          if(!zeroed_page_address) {
45                  pr_alert("Error allocating zeroed page!\n");
46                  return -ENOMEM;
47          }
48          pr_notice("Page address: %lx\n",zeroed_page_address);
49
50          return 0;
51  }
52
53  static void __exit mallocmod_exit(void)
54  {
55          if(zeroed_page_address)
56                  free_page(zeroed_page_address);
57          if(single_page_address)
58                  free_page(single_page_address);
59          if(multiple_pages_address)
60                  free_pages(multiple_pages_address,ORDER);
61          if(page_pointer)
62                  __free_page(page_pointer);
63          if(pages_pointer)
64                  __free_pages(pages_pointer,ORDER);
65  }
66
67  module_init(mallocmod_init);
68  module_exit(mallocmod_exit);
69
70  MODULE_LICENSE("GPL");
71  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
72  MODULE_DESCRIPTION("A module demonstrating the usage of zone allocator.");
73  MODULE_VERSION("1.0");
```

Wykładnik potęgi określającej liczbę przydzielanych stron dla wywołań `alloc_pages()` i `get_pages()` został określony za pomocą stałej `ORDER`. W przypadku powodzenia przydziału wyświetlany jest adres przydzielonego obszaru, a w przypadku niepowodzenia kod wyjątku oraz funkcja inicjująca moduł zwraca błąd określony wyrażeniem `-ENOMEM`. Stała `ENOMEM` przewidziana jest dla wyjątków spowodowanych brakiem pamięci lub niepowodzeniem jej przydzielenia. Należy zaznaczyć, że pamięć przydzielana przez alokator strefowy jest fizycznie ciągła.

3. Alokator plastrowy

Alokator plastrowy posługuje się dwoma mechanizmami celem przydziału struktur potrzebnych do pracy innych podsystemów jądra - pamięciami podręcznymi i pulami pamięci. W dalszej części rozdziału zostanie opisane API obu z nich.

3.1. Pamięci podręczne

Ta część instrukcji zawiera opis funkcji, które służą do zarządzania pamięciami podręcznymi oraz do przydzielania i zwalniania struktur z tych pamięci. Wszystkie one są zdefiniowane w pliku `linux/slatb.h`.

`struct kmem_cache *kmem_cache_create(const char *, size_t, size_t, unsigned long, void (*)(void *))`
- ta funkcja tworzy nową pamięć podręczną i zwraca wskaźnik na jej deskryptor, czyli strukturę typu `struct kmem_cache`. Przyjmuje ona pięć argumentów wywołania. Pierwszym jest łańcuch znaków zawierający czytelną dla człowieka nazwę pamięci podręcznej. Jest ona prezentowana w pliku

`/proc/slabinfo` zawierającym dane statystyczne wszystkich pamięci podręcznych. Drugim argumentem jest wielkość pojedynczego obiektu (struktury) przechowywanego w pamięci podręcznej. Trzeci argument to wielkość przesunięcia pierwszego obiektu względem początku plastra. Zazwyczaj jego wartość wynosi `0`. Czwarty argument to flagi. Do dyspozycji programiści mają kilka flag, które mogą być przekazane pojedynczo lub zsumowane logicznie:

`SLAB_HWCACHE_ALIGN` - flaga powoduje wyrównanie wszystkich obiektów w plastrach do wielkości linii w sprzętowej pamięci podręcznej. Jej ustawienie zwiększa wydajność działania alokatora plastrowego, ale zwiększa wykorzystanie pamięci. Zaleca się jej stosowanie tylko dla pamięci podręcznych obiektów, które muszą działać szybko.

`SLAB_POISON` - powoduje wypełnienie plastrów określoną wartością, aby ułatwić wychwycenie prób dostępu do niezainicjowanej pamięci.

`SLAB_RED_ZONE` - ustawienie tej flagi ułatwia wykrycie błędów typu „przekroczenie bufora” (ang. *buffer overrun*).

`SLAB_PANIC` - ustawienie tej flagi powoduje, że niepowodzenie przydziału obiektu kończy się krytycznym błędem jądra (ang. *kernel panic*).

`SLAB_CACHE_DMA` - instruuje alokator, aby przydzielił pamięć na plastry w strefie DMA.

Ostatni argument to wskaźnik na funkcję, która nie zwraca żadnej wartości, ale przyjmuje jeden parametr typu `void *`. Ta funkcja pełni rolę konstruktora, czyli dokonuje inicjacji każdej struktury stworzonej w pamięci podręcznej. Programista nie musi jej definiować. Jeśli nie chce tego robić, może jako ostatni parametr `kmem_cache_create()` przekazać wartość `NULL` lub po prostu `0`.

`void *kmem_cache_alloc(struct kmem_cache *, gfp_t flags)` - ta funkcja służy do przydziału pojedynczej struktury (obiektu) z pamięci podręcznej. Zwraca wskaźnik typu `void *`. Jego poprawność można sprawdzić za pomocą makra `is_err`, a ewentualny kod wyjątku ustalić makrem `ptr_err`. Funkcja przyjmuje dwa argumenty wywołania: wskaźnik na pamięć podręczną, z której mam nastąpić przydział oraz znacznik typu.

`void kmem_cache_free(struct kmem_cache *, void *)` - ta funkcja zwalnia strukturę (obiekt) przydzieloną z pamięci podręcznej, do której wskaźnik jest jej przekazywany jako pierwszy argument wywołania. Jako drugi argument przekazywany jest jej wskaźnik na zwalnianą strukturę.

`void kmem_cache_destroy(struct kmem_cache *)` - ta funkcja usuwa pamięć podręczną, na którą wskaźnik został jej przekazany jako argument jej wywołania.

Listing 2 zawiera kod źródłowy modułu, który tworzy pamięć podręczną dla przykładowych struktur i dokonuje przydziału, a następnie zwolnienia pojedynczej takiej struktury.

Listing 2: Użycie alokatora plastrowego - pamięć podręczna

```
1  #include<linux/module.h>
2  #include<linux/slab.h>
3  #include<linux/string.h>
4
5  static struct example_struct {
6          unsigned int id;
7          char example_string[10];
8  } *example_struct_pointer;
9
10 static struct kmem_cache *example_cachep;
11
12 static void example_constructor(void *argument)
13 {
14         static unsigned int id;
15         static char test_string[] = "Test";
```

```
16          struct example_struct *example = (struct example_struct *)argument;
17          example->id = id;
18          strcpy(example->example_string,test_string);
19          id++;
20  }
21
22  void print_example_struct(struct example_struct *example)
23  {
24          pr_notice("Example struct id: %u\n",example->id);
25          pr_notice("Example string field content: %s\n",example->example_string);
26  }
27
28  static int __init slabmod_init(void)
29  {
30          example_cachep = kmem_cache_create("example cache",
31                          sizeof(struct example_struct),0,
32                          SLAB_HWCACHE_ALIGN|SLAB_POISON|SLAB_RED_ZONE,
33                          example_constructor);
34          if(IS_ERR(example_cachep)) {
35                  pr_alert("Error creating cache: %ld\n",PTR_ERR(example_cachep));
36                  return -ENOMEM;
37          }
38
39          example_struct_pointer = (struct example_struct *)
40                  kmem_cache_alloc(example_cachep,GFP_KERNEL);
41          if(IS_ERR(example_struct_pointer)) {
42                  pr_alert("Error allocating form cache: %ld\n",
43                                  PTR_ERR(example_struct_pointer));
44                  kmem_cache_destroy(example_cachep);
45                  return -ENOMEM;
46          }
47
48          return 0;
49  }
50
51  static void __exit slabmod_exit(void)
52  {
53          if(example_cachep) {
54                  if(example_struct_pointer) {
55                          print_example_struct(example_struct_pointer);
56                          kmem_cache_free(example_cachep,example_struct_pointer);
57                  }
58                  kmem_cache_destroy(example_cachep);
59          }
60  }
61
62  module_init(slabmod_init);
63  module_exit(slabmod_exit);
64
65  MODULE_LICENSE("GPL");
66  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
67  MODULE_DESCRIPTION("A module demonstrating the use of the slab allocator.");
68  MODULE_VERSION("1.0");
```

Wiersze 5-8 zawierają definicję typu przykładowej struktury, dla której moduł utworzy pamięć pod-

ręczną. Dodatkowo w wierszu 8 zadeklarowano wskaźnik dla tej struktury. W wierszu 10 znajduje się definicja wskaźnika do pamięci podręcznej. Na mocy konwencji przyjętej przez programistów jądra Linuksa wszystkie takie wskaźniki mają nazwę kończącą się na „ep". Funkcja `example_constructor()` (wiersze 12-20) jest odpowiedzialna za zainicjowanie każdej struktury umieszczonej w pamięci lokalnej. Taka struktura (obiekt) jest jej przekazywana przez parametr `argument` będący wskaźnikiem typu `void *`, który jest w wierszu 16 jest rzutowany na lokalny wskaźnik typu `struct example`. Dodatkowo ta funkcja posiada dwie statyczne zmienne lokalne. Pierwsza jest typu `int`, a druga to tablica znaków. Uczynienie zmiennej `id` lokalną pozwala zwiększać jej wartość podczas kolejnych wywołań funkcji. Dzięki temu pola `id` kolejnych struktur będą otrzymywały wartości będące kolejnymi liczbami naturalnymi. Pole `example_string` będzie dla odmiany otrzymywało tę samą wartość, czyli ciąg znaków „`Test`", skopiowany ze zmiennej `test_string`. Ta zmienna jest statyczna z innego powodu niż zmienna `id`. **W jądrze Linuksa stos ma wielkość dwóch stron. W przypadku komputerów PC jest to 8 KiB, dlatego trzeba oszczędnie tą pamięcią gospodarować. Należy unikać tworzenia dużych zmiennych lokalnych lub tworzyć jej jako zmienne statyczne.** Funkcja `print_example()` zdefiniowana w wierszach 22-26. Służy do umieszczania zawartości przykładowej struktury do bufora jądra. Reszta kodu zawiera wywołania opisanych wcześniej funkcji. Proszę zwrócić uwagę na to, że w module najpierw tworzona jest pamięć podręczna, a potem przykładowa struktura, ale są one zwalniane w odwrotnej kolejności - najpierw struktura (obiekt), później pamięć podręczna. Proszę także zwrócić uwagę na obsługę wyjątku funkcji `kmem_cache_alloc()`. Jeśli przydział się nie powiedzie, to nie tylko sygnalizowany jest błąd przydziału, ale zwalniana jest pamięć podręczna.

3.2. Pule pamięci

Do zarządzania pulami pamięci przeznaczone są dwie funkcje, które stają się dostępne w module jądra po włączeniu nagłówka `linux/mempool.h`. Są to:

`mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn, mempool_free_t *free_fn, void *pool_data)` - funkcja ta zwraca wskaźnika na utworzoną pulę pamięci (typu `mempool_t`). Jeśli przydział się nie powiedzie, to można to wykryć za pomocą makra `IS_ERR`, a kod wyjątku uzyskać z tego wskaźnika za pomocą `PTR_ERR`. Funkcja przyjmuje cztery argumenty: wartość określającą minimalną liczbę obiektów, która zawsze musi być dostępna, wskaźnik na funkcję przydzielającą obiekt, wskaźnik na funkcję zwalniającą obiekt oraz wskaźnik na obszar pamięci, z którego pula może zostać utworzona. Zwykle ostatnim argumetem jest wskaźnik na pamięć podręczą. Definiowanie własnych funkcji przydziału i zwalniania obiektu też nie jest konieczne. Są zdefiniowane dwa wskaźniki o nazwach `mempool_alloc_slab` i `mempool_free_slab`, które wskazują na gotowe funkcje przydziału i zwalniania o nazwach, odpowiednio, `mempool_alloc()` i `mempool_free()`. Wystarczy je przekazać jako drugi i trzeci argument `mempool_create()`, aby móc wskazywanymi przez nie funkcjami się posługiwać.

`void mempool_destroy(mempool_t *pool)` - ta funkcja usuwa pulę pamięci, na którą wskaźnik jest jej przekazany jako argument wywołania.

Listing 3 zawiera moduł ilustrujący sposób użycia puli pamięci.

Listing 3: Użycie alokatora plastrowego - pula pamięci

```
1  #include<linux/module.h>
2  #include<linux/slab.h>
3  #include<linux/mempool.h>
4  #include<linux/string.h>
5
6  #define MINIMUM_MEMPOOL_OBJECTS 10
7
8  static struct example_struct {
9          unsigned int id;
10         char example_string[10];
11 } *example_struct_pointer;
```

```
12
13  static struct kmem_cache *example_cachep;
14  static mempool_t *mempool_pointer;
15
16  static void example_constructor(void *argument)
17  {
18          static unsigned int id;
19          static char test_string[] = "Test";
20          struct example_struct *example = (struct example_struct *)argument;
21          example->id = id;
22          strcpy(example->example_string,test_string);
23          id++;
24  }
25
26  void print_example_struct(struct example_struct *example)
27  {
28          pr_notice("Example struct id: %u\n",example->id);
29          pr_notice("Example string field content: %s\n",example->example_string);
30  }
31
32  static int __init slabmod_init(void)
33  {
34          example_cachep = kmem_cache_create("example cache",
35                  sizeof(struct example_struct),0,
36                  SLAB_HWCACHE_ALIGN|SLAB_POISON|SLAB_RED_ZONE,
37                  example_constructor);
38          if(IS_ERR(example_cachep)) {
39                  pr_alert("Error creating cache: %ld\n",PTR_ERR(example_cachep));
40                  return -ENOMEM;
41          }
42
43          mempool_pointer = mempool_create(MINIMUM_MEMPOOL_OBJECTS,
44                          mempool_alloc_slab,mempool_free_slab,example_cachep);
45          if(IS_ERR(mempool_pointer)) {
46                  pr_alert("Error creating cache: %ld\n",PTR_ERR(mempool_pointer));
47                  kmem_cache_destroy(example_cachep);
48                  return -1;
49          }
50
51          example_struct_pointer = (struct example_struct *)
52                                  mempool_alloc(mempool_pointer,GFP_KERNEL);
53          if(IS_ERR(example_struct_pointer)) {
54                  pr_alert("Error allocating form cache: %ld\n",
55                                          PTR_ERR(example_struct_pointer));
56                  mempool_destroy(mempool_pointer);
57                  kmem_cache_destroy(example_cachep);
58                  return -ENOMEM;
59          }
60
61          return 0;
62  }
63
64  static void __exit slabmod_exit(void)
65  {
66          if(example_cachep) {
```

```
67                  if(mempool_pointer) {
68                          if(example_struct_pointer) {
69                                  print_example_struct(example_struct_pointer);
70                                  mempool_free(example_struct_pointer,mempool_pointer);
71                          }
72                          mempool_destroy(mempool_pointer);
73                  }
74                  kmem_cache_destroy(example_cachep);
75          }
76  }
77
78  module_init(slabmod_init);
79  module_exit(slabmod_exit);
80
81  MODULE_LICENSE("GPL");
82  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
83  MODULE_DESCRIPTION("A module demonstrating the use of a mempool.");
84  MODULE_VERSION("1.0");
```

Jak łatwo się zorientować moduł ten jest przerobioną wersją tego, który demonstrował użycie pamięci podręcznych. Elementami, które zostały dodane to tworzenie puli pamięci na podstawie utworzonej wcześniej pamięci podręcznej oraz jej usuwanie. Proszę zwrócić uwagę na wywołania `mempool_alloc()` (wiersz 52) i `mempool_free()`. Pierwsze przyjmuje dwa argumenty: wskaźnik na pulę pamięci, z której ma być przydzielony obiekt, oraz znacznik typu, a zwraca wskaźnik typu `void *`. Drugie nic nie zwraca, ale przyjmuje wskaźnik na strukturę do zwolnienia oraz na pulę pamięci.

4. Inne metody przydziału przydziału/zwalniania pamięci

Istnieją jeszcze cztery funkcje, które służą do dynamicznego zarządzania pamięcią w przestrzeni jądra. Zostały one opisane w tabeli 1.

Prototyp	Opis
`void *kmalloc(size_t size, gfp_t flags)`	Funkcja ta dostępna jest po włączeniu do kodu pliku nagłówkowego `linux/slab.h`. Przydziela pamięć ciągłą fizycznie. Pierwszym argumentem jej wywołania jest rozmiar tworzonego obszaru, a drugim znacznik typu. Funkcja zwraca wskaźnik na przydzielony obszar lub informację o wyjątku, którą można obsłużyć za pomocą makr `IS_ERR` i `PTR_ERR`. Ponieważ ta funkcja korzysta z alokatora strefowego, to przydzielony obszar jest zawsze wielokrotnością wielkości strony, ale programista powinien wykorzystać tylko tyle z tego obszaru, ile określił w pierwszym argumencie wywołania tej funkcji.
`void kfree(const void *)`	Ta funkcja służy do zwalniania pamięci przydzielonej za pomocą `kmalloc()` i tylko przy jej użyciu można taką pamięć zwalniać.
`void *vmalloc(unsigned long size)`	Jest to funkcja, która przydziela pamięć ciągłą wirtualnie, ale niekoniecznie ciągłą fizycznie. Można z niej skorzystać po włączeniu nagłówka `linux/vmalloc.h` Przyjmuje jeden argument wywołania, którym jest rozmiar pamięci jaka ma zostać przydzielona. Wartością przez nią zwracaną jest adres przydzielonej pamięci lub wartość sygnalizująca błąd, którą można zinterpretować za pomocą makr `IS_ERR` oraz `PTR_ERR`.
`void vfree(const void *addr)`	Ta funkcja zwalnia pamięć przydzielaną za pomocą `vmalloc()` i tylko jej należy użyć do zwalniania takiej pamięci. Nie zwraca ona żadnej wartości, ale przyjmuje jeden argument wywołania jakim jest wskaźnik na zwalniany obszar pamięci.

Tabela 1: Inne funkcje związane z dynamicznym zarządzaniem pamięcią w przestrzeni jądra

Obszary pamięci ciągłe fizycznie są wymagane głównie przez sterowniki urządzeń korzystających z transmisji DMA do tworzenia buforów. Jednakże w kodzie jądra częściej można spotkać pamięć przydzielaną za pomocą `kmalloc()` niż `vmalloc()`, ponieważ dostęp do tej pierwszej jest bardziej efektywny. Dostęp do pamięci przydzielonej przez `vmalloc()` wymaga wykonania translacji adresów z użyciem tablicy stron. Funkcja `vmalloc()` jest zatem częściej używana w kodzie związanym z przydziałem pamięci dla przestrzeni użytkownika.

Listing 4 zawiera kod modułu, który przydziela i zwalnia pamięć opisanymi w tabel 1 funkcjami.

Listing 4: Użycie `kmalloc()` i `vmalloc()`

```
1  #include<linux/module.h>
2  #include<linux/vmalloc.h>
3  #include<linux/slab.h>
4
5  #define NUMBER_OF_ELEMENTS 20
6  #define MEMORY_SIZE NUMBER_OF_ELEMENTS*sizeof(int)
7
8  static int *first_array, *second_array;
9
10
11 static int __init mallocmod_init(void)
12 {
13         first_array = (int *)vmalloc(MEMORY_SIZE);
```

```
14          if(IS_ERR(first_array)) {
15                  pr_alert("Error allocating memory with the use of vmaloc(): %ld\n",
16                  PTR_ERR(first_array));
17                  return -ENOMEM;
18          }
19          pr_notice("The address in first_array: %p\n",first_array);
20
21          second_array = (int *)kmalloc(MEMORY_SIZE,GFP_KERNEL);
22          if(IS_ERR(second_array)) {
23                  pr_alert("Error allocating memory with the use of vmaloc(): %ld\n",
24                  PTR_ERR(second_array));
25                  return -ENOMEM;
26          }
27          pr_notice("The address in second_array: %p\n",second_array);
28
29          return 0;
30  }
31
32  static void __exit mallocmod_exit(void)
33  {
34          int i;
35
36          if(second_array) {
37                  pr_notice("second_array content:\n");
38                  for(i=0;i<NUMBER_OF_ELEMENTS;i++)
39                          pr_notice("%d ",second_array[i]);
40                  pr_notice("\n");
41                  kfree(second_array);
42          }
43
44          if(first_array) {
45                  pr_notice("first_array content:\n");
46                  for(i=0;i<NUMBER_OF_ELEMENTS;i++)
47                          pr_notice("%d ",first_array[i]);
48                  pr_notice("\n");
49                  vfree(first_array);
50          }
51  }
52
53  module_init(mallocmod_init);
54  module_exit(mallocmod_exit);
55
56  MODULE_LICENSE("GPL");
57  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
58  MODULE_DESCRIPTION("A module demonstrating the usage of different kernel\
59          memory allocators.");
60  MODULE_VERSION("1.0");
```

Zadania

1. [2 punkty] Zmień moduł z listingu 2 tak, aby przydzielał pięć przykładowych struktur i zapisywał ich zawartość w buforze jądra. Sprawdź, badając wypisywane pola `id` tych struktur, czy kolejność przydzielania obiektów z plastra jest taka sama jak ich inicjacji.

2. [4 punktów] Napisz moduł, w którym w konstruktorze stworzysz stos, a w destruktorze wypiszesz

jego zawartość i go usuniesz.

3. `[6 punktów]` Napisz moduł, w którym w konstruktorze stworzysz listę dwukierunkową, a w destruktorze wypiszesz jej zawartość w obie strony i usuniesz tę listę.

4. `[2 punkty]` Wprowadź zmiany opisane w zadaniu pierwszym do modułu z listingu 3.

5. `[4 punktów]` Napisz moduł, w którym w konstruktorze stworzysz kolejkę FIFO, a w destruktorze wypiszesz jej zawartość i ją usuniesz.

6. `[6 punktów]` Napisz moduł, w którym w konstruktorze stworzysz jednokierunkową listę cykliczną, a w destruktorze wypiszesz jej zawartość i ją usuniesz. Jednokierunkowa lista cykliczna to lista jednokierunkowa, której elementy są połączone w okrąg, a żaden ze wskaźników listy nie jest pusty.