Sprawozdanie z projektu NiDUC

Ochrona funkcji systemowych

Maciej Myśków 272794

Kacper Kostrzewa 272855

Data oddania

25.06.2024

Spis treści

Wstęp

Współczesne systemy operacyjne, w tym Linux wymagają skutecznego zarządzania pamięcią, aby zapewnić stabilność, efektywność i bezpieczeństwo działania aplikacji. Dynamiczne przydzielanie pamięci jest kluczowym elementem w programowaniu systemowym i aplikacjach operujących na niskim poziomie. Funkcje takie jak kmalloc, kmalloc_array i kcalloc w jądrze Linux są fundamentalne do tego procesu, umożliwiając programistom elastyczne zarządzanie pamięcią w czasie rzeczywistym.

Dynamiczna alokacja pamięci jest procesem wymagającym dosyć sporej świadomości od programistów, gdyż niewłaściwa alokacja może zepsuć program a nawet uszkodzi system operacyjny.

Paradoksalnie sytuacje utrudniają tzw. Funkcje bliźniacze (funkcje dynamicznie alokujące pamięć). Przez to, że jest ich tak dużo i w swoim działaniu a nawet sygnaturze są tak podobne, może dojść do tzw. Nadużyć (skorzystanie z niewłaściwej funkcji systemowej). Z drugiej strony mnogość wyżej wspomnianych funkcji rozwiązuje wiele problemów i wprowadza większą elastyczność programistą – stąd tak ważne jest aby dokładnie wiedzieć z jakiej funkcji skorzystać.

W projekcie przyjrzeliśmy się trzem najpowszechniej korzystanym funkcją: Kmalloc, Kmalloc_array, Kcalloc. Każdą z tych funkcji zbadaliśmy (sygnatury, jej działanie, testy niezawodności) a następnie wyciągnęliśmy wnioski z wyników testowych.

Cele projektu

Celem projektu jest zbadanie i analiza ochrony funkcji systemowych w jądrze systemu Linux, ze szczególnym uwzględnieniem trzech funkcji dynamicznej alokacji pamięci: kmalloc, kmalloc_array oraz kcalloc. Projekt ma na celu:

1. Zrozumienie charakterystyki funkcji systemowych**:**
    - Analiza sygnatur i sposobu działania funkcji kmalloc, kmalloc_array, kcalloc.
    - Zrozumienie różnic i podobieństw pomiędzy tymi funkcjami oraz ich praktycznego zastosowania w systemie operacyjnym.
2. Napisanie sterownika do jądra**:**
    - Opracowanie i implementacja modułu sterownika do jądra Linux, który umożliwi przeprowadzenie testów funkcji alokujących pamięć.
    - Testowanie działania sterownika w zależności od potrzeb – testy dla wybranych wariantów alokacji pamięci.
3. Przeprowadzenie testów funkcji systemowych**:**
    - Przeprowadzenie testów wydajności, poprawności oraz zachowania się alokacji pamięci przez wybrane funkcje.
    - Przeprowadzenie testów złożoności czasowej alokacji i zwalniania pamięci.
4. Ocena efektywności i stabilności funkcji**:**
    - Analiza wyników testów pod kątem wydajności, stabilności oraz efektywności alokacji pamięci.
    - Analiza czasowa złożoności funkcji alokujących – pomiary czasu
5. Wnioski i obserwacje**:**
    - Wyciągnięcie wniosków z przeprowadzonych testów i przedstawienie subiektywnego spojrzenia dotyczącego wyboru odpowiednich funkcji systemowych w zależności od kontekstu użycia.
    - Krótkie omówienie potencjalnych zagrożeń związanych z niewłaściwym wykorzystaniem funkcji systemowych.

Projekt ma na celu pogłębienie wiedzy na temat zarządzania pamięcią w systemach operacyjnych oraz dostarczenie praktycznych wskazówek dotyczących efektywnego wykorzystania funkcji dynamicznej alokacji pamięci w jądrze Linux.

Charakterystyka funkcji systemowych

Każda z wymienionych wyżej funkcji posiada swoją sygnaturę oraz sposób działania. Bardzo istotne jest aby zrozumieć charakterystyki tych funkcji, tak aby ich rozsądnie używać. Rozsądne używanie funkcji systemowych może ominąć problemy takie jak niekontrolowany dostęp do pamięci, wyciek pamięci czy też zepsucie systemu operacyjnego. Dlatego tak ważne jest aby znać te charakterystyki.

Sygnatury funkcji

Funkcja	Sygnatura	Parametry
Kmalloc	void *kmalloc(size_t size, gfp_t flags);	size, flaga
Kmalloc_array	void *kmalloc_array(size_t n, size_t size, gfp_t flags);	n, size, flaga
Kcalloc	void *kcalloc(size_t n, size_t size, gfp_t flags);	n, size, flaga

size_t: oznacza rozmiar alokowanej pamięci w bajtach

flags: flagi sterujące, określają m.in. rodzaj pamięci, którą chcemy zaalokować

n: liczba elementów w tablicy

size: rozmiar pojedynczego elementu dla tablicy

Kmalloc

Jest to podstawowa funkcja do alokacji pamięci w Linuxie.

Praktyczne zastosowania:

- Przechowywanie danych: Alokacja pamięci na struktury danych, listy, bufory.
- Obsługa sterowników: Przechowywanie kontekstów sterowników, buforowanie danych wejściowych/wyjściowych.
- Obsługa plików: Bufory na dane do odczytu/zapisu.

Kmalloc_array

Jest rozszerzeniem kmalloc i służy do alokacji dynamicznej tablicy o określonym rozmiarze.

Praktyczne zastosowania:

- Tablice dynamiczne: Przechowywanie listy elementów o zmiennej liczbie.
- Zarządzanie pamięcią: Alokacja pamięci dla dynamicznych struktur danych, np. list wiązanych.

Kcalloc

kcalloc to specjalna odmiana kmalloc_array, która dodatkowo zeruje zaalokowaną pamięć. Mówiąc krótko stosując ten typ alokacji tablicowej, dostajemy zaalokowany obszar pamięci przeznaczony na typ tablicowy, którego wszystkie elementy są inicjalizowane zerami już podczas kompilacji.

Praktyczne zastosowania:

- Inicjalizacja zerami: Tworzenie tablic, które wymagają początkowej inicjalizacji zerami.
- Bezpieczeństwo danych: Zapewnienie, że pamięć jest czysta i nie zawiera danych z poprzednich alokacji.

Flagi i ich znaczenie

W funkcjach kmalloc, kmalloc_array i kcalloc w jądrze Linux, istotną rolę odgrywają flagi, które określają sposób alokacji pamięci. Flagi te są przekazywane jako parametry do funkcji alokujących i wpływają na zachowanie procesu przydzielania pamięci. Zrozumienie tych flag jest kluczowe dla prawidłowego i efektywnego zarządzania pamięcią w jądrze Linux. W naszym projekcie przyjrzymy się dwom flagom, mianowicie: GFP_kernel oraz GFP_atomic.

- **GFP_KERNEL**
  Flaga ta jest używana, gdy alokacja pamięci jest wykonywana w kontekście jądra, gdzie można bezpiecznie wywołać funkcje, które mogą być w trybie sleep. Ponadto ta flaga nie może być wykorzystywana, gdy używamy przerwań systemowych podczas alokacji.

  Zastosowanie:
  Jest to najczęściej używana flaga dla alokacji pamięci w większości części jądra.

  Zachowanie:
  Funkcja może blokować wykonanie i czekać na dostępność pamięci, co zapewnia wysoką szansę na powodzenie alokacji (wywołanie alokacji funkcji nie zwróci nam wartości NULL).

- **GFP_ATOMIC**
  Flaga ta jest używana w kontekstach, gdzie alokacja pamięci nie może być w trybie sleep, czyli w sekcjach kodu, które muszą być wykonywane szybko i nieprzerwanie. Pozwala na zastosowanie przerwań systemowych.

  Zastosowanie:
  Używana w kontekstach przerwań i innych miejscach, gdzie alokacja musi być natychmiastowa.

  Zachowanie:
  Funkcja nie może blokować wykonania. Jeśli pamięć nie jest dostępna, alokacja może się nie powieść, co oznacza mniejsze szanse na powodzenie w porównaniu z GFP_KERNEL – możliwe zwrócenie wartości NULL przy próbie alokacji.

Proces kompilacji sterowników jądra

Aby poprawnie skompilować i uruchomić testy u siebie lokalnie, trzeba wykonać kilka podstawowych kroków, które opiszemy poniżej.

Po pierwsze trzeba wykonać polecenie make w katalogu bieżącym całego kodu, korzystając z dołączonego przez nas w sprawozdaniu kodu Makefile [kod5.].

Następnie aby dołączyć jądro do kernela użyliśmy polecenia: **sudo insmod [nazwa_modułu].ko**

Aby wyświetlić systemowy dziennik zdarzeń jądra Linuxa, użyliśmy polecenia: **sudo dmesg | tail**

Trzeba pamiętać, że po zakończonych testach należy odłączyć moduł jądra, gdyż w przeciwnym przypadku możemy nadpisać pozostały moduł, co prowadzi do licznych błędów – w tym crash systemu (który doświadczyliśmy niejednokrotnie). Wykonujemy to poleceniem**: sudo rmmod [nazwa_modułu]**

Jakich testów dokonano

W projekcie przeprowadziliśmy testy na najpowszechniej stosowanych funkcjach systemowych: kmalloc, kmalloc_array, kcalloc. Testy zostały u nas podzielone na dwie kategorie: testy działania i poprawności alokacji pamięci przez funkcje oraz testy złożoności czasowej alokacji i zwalniania pamięci.

Testy alokacji pamięci

Testy te miały na celu sprawdzić wydajność oraz poprawność alokacji pamięci w zależności od inicjalizacji. Testy te są bardzo istotne z punktu widzenia wyboru funkcji systemowej, gdyż ich wyniki mogą być pomocne w podjęciu właściwej decyzji co do wyboru funkcji bliźniaczej.

Testy czasów alokacji i zwalniania pamięci

Badanymi funkcjami w tym teście były Kmalloc_array oraz Kcalloc. Testów dokonano w celu zbadania czasów użytkowania funkcji systemowych alokujących struktury tablicowe w zależności od ich rozmiaru. Wyniki tych badań pozwalają odpowiedzieć na pytanie, której funkcji użyć jeśli np. obie spełniają tak samo testy i wymagania alokacji pamięci.

Jak dokonywano testów

Ponieważ testowaliśmy alokacje pamięci bezpośrednio na jądrze (kernel) Linuxa, używając języka C, wymagało to od nas znajomości programowania powłoki kernel oraz napisania własnego małego modułu do poruszania się po tej powłoce.

Wszystkich testów dokonywano w środowisku testowym – zwirtualizowanym na podstawie obrazu systemu ISO – Linux Ubuntu wraz zainstalowanym nagłówkiem jądra systemu, który umożliwiał kompilacje do jądra. Stworzono również Makefile, aby zautomatyzować proces kompilacji.

Wygenerowaliśmy w sumie cztery moduły, gdzie każdy z nich był odpowiedzialny za inny rodzaj przeprowadzanych testów.

Moduł o nazwie test_performance_allocation zawiera testy alokacji dużych bloków pamięci. Testowaliśmy tutaj wydajność alokacji a jednostką wydajności były tzw. Jiffies czyli jednostka czasu używana przez jądro systemu operacyjnego Linux'a do mierzenia upływu czasu. Jest to liczba ticków (tyknięć) zegara systemowego, które upłynęły od momentu uruchomienia systemu. Rezultat pomiaru może wskazać najwydajniejszą funkcję do alokacji pamięci z trzech testowanych.

Moduł o nazwie test_non_initialized testuje poprawność inicjalizacji zaalokowanej pamięci przez funkcji systemowe. W tym teście celowo nie inicjalizowaliśmy pamięci żadnymi wartościami początkowymi, aby sprawdzić jak funkcje sobie radzą z zerową inicjalizacją. W szczególności skupiliśmy się na wyniku zerowej inicjalizacji Kmalloc_array i Kcalloc, gdyż według dokumentacji ta druga funkcja powinna zainicjalizować na tablicę zerami.

Moduł test_max_allocation sprawdza jak funkcje radzą sobie z inicjalizacją maksymalnych rozmiarów bloków pamięci. Wynikiem tego testu jest informacja czy funkcja zdołała przydzielić zaalokowaną pamięć dla bloku czy też próba alokacji skończyła się błędem – co by świadczyło o poprawnym wykryciu przez funkcje przekroczenia dozwolonego rozmiaru oraz zablokowaniu takiej możliwości. W przeciwnym razie mogłoby dojść do braku pamięci dla innych procesów np. systemowych co skutkuje zawieszeniem systemu.

Ostatnim modułem testowym jest moduł test_times. Moduł ten testuje czasy alokacji pamięci i zwalniania jej dla funkcji alokujących tablice – Kmalloc_array i Kcalloc. Pomiary czasu są dokonywane w zależności od reprezentatywnych ilości elementów tablicy n (od 1 do 1000) a wynik jest podawany w nanosekundach. Test jest wykonywany za pomocą pętli a czas jest mierzony za pomocą modułu standardowego time. Test ten miał na celu sprawdzić jak funkcje radzą sobie z alokacją danych tablicowych w kontekście złożoności czasowej.

Wyniki testów

Po przeprowadzeniu wszystkich wyżej wymienionych testów, zebrano wyniki oraz przedstawiono je w formacie tabelarycznym.

Wyniki alokacji pamięci

Funkcja	Wydajność alokacji dużych bloków pamięci (Jifies)	Test pustej inicjalizacji - zwrócona wartość	Test maksymalnej alokacji (sukces/błąd)
Kmalloc	52	pusta alokacja	błąd
Kmalloc_array	28	pusta alokacja	błąd
Kcalloc	31	0	błąd

1. Wydajność alokacji dużych bloków pamięci:

    - **kmalloc_array** wykazała najwyższą wydajność (28 jiffies) w porównaniu do kmalloc (52 jiffies) i kcalloc (31 jiffies). Sugeruje to, że kmalloc_array jest najbardziej optymalną funkcją do alokacji dużych bloków pamięci.
    - **kcalloc** jest nieco mniej wydajna od kmalloc_array, ale bardziej efektywna niż kmalloc.

2. Pusta inicjalizacja:

    - Zarówno kmalloc, jak i kmalloc_array zwracają pustą alokację, co oznacza brak dodatkowej inicjalizacji przydzielonej pamięci.
    - **kcalloc** zwraca 0, co oznacza, że dodatkowo inicjalizuje przydzieloną pamięć do zera, co może być korzystne w programach, które wymagają czystej pamięci po alokacji.

3. Maksymalna alokacja:

    - Wszystkie testowane funkcje nie były w stanie przydzielić maksymalnego bloku pamięci, co wskazuje na ograniczenia funkcji systemowych, dzięki czemu unikamy możliwości zalokowania większej ilości pamięci niż dozwolona. Wskazuje to na poprawne działanie funkcji

Wyniki złożoności czasowej

Funkcja	Czasy alokacji/zwalniania dla tablic n = 10 (ns)	Czasy alokacji/zwalniania dla tablic n = 100 (ns)	Czasy alokacji/zwalniania dla tablic n = 1000 (ns)
Kmalloc_array	674/386	10887/6154	173215/94821
Kcalloc	554/829	12844/6313	147357/188358
Uwagi	Przy alokacji tablicy powyżej 1tys. elementów dochodzi do uszkodzenia jądra systemu operacyjnego na wskutek przepełnienia bufora		

- **kmalloc_array** wykazała ogólnie szybsze czasy zwalniania pamięci niż kcalloc.
- Przy mniejszych tablicach (n = 10), kcalloc była szybsza w alokacji, ale wolniejsza w zwalnianiu w porównaniu do kmalloc_array.
- Przy większych tablicach (n = 100 i n = 1000), **kmalloc_array** była szybsza zarówno w alokacji, jak i zwalnianiu w porównaniu do kcalloc.

Na szczególną uwagę zasługuje fakt, iż alokacja pamięci w przypadku tablicy powyżej tysiąca elementów, dochodzi do uszkodzenia systemu, na wskutek przepełnienia bufora. Nie jest to pożądane zjawisku, gdyż w najlepszym razem nie mamy już pamięci na inne procesy a w najgorszym może dojść z uszkodzenia systemu operacyjnego.

Wnioski końcowe

Po przeprowadzeniu testów oraz po wnikliwej analizie możemy wyciągnąć kilka wniosków. Możemy również przetestować tezę, iż wybór funkcji systemowej nie jest zależny tylko i wyłącznie od wydajności, czasu czy stabilności jej działania, lecz od głównego kontekstu użycia.

W przypadku testów wydajnościowych najlepiej wychodzi funkcja kmalloc_array a najgorzej standardowe kmalloc. Nie determinuje to jednak wyboru, gdyż te dwie funkcje, pomimo, że są bliźniacze stworzone są do czegoś innego. Kmalloc to standardowa alokacja, podczas gdy kcalloc_array to alokacja dla tablic. Stąd na tym polu mamy niejednoznaczność wyboru.

Jeśli chodzi o pustą inicjalizację, to tutaj każda z funkcji faktycznie zachowuje się tak jak jest to podane w dokumentacji. Mianowicie Kmalloc oraz Kmalloc_array zwracają pustą inicjalizację, podczas gdy Kcalloc zwraca tablicę zainicjalizowaną zerami. Jeśli chodzi o wydajność to Kcalloc nieznacznie ustępuje jego bliźniakowi Kmalloc_array, natomiast ten pierwszy gwarantuje poprawność tzw. pustej inicjalizacji – stąd do alokacji danych tablicowych naszym subiektywnym zdaniem lepiej sprawdzi się Kcalloc.

W przypadku zbyt dużego przydziału pamięci przez funkcję, każda z nich posiada – jak wykazaliśmy powyżej sprawny mechanizm zabezpieczający przed alokacją maksymalnego bloku pamięci. Co chroni system przed np. brakiem pamięci na inne procesy (w tym systemowe).

Pomimo, że kcalloc jest szybszy w alokacji bardzo małych (10 elementów) i bardzo dużych tablic (1000 elementów), kmalloc_array jest ogólnie bardziej wydajny w zwalnianiu pamięci i w alokacji średnich rozmiarów tablic (100 elementów). To czyni kmalloc_array bardziej wszechstronnym wyborem w przypadku, gdy zwalnianie pamięci jest częste i alokacje mają różny rozmiar. Chociaż kcalloc jest szybki w alokacji, jego czasy zwalniania pamięci są znacznie dłuższe niż kmalloc_array, szczególnie dla bardzo dużych tablic. To może prowadzić do gorszej ogólnej wydajności w systemach, które często alokują i zwalniają pamięć.

Stąd jeśli zależy nam na poprawności inicjalizacji początkowej – skorzystajmy z Kcalloc, który zagwarantuje nam inicjalizację zerami. Natomiast jeśli zależy nam złożoności czasowej, to powinniśmy skorzystać z Kmalloc_array.

Powyższe rozważanie potwierdzają tezę, że nie ma jednoznacznego wyboru, który będzie wyraźnie lepszy od drugiego. Każda z tych trzech funkcji – choć bliźniacze, to są stworzone do nieco innych zastosowań, co powinniśmy mieć na uwadze, przy podejmowaniu wyboru.

Kod

KOD 1. WYDAJNOŚĆ ALOKACJI DUŻYCH BLOKÓW PAMIĘCI (JIFIES)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/jiffies.h>

MODULE_DESCRIPTION("Test wydajności alokacji pamięci: kmalloc, kcalloc,
kmalloc_array");

#define NUM_ALLOC 1000000
#define ALLOC_SIZE 32

static int __init test_performance_allocation_init(void)
{
    char *kmalloc_buf;
    int *kcalloc_buf;
    int *kmalloc_array_buf;
    unsigned long start, end;
    int i;

    printk(KERN_INFO "Test wydajności alokacji pamięci\n");

    // Test kmalloc
    start = jiffies;
    for (i = 0; i < NUM_ALLOC; i++) {
        kmalloc_buf = kmalloc(ALLOC_SIZE, GFP_KERNEL);
        kfree(kmalloc_buf);
    }
    end = jiffies;
    printk(KERN_INFO "kmalloc: %u jiffies\n", (unsigned int)(end - start));

    // Test kcalloc
    start = jiffies;
    for (i = 0; i < NUM_ALLOC; i++) {
        kcalloc_buf = kcalloc(ALLOC_SIZE, sizeof(int), GFP_KERNEL);
        kfree(kcalloc_buf);
    }
    end = jiffies;
    printk(KERN_INFO "kcalloc: %u jiffies\n", (unsigned int)(end - start));

    // Test kmalloc_array
    start = jiffies;
    for (i = 0; i < NUM_ALLOC; i++) {
        kmalloc_array_buf = kmalloc_array(ALLOC_SIZE, sizeof(int),
GFP_KERNEL);
        kfree(kmalloc_array_buf);
```

```
    }
    end = jiffies;
    printk(KERN_INFO "kmalloc_array: %u jiffies\n", (unsigned int)(end -
start));

    return 0;
}

static void __exit test_performance_allocation_exit(void)
{
    printk(KERN_INFO "Test wydajności zakończony\n");
}

module_init(test_performance_allocation_init);
module_exit(test_performance_allocation_exit);
```

KOD 2. TEST PUSTEJ INICJALIZACJI - ZWRÓCONA WARTOŚĆ

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>

MODULE_DESCRIPTION("Test nieinicjalizowanej pamięci: kmalloc, kcalloc,
kmalloc_array");

#define ALLOC_SIZE 1024

static void test_kmalloc(void)
{
    char *kmalloc_buf;

    printk(KERN_INFO "Test kmalloc bez inicjalizacji\n");

    kmalloc_buf = kmalloc(ALLOC_SIZE, GFP_KERNEL);
    if (!kmalloc_buf)
    {
        printk(KERN_ERR "kmalloc failed\n");
        return;
    }

    // Próba odczytu z niezainicjalizowanej pamięci
    printk(KERN_INFO "Niezainicjalizowana pamięć (kmalloc): %s\n",
kmalloc_buf);

    // Zwalnianie pamięci
    kfree(kmalloc_buf);
}
```

```c
static void test_kcalloc(void)
{
    int *kcalloc_buf;

    printk(KERN_INFO "Test kcalloc bez inicjalizacji\n");

    kcalloc_buf = kcalloc(ALLOC_SIZE / sizeof(int), sizeof(int),
GFP_KERNEL);
    if (!kcalloc_buf)
    {
        printk(KERN_ERR "kcalloc failed\n");
        return;
    }

    // Próba odczytu z niezainicjalizowanej pamięci
    printk(KERN_INFO "Niezainicjalizowana pamięć (kcalloc): %d\n",
kcalloc_buf[0]);

    // Zwalnianie pamięci
    kfree(kcalloc_buf);
}

static void test_kmalloc_array(void)
{
    int *kmalloc_array_buf;

    printk(KERN_INFO "Test kmalloc_array bez inicjalizacji\n");

    kmalloc_array_buf = kmalloc_array(ALLOC_SIZE / sizeof(int),
sizeof(int), GFP_KERNEL);
    if (!kmalloc_array_buf)
    {
        printk(KERN_ERR "kmalloc_array failed\n");
        return;
    }

    // Próba odczytu z niezainicjalizowanej pamięci
    printk(KERN_INFO "Niezainicjalizowana pamięć (kmalloc_array): %d\n",
kmalloc_array_buf[0]);

    // Zwalnianie pamięci
    kfree(kmalloc_array_buf);
}

static int __init test_uninitialized_memory_init(void)
{
    test_kmalloc();
    test_kcalloc();
    test_kmalloc_array();
    return 0;
```

```
}

static void __exit test_uninitialized_memory_exit(void)
{
    printk(KERN_INFO "Test nieinicjalizowanej pamięci zakończony\n");
}

module_init(test_uninitialized_memory_init);
module_exit(test_uninitialized_memory_exit);
```

KOD 3. TEST MAKSYMALNEJ ALOKACJI (SUKCES/BŁĄD)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>


static int __init test_max_allocation_init(void)
{
    char *kmalloc_buf;
    int *kcalloc_buf;
    int *kmalloc_array_buf;
    size_t max_size = SIZE_MAX;

    printk(KERN_INFO "Test maksymalnej alokacji pamięci\n");

    // Test kmalloc
    kmalloc_buf = kmalloc(max_size, GFP_KERNEL);
    if (!kmalloc_buf) {
        printk(KERN_ERR "kmalloc failed for max size\n");
    } else {
        printk(KERN_INFO "kmalloc succeeded for max size\n");
        kfree(kmalloc_buf);
    }

    // Test kcalloc
    kcalloc_buf = kcalloc(max_size, sizeof(int), GFP_KERNEL);
    if (!kcalloc_buf) {
        printk(KERN_ERR "kcalloc failed for max size\n");
    } else {
        printk(KERN_INFO "kcalloc succeeded for max size\n");
        kfree(kcalloc_buf);
    }

    // Test kmalloc_array
    kmalloc_array_buf = kmalloc_array(max_size, sizeof(int), GFP_KERNEL);
    if (!kmalloc_array_buf) {
        printk(KERN_ERR "kmalloc_array failed for max size\n");
```

```
    } else {
        printk(KERN_INFO "kmalloc_array succeeded for max size\n");
        kfree(kmalloc_array_buf);
    }

    return 0;
}

static void __exit test_max_allocation_exit(void)
{
    printk(KERN_INFO "Test maksymalnej alokacji zakończony\n");
}

module_init(test_max_allocation_init);
module_exit(test_max_allocation_exit);
```

KOD 4. CZASY ALOKACJI I ZWALNIANIA PAMIĘCI W ZALEŻNOŚCI OD ROZMIARU TABLICY N

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/time.h>

MODULE_DESCRIPTION("Test wydajnościowy alokacji pamięci: kmalloc, kcalloc,
kmalloc_array");

#define NUM_ALLOCS 1000 // Liczba alokacji
#define BLOCK_SIZE 256  // Rozmiar bloków w bajtach

static void test_kcalloc(void)
{
    int i;
    int *buf[NUM_ALLOCS];
    ktime_t start, end;
    s64 total_time_alloc = 0, total_time_free = 0;

    printk(KERN_INFO "Rozpoczęcie testu wydajnościowego alokacji pamięci:
kcalloc\n");

    start = ktime_get();
    for (i = 0; i < NUM_ALLOCS; i++)
    {
        buf[i] = kcalloc(BLOCK_SIZE / sizeof(int), sizeof(int),
GFP_KERNEL);
        if (!buf[i])
        {
```

```
            printk(KERN_ERR "Alokacja pamięci nie powiodła się na iteracji
%d za pomocą kcalloc\n", i);
            while (i--)
            {
                kfree(buf[i]);
            }
            return;
        }
    }
    end = ktime_get();
    total_time_alloc = ktime_to_ns(ktime_sub(end, start));
    printk(KERN_INFO "Czas alokacji %d bloków za pomocą kcalloc: %lld
ns\n", NUM_ALLOCS, total_time_alloc);

    start = ktime_get();
    for (i = 0; i < NUM_ALLOCS; i++)
    {
        kfree(buf[i]);
    }
    end = ktime_get();
    total_time_free = ktime_to_ns(ktime_sub(end, start));
    printk(KERN_INFO "Czas zwalniania %d bloków za pomocą kcalloc: %lld
ns\n", NUM_ALLOCS, total_time_free);

    printk(KERN_INFO "Zakończenie testu wydajnościowego alokacji pamięci:
kcalloc\n");
}

static void test_kmalloc_array(void)
{
    int i;
    int *buf[NUM_ALLOCS];
    ktime_t start, end;
    s64 total_time_alloc = 0, total_time_free = 0;

    printk(KERN_INFO "Rozpoczęcie testu wydajnościowego alokacji pamięci:
kmalloc_array\n");

    start = ktime_get();
    for (i = 0; i < NUM_ALLOCS; i++)
    {
        buf[i] = kmalloc_array(BLOCK_SIZE / sizeof(int), sizeof(int),
GFP_KERNEL);
        if (!buf[i])
        {
            printk(KERN_ERR "Alokacja pamięci nie powiodła się na iteracji
%d za pomocą kmalloc_array\n", i);
            while (i--)
            {
                kfree(buf[i]);
            }
```

```
            return;
        }
    }
    end = ktime_get();
    total_time_alloc = ktime_to_ns(ktime_sub(end, start));
    printk(KERN_INFO "Czas alokacji %d bloków za pomocą kmalloc_array: %lld
ns\n", NUM_ALLOCS, total_time_alloc);

    start = ktime_get();
    for (i = 0; i < NUM_ALLOCS; i++)
    {
        kfree(buf[i]);
    }
    end = ktime_get();
    total_time_free = ktime_to_ns(ktime_sub(end, start));
    printk(KERN_INFO "Czas zwalniania %d bloków za pomocą kmalloc_array:
%lld ns\n", NUM_ALLOCS, total_time_free);

    printk(KERN_INFO "Zakończenie testu wydajnościowego alokacji pamięci:
kmalloc_array\n");
}

static int __init performance_test_init(void)
{
    printk(KERN_INFO "Rozpoczęcie testu wydajnościowego alokacji
pamięci\n");

    test_kcalloc();
    test_kmalloc_array();

    printk(KERN_INFO "Zakończenie testu wydajnościowego alokacji
pamięci\n");

    return 0;
}

static void __exit performance_test_exit(void)
{
    printk(KERN_INFO "Moduł testowy wydajnościowy został usunięty\n");
}

module_init(performance_test_init);
module_exit(performance_test_exit);
```

KOD 5. PLIK MAKEFILE

```
obj-m += test_max_allocation.o
obj-m += test_non_initialized.o
obj-m += test_performance_allocation.o
obj-m += test_times.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Zebrane dane do testów – wyjścia z terminala

OUTPUT *1*. WYDAJNOŚĆ ALOKACJI DUŻYCH BLOKÓW PAMIĘCI (JIFIES)

```
Test wydajności alokacji pamięci
kmalloc: 52 jiffies
kcalloc: 31 jiffies
kmalloc_array: 28 jiffies
```

OUTPUT *2*. TEST PUSTEJ INICJALIZACJI - ZWRÓCONA WARTOŚĆ

```
Test kmalloc bez inicjalizacji
Niezainicjalizowana pamięć (kmalloc):
Test kcalloc bez inicjalizacji
Niezainicjalizowana pamięć (kcalloc): 0
Test kmalloc_array bez inicjalizacji
Niezainicjalizowana pamięć (kmalloc_array):
```

OUTPUT 3. TEST MAKSYMALNEJ ALOKACJI (SUKCES/BŁĄD)

```
RAX: ffffffffffffffda RBX: 0000580e88264740 RCX: 0000766d4cd2725d
RDX: 0000000000000000 RSI: 0000580e864d2e52 RDI: 0000000000000003
RBP: 00007ffd1e52c540 R08: 0000000000000040 R09: 0000000000000000
R10: 0000766d4ce03b20 R11: 0000000000000246 R12: 0000580e864d2e52
R13: 0000000000000000 R14: 0000580e88264700 R15: 0000000000000000
</TASK>
---[ end trace 0000000000000000 ]---
kmalloc failed for max size
kcalloc failed for max size
kmalloc_array failed for max size
```

OUTPUT 4. CZASY ALOKACJI I ZWALNIANIA PAMIĘCI W ZALEŻNOŚCI OD ROZMIARU TABLICY N

```
Czas alokacji 10 bloków za pomocą kcalloc: 554 ns
Czas zwalniania 10 bloków za pomocą kcalloc: 829 ns
Zakończenie testu wydajnościowego alokacji pamięci: kcalloc
Rozpoczęcie testu wydajnościowego alokacji pamięci: kmalloc_array
Czas alokacji 10 bloków za pomocą kmalloc_array: 674 ns
Czas zwalniania 10 bloków za pomocą kmalloc_array: 386 ns
Zakończenie testu wydajnościowego alokacji pamięci: kmalloc_array
Zakończenie testu wydajnościowego alokacji pamięci

Czas alokacji 100 bloków za pomocą kcalloc: 12844 ns
Czas zwalniania 100 bloków za pomocą kcalloc: 6313 ns
Zakończenie testu wydajnościowego alokacji pamięci: kcalloc
Rozpoczęcie testu wydajnościowego alokacji pamięci: kmalloc_array
Czas alokacji 100 bloków za pomocą kmalloc_array: 10887 ns
```

```
Czas zwalniania 100 bloków za pomocą kmalloc_array: 6154 ns
Zakończenie testu wydajnościowego alokacji pamięci: kmalloc_array
Zakończenie testu wydajnościowego alokacji pamięci

Czas alokacji 1000 bloków za pomocą kcalloc: 147357 ns
Czas zwalniania 1000 bloków za pomocą kcalloc: 188358 ns
Zakończenie testu wydajnościowego alokacji pamięci: kcalloc
Rozpoczęcie testu wydajnościowego alokacji pamięci: kmalloc_array
Czas alokacji 1000 bloków za pomocą kmalloc_array: 173215 ns
Czas zwalniania 1000 bloków za pomocą kmalloc_array: 94821 ns
Zakończenie testu wydajnościowego alokacji pamięci: kmalloc_array
Zakończenie testu wydajnościowego alokacji pamięci
```

Spis literatury

1. Artykuł bazowy: **Understanding and Mitigating Twin Function Misuses in Operating System Kernel**
2. Dokumentacja: The Linux Kernel: **Memory Allocation Guide.** https://www.kernel.org/doc/html/v5.0/core-api/memory-allocation.html
3. Zrozumienie flag alokacji: https://www.chiark.greenend.org.uk/doc/linux-doc-3.2/html/kernel-api/API-kmalloc-array.html
4. Dokumentacja: **Memory Allocation Guide.** https://www.kernel.org/doc/html/next/core-api/memory-allocation.html