Laboratorium 3: „Struktury danych jądra Linuksa” (dwa zajęcia)

dr inż. Arkadiusz Chrobot

16 marca 2024

Spis treści

Wprowadzenie

W tej instrukcji zawarto opisy gotowych implementacji struktur danych, które są używane w kodzie jądra Linuksa i które zostały udostępnione do wykorzystania między innymi dla programistów tworzących moduły jądra. Rozdział 1 zawiera informacje, opis API oraz przykład ilustrujący wykorzystanie implementacji dwukierunkowych list. Rozdział 2 opisuje implementację kolejki FIFO wraz z jej interfejsem i przykładowym modułem, który z niej korzysta. Rozdziały 3 oraz 4 opisują budowę, API oraz przykłady wykorzystania odpowiednio drzew czerwono-czarnych oraz drzew pozycyjnych. Rozdział 5 krótko wymienia inne struktury danych, które są dostępne w przestrzeni jądra Linuksa, ale nie są bardziej szczegółowo opisane w tej instrukcji. W ostatnim rozdziale zamieszczono zadania do samodzielnego wykonania.

1. Listy dwukierunkowe

Jądro Linuksa dostarcza uniwersalnej implementacji listy, z której korzystają programiści tworzący główny kod jądra, a także programiści piszący moduły. Eliminuje to konieczność opracowywania kolejnych wersji tej struktury dla poszczególnych podsystemów, zmniejsza koszty utrzymania kodu, a także ułatwia pracę programistom.

1.1. Opis API

Implementacja listy w jądrze Linuksa jest oparta na dwukierunkowej liście cyklicznej i dostępna jest po włączeniu do kodu pliku `linux/list.h`. Aby utworzyć listę struktur określonego typu należy w definicji tego typu umieścić pole typu `struct list_head`, które zawiera dwa wskaźniki tego samego typu o nazwach `next` i `prev`. Do obsługi listy przeznaczone są następujące makra i funkcje:

`void INIT_LIST_HEAD(struct list_head *list)` - funkcja, która służy do inicjacji wskaźników listy w jej pojedynczym elemencie. Przekazywany jest do niej wskaźnik na strukturę zawierającą te wskaźniki.

`LIST_HEAD(name)` - makro, które służy do tworzenia głównego elementu listy, który stanowi jej „uchwyt”. Przekazywana jest do niego nazwa tego elementu.

`list_entry(ptr, type, member)` - makro, które pozwala uzyskać wskaźnik na strukturę, w której zawarte jest pole typu `struct head_list`. Pierwszym jego argumentem jest wskaźnik na strukturę typu `struct head_list`, drugim nazwa typu struktury zawierającej wspomniane pole, a trzecim nazwa pola typu `struct list_head`.

`void list_add(struct list_head *new, struct list_head *head)` - funkcja, która dodaje nowy element listy tuż za jej głównym elementem. Dzięki temu ta lista może być użyta w roli stosu. Jako pierwszy argument funkcja przyjmuje wskaźnik na strukturę wskaźników zawartą w nowym elemencie listy, a jako drugi wskaźnik na główny element listy.

`void list_add_tail(struct list_head *new, struct list_head *head)` - funkcja, która dodaje nowy element listy tuż przed jej głównym elementem. Dzięki temu ta lista może być użyta w roli kolejki FIFO. Jako pierwszy argument funkcja przyjmuje wskaźnik na strukturę wskaźników zawartą w nowym elemencie listy, a jako drugi wskaźnik na główny element listy.

`void list_del(struct list_head *entry)` - funkcja, która usuwa element z listy. Nie oznacza to, że pamięć na ten element jest zwalniana! Przyjmuje jako argument wywołania wskaźnik na pole typu `struct list_head` usuwanego elementu.

`void list_del_init(struct list_head *entry)` - funkcja, która usuwa element z listy i przeprowadza ponowną inicjację jego wskaźników. Przyjmuje jako argument wywołania wskaźnik na pole typu `struct list_head` zwalnianego elementu.

`void list_move(struct list_head *list, struct list_head *head)` - funkcja usuwa z listy element i umieszcza go na innej liście. Pierwszym argumentem jej wywołania jest wskaźnik na pole typu `struct list_head` tego przenoszonego elementu, a drugim wskaźnik na główny element docelowej listy. Przenoszony element wstawiany jest tuż za elementem głównym.

`void list_move_tail(struct list_head *list, struct list_head *head)` - funkcja usuwa z listy element i umieszcza go na innej liście. Pierwszym argumentem jej wywołania jest wskaźnik na pole typu `struct list_head` tego przenoszonego elementu, a drugim wskaźnik na główny element docelowej listy. Przenoszony element wstawiany jest tuż przed elementem głównym.

`int list_empty(const struct list_head *head)` - funkcja sprawdza, czy lista jest pusta. Jeśli tak jest, to zwraca ona wartość różną od zera, a zero w przeciwnym przypadku. Jako argument wywołania przyjmuje wskaźnik na główny element listy.

`void list_splice(const struct list_head *list, struct list_head *head)` - funkcja łączy ze sobą dwie listy. Lista, której główny element jest wskazywany przez pierwszy parametr funkcji jest dołączana w całości tuż za głównym elementem drugiej listy, na który wskaźnik jest przekazywany przez drugi parametr funkcji. Może służyć do łączenia list używanych jako stosy.

`void list_splice_tail(struct list_head *list, struct list_head *head)` - funkcja łączy ze sobą dwie listy. Lista, której główny element jest wskazywany przez pierwszy parametr funkcji jest dołączana w całości tuż przed elementem drugiej listy, na który wskaźnik jest przekazywany przez drugi parametr funkcji. Funkcja może być użyta do łączenia list używanych jako kolejki FIFO.

`void list_splice_init(struct list_head *list, struct list_head *head)` - funkcja służ do łączenia ze sobą dwóch list, tak jak `list_splice()`, ale dołączana lista jest opróżniana i ponownie inicjowana.

`list_for_each(pos, head)` - makro pozwalające przeglądać listę, inaczej *iterator*. Pierwszym argumentem tego marka jest wskaźnik typu `struct list_head *`, którym będą wskazywane kolejne elementy listy, w kolejnych iteracjach. Drugim jest wskaźnik na główny element listy.

`list_for_each_entry(pos, head, member)` - makro, które pozwala, tak jak `list_for_each` przeglądać listę, ale przy pomocy wskaźnika na właściwą strukturę jej elementów. Ten wskaźnik jest przekazywany do niego jako pierwszy argument. Drugim argumentem jest wskaźnik na pierwszy element listy, a ostatnim nazwa pola typu `struct list_head` w elemencie.

`list_for_each_entry_reverse(pos, head, member)` - makro przyjmuje te same argumenty co opisane wcześniej `list_for_each_entry()`, ale przegląda listę w odwrotnym kierunku.

`list_for_each_safe(pos, n, head)` - makro pozwala przeglądać listę i usuwać z niej elementy. Pierwszym jego argumentem jest wskaźnik na element listy, którym będą wskazywane kolejne elementy listy podczas kolejnych iteracji. Drugi argument jest tego samego typu co pierwszy i służy on do tymczasowego przechowywania adresu elementów listy, a trzeci jest wskaźnikiem na główny element listy.

`list_for_each_entry_safe(pos, n, head, member)` - makro działa podobnie jak `list_for_each_entry`, ale pozwala podczas przeglądania na usuwanie elementów listy. Przyjmuje te same argumenty co wspomniane makro, ale ma jeszcze jeden dodatkowy argument, który jest przekazywany mu jako drugi w kolejności. Jest to wskaźnik na pojedynczy element listy, który służy do tymczasowego przechowywania adresów elementów listy.

`list_for_each_entry_safe_reverse(pos, n, head, member)` - makro, tak jak `list_for_each_entry_safe` pozwala usuwać elementy z listy podczas jej przeglądania, ale przegląda ją w odwrotnym kierunku.

Istnieją również inne funkcje i makra związane z obsługą listy, ale nie będą one w tej instrukcji opisywane.

1.2. Przykład

Listing 1 zawiera kod źródłowy modułu korzystającego z implementacji listy dostępnej w jądrze Linuksa.

Listing 1: Moduł korzystający z listy

```
#include<linux/module.h>
#include<linux/list.h>
#include<linux/slab.h>
#include<linux/random.h>

struct example_struct
{
        int random_value;
        struct list_head list_element;
};

static LIST_HEAD(head);

static int __init listmod_init(void)
{
        struct example_struct *element;
        struct list_head *entry;
        u8 i;

        for(i=0;i<4;i++) {
                element = (struct example_struct *)
                        kmalloc(sizeof(struct example_struct),GFP_KERNEL);
                if(!IS_ERR(element)) {
                        get_random_bytes((int *)&element->random_value,
                                         sizeof(element->random_value));
                        INIT_LIST_HEAD(&element->list_element);
                        list_add_tail(&element->list_element,&head);
                }
        }
        list_for_each(entry, &head) {
                element = list_entry(entry, struct example_struct, list_element);
                pr_notice("Element's value: %u\n",element->random_value);
        }

        return 0;
}
```

```
37
38 static void __exit listmod_exit(void)
39 {
40         struct example_struct *element, *next;
41
42          list_for_each_entry_safe(element, next, &head, list_element) {
43                  list_del(&element->list_element);
44                  pr_notice("Element's value: %u\n",element->random_value);
45                  kfree(element);
46          }
47 }
48
49 module_init(listmod_init);
50 module_exit(listmod_exit);
51
52 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
53 MODULE_LICENSE("GPL");
54 MODULE_DESCRIPTION("An exemplary kernel module that demonstrates the usage of a kernel list.");
55 MODULE_VERSION("1.0");
```

Do kodu źródłowego modułu z listingu 1 dołączane są cztery pliki nagłówkowe. Pierwszy z nich (wiersz nr 1) zawiera definicje funkcji i mark do obsługi modułów, w drugim zdefiniowano (wiersz nr 2) funkcje i makra do obsługi list. Trzeci (wiersz nr 3) zwiera definicję funkcji `kmalloc()`, która będzie służyła do dynamicznego przydziału pamięci na elementy listy, a czwarty (wiersz nr 4) zawiera definicję funkcji `get_random_bytes()`, której dokładniejszy opis będzie zamieszczony później. Służy ona do uzyskiwania losowych wartości z generatora liczb losowych jądra[1]. Wiersze 6-10 zawierają definicję typu przykładowej struktury elementów listy. Zawiera ona pole na dane (`random_value`) oraz pole typu `struct list_head`, ze wskaźnikami listy, dzięki któremu elementy będą mogły być połączone w listę. W 12 wierszu deklarowany jest główny element listy.

W konstruktorze modułu tworzona jest lista, a następnie zawartość pola `random_value` jej elementów jest zapisywana do bufora jądra. W wierszu 16 tej funkcji jest deklarowany wskaźnik na pojedynczy element listy, w wierszu 17 umieszczona jest deklaracja wskaźnika na strukturę typu `struct list_head`, a w 18 na licznik pętli `for`. W tej pętli (wiersze 20-29) przydzielana jest pamięć na cztery elementy listy (wiersze 21-22). Jeśli przydział się powiódł, to inicjowane jest pole `random_value` (wiersz 24) każdego takiego elementu losową wartością pobraną z generatora liczb losowych przy pomocy funkcji `get_random_bytes()`. Ta funkcja nic nie zwraca, ale przyjmuje dwa argumenty wywołania. Pierwszym jest adres zmiennej, do której mam być zapisana wartość, drugim natomiast liczba bajtów, które mają być odczytane z generatora. W przypadku opisywanego modułu jest ona równa rozmiarowi pola `random_value`. Po inicjacji tego pola jest inicjowane pole `list_element`, a następnie cały element jest dodawany do listy, której głównym elementem jest `head`. W wierszach 30-33 lista jest przeglądana przy pomocy makra `list_for_each`, a wartości pól `random_value` każdego jej elementu zapisywane do bufora jądra. Proszę zwrócić uwagę na wiersz 31, w którym uzyskiwany jest wskaźnik na element listy przy pomocy makra `list_entry`. Ten fragment funkcji inicjującej ilustruje również w jaki sposób poprawnie korzystać z makra `list_for_each`.

W destruktorze modułu lista jest przeglądana za pomocą makra `list_for_each_entry_safe`. W trakcie przeglądania kolejne elementy z listy są usuwane, zawartości ich pola `random_value` trafia do bufora jądra, a pamięć przydzielona na element jest zwalniana przy pomocy funkcji `kfree()`. Proszę zwrócić uwagę na deklaracje wskaźników `element` i `next` w 40 wierszu modułu oraz na sposób ich przekazania do wspomnianego makra.

2. Kolejki FIFO

Kolejną strukturą danych, która została w sposób uniwersalny zaimplementowana w jądrze Linuksa jest kolejka `fifo`. Wprawdzie można taką strukturę uzyskać stosują opisaną wcześniej dwukierunkową listę cykliczną, ale osobna implementacja tej kolejki jest bardziej efektywna, szczególnie jeśli jest ona

[1] Formalnie jest to generator liczb pseudolosowych, ale spełniający wymogi bezpieczeństwa dla zastosowań kryptograficznych, stąd dla odróżnienia od typowych generatorów liczb pseudolosowych można go nazwać generatorem liczb losowych.

używana w zagadnieniu sprowadzającym się do problemu producenta i konsumenta. Implementacja kolejki FIFO w jądrze Linuksa jest oparta na cyklicznie indeksowanej tablicy. Pozwala ona na zapisywanie i odczytywanie elementów o zmiennej wielkości.

2.1. Opis API

Aby użyć uniwersalnej implementacji kolejki FIFO należy włączyć plik nagłówkowy `linux/kfifo.h` oraz zdefiniować zmienną typu `struct kfifo`. Do obsługi tej zmiennej, przeznaczone są następujące makra:

`kfifo_alloc(fifo, size, gfp_mask)` - makro, które dynamicznie przydziela pamięć na kolejkę FIFO Jego pierwszym argumentem jest wskaźnik na strukturę typu `struct kfifo`. Drugim jest rozmiar wszystkich elementów kolejki, który musi być wyrażony potęgą dwójki. Trzeci argument to znacznik typu przydziału. Makro zwraca zero jeśli alokacja zakończy się pomyślnie lub kod błędu w przypadku niepowodzenia przydziału.

`kfifo_init(fifo, buffer, size)` - makro pozwala zainicjować kolejkę `fifo` w buforze, na który pamięć została przydzielona wcześniej. Jest ono używane zamiast `kfifo_alloc`. Pierwszym argumentem tego makra jest wskaźnik na strukturę `struct kfifo`, drugim wskaźnik typu `void *` na bufor, a trzecim rozmiar bufora, który musi być wyrażony potęgą dwójki. Makro zwraca zero w przypadku pomyślnego zakończenia inicjacji kolejki lub kod błędu w przeciwnym przypadku.

`DECLARE_KFIFO(fifo, type, size)` - makro, które pozwala statycznie, czyli podczas kompilacji, stworzyć kolejkę FIFO o określonym rozmiarze. Pierwszym parametrem tego makra jest nazwa tej kolejki, drugim nazwa typu pojedynczego elementu, a trzecim rozmiar, który musi być wyrażony potęgą dwójki.

`INIT_KFIFO(fifo)` - makro, które służy do inicjowania kolejki stworzonej przy pomocy `DECLARE_FIFO`. Jego jedynym argumentem jest nazwa tej kolejki FIFO.

`kfifo_in(fifo, buf, n)` - makro, które pozwala dodać element (lub elementy) do kolejki. Pierwszym jego argumentem jest wskaźnik do kolejki FIFO (wskaźnik typu `struct kfifo`), drugim jest wskaźnik na element, a trzecim rozmiar tego elementu. Jeśli dodanie do kolejki się powiedzie, to makro zwróci rozmiar dodanego elementu.

`kfifo_out(fifo, buf, n)` - makro pozwala zdjąć element (lub elementy) z kolejki. Jako pierwszy argument przyjmuje wskaźnik na kolejkę FIFO, jako drugi wskaźnik na zmienną, gdzie ma być zapisana wartość elementu, a jako trzeci rozmiar zdejmowanego elementu. W przypadku powodzenia operacji zdejmowania makro zwróci rozmiar zdjętego elementu.

`kfifo_peek(fifo, val)` - makro pozwala odczytać wartość elementu (lub elementów) bez jego zdejmowania z kolejki. Jako pierwszy argument przyjmuje wskaźnik na kolejkę FIFO, jako drugi wskaźnik na zmienną, gdzie wartość tego elementu ma być zapisana. Makro w przypadku powodzenia operacji zwraca rozmiar odczytanego elementu.

`kfifo_size(fifo)` - makro zwraca rozmiar kolejki FIFO. Jako jedyny argument przyjmuje wskaźnik na tę kolejkę.

`kfifo_len(fifo)` - makro zwraca ilość zajętego miejsca w kolejce, na którą wskaźnik jest mu przekazywany jako jedyny jego argument.

`kfifo_avail(fifo)` - makro zwraca ilość wolnego miejsca w kolejce, na którą wskaźnik jest mu przekazywany jako jedyny jego argument.

`kfifo_is_empty(fifo)` - makro zwraca wartość różną od zera (prawdę), jeśli kolejka, do której wskaźnik jest mu przekazany jako pierwszy argument, jest pusta.

`kfifo_is_full(fifo)` - makro zwraca wartość różną od zera (prawdę), jeśli kolejka, do której wskaźnik jest mu przekazany jako pierwszy argument, jest pełna.

`kfifo_reset(fifo)` - usuwa całą zawartość kolejki FIFO, na którą wskaźnik jest mu przekazany jako jedyny argument.

`kfifo_free(fifo)` - makro zwalnia pamięć przydzieloną na kolejkę FIFO, która była przydzielona przez makro `kfifo_alloc`.

Istnieją również inne makra związane z obsługą kolejki FIFO, które nie będą tu opisywane.

2.2. Przykład

Listing 2 zawiera kod źródłowy modułu, który tworzy kolejkę FIFO i wypełnia ją losowymi liczbami naturalnymi.

Listing 2: Moduł korzystający z kolejki FIFO

```
#include<linux/module.h>
#include<linux/random.h>
#include<linux/kfifo.h>

#define NUMBER_OF_ELEMENTS 8

static struct kfifo fifo_queue;

static int __init fifomod_init(void)
{
        int returned_value;
        u32 random_value;
        unsigned int returned_size;

        returned_value =
                kfifo_alloc(&fifo_queue,NUMBER_OF_ELEMENTS*sizeof(random_value),GFP_KERNEL);
        if(returned_value) {
                pr_alert("Error allocating kfifo!\n");
                return -ENOMEM;
        }
        while(!kfifo_is_full(&fifo_queue)) {
                get_random_bytes(&random_value,sizeof(random_value));
                random_value %= 100;
                returned_size = kfifo_in(&fifo_queue,&random_value,sizeof(random_value));
                if(returned_size!=sizeof(random_value))
                        pr_alert("Enqueue error\n");
        }
        returned_size = kfifo_out_peek(&fifo_queue,&random_value,sizeof(random_value));
        if(returned_size!=sizeof(returned_size))
                pr_alert("Error peeking element form the queue!\n");
        pr_notice("Value of the first element in the queue: %u\n",random_value);

        return 0;
}

static void __exit fifomod_exit(void)
{
        int returned_size;
        u32 value;

        while(!kfifo_is_empty(&fifo_queue)) {
                returned_size = kfifo_out(&fifo_queue,&value,sizeof(value));
                if(returned_size!=sizeof(value))
                        pr_alert("Dequeue error!\n");
                pr_notice("Value from the queue: %u\n",value);
```

```
46          }
47
48          kfifo_free(&fifo_queue);
49  }
50
51  module_init(fifomod_init);
52  module_exit(fifomod_exit);
53
54  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
55  MODULE_LICENSE("GPL");
56  MODULE_DESCRIPTION("An exemplary module demonstrating the usage of a kernel FIFO queue.");
57  MODULE_VERSION("1.0");
```

W funkcji inicjującej moduł przydzielana jest pamięć na kolejkę, która pomieści osiem liczby typu `int`, a następnie losowane są liczby dwu i jednocyfrowe i zapisywane do tej kolejki w pętli `while`, która tak długo jest wykonywana, jak długo jest miejsce w kolejne na nowe elementy. Na koniec odczytywana jest wartość liczby znajdującej się na czole kolejki i umieszczana w buforze jądra. W funkcji badana jest poprawność wykonania każdej operacji na kolejce.

W funkcji sprzątającej modułu, w pętli `while` usuwane są kolejne elementy z czoła kolejki i ich wartość jest umieszczana w buforze jądra. Poprawność operacji usuwania jest za każdym razem kontrolowana. Wykonanie pętli kończy się w chwili opróżnienia kolejki FIFO. Potem zwalniana jest pamięć przydzielona na kolejkę.

Inne przykłady zastosowania kolejek można znaleźć w podkatalogu `sample/kfifo` znajdującym się w katalogu z kodami źródłowymi jądra.

3. Drzewa czerwono-czarne

Drzewa czerwono-czarne to wyważone drzewa BST. Wyważenie uzyskuje się nadając każdemu węzłowi drzewa dodatkowy atrybut, który nazywa się kolorem i zapewniając, że po każdym wstawieniu nowego elementu do drzewa lub usunięciu istniejącego elementu będą zachowane następujące reguły[2]:

1. Korzeń ma kolor czarny.
2. Każdy węzeł ma kolor czarny lub czerwony.
3. Potomkowie węzła czerwonego muszą być zawsze czarni.
4. Liście drzewa są czarne.
5. Wszystkie proste ścieżki od wybranego węzła do dowolnego podlegającego mu liścia zawierają tyle samo węzłów czarnych.

Jeśli po wykonaniu którejś z opisanych operacji te reguły nie są zachowane to za pomocą czynności polegających na wykonywaniu rotacji określonych poddrzew i/lub zmianie kolorów węzłów ich spełnienie jest przywracane.

3.1. Opis API

Ponieważ drzewa czerwono-czarne powinny być w jądrze systemu operacyjnego systemu zaimplementowane w sposób uniwersalny, to programiści Linuksa dostarczają tylko środków do przekształcania dowolnych struktur w węzły tego drzewa i do jego wyważania. Operacje wstawiania i wyszukiwania elementów w drzewie, podobne do tych stosowanych w drzewach BST należy zaimplementować samodzielnie[3].

[2] Różne źródła podają różne zestawy reguł. Zamieszczony w instrukcji zestaw pochodzi, oprócz reguły nr 1, która jest nadmiarowa w stosunku do reguły nr 4, z książki *Wprowadzenie do algorytmów* autorstwa T. H. Cormena, Ch. E. Leisersona i R. L. Rivesta, wydanie z 1998 roku. Jest on w pełni wystarczający do poprawnego działania takiej struktury.

[3] Informacje wraz z przykładami jak to zrobić można znaleźć między innymi w dokumentacji jądra, w pliku `linux/Documentation/rbtree.txt`.

Z drzew czerwono-czarnych można korzystać po włączeniu pliku nagłówkowego `linux/rbtree.h`. Podstawową strukturą dla drzewa czerwono-czarnego jest struktura typu `struct rb_node`. Należy ją zadeklarować jako pole w strukturze, która ma być umieszczona w drzewie czerwono-czarnym, na podobnej zasadzie jak umieszczaliśmy strukturę typu `struct list_head` w strukturze, która określa typ elementu listy. Struktura umieszczana w opisywanym drzewie musi posiadać także pole pełniące rolę klucza, tj. atrybutu, według którego będą porządkowane węzły drzewa. Drugą ważną strukturą jest struktura typu `struct rb_root`, która stanowi korzeń drzewa czerwono-czarnego. Do obsługi takiego drzewa zdefiniowano następujące makra i funkcje:

`RB_ROOT` - makro inicjuje korzeń drzewa czerwono-czarnego. Jego wartość należy przypisać do zmiennej typu `struct rb_root`.

`rb_entry(ptr, type, member)` - makro, które pozwala uzyskać wskaźnik na strukturę, w której zawarte jest pole typu `struct rb_node`. Pierwszym jego argumentem jest wskaźnik na strukturę typu `struct rb_node`, drugim nazwa typu struktury zawierającej wspomniane pole, a trzecim nazwa pola typu `struct rb_node`.

`struct rb_node *rb_first(const struct rb_root *)` - funkcja używana do przechodzenia (iterowania) węzłów drzewa czerwono-czarnego w porządku niemalejącym. Jako argument wywołania przyjmuje wskaźnik na korzeń drzewa (zmienną typu `struct rb_root *`), a zwraca wskaźnik na pole typu `struct rb_node` zawarte w pierwszym (skrajnie lewym) węźle drzewa.

`struct rb_node *rb_next(const struct rb_node *)` - funkcja wywoływana po `rb_first()`, w pętli. Przyjmuje jako argument wywołania wskaźnik na pole typu `struct rb_node` bieżącego elementu drzewa czerwono- czarnego, a zwraca wskaźnik na pole tego samego typu zawarte w kolejnym do odwiedzenia, według porządku niemalejącego, węźle. Jeśli taki węzeł nie istnieje, to funkcja zwróci `NULL`.

`struct rb_node *rb_last(const struct rb_root *)` - funkcja, która działa podobnie jak `rb_first()`, ale pozwala rozpocząć przechodzenie drzewa węzłów drzewa czerwono-czarnego w porządku nierosnącym. Zwraca wskaźnik na pole typu `struct rb_node` zawarte w ostatnim (skrajnie prawym) elemencie drzewa.

`struct rb_node *rb_prev(const struct rb_node *)` - funkcja, która jest odpowiedniczką `rb_next()` dla przechodzenia węzłów drzewa w porządku nierosnącym. Używa się jej tak samo, jak wymienionej funkcji.

`void rb_link_node(struct rb_node *node, struct rb_node *parent, struct rb_node **rb_link)` - funkcja, która pozwala dołączyć nowy węzeł do drzewa czerwono-czarnego. Przyjmuje ona trzy argumenty wywołania, ale nie zwraca żadnej wartości. Tymi argumentami wywołania są: wskaźnik na pole typu `struct rb_node` nowego węzła, wskaźnik na pole tego samego typu znajdujące się w rodzicu węzła, do którego ma być dowiązany nowy węzeł i które wskazuje na ten węzeł oraz wskaźnik na jedno z dwóch pól wskaźnikowych struktury typu `struct rb_node` (`rb_right` lub `rb_left`) węzła, do którego nowy węzeł ma zostać dowiązany.

`void rb_insert_color(struct rb_node *, struct rb_root *)` - funkcja, która sprawdzenia, czy drzewo czerwono-czarne wymaga wyważenia i w razie takiej konieczności przeprowadza tę operację. Jest ona wywoływana bezpośrednio po `rb_link_node()`. Przyjmuje dwa argumenty wywołania: wskaźnik na węzeł nowo wstawionego węzła drzewa i wskaźnik na korzeń drzewa.

`void rb_replace_node(struct rb_node *victim, struct rb_node *new, struct rb_root *root)` - funkcja pozwala zastąpić węzeł drzewa wskazywany przez pierwszy argument jej wywołania węzłem wskazywanym przez drugi argument jej wywołania. Trzecim argumentem jest wskaźnik na korzeń drzewa. **Oba węzły, zastępowany i zastępujący, powinny mieć ten sam klucz, czyli wartość, względem której są porządkowane w drzewie czerwono-czarnym.** Funkcja ta nie przeprowadza operacji wyważania, a więc nie należy za jej pomocą zastępować węzłów innymi węzłami, o innych kluczach.

`void rb_erase(struct rb_node *, struct rb_root *)` - funkcja, która usuwa z drzewa czerwono-czarnego węzeł wskazywany przez pierwszy argument jej wywołania. Nie zwalania jednak pamięci przydzielonej na ten węzeł. Drugim argumentem jej wywołania jest wskaźnik na korzeń drzewa.

Podobnie jak w przypadku poprzednich struktur ten opis API nie jest kompletny. Wymieniono w nim tylko najważniejsze funkcje i makra dotyczące obsługi drzew czerwono-czarnych.

3.2. Przykład

Listing 3 zawiera kod źródłowy przykładowego modułu jądra, który posługuje się drzewem czerwono-czarnym do przechowywania liczb naturalnych. Liczby te stanowią zarazem klucz, jak i jedyną wartość węzłów tego drzewa.

Listing 3: Moduł korzystający z drzewa czerwono-czarnego

```
#include<linux/module.h>
#include<linux/rbtree.h>
#include<linux/slab.h>

struct example_struct
{
        int data;
        struct rb_node node;
};

static struct rb_root root = RB_ROOT;

static struct example_struct *find_node(struct rb_root *root, int number)
{
        struct rb_node *node = root->rb_node;

        while(node) {
                struct example_struct *current_data = rb_entry(node, struct example_struct, node);

                if(number<current_data->data)
                        node = node->rb_left;
                else if(number>current_data->data)
                        node = node->rb_right;
                else
                        return current_data;
        }
        return NULL;
}

static bool insert_node(struct rb_root *root, struct example_struct *node)
{
        struct rb_node **new_node = &(root->rb_node), *parent = NULL;

        while(*new_node) {
                struct example_struct *this = rb_entry(*new_node, struct example_struct, node);

                parent = *new_node;
                if(node->data<this->data)
                        new_node = &((*new_node)->rb_left);
                else if(node->data>this->data)
                        new_node = &((*new_node)->rb_right);
                else return false;
        }

        rb_link_node(&node->node,parent,new_node);
        rb_insert_color(&node->node,root);
        return true;
}

```

```
50  static int __init rbtreemod_init(void)
51  {
52          int i;
53          struct example_struct *node = NULL;
54
55          for(i=0;i<5;i++) {
56                  node = (struct example_struct *)kmalloc(sizeof(struct example_struct), GFP_KERNEL);
57                  if(!IS_ERR(node)) {
58                          node->data = i;
59                          if(!insert_node(&root,node))
60                                  pr_alert("Error inserting new node!\n");
61
62                  } else
63                          pr_alert("Error allocating memory for new node: %ld\n",PTR_ERR(node));
64
65          }
66
67          for(i=0;i<5;i++) {
68                  node = find_node(&root,i);
69                  if(node)
70                          pr_notice("Value of the node: %d\n",node->data);
71                  else
72                          pr_alert("Error retrieving node from red-black tree!\n");
73          }
74
75          return 0;
76  }
77
78  static void __exit rbtreemod_exit(void)
79  {
80          int i;
81          struct example_struct *node = NULL;
82
83          for(i=0;i<5;i++) {
84                  node = find_node(&root,i);
85                  if(node) {
86                          rb_erase(&node->node, &root);
87                          kfree(node);
88                  } else
89                          pr_alert("Error retrieving node from red-black tree!\n");
90          }
91  }
92
93  module_init(rbtreemod_init);
94  module_exit(rbtreemod_exit);
95  MODULE_LICENSE("GPL");
96  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
97  MODULE_DESCRIPTION("A module that demonstrated the usage of a kernel red-black tree implementation.");
98  MODULE_VERSION("1.0");
```

W wierszach 5-9 zdefiniowano typ strukturalny opisujący budowę każdego węzła drzewa czerwono-czarnego. Jest w nim pole o nazwie `data` dla liczb, które będą kluczami i wartościami węzłów w tym drzewie oraz pole `node` typu `struct rb_node`, które zawiera wskaźniki `rb_right` i `rb_left`. Podobnie jak w przypadku list, to pole pozwala łączyć węzły w strukturę drzewa. W wierszu nr 11 zadeklarowano i zainicjowano zmienną `root`, która będzie korzeniem drzewa czerwono-czarnego.

W kodzie modułu zdefiniowano także dwie funkcje o nazwach `find_node()` oraz `insert_node()`. Pierwsza z nich wyszukuje węzeł w drzewie o wartości klucza zadanej jej parametrem `number`. Z uwagi na ograniczony rozmiar stosu jądra (typowo są to tylko 2 strony pamięci), to używanie rekurencji przez funkcje jądra nie jest wskazane. W związku z tym funkcja `find_node()` przeszukuje drzewo z użyciem pętli `while`. W wierszu nr 15 we wskaźniku lokalnym `node` zapamiętywany jest adres zapisany w polu

`rb_node`. Jest to jedyne pole struktury `struct rb_root`, które zawiera adres węzła będącego korzeniem drzewa. Następnie w pętli `while` za pomocą makra `rb_entry` ustalany jest adres struktury `struct example`, w którą wbudowane jest pole wskazywane przez wskaźnik `node` (wiersz 18). Jest to po prostu węzeł drzewa. Jeśli wartość zapisana w polu `number` tego węzła jest większa od tej zawartej w parametrze `number`, to wskaźnik `node` jest przestawiany na lewego potomka tego węzła (o ile istnieje), a jeśli mniejsza, to wskaźnik `node` jest przestawiany na prawego potomka (o ile istnieje). Jeśli obie wartości są takie same, to jest zwracany adres bieżącego węzła (wiersz 25). Jeśli wartość przekazana przez parametr `number` nie znajduje się w drzewie, to pętla `while` zakończy się i zwrócona zostanie wartość `NULL` (wiersz 27).

Funkcja `insert_node()` jest bardziej skomplikowana. Jej zdaniem jest umieszczenie nowego węzła w drzewie czerwono-czarnym. Jeśli jej się to uda, to zwróci wartość `true`, a w przeciwnym przypadku `false`. Funkcja ta posiada dwa parametry: wskaźnik na korzeń drzewa i wskaźnik na element, który powinien być do niego wstawiony. W wierszu 32 zadeklarowane są i zainicjowane są dwa wskaźniki. Pierwszy (`new_node`) jest podwójnym wskaźnikiem, który będzie zwierał początkowo adres korzenia drzewa. Będzie on służył do „przemieszczania" się po drzewie i ostatecznie będzie zawierał adres węzła, do którego będzie dołączony nowy węzeł. Drugi (`parent`) będzie wskazywał na przodka węzła, którego adres znajdzie się w `new_node`, dlatego jego wartość początkowo wynosi `NULL` (korzeń nie ma przodka). W pętli `while`, w wierszu 35 ustalany jest i zapisywany we wskaźniku `this` adres struktury `struct example` obudowującej pole `node` wskazywane przez `*new_node`. W wierszu 37 zapamiętywany jest we wskaźniku `parent` adres pola `node` wskazywanego przez `*new_node`. Następnie wartość pola `number` zawartego w nowym węźle jest porównywana do wartości tego samego pola w bieżąco odwiedzanym węźle. Jeśli jest mniejsza, to wskaźnik `new_node` jest „przestawiany" na lewego potomka bieżącego węzła (o ile istnieje), jeśli większa, to na prawego potomka tego węzła. Jeśli te wartości są równe, to funkcja kończy działanie i zwraca wartość `false` - implementacja drzewa czerwono-czarnego, która zrealizowana jest w jądrze Linuksa zakłada, że klucze w drzewie się nie powtarzają. Jeśli zostanie znaleziony węzeł, którego odpowiedni potomek nie istnieje, to wykonanie pętli `while` jest kończone i nowy węzeł staje się tym potomkiem, czyli jest wstawiany do drzewa za pomocą `rb_link_node()` i wywoływana jest funkcja `rb_insert_color()`, na wypadek, gdyby drzewo wymagało wyważenia. Po tym funkcja `insert_node()` kończy swoje działanie i zwraca wartość `true`.

W funkcji inicjującej moduł tworzonych jest 5 węzłów, które otrzymują wartości pola `number` kolejno: `0`, `1`, `2`, `3`, `4`. Są one następnie wstawiane do drzewa za pomocą wywołania funkcji `insert_node()`. Następnie, w kolejnej pętli te węzły są wyszukiwane i ich wartość jest umieszczana w buforze jądra.

W funkcji sprzątające węzły są wyszukiwane według kluczy, usuwane z drzewa i zwalniana jest pamięć na nie przeznaczona.

4. Drzewa pozycyjne

Drzewa pozycyjne (ang. *radix tree*) są odmianą drzew trie, które są zaprojektowane pod względem oszczędności miejsca w pamięci. W Linuksie te drzewa pozwalają powiązać klucz, który jest liczbą całkowitą z wskaźnikiem na zmienną zawierającą właściwą wartość, czyli utworzyć pary klucz-wartość[4]. Dodatkowo linuksowa implementacja pozwala nadać etykiety elementom drzewa. Każdy węzeł w drzewie zawiera tablicę zazwyczaj 64 wskaźników. Każdy z tych elementów jest indeksowany liczbą całkowitą typu `long int`. Zazwyczaj w drzewie pozycyjnym klucze są jednak większe niż 64, zatem potrzeba więcej niż jednego węzła do zapamiętania związanych z nim wartości. Podczas przeszukiwania drzewa sześć najstarszych bitów w kluczu używanych jest do odnalezienia odpowiedniego wskaźnika w tablicy znajdującej się w korzeniu drzewa. Następne sześć bitów indeksuje odpowiedni wskaźnik w tablicy węzła wskazanego przez poprzedni węzeł, i tak dalej, aż sześć najmłodszych bitów pozwoli zlokalizować wskaźnik w tablicy znajdującej się w liściu drzewa, który wskazuje właściwą daną.

4.1. Opis API

Aby posługiwać się w module jądra drzewem pozycyjnym należy włączyć do jego kodu źródłowego plik nagłówkowy `radix-tree`. Są w nim, między innymi, zdefiniowane następujące makra i funkcje związane z obsługą takich drzew:

[4] Więcej na temat implementacji drzew w jądrze Linuksa można przeczytać w artykule pt. "Trees I: Radix trees" autorstwa Johnatana Corbeta: `https://lwn.net/Articles/175432/`

`RADIX_TREE` - makro, które pozwala utworzyć zmienną typu `struct radix_tree_root` będącą korzeniem drzewa pozycyjnego. Przyjmuje ono dwa argumenty: nazwę tej zmiennej oraz znacznik typu przydziału.

`int radix_tree_insert(struct radix_tree_root *root, unsigned long index, void *entry)` - funkcja pozwalająca wstawić nową wartość do drzewa pozycyjnego. Pobiera ona trzy argumenty: wskaźnik na korzeń drzewa, klucz określający położenie tej wartości w drzewie oraz wskaźnik typu `void *` na zmienną zawierającą tę wartość. Jeśli wstawienie się powiedzie, to funkcja zwróci wartość `0`. Wartość różna od zera zwracana jest w przeciwnym przypadku.

`void *radix_tree_lookup(struct radix_tree_root *, unsigned long)` - funkcja pozwala wyszukać na podstawie klucza wartość w drzewie pozycyjnym. Pobiera dwa argumenty wywołania: wskaźnik na korzeń drzewa oraz klucz. Zwraca wskaźnik na zmienną zawierającą szukaną wartość lub `NULL` jeśli taka wartość nie istnieje.

`void *radix_tree_delete(struct radix_tree_root *, unsigned long)` - funkcja usuwa wartość związaną z danym kluczem z drzewa, zwracając przy tym wskaźnik na zmienną zawierającą tę wartość (typu `void *`). Jako argumenty wywołania przyjmuje ona wskaźnik na korzeń drzewa pozycyjnego oraz klucz związany z wyszukiwaną wartością. Jeśli usuwana wartość nie istnieje funkcja zwraca `NULL`.

Opis API dla drzewa pozycyjnego w tej instrukcji ogranicza się jedynie do funkcji i makr używanych w przykładowym module.

4.2. Przykład

Listing 4 zawiera kod źródłowy przykładowego modułu korzystającego z drzewa pozycyjnego do przechowywania par klucz-wartość, gdzie wartościami są wylosowane liczby typu `int`.

Listing 4: Moduł korzystający z drzewa pozycyjnego

```
1  #include<linux/module.h>
2  #include<linux/random.h>
3  #include<linux/slab.h>
4  #include<linux/radix-tree.h>
5
6  #define UPPER_LIMIT 65535
7
8  static RADIX_TREE(root,GFP_KERNEL);
9
10 static int __init radixtreemod_init(void)
11 {
12         long int i;
13         int *random_number=NULL, *data = NULL;
14
15         for(i=1;i<UPPER_LIMIT;i<<=3) {
16                 random_number = (int *)kmalloc(sizeof(int), GFP_KERNEL);
17                 if(IS_ERR(random_number)) {
18                         pr_alert("Error allocating memory: %ld\n",PTR_ERR(random_number));
19                         return -ENOMEM;
20                 }
21                 get_random_bytes(random_number,sizeof(int));
22                 *random_number%=2017;
23                 if(radix_tree_insert(&root,i,(void *)random_number))
24                         pr_alert("Error inserting item to radix tree!\n");
25         }
26
27         for(i=1;i<UPPER_LIMIT;i<<=3) {
28                 data = (int *)radix_tree_lookup(&root,i);
29                 if(data)
30                         pr_notice("Value retrieved from radix tree: %d for index: %ld\n",*data,i);
```

```
31                  else
32                          pr_alert("Error retrieving data from tree!\n");
33          }
34
35          return 0;
36  }
37
38  static void __exit radixtreemod_exit(void)
39  {
40          int *data=NULL;
41          long int i;
42
43          for(i=1;i<UPPER_LIMIT;i<<=3) {
44                  data  = (int *)radix_tree_delete(&root,i);
45                  if(data) {
46                          pr_notice("Value retrieved from radix tree: %d for index: %ld\n",*data,i);
47                          kfree(data);
48                  } else
49                          pr_alert("Error retrieving data from tree!\n");
50          }
51  }
52
53  module_init(radixtreemod_init);
54  module_exit(radixtreemod_exit);
55
56  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
57  MODULE_LICENSE("GPL");
58  MODULE_DESCRIPTION("A module that demonstrates the usage of kernel radix tree implementation.");
59  MODULE_VERSION("1.0");
```

Stała `UPPER_LIMIT` określa górny zakres wartości klucza. Kolejne klucze dla wartości będą wyznaczane poprzez pomnożenie ich poprzedników przez osiem, dzięki temu drzewo będzie zawierało więcej niż jeden węzeł.

W wierszu 8 modułu tworzona jest zmienna o nazwie `root`, która będzie węzłem drzewa pozycyjnego. Drugi argument makra `RADIX_TREE` określa sposób przydzielania pamięci na kolejne węzły tego drzewa.

W funkcji inicjującej moduł, w pierwszej pętli `for`, tworzone są zmienne dynamiczne, do których zapisywane są losowane liczby. Te zmienne wstawiane są następnie do drzewa pozycyjnego z kluczami określonymi przez wartość licznika pętli. W drugiej pętli `FOR` wartości tych zmiennych są wyszukiwane po kluczach w drzewie pozycyjnym i zapisywane, wraz z kluczami, do bufora jądra.

W funkcji sprzątającej modułu, zmienne dynamiczne przechowujące wartości są usuwane według kluczy z drzewa pozycyjnego. Ich wartości, wraz z kluczami są umieszczane w buforze jądra, a następnie zwalniana jest pamięć na nie przydzielona.

5. Inne struktury danych

Jądro Linuksa posiada także implementacje innych struktur danych, takich jak np. odwzorowania (ang. *maps*), które w Linuksie są nazywane *idr* od *Integer ID management* (pol. *zarządzanie identyfikatorami całkowitymi*) lub cykliczne listy jednokierunkowe, które są używane w jądrze systemu do tworzenia tablic haszujących. Nie będą one jednak opisywane w tej instrukcji.

Jako przykład zastosowania opisanych wcześniej struktur można podać przykład listy procesów, która gromadzi deskryptory wszystkich procesów wykonywanych w systemie komputerowym pracującym pod kontrolą Linuksa. Każdy deskryptor procesu (struktura typu `struct task_struct`) zawiera pole typu `struct list_head`, co pozwala dołączyć go do takiej listy. Dodatkowo programiści Linuksa zdefiniowali trzy makra ułatwiające posługiwanie się taką listą: `for_each_process` pozwala iterować po całej liście procesów. Jako argument przyjmuje wskaźnik na deskryptor procesu, którym „porusza” się po liście. Marko `next_task` przyjmuje jako argument wskaźnik na deskryptor procesu znajdującego się na liście

i zwraca wskaźnik na deskryptor kolejnego procesu, a makro `next_prev` przyjmuje taki sam argument jak `next_task` i zwraca wskaźnik na deskryptor poprzedniego procesu na liście.

Listing 5 zawiera kod źródłowy modułu, którego funkcja inicjująca i sprzątająca przeglądają listę deskryptorów procesów i wypisują nazwę (pole `comm` deskryptor), `pid` i stan każdego z procesów.

Listing 5: Moduł odczytujący listę procesów

```
#include<linux/module.h>
#include<linux/sched/signal.h>

static int __init init_tasklist(void)
{
        struct task_struct *p;

        for_each_process(p) {
                pr_info("Process name is: %s, and its pid is: %i. ",p->comm, task_pid_nr(p));
                if(p->state==TASK_RUNNING)
                        pr_info("Process state is TASK_RUNNING\n");
                if(p->state==TASK_INTERRUPTIBLE)
                        pr_info("Process state is TASK_INTERRUPTIBLE\n");
                if(p->state==TASK_IDLE)
                        pr_info("Process state is TASK_IDLE\n");
                if(p->state==TASK_UNINTERRUPTIBLE)
                        pr_info("Process state is TASK_UNINTERRUPTIBLE\n");
                if(p->state==TASK_STOPPED)
                        pr_info("Proces state is TASK_STOPPED\n");
        }
        return 0;
}

static void __exit exit_tasklist(void) {

        struct task_struct *p;

        for_each_process(p) {
                pri_info("Process name is: %s, and its pid is: %i. ",p->comm, task_pid_nr(p));
                if(p->state==TASK_RUNNING)
                        pr_info("Process state is TASK_RUNNING.\n");
                if(p->state==TASK_INTERRUPTIBLE)
                        pr_info("Process state is TASK_INTERRUPTIBLE.\n");
                if(p->state==TASK_IDLE)
                        pr_info("Process state is TASK_IDLE\n");
                if(p->state==TASK_UNINTERRUPTIBLE)
                        pr_info("Process state is TASK_UNINTERRUPTIBLE.\n");
                if(p->state==TASK_STOPPED)
                        pr_info("Proces state is TASK_STOPPED.\n");
        }
}

module_init(init_tasklist);
module_exit(exit_tasklist);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple module that prints name, pid and state of every active process.");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_VERSION("1.0");
```

Proszę zwrócić uwagę, że pole `pid` deskryptora nie jest odczytywane bezpośrednio, tylko za pomocą funkcji `task_pid_nr()`. Jest to sposób zalecany przez programistów jądra systemu Linux. Ponadto moduł uwzględnia stan procesu, który nie był opisany na wykładzie. Chodzi o `TASK_IDLE`, który jest podobny do

stanu `TASK_UNINTERRUPTIBLE`, ale dane statystyczne procesów, które się w nim znajdują nie są wliczane do średniego obciążenia systemu.

Zdania

1. `[6 punktów]` Napisz program dla przestrzeni użytkownika, który będzie się tak długo wykonywał, aż użytkownik zakończy go wprowadzeniem znaku `q`. Napisz moduł, który odszuka deskryptor tego programu na liście procesów i zapisze wybrane przez Ciebie informacje z tego deskryptora do bufora jądra. Definicja typu struktury deskryptora znajduje się w pliku `linux/sched.h`.

2. `[2 punkty]` Napisz moduł, w którym wykorzystasz implementację listy do utworzenia stosu. Wartości dla elementów stosu przekaż przez parametry modułu.

3. `[2 punkty]` Napisz moduł, w którym wykorzystasz implementację listy do utworzenia kolejki FIFO. Wartości dla elementów kolejki przekaż przez parametry modułu.

4. `[4 punktów]` Napisz moduł, w którym wykorzystasz implementację kolejki FIFO do przechowywania łańcuchów znaków o różnej długości. Te łańcuchy przekaż przez parametry modułu.

5. `[4 punktów]` Powtórzy polecenie z zadania 4, ale zamiast kolejki FIFO użyj drzewa czerwono-czarnego.

6. `[6 punktów]` Powtórzy polecenie z zadania 5, ale zamiast drzewa czerwono-czarnego użyj drzewa pozycyjnego.

7. `[4 punktów]` Użyj w module drzewa czerwono-czarnego do przechowywania struktur, które będą miały trzy pola. Pierwsze pole będzie przechowywało klucz będący liczbą naturalną, drugie pole będzie przechowywało wylosowaną małą literę, a trzecie będzie typu `struct rb_node`. Wartości z tego drzewa umieść w buforze jądra.

8. `[6 punktów]` Rozbuduj moduł 3.2, tak aby dodawanych było więcej węzłów do drzewa czerwono-czarnego, a następnie użyj opisanych w rozdziale o tej strukturze funkcji do umieszczania w buforze wartości tych węzłów w porządku niemalejącym i nierosnącym. Sprawdź też jak działa funkcja zastępująca węzły drzewa.