Laboratorium 4: „Systemy plików `procfs` i `sysfs`” (dwa zajęcia)

dr inż. Arkadiusz Chrobot

2 kwietnia 2024

Spis treści

Wprowadzenie

W pierwszej instrukcji do zajęć laboratoryjnych zostało przedstawione zagadnienie parametrów dla modułów jądra. Pozwalają one na jednostronną i jednorazową komunikację z modułem, tzn. podczas jego ładowania użytkownik jest w stanie określić za pomocą parametrów jego zachowanie. Jądro Linuksa dostarcza jednak innych mechanizmów, które umożliwiają dwustronną komunikację z modułami, która dodatkowo może się odbywać w trakcie ich wykonywania. Należą do nich wirtualne systemy plików, takie jak `procfs` i `sysfs`, które będą przedmiotem tej instrukcji. W rozdziale 1 opisany jest system plików `procfs`, jego rola w systemie, sposób użytkowania, API oraz przykładowy moduł korzystający z tego systemu plików. Rozdział 2 zawiera podobne informacje na temat systemu plików `sysfs`. Instrukcja kończy się listą zadań do samodzielnego wykonania w ramach zajęć laboratoryjnych.

1. System plików `procfs`

System plików `procfs` istnieje wyłącznie w pamięci operacyjnej komputera, co oznacza, że jego struktura katalogów i zawartość plików jest generowana w trakcie działania jądra. Głównie są to informacje statystyczne na temat działania sprzętu, poszczególnych podsystemów jadra i procesów użytkownika. Są one umieszczane w plikach tekstowych, a więc można je wyświetlić na ekranie np. za pomocą polecenia powłoki `cat`. Przykładowo, informacje o procesorze lub procesorach zainstalowanych w komputerze można uzyskać wydając polecenie `cat /proc/cpuinfo`. Istnieją także programy- polecenia, które interpretują i wyświetlają w bardziej zwartej formie informacje przechowywane w plikach `procfs`. Przykładem takiego polecenia może być `pmap`. Część z tych plików jest także modyfikowalna i służy do dynamicznego (tzn. w trakcie działania) konfigurowania parametrów pracy podsystemów jądra. Najczęściej zapisu takich plików można dokonać za pomocą przekierowania przy pomocy operatora `>` wyjścia polecenia powłoki `echo`. W większości dystrybucji systemu Linux system plików `procfs` jest montowany w katalogu `/proc`.

1.1. Opis API

Sposób obsługi plików `procfs` został zmieniony w wersji 3.10 jądra. Zmiany te zostały opisane na stronie `http://tuxthink.blogspot.com/2013/10/creating-read-write-proc-entry-in.html`.

Większość funkcji i struktur danych związanych z obsługą tego systemu plików jest zawarta w pliku nagłówkowym `linux/proc_fs.h`. Główną strukturą opisującą pliki i katalogi w `procfs` jest `struct proc_dir_entry`. Niestety począwszy od wspomnianej wersji jądra jest ona niedostępna dla programistów modułów jądra w sposób bezpośredni. Główni programiści jądra zdecydowali się ukryć szczegóły jej implementacji, aby zapobiec wielu niebezpiecznym błędom popełnianym przez twórców modułów. Opis funkcji związanych z obsługa `procfs` zaczniemy od tych, które pozwalają utworzyć lub usunąć pliki i podkatalogi. Oto niektóre z nich:

`struct proc_dir_entry *proc_mkdir(const char *, struct proc_dir_entry *)` - ta funkcja pozwala utworzyć nowy podkatalog w katalogu, gdzie zamontowany jest system `procfs`. Jej pierwszym argumentem wywołania powinna być nazwa katalogu, który ma być utworzony. Drugim argumentem jest wskaźnik na strukturę typu `struct proc_dir_entry` związaną z katalogiem nadrzędnym dla tego, który ma zostać utworzony. Jeśli wartością tego argumentu będzie `NULL`, to podkatalog zostanie utworzony w głównym katalogu systemu plików `procfs`. Funkcja zwraca `NULL` w razie niepowodzenia lub wskaźnik na strukturę typu `STRUCT PROC_ENTRY` nowo utworzonego podkatalogu.

`struct proc_dir_entry *proc_mkdir_mode(const char *, umode_t, struct proc_dir_entry *)` - ta funkcja działa podobnie jak `proc_mkdir()`, ale przyjmuje dodatkowy (drugi) argument wywołania, który określa prawa dostępu do utworzonego katalogu. Te prawa mogą być zapisane w postaci liczby ósemkowej lub jako suma bitowa następujących stałych:

`S_IRWXU` - wszystkie prawa dla właściciela pliku/katalogu,

`S_IRUSR` - prawo do odczytu dla właściciela pliku/katalogu,

`S_IWUSR` - prawo do zapisu dla właściciela,

`S_IXUSR` - prawo do wykonania dla właściciela,

`S_IRWXG` - wszystkie prawa dla grupy użytkowników, do której należy właściciel pliku/katalogu,

`S_IRGRP` - prawo do odczytu dla grupy użytkowników, do której należy właściciel pliku/katalogu,

`S_IWGRP` - prawo do zapisu dla grupy użytkowników, do której należy właściciel pliku/katalogu,

`S_IXGRP` - prawo do wykonania dla grupy użytkowników, do której należy właściciel pliku/katalogu,

`S_IRWXO` - wszystkie prawa dla pozostałych użytkowników,

`S_IROTH` - prawo do odczytu dla pozostałych użytkowników,

`S_IWOTH` - prawo do zapisu dla pozostałych użytkowników,

`S_IXOTH` - prawo do wykonania dla pozostałych użytkowników,

`S_IRWXUGO` - wszystkie prawa dostępu dla wszystkich użytkowników,

`S_IRUGO` - prawo do odczytu dla wszystkich użytkowników,

`S_IWUGO` - prawo do zapisu dla wszystkich użytkowników,

`S_IXUGO` - prawo do wykonania dla wszystkich użytkowników,

Można również użyć tych stałych pojedynczo.

`struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct proc_dir_entry *parent, const struct file_operations *proc_fops)` - ta funkcja tworzy plik w systemie plików `procfs`. Jako pierwszy argument przyjmuje nazwę tego pliku, jako drugi prawa dostępu, które zostały objaśnione w opisie funkcji `proc_mkdir()`. Kolejny argument to wskaźnik na strukturę typu `struct proc_dir_entry` związaną z katalogiem, w którym ma być utworzony plik. Ostatnim argumentem ten funkcji jest wskaźnik na strukturę typu `struct file_operations`, która zawiera wskaźniki do funkcji realizujących operacje na pliku (będzie objaśniona dalej w tym rozdziale). Funkcja `proc_create()` zwraca wskaźnik do struktury typu `proc_dir_entry` związanej z nowo utworzonym plikiem lub `NULL` jeśli pliku nie udało się utworzyć.

`struct proc_dir_entry *proc_create_data(const char *, umode_t, struct proc_dir_entry *, const struct file_operations *, void *)` - ta funkcja jest podobna w działaniu do `proc_create()`, ale pobiera dodatkowy, ostatni, argument będący wskaźnikiem (typu `void *`) na bufor, gdzie będą trzymane dane zawarte w pliku.

`struct proc_dir_entry *proc_symlink(const char *, struct proc_dir_entry *, const char *)` - ta funkcja służy do tworzenia dowiązania symbolicznego do pliku należącego do systemu plików

`procfs`. Przyjmuje ona trzy argumenty: nazwę dowiązania, wskaźnik na strukturę typu `proc_dir_entry`, która opisuje katalog, w którym dowiązanie ma zostać utworzone oraz nazwę pliku, do którego ma powstać dowiązanie. Funkcja zwraca wskaźnik do struktury typu `struct proc_dir_entry` związanej z nowo powstałym dowiązaniem lub wartość `NULL` jeśli utworzenie dowiązania się nie powiodło.

`void proc_remove(struct proc_dir_entry *)` - ta funkcja służy do usuwania pliku lub katalogu z systemu plików `procfs`. Nie zwraca ona żadnej wartości, a jako argument wywołania przyjmuje wskaźnik na strukturę typu `struct proc_dir_entry` związaną z plikiem lub katalogiem, który ma zostać usunięty.

Moduły posługujące się `procfs` korzystają ze struktury `struct file_operations`, która pozwala zdefiniować funkcje (metody) wykonujące podstawowe operacje na plikach, takiej jak zapis i odczyt. Listing 1 przedstawia jej budowę.

Listing 1: Definicja struktury `struct file_operations`

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long,
               unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
        unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
        unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
};
```

Nawet pobieżna analiza definicji tej struktury pozwala stwierdzić, że jest ona bardzo złożona i umożliwia definiowanie wielu metod działających na plikach. Na szczęście programiści jądra nie muszą definiować ich wszystkich. Najczęściej definiowane są: `open()`, `release()`, `read()`, `write()` i `llseek()` oraz nadawana jest wartość polu `owner`, aby zapobiec wcześniejszemu usunięciu tej struktury z pamięci przez inne podsystemy jądra. Aby dodatkowo ułatwić pracę programistom modułów, w jądrze zdefiniowany został mechanizm *plików sekwencyjnych*, który zawiera gotowe metody wykonujące typowe operacje na takich plikach, jakie najczęściej są stosowane w systemie `procfs`.

Zatem w niektórych przypadkach wystarczy tylko zdefiniować trzy funkcje: obsługującą zapis do pliku, obsługującą odczyt z pliku oraz otwierającą plik. Nagłówki dwóch pierwszych muszą być zgodne z następującymi prototypami:

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

oraz:

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Obie funkcje zwracają liczbę zapisanych/odczytanych bajtów z pliku lub `-1` w przypadku pojawienia się wyjątku. Pierwszy parametr tych funkcji, to wskaźnik na strukturę związaną z otwartym plikiem, na którym dana operacja ma być przeprowadzona. Drugi parametr to wskaźnik do bufora związanego z przestrzenią użytkownika (stąd znacznik `__user` przed ich deklaracją), z którego funkcja zapisu będzie pobierała informacje do zapisania w pliku, a funkcja odczytu będzie umieszczała dane z pliku przekazywane do przestrzeni użytkownika. Trzeci parametr to rozmiar odczytywanej/zapisywanej porcji pliku wyrażony w bajtach, a ostatni argument to wskaźnik do zmiennej lokalizującej początek tej porcji informacji (wskaźnik pliku).

W definicji funkcji pełniących rolę metod `write()` i `read()` przydatne będą dwie funkcje zdefiniowane w pliku `linux/uaccess.h`:

- `long copy_to_user(void __user *to, const void *from, unsigned long n)` - funkcja ta umożliwia kopiowanie danych z bufora znajdującego się w przestrzeni jądra i wskazywanego przez wskaźnik `form` do bufora znajdującego się w przestrzeni użytkownika i wskazywanego przez wskaźnik `to`. Rozmiar danych do przekopiowania określa (w bajtach) wartość parametru `n`. Funkcja zwraca liczbę bajtów, których **nie** udało jej się przekopiować, a więc w przypadku sukcesu zwróci `0`.
- `long copy_from_user(void *to, const void __user * from, unsigned long n)` - funkcja ta kopiuje dane z bufora znajdującego się w przestrzeni użytkownika i wskazywanego przez parametr `from` do bufora znajdującego się w przestrzeni jądra i wskazywanego przez wskaźnik `to`. Parametr `n` określa liczbę bajtów do przekopiowania. Funkcja zwraca liczbę bajtów, których **nie** udało się jej przekopiować, a więc w przypadku sukcesu zwróci `0`.

Polu `owner` struktury typu `struct file_operations` przypisuje się wartość makra `THIS_MODULE`. Jest ono dostępne w każdym module i zwraca wskaźnik na strukturę typu `struct module` reprezentującą dany moduł w jądrze systemu Linux.

1.2. Pliki sekwencyjne

Aby uprościć tworzenie prostych systemów plików, których pliki są głównie odczytywane sekwencyjnie, programiści jądra stworzyli odpowiednie funkcje i struktury, które razem tworzą mechanizm nazywany po prostu *plikami sekwencyjnymi*. Wymaga on zdefiniowania kilku funkcji, których zadaniem jest umożliwienie łatwego iterowania po zawartości pliku, nawet jeśli jego wielkość przekracza rozmiar strony. Z drugiej strony mechanizm ten dostarcza również gotowych funkcji, które mogą pełnić rolę metod dla obiektów plików. Wbrew nazwie opisywane rozwiązanie pozwala także na dostęp swobodny do plików, ale nie jest on zbyt efektywny. Mechanizm plików sekwencyjnych można z powodzeniem zastosować w modułach jądra korzystających z `procfs`[1]

1.2.1. Opis API plików sekwencyjnych

Funkcje i struktury danych związane z mechanizmem plików sekwencyjnych są zadeklarowane lub zdefiniowane w pliku nagłówkowym `linux/seq_file.h` Jednym z podstawowych elementów tego mechanizmu jest typ strukturalny `struct seq_operations`, którego budowę pokazano na listingu 2.

Listing 2: Definicja struktury `struct seq_operations`

```
1 struct seq_operations {
2         void * (*start) (struct seq_file *m, loff_t *pos);
3         void (*stop) (struct seq_file *m, void *v);
```

[1]Mechanizm plików sekwencyjnych został opisany między innymi na stronie o adresie: `http://crashcourse.ca/introduction-linux-kernel-programming/lesson-13-proc-files-and-sequence-files-part-3`.

```
4           void * (*next) (struct seq_file *m, void *v, loff_t *pos);
5           int (*show) (struct seq_file *m, void *v);
6   };
```

Z tej definicji wynika, że aby skorzystać z mechanizmu plików sekwencyjnych trzeba zdefiniować cztery funkcje tworzące razem iterator udostępniający w sposób sekwencyjny zawartość pliku. Adresy tych funkcji należy następnie przypisać do odpowiednich wskaźników będących polami struktury z listingu 2.

Wskaźnik `start` powinien wskazywać na funkcję, która będzie odpowiedzialna z inicjację działania iteratora. Musi ona posiadać dwa parametry. Przez pierwszy będzie jej przekazany wskaźnik na strukturę reprezentującą plik sekwencyjny, a przez drugi adres wskaźnika pliku ustawionego na pozycję początkową pliku[2]. Funkcja powinna zwrócić adres pierwszego elementu bufora do odczytania z pliku sekwencyjnego.

Wskaźnik `stop` powinien wskazywać na funkcję finalizującą działanie iteratora. Funkcja ta powinna posiadać dwa parametry, pierwszy wskazujący strukturę reprezentującą plik sekwencyjny, a drugi na bufor zwierający dane umieszczone w pliku sekwencyjnym. Jeśli ten bufor w funkcji wskazywanej przez wskaźnik `start` był utworzony poprzez dynamiczny przydział pamięci, to w funkcji wskazywanej przez `stop` ten bufor powinien być usunięty poprzez zwolnienie tej pamięci. Funkcja ta nic nie zwraca. W wielu przypadkach jej definicja pozostaje pusta, jeśli nie ma żadnej czynności do wykonania podczas finalizacji iteratora.

Wskaźnik `next` powinien wskazywać na funkcję, która zwróci wskaźnik na bieżący element do odczytania z pliku sekwencyjnego oraz ustawi wskaźnik pliku na kolejny element do odczytania. Jeśli taki element nie istnieje powinna ona zwrócić `NULL`. Funkcja ta musi posiadać trzy parametry. Przez pierwszy przekazywany jest do niej wskaźnik na strukturę reprezentującą plik sekwencyjny, przez drugi parametr jest przekazywany wskaźnik do bufora, a przez trzeci przekazywany jest adres wskaźnika pliku. Przed zakończeniem działania funkcji jego wartość powinna być uaktualniona tak, aby wskazywał on następny element bufora do odczytu.

Wskaźnik `show` powinien wskazywać na funkcję, która będzie umieszczała dane z bufora w pliku. Powinna ona posiadać dwa parametry wskaźnikowe. Przez pierwszy będzie jej przekazywany adres struktury reprezentującej plik sekwencyjny, a przez drugi wskaźnik na dane, które mają być przekazane do przestrzeni użytkownika. Zwraca ona zero, jeśli odczyt się powiedzie. W jej implementacji mogą być pomocne następujące funkcje:

- `void seq_printf(struct seq_file *m, const char *fmt, …)` - funkcja ta ma podobną listę parametrów i sposób wywołania jak `printf()` z przestrzeni użytkownika. Jej pierwszym argumentem wywołania jest wskaźnik na strukturę reprezentującą plik sekwencyjny, kolejnym ciąg znaków zawierający ciąg lub ciągi formatujące. Następnymi są zmienne (elementy bufora), których wartości mają być podstawione za ciągi formatujące.
- `void seq_putc(struct seq_file *m, char c)` - funkcja ta umożliwia odczytanie pojedynczego znaku z bufora pliku sekwencyjnego. Jako pierwszy argument wywołania przyjmuje adres struktury reprezentującej plik sekwencyjny, a jako drugi znak, który będzie odczytany z pliku sekwencyjnego.
- `void seq_puts(struct seq_file *m, const char *s)` - funkcja ta umożliwia odczyt ciągu znaków z bufora pliku sekwencyjnego. Jako pierwszy argument wywołania przyjmuje adres struktury reprezentującej plik sekwencyjny, a jako drugi adres łańcucha znaków, który znajduje się w buforze pliku sekwencyjnego i ma być z niego odczytany.

Na wstępie do podrozdziału o plikach sekwencyjnych znajduje się informacja, że ich mechanizm zawiera gotowe funkcje, które mogą pełnić rolę metod dla obiektu pliku. One również zostały zadeklarowane w pliku `linux/seq_file.h`. Należą do nich:

- `int seq_open(struct file *, const struct seq_operations *)` - funkcja ta otwiera plik sekwencyjny, czyli inicjuje pracę z nim. Najczęściej wywołuje się ją w definicji funkcji, która będzie wskazywana przez wskaźnik `open` struktury typu `struct file_operations`. Jako pierwszy argument jej wywołania przekazuje się drugi parametr tej funkcji, a jako drugi adres zainicjowanej struktury typu `struct seq_operations`. Zwraca ona kod otwarcia, który powinien być również zwrócony przez funkcję implementującą metodę `open()`.

[2] Niekoniecznie w przypadku tej funkcji wartość początkowa wskaźnika pliku musi wynosić zero.

`int single_open(struct file *, int (*)(struct seq_file *, void *), void *)` - funkcja ta jest zamiennikiem dla `seq_open()` jeśli z pliku ma być odczytywana pojedyncza wartość. Jako pierwszy jej argument wywołania przekazuje się ostatni parametr funkcji implementującej metodę `open()`, jako drugi adres implementacji funkcji `show()` dla pliku sekwencyjnego. Jako ostatni argument jej wywołania jest przekazywany adres bufora pliku sekwencyjnego.

`int single_release(struct inode *, struct file *)` - adres tej funkcji jest przypisywany do pola `release` struktury typu `struct file_operations`, jeśli do implementacji metody `open()` została użyta funkcja `single_open()`.

`int seq_release(struct inode *, struct file *)` - adres tej funkcji jest przypisywany do pola `release` struktury typu `struct file_operations`, jeśli do implementacji metody `open()` została użyta funkcja `seq_open()`.

`ssize_t seq_read(struct file *, char __user *, size_t, loff_t *)` - adres tej funkcji jest przypisywany do pola `read` struktury typu `struct file_operations`.

`loff_t seq_lseek(struct file *, loff_t, int)` - adres tej funkcji jest przypisywany do pola `llseek` struktury typu `struct file_operations`.

1.3. Przykład

Listing 3 przedstawia kod źródłowy modułu jądra, który tworzy w katalogu `proc` podkatalog `procfs_test`, a w nim plik o nazwie `procfs_file`, który może być zapisywany i odczytywany z przestrzeni użytkownika.

Listing 3: Moduł korzystający z procfs

```
1  #include<linux/module.h>
2  #include<linux/uaccess.h>
3  #include<linux/proc_fs.h>
4  #include<linux/seq_file.h>
5
6  static struct proc_dir_entry *directory_entry_pointer, *file_entry_pointer;
7  static char *directory_name = "procfs_test", *file_name = "procfs_file";
8  static char file_buffer[PAGE_SIZE];
9
10 static int procfsmod_show(struct seq_file *seq, void *data)
11 {
12         char *notice = (char *)data;
13         seq_putc(seq,*notice);
14         return 0;
15 }
16
17 static void *procfsmod_seq_start(struct seq_file *s, loff_t *position)
18 {
19         loff_t buffer_index = *position;
20         return (buffer_index<PAGE_SIZE && file_buffer[buffer_index]!='\0')
21                 ?(void *)&file_buffer[buffer_index]:NULL;
22 }
23
24 static void *procfsmod_seq_next(struct seq_file *s, void *data, loff_t *position)
25 {
26         loff_t next_element_index = ++*position;
27         return (next_element_index<PAGE_SIZE && file_buffer[next_element_index]!='\0')
28                 ?(void *)&file_buffer[next_element_index]:NULL;
29 }
30
31 static void procfsmod_seq_stop(struct seq_file *s, void *data)
32 {
33 }
34
```

```
35  static struct seq_operations procfsmod_seq_operations = {
36          .start = procfsmod_seq_start,
37          .next = procfsmod_seq_next,
38          .stop = procfsmod_seq_stop,
39          .show = procfsmod_show
40  };
41
42  static int procfsmod_open(struct inode *inode, struct file *file)
43  {
44          return seq_open(file, &procfsmod_seq_operations);
45  }
46
47  static ssize_t procfsmod_wirte(struct file *file, const char __user *buffer, size_t count,
48                  loff_t *position)
49  {
50          int length = count;
51          if(count>PAGE_SIZE)
52                  length=PAGE_SIZE-1;
53          if(copy_from_user(file_buffer,buffer,length))
54                  return -EFAULT;
55          file_buffer[length]='\0';
56          return length;
57  }
58
59  static struct file_operations procfsmod_fops = {
60          .owner = THIS_MODULE,
61          .open = procfsmod_open,
62          .read = seq_read,
63          .write = procfsmod_wirte,
64          .llseek = seq_lseek,
65          .release = seq_release
66  };
67
68  static int __init procfsmod_init(void)
69  {
70          directory_entry_pointer = proc_mkdir(directory_name, NULL);
71          if(IS_ERR(directory_entry_pointer)) {
72                  pr_alert("Error creating procfs directory: %s. Error code: %ld\n",
73                          directory_name,PTR_ERR(directory_entry_pointer));
74                  return -1;
75          }
76          file_entry_pointer = proc_create_data(file_name,0666,directory_entry_pointer,
77                                  &procfsmod_fops,(void *)file_buffer);
78          if(IS_ERR(file_entry_pointer)) {
79                  pr_alert("Error creating procfs file: %s. Error code: %ld\n",
80                          file_name,PTR_ERR(file_entry_pointer));
81                  proc_remove(directory_entry_pointer);
82                  return -1;
83          }
84
85          return 0;
86  }
87
88  static void __exit procfsmod_exit(void)
89  {
90          if(file_entry_pointer)
91                  proc_remove(file_entry_pointer);
92          if(directory_entry_pointer)
93                  proc_remove(directory_entry_pointer);
94  }
```

```
95
96  module_init(procfsmod_init);
97  module_exit(procfsmod_exit);
98  MODULE_LICENSE("GPL");
99  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
100 MODULE_DESCRIPTION("Procfs capabilities showing module.");
101 MODULE_VERSION("1.0");
```

Wiersz nr 6 listingu 3 zawiera deklaracje dwóch wskaźników na struktury typu `struct proc_dir_example`. Pierwszy będzie wskazywał na strukturę związaną z utworzonym podkatalogiem, a drugi na strukturę związaną z utworzonym plikiem. W wierszu nr 7 zadeklarowano i zainicjowano dwie zmienne, które wskazują na łańcuch będące nazwami odpowiednio tworzonego przez moduł podkatalogu i pliku. Kolejny wiersz zawiera deklarację zmiennej będącej buforem na dane pliku. Jest to tablica znaków o wielkości 4KiB, czyli zawiera 4096 elementów.

Funkcja `procfsmod_show()` (wiersze nr 10 - 15) jest implementacją funkcji `show()` dla pliku sekwencyjnego. W ciele tej funkcji następuje rzutowanie wskaźnika typu `void *` na wskaźnik na znak (typu `char *`), a następnie znak wskazywany przez ten wskaźnik jest umieszczany pliku o nazwie `procfs_file` przy użyciu funkcji `seq_putc()` i zwracana jest liczba `0`.

Funkcja `procfsmod_seq_start()` (wiersze nr 17 - 22) jest z kolei implementacją funkcji `start()` dla pliku sekwencyjnego. W ciele tej funkcji, w wierszu nr 19 zadeklarowana jest jest zmienna lokalna, do której przypisywana jest przekazana funkcji wartość początkowa wskaźnika pliku. Następnie funkcja sprawdza (wiersz nr 20), czy wskaźnik ten nie jest większy od rozmiaru bufora pliku lub czy element, który on określa nie zawiera znaku końca ciągu (`\0`). Jeśli powyższe warunki nie są spełnione to funkcja zwraca adres pierwszego elementu bufora do odczytania, a w przeciwnym przypadku zwraca wartość `NULL`.

Funkcja `procfsmod_seq_next()` (wiersze nr 24 - 29) jest implementacją funkcji `next()` dla pliku sekwencyjnego. Działa ona trochę inaczej, niż zostało to przedstawione w opisie `next()`. Ta funkcja w wierszu nr 26 najpierw wyznacza indeks kolejnego elementu w buforze do odczytania i zapisuje go w zmiennej lokalnej `next_element_index`. Następnie sprawdza te same warunki, co `procfsmod_seq_start()`. Jeśli po ich sprawdzeniu okazuje się, że wartość indeksu jest prawidłowa, to funkcja ta zwraca adres na element bufora przez niego określany, a jeśli nie to zwraca `NULL`.

Funkcja `procfsmod_seq_stop()` (wiersze nr 31 - 33) jest implementacją funkcji `stop()` dla pliku sekwencyjnego. Jej ciało jest puste, bowiem sposób implementacji pliku `procfs` zastosowany w module nie wymaga żadnych czynności związanych z finalizacją iteratora. Bufor pliku jest tworzony w module statycznie i nie trzeba zwalniać pamięci na niego przydzielonej.

W wierszach nr 35 - 40 jest zadeklarowana i zainicjowana struktura `procfsmod_seq_operations` typu `struct seq_operations`. Do jej poszczególnych pól wskaźnikowych przypisywane są adresy odpowiednich funkcji, które zostały wcześniej opisane.

Funkcja `procfsmod_open()` (wiersze nr 42 - 45) jest implementacją metody `open()` dla obiektu pliku. W jej ciele wywołana jest funkcja `seq_open()`, której jako argumenty wywołania przekazano wskaźnik na obiekt pliku (drugi parametr funkcji `procfsmod_open()`) i adres struktury `procfsmod_seq_operations` zawierającej adresy funkcji implementujących operacje na pliku sekwencyjnym. Kod zwrócony przez `seq_open()` jest również zwracany przez opisywaną funkcję.

Funkcja `procfsmod_write()` (wiersze nr 47 - 57) jest implementacją metody `wirte()` dla obiektu pliku. W jej ciele, w wierszu nr 50 zapisywana jest w zmiennej lokalnej `length` liczba bajtów do zapisania w buforze pliku z przestrzeni użytkownika. Jest ona przekazywana do funkcji przez parametr `count`. W kolejnym wierszu funkcja sprawdza, czy ta liczba jest większa od rozmiaru bufora. Jeśli tak, to w zmiennej `length` jest zapisywany indeks ostatniego elementu bufora. Zapobiega to przekroczeniu obszaru bufora (zakładamy, że maksymalny rozmiar pliku, którym będzie się posługiwał moduł nie może przekraczać 4 KiB). Następnie z przestrzeni dane z bufora należącego do przestrzeni użytkownika i wskazywanego przez wskaźnik `buffer` są kopiowane do bufora pliku przy użyciu funkcji `copy_from_user()`. Jeśli to kopiowanie się nie powiedzie, to funkcja zwróci odpowiedni numer wyjątku. W wierszu nr 55, w ostatnim elemencie bufora pliku zapisywany jest znak końca ciągu. Jest to konieczne w przypadku gdyby proces z przestrzeni użytkownika próbował zapisać do pliku więcej informacji niż 4 KiB. W takim wypadku należy zadbać o zakończenie kopiowanego ciągu znaków. Po prawidłowym wykonaniu kopiowania funkcja zwraca liczbę przekopiowanych bajtów.

W wierszach 59 - 66 kodu źródłowego modułu zadeklarowana jest i zainicjowana struktura operacji (metod) dla obiektu pliku. Jej poszczególnym polom wskaźnikowym przypisywane są adresy odpowiednich funkcji zdefiniowanych w module lub gotowych funkcji, które są dostarczane przez mechanizm plików sekwencyjnych.

W funkcji inicjującej moduł najpierw tworzony jest podkatalog katalogu `proc`, przy pomocy funkcji `proc_mkdir()`. Jako pierwszy argument jej wywołania jest przekazywana nazwa pliku, a jako drugi stała `NULL`. Drugi argument jest adresem struktury związanej z katalogiem nadrzędnym względem tworzonego. Jeśli wartość tego argumentu wynosi `NULL`, to oznacza, że tym katalogiem będzie punkt montowania systemu plików `procfs`, czyli najczęściej katalog `proc`. Po wywołaniu funkcji sprawdzana jest wartość zwróconego przez nią adresu. W przypadku pojawienia się wyjątku w buforze jądra umieszczany jest odpowiedni komunikat, zwracana jest wartość sygnalizująca błąd działania i kończona jest praca funkcji inicjującej. Jeśli jednak katalog został poprawnie utworzony, to w wierszu nr 76 za pomocą wywołania `proc_create_data()` tworzony jest plik o nazwie `procfs_test`. Jako pierwszy argument wywołania tej funkcji jest przekazywana zmienna wskazująca na nazwę tworzonego pliku, następnie liczba ósemkowa określająca prawa dostępu do niego, adres struktury opisującej katalog, w którym plik zostanie utworzony, adres struktury operacji (metod) dla obiektu pliku i na końcu adres bufora na dane dla tego pliku[3]. Wynik działania tej funkcji też jest sprawdzany, na wypadek, gdyby podczas jej wykonania pojawił się wyjątek. Jeśli tak się stanie to jak poprzednio w buforze jądra umieszczany jest odpowiedni komunikat, ale następnie usuwany jest przy pomocy wywołania funkcji `proc_remove()` utworzony wcześniej podkatalog. Dopiero po wykonaniu tej czynności funkcja inicjująca kończy działania zwracając wartość `-1`. Jeśli jednak utworzenie pliku się powiedzie, to funkcja inicjująca zwróci wartość `0` i również zakończy swoje działanie.

Po zakończeniu funkcji inicjującej użytkownik możne zapisywać ciągi znaków do pliku `procfs_file`, np. w ten sposób:

```
echo "test" > /proc/procfs_test/procfs_file
```

Aby odczytać zawartość pliku użytkownik może wydać np. takie polecenie:

```
cat /proc/procfs_test/procfs_file
```

Wspomniany plik i katalog są usuwane wraz z usunięciem modułu z jądra systemu. Zajmuje się tym funkcja sprzątająca (finalizująca) modułu, która najpierw sprawdza, czy katalog i plik udało się utworzyć, czyli czy wskaźniki na struktury je opisujące mają wartość różną od `NULL`. Jeśli tak jest, to każdy z tych wskaźników jest przekazywany jako argument wywołania funkcji `proc_remove()`.

2. System plików `sysfs`

System plików `sysfs` jest ściśle powiązany z modelem urządzenia, czyli podsystemem jądra Linuksa, który został wprowadzony do niego aby ułatwić takie czynności zarządzania urządzeniami, jak odwzorowywanie topologi połączeń, tworzenie hierarchii i klasyfikację urządzeń oraz zarządzanie energią. Podstawowym elementem tego modelu jest struktura obiektu jądra. Każdy obiekt jądra jest odwzorowywany jako katalog w systemie plików `sysfs`, który w większości dystrybucji Linuksa jest montowany w katalogu `/sys`. Obiekty jądra odwzorowują topologię połączeń urządzeń tworzących system komputerowy nadzorowany przez Linuksa, zatem struktura `sysfs` odzwierciedla tę topologię w przestrzeni użytkownika. Dodatkowo każdy obiekt jądra może posiadać atrybuty, które są widoczne w przestrzeni użytkownika jako pliki. Zawartość tych plików może być odczytywana przez większość użytkowników, ale tylko użytkownicy uprzywilejowani (zwykle *root* lub użytkownicy należący do tej samej grupy co on) mogą je modyfikować. Pliki związane z atrybutami na ogół są plikami tekstowymi, ale istnieją wśród nich także pliki binarne. Zwykle pojedynczy plik zawiera pojedynczą wartość. Odczytu większości tych plików można dokonać za pomocą wspomnianego wcześniej polecenia powłoki `cat`. Modyfikacja zwykle wymaga dostępu do powłoki administratora systemu. Można go osiągnąć z poziomu odpowiednio skonfigurowanego użytkownika za pomocą polecenia `sudo su`. System plików `sysfs` emituje również powiadomienia o zdarzeniach dla przestrzeni użytkownika, jeśli wartość jakiegoś atrybutu ulegnie modyfikacji przez podsystemu jądra. Tak np. demon `udev` jest powiadamiany o dołączeniu do systemu nowego urządzenia zewnętrznego. Ten temat nie będzie szerzej opisywany w niniejszej instrukcji, ponieważ wymaga on wiedzy na temat komunikacji za pomocą gniazd przy użyciu protokołu *netlink*, co będzie tematem odrębnej instrukcji.

[3] Jest to, w skrócie, bufor pliku.

2.1. Opis API

Najważniejszymi strukturami danych w modelu urządzenia stosowanym w Linuksie są struktury typu `struct kobject`. Te struktury to po prostu obiekty jądra. Klasę takich obiektów (typ) określa inny typ struktur, o nazwie `kobj_type`. Pojedynczą taką strukturę można powiązać z wieloma obiektami jądra, tym samym nadając im te same cechy (atrybuty i zachowanie). Trzecim ważnym typem struktury we wspomnianym modelu jest `struct kset`. Są to kontenery (zbiory) obiektów jądra, które pełnią podobne funkcje, np. są powiązane ze sterownikami pewnej grupy urządzeń.

Listing 4 zawiera definicje typu struktury obiektu jądra, którą można znaleźć w pliku nagłówkowym `linux/kobject.h`.

Listing 4: Definicja typu struktury `struct kobject`

```
struct kobject {
        const char              *name;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct kernfs_node      *sd; /* sysfs directory entry */
        struct kref             kref;
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
        struct delayed_work     release;
#endif
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
```

Pole `name` w tej strukturze wskazuje na ciąg znaków będący nazwą obiektu i tym samym związanego z nim katalogu w systemie plików `sysfs`. Pole `parent` wskazuje na obiekt jądra, który jest nadrzędy w stosunku do opisywanego. W systemie plików `sysfs` jest to katalog nadrzędny. Pola wskaźnikowe `kset` i `ktype` przechowują adresy, odpowiednio, klasy obiektu i zbioru. Pole `kref` służy do zliczania odwołań do obiektu. Pozostałe pola nie są interesujące z punktu widzenia tej instrukcji.

Listing 5 zawiera definicję typu struktury opisującej klasę obiektów.

Listing 5: Definicja typu struktury `struct kobj_type`

```
struct kobj_type {
        void (*release)(struct kobject *kobj);
        const struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;
        const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
        const void *(*namespace)(struct kobject *kobj);
};
```

Pole wskaźnikowe `release` wskazuje na funkcję, która odpowiedzialna jest za finalizację (sprzątanie) obiektu, kiedy przestaje on być używany. Jako argument wywołania przyjmuje adres obiektu i jest wywoływana automatycznie, gdy taki obiekt jest niszczony. Pole wskaźnikowe `sysfs_ops` wskazuje na strukturę typu `sysfs_ops`, którego definicja jest umieszczona w listingu 6. Ostatnie pole tej struktury, które zostanie opisane w tej instrukcji to `default_attrs`. Jest to wskaźnik na tablicę struktur typu `struct attribute`. Definicja typu tej struktury jest umieszczona w listingu 7. Te struktury są reprezentowane w systemie plików `sysfs` jako pliki.

Listing 6 zawiera definicję typu struktury `struct sysfs_ops`.

Listing 6: Definicja typu struktury `struct sysfs_ops`

```
struct sysfs_ops {
        ssize_t (*show)(struct kobject *, struct attribute *, char *);
        ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
};
```

Zawiera ona dwa wskaźniki na funkcje, które pełnią rolę metod obiektu. Pierwsza funkcja jest wywoływana kiedy plik związany z obiektem jądra jest odczytywany z przestrzeni użytkownika. Adres obiektu jest przekazywany jej jako pierwszy argument wywołania, jako drugi przekazywany jest adres atrybutu, który reprezentowany jest jako wspomniany plik w systemie plików `sysfs`. Trzeci argument jej wywołania to adres bufora o wielkości 4 KiB, do którego należy skopiować dane z pliku. Druga funkcja jest wywoływana, gdy plik jest zapisywany z przestrzeni użytkownika. Pierwsze trzy argumenty jej wywołania są podobne do tych opisanych wcześniej, ale pojawia się czwarty argument. Jest nim liczba bajtów, które mają być zapisane w pliku. Funkcja ta musi skopiować dane z bufora, o rozmiarze określonym wartością jej czwartego parametru, do odpowiedniej zmiennej w module jądra. Obie te funkcję muszą być zdefiniowane przez programistę, którego moduł korzysta z pliku w systemie `sysfs`.

Listing 7 zawiera definicję typu struktury `struct attribute`, która reprezentowana jest przez pliki w systemie `sysfs`.

Listing 7: Definicja struktury `struct attribute`

```
struct attribute {
        const char              *name;
        umode_t                 mode;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        bool                    ignore_lockdep:1;
        struct lock_class_key   *key;
        struct lock_class_key   skey;
#endif
};
```

Interesujące nas pola w tej strukturze to `name`, które jest nazwą atrybutu i tym samym nazwą pliku oraz `mode`, które określa prawa dostępu do pliku.

Jeżeli z jakiś powodów nie chcemy definiować klasy obiektów jądra, ale chcemy w module korzystać z plików w systemie `sysfs`, to możemy użyć skorzystać ze struktury typu `struct kobj_attribute`. Definicja tego typu jest zawarta w listingu 8.

Listing 8: Definicja typu struktury `struct kobj_attribute`

```
struct kobj_attribute {
        struct attribute attr;
        ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf);
        ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr,
                         const char *buf, size_t count);
};
```

Struktura ta zawiera pole typu `struct attribute`, które reprezentowane jest w systemie `sysfs` jako plik oraz dwa pola wskaźnikowe na funkcje, które wywoływane są, gdy z przestrzeni użytkownika wykonywany jest odpowiednio: odczyt i zapis pliku. Proszę zauważyć, że typy wskaźników tych funkcji są podobne do tych ze struktury typu `struct sysfs_ops`. Różni jej jedynie typu drugiego parametru funkcji. Zamiast `struct attribute` jest to `struct kobj_attribute`.

Do obsługi obiektów jądra służą następujące funkcje[4], które stają się dostępne po włączeniu do kodu modułu pliku nagłówkowego `linux/kobject.h`:

`void kobject_init(struct kobject *kobj, struct kobj_type *ktype)` - funkcja ta służy do powiązania obiektu jądra, do którego wskaźnik jest jej przekazywany jako pierwszy argument wywołania,

[4] Podobnie jak w przypadku przykładowego modułu dla systemu plików `procfs` w instrukcji opisane są głównie funkcje wykorzystane w przykładzie.

ze strukturą określającą typ tego obiektu, do której wskaźnik jest jej przekazywany jako drugi argument wywołania. Funkcja nic nie zwraca.

`int kobject_add(struct kobject *kobj, struct kobject *parent, const char *fmt, …)` - funkcja ta wykonuje dwie czynności. Po pierwsze nadaje obiektowi jądra, do którego wskaźnik jest jej przekazywany jako pierwszy argument wywołania, nazwę określoną za pomocą ciągu formatującego `fmt`. Ciąg ten jest używany tak jak np. w przypadku funkcji `printf()` znanej z przestrzeni użytkownika. Drugą operacją wykonywaną przez tę funkcję jest dodanie wspomnianego obiektu jądra do struktury opisującej hierarchię wszystkich takich obiektów utworzonych w jądrze systemu. Drugi argument wywołania tej funkcji wskazuje na obiekt, który będzie rodzicem obiektu wskazywanego przez pierwszy argument wywołania funkcji. Ponieważ w systemie plików `sysfs` obiekty jądra są odwzorowywane w postaci katalogów, to obiekt wskazywany przez pierwszy argument funkcji będzie w takiej sytuacji podkatalogiem drugiego obiektu. Jeśli drugi argument wywołania opisywanej funkcji będzie miał wartość `NULL`, to możliwe są dwa scenariusze: jeśli obiekt wskazywany przez pierwszy argument funkcji przypisany jest do określonego zbioru, to zostanie on dowiązany do obiektu tego zbioru, a jeśli nie, to będzie on widoczny jako katalog w głównym katalogu systemu `sysfs`. Funkcja zwraca ujemną liczbę w przypadku wystąpienia wyjątku.

`extern struct kobject * __must_check kobject_create_and_add(const char *name, struct kobject *parent)` - funkcja ta tworzy obiekt jądra, nadaje mu nazwę i umieszcza go w hierarchii obiektów jądra. Jako pierwszy argument wywołania przyjmuje ciąg znaków będący nazwą obiektu, jako drugi wskaźnik do obiektu, który będzie jego rodzicem w hierarchii. Wobec tego argumentu obowiązują te same reguły, które zostały opisane w funkcji `kobject_add()`. Obiekt jądra jest tworzony przez tę funkcję w sposób dynamiczny, tj. przydzielana jest pamięć na tę zmienną. Jeśli tworzenie obiektu się nie powiedzie funkcja zwróci wartość `NULL`, a w przeciwnym przypadku adres nowo utworzonego obiektu. Znacznik (makro) `__must_check` jest informacją dla kompilatora, aby sprawdził, czy wynik funkcji jest przypisywany do zmiennej.

`void kobject_put(struct kobject *kobj)` - funkcja ta jest zmniejsza licznik referencji (odwołań) do obiektu jądra, do którego wskaźnik jest jej przekazany jako argument wywołania. Jeśli ten licznik się wyzeruje, to funkcja ta usuwa wspomniany obiekt jądra.

`void kobject_del(struct kobject *kobj)` - funkcja ta usuwa obiekt jądra, na który wskaźnik jest jej przekazywany jako argument wywołania, z hierarchii wszystkich obiektów jądra i wywołuje funkcję `kobject_put()`.

Funkcje i makra, które obsługują atrybuty obiektów jądra, widoczne jako pliki w systemie plików `sysfs`, stają się dostępne po włączeniu do kodu modułu pliku nagłówkowego `linux/sysfs.h`. Oto opisy niektórych z nich:

`__ATTR(_name, _mode, _show, _store)` - makro to inicjuje strukturę typu `struct kobj_attribute`, reprezentującą strukturę, która jest w systemie `sysfs` widoczna jako plik. Pierwszym argumentem makra jest nazwa pliku, drugim liczba ósemkowa określająca prawa dostępu do pliku, albo wyrażenie złożone ze stałych odpowiadających poszczególnym prawom. Zgodnie z tym, co napisano wcześniej w instrukcji, nie jest możliwe w systemie `sysfs` nadanie praw do modyfikacji pliku zwykłemu użytkownikowi. Dwa ostatnie argumenty to wskaźniki odpowiednio do funkcji wywoływanej podczas odczytu pliku (jest to implementacja funkcji `show()`) i podczas zapisu (implementacja funkcji `store()`. Sposób implementacji tych funkcji również został wcześniej opisany w instrukcji.

`int __must_check sysfs_create_file(struct kobject *kobj, const struct attribute *attr)` - funkcja ta dodaje do obiektu jądra wskazywanego przez jej pierwszy argument wywołania, atrybut opisany strukturą typu `struct attribute`, wskazywaną przez drugi argument jej wywołania. Tym samym tworzy ona plik w systemie plików `sysfs` będący plikiem umieszczonym w katalogu reprezentowanym przez wspomniany obiekt jądra. W przypadku niepowodzenia dodawania atrybutu do obiektu funkcja zwraca liczbę ujemną, będącą kodem wyjątku.

`void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr)` - funkcja ta usuwa atrybut opisany strukturą, na wskaźnik jest jej przekazywany przez drugi argument wywołania z obiektu jądra, na który wskaźnik jest jej przekazywany jako pierwszy argumenty wywołania. Tym samym usuwa ona plik związany z atrybutem z katalogu, związanego ze wspomnianym obiektem jądra.

2.2. Przykłady

W pierwszym przykładzie, którego kod jest umieszczony w listingu 9, moduł jądra tworzy pliku w systemie plików **sysfs**, w którym procesy z przestrzeni użytkownika mogą zapisać pojedynczą liczbę całkowitą, lub ją odczytać. Liczba ta jest umieszczana w zmiennej modułu, a więc w ten sposób procesy użytkownika mogą mu przekazywać informację, jak również odbierać od niego informację.

Listing 9: Moduł korzystający z sysfs

```
#include<linux/module.h>
#include<linux/sysfs.h>
#include<linux/kobject.h>

static struct kobject *kernel_object;
static int number;

static ssize_t
number_show(struct kobject *kernel_object, struct kobj_attribute *attribute, char *buffer)
{
        return sprintf(buffer,"%d\n",number);
}

static ssize_t
number_store(struct kobject *kernel_object, struct kobj_attribute *attribute, const char *buffer,
                size_t count)
{
        sscanf(buffer,"%d",&number);
        return count;
}

static struct kobj_attribute number_kattribute =
__ATTR(number,0664,number_show,number_store);

static struct attribute *number_attribute = &number_kattribute.attr;

static int __init sysfs_test_init(void)
{
        kernel_object = kobject_create_and_add("test",&THIS_MODULE->mkobj.kobj);
        if(!kernel_object)
                goto err;

        if(sysfs_create_file(kernel_object,number_attribute))
                goto err1;

        return 0;
        err:
                printk(KERN_ALERT "Could not create a kobject!\n");
                return -ENOMEM;
        err1:
                kobject_del(kernel_object);
                printk(KERN_ALERT "Could not create a sysfs file!\n");
                return -ENOMEM;
}

static void __exit sysfs_test_exit(void)
{
        sysfs_remove_file(kernel_object,number_attribute);
        if(kernel_object)
                kobject_del(kernel_object);
}
```

```
52
53 module_init(sysfs_test_init);
54 module_exit(sysfs_test_exit);
55 MODULE_DESCRIPTION("A kernel module demonstrating the usage of sysfs.");
56 MODULE_LICENSE("GPL");
```

Wiersz nr 5 kodu modułu zawiera wskaźnik na obiekt jądra, który zostanie utworzony dynamicznie, a wiersz nr 6 zawiera deklarację zmiennej `number`, której wartość będzie odczytywana i zapisywana z przestrzeni użytkownika przy pomocy utworzonego przez moduł pliku w systemie plików `sysfs`.

Wiersze 8-12 zawierają definicję funkcji `number_show()`, która wywoływana jest, gdy plik w systemie plików `sysfs` jest odczytywany. W ciele tej funkcji (wiersz nr 12), przy pomocy funkcji `sprintf()`, liczba umieszczona w zmiennej `number` jest zamieniana na ciąg znaków i umieszczana w buforze wskazywanym przez parametr funkcji o nazwie `buffer`. Funkcja `show_number()` zwraca ile bajtów zapisała do tego bufora, czyli informację zwróconą przez `sprintf()` i kończy działanie.

Wiersze 14-20 zawierają definicję funkcji `number_store()`, która zapisuje w zmiennej `number` liczbę przekazaną jej z przestrzeni użytkownika w postaci ciągu znaków, w buforze wskazywanym przez jej parametr `buffer`. Konwersja łańcucha znaków na wartość liczbową i zapis do zmiennej wykonywany jest za pomocą funkcji `sscanf()`. Funkcja `number_store()` zwraca liczbę bajtów odczytanych ze wspomnianego bufora, która jest zapisana w jej parametrze `count` i kończy swoje działanie.

W wierszu nr 22 zdefiniowana jest struktura atrybutu typu `struct kobj_attribute`, która w wierszu nr 23 jest inicjowana z użyciem makra `__ATTR`. Ten atrybut, a więc także plik z nim związany ma nadawaną nazwę `number` oraz prawa dostępu `0664`, czyli prawo do odczytu i zapisu dla właściciela pliku i grupy, do której on należy oraz prawo odczytu dla pozostałych użytkowników. Jądro systemu nie pozwoli na zapis takiego pliku przez zwykłych użytkowników, więc ustawienie ostatnie cyfry na `6` byłoby bezcelowe. Funkcje `number_show()` oraz `number_store()` będą obsługiwały odczyt i zapis tego pliku.

W wierszu nr 25 modułu tworzony jest wskaźnik na strukturę typu `struct attribute`. Wskazuje on na pole `attr` struktury `number_kattribute`. Jego rola zostanie objaśniona w opisie konstruktora modułu.

Wiersze 27-44 zawierają kod konstruktora modułu. W wierszu nr 29 tworzony jest i inicjowany obiekt jądra przy użyciu funkcji `kobject_create_and_add()`. Jako jej pierwszy argument wywołania przekazany jest ciąg znaków „test”. Jest to nazwa nadana utworzonemu obiektowi jądra. Tę nazwę będzie miał również katalog w systemie plików `sysfs`, z którym ten obiekt będzie związany. Jak drugi argument jej wywołania przekazywany jest adres obiektu jądra związanego z modułem. Każdy załadowany do jądra moduł jest reprezentowany za pomocą struktury typu `struct module`. Jednym z jej pól jest struktura `mkobj`, która z kolei jako jedno ze swoich pól zawiera obiekt jądra `kobj`. Ten obiekt ma nadawaną taką samą nazwę, jak plik ze skompilowanym modułem (bez rozszerzenia `.ko`). Dostęp do struktury reprezentującej dany moduł w jądrze można uzyskać z poziomu jego kodu za pomocą makra `THIS_MODULE`. Zatem obiektem nadrzędnym w stosunku do obiektu tworzonego w wierszu nr 29 będzie obiekt związany z modułem. Co oznacza, że katalog o nazwie `test` będzie podkatalogiem katalogu o nazwie `sysfs_test` (taką nazwę będzie miał moduł) w systemie plików `sysfs`. Proszę zwrócić uwagę na obsługę wyjątku. Jeśli nie uda się utworzyć obiektu jądra, to sterowanie jest przekazywane przy pomocy instrukcji `goto` do miejsca w funkcji oznaczonego etykietą `err`. Tam wywoływana jest funkcja `printk()`, która umieszcza w buforze jądra stosowny komunikat i konstruktor kończy działanie zwracając kod błędu `-ENOMEM`. Używanie instrukcji `goto` do obsługi sytuacji wyjątkowych jest często stosowaną praktyką wśród programistów jądra. Należy pamiętać, aby etykieta, do której kieruje ta instrukcja znajdowała się w tej samej funkcji, to związana z nią instrukcja `goto`. W wierszu nr 33 konstruktora z utworzonym obiektem wiązany jest atrybut wskazywany przez wskaźnik `number_attribute`. Jest to dokonywane poprzez wywołanie funkcji `sysfs_create_file()` i jest jednoznaczne z utworzeniem pliku w systemie plików `sysfs`. Jako drugiego argumentu wywołania tej funkcji moglibyśmy użyć wyrażenia `&number_kattribute.attr`, ale taki zapis byłby mniej czytelny, dlatego wcześniej został zdefiniowany wskaźnik `number_attribute`, który zawiera ten adres. W przypadku pojawienia się wyjątku instrukcja `goto` przekazuje sterowanie do miejsca w konstruktorze oznaczonego etykietą `err1`. Obsługa wyjątku przebiega podobnie jak w wierszach 37-39, ale tym razem dodatkowo jest usuwany wcześniej stworzony obiekt jądra (wiersz nr 41).

Wiersze 46-51 zawierają kod destruktora modułu. W wierszu nr 48 tej funkcji usuwany jest plik związany z atrybutem `number_attribute`, a w wierszu nr 50 likwidowany jest obiekt jądra utworzony w konstruktorze.

Plik związany z atrybutem `number_kattribute` ma nazwę `number` i znajduje się w katalogu, do którego prowadzi ścieżka: `/sys/module/sysfs_test/test`. Jak wspomniano wcześniej `/sys` jest katalogiem do którego montowany jest system plików `sysfs`. Katalogu `module` gromadzi katalogi związane z poszczególnymi modułami załadowanymi do jądra. Katalog `sysfs_test` jest katalogiem związanym z opisywanym przykładowym modułem. Katalogu `test` jest z kolei katalogiem związanym z utworzonym w tym module obiektem jądra i zawiera plik `number`. Użytkownik `root` może zapisywać ten plik za pomocą polecenia powłoki `echo`. Odczyt tego pliku może być wykonany przez dowolnego użytkownika za pomocą polecenie `cat`.

Listing 10 zawiera kod innego modułu, który korzysta w ten sam sposób, co poprzedni moduł z pliku w systemie `sysfs`, ale tworzy go w inny sposób. Tym razem wykorzystuje do tego klasę obiektu.

Listing 10: Moduł korzystający z `sysfs` przy użyciu `kobj_type`

```
#include<linux/module.h>
#include<linux/sysfs.h>
#include<linux/kobject.h>
#include<linux/slab.h>
#include<linux/string.h>

static struct kobject *kernel_object;
static int number;
static const char attribute_name[] = "number";

static ssize_t number_show(struct kobject *kernel_object, struct attribute *attribute, char *buffer)
{
        return sprintf(buffer,"%d\n",number);
}

static ssize_t number_store(struct kobject *kernel_object, struct attribute *attribute, const char *buffer, size_t count)
{
        sscanf(buffer,"%d",&number);
        return count;
}

static struct sysfs_ops operations = {
        .show = number_show,
        .store = number_store,
};

static void number_release(struct kobject *kernel_object)
{
        kfree(kernel_object);
        pr_notice("Release function activated!\n");
}

static struct attribute number_attribute = {
        .name = attribute_name,
        .mode = 0600
};

static struct attribute *attributes[] = {
        &number_attribute,
        NULL
};

static struct kobj_type ktype =
{
        .release = number_release,
        .sysfs_ops = &operations,
        .default_attrs = attributes,
};

static int __init sysfs_test_init(void)
{
        kernel_object = (struct kobject *)kmalloc(sizeof(struct kobject),GFP_KERNEL);
        if(IS_ERR(kernel_object))
                goto err1;
```

```
55          memset(kernel_object,0,sizeof(struct kobject));
56          kobject_init(kernel_object,&ktype);
57          if(kobject_add(kernel_object,&THIS_MODULE->mkobj.kobj,"test%d",2))
58                  goto err2;
59
60          return 0;
61  err2:
62          kfree(kernel_object);
63          kernel_object = NULL;
64  err1:
65          pr_alert("Error adding a kobject\n");
66          return -ENOMEM;
67  }
68
69  static void __exit sysfs_test_exit(void)
70  {
71          if(kernel_object)
72                  kobject_put(kernel_object);
73  }
74
75  module_init(sysfs_test_init);
76  module_exit(sysfs_test_exit);
77  MODULE_LICENSE("GPL");
78  MODULE_DESCRIPTION("A second kernel module demonstrating the usage of sysfs.");
```

Podobnie jak w poprzednim module, obiekt jądra jest tworzony dynamicznie. Wskaźnik na ten obiekt jest deklarowany w wierszu nr 7. Dodatkowo w wierszu nr 9 tworzona jest tablica znaków, w której zapamiętywana jest nazwa pliku, którym będzie się posługiwał moduł.

Funkcje `number_show()` i `number_store()` są zdefiniowane podobnie jak w poprzednim module. Różni je jedynie typ drugiego parametru. Tym razem jest to `struct attribute` zamiast `struct kattribute`.

Wskaźniki do tych funkcji zapisywane są w odpowiednich polach struktury `operations` typu `struct sysfs_ops` (wiersze 22-25).

W wierszach 27-31 zawarta jest definicja funkcji `number_release()`, która wywoływana będzie, gdy przestanie on być używany. Dzieje się tak po wywołaniu dla tego obiektu funkcji `kobject_put()`. Zgodnie z zaleceniami programistów jądra, ta funkcja nie powinna być pusta, dlatego zwalnia pamięć przydzieloną dla obiektu jądra i umieszcza odpowiedni komunikat w buforze jądra.

W wierszach 33-36 deklarowana i inicjowana jest struktura o nazwie `number_attribute` i typie `struct attribute`. Do jej pola `name` zapisywany jest adres tablicy znaków zawierających nazwę atrybutu i tym samy nazwę związanego z nim pliku. W polu `mode` zapisywane są prawa dostępu do tego pliku. Tym razem plik będzie mógł odczytywać i zapisywać jedynie użytkownik `root`.

W wierszach 38-41 tworzona jest tablica atrybutów, ponieważ inicjacja jednego z pól klasy obiektu wymaga takiej tablicy. Będzie ona zawierała jedynie dwa elementy. Pierwszy będzie zawierał adres struktury `number_attribute`, a drugi będzie miał wartość `NULL`. Oznacza ona, że jest to ostatni element tej tablicy.

W wierszach 43-48 jest tworzona i inicjowana struktura o nazwie `ktype` i typie `struct kobj_type`, czyli inaczej klasa obiektu. Inicjowane są tylko trzy pola tej struktury: pole `default_attrs` wskazujące na tablicę atrybutów, pole `sysfs_ops` wskazujące na strukturę zawierającą wskaźniki do metod `show()` i `store()`, oraz pole `release` wskazujące na funkcje finalizującą obiekt.

W konstruktorze modułu, w wierszu nr 52 tworzony jest obiekt jądra za pomocą wywołania funkcji `kmalloc()`. Jeśli jego utworzenie się nie powiedzie, to instrukcja `goto` przekaże sterowanie do wiersza oznaczonego etykietą `err1`, gdzie znajduje się kod obsługujący ten wyjątek. Po udanym obiektu jądra, jego zawartość jest zerowana przy pomocy wywołania funkcji `memset()` (wiersz nr 55), a następnie jest on inicjowany (wiersz nr 56) za pomocą funkcji `kobject_init()` i wiązany ze struktura klasy. Potem, w wierszu nr 57 jest mu nadawana nazwa `test2` i jest on dodawany do hierarchii wszystkich obiektów jądra. Te czynności są realizowane przez funkcję `kobject_add()`. Jeśli jej wykonanie zakończy się niepowodzeniem, to instrukcja `goto` przekaże sterowanie do miejsca konstruktora oznaczonego etykieta `err2`, gdzie znajduje się kod odpowiedzialny za obsługę tego wyjątku, polegającą na zwolnieniu pamięci przydzielonej dla obiektu i wyzerowaniu wskaźnika na niego, a następnie wykonaniu tych wszystkich czynności, które są także wykonywane w obsłudze wyjątku funkcji `kmalloc()`.

W wierszach 69-73 zdefiniowano destruktor modułu. Jedyną czynnością w nim wykonywaną jest

wywołanie funkcji `kobject_put()` dla obiektu jądra. Wywołanie tej funkcji zmniejsza wartość licznika odwołań do danego obiektu. Ponieważ dla obiektu używanego przez ten moduł licznik odwołań jest ustawiany na `1`, poprzez wywołanie `kobject_init()`, a potem nie jest zmieniany, to użycie `kobject_put()` wyzeruje ten licznik i tym samym ten obiekt zostanie oznaczony jako nieużywany i usunięty z hierarchii obiektów jądra.

Plik utworzony przez moduł z listingu 10 ma taką samą nazwę jak plik tworzony przez poprzedni moduł, ale ścieżka do niego jest trochę inna. Zamiast katalogu `sysfs_test` jest `sysfs_test2`, a zamiast katalogu `test` jest `test2`. Sposób użycia tego pliku pozostał bez zmian, z tym, że prawo jego odczytu i zmiany ma tylko użytkownik `root`.

Przykłady innych modułów, korzystających z systemu plików `sysfs` można znaleźć w źródłach jądra Linuksa w podkatalogu `samples/kobject`. W pliku `kset-example.c` znajduje się kod modułu, który korzysta z mechanizmu zdarzeń użytkownika.

Zdania

1. `[2 punkty]` Zmień kod modułu z listingu 3, tak aby z pliku odczytywana była tylko jedna wartość. Skorzystaj z funkcji `single_open()` i `single_release()`. Zauważ, że w opisywanym przypadku musisz jedynie zaimplementować funkcję `show()`.

2. `[4 punktów]` Przekształć kod modułu z listingu 3 tak, aby do realizacji odczytu nie korzystał on z mechanizmu plików sekwencyjnych.

3. `[6 punktów]` Przerób kod modułu z listing 3 tak, aby bufor dla pliku był listą dwukierunkową.

4. `[2 punkty]` Pozbądź się instrukcji `goto` z konstruktora modułu, którego kod umieszczono w listingu 10, ale tak, aby zachowanie tej funkcji w przypadku pojawienia się wyjątku pozostało bez zmian.

5. `[4 punktów]` Przerób moduł z listingu 9 lub 10 tak, aby tworzył on katalog z plikiem w katalogu `/sys`.

6. `[6 punktów]` Stwórz moduł, który w podkatalogu swojego katalogu w `/sys` stworzy plik, do którego użytkownik będzie mógł zapisywać wyłącznie liczby naturalne od `1` do `7`. Wszystkie inne wartości powinny być ignorowane. Z chwilą wpisania prawidłowej liczby moduł stworzyć tyle nowych plików w podkatalogu ile wynosi ta liczba. Nazwy tych plików możesz stworzyć dodając do nazwy podstawowej kolejny numer porządkowy. W plikach moduł powinien zapisać losowe liczby. Pliki wraz z podkatalogiem powinny być usuwane przez destruktor modułu.