Laboratorium 8: „Sterowniki urządzeń znakowych” (dwa zajęcia)

dr inż. Arkadiusz Chrobot

4 maja 2024

Spis treści

Wprowadzenie

W systemie Linux, podobnie jak w innych systemach kompatybilnych z Uniksem wyróżnione są trzy główne kategorie urządzeń:

urządzenia znakowe - są to urządzenia, które typowo transmitują dane małymi porcjami, o dosyć często zmiennej długości i najczęściej w sposób sekwencyjny;

urządzenia blokowe - są to urządzenia, które typowo transmitują informacje w dużych porcjach, których rozmiar jest zazwyczaj wielokrotnością 512 bajtów i umożliwiają swobodny dostęp do danych;

urządzenia sieciowe - urządzenia, które realizują transmisję danych w sieci.

Urządzenia znakowe i blokowe są udostępniane przez jądro systemu do przestrzeni użytkownika przy pomocy specjalnych plików, nazywanych plikami urządzeń, które zazwyczaj są umieszczone w katalogu `/dev`. Każdy taki plik, oprócz nazwy posiada trzy atrybuty, których wartości można sprawdzić wykonując na takim pliku polecenie `ls -la`. Pliki związane z urządzeniami znakowymi są oznaczone literą `c`, a pliki związane z urządzeniami blokowymi mają etykietę `b`. Dodatkowo, z każdym takim plikiem związane są dwie liczby naturalne, nazywane numerem głównym (ang. *major number*) i numerem pobocznym (ang. *minor number*). Numer główny identyfikuje sterownik obsługujący określoną grupę urządzeń, a numer poboczny konkretne urządzenie należące do tej grupy. Numery poboczne mogą są unikatowe dla pojedynczego sterownika, ale powtarzają się między różnymi sterownikami. Numery główne są unikatowe w obrębie jednej klasy urządzeń (np. znakowych), ale powtarzają się między klasami.

W przestrzeni jądra obsługa wszystkich trzech kategorii urządzeń jest powiązana z Wirtualnym Systemem Plików (ang. *Virtual File System - VFS*). Każde żądanie dostępu do urządzenia, inicjowane przez proces lub wątek użytkownika jest przekazywane do przestrzeni jądra poprzez wywołania systemowe, następnie jest kojarzone z odpowiednimi obiektami w podsystemie VFS i w zależności od kategorii urządzania podlega dalszemu przetwarzaniu lub jest bezpośrednio przekazywane do sterownika urządzenia, który je realizuje.

Niniejsza instrukcja dotyczy tworzenia sterowników dla urządzeń znakowych. Rozdział 1 poświęcony jest opisowi typowej struktury i zachowania sterownika urządzenia znakowego. Rozdział 2 zawiera opis API wykorzystywanego do implementacji sterowników urządzeń znakowych. Rozdział 3 wyjaśnia rolę procesu użytkownika o nazwie `udev` w obsłudze urządzeń znakowych i blokowych, a rozdział 4 zawiera listing przykładowego sterownika urządzenia znakowego. Ze względu na środowisko używane na zajęciach laboratoryjnych nie jest to fizyczne urządzenie, ale pseudo urządzenie, tzn. takie, którego działanie jest całkowicie symulowane przez oprogramowanie. Instrukcja kończy się listą zadań do samodzielnej realizacji w ramach zajęć laboratoryjnych.

1. Sterowniki urządzeń znakowych

Sterowniki urządzeń znakowych korzystają z dwóch obiektów VFS: obiektu pliku i obiektu i-węzła. Każdy sterownik urządzenia znakowego ma do wykonania dwa podstawowe zadania, jakim jest inicjacja

współpracy urządzenia z jądrem systemu i dostarczenie metod obiektu pliku realizujących operacje na urządzeniu, które będą uruchamiane w ramach wywołań systemowych.

W przypadku fizycznych urządzeń inicjacja może oznaczać, np. dostarczenie zasilania do urządzenia i wykonania jego diagnostyki. Sterownik jest także odpowiedzialny za dostarczenie i rejestrację procedury obsługi przerwań zgłaszanych przez to urządzenie oraz związanych z nią mechanizmów dolnych połówek. Ponadto, w kodzie metod obiektu pliku musi być uwzględnione, to że operacje na fizycznym urządzeniu mogą wymagać oczekiwania na ich zakończenie oraz być wykonywane współbieżnie. Koniecznym zatem staje się użycie odpowiednich środków synchronizacji, które będą także pozwalały przerwać oczekiwanie na zakończenie operacji, jeśli proces lub wątek użytkownika, który ją zapoczątkował, otrzyma sygnał.

Obsługa pseudo urządzeń znakowych jest prostsza, a ponad to w dużej mierze opiera się na tych samych mechanizmach, co obsługa urządzeń fizycznych. Jej podstawowe elementy to:

1. pozyskanie numeru głównego i pobocznego,
2. zainicjowanie i dodanie do jądra struktury typu `struct cdev`,
3. zainicjowanie i dodanie do jądra struktur związanych z systemem plików `sysfs`,
4. zainicjowanie i dodanie do jądra struktury metod obiektu pliku.

Jeśli urządzenie umożliwia współbieżny dostęp, to musi to być uwzględnione w kodzie sterownika. Bardzo często twórcy sterowników definiują własne struktury, które zawierają wszystkie informacje dotyczące urządzenia i jego stanu. Jeśli sterownik ma obsługiwać lub symulować działanie większej liczby takich urządzeń, to każde z nich powinno mieć własną taką strukturę. Zadaniem sterownika jest także określenie ile procesów lub wątków użytkownika może jednocześnie korzystać z urządzenia. Jego twórca powinien zatem zadbać, o obsługę współbieżności lub zastosować środki, które tę współbieżność wykluczają.

2. Opis API sterowników urządzeń znakowych

Z punktu widzenia twórcy sterownika urządzenia znakowego, najbardziej istotne są trzy typy struktur: `struct cdev`, który definiuje strukturę reprezentującą urządzenie znakowe w jądrze systemu, `struct file`, który definiuje strukturę opisującą atrybuty obiektu pliku oraz `struct file_operations`, która definiuje wskaźniki na funkcje realizujące operacje na obiekcie pliku, czyli na metody związane z tym obiektem. Obiekt pliku reprezentuje w systemie otwarte pliki, a dwie ostatnie struktury definiują jego klasę. Sterownik urządzenia znakowego może także korzystać z czwartej struktury, jaką jest struktura typu `struct inode`, która definiuje atrybuty obiektu i-węzła.

Definicję typu struktury `cdev` przedstawia listing 1. Większość pól zawartych w tej strukturze jest inicjowana przy pomocy odpowiednich funkcji i makr, które będą opisane dalej w instrukcji. Pole `dev` zawiera numer urządzenia, który składa się z numeru głównego i pobocznego, pole `ops` to pole zawierające wskaźnik na strukturę metod obiektu pliku, pole `kobj` jest obiektem jądra, związanym z modelem urządzeń i systemem plików `sysfs`, który jest opisany w czwartej instrukcji. Programista piszący moduł jądra musi pamiętać o bezpośredniej inicjacji pola `owner`, będącego wskaźnikiem na strukturę reprezentującą moduł, w którym struktura typu `struct cdev` została zadeklarowana.

Listing 1: Definicja struktury `struct cdev`

```
1 struct cdev {
2         struct kobject kobj;
3         struct module *owner;
4         const struct file_operations *ops;
5         struct list_head list;
6         dev_t dev;
7         unsigned int count;
8 };
```

Listing 2 przedstawia definicję typu `struct file`, która przekazywana jest przez VFS do większości funkcji wskazywanych przez pola struktury typu `struct file_operations`. Twórcy sterowników najczęściej korzystają z pola wskaźnikowego `private_data` tej struktury. Zgodnie z nazwą może ono wskazywać

na obszar pamięci zawierający lokalne (prywatne) dane sterownika. Jeśli ten obszar jest przydzielany dynamicznie, to należy pamiętać o jego zwolnieniu przed usunięciem sterownika z jądra systemu. Inne pola, takiej jak `f_op` - wskaźnik na strukturę metod obiektu pliku, `f_flags` - flagi otwarcia pliku, `f_mode` - tryb dostępu do pliku, czy `f_pos` - wskaźnik pliku, też mogą być przydatne.

Listing 2: Definicja struktury `struct file`

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
        struct inode            *f_inode;       /* cached value */
        const struct file_operations    *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
        struct list_head        f_ep_links;
        struct list_head        f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

Definicja typu struktury `struct file_operations` jest przedstawiona przez listing 3. Jej pola są wskaźnikami na funkcje, które stanowią metody obiektu pliku, są uruchamiane z poziomu wywołań systemowych i bezpośrednio zajmują się obsługą urządzenia. Twórca sterownika nie musi definiować ich wszystkich. Najważniejsze z nich, to `open()`, `release()`, `read()` i `write()`. Niektóre urządzenia znakowe oferują swobodny dostęp do danych. W ich przypadku jest definiowana również funkcja `llseek()`. Sposób definiowania tych metod będzie opisany bardziej szczegółowo w dalszej części instrukcji. Definicje pozostałych są rzadziej spotykane w sterownikach urządzeń znakowych, ale warte odnotowania są następujące: `unlocked_ioctl()` i `compat_ioctl()` - metody uruchamiane z poziomu wywołania `ioctl()`, które pozwalają zrealizować operacje na urządzeniu, nie dające się wprost zaimplementować przy pomocy pozostałych metod, `poll()` i `fasync()` - metody powiadamiające procesy/wątki użytkownika o pojawieniu się nowych danych w urządzeniu znakowym.

Listing 3: 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
};
```

Nie wszystkie metody, które może wskazywać struktura typu `struct file_operations` muszą być zaimplementowane. W skrajnych przypadkach wystarczy zaimplementować jedną z nich, np. `read()`. Najczęściej definiowane z nich muszą spełniać następujące wymagania:

`int (*open) (struct inode *, struct file *)` - metoda wskazywana przez to pole jest uruchamiana z poziomu wywołania systemowego `open()`, czyli za każdym razem, gdy plik urządzenia jest otwierany z poziomu przestrzeni użytkownika. Może ona, ale nie musi, korzystać ze struktur, których adresy są jej przekazywane przez parametry. W przypadku tego drugiego wariantu najczęściej metoda ta korzysta z pola `private_data` struktury typu `struct file`, której adres otrzymuje przez drugi parametr. Rola tego pola była objaśniana wcześniej. To właśnie w opisywanej metodzie alokowana jest pamięć, której adres jest zapisywany w tym polu. Wskazywany przez nie obszar pamięci może posłużyć jako dogodny punkt wymiany informacji między pozostałymi metodami zdefiniowanymi w sterowniku. Dodatkowo metoda ta wykonuje wszystkie prace inicujące współpracę między oprogramowaniem a urządzeniem sprzętowym lub pseudo urządzeniem. Jeśli jej działanie zakończy się sukcesem, to powinna ona zwrócić wartość `0`. W przypadku niepowodzenia metoda ta może zwrócić wartość `-EBUSY` oznaczającą, że urządzenie nie jest gotowe do użycia lub, że inny proces/wątek z przestrzeni użytkownika jest w jego posiadaniu.

`int (*release) (struct inode *, struct file *)` - metoda wskazywana przez to pole jest uruchamiana z poziomu wywołania systemowego `close()`, czyli wówczas, gdy plik urządzenia jest zamykany z poziomu przestrzeni użytkownika. Podobnie jak metoda wskazywana przez wskaźnik `open`, może ona, ale nie musi korzystać z przekazanych jej argumentów wywołania. Najczęściej metoda ta przeprowadza czynności związane z finalizacją pracy urządzenia, np. wyłącznie zasialania, zwolnienie

pamięci wskazywanej przez pole `private_data` struktury typu `struct file`. Pomyślne zakończenie działania powinno być sygnalizowane przez tę funkcję zwróceniem wartości `0`, wyjątki są zazwyczaj sygnalizowane liczbami ujemnymi.

`ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)` - metoda wskazywana przez ten wskaźnik jest uruchamiana z poziomu wywołania systemowego `read()` i realizuje odczyt danych z urządzenia. Przez pierwszy parametr do tej metody jest przekazywany adres struktury typu `struct file`, drugi parametr jest wskaźnikiem do bufora w przestrzeni użytkownika, w którym metoda powinna zapisać odczytane dane. Trzeci zawiera rozmiar żądanych przez przestrzeń użytkownika danych, a trzeci jest wskaźnikiem do zmiennej, która jest wskaźnikiem pliku. Ostatni parametr jest często używany do określenia, które dane mają być odczytane i służy także do sprawdzenia, czy żądanie to nie wykracza poza rozmiar obsługiwanego urządzenia. Pierwszy wykorzystywany jest do pozyskiwania wskaźnika na prywatne dane. Drugi parametr powinien być obsługiwany za pośrednictwem funkcji `copy_to_user()`, gdyż sprawdza ona poprawność tego wskaźnika. Metoda ta powinna zwrócić ilość odczytanych danych, najczęściej wyrażaną liczbą bajtów. W przypadku zakończenia odczytu powinna zwrócić wartość `0`. Wyjątki są sygnalizowane przykładowymi wartościami: `-EFAULT` - błędny wskaźnik bufora w przestrzeni użytkownika, `-EIO` - ogólny błąd wejścia-wyjścia, `-EINTR` - odczyt przerwany przez sygnał.

`ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)` - wskaźnik ten wskazuje na metodę uruchamianą z poziomu wywołania systemowego `write()` i realizuje zapis danych do urządzenia. Znaczenie jej parametrów jest takie samo, jak w przypadku metody wskazywanej przez wskaźnik `read`. Zmienia się jedynie charakter bufora wskazywanego przez drugi parametr, z bufora wyjściowego na bufor wejściowy, gdyż zawiera on dane, które mają być zapisane do urządzenia. Powinien on być obsługiwany za pomocą funkcji `copy_from_user()`. Metoda powinna zwracać te same wartości co ta, wskazywana przez `read`, z tą poprawką, że ilość dotyczy zapisanych, a nie odczytanych danych.

`loff_t (*llseek) (struct file *, loff_t, int)` - funkcja wskazywana przez ten wskaźnik jest uruchamiana z poziomu wywołania systemowego `lseek()` i jej zadaniem jest zmiana wartości wskaźnika pliku. Przez jej pierwszy parametr jest przekazywany adres obiektu pliku, przez drugi nowa wartość wskaźnika pliku, a przez trzeci jedna z trzech stałych: `SEEK_SET` - nowa wartość wskaźnika pliku jest liczona jest względem jego początku, co oznacza, że powinien on być ustawiony na taką wartość, jaka została przekazana metodzie przez drugi parametr, po uprzednim sprawdzeniu jej poprawności, `SEEK_CUR` nowa wartość wskaźnika pliku jest liczona względem jego bieżącej wartości, tzn. jest on ustawiany na sumę jego wartości bieżącej i tej przekazanej metodzie przez drugi parametr, o ile ta suma jest poprawna, `SEEK_END` nowa wartość wskaźnika jest liczona względem końca pliku, tzn. od maksymalnej wartości wskaźnika dla pliku odejmowana jest wartość przekazana przez drugi parametr i jeśli ta różnica jest poprawna, to na tę wartość wskaźnik pliku jest ustawiany. Metoda zwraca nową wartość wskaźnika pliku, lub wartość `-EINVAL`, jeśli nowa wartość byłaby niepoprawna.

Listing 4 przestawia strukturę typu `struct inode`, która razem ze strukturą metod, czyli strukturą typu `struct inode_operations` definiuje klasę obiektów i-węzłów. Drugi rodzaj struktur nie będzie opisywany w tej instrukcji, gdyż wykracza to poza jej tematykę. Sterowniki urządzeń znakowych typowo korzystają jedynie ze struktury typu `struct inode`, a konkretniej z jej pól `i_rdev` i `i_cdev`. Pierwsze jest polem typu `dev_t` i zawiera numer urządzenia. Ponieważ sterowniki najczęściej są zainteresowane nie całym numerem, ale numerem głównym i pobocznym, to to pole nie jest bezpośrednio odczytywane, ale za pomocą funkcji, które będą opisane dalej. Drugie wspomniane pole zawiera wskaźnik na strukturę typu `i_cdev` i może być użyte np. przez metodę `open()` lub `release()` do ustalenia, które z urządzeń obsługiwanych przez sterownik musi być obsłużone w danym wywołaniu.

Listing 4: Definicja struktury `struct inode`

```
1 truct inode {
2         umode_t                 i_mode;
3         unsigned short          i_opflags;
```

```
        kuid_t                  i_uid;
        kgid_t                  i_gid;
        unsigned int            i_flags;

#ifdef CONFIG_FS_POSIX_ACL
        struct posix_acl        *i_acl;
        struct posix_acl        *i_default_acl;
#endif

        const struct inode_operations   *i_op;
        struct super_block      *i_sb;
        struct address_space    *i_mapping;

#ifdef CONFIG_SECURITY
        void                    *i_security;
#endif

        /* Stat data, not accessed from path walking */
        unsigned long           i_ino;
        /*
         * Filesystems may only read i_nlink directly.  They shall use the
         * following functions for modification:
         *
         *    (set|clear|inc|drop)_nlink
         *    inode_(inc|dec)_link_count
         */
        union {
                const unsigned int i_nlink;
                unsigned int __i_nlink;
        };
        dev_t                   i_rdev;
        loff_t                  i_size;
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        spinlock_t              i_lock; /* i_blocks, i_bytes, maybe i_size */
        unsigned short          i_bytes;
        unsigned int            i_blkbits;
        blkcnt_t                i_blocks;

#ifdef __NEED_I_SIZE_ORDERED
        seqcount_t              i_size_seqcount;
#endif

        /* Misc */
        unsigned long           i_state;
        struct mutex            i_mutex;

        unsigned long           dirtied_when;   /* jiffies of first dirtying */
        unsigned long           dirtied_time_when;

        struct hlist_node       i_hash;
        struct list_head        i_io_list;      /* backing dev IO list */
#ifdef CONFIG_CGROUP_WRITEBACK
        struct bdi_writeback    *i_wb;          /* the associated cgroup wb */

        /* foreign inode detection, see wbc_detach_inode() */
        int                     i_wb_frn_winner;
        u16                     i_wb_frn_avg_time;
        u16                     i_wb_frn_history;
```

```
#endif
        struct list_head        i_lru;          /* inode LRU list */
        struct list_head        i_sb_list;
        union {
                struct hlist_head       i_dentry;
                struct rcu_head         i_rcu;
        };
        u64                     i_version;
        atomic_t                i_count;
        atomic_t                i_dio_count;
        atomic_t                i_writecount;
#ifdef CONFIG_IMA
        atomic_t                i_readcount; /* struct files open RO */
#endif
        const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
        struct file_lock_context        *i_flctx;
        struct address_space    i_data;
        struct list_head        i_devices;
        union {
                struct pipe_inode_info  *i_pipe;
                struct block_device     *i_bdev;
                struct cdev             *i_cdev;
                char                    *i_link;
        };

        __u32                   i_generation;

#ifdef CONFIG_FSNOTIFY
        __u32                   i_fsnotify_mask; /* all events this inode cares about */
        struct hlist_head       i_fsnotify_marks;
#endif

        void                    *i_private; /* fs or device private pointer */
};
```

Oprócz opisanych wcześniej typów struktur, które zdefiniowane są w plikach nagłówkowych `linux/fs.h` i `linux/cdev.h`, sterowniki urządzeń znakowych korzystają także z makr i funkcji, które są dostępne po włączeniu do kodu wyżej wymienionych plików oraz pliku `linux/device.h`. Poniżej zamieszczone są opisy najważniejszych z nich.

`MKDEV(ma,mi)` - makro, które tworzy i zwraca numer urządzenia (wartość typu `dev_t` na podstawie przekazanych mu numerów: głównego (`ma`) i pobocznego (`mi`)).

`MAJOR(dev)` - makro, które z przekazanego mu numeru urządzenia odczytuje i zwraca numer główny.

`MINOR(dev)` - makro, które z przekazanego mu numer urządzenia odczytuje i zwraca numer poboczny.

`int register_chrdev_region(dev_t, unsigned, const char *)` - funkcja, które rezerwuje na potrzeby sterownika określony zakres numerów urządzeń. Jako pierwszy argument wywołania jest jej przekazywany pierwszy numer z zakresu do zarezerwowania, jako drugi liczba tych numerów, a jako trzeci wskaźnik na ciąg znaków będący nazwą urządzenia. Pierwszy numer urządzenia zazwyczaj tworzy się przy pomocy makra `MKDEV`, określając numer główny, a jako numer poboczny podając `0`. Zatem problem polega na znalezieniu pierwszego wolnego numeru głównego. Jeśli sterownik ma być używany tylko na jednym komputerze, to można sprawdzić już zarezerwowane numery główne w pliku `/proc/devices`. Jeśli natomiast ma być udostępniony do użytku publicznego, to należy się zwrócić do organizacji *The Linux Assigned Names And Numbers Authority* (w skrócie `LANANA`, strona: `http://www.lanana.org/` z prośbą o przydzielenie takiego numeru. Ponieważ ten proces jest kłopotliwy, to sterowniki częściej korzystają z następnej z opisywanych funkcji. Funk-

cja `register_chardev_region()` zwraca zero, jeśli rezerwacja się powiedzie lub liczbę ujemną w przeciwnym przypadku.

`int alloc_chrdev_region(dev_t *, unsigned, unsigned, const char *)` - funkcja przedziela określony zakres numerów urządzeń. Pierwszy numer urządzenia z tego zakresu zapisywany jest w zmiennej typu `dev_t`, której adres jest przekazywany funkcji przez jej pierwszy parametr. Przez drugi parametr przekazywany jest pierwszy numer poboczny (zazwyczaj o wartości `0`). Przez trzeci parametr przekazywana jest liczba numerów urządzeń z żądanym zakresie, a przez trzeci wskaźnik na ciąg znaków będący nazwą urządzenia. Funkcja zwraca zero jeśli przydział się powiedzie, lub liczbę ujemną w przeciwnym przypadku.

`void unregister_chrdev_region(dev_t, unsigned)` - funkcja, które wyrejestrowuje zakres numerów urządzeń przydzielony lub zarezerwowany przez wcześniej opisane funkcje. Nie zwraca ona żadnej wartości, ale przyjmuje dwa argumenty wywołania. Pierwszym jest pierwszy numer urządzenia z przydzielonego zakresu, a drugim liczba numerów w tym zakresie.

`void cdev_init(struct cdev *, const struct file_operations *)` - funkcja, która inicjuje strukturę typu `struct cdev`. Jako pierwszy argument jej wywołania jest przekazywany adres tej struktury, a jako drugi adres struktury typu `struct file_operations` (struktury metod obiektu pliku). Funkcja ta nic nie zwraca.

`int cdev_add(struct cdev *, dev_t, unsigned)` - funkcja, która dodaje strukturę typu `struct cdev` do systemu. Adres tej struktury jest przekazywany opisywanej funkcji jako pierwszy argument jej wywołania. Struktura ta reprezentuje w jądrze systemu urządzenie znakowe i dołączana jest do listy, która gromadzi struktury wszystkich takich urządzeń. Pojedyncza struktura jest tworzona dla każdego obsługiwanego przez sterownik urządzenia z osobna. Drugim argumentem wywołania funkcji jest pierwszy numer urządzenia przypisany danemu urządzeniu, a trzecim liczba kolejnych takich numerów, które związane są z danym urządzeniem. Funkcja zwraca zero jeśli jej działanie zakończy się pomyślnie lub liczbę ujemną w przeciwnym przypadku.

`void cdev_del(struct cdev *)` - funkcja, która usuwa strukturę typu `struct cdev` z listy wszystkich takich struktur. Nie zwraca ona żadnej wartości.

`class_create(owner, name)` - makro, które tworzy katalogi i pliki związane ze sterownikiem w katalogu `/sys` oraz strukturę typu `struct class`, której adres zwraca. Jako jego argumenty przekazywane są odpowiednio: adres struktury reprezentującej w systemie moduł oraz łańcuch znaków będący nazwą urządzenia.

`void class_destroy(struct class *cls)` - funkcja, która zwalnia pamięć na strukturę utworzoną przez makro `class_create`. Adres tej struktury jest przekazywany jako jej argument wywołania.

`struct device *device_create(struct class *cls, struct device *parent, dev_t devt, void *drvdata, const char *fmt, …)` - funkcja tworząca i zwracająca strukturę typu `struct device`, a przede wszystkim wysyłająca komunikaty do demona `udevd`, o konieczności utworzenia plików urządzeń. Jako pierwszy argument wywołania przyjmuje ona wskaźnika na strukturę typu `struct class`. Drugim jej argumentem jest adres struktury typu `struct device`, która będzie strukturą macierzystą dla nowej struktury. Jeśli takiej nie ma to ten argument ma wartość `NULL`. Trzeci argument to numer urządzenia. Czwartym argumentem jest wskaźnik na dane dla funkcji wywoływanych zwrotnie. Najczęściej jego wartością jest także `NULL`. Piątym argumentem jest łańcuch znaków zawierający nazwę urządzenia. Może on zawierać ciągi formatujące, więc po tym argumencie mogą opcjonalnie występować inne argumenty w liczbie i typach określonymi liczbą i rodzajem tych ciągów.

`void device_destroy(struct class *cls, dev_t devt)` - funkcja, która usuwa strukturę typu `struct device` i wysyła komunikaty do demona `udevd` z przestrzeni użytkownika, o konieczności usunięcia odpowiednich plików urządzeń. Przyjmuje dwa argumenty wywołania. Pierwszym jest adres

struktury typu `struct class`, a drugim numer urządzenia.

`unsigned imajor(const struct inode *inode)` - funkcja `inline`, która odczytuje numer główny z pola `i_rdev` obiektu i-węzła i go zwraca. Jako argument wywołania przyjmuje adres obiektu i-węzła.

`unsigned iminor(const struct inode *inode)` - funkcja `inline`, która odczytuje numer poboczny z pola `i_rdev` obiektu i-węzła i go zwraca. Jako argument wywołania przyjmuje adres obiektu i-węzła.

3. Konfiguracja `Udev`

W opisie funkcji `device_create()` i `device_destroy()` wspomniano o tym, że wysyłają one komunikaty do przestrzeni użytkownika. Te komunikaty są przetwarzane przez system, który nazywa się `udev`. Jego najważniejszym elementem jest demon[1] `udevd`, który reaguje na te komunikaty tworząc lub usuwając pliki urządzeń. Domyślnie te pliki tworzone są w katalogu `/dev` i dostęp do nich ma tylko użytkownik `root`. Można jednak wskazać temu demonowi jakie atrybuty powinien mieć taki plik po utworzeniu, tworząc plik z odpowiednimi regułami konfiguracyjnymi i umieszczając go w katalogu `/etc/udev/rules.d/`. Listing 5 zawiera zestaw takich reguł dla pseudo urządzenia znakowego tworzonego przez sterownik, którego kod źródłowy zawiera listing 6. Liczba będąca przedrostkiem nazwy tego pliku określa w jakiej kolejności, w stosunku do innych plików z regułami, ten plik będzie przetwarzany przez `udevd`. Token `KERNEL` określa nazwę komunikatu, którego dotyczą dalsze tokeny. Token `NAME` określa nazwę pliku urządzenia. Token `OWNER` określa nazwę właściciela pliku urządzenia. Ponieważ sterownik był pisany dla Linuksa Raspbian, to właścicielem będzie użytkownik `pi`. Token `MODE` określa tryb dostępu do pliku urządzenia i w tym przypadku jest to zapis i odczyt dla właściciela. Istnieją jeszcze inne tokeny, które nie zostały użyte w tym pliku. Ich opis, a także opis tworzenia reguł można znaleźć w podręczniku systemowym dostępnym po wydaniu polecenia `man udev`.

Listing 5: Plik konfiguracyjny `41-fibdev.rules` dla demona `udevd`

```
KERNEL=="fibdev", NAME="fibdev", OWNER="pi", MODE="0660"
```

4. Przykład

Listing 6 zawiera kod źródłowy pseudo urządzenia znakowego, które generuje kolejne wyrazy ciągu Fibonacciego, dopóki ich wartość nie przekracza górnej granicy zakresu typu `uint64_t`. Ten typ jest typem wprowadzonym do standardu C99. Pozwala on przechowywać liczby naturalne i jego rozmiar, wynoszący 64 bity jest niezależny od platformy sprzętowej. Wyrazy tego ciąg można wyświetlić na ekranie np. poleceniem `cat /dev/fibdev`.

Listing 6: Sterownik pseudo urządzenia znakowego

```
#include<linux/module.h>
#include<linux/fs.h>
#include<linux/cdev.h>
#include<linux/device.h>
#include<linux/uaccess.h>

#define NAME "fibdev"

static uint64_t first, second;

static ssize_t fib_read(struct file *f, char __user *u, size_t size, loff_t* pos)
```

[1] Tak w terminologii uniksowej nazywa się proces-serwer. Obecnie demon `udev` jest częścią `systemd`.

```
{
        uint64_t tmp;
        char fibnum[100];
        size_t trans_unit = snprintf(fibnum,sizeof(fibnum),"%llu\n",first);
        if(trans_unit<0)
                return -EIO;
        if(copy_to_user(u,(void *)fibnum,trans_unit))
                return -EIO;

        tmp = first+second;
        if(tmp>=second) {
                first = second;
                second = tmp;
        } else
                return 0;

        return trans_unit;
}

static ssize_t fib_write(struct file *f, const char __user *u, size_t size, loff_t* pos)
{
        return 0;
}

static int fib_open(struct inode *ind, struct file *f)
{
        first = 0;
        second = 1;
        return 0;
}

static int fib_release(struct inode *ind, struct file *f)
{
        return 0;
}

static struct file_operations fibop =
{
        .owner = THIS_MODULE,
        .open = fib_open,
        .release = fib_release,
        .read = fib_read,
        .write = fib_write,
};

static dev_t number = 0;
static struct cdev fib_cdev;
static struct class *fib_class;
static struct device *fib_device;

static int __init fibchar_init(void)
{
        if(alloc_chrdev_region(&number,0,1,NAME)<0) {
                printk(KERN_ALERT "[fibdev]: Region allocation error!\n");
                return -1;
        }

        fib_class = class_create(THIS_MODULE,NAME);
        if(IS_ERR(fib_class)) {
                printk(KERN_ALERT "[fibdev]: Error creating class: %ld!\n",PTR_ERR(fib_class));
```

```
72                  unregister_chrdev_region(number,1);
73                  return -1;
74          }
75
76          cdev_init(&fib_cdev,&fibop);
77          fib_cdev.owner = THIS_MODULE;
78
79          if(cdev_add(&fib_cdev,number,1)) {
80                  printk(KERN_ALERT "[fibdev]: Error adding cdev!\n");
81                  class_destroy(fib_class);
82                  unregister_chrdev_region(number,1);
83                  return -1;
84          }
85
86          fib_device = device_create(fib_class, NULL, number, NULL, NAME);
87          if(IS_ERR(fib_device)) {
88                  printk(KERN_ALERT "[fibdev]: Error creating device: %ld!\n",PTR_ERR(fib_device));
89                  cdev_del(&fib_cdev);
90                  class_destroy(fib_class);
91                  unregister_chrdev_region(number,1);
92                  return -1;
93          }
94
95          return 0;
96  }
97
98  static void __exit fibchar_exit(void)
99  {
100         if(fib_device)
101                 device_destroy(fib_class,number);
102         cdev_del(&fib_cdev);
103         if(fib_class)
104                 class_destroy(fib_class);
105         if(number>=0)
106                 unregister_chrdev_region(number,1);
107 }
108
109 module_init(fibchar_init);
110 module_exit(fibchar_exit);
111 MODULE_LICENSE("GPL");
112 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
113 MODULE_DESCRIPTION("A pseudo character device that generates Fibonacci numbers");
114 MODULE_VERSION("1.0");
```

W wierszu nr 5 kodu źródłowego z listingu 6 włączany jest plik nagłówkowy zawierający deklarację funkcji `copy_to_user()`, która była opisywana w instrukcji poświęconej systemom plików `procfs` i `sysfs`. W wierszu nr 7 zdefiniowany jest ciąg znaków będący nazwą urządzenia, która będzie także pełniła nazwę pliku urządzenia. W wierszu nr 9 zdefiniowane są dwie zmienne, które będą służyły do generowania kolejnych wyrazów ciągu Fibonacciego. Wiersze 11-29 zawierają definicję funkcji `fib_read()`, która jest implementacją metody `read()` obiektu pliku. Wiersze 13-15 zawierają deklaracje zmiennych lokalnych. Pierwsza z nich (`tmp`) pełni rolę pomocniczą przy generowaniu kolejnych wyrazów ciągu Fibonacciego. Druga zmienna `fibnum` jest tablicą znaków, w której będzie zapisany określony wyraz wspomnianego ciągu w postaci łańcucha znaków. Konwersja tego wyrazu, który zapisany jest w zmiennej `first` wykonywana jest przez funkcję `snprintf()` w wierszu nr 15. Wynik jej działania, czyli długość łańcucha jest zapisywana w zmiennej `trans_unit`. Jeśli ta konwersja się nie powiedzie, to funkcja `snprintf()` zwróci liczbę ujemną, a funkcja `fib_read()` zakończy swoje działania sygnalizując błąd zwracaną wartością `-EIO`. Podobnie funkcja zachowa się, jeśli nie powiedzie się kopiowanie łańcucha znaków do bufora w przestrzeni użytkownika przez funkcję `copy_to_user()` (wiersz nr 18). W wierszu nr 21 wyznaczana jest wartość kolejnego wyrazu ciągu Fibonacciego, na podstawie wartości dwóch poprzednich wyrazów.

Jeśli jest ona większa lub równa wartości drugiego z nich, to znaczy, że została wyliczona poprawnie i można ją udostępnić przestrzeni użytkownika. Jeśli nie, to znaczy to, że została przekroczona górna granica zakresu typu `uint64_t`. W takim wypadku należy zakończyć generowanie ciągu Fibonacciego i metoda zwraca w związku z tym wartość `0`. Proszę zwrócić uwagę, że ostatni wyraz ciągu Fibonacciego, zawarty w zmiennej `second` nie jest udostępniany przestrzeni użytkownika.

Funkcja `fib_write()` zdefiniowana w wierszach 31-34 jest implementacją metody `write()` obiektu pliku, ale poza zwracaniem wartości `0` nie wykonuje ona dodatkowych czynności. Została ona zdefiniowana tylko po to, aby próba zapisu do urządzenia nie kończyła się błędem.

Funkcja `fib_open()` zdefiniowana w wierszach 36-41 jest implementacją metody `open()` obiektu pliku. Nadaje ona wartości początkowe zmiennym `first` i `second` oraz zwraca liczbę `0`.

Funkcja `fib_release()` zdefiniowana w wierszach 43-46 jest implementacją metody `release()` obiektu pliku. Zwraca ona wartość `0` i nie wykonuje żadnych dodatkowych czynności. Zatem zamknięcie pliku urządzenia, a dokładniej ostatnie jego zamknięcie, kończy się zawsze sukcesem.

W wierszach 48-55 jest zdefiniowana struktura `fibop` typu `struct file_operations` i są inicjowane jej pola wskaźnikowe, które wskazują na implementacje metod obiektu plikowego, oraz pole `owner`, któremu jest przypisywany jest adres struktury typu `struct module`, który zwracany jest przez makro `THIS_MODULE`.

W wierszu nr 57 deklarowana i inicjowana jest zmienna, która będzie przechowywała numer urządzenia. W wierszu nr 58 deklarowana jest struktura typu `struct cdev`, a wiersze 59 i 60 zawierają deklaracje wskaźników na struktury typu `struct class` i `struct device`. Zazwyczaj wymienione w tym akapicie zmienne definiowane jako pola osobnej struktury, ale w przypadku sterownika prostego urządzenia znakowego nie ma takiej potrzeby.

Wiersze 62-96 zawierają definicję konstruktora modułu. W wierszu nr 64 przydzielany jest numer urządzenia dla sterownika. Ze względu na to, że będzie on obsługiwał tylko jedno pseudo urządzenie, to wystarczy tylko jeden taki numer. Jeśli ten przydział się nie powiedzie, to w buforze jądra zostanie umieszczony odpowiedni komunikat (wiersz nr 65) i funkcja zakończy działanie zwracając wartość `-1`. W wierszu nr 69 tworzona jest struktura typu `struct class` oraz odpowiednie katalogi i pliki w systemie plików `sysfs`. W wierszu nr 70 sprawdzane jest, czy ta ostatnia operacja się powiodła. Jeśli nie, to kod jej błędu będzie zawarty w zwróconym adresie. W takim wypadku konstruktor zapisze ten kod w buforze jądra, a następnie (wiersz nr 72) zwolni przydzielony numer urządzenia i (wierz nr 73) zakończy swoje działanie. W wierszu 76 inicjowana jest struktura typu `struct cdev`. Jej pole `owner` musi zostać zainicjowane osobno (wiersz nr 77) adresem zwróconym przez makro `THIS_MODULE`. W wierszu nr 79 ta struktura jest dodawana do systemu. Jeśli ta operacja nie uda się, to oprócz umieszczenia odpowiedniego komunikatu w buforze jądra konstruktor zwalnia wcześniej utworzoną strukturę i numer urządzenia, a następnie kończy działanie. W wierszu nr 86 jest tworzona struktura `STRUCT DEVICE` i wysyłany jest komunikat do demona `udevd` z przestrzeni użytkownika. Jeśli ta operacja się nie powiedzie, to cofane są skutki wszystkich poprzednich operacji, zanim konstruktor skończy pracę sygnalizując błąd. Jeśli jednak ta operacja się powiedzie, to konstruktor kończy działanie zwracając zero.

Destruktor modułu jest zdefiniowany w wierszach 98-107. W tej funkcji zwalniana jest pamięć na struktury typu `struct device` i `struct class` (wiersze nr 101 i nr 104) oraz usuwana jest z systemu struktura typu `cdev` (wiersz nr 102) i zwalniany jest numer urządzenia (wiersz nr 106), pod warunkiem, że operacje związane z tymi elementami zakończyły się w konstruktorze sukcesem. Proszę zwrócić uwagę, że te operacje są wykonywane w odwrotnej kolejności niż były przeprowadzane ich odpowiedniczki z konstruktora. Jest to prawidłowa kolejność finalizacji.

Zdania

1. `[2 punkty]` Zmień kod konstruktora z listingu 6 tak, aby wykorzystywał on do obsługi wyjątków instrukcję `goto`, ale zachowywał się tak samo.

2. `[4 punktów]` Zmień kodu modułu z listingu 6 tak, aby wyrazy ciągu Fibonacciego były zapisywane do tablicy oraz zaimplementuj metodę `llseek()`, aby można było wskazywać wyraz, który powinien być odczytany. Napisz program dla przestrzeni użytkownika, który będzie wykorzystywał tę własność sterownika.

3. [6 punktów] Napisz sterownik, który będzie obsługiwał dwa pseudo urządzenia znakowe. Pierwsze będzie zwracało kolejną liczbę naturalną w stosunku do tej, jaka zostanie do niego zapisana, a drugie poprzednią. Liczby te powinny mieścić się w zakresie typu `u64`. Pamiętaj o synchronizacji zapisu i odczytu.

4. [2 punkty] Zmień tak kod modułu z listingu 6, aby funkcja `fib_open()` zwracała jedynie `0`, ale żeby działanie modułu zostało zachowane.

5. [4 punktów] Napisz moduł jądra, który będzie tworzył pseudo urządzenie znakowe o nazwie `clipboard` pozwalające zapisać (np. przy pomocy polecenia `echo`) i odczytać łańcuch nie dłuższy niż `1024` znaki. Pamiętaj o synchronizacji odczytu i zapisu.

6. [6 punktów] Napisz sterownik pseudo urządzenia znakowego, które będzie zwracało tekst dostarczony mu za pomocą pliku w systemie plików `sysfs`. Pamiętaj o synchronizacji odczytu z zapisem z poziomu systemu `sysfs`.