# Systemy Operacyjne 2

## Urządzenia znakowe i blokowe

Arkadiusz Chrobot

Katedra Systemów Informatycznych

24 maja 2024

# Plan

# Wstęp

Jednym z zadań Wirtualnego Systemu Plików jest obsługa urządzeń wejścia-wyjścia. Słowo „urządzenie” w tym kontekście nie musi koniecznie oznaczać fizyczny sprzęt. Może to być także wirtualne urządzenie nazywane także pseudourządzeniem. Większość systemów kompatybilnych z Uniksem wyróżnia trzy kategorie urządzeń — znakowe, blokowe i sieciowe — które dostępne są dla oprogramowania użytkowego. Linux stosuje również pewne podkategorie, ale są one dostępne tylko wewnątrz jądra systemu operacyjnego. Wykład przedstawia jak urządzenia należące do dwóch pierwszych z wymienionych kategorii są obsługiwane przez jądro Linuksa. Jest w nim także zawarte krótkie wprowadzenie do Modelu Urządzeń Jądra Linuksa (ang. *Linux Kernel Device Model*), nazywanego krótko Modelem Urządzeń Linuksa (ang. *Linux Device Model*).

# Wstęp

## Wprowadzenie do Modelu Urządzeń Jądra Linuksa

Współczesne urządzenia peryferyjne mogą być podłączane do systemu komputerowego i odłączane w trakcie jego działania. Mają one także zmienne wymagania odnośnie ilości dostarczanej do nich energii elektrycznej oraz korzystają z magistral (szyn), o złożonej budowie. Aby zaspokoić te potrzeby programiści wprowadzili do jądra podsystem nazwany Modelem Urządzeń Linuksa (ang. *Linux Device Model*), w skrócie LDM. Dzięki niemu umożliwiono [1]:

- zarządzanie urządzeniami wejścia-wyjścia z poziomu przestrzeni użytkownika,
- sterowaniem kolejnością wyłączania urządzeń,
- reprezentowanie wewnątrz jądra urządzeń i struktury ich połączeń,
- zarządzanie czasem życia struktur danych związanych z urządzeniami,
- wielokrotne wykorzystanie kodu w podsystemie urządzeń peryferyjnych.

# Wstęp

## Wprowadzenie do Modelu Urządzeń Jądra Linuksa

LDM był pierwszym podsystemem, jądra, w którego kodzie zastosowano podejście obiektowe. ▸ Główną strukturą danych w tym podsystemie są ▸ obiekty jądra typu `struct kobject`. Zwykle są one składowymi większych struktur, ale również powiązane są ze sobą hierarchicznie. Każdy obiekt jądra jest reprezentowany w systemie plików `sysfs`, umożliwiając tym samym komunikację jądra z przestrzenią użytkownika i także, np. zmianę ustawień urządzenia peryferyjnego. Zawiera on również adresy struktur typu `struct kset` i `struct kobj_type`. Struktury pierwszego typu służą do grupowania obiektów jądra mających to samo zastosowanie, czyli stanowią dla nich kontenery. Umożliwiają one zarządzanie obiektami jądra obsługującymi urządzenia typu *hotplug* (dołączane/odłączane dynamicznie). Z kolei struktury typu `struct kobj_type` określają *klasę* obiektów jądra, która definiuje w jaki sposób obiekt zostanie zwolniony, kiedy jego wewnętrzny licznik odwołań osiągnie zero i jak będzie reprezentowany w systemie plików `sysfs`.

# Wstęp

## Wprowadzenie do Modelu Urządzeń Jądra Linuksa

Na bazie obiektów jądra budowane są struktury ▸ wyższego poziomu, które także są obiektami i reprezentują magistrale (ang. *buses*), sterowniki (ang. *drivers*) i urządzenia oraz klasy urządzeń. Struktury typu `struct bus_type` reprezentują różnego rodzaju magistrale, zarówno fizyczne, jak i wirtualne, do których podłączone są urządzenia. Są one obiektami, dla których programiści najczęściej definiują dwie metody[1]: `match()` i `uevent()`. Pierwsza wywoływana jest po dodaniu nowego urządzenia lub sterownika do systemu. Jej zadaniem jest porównanie identyfikatora urządzenia z identyfikatorem sterownika. Dzięki tym metodom jądro może powiązać ze sobą sterowniki z urządzeniami. Metoda `uevent()` jest odpowiedzialna za dodanie zmiennych środowiskowych, z których korzysta program `udev`, będący współcześnie częścią `systemd`, a którego zadaniem jest stworzenie pliku urządzenia, najczęściej w katalogu `/dev`.

[1] Jest ich więcej, ale te są najistotniejsze.

# Wstęp

## Wprowadzenie do Modelu Urządzeń Jądra Linuksa

Struktury typu `struct device` reprezentują urządzenia, które mogą być zarówno fizyczne jak i wirtualne (tzw. pseudourządzenia). Sterowniki, czyli oprogramowanie odpowiedzialne za obsługę urządzeń, jest z kolei reprezentowane przez strukturę (obiekt) typu `struct device_driver`. W końcu struktury (objekty) typ `struct class` grupują urządzenia posiadające te same lub podobne funkcje (np. urządzenia wejściowe).

# Wstęp

## Obsługa urządzeń I/O w Linuksie — ogólny zarys

W systemach kompatybilnych z Uniksem urządzenia znakowe i blokowe są obsługiwane tak jak pliki, tj. z użyciem tych samych wywołań systemowych. Są one również reprezentowane przez pliki, które oprócz nazwy, mają trzy dodatkowe atrybuty: *typ*, który określa, czy reprezentowane urządzenie jest znakowe czy blokowe, numer główny (ang. *major*) i numer poboczny (ang. *minor*). W jądrze te numery są łączone w 32-bitowy *numer urządzenia* typu `dev_t`. Począwszy od serii 2.6 jądra numer główny zajmuje 12 najbardziej znaczących bitów numeru urządzenia, a numer poboczny 20 najmniej znaczących. Dostęp do tych numerów wewnątrz numeru urządzenia powinien się zawsze odbywać za pomocą makr `MAJOR` i `MINOR`, a ich łączenie do takiego numeru powinno być realizowane makrem `MKDEV`. Powodem takiego wymagania jest to, że we wcześniejszych wersjach jądra rozmiary głównego i pobocznego numeru urządzenia były takie same: 16 bitów. Uległy one zmianie w serii 2.6 jądra i możliwe, że będą zmieniane także w przyszłych wersjach.

# Wstęp

## Obsługa urządzeń I/O w Linuksie — ogólny zarys

Numer główny identyfikuje sterownik odpowiedzialny w jądrze za obsługę rodziny urządzeń (np. drukarek). Numer poboczny określa konkretne urządzenie obsługiwane przez ten sterownik. Jest to przydatne rozwiązanie, kiedy więcej niż jedno urządzenie należące do danej rodziny jest podłączone do komputera. Sterowniki mogą być umieszczone na stałe w jądrze lub dołączane w postaci modułów.

# Sterowniki urządzeń znakowych

Urządzenia znakowe zazwyczaj zapewniają sekwencyjny dostęp do danych i przesyłają je względnie małymi porcjami np. o rozmiarze kilku bajtów. Rozmiar ten może być inny dla każdej transmisji. Przykładem takich urządzeń są mysz i klawiatura. Pierwszą czynnością wykonywaną przez sterownik urządzenia znakowego jest uzyskanie jednego lub kilku numerów urządzeń przy pomocy funkcji:

```
int register_chrdev_region(dev_t first, unsigned int
                           count, char *name);
```

Parametr `first` określa pierwszy numer urządzenia z puli, którą należy pozyskać. Jeśli sterownik ma być udostępniony szerszej rzeszy użytkowników Linuksa, to te numery muszą być przyznane przez organizację *The Linux Assigned Name and Numbers Authority* (www.lanana.org). W przeciwnym przypadku dostępność tych numerów może być zweryfikowana w pliku `/proc/devices` lub katalogu `/sys`. Parametr `count` określa liczbę numerów urządzeń do uzyskania, a przez `name` przekazywany jest łańcuch znaków będący nazwą urządzenia.

# Sterowniki urządzeń znakowych

Funkcja zwraca 0 jeśli uda jej się uzyskać wymagane numery urządzeń. Wygodniejszą od niej w użyciu jest funkcja:

```
int alloc_chrdev_region(dev_t *dev, unsigned int
    firstminor, unsigned int count, char *name);
```

Alokuje dla sterownika określona liczbę numerów urządzeń począwszy od pierwszego dostępnego. Programistka lub programista nie musi go określać. Parametr `dev` jest parametrem wyjściowym. Funkcja używa go do zwracania pierwszego przydzielonego numeru urządzenia. Parametr `firstminor` określa wartość pierwszego numeru pobocznego, który powinien być pozyskany. Zazwyczaj jest to 0. Pozostałe dwa parametry są takie same jak w przypadku funkcji `register_chardev_region()`. Jeśli uzyskanie numerów się powiedzie, funkcja zwróci 0. Jeśli numery urządzenie nie będą już używane, to powinny zostać wyrejestrowane z użyciem funkcji:

```
void unregister_chrdev_region(dev_t first, unsigned int
    count);
```

# Sterowniki urządzeń znakowych

Sterowniki urządzeń znakowych używają trzech struktur danych VFS: obiektu pliku, tablicy metod pliku i obiektu i-węzła. Tablica metod pliku powinna zawierać adresy funkcji, które wykonują operacje na pliku urządzenia. Jeśli sterownik jest zaimplementowany w postaci modułu jądra, to do pola `owner` tej tablicy metod powinna być przypisana wartość makra `THIS_MODULE`. To zapobiega usunięciu modułu, gdy co najmniej jedna z tych metod jest w użyciu. Zazwyczaj programiści piszący sterowniki urządzeń implementują cztery metody: `open()`, `read()`, `write()` i `release()`, mimo, że zdefiniowanie ich wszystkich nie jest konieczne. Jeśli urządzenie wymaga specyficznych operacji, które nie mogą być zrealizowane przez te metody, to wówczas należy zaimplementować jedną z metod `ioctl()`. Inne metody nie muszą być definiowane. Sterownik może korzystać z następujących pól obiektu pliku: `f_mode` — prawa dostępu, `f_pos` — wskaźnik pliku, `f_flags` — flagi, `f_op` — wskazuje na tablicę metod, `private_data` — dane prywatne sterownika.

# Sterowniki urządzeń znakowych

Zawartość pola `f_mode` może być sprawdzana przez metodę `open()`, ale nie jest to konieczne, ponieważ takiej weryfikacji dokonują inne części jądra przez wywołaniem tej metody. Sterownik może sprawdzić pole flag by zdecydować, czy operacje na pliku mają być synchroniczne, czy asynchroniczne. Wartość pola `f_pos` (64-bitowa) może być używana przez metodę `llseek()`, która zwraca zmodyfikowaną wartość wskaźnika pliku. Również metody `read()` i `write()` używają tego wskaźnika i jest on im przekazywany przez ich ostatni parametr. Pole `private_data` jest wskaźnikiem typu `void *`, które może wskazywać na dynamicznie przydzielony obszar pamięci używany przez sterownik do przechowywania danych, które nie powinny być utracone między wywołaniami metod. Ten obszar powinien być przydzielony przez funkcję `open()` przy jej pierwszym wywołaniu i zwolniony przez metodę `release()` kiedy jest ona uruchamiana w wyniku ostatniego wywołania funkcji `close()` z przestrzeni użytkownika.

# Sterowniki urządzeń znakowych

Z obiektu i-węzła sterownik może użyć pola `i_rdev` przechowującego numer urządzenia. By uzyskać numer główny i poboczny z tego pola należy użyć makr:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

Innym polem tego obiektu jest `i_cdev` wskazujące na strukturę reprezentującą w jądrze urządzenie znakowe obsługiwane przez sterownik. Ta struktura może być tworzona dynamicznie i inicjowana z użyciem funkcji `cdev_alloc()`. Statycznie utworzona taka struktura może być zainicjowana funkcją:

```
void cdev_init(struct cdev *cdev, struct
               file_operations *fops);
```

W obu przypadkach wartość makra `THIS_MODULE` musi być umieszczona w jej polu `owner`. Jeśli ta struktura jest tworzona z użyciem funkcji `cdev_alloc()`, to jej pole `ops` musi być zainicjowane bezpośrednio.

# Sterowniki urządzeń znakowych

Po utworzeniu struktura `cdev` musi zostać dodana do innych takich struktur przechowywanych przez jądro, przy użyciu funkcji:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int
        count);
```

Funkcją usuwająca strukturę `cdev` z repozytorium takich struktur w jądrze jest:

```
void cdev_del(struct cdev *dev);
```

Każde urządzenie znakowe obsługiwane przez sterownik musi mieć swoją strukturę `cdev`. W wcześniejszych wersjach jądra sterownik nie musiał jej tworzyć. Urządzenie było przez niego rejestrowane przy pomocy funkcji `register_chrdev()`, a wyrejestrowywane z użyciem funkcji `unregister_chrdev()`. Począwszy od pewnej wersji w serii 2.6 jądra, podsystem modelu urządzeń musi być poinformowany o nowym sterowniku. Ta czynność powinna nastąpić podczas inicjowania sterownika i wymaga użycia określonego makra i funkcji. Makro tworzy strukturę opisującą klasę urządzenia obsługiwanego przez sterownik.

# Sterowniki urządzeń znakowych

Nagłówek tego makra jest następujący:

```
class_create(owner, name)
```

Jego pierwszym argumentem jest wartość makra `THIS_MODULE`, a drugim jest nazwa klasy. Prototyp funkcji jest następujący:

```
struct device *device_create(struct class *class,
struct device *parent, dev_t devt, void *drvdata, const
char fmt, …);
```

Tworzy ona i rejestruje w systemie plików `sysfs` strukturę reprezentującą urządzenie. Jej pierwszym argumentem jest adres struktury klasy. Drugim jest adres nadrzędnej struktury urządzenia — może to być `NULL` jeśli taka struktura nie istnieje. Trzecim argumentem jest numer urządzenia, a czwartym adres danych przechowywanych w strukturze i wykorzystywanych przez funkcje wywołań zwrotnych (ang. *callback functions*) — jego wartość również może być `NULL`. Piątym argumentem jest ciąg znaków będący nazwą urządzenia. Może on zawierać sekwencje konwertujące.

# Sterowniki urządzeń znakowych

Struktura tworzona przez `device_create()` może być usunięta przy pomocy funkcji:

```
void device_destroy(struct class *cls, dev_t devt);
```

Pobiera ona dwa argumenty: adres struktury klasy i numer urządzenia. Struktura klasy jest usuwana przez funkcję:

```
void class_destroy(struct class *cls);
```

Jako argument pobiera ona adres struktury klasy. Działanie metod sterownika urządzenia musi podlegać określonemu protokołowi. Metoda `open()` powinna:

- zidentyfikować urządzenie obsługiwane przez sterownik — uzyskać numer poboczny tego urządzenia,
- sprawdzić, czy nie wystąpiły wyjątki związane z urządzeniem,
- zainicjować urządzenie przy pierwszym otwarciu jego pliku,
- zaktualizować wskaźnik pliku, jeśli jest to konieczne,
- przydzielić i zainicjować pamięć na dane prywatne sterownika, jeśli jest to konieczne.

# Sterowniki urządzeń znakowych

Również metoda `release()` powinna działać wedle następującego protokołu:

- zwolnić pamięć na dane prywatne sterownika, jeśli była ona przydzielona przez metodę `open()`,
- wyłączyć urządzenie po ostatnim wywołaniu funkcji `close()` z przestrzeni użytkownika.

Także implementacje metod `read()` i `write()` powinny przestrzegać kilku reguł. Te funkcje powinny zwracać liczbę faktycznie przeczytanych lub zapisanych bajtów. W przypadku niepowodzenia powinny zwrócić kod błędu pozwalający zidentyfikować przyczynę, taki jak: `-EINTR` — otrzymano sygnał, `-EFAULT` — zły adres lub `-EIO` — ogólny błąd wejścia-wejścia.
Bardziej szczegółowy opis API sterowników urządzeń znakowych znajduje się w ósmej instrukcji laboratoryjnej.

# Sterowniki urządzeń blokowych

Sterowniki urządzeń blokowych używają podobnych struktur i operacji jak sterowniki urządzeń znakowych. Obsługa urządzeń blokowych jest jednak bardziej skomplikowana i niektórej jej szczegóły zostaną przedstawione w następnym wykładzie. Urządzenia blokowe zapewniają swobodny dostęp do danych i przesyłają je porcjami nazywanymi *blokami*, stąd ich nazwa. Rozmiar bloku jest parzystą wielokrotnością rozmiaru sektora, a w szczególności może być mu równy. Jądro przyjmuje, że rozmiar sektora wynosi 512 bajtów.
Pierwszą czynnością wykonywaną przez sterownik urządzenia blokowego podczas jego inicjacji, jest pozyskanie numeru głównego przy pomocy funkcji `register_blkdev()` zadeklarowanej w pliku nagłówkowym `linux/fs.h` następująco:

```
int register_blkdev(unsigned int major, const char
                                        *name);
```

Jeśli jej pierwszym argumentem jest 0, to przydzieli i zwróci ona pierwszy dostępny numer główny.

# Sterowniki urządzeń blokowych

Przydzielony numer główny może być zwolniony przy użyciu funkcji:

```
void unregister_blkdev(unsigned int major, const char
*name);
```

Sterowniki urządzeń blokowych mają swoją własną tablicę metod, która jest strukturą typu `struct block_device_operations` zadeklarowaną w pliku nagłówkowym `linux/blkdev.h`. Posiada ona pole `owner` i kilka innych składowych, które powinny wskazywać takie metody jak: `open()`, `release()`, `ioctl()`, `compat_ioctl()`, `check_events()` i `revalidate_disk()`. Metoda `check_events()` jest wywoływana głównie wtedy, gdy zmieniany jest nośnik w urządzeniu, a po niej następuje wywołanie metody `revalidate_disk()`. Tak jak urządzenie znakowe jest reprezentowane przez strukturę `cdev`, tak urządzenie blokowe jest reprezentowane za pomocą struktury typu `struct gendisk`, zdefiniowanego w pliku nagłówkowym `linux/genhd.h`.

# Sterowniki urządzeń blokowych

Ta struktura ma następujące pola: `major` — przechowuje numer główny, `first_minor` — przechowuje pierwszy numer poboczny, `minors` — zawiera liczbę numerów pobocznych, `disk_name` — przechowuje ciąg znaków będący nazwą urządzenia (maksymalnie 32 znaki), `fops` — zawiera adres struktury `block_device_operations`, `queue` — przechowuje adres *kolejki żądań*, `flags` — zawiera flagi (rzadko używane pole, zazwyczaj tylko w przypadku urządzeń typu *hot-plug* i dysków optycznych) oraz `private_data` — wskazuje obszar pamięci z prywatnymi danymi sterownika. Ponadto struktura `gendisk` przechowuje pojemność urządzenia blokowego wyrażoną w sektorach. Ta wartość jest ustawiana za pomocą funkcji `set_capacity()`. Wspomniana struktura jest tworzona przez funkcję `alloc_disk()`, a usuwania, gdy wartość jej licznika odwołań spadnie od 0, przez funkcję `put_disk()`:

```
struct gendisk *alloc_disk(int minors);
void put_disk(struct gendisk *disk);
```

# Sterowniki urządzeń blokowych

Każda struktura `gendisk` reprezentuje pojedyncze urządzenia blokowe obsługiwane przez sterownik, np. jedną partycję dysku twardego. Aby to urządzenie stało się dostępne dla reszt jądra, sterownik powinien wywołać funkcję `add_disk()` dla jego struktury `gendisk`:

```
void add_disk(struct gendisk *gd);
```

Usunięcia tej struktury dokonuje funkcja `del_gendisk()`:

```
void del_gendisk(struct gendisk *gd);
```

Najważniejszym polem struktury `gendisk` jest pole `queue`, wskazujące na kolejkę żądań (ang. *request queue*). Pamięć na tę kolejkę jest przydzielana przez funkcję `blk_init_queue()`:

```
request_queue_t *blk_init_queue(request_fn_proc
    *request, spinlock_t *lock);
```

Jej pierwszym argumentem jest adres funkcji przetwarzającej pojedyncze żądanie z tej kolejki, zaś drugim jest adres rygla pętlowego chroniącego tę kolejkę.

# Sterowniki urządzeń blokowych

Jeśli sterownik obsługuje urządzenie, które w przeciwieństwie do dysków twardych oferuje prawdziwie bezpośredni dostęp do danych (np. pendrive), to kolejka żądań jest zbędna. W tym przypadku pole `queue` struktury `gendisk` jest inicjowane z użyciem funkcji `blk_alloc_queue()`:

```
request_queue_t *blk_alloc_queue(int flags);
```

Sterownik wspomnianego urządzenia powinien dostarczyć implementacji funkcji `make_request()`, która powinna przetwarzać pojedyncze żądania. Jest ona rejestrowana przy pomocy następującej funkcji:

```
void blk_queue_make_request(request_queue_t *queue,
                            make_request_fn *func);
```

Usunięcia kolejki żądań dokonuje następująca funkcja:

```
void blk_cleanup_queue(struct request_queue *q);
```

Bardziej szczegółowy opis API sterowników urządzeń blokowych znajduje się w dziewiątej instrukcji laboratoryjnej.

# Literatura

 John Madieu. *Linux Device Drivers Development*. Birmingham, UK: Packt Publishing, 2017.

# Pytania

?

# Koniec

Dziękuję Państwu za uwagę!