Jądro systemu Linux
– podsystem wejścia - wyjścia

Marek Fajkowski Łukasz Krzeszczakowski

Michał Przybylski Marcin Raczyński Jacek Świergocki

– studenci III roku INF UW

7 stycznia 1999

Spis treści

Rozdział 1

Wstęp

1.1 Pliki specjalne

1.1.1 Reprezentacja urządzenia

W systemie Linux urządzenia są reprezentowane przez pliki specjalne. Pliki specjalne są to pliki, które nie przechowują żadnych danych. Strategia systemu operacyjnego Linux w stosunku do urządzeń zewnętrznych polega na reprezentowaniu każdego urządzenia jako oddzielnego pliku lub plików. Pozwala to na zunifikowanie sposobu odwołania się do wszystkich urządzeń oraz określanie praw dostępu podobnie jak dla zwykłych plików. Pliki specjalne są reprezentowane przez i-węzły. I-węzły plików specjalnych różnią się od i-węzłów zwykłych plików zawartością i sposobem tworzenia. Są tworzone za pomocą operacji **mknod**. Poniżej opiszę strukturę i-węzła, która interesuje nas z punktu widzenia plików specjalnych.

Plik źródłowy **include/linux/fs.h**:

```
struct inode {
   kdev_t i_rdev;        numer urządzenia ( główny i drugorzędny )
   umode_t i_mode;          typ urządzenia ( blokowe lub znakowe )
   struct inode_operations *i_op;      dostępne operacje na pliku
   ...
};
```

Urządzenie może mieć wiele plików specjalnych. Bez ich istnienia nie da się komunikować z żadnym urządzeniem. Pliki specjalne dla urządzeń blokowych i znakowych są rozróżniane w systemie. Do niektórych urządzeń możliwy jest dostęp zarówno blokowy jak i znakowy.

1.1.2 Algorytm tworzenia pliku specjalnego - mknod

Wywoływana jest funkcja **sys_mknod**, która działa według następującego schematu (plik źródłowy **fs/namei.c**):

```
int sys_mknod(const char *filename, int mode, dev_t dev)
```

mode — tryb pliku, czyli odpowiednie flagi
dev — numer urządzenia (główny i drugorzędny)

1. Sprawdzamy, czy tryb jest trybem katalogu lub czy tryb nie jest trybem kolejki FIFO i użytkownik nie jest super–użytkownikiem. Jeżeli tak to zwracamy błąd.
2. Sprawdzamy, czy został podany prawidłowy tryb, jeżeli nie został podany żaden to przyjmujemy tryb zwykłego pliku. W p.p. zwracamy błąd.
3. Zapisujemy nazwę pliku do przestrzeni adresowej jądra.
4. Wywołujemy funkcje **do_mknod(nazwa, tryb, dev)**.
5. Zwalniamy pamięć jądra zajętą na nazwę pliku.

```
int do_mknod(const char *filename, int mode, dev_t dev )
```

1. Pobieramy i-węzeł katalogu macierzystego.
2. Jeżeli pobrany katalog ma tylko prawo do odczytu, lub nie ma prawa do pisania i czytania to zwalniamy jego i-węzeł i zwracamy błąd.
3. Jeżeli pobrany katalog nie ma dostępnej operacji **mknod**, to zwalniamy jego i-węzeł i zwracamy błąd.
4. Zwiększamy liczbę dowiązań plików w katalogu.
5. Blokujemy dojście do katalogu opuszczając semafor.
6. Wywołujemy operacje **mknod(dir, name, length, mode, rdev)**, która należy do dostępnych operacji katalogu macierzystego. Jest ona specyficzna dla systemu plików, w którym znajduje się katalog.
7. Podnosimy opuszczony semafor.
8. Zwalniamy i-węzeł katalogu i wynik operacji **mknod**.

Przykładowa funkcja **mknod** dla systemu plików **ext2** (plik źródłowy **fs/ext2/-/namei.c**):

```
int ext2_mknod(struct inode *dir, const char *name, int length,
                 int mode, int rdev)
```

dir – i-węzeł katalogu macierzystego

1. Sprawdzamy, czy podany plik istnieje. Jeżeli tak, to kończymy zwalniając i-węzeł macierzystego katalogu i zwracając błąd.

2. Tworzymy nowy i-węzeł. Jeśli się nie powiodło, to analogicznie kończymy zwalniając i-węzeł i zwracając błąd.

3. Ustawiamy w nowym i-węźle właściciela pliku, tryb oraz dostępne operacje. W przypadku urządzeń blokowych i znakowych są to referencje do **blk_inode_operations** lub **chrdev_inode_operations**.

4. W przypadku pliku dla urządzenia blokowego lub znakowego (flagi **S_ISCHR** lub **S_ISBLK**) inicjalizowane jest też pole **i_rdev** (numer główny i drugorzędny) przez zastosowanie funkcji **to_kdev_t(dev)**.

5. Dodajemy nowy i-węzeł do macierzystego katalogu. Jeśli się nie powiodło, to kończymy zwalniając i-węzeł macierzystego katalogu i zwracając błąd. Jeśli się powiodło to otrzymujemy bufor.

6. Dokonujemy inicjalizacji odpowiednich struktur.

7. Jeżeli plik jest synchroniczny to dokonujemy zapisu z bufora do urządzenia.

8. Zwalniamy bufor, nowy i-węzeł oraz i-węzeł katalogu macierzystego.

1.2 Urządzenia blokowe i znakowe

Komunikacja z urządzeniami wejścia-wyjścia polega na przesyłaniu danych do urządzenia oraz odbieraniu ich. Dostęp do poszczególnych urządzeń następuje poprzez pliki specjalne – blokowe lub znakowe.

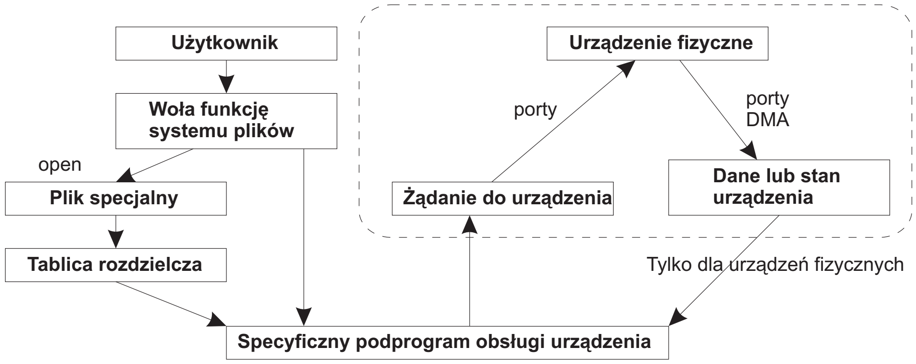
Czasami przesyłane są pojedyncze dane, a czasami ich duże ilości. W przypadku urządzeń, z którymi wymieniane są duże ilości danych zastosowany został mechanizm buforowania. Dostęp do takich urządzeń jest dostępem bezpośrednim. Urządzenia takie nazwano urządzeniami blokowymi. Urządzenia blokowe podzielone są na dwie klasy:
- klasa urządzeń SCSI
- klasa urządzeń IDE

W przypadku pozostałych urządzeń wymiana danych następuje poprzez przekazywanie pojedynczych danych. Te urządzenia nazwano wspólnie urządzeniami znakowymi.

Urządzenia blokowe można traktować również jako znakowe, lecz nie wszystkie udostępniają takie możliwości (z powodu braku implementacji). Aby z urządzenia można było korzystać jako blokowego i znakowego potrzebne są dwa pliki specjalne – jeden blokowy i jeden znakowy. Obecność dwóch plików specjalnych w systemie może prowadzić do konfliktów. Jest to możliwe, kiedy równocześnie korzystamy z urządzenia przy wykorzystaniu obydwu trybów dostępu. Nie grozi to zniszczeniem sprzętu, lecz utratą danych.

1.3 Ogólny schemat dostępu do urządzenia

Ten podrozdział ma na celu zobrazowanie całego procesu odwołania się do urządzenia. W dalszych rozdziałach przedstawione zostaną szczegółowe opisy poszczególnych etapów. Teraz pokażemy je jedynie poglądowo, aby zobaczyć ich wzajemne związki.



Rysunek 1.1: Ogólny schemat dostępu do urządzenia

Odwołanie do urządzenia zaczyna się od wywołania przez użytkownika funkcji systemu plików. Pierwszą funkcją, którą użytkownik musi wywołać jest funkcja **open**. Przed wywołaniem **open** nie można wywołać żadnej innej. Funkcja **open** pobiera jako parametr nazwę pliku specjalnego urządzenia. Następnie z tablicy rozdzielczej pobiera strukturę **file_operations**, właściwą dla urządzenia i umieszcza w strukturze **file** otwieranego pliku (szczegóły w rozdziale 2). Od tej pory możliwe już jest wywołanie wszystkich dostępnych funkcji. Tutaj kończy się różnica pomiędzy **open**, a innymi funkcjami. Dalej wywoływana jest właściwa funkcja podprogramu obsługi urządzenia. Jeżeli jest to urządzenie logiczne, to schemat odwołania kończy się na podprogramie obsługi. W przypadku urządzenia fizycznego podprogram wysyła przez porty pewne zlecenie (szczegóły w rozdziale 3). Urządzenie je realizuje. Po wykonaniu zlecenia urządzenie wysyła odpowiedź przez porty bądź kanałami DMA. Odpowiedzią mogą być dane lub informacje o stanie urządzenia. Następnie sterowanie powraca do podprogramu obsługi i dalej do użytkownika.

1.4 Co w następnych rozdziałach

W następnych rozdziałach omówiliśmy szczegółowo podsystem systemu operacyjnego Linux dotyczący wejścia - wyjścia. Rozdział drugi został poświęcony przedstawieniu interfejsu jądra. Opisane zostały tam operacje na plikach specjalnych, tablice rozdzielcze i algorytmy wybranych funkcji. W rozdziale trzecim przedstawiliśmy część systemu dotyczącą komunikacji z urządzeniami fizycznymi, czyli obsługę przerwań, porty wejścia-wyjścia, kanały DMA. Rozdziały czwarty i piąty poświęcone są urządzeniom blokowym i znakowym odpowiednio. W pierwszym z nich oprócz pokazania algorytmów odczytu i zapisu bloków omówione zostało szeregowanie żądań i procedury strategii. Natomiast ostatni rozdział przedstawia urządzenia znakowe na przykładzie terminali.

Rozdział 2

Interfejs jądra dla operacji odwołujących się do urządzeń

2.1 Wstęp

W poniższym rozdziale zostanie omówiony interfejs jądra, czyli sposób, w jaki funkcje systemowe korzystają z podprogramów obsługi urządzeń.

W systemie Linux dla każdego urządzenia, zarówno blokowego jak i znakowego, pamiętana jest informacja o podprogramie obsługującym to urządzenie. Informacja ta składa się z 13 wskaźników do odpowiednich funkcji podprogramu. Struktura **file_operations** (plik źródłowy **include/kernel/fs.h**):

```
struct file_operations {
   int (*lseek) (struct inode *, struct file *, off_t, int);
   int (*read) (struct inode *, struct file *, char *, int);
   int (*write) (struct inode *, struct file *, const char *, int);
   int (*readdir) (struct inode *, struct file *, void *, filldir_t);
   int (*select) (struct inode *, struct file *, int, select_table *);
   int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
                                                               long);
   int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   void (*release) (struct inode *, struct file *);
   int (*fsync) (struct inode *, struct file *);
   int (*fasync) (struct inode *, struct file *, int);
   int (*check_media_change) (kdev_t dev);
   int (*revalidate) (kdev_t dev);
};
```

Wskaźniki z tej struktury albo wskazują na odpowiednią funkcję, albo są równe **NULL**, co oznacza, że dla danego urządzenia funkcja nie została zaimplementowana (czyli urządzenie nie obsługuje takiej operacji). Poszczególne funkcje realizują między innymi:

- otwarcie i zainicjowanie urządzenia (funkcja **open**)
- czytanie z urządzenia (**read**)
- zapis do urządzenia (**write**)
- wybór pozycji odczytu/zapisu (**lseek**)
- zamknięcie urządzenia (**release**)

Do ciekawszych funkcji należy także **ioctl**, która pozwala wykonywać operacje specyficzne dla urządzenia, których nie obejmują funkcje systemowe. Jest to realizowane w ten sposób, że jako parametr funkcji systemowej **ioctl** podaje się numer operacji i jej argument (na przykład jakiś wskaźnik), zaś podprogram obsługi na podstawie podanego numeru sam rozpoznaje jaka operacja ma zostać wykonana.

Powstaje pytanie, gdzie jest przechowywana informacja o urządzeniu (a więc struktura **file_operations**). W trakcie działania systemu jest ona pamiętana w specjalnych tablicach systemowych, zwanych tablicami rozdzielczymi urządzeń (zostaną one omówione szerzej w punkcie 2.4). W czasie uruchamiania systemu podprogram obsługi urządzenia sam musi zadbać o wpisanie odpowiedniej informacji do tablicy rozdzielczej (proces ten nazywa się rejestracją urządzenia i zostanie omówiony w punkcie 2.5).

W i-węźle pliku specjalnego odnajdujemy pole **i_op**:

```
struct inode {
   kdev_t i_rdev;
   struct i_node_operations * i_op;
   ...
};

struct inode_operations {
   struct file_operations * default_file_ops;
   ...
};
```

W strukturze opisywanej przez to pole znajdują się operacje, które można wykonać na i-węźle (np. **create**, **unlink**) oraz (co dla nas jest bardziej interesujące) pole **default_file_ops** zawierające wskaźnik do struktury **file_operations**. Okazuje się jednak, ze pole to wskazuje nie na właściwy zbiór operacji, ale na operacje domyślne dla danego typu urządzeń – blokowego lub znakowego (patrz algorytm **mknod**, punkt 1.1.2). W domyślnym zbiorze operacji (odpowiednio **def_blk_fops** i **def_chr_fops)** wszystkie wskaźniki są ustawione na **NULL**, poza **open**, które wskazuje na funkcję wspólną dla danego typu urządzeń (czyli **blkdev_open** lub **chrdev_open**).

Strukturę **file_operations** odnajdujemy również w strukturze **file**, odpowiadającej otwartemu plikowi. Tu jednak zestaw operacji jest pełny – stąd informację o podprogramach obsługi będą pobierać wszystkie funkcje systemowe (poza **open**).

Dlaczego domyślny zestaw operacji przechowywany w i-węźle jest wystarczający i skąd biorą się pozostałe wskaźniki w otwartym pliku – zostanie to wyjaśnione w kolejnych punktach.

2.2 Schemat odwołania się do urządzenia

W poniższym punkcie zostanie krótko omówiony schemat odwołania się do urządzenia, czyli jakie kolejne kroki muszą zostać wykonane, aby można było przeprowadzić transmisję danych z lub do urządzenia. Schemat ten nie obejmuje ani tworzenia pliku specjalnego, ani rejestracji urządzenia. Zadaniem schematu jest jedynie przybliżenie metody działania jądra przy dostępie do urządzenia. Dokładne algorytmy zostaną omówione w dalszej części tego rozdziału.

- Program użytkownika wywołuje funkcję systemową **open** podając nazwę pliku urządzenia.
- Funkcja systemowa sprawdza, czy w i-węźle pliku istnieje zestaw operacji dozwolonych operacji dla danego urządzenia i czy w tym zestawie znajduje się wskaźnik do funkcji **open** – jeśli tak, to ją wywołuje.
- Wywołana funkcja (czyli **blkdev_open** lub **chrdev_open**) wstawia do struktury pliku właściwy zestaw operacji pobrany z odpowiedniej tablicy rozdzielczej oraz wywołuje funkcję **open** specyficzną dla urządzenia.
- Program użytkownika wywołuje funkcję systemową (np. **read**, **write**, **lseek**) podając deskryptor pliku.
- Funkcja systemowa sprawdza odpowiednią pozycję w zestawie operacji dostępnych dla pliku o podanym deskryptorze; jeżeli właściwa operacja znajduje się w tym zestawie (wskaźnik różny od **NULL**), to jest ona wywoływana.

2.3 Numer główny i drugorzędny urządzenia

Każdemu urządzeniu zostaje przypisany numer główny (**MAJOR**) oraz numer drugorzędny (**MINOR**).

Numer główny określa grupę urządzeń, której odpowiada konkretna pozycja w tablicy rozdzielczej, zatem określa jednoznacznie podprogram obsługi urządzenia.

Numer drugorzędny określa numer jednostki i zapewnia unikatowość w danej grupie. Może być on interpretowany wewnątrz konkretnej funkcji podprogramu obsługi. Numer drugorzędny może decydować o tym, o które z wielu takich samych urządzeń nam chodzi, np. dla urządzenia blokowego o numerze głównym równym 3 numer drugorzędny oznacza:

0	dysk 0
1	partycja 1 dysku 0
...	
63	partycja 63 dysku 0
64	dysk 1
65	partycja 1 dysku 1
...	
127	partycja 63 dysku 1

Może też być tak, że różne urządzenia współdzielą ten sam numer główny, np. dla urządzenia znakowego o numerze głównym 1 numer drugorzędny oznacza m.in.:

1	pamięć fizyczna
2	pamięć wirtualna jądra
3	urządzenie **/dev/null**
4	dostęp do portów wejścia/wyjścia
5	generator zer (**/dev/zero**)
8	generator liczb losowych

Powyższe urządzenia spełniają dość różne funkcje, jednak wszystkie są obsługiwane przez ten sam podprogram obsługi.

Numeracja urządzeń blokowych i znakowych jest niezależna, to znaczy mogą istnieć dwie grupy urządzeń o tym samym numerze głównym, o ile jedną grupę stanowią urządzenia blokowe, a drugą znakowe.
Przykładowe numery główne urządzeń (plik **include/linux/major.h**):

```
#define  MEM_MAJOR      1           //urządzenie znakowe
#define  RAMDISK_MAJOR  1           //urządzenie blokowe
#define  FLOPPY_MAJOR   2
#define  IDE0_MAJOR     3
#define  HD_MAJOR       IDE0_MAJOR
#define  TTY_MAJOR      4
#define  LP_MAJOR       6
...
```

W i-węźle numery urządzeń są pamiętane w polu **i_rdev** (typu **unsigned short**). Osiem najbardziej znaczących bitów pola oznacza numer główny, pozostałe osiem bitów – numer drugorzędny.

Widać zatem, że funkcje z podprogramu obsługi mogą łatwo rozpoznawać poszczególne urządzenia – wystarczy bowiem, że z i-węzła pliku specjalnego (przekazywanego prawie zawsze jako parametr – patrz punkt 2.1) pobiorą numer drugorzędny urządzenia i zachowają się odpowiednio dla danego numeru. Na przykład wspomniane już urządzenie **mem** (numer główny 1) w funkcji **open** (z podprogramu obsługi), w zależności od numeru drugorzędnego w pole **f_op struktury file** wpisuje inny zestaw operacji (zamazując ten wpisany przez **chrdev_open**).

2.4 Tablice rozdzielcze urządzeń

W systemie istnieją dwie tablice przechowujące informacje o podprogramach obsługi urządzeń. Tablice te nazywamy tablicami rozdzielczymi urządzeń. Jedna z tablic zawiera informacje o podprogramach obsługi urządzeń blokowych, druga – znakowych. Tablice indeksowane są numerem głównym urządzenia.

Tablice rozdzielcze są zdefiniowane następująco (plik **fs/devices.c**):

```
static struct device_struct chrdevs [MAX_CHRDEV];
static struct device_struct blkdevs [MAX_BLKDEV];
```

Stałe **MAX_CHRDEV** oraz **MAX_BLKDEV** są zdefiniowane w pliku **include/linux/major.h**, a ich wartość jest równa 128.

```
struct device_struct {
     const char *name;
     struct file_operations *fops;
};
```

Informacja o urządzeniu, pamiętana w **device_struct**, składa się z nazwy urządzenia oraz z omówionego wcześniej zestawu operacji dozwolonych dla danego urządzenia. Informację zapisaną w tablicach rozdzielczych można uzyskać za pomocą trzech funkcji:

get_device_list (char *) – na podaną zmienną wypisuje po kolei wszystkie urządzenia, najpierw znakowe, potem blokowe, podając numer główny i nazwę urządzenia

get_blkfops (unsigned int) – zwraca strukturę **file_operations** dla urządzenia blokowego o podanym numerze głównym

get_chrfops (unsigned int, unsigned int) – zwraca strukturę **file_operations** dla urządzenia znakowego o podanych numerach: głównym i drugorzędnym

2.5 Rejestracja urządzeń

Ze względu na fakt, że początkowo tablice rozdzielcze nie zawierają żadnych informacji, podprogramy obsługi urządzeń same muszą zadbać o wpisanie odpowiedniej struktury **file_operations** do tablicy rozdzielczej. Zwykle podprogram obsługi dostarcza dodatkowo specjalną funkcję inicjującą, której głównym zadaniem jest właśnie wpisanie informacji do odpowiedniej tablicy rozdzielczej (oprócz tego podprogram może w funkcji inicjującej np. zainicjować swoje struktury danych). Funkcje inicjujące wszystkich podprogramów są uruchamiane albo w czasie ładowania systemu, albo podczas instalowania modułu z urządzeniem.

Ponieważ funkcja inicjująca nie ma bezpośredniego dostępu do tablic rozdzielczych, musi więc przy zapisie informacji o urządzeniu korzystać z odpowiedniej funkcji systemowej. Rolę tę spełnia para funkcji **register_blkdev** i **register_chrdev**. Jako parametry tych funkcji podaje się numer główny, pod którym urządzenie zostanie zarejestrowane oraz informację o tym urządzeniu, czyli jego nazwę i zestaw dostępnych operacji. Można zamiast konkretnego numeru głównego podać 0 – wtedy funkcja znajdzie pierwszą wolną pozycję w odpowiedniej tablicy.

Para funkcji rejestrujących ma swój odpowiednik w postaci pary funkcji wyrejestrowujących: **unregister_blkdev** i **unregister_chrdev**. Służą one do wyczyszczenia pozycji w tablicy rozdzielczej pod podanym numerem głównym.

2.6 Algorytmy

W poniższym punkcie zostaną omówione algorytmy działania funkcji systemowych dotyczących urządzeń. Szczegółowo zostaną przedstawione algorytmy funkcji: **open** i **close**. Z pozostałych funkcji systemowych zostanie omówiona jedynie para funkcji **read** i **write**, ponieważ algorytmy zastosowane w innych funkcjach nie odbiegają znacząco od algorytmu **read** / **write** – wywoływana jest tylko inna funkcja z podprogramu obsługi urządzenia. Poniższe opisy algorytmów zawierają tylko fragmenty istotne dla obsługi urządzeń – fragmenty dotyczące obsługi systemu plików zostały opisane w sposób bardzo ogólny, bez zagłębiania się w szczegóły implementacji.

2.6.1 open

użytkownik:
- wywoływana jest funkcja sys_open z pliku open.c

sys_open:
- pobierany jest kolejny deskryptor pliku, który od tej pory będzie służył do identyfikacji pliku urządzenia (o ile funkcja open się powiedzie)
- wywoływana jest funkcja do_open z deskryptorem jako dodatkowym parametrem

do_open:
```
- tworzona jest nowa struktura zawierająca informację o ot-
wartym pliku ( struct file )
- i-węzeł pliku jest kopiowany do pamięci
- do pola f_op struktury file jest kopiowany zestaw operacji
z i-węzła ( zestaw ten zawiera jedynie wskaźnik do funkcji
blkdev_open lub chrdev_open, patrz punkt 2.1 )
- wywoływana jest ww. funkcja blkdev_open (lub chrdev_open)
z pliku devices.c
```

blkdev_open lub chrdev_open:
```
- w strukturze pliku zestaw operacji zostaje zastąpiony
właściwym zestawem, pobranym z odpowiedniej tablicy roz-
dzielczej funkcją get_blkfops ( lub get_chrfops ) - numer
główny i drugorzędny jest obliczany przy pomocy pola i_rdev
z i-węzła
- wywoływana jest funkcja open z podprogramu obsługi
urządzenia ( o ile jest w zestawie dostępnych operacji )
```

open z podprogramu obsługi:
```
- działanie funkcji zależy od konkretnego urządzenia i
zostanie opisane w odpowiednich rozdziałach dotyczących
urządzeń blokowych i znakowych
```

do_open:
```
- struktura pliku jest włączana do tablicy plików aktualnie
otwartych w systemie
```

Wartości zwrócone przez funkcję **open** podprogramu obsługi (w tym wartości błędów) są propagowane "w górę" aż do funkcji **sys_open**, wywołując po drodze odpowiednie reakcje (np. zwolnienie deskryptora w przypadku niepowodzenia).

2.6.2 close

użytkownik:
```
- wywoływana jest funkcja sys_close z pliku open.c
```

sys_close:
```
- zwalniany jest deskryptor pliku i usuwana jest pozycja
tabeli otwartych plików
- wywoływana jest funkcja close_fp
```

close_fp:
```
- sprawdzany jest licznik odwołań do pliku - jeśli jest on
różny od 0 ( zamknięcie jest prawidłowe ), to wywoływana
jest funkcja fput zdefiniowana w pliku file.h
```

fput:
```
- jeśli licznik odwołań jest równy 1 ( czyli aktualnie
zamykane jest ostatnie odwołanie ), to wywoływana jest
funkcja __fput
```

__fput:
```
- jeśli w zestawie operacji dozwolonych na urządzeniu
znajduje się funkcja release, to jest ona wywoływana
```

release:
```
- tak jak w przypadku funkcji open - działanie zale-
ży od podprogramu obsługi
```

fput:
```
- wartość licznika odwołań jest zmniejszana o 1
```

2.6.3 read i write

użytkownik:
```
- wywoływana jest funkcja sys_read ( sys_write ) z pliku
read_write.c
```

sys_read (sys_write):
```
- sprawdzana jest poprawność deskryptora - czy istnieje plik
z nim związany, czy istnieje i-węzeł odpowiadający plikowi,
czy plik umożliwia odczyt ( zapis ), itd.
- jeżeli w zestawie operacji dozwolonych dla pliku znajduje
się funkcja read ( write ), to jest ona wywoływana
```

read (write):
```
- działanie zależy od urządzenia
```

Dodatkowo, dla funkcji **sys_write** wywołanie **write** z podprogramu obsługi urządzenia jest sekcją krytyczną – jest otoczone opuszczeniem i podniesieniem semafora z i-węzła.

Rozdział 3

Obsługa urządzeń fizycznych

3.1 Wstęp

W pozostałych rozdziałach podprogramy obsługi urządzeń omówione zostały głównie z punktu widzenia logicznego. Ten rozdział zawiera omówienie fragmentów jądra dotyczących bezpośrednio komunikacji z urządzeniami fizycznymi. Opisane funkcje dotyczą architektury sprzętowej i386, choć dla innych architektur powinny być one podobne. Różnice nie będą jednak omawiane. Rozdział oprócz przedstawienia części jądra odpowiedzialnej za bezpośrednie odwoływanie się do sprzętu szczegółowo przedstawia przerwania i implementację ich obsługi. Jest to tematem podrozdziału 3.4.

Kilka słów na temat architektury:
Komendy dla urządzeń przesyłane są przez tak zwane porty wejścia-wyjścia opisane szczegółowo w podrozdziale 3.2. Porty te umożliwiają ponadto odczyt informacji o stanie urządzenia oraz przesyłanie danych. Transmisja danych może odbywać się na kilka sposobów, zależnych przeważnie od szybkości urządzeń. Najwolniejsze z nich przesyłają informacje poprzez porty. Urządzenia, które wymagają możliwie najszybszego przesyłania dużych ilości danych stosują mechanizm mapowania pamięci (ang. *memmory mapped devices*). Oznacza to, że rejestry urządzenia znajdują się bezpośrednio w przestrzeni adresowej procesora. Przykładem mogą być karty graficzne. Nie będziemy się tym zajmować szczegółowo, ponieważ obsługa takich urządzeń odbywa się poza jądrem – w trybie użytkownika (przykład biblioteka **vgalib**). Inną metodą przesyłania danych jest mechanizm dualny do powyższego. Urządzeniu umożliwia się bezpośredni dostęp do pamięci operacyjnej (DMA – ang. *Direct Memmory Access*) poprzez tak zwane kanały DMA. Podczas przesyłania danych procesor nie jest zajmowany. Tę metodę stosują dosyć szybkie urządzenia takie jak na przykład dyski. Fragment jądra dotyczący DMA przedstawiony został w podrozdziale 3.5.

3.2 Porty wejścia-wyjścia

Fizyczne urządzenia posiadają specjalny, programowalny podukład służący do komunikacji z jednostką centralną, tzw. kontroler. Porty wejścia-wyjścia stanowią interfejs między procesorem, a rejestrami kontrolera. Procesor za pomocą specjalnych instrukcji **in** i **out** może odczytywać bądź zapisywać wartości portów. W ten sposób mogą być przesyłane dane lub komendy dla urządzeń oraz odczytywany ich stan. Porty mają numery od 0 do 65535. Znaczenie każdego z nich jest zależne od aktualnej konfiguracji sprzętu i nie można tego zmienić programowo. Podprogram obsługi musi albo znać numery portów odpowiadających obsługiwanemu urządzeniu, albo znaleźć je podczas swojej inicjalizacji.

3.2.1 Rezerwowanie portów wejścia-wyjścia

Pierwszą czynnością, którą wykona podprogram obsługi, powinno być sprawdzenie, czy obsługiwane urządzenie fizyczne rzeczywiście znajduje się w systemie. Sprawdzenie polega na wpisaniu do portów odpowiednich wartości kontrolnych i oczekiwaniu przez pewien czas na odpowiedź urządzenia. Aby uniknąć sytuacji, w której dany podprogram pisze do portów obcego urządzenia, jądro umożliwia rezerwowanie rozpoznanych portów. Mimo to, niepożądana sytuacja może się zdarzyć podczas inicjalizacji, gdy w systemie zarejestrowanych jest jeszcze niewiele urządzeń. Zaleca się więc aby te podprogramy dawały użytkownikowi możliwość wskazania aktualnej konfiguracji sprzętu.

Zarządzanie obszarami portów wejścia-wyjścia (ang. *io-region*) zaimplementowane zostało w pliku **kernel/resorce.c**. Zarezerwowany obszar opisuje struktura **resource_entry_t**:

```
struct resource_entry_t {
       u_long from, num;
       const char *name;
       struct resource_entry_t *next;
};
```

Kolejne pola oznaczają:

- adres pierwszego zarezerwowanego portu
- długość zarezerwowanego obszaru
- nazwę urządzenia rezerwującego obszar
- wskaźnik do następnego elementu z listy **iolist**

Struktury **resource_entry_t** zawarte są w tablicy **iotable** o rozmiarze ograniczonym do **IOTABLE_SIZE** (64 elementow) i połączone w uporządkowaną listę **iolist**. Do operacji na powyższych strukturach służą następujące funkcje:

- `int check_region(unsigned int from, unsigned int num)`
  sprawdza, czy dany obszar jest już zarezerwowany

- `void request_region(unsigned int, unsigned int, const char*)`
  jeśli podany obszar nie pokrywa się z już zarezerwowanym, to zostanie dodany do **iolist**

- `void release_region(unsigned int, unsigned int)`
  usuwa obszar z **iolist**

3.2.2 Korzystanie z portów wejścia-wyjścia

Do zapisywania i odczytywania wartości do/z portów służą funkcje zdefiniowane w pliku **include/asm-i386/io.h**:

- `outb/outw/outl(unsigned x value, unsigned short port)`
  zapisują do portu wartość typu x - odpowiednio char/short/int

- `inb/inw/inl(unsigned short port)`
  czytają z portu wartość odpowiedniego typu

- `outb_p, outw_p, outl_p, inb_p, inw_p, inl_p`
  zmodyfikowane wersje powyższych funkcji.
  Przyrostek **_p** oznacza "pauzę" – wytracenie kilku cykli przez procesor po zapisaniu bądź odczytaniu wartości. Stosuje się je w przypadku, gdy kontroler, z reguły wolniejszy, nie nadąża za procesorem z przekazywaniem danych.

- `outsb/outsw/outsl(unsigned short port, const void * addr, unsigned long count)`
  zapisują do portu ciąg wartości odpowiedniego typu z pod wskazanego adresu w pamięci; **count** określa długość ciągu

- `insb/insw/insl(unsigned short port, const void * addr, unsigned long count)`
  analogicznie czytają z portu ciąg wartości i umieszczają pod wskazanym adresem

Korzystanie z powyższych funkcji nie jest powiązane z mechanizmem ochrony portów opisanym w poprzednim podrozdziale. Nie jest nigdzie sprawdzane czy odwołanie odbywa się do zarezerwowanego portu. Nie mniej jednak dobrze napisany podprogram obsługi urządzenia powinien pisać wyłącznie do swoich zarezerwowanych obszarów.

3.3 Odpytywanie a przerwania

Podprogramy obsługi przekazują do urządzeń różnego rodzaju żądania zlecające wykonanie pewnych operacji. W przypadku twardego dysku są to na przykład (plik **include/hdreg.h**):

- **WIN_READ** - żądanie odczytania sektora z dysku

- **WIN_WRITE** - żądanie zapisania sektora na dysku
- **WIN_READDMA** - żądanie odczytania sektora z użyciem DMA
- **WIN_WRITEDMA** - żądanie zapisania sektora z użyciem DMA
- **WIN_FORMAT** - żądanie sformatowania dysku
- **WIN_IDENTYFY** - żądanie przekazania pewnych informacji o rodzaju dysku
- i wiele innych ...

Podprogram obsługi zwykle musi zorientować się, kiedy zlecona operacja zostanie zakończona albo, że wystąpił błąd. Ponieważ sprzęt zazwyczaj jest wolniejszy od jednostki centralnej podprogram musi przez pewien czas być w stanie oczekiwania. Schemat ten występuje w większości podprogramów obsługi urządzeń. Ze względu na sposób oczekiwania dzieli się je na dwie kategorie:

- podprogramy odpytujące (ang. *polling device drivers*)
- podprogramy przerwaniowe (ang. *interrupt-driven device drivers*)

Odpytywanie oznacza czytanie specjalnego rejestru stanu urządzania (ang. *status register*) do chwili gdy zmieni się jego wartość, wskazując zakończenie operacji. Cechą odpytywania jest aktywne czekanie. Może mieć to tragiczne skutki w postaci zablokowania systemu, ponieważ podprogram obsługi jest częścią jądra. Zwykle korzysta się więc z zegarów systemowych. Procedura odpytująca jest wykonywana w pewnych odstępach czasu i dzięki temu nie blokuje procesora. Zaletą odpytywania jest prosta budowa podprogramu obsługi oraz nie zajmowanie przerwań. Szczególnie druga cecha jest często dość istotna ze względu na ich małą ilość. Odpytywanie nie jest jednak tak wydajne jak mechanizm przerwań. Dzięki przerwaniom unika się całkowicie aktywnego czekania. W chwili zakończenia operacji urządzenie zgłasza przerwanie powodując uruchomienie funkcji obsługi. Do tego czasu procesor jest całkowicie zwolniony dla systemu. Ze względu na możliwość awarii lub zgubienia się przerwania, podprogramy obsługi często ustawiają sobie maksymalny czas oczekiwania, po upływie którego próbują na przykład ponowić żądanie. W Linuksie większość podprogramów dotyczących obsługi urządzeń fizycznych to podprogramy przerwaniowe. Na uwagę zasługuje podprogram obsługi drukarki, który standardowo jest odpytujący, ale po rozpoznaniu sprzętu dostarczającego przerwanie dla drukarki może przejść w tryb przerwaniowy. Szczegóły dotyczące implementacji korzystania z przerwań przedstawione zostały w następnym podrozdziale.

3.4 Obsługa przerwań

Na pytanie czym jest przerwanie (ang. *interrupt*) można odpowiedzieć następująco: jest to zmiana przepływu sterowania występująca w ściśle określonej sytuacji, polegająca na wywołaniu specjalnej procedury obsługi (ang. *handler*). Procedura ta jest globalna dla wszystkich programów w systemie. W architekturze i386 przerwanie występuje na skutek jednej z następujących przyczyn:

- zgłoszenie przerwania z zewnątrz procesora
- wywołanie przerwania przez program – za pomocą specjalnej instrukcji **int**
- wystąpienie błędu podczas wykonywania instrukcji – mówimy wtedy o wyjątku (ang. *exception*)

W tym podrozdziale zajmiemy się jedynie pierwszą sytuacją, ponieważ pozostałe nie są związane z tematem obsługi urządzeń.

Przerwania zewnętrzne są dwojakiego rodzaju:

- NMI – niemaskowalne (muszą zostać obsłużone przez procesor), powstają w skutek poważnych awarii sprzętowych (np. zanik napięcia)
- maskowalane, którymi się zajmiemy, powstają po otrzymaniu sygnału na linii procesora o nazwie INTR.

Maskowalność oznacza, że procesor może odmówić przyjęcia przerwania. Dzieje się tak gdy wyzerowana jest flaga **IF** procesora. Do zerowania tej flagi, czyli blokowania przerwań, służy makro **cli()**, a do ustawiania makro **sti()** z pliku **include/asm-i386/system.h**.

Atrybutem każdego przerwania jest nieujemna liczba całkowita, tak zwany wektor przerwania (ang. *interrupt vector*). W architekturze i386 dostępne są wektory od 0 do 255. Wektory służą między innymi jako indeksy w tablicy deskryptorów przerwań IDT (ang. *Interrupt Descriptor Table*), dzięki której można wskazać adres procedury obsługi danego przerwania. Do rejestracji takiej procedury służy makro **set_intr_gate(n, addr)** z pliku **include/asm-i386/system.h**. **N** jest wektorem przerwania a **addr** adresem tej procedury. Podprogramy obsługi urządzeń nie zajmują się jednak tak niskopoziomowym zarządzaniem przerwaniami. W tym celu jądro udostępnia im wspólny interfejs, który zostanie przedstawiony w podrozdziale 3.4.1.

W architekturze i386 za przekazywanie sygnałów przerwań od urządzeń fizycznych do procesora służy kontroler 8259A, zwany PIC (ang. *Programmable Interrupt Controler*). Układ ten udostępnia osiem wejść, do których można podłączyć urządzenia. Wejścia te nazywane są liniami **IRQ** (ang. *Interrupt Request Line*). Możliwe jest dzielenie linii **IRQ**, to znaczy podłączenie kilku urządzeń do jednego wejścia kontrolera. W systemie znajdują się dwa układy 8259A: PIC1 i PIC2. Wyjście PIC2 podłączone jest do wejścia numer 2 układu PIC1, a wyjście PIC1 do linii INTR procesora. System dysponuje więc 15 wolnymi liniami **IRQ** o numerach 0 – 15 (z wyjątkiem 2). Numerom tym odpowiadają wektory przerwań 0x20-0x2f. Układy PIC posiadają specjalne rejestry dostępne przez porty 0x20-0x3f (PIC1) i 0xa0-0xbf (PIC2). Dzięki nim po otrzymaniu sygnału INTR procesor potrafi rozpoznać, która linia IRQ zgłosiła przerwanie. Ponadto umożliwiają blokowanie i odblokowywanie pojedynczych linii. Szczegóły dotyczące znaczenia poszczególnych rejestrów kontrolera PIC zostały dokładnie opisane w projekcie **LabLinux**.

3.4.1 Rezerwacja przerwania i rejestracja procedury obsługi

Podobnie jak w przypadku portów wejścia-wyjścia jądro przechowuje informacje na temat przydzielonych przerwań **IRQ**. Dla wszystkich podprogramów obsługi urządzeń udostępniony jest wspólny, wygodny interfejs rezerwacji przerwań i rejestracji procedury obsługi. Implementacja znajduje się w pliku **/arch/i386/kernel/irq.c**. Procedura obsługi przerwania, którą rejestruje dany podprogram, musi mieć następujący typ:

```
void handler(int irq, void * dev_id, struct pt_regs * regs)
```

Po zgłoszeniu się przerwania będzie ona wywoływana przez niskopoziomowy podprogram obsługujący, o czym później.
Parametry oznaczają:

irq - numer przerwania

regs - struktura przechowująca rejestry procesora

dev_id - parametr wykorzystywany w przypadku, gdy przerwanie jest dzielone.

Przerwania mogą być dzielone między kilka podprogramów obsługi urządzeń. Z tymi przerwaniami związanych jest odpowiednio kilka procedur obsługi. Po zgłoszeniu przerwania są one wywoływane wszystkie po kolei. Taka procedura musi w jakiś sposób dowiedzieć się, czy przerwanie pochodzi od jej urządzenia. Podczas rejestracji procedury (o czym później) przekazywany jest **dev_id** – wskaźnik do pewnych danych identyfikujących urządzenie i jego podprogram obsługi. Ten sam wskaźnik (dla każdej procedury inny) przekazywany jest do procedury w momencie przerwania. Dzięki ustawieniu odpowiednich wartości tym danym procedura jest w stanie rozpoznać, czy przerwanie pochodzi od jej urządzenia. Inną metodą jest sprawdzenie rejestru stanu urządzenia. Podprogramy obsługi urządzeń umożliwiające dzielenie przerwań rejestrują swoje procedury obsługi z flagą **SA_SHIRQ**. Wszystkie muszą zgodzić się na współdzielenie przerwania.

Funkcją realizującą rejestrację jest:

```
int request_irq(unsigned int irq,
   void (*handler)(int , void *, struct pt_regs *),
   unsigned long flags,
   const char *device,
```

```
   void *dev_id)
```

Parametry oznaczają:

irq - numer przydzielanego przerwania

handler - wskaźnik do procedury obsługi przerwania

flags - **SA_INTERRUPT**, **SA_SHIRQ** – możliwe w alternatywie **OR**; znacznie **SA_INTERRUPT** zostanie wyjaśnione później

device - wskaźnik do nazwy urządzenia

dev_id - wskaźnik przekazywany do handlera opisany wcześniej

Funkcja zwraca 0 w przypadku sukcesu, **-EINVAL** gdy **irq** poza 0 – 15 lub **-EBUSY**, gdy przerwanie zajęte oraz nie zostało uzgodnione współdzielenie.

Usunięcie procedury obsługi przerwania następuje przy użyciu funkcji:

```
void free_irq(unsigned int irq, void *dev_id)
```

irq - numer przerwania

dev_id - wskaźnik do identyfikatora. W przypadku gdy przerwanie jest dzielone usuwa się tylko jedną procedurę.

Do przechowywaniu informacji o przerwaniach **IRQ** służą następujące struktury danych:

```
struct irqaction {
   void (*handler)(int, void *, struct pt_regs *);
   unsigned long flags;
   unsigned long mask;
   const char *name;
   void *dev_id;
   struct irqaction *next;
};
```

Struktura **irqaction** z pliku **include/kernel/interrupt.h** przechowuje informacje dotyczące jednej procedury obsługi. Pola tej struktury odpowiadają parametrom przekazanym w **request_irq()**. Ponadto wskaźnik **next** umożliwia połączenie w listę. Niewyjaśnione znaczenie ma dla mnie pole **mask**. Nie spotkałem się z sytuacją by było ono różne od 0. Funkcje służące do blokowania i odblokowywania linii **IRQ** (**disable_irq(irq_nr)** i **enable_irq(irq_nr)** z pliku **arch/i386/kernel/irq.c**) odwołują się bezpośrednio do rejestrów kontrolera PIC i nie korzystają ze struktury **irqaction** na przykład w celu zaznaczenia, że przerwanie jest zamaskowane.

```
struct irqaction * irq_action[16]
```

Plik **arch/i386/kernel/irq.c**. Struktury **irqaction** przydzielane są dynamicznie wewnątrz funkcji **request_irq()** i zapamiętywane w tablicy **irq_action**. Indeks tej tablicy oznacza numer przerwania. Pole o wartości **NULL** oznacza, że procedura obsługi nie została zarejestrowana. Dla przerwań dzielonych pola **irq_action** przechowują listę.

Podprogram obsługi nie zawsze musi znać numer przerwania, do którego jest podłączone jego urządzenie. W pliku **include/linux/interrupt.h** przedstawiony jest algorytm korzystania z funkcji **probe_irq_on()** i **probe_irq_off()** w celu znalezienia tego numeru:

1. wyzeruj wewnętrzne przerwania urządzenia
2. **sti()**
3. `irqs = probe_irq_on()` - zapamiętaj maskę przerwań nie mających procedur obsługi i nieaktywnych
4. spowoduj, aby urządzenie zgłosiło przerwanie
5. odczekaj pewien czas potrzebny do nadejścia przerwania
6. `irq = probe_irq_off(irqs)` - zwrócony zostaje numer przerwania które zaszło, zero gdy nie było żadnego lub jeśli było kilka – minus numer największego
7. Obsłuż urządzenie, aby wróciło do normalnego trybu pracy

3.4.2 Niskopoziomowa obsługa przerwań

Powyżej przedstawiony został interfejs umożliwiający podprogramom rejestracje procedur obsługi przerwań. W tym podrozdziale omówiony zostanie niskopoziomowy kod jądra, odpowiadający między innymi za wywoływanie tych procedur. Wszystkie omawiane funkcje i struktury danych znajdują się w plikach **arch/i386/kernel/irq.c** i **include/asm-i386/irq.h**.

Jak wspomniano wcześniej, procesor po otrzymaniu sygnału przerwania znajduje odpowiedni adres zapamiętany w **IDT**, pod który następuje skok. Jądro przechowuje trzy tablice adresów niskopoziomowych procedur:

- **void (*interrupt[16])(void)** - zwykłe
- **void (*fast_interrupt[16])(void)** - "szybkie"
- **void (*bad_interrupt[16])(void)** - niezarezerwowane.

W IDT na pozycjach odpowiadającym wektorom linii **IRQ** zawsze znajdują się adresy pochodzące z jednej z tych trzech tablic. Podczas inicjalizacji (przy starcie systemu) za pomocą **set_intr_gate()** ustawia się je wszystkie na **bad_interrupt**. Adresy te zmieniane są wewnątrz **request_irq()** i **free_irq()**. Pierwsza po wypełnieniu struktury **irqaction** ustawia odpowiedni adres na **interrupt** bądź **fast_interrupt**. Rodzaj określa flaga **SA_INTERRUPT** oznaczająca przerwanie "szybkie". W przypadku przerwań dzielonych wszystkie podprogramy muszą zgodzić się co do wartości tej flagi. **Free_irq()**, po zwolnieniu odpowiedniej struktury **irqaction**, przywraca adres **bad_interrupt**, pod warunkiem, że zwalniana struktura była jedyną na liście.

Obsługa **bad_interrupt**, powoduje jedynie zamaskowanie przerwania (aby później nie obsługiwać ich niepotrzebnie). Natomiast dla procedur zwykłych i "szybkich" algorytmy obsługi przerwania, różnią się nieznacznie (**nr** oznacza numer przerwania):

1. zachowaj rejestry – zwykła zachowuje wszystkie, "szybka" tylko część
2. zamaskuj przerwanie **nr**
3. **STI** - wyłącz przerwania – tylko zwykła
4. wywołaj wszystkie funkcje obsługi przerwania z listy **irq_action[nr]**
5. **CLI** - włącz przerwania
6. odmaskuj przerwanie **nr**
7. "szybka" - przywróć rejestry, zwykła – wywołaj **ret_from_sys_call()**

W przypadku procedury "szybkiej" powrót następuje bezpośrednio do miejsca, w którym nastąpiło przerwanie. Procedura zwykła wywołuje funkcje **ret_from_sys_call()**, czyli powrót jest taki jak ze zwykłej funkcji systemowej. **Ret_from_sys_call()** oprócz szeregowania procesów obsługuje, między innymi dolne połowy, o których w następnym podrozdziale.

3.4.3 Górne i dolne połowy

W algorytmie z poprzedniego podrozdziału przerwania są maskowane i wyłączane. Pożądane jest aby taki stan nie trwał zbyt długo, ponieważ w tym czasie może zostać zgłoszone przerwanie, które w rezultacie nie zostanie obsłużone co dalej wiąże się z utratą pewnych danych. Z drugiej strony podprogram obsługi przerwania musi nieraz wykonywać czasochłonne operacje na przykład kopiować dane z portów do buforów. W tym celu kod obsługujący przerwanie dzieli się na dwie części. Są to tak zwane górna i dolna połowa:

Górna połowa (ang. *top half*) to wszystko to o czym była mowa do tej pory: niskopoziomowe procedury jądra i funkcje rejestrowane przez podprogramy obsługi urządzeń. Pożądane jest aby wykonywały się jak najszybciej.

Dolna połowa (ang. *bottom half*) to funkcja zarejestrowana do wykonania w późniejszym czasie z odmaskowanymi przerwaniami.

Jądro utrzymuje tablicę **bh_base** (plik **include/linux/interrupt.h**), w której przechowywane są wskaźniki do funkcji. Do operacji na tablicy służą:

- `void init_bh(int nr, void (*routine)(void))`
  zapamiętuje funkcje na pozycji nr

- `void mark_bh(int nr)`
  zaznacza pozycję nr do wywołania.

Wywołanie zaznaczonych funkcji w kolejności rosnących numerów odbywa się przy najbliższym wykonaniu funkcji **schedule()** lub **ret_from_sys_call()**. Po wykonaniu są one odznaczane. Rozmiar tablicy **bh_base** wynosi 32, co okazało się niewystarczające. W związku z tym w Linuksie zaimplementowane zostały kolejki zadań - dodatkowy mechanizm powiązany ściśle z dolnymi połowami.

3.4.4 Kolejki zadań

Kolejka zadań służy do zapamiętania listy funkcji typu "dolna połowa". Jednej takiej funkcji odpowiada następująca struktura (plik **include/linux/tqueue.h**):

```
struct tq_struct {
   struct tq_struct * next;
   int sync;
   void (* routine)(void *);
   void *data;
};
```

Kolejne pola oznaczają:

- wskaźnik do następnego elementu na liście
- pomocnicze, musi być zainicjalizowane na 0
- zapamiętana funkcja - dolna połowa
- dane przekazywane do funkcji w chwili uruchomienia kolejki

Typ definiujący kolejkę zadań:

```
typedef struct tq_struct * task_queue
```

Do operacji na kolejkach zadań służą:

```
void queue_task(struct tq_struct *bh_pointer, task_queue *bh_list)
void run_task_queue(task_queue *list)
```

Pierwsza umieszcza strukturę **tq_struct** na liście. Druga powoduje wywołanie wszystkich funkcji pamiętanych w kolejce. Funkcje wywoływane są w kolejności odwrotnej do zapamiętywania. Przed wywołaniem są one wszystkie usunięte z listy, w związku z czym mogą dowolnie operować na kolejce – na przykład ponownie umieścić w niej siebie.

Kolejki zadań są dość wygodnym mechanizmem obsługi przerwań. Są one wykorzystywane przez podprogramy obsługi urządzeń blokowych – w szczególności dysków. Obsługa przerwań z podziałem na górne i dolne połowy oraz z wykorzystaniem kolejek zadań ma następujący schemat:

1. Inicjacja podprogramu obsługi:
   - utworzenie kolejki zadań i odpowiednich struktur **tq_struct**
   - umieszczenie funkcji dolnej połowy w tablicy **bh_base**
   - zarejestrowanie procedury obsługi przerwania (górnej połowy)
2. Procedura obsługi przerwania:
   - umieszczenie w kolejce przeznaczonych zadań (struktur **tq_struct**)
   - zaznaczenie do wykonania funkcji dolnej połowy
3. Dolna połowa
   - uruchomienie kolejki zadań

3.5 Bezpośredni dostęp do pamięci (DMA)

Funkcje dotyczące przydzielania i używania kanałów DMA zaimplementowane zostały w plikach **kernel/dma.c** oraz **include/asm-i386/dma.h**.

Kanały DMA są urządzeniami, do których dostęp umożliwiają porty dwóch kontrolerów. Kontroler 1 odpowiada za kanały 0-3, a kontroler 2 za kanały 4-7, przy czym kanał 4 używany jest do kaskadowego połączenia tych kontrolerów. Dostępnych kanałów jest więc niewiele. Ponadto, w przeciwieństwie do przerwań **IRQ**, nie można ich współdzielić.
Istnieją również następujące ograniczenia:

- kanały 0-3 przesyłają dane w porcjach jednobajtowych, 4-7 w dwubajtowych
- transfer jest ograniczony do dolnych 16MB pamięci, ze względu na sposób adresowania
- jednorazowo maksymalnie można przesłać 64kb w przypadku kanałów 0-3, 128 kb w przypadku 4-7

3.5.1 Rezerwowanie kanałów DMA

Jądro utrzymuje następujące struktury danych dotyczące rezerwacji kanałów **DMA** zadeklarowane w **kernel/dma.c**:

- ```
  struct dma_chan {
      int lock;
      const char *device_id;
  }
  ```

- `static struct dma_chan dma_chan_busy[MAX_DMA_CHANNELS]`

Tablica **dma_chan_busy** ma rozmiar **MAX_DMA_CHANNELS** równy 8. Pole **lock** przyjmuje wartości: 1 - zajęty, 0 - wolny. **Device_id** zawiera nazwę urządzenia, korzystającego z kanału.
Do operacji na tych strukturach, czyli do przydzielania i zwalniania kanałów służą funkcje:

- `int request_dma(unsigned int dmanr, const char * device_id)` rezerwuje kanał o numerze **dmanr** lub zwraca błąd jeśli jest on niedostępny

- `void free_dma(unsigned int dmanr)` zwalnia kanał lub zwraca błąd jeśli nie był on zarezerwowany

3.5.2 Korzystanie z kanałów DMA

Podprogram obsługi urządzenia korzysta z **DMA** w następujący sposób:

1. przydziela pamięć na bufor w przypadku trybu **DMA_MODE_READ**

2. inicjuje transmisje danych

3. oczekuje na zakończenie (sygnalizowane przerwaniem)

Inicjacja transmisji **DMA** odbywa się poprzez określenie kierunku przepływu danych, adresu bufora w pamięci operacyjnej i liczby przesyłanych bajtów (w przypadku kanałów 0-3) lub liczby słów (dla 4-7). Służą do tego następujące funkcje z pliku **include/asm-i386/dma.h** (**dmanr** oznacza numer kanału):

- `void set_dma_mode(unsigned int dmanr, char mode)`
  mode to tryb (kierunek) transmisji: **DMA_MODE_READ** lub **DMA_MODE_WRITE**

- `set_dma_addr(unsigned int dmanr, unsigned int a)` **a** to adres początku bufora. Wykorzystuje się tylko młodsze 24-bity. Szczegóły dotyczące znaczenia poszczególnych bitów są dobrze udokumentowane w pliku źródłowym.

- `set_dma_count(unsigned int dmanr, unsigned int count)` **count** oznacza liczbę przesyłanych bajtów, zmniejszoną o 1 (dla kanałów 0-3) lub o 2 (dla 5-7). Ponadto w tym drugim przypadku musi być parzysta.
  ```

- `void enable_dma(unsigned int dmanr)` włączenie danego kanału
- `void disable_dma(unsigned int dmanr)` wyłączenie danego kanału

Inicjacja odczytu może więc wyglądać następująco:

```
cli();                                  /* Zablokuj przerwania */
disable_dma(channel);                   /* Wyłącz kanał */
set_dma_mode(channel, DMA_MODE_READ);   /* Ustaw tryb odczytu */
set_dma_addr(channel, buffer);          /* Ustaw adres bufora */
set_dma_count(channel, count);          /* Ustaw rozmiar danych */
enable_dma(channel);                    /* Włącz kanał */
sti();                                  /* Odblokuj przerwania */
```

Podobnie jak w przypadku portów wejścia-wyjścia korzystanie z kanałów nie jest powiązane z ich rezerwacją. Nie jest nigdzie sprawdzane czy używany kanał jest rzeczywiście zarezerwowany.

Rozdział 4

Urządzenia blokowe

Cechy urządzeń blokowych:

- Reprezentowane w systemie plików jako specjalne pliki blokowe
- Dostęp do zasobów za pomocą mechanizmu buforowania (ang. *buffer cache*)
- Urządzenie blokowe jest urządzeniem o dostępie bezpośrednim (ang. *random access device*)
- Specjalne pliki blokowe mogą być montowanie w systemie plików
- Urządzenia blokowe podzielone są na dwie klasy:
  - klasa urządzeń **SCSI**
  - klasa urządzeń **IDE**

4.1 Inicjalizacja

W czasie startu systemu lub inicjalizacji modułu wywoływana jest funkcja **blk_dev_init** (plik **ll_rw_blk.c**). Inicjuje ona wszystkie urządzenia blokowe, których program obsługi został wkompilowany w jądro. Funkcja **blk_dev_init** dla każdego z urządzeń wykonuje jego własną funkcję inicjalizacyjną - funkcję **urz_init()**, gdzie **urz** jest kilkuliterowym skrótem nazwy urządzenia. Funkcja **urz_init()** dla wszystkich urządzeń blokowych zadeklarowana jest w pliku **include/linux/blk.h**.

Funkcja **register_blkdev** (plik **/fs/devices.c**) rejestruje podprogram urządzenia blokowego. Korzysta z niej np. funkcja **urz_init** inicjująca dane urządzenie.

4.2 Operacje I/O

4.2.1 Wstęp

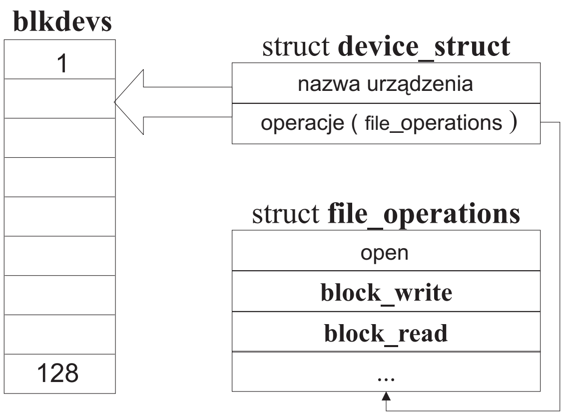
- Procedura obsługi urządzenia blokowego dostarcza jak inne procedury obsługi urządzeń strukturę **file_operations**, zawierającą funkcje implementujące interfejs do danego urządzenia.

- Program obsługi urządzenia blokowego nie implementuje procedur **read** i **write**. W strukturze **file_operations** wskaźniki **read** i **write** ustawia na procedury **block_read** i **block_write**, wspólne dla wszystkich urządzeń blokowych.

- Realizacja operacji I/O:

  1. Program żąda operacji wejścia / wyjścia na danym urządzeniu blokowym
  2. System plików w którym zamontowane jest to urządzenie ustala numer bloku w którym znajdują się żądane dane.
  3. Jądro wysyła do podręcznej pamięci buforowej żądanie pobrania z danego urządzenia odpowiedniego bloku.
  4. Blok ten może znajdować się już w podręcznej pamięci buforowej i wtedy nie trzeba wczytywać danych z urządzenia
  5. Jeśli blok danych nie znajduje się w żadnym buforze wtedy jądro wysyła żądanie do urządzenia pobrania odpowiedniego bloku

4.2.2 Struktury danych



Rysunek 4.1: Struktura **device_struct**

- `device_struct blkdevs[MAX_BLKDEV]`
  Plik **fs/devices.c**
  Tablica w której trzymane są zarejestrowane urządzenia blokowe. Tablica jest indeksowana za pomocą głównego numeru urządzenia.

- ```
  struct device_struct {
          const char * name;
          struct file_operations * fops;
  };
  ```
  Plik **fs/devices.c**.

- `int * blk_size[MAX_BLKDEV]`
  Plik **include/linux/blkdev.h**.
  Dwuwymiarowa tablica zawierająca rozmiary wszystkich urządzeń blokowych w jednostkach 1024 bajtowych sektorów. Odwołanie do elementu tablicy: **blk_size[MAJOR][MINOR]**

- `int * blksize_size[MAX_BLKDEV]`
  Plik **include/linux/blkdev.h**.
  Dwuwymiarowa tablica zawierająca wielkość bloku dla każdego urządzenia blokowego. Jeśli **blksize_size[MAJOR]** == **NULL**, to przyjmowana jest wielkość 1024 bajtów. Odwołanie do elementu tablicy: **blksize_size[MAJOR][MINOR]**

- `#define MAX_BLKDEV 128` Plik **include/linux/major.h**

4.2.3 Algorytm block_read (plik fs/block_dev.c)

Procedura **block_read** ma za zadanie wczytanie do pamięci pod wskazany adres zadanej liczby bajtów z danego pliku specjalnego. Jest to procedura wspólna dla wszystkich urządzeń blokowych, dlatego podprogram obsługi urządzenia nie implementuje własnej procedury **read** (zob. **4.2.1 Wstęp**).
Ogólne działanie:

1. Ustal pierwszy blok i liczbę bloków do przeczytania.
2. Pobierz bufory z pamięci buforowej.
3. Utwórz kolejkę żądań dla buforów, które nie zawierają aktualne dane.
4. Wyślij żądania do urządzenia.
5. Poczekaj na wypełnienie buforów danymi z urządzenia.
6. Skopiuj dane pod wskazany adres w pamięci.
7. Zwolnij pobrane bufory.

Szczegółowe działanie:

- Wejście:
  - struct inode * inode - i-węzeł pliku specjalnego
  - struct file * filp - struktura opisująca czytany plik
  - char * buf - bufor na odczytane dane
  ```

  - int count - liczba bajtów do przeczytania

- Wyjście: Liczba przeczytanych bajtów lub kod błędu EIO.
- Główne lokalne zmienne i struktury danych:
  - buflist[NBUF] - lista pobranych buforów
  - bhreq[NBUF] - lista pobranych buforów, które zawierają nieaktualne dane

```
 block_read
{
   Ustal wielkość bloku dla danego urządzenia;
   Ustal rozmiar danego urządzenia;
   Oblicz numer pierwszego bloku do odczytania i offset w tym bloku;
   Oblicz liczbę bloków do odczytania;
   do {
     while (pozostały bloki do przeczytania) {
       Pobierz bufor z pamięci buforowej
         ( getblk (urządzenie, nr_bufora, wielkość_bloku) );
       Dodaj bufor do listy buflist;
       if ( bufor zawiera nieaktualne dane )
         dodaj bufor do kolejki żądań (bhreq);
       if ( jest to pierwszy obrót pętli i bufor jest aktualny )
         wyjdź z pętli;
       if ( tablica buforów buflist się zapełniła )
         wyjdź z pętli;
     }
     Zażądaj wczytania danych do buforów będących na liście bhreq
       ( ll_rw_block(READ, liczba_buforów_do_wczytania, bhreq) );
     do {
       Weź kolejny nieobsłużony bufor z listy buflist;
       Czekaj na wczytanie danych do tego bufora ( wait_on_buffer );
       if ( bufor nie jest aktualny ) { /* błąd czytania? */
         Zwolnij bufor (brelse);
         liczba bajtów do przeczytania = 0;
         Wyjdź z pętli;
       }
       Uaktualnij:
         - liczbę bajtów do przeczytania,
         - liczbę przeczytanych bajtów,
         - pozycję w pliku;
       Kopiuj dane z bufora do przestrzeni użytkownika;
       Zwolnij bufor;
     } while ( liczba bajtów do przeczytania > 0 i
         na liście buflist znajdują się nieobsłużone bufory i
         właśnie obsługiwany bufor nie był zajęty (locked) )
```

```
  } while ( liczba bajtów do przeczytania > 0 );
  Zwolnij wszystkie pozostałe bufory (brelse);
  if (liczba przeczytanych bajtów = 0)
    return -EIO;
  return (liczba przeczytanych bajtów);
}
```

4.2.4 Algorytm block_write (plik fs/block_dev.c)

Procedura **block_write** ma za zadanie zapisanie do danego pliku specjalnego zadanej liczby bajtów pobranych z podanego adresu w pamięci. Jest to procedura wspólna dla wszystkich urządzeń blokowych, dlatego podprogram obsługi urządzenia nie implementuje własnej procedury **write** (zob. **4.2.1 Wstęp**).
Ogólne działanie:

1. Ustal pierwszy blok i liczbę bloków do zapisania.
2. Pobierz bufory z pamięci buforowej.
3. Jeśli zapisujemy niepełny blok i ten blok jest nieaktualny to wczytaj blok z urządzenia.
4. Kopiuj dane z podanego adresu w pamięci do buforów.
5. Jeśli zapis ma być synchroniczny to wyślij żądania zapisu do urządzenia, czekaj na zakończenie zapisu i zwolnij bufory.

Szczegółowe działanie:

- Wejście:
  - struct inode * inode - i-węzeł pliku specjalnego
  - struct file * filp - struktura opisująca plik, na którym zapisujemy dane
  - const char * buf - dane w pamięci, które zapisujemy do pliku
  - int count - liczba bajtów, które zapisujemy
- Wyjście: liczba zapisanych bajtów lub kody błędów: EPERM, ENOSPC, EIO.
- Główne lokalne zmienne i struktury danych:
  - struct buffer_head * bhlist[NBUF] - lista buforów wymagających odczytu z urządzenia (czyli buforów nieaktualnych)
  - struct buffer_head * bh, *bufferlist[NBUF] - lista buforów przeznaczonych do zapisu synchronicznego

```
 block_write
{
   if (plik jest tylko do odczytu)
     return -EPERM;
   Ustal wielkość bloku dla danego urządzenia;
   Oblicz numer pierwszego bloku do zapisu i offset w tym bloku;
   Ustal rozmiar urządzenia;
   while (liczba bajtów pozostałych do zapisania > 0) {
     if (numer bloku do zapisania > rozmiar urządzenia)
       return -ENOSPC;
     Pobierz bufor z pamięci buforowej ( getblk(urządzenie, numer_bloku,
         wielkość_bloku) );
     if (zapisujemy niepełny blok i blok jest nieaktualny)
       if (czytanie z wyprzedzeniem) {
         Pobierz bufory na bloki czytane z wyprzedzeniem (getblk);
         if (nie udało się pobrać jakiegoś bloku)
           return (liczba przeczytanych bajtów lub -EIO, gdy
               liczba przeczytanych bajtów = 0);
         Stwórz listę żądań z pobranych buforów;
         Wyślij żądania odczytu do urządzenia
             ( ll_rw_block(READ, liczba_bloków_do_odczytu, bhlist)
);
         Zwolnij wszystkie bufory oprócz pierwszego (brelse);
         Czekaj na zakończenie wczytywania danych do pierwszego
           bufora (wait_on_buffer);
       } else {
         Zwolnij bufor (brelse);
         Wczytaj blok z urządzenia (bread);
       }
     Kopiuj dane z przestrzeni użytkownika do bufora;
     Zaznacz bufor jako aktualny (mark_buffer_uptodate);
     if (zapis ma być synchroniczny)
       Dodaj bufor do listy (bufferlist) buforów przeznaczonych do
           zapisu synchronicznego;
     if ( zapełniona tablica bufferlist ) {
       Wyślij żądania zapisu bloków z bufferlist do urządzenia (ll_rw_block);
       Czekaj na zapis wszystkich buforów (wait_on_buffer);
       if (jakiś bufor zawiera nieaktualne dane)
         return -EIO;
       Zwolnij wszystkie bufory;
       Opróżnij listę bufferlist;
     }
   }
   if ( jest coś na liście bufferlist ) {
     Wyślij żądania zapisu bloków z bufferlist do urządzenia (ll_rw_block);
     Czekaj na zapis wszystkich buforów (wait_on_buffer);
```

```
    if (jakiś bufor zawiera nieaktualne dane)
      return -EIO;
    Zwolnij wszystkie bufory;
    Opróżnij listę bufferlist;
  }
  return liczba zapisanych bajtów;
}
```

4.3 Szeregowanie żądań

Jądro może żądać wielu operacji wejścia/wyjścia dla danego urządzenia. Dlatego implementuje listy żądań obsługi dla urządzeń blokowych. Obsługa wielu żądań może zostać usprawniona, jeśli żądania zostaną obsłużone w odpowiedniej kolejności. Jądro stosuje w tym celu algorytm **C-LOOK** tzn. wstawia żądania w taki sposób, aby:

- głowica dysku obsługiwała żądania poruszając się tylko w jednym kierunku (tzn. powrotny ruch głowicy jest jałowy);
- głowica poruszała się tylko pomiędzy skrajnymi żądaniami (a nie po całym dysku);

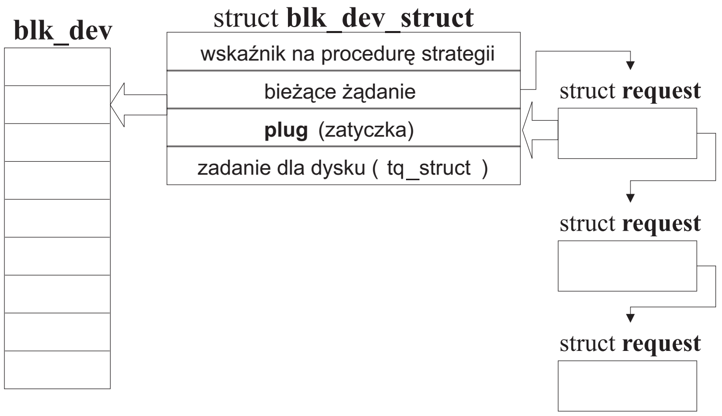
Za każdym razem kiedy dodawane jest nowe żądanie, zostaje ono wstawione na odpowiednie miejsce w kolejce żądań. Utworzeniem, wstawieniem do kolejki i wysyłaniem żądań związanych z listą buforów zajmuje się procedura **ll_rw_block**. Korzysta ona z procedur **make_request** i **add_request**. Procedura **make_request** zajmuje się utworzeniem pojedynczego żądania wejścia/wyjścia. Procedura **add_request** zajmuje się wstawieniem żądania do kolejki żądań danego urządzenia. Obsługą kolejnych żądań z kolejki zajmuje się już podprogram obsługi urządzenia przy pomocy procedury strategii (ang. *strategy routine*).

4.3.1 Struktury danych

Każde urządzenie blokowe posiada w tablicy **blk_dev** (na pozycji równej numerowi głównemu tego urządzenia) strukturę (**blk_dev_struct**) opisującą kolejkę żądań skierowanych do niego oraz procedurę obsługi tych żądań (ang. *strategy routine*).

- ```
  blk_dev_struct {
          void (*request_fn)(void);
          struct request * current_request;
          struct request plug;
          struct tq_struct plug_tq;
  };
  ```

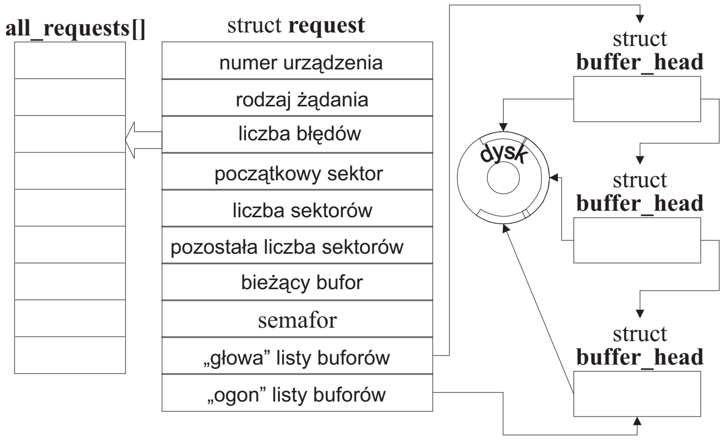
  Plik **include/linux/blkdev.h**.
  Gdzie:
  ```



Rysunek 4.2: Struktura **blk_dev_struct**

- **request_fn** - procedura strategii obsługująca kolejne żądania z kolejki.
- **current_request** - wskaźnik na początek kolejki żądań obsługi wejścia/wyjścia dla danego urządzenia (na bieżące żądanie).
- **plug** - "wtyczka", specjalne nieaktywne ("obsłużone") żądanie. Umożliwia rozpoczęcie transferu danych dopiero po wstawieniu do kolejki wszystkich żądań. Wstawienie "wtyczki" do kolejki zapobiega natychmiastowemu uruchomieniu procedury strategii w procedurze **add_request** (która czyni to, gdy wstawia żądanie do pustej kolejki). Wskaźnik **current_request** ustawiany jest na tę "wtyczkę" w momencie kiedy tworzone jest nowe żądanie, a kolejka żądań jest pusta, zanim zostanie wywołana procedura **add_request**.
- **plug_tq** - struktura typu **task_queue** zawierająca procedurę **unplug_device**. Struktura ta jest wstawiana do kolejki zadań **tq_disk**, zaraz po tym jak wskaźnik **current_request** ustawiony zostaje na "wtyczkę" (**plug**). Oznacza to, że jeśli później wykonamy procedurę **run_task_queue** dla kolejki **tq_disk** (co jest robione np. w **wait_on_buffer**) to zostanie uruchomiona procedura **unplug_device**, która z kolejki żądań usuwa "wtyczkę" (**plug**) a następnie wywołuje procedurę strategii dla pozostałych żądań w tej kolejce.

- `struct blk_dev_struct blk_dev[MAX_BLKDEV];`
  Plik **devices/block/ll_rw_blk.c**.
  Tablica struktur wskazujących na procedurę strategii i kolejkę żądań, inicjalizowana przez **blk_dev_init()**. Indeksowana numerem głównym urządzenia.



Rysunek 4.3: Struktura **request**

- ```
  struct request {
          volatile int rq_status;
          kdev_t rq_dev;
          int cmd;
          int errors;
          unsigned long sector;
          unsigned long nr_sectors;
          unsigned long current_nr_sectors;
          char * buffer;
          struct semaphore * sem;
          struct buffer_head * bh;
          struct buffer_head * bhtail;
          struct request * next;
  };
  ```
  Plik **include/linux/blkdev.h**.
  Struktura opisuje pojedyncze żądanie.
  Gdzie:

  - **rq_status** - opisuje czy żądanie zostało już wykonane.
  - **rq_dev** - określa urządzenie do którego skierowane jest żądanie.
  - **cmd** - rodzaj żądania (odczyt lub zapis).
  - **errors** - licznik błędów otrzymanych podczas realizacji żądania. Jeśli liczba błędów przekroczy próg ustalony przez driver, realizacja żądania zostaje zakończona.
  - **sector**, **nr_sectors** - początkowy sektor i liczba sektorów, których dotyczy żądanie (żądanie związane jest ze spójnym obszarem urządzenia).
  ```

- **current_nr_sectors** - liczba sektorów, która pozostała do odczytania/zapisania.
- **buffer** - wskaźnik do bieżącego bufora.
- **sem** - semafor używany przez programy obsługi urządzeń **SCSI** i przy korzystaniu z pliku wymiany.
- **bh**, **bhtail** - wskaźniki na początek i koniec listy buforów odpowiadających spójnemu fragmentowi (którego żądanie dotyczy) urządzenia.
- **next** - następne żądanie w kolejce żądań do danego urządzenia.

• `#define NR_REQUEST 64`
  Plik **include/linux/blk.h**.

• `static struct request all_requests[NR_REQUEST];`
  Plik **devices/block/ll_rw_blk.c**.
  Wszystkie żądania dla wszystkich urządzeń są alokowane statycznie i mieszczą się w tablicy **all_requests**. Linux ogranicza liczbę wszystkich żądań zapisu do 2/3 * **NR_REQUEST**. Zatem 1/3 tablicy **all_requests** może być zajęta tylko przez żądania odczytu.

4.3.2 Algorytm ll_rw_block

Algorytm **ll_rw_block** (plik **devices/block/ll_rw_blk.c**) zajmuje się:

- wysyłaniem żądań zapełnienia listy buforów danymi odczytanymi z urządzenia
- zapisem danych z buforów na urządzenie.

Wejście:

- rodzaj żądania
- liczba buforów na liście
- lista buforów

```
 ll_rw_block
{
   if (wielkość buforów nie odpowiada buforom urządzenia)
   {
     błąd;
     return;
   }
   if (żądania zapisu na urządzenie i urządzenie jest tylko do odczytu)
   {
     błąd;
     return;
   }
   Dla każdego bufora wywołaj procedurę
     make_request(nr główny urządzenia, rodzaj żądania, bufor);
}
```

4.3.3 Algorytm make_request

Algorytm **make_request** (plik **devices/block/ll_rw_blk.c**) zajmuje się utworzeniem żądania wejścia/wyjścia dla podanego bufora.
Wejście:

- główny numer urządzenia
- rodzaj żądania
- wskaźnik na bufor

```
 make_request
{
   Zablokuj bufor;
   if (odwołanie poza zakres urządzenia) {
     błąd;
     return;
   }
   if (bufor jest aktualny) {
     Odblokuj bufor;
     return;
   }
   if (kolejka żądań jest pusta i nie jest to
       urządzenie loop lub MD)
   {
     Wstaw do kolejki żądań "wtyczkę" (plug);
     Wstaw do kolejki tq_disk strukturę plug_tq;
   } else {
     Pomiń pierwszy element w kolejce; /*może być już obsługiwany*/
     Dla każdego elementu (żądania) w kolejce:
       if ( bufor można wstawić na początek listy buforów żądania,
           tak aby bufory odnosiły się do spójnego obszaru urz. )
       {
         Wstaw bufor na początek listy buforów tego żądania;
         return;
       } else if ( bufor można wstawić na koniec listy buforów
           żądania, tak aby bufory odnosiły się do spójnego
           obszaru urządzenia )
       {
         Wstaw bufor na koniec listy buforów tego żądania;
         return;
       }
   }
}
```

4.3.4 Algorytm add_request

Algorytm **add_request** (plik **devices/block/ll_rw_blk.c**) dodaje gotową strukturę **request** do kolejki żądań danego urządzenia. Realizuje przy tym planowanie dostępu do dysku metodą **C-LOOK**.

Kolejka żądań jest sortowana najpierw ze względu na numer urządzenia. Ponieważ do jednego urządzenia może być wiele żądań, więc żądania grupują się w podkolejki - dla każdego urządzenia po jednej. Umownie sektorem żądania będę nazywał pierwszy sektor do którego odnosi się to żądanie.

- W każdej podkolejce oprócz pierwszej (tj. tej, w której znajduje się bieżące żądanie) wszystkie żądania są posortowane rosnąco według numeru sektora żądania. Gdy pojawia się nowe żądanie to wstawiamy je w takim miejscu aby zachować uporządkowanie.

- Pierwsza kolejka (tj. ta, w której jest bieżące żądanie) nie może być po prostu posortowana wg numeru sektora. Dlaczego? Spróbujmy odpowiedzieć na poniższe pytania:

  - Co się dzieje jeśli pojawia się żądanie z mniejszym numerem sektora niż numer sektora żądania bieżącego?
  - Gdzie mamy takie żądanie umieścić? Przecież nie możemy wstawić żądania przed żądaniem bieżącym!

  Odpowiedź:
  Kolejka zawierająca bieżące żądanie składa się z dwóch części. Pierwsza część kolejki = bieżące żądanie + wszystkie żądania z sektorem o większym numerze niż numer sektora bieżącego żądania. Druga część kolejki = wszystkie żądania z sektorem o mniejszym numerze niż numer sektora bieżącego żądania. Obie te części są posortowane rosnąco według numeru sektora. Gdy więc pojawia się żądanie z sektorem o mniejszym numerze niż numer sektora bieżącego żądania to wstawiamy je do drugiej części kolejki, tak aby zachować uporządkowanie w tej drugiej części. Taki podział zapewnia, że głowica obsługując żądania przesuwa się w jednym kierunku. Kiedy dojdzie do końca "pierwszej części", oznacza to, że doszła do skrajnego sektora (dla którego istniało żądanie) i musi powrócić do drugiego skrajnego sektora, dla którego żądanie właśnie znajduje się na początku "drugiej części". Oczywiście w tym momencie "druga część" staje się "częścią pierwszą", a nowa "druga część" jest pusta.

```
add_request
{
     /* Wstaw żądanie do kolejki zgodnie z algorytmem C-LOOK: */
     if (wstawiane żądanie nie dotyczy tej samej podkolejki co
         bieżące żądanie)
       Wstaw żądanie do kolejki, tak aby zostało zachowane
           uporządkowanie;
```

```
    else /*wstawiamy do podkolejki w której jest bieżące żądanie*/
      if (numer sektora wstawianego żądania > numer sektora
          bieżącego żądania)
        Wstaw żądanie do pierwszej części kolejki
      else
        Wstaw żądanie do drugiej części kolejki;
    if (w kolejce przed wstawieniem nie było żadnych żądań lub
        urządzenie jest typu SCSI)
      Wywołaj procedurę strategii danego urządzenia
}
```

Procedura strategii danego urządzenia obsługuje pierwsze żądanie w kolejce i jeśli kolejka zawiera następne żądania procedura strategii zostanie wywołana dla każdego z nich.

4.4 Procedury strategii

Procedura strategii jest ważną procedurą każdego podprogramu obsługi urządzenia blokowego. Wszystkie żądania odczytu i zapisu bloków, są realizowane przez tę procedurę. Korzysta ona z niskopoziomowych instrukcji do komunikowania się z urządzeniem. Procedura strategii jest częścią podprogramu obsługi urządzenia dlatego jej sposób działania zależy od konkretnego urządzenia, które ten podprogram obsługuje. Można jednak podać ogólny zarys procedury i warunki jakie musi ona spełniać aby zapewnić poprawne działanie systemu.

4.4.1 Opis procedury strategii

Procedura strategii wywoływana jest bez żadnych parametrów, nie zwraca też żadnych wartości, ale wie gdzie znaleźć kolejkę żądań (**CURRENT** wskazuje na bieżące żądanie dla aktualnego urządzenia) i dane do transferu. Wywoływana jest z zablokowanymi przerwaniami i na nią spada odpowiedzialność ich odblokowania (**sti()**). Na początku powinna wykonywać makro **INIT_REQUEST**, które sprawdza czy zachodzą pewne niezbędne warunki (np. czy lista żądań nie jest pusta). Jeśli driver wykorzystuje przerwania (ang. *"is interrupt-driven"*) procedura strategii tylko inicjuje żądanie (komunikując się z urządzeniem przez porty) i ustawia odpowiednią obsługę przerwania, które zostanie zgłoszone przez urządzenie, gdy będzie gotowe do transmisji. Procedura obsługująca to przerwanie realizuje transmisję danych, wywołuje **end_request** i jeśli są następne żądania to wywołuję ponownie procedurę strategii. Jeśli driver nie obsługuje przerwań to procedura strategii kończy działanie dopiero, gdy wszystkie żądania obsługi zostaną obsłużone. Jeżeli z jakiegoś powodu operacja I/O nie może być wykonana procedura obsługi wywołuje **end_request** z parametrem 0.

4.4.2 Algorytm end_request

Procedura **end_request** jest wywoływana przez driver po zakończeniu wykonywania każdego żądania.
Wejście:

- int uptodate - 1 gdy żądanie zakończyło się pomyślnie, 0 gdy z błędem

```
 end_request
{
   if (uptodate == 0)
     Wypisz komunikat o błędzie;
   if (uptodate == 1)
     Zaznacz bieżący bufor jak aktualny (mark_buffer_uptodate);
   Odblokuj bufor;
   if (żądanie zawiera następny bufor do obsługi) {
     Następny bufor staje się aktualnym buforem;
     return;
   }
   Zaznacz bieżące żądanie jako wykonane (nieaktywne);
   Bieżącym żądaniem staje się następne żądanie;
   Obudź procesy czekające na miejsce w tablicy żądań
       (all_requests[]);
}
```

Rozdział 5

Urządzenia znakowe

5.1 Wstęp

Urządzenia znakowe, to urządzenia o dostępie sekwencyjnym. Należą do nich terminale, myszy, karty muzyczne itp. Ponieważ nie mają one żadnych wspólnych funkcji ani struktur, więc zajmę się tylko omówieniem jednej, chyba najważniejszej grupy urządzeń znakowych – terminali. Terminale są tak ważne (i powszechne) ponieważ służą do kontaktu między użytkownikiem a systemem. Najpierw omówię funkcje i struktury danych wspólne dla wszystkich urządzeń terminalowych, następnie najczęściej używaną dyscyplinę linii **N_TTY** i na końcu dwa podprogramy obsługi urządzeń terminalowych – konsole i pseudoterminale.

5.2 Terminale

5.2.1 Pojęcia podstawowe

Terminal sterujący - z tego terminala proces odbiera sygnały **SIGHUP**, **SIGINT**, **SIGSTP**. Proces zwykle dziedziczy po swoim przodku ten terminal, ale może go zmienić za pomocą funkcji **ioctl**. Terminal ten jest opisywany przez strukturę **tty_struct** wskazywaną przez pole **tty** w **task_struct** (struktura opisująca proces).

Grupa terminalowa - grupa procesów związana z danym terminalem T, której identyfikator jest równy identyfikatorowi grupy terminalowej terminala T (pole **pgrp** w strukturze **tty_struct**). Najczęściej proces należy do grupy terminalowej swojego terminala sterującego.

Przywódca grupy terminalowej - proces, którego identyfikator sesji (pole **session** w **task_struct**) jest równy identyfikatorowi sesji terminala grupy procesów do której należy ten proces (wtedy też **task_struct.leader** wynosi 1). Zwykle przywódcą grupy terminalowej jest proces **shell**.

Numeracja urządzeń - Każde urządzenie terminalowe w Linuksie ma numer główny 4. Podział numerów drugorzędnych jest następujący:

0	aktywna konsola (ta do której kierowane są znaki z klawiatury)
1 ... 63	konsole
64 ... 127	terminale podłączone do portów szeregowych
128 ... 191	pseudoterminale nadrzędne
192 ... 255	pseudoterminale podrzędne, przy czym pseudoterminal podrzędny o numerze n odpowiada terminalowi nadrzędnemu o numerze n − 64

5.2.2 Główne struktury danych

```
struct termio {
    unsigned short c_iflag;
    unsigned short c_oflag;
    unsigned short c_cflag;
    unsigned short c_lflag;
    char c_cline;
    char c_cc[NCC];
};
```

Opis pól:

c_iflag - definiuje sposób przetwarzania znaków wejściowych (otrzymanych od urządzenia).

c_oflag - określaje sposób traktowania znaków wyjściowych (wysyłanych do urządzenia).

c_cflag - definiuje techniczne parametry pracy urządzenia terminalowego .

c_lflag - określają tryb działania dyscypliny linii, tzn. czy znaki nadchodzące z urządzenia układać w wiersze i czy wyświetlać ich ech, a jeśli tak, to w jaki sposób.

c_line - rodzaj dyscypliny linii związanej z danym urządzeniem terminalowym.

c_cc - tablica znaków specjalnych (sterujących, jak np. znak końca pliku).

W Linuksie używana jest też druga, podobna do **termio**, struktura **termios**. **Termios** używana jest w jądrze, natomiast **termio** pozostaje w celu zachowania zgodności z wcześniej napisanymi programami. Różnica między tymi dwiema strukturami jest niewielka: różni je tylko typ niektórych pól (w **termio** jest to **unsigned short**, a w **termios** – **unsigned int**).

```
struct tty_struct {
     ...
     struct tty_driver driver;
     struct tty_ldisc ldisc;
     struct termios *termios, *termios_locked;
     int pgrp;
     int session;
     ...
};
```

Podstawowa struktura opisująca terminal zdefiniowana w **include/linux/tty.h**. Opis najważniejszych pól:

driver - podprogram obsługi urządzenia

ldisc - dyscyplina linii

termios - parametry pracy urządzenia terminalowego

termios_locked jeśli pewien bit na tym polu jest ustawiony, to wywołania funkcji **ioctl** nie są w stanie zmieniać wartości bitu o tym samym numerze w polu **termios** (zabezpiecza przed lekkomyślnym zmienianiem flag sterujących terminalem).

pgrp - identyfikator grupy terminalowej

session - identyfikator sesji (równy identyfikatorowi sesji przywódcy grupy terminalowej tego terminala).

```
struct tty_driver { ...}
```

Struktura (zdefiniowana w **include/linux/tty_driver.h**) opisująca podprogram obsługi urządzenia. Zawiera ona również deklaracje funkcji stanowiących interfejs do konkretnego urządzenia (urządzenie jako sprzęt – na bardzo niskim poziomie).

```
struct tty_ldisc {
     ...
     (*open) (struct tty_struct *);
     (*close) (struct tty_struct *);
     (*read) (struct tty_struct *tty, struct file *file,
               unsigned char *buf, unsigned int nr);
```

```
    (*write) (struct tty_struct *tty, struct file *file,
              const unsigned char *buf, unsigned int nr);
    (*ioctl) (struct tty_struct *tty, struct file *file,
              unsigned int cmd, unsigned long arg);
    ...
};
```

Struktura zdefiniowana w **include/linux/tty_ldisc.h**. Zawiera wskaźniki do kolejnych funkcji udostępnianych przez dyscyplinę linii (będzie omówiona dalej).

5.2.3 Główne algorytmy

Wszystkie opisane funkcje zdefiniowane są w pliku **driver/char/tty_io.c**.

init_dev – zaalokowanie pamięci i zainicjowanie struktury opisującej urządzenie terminalowe

1. Znajdź strukturę opisującą sterownik terminala (**get_tty_driver()**)
2. Ze struktury opisującej sterownik urządzenia pobierz wskaźniki do struktur opisujących nasze urządzenie: **tty**, **termios**, **termios_locked**
3. Dla każdej ze struktur wykonaj:
   jeśli wskaźnik do struktury był **NULL** to zaalokuj pamięć, w p.p. zakończ algorytm
4. Jeśli otwierane urządzenie jest pseudoterminalem (**PTY**), to powtórz poprzedni krok, aby otworzyć drugiego członka pary (patrz 5.6 Pseudoterminale)

tty_open

1. Wykonaj funkcję **init_dev**
2. Jeśli sterownik związany z urządzeniem terminalowym ma procedurę otwarcia (czyli `driver->open != NULL`) to ją wykonaj, w p.p. zwróć błąd **ENODEV**

tty_read

1. Sprawdź, czy nie zaszedł błąd (np. struktura opisująca urządzenie terminalowe jest pusta), jeśli tak to zakończ
2. Wywołaj funkcję **tty->ldisc->read**

tty_write – analogiczna do **tty_read**

do_tty_hangup – poinformowanie procesów podłączonych do danego terminala o tym, że nastąpiło zerwanie komunikacji (np. użytkownik wylogował się lub nastąpiło przerwanie transmisji przez łącze szeregowe)

1. Przeglądaj wszystkie i-węzły plików specjalnych (musimy tak robić, gdyż kilka plików specjalnych może wskazywać na jedno urządzenie) i dla tych, które są związane ze zrywanym właśnie terminalem wykonaj funkcję **tty_fasync** (usunięcie pliku z kolejki **tty->fasync** – plików związanych z terminalem)
2. Ustaw operacje dostępne na pliku na operacje otrzymane jako parametr
3. Opróżnij bufor dyscypliny linii (**ldisc->flush_buffer**)
4. Opróżnij bufor podprogramu obsługi (**driver->flush_buffer**)
5. Zamknij aktualną dyscyplinę linii i zainicjuj dyscyplinę **N_TTY**
6. Dla każdego procesu, który spełnia warunki:
   – należy do sesji związanej z danym terminalem (`p->session == tty->session`)
   – jest przywódcą grupy terminalowej (`p->leader == 1`)
   wyślij sygnały **SIGHUP**, **SIGCNT** i jego terminal sterujący ustaw na **NULL**
7. Wywołaj funkcję zerwania dla podprogramu obsługi (**driver->hangup**)

tty_ioctl - wykonanie poleceń specyficznych dla terminali

1. Wykonaj instrukcję **switch** (ze względu na podany parametr **cmd** – polecenie do wykonania)
2. Jeśli parametr **cmd** nie został znaleziony, to wykonaj instrukcję **driver->ioctl**, jeśli zwróci ona błąd **ENOIOCTLCMD**, to wykonaj **ldisc->ioctl**

5.3 Dyscyplina linii

Dyscyplina linii, to wstępne przetwarzanie znaków wejściowych i wyjściowych pomiędzy terminalem, a procesem.

Główne funkcje dyscypliny linii:

- dzieli wejściowy ciąg znaków na wiersze
- przetwarza znaki ścierające
- przetwarza znak kasowania wiersza, który unieważnia wszystkie dotychczas wprowadzone znaki bieżącego wiersza
- wypisuje na terminalu echo odebranych znaków
- rozbudowuje dane wyjściowe (np. zamienia znaki tabulacji na ciągi spacji)
- wysyła sygnały do procesów informując o zawieszeniu terminala, zerwaniu z nim łączności lub w odpowiedzi na naciśnięcie przez użytkownika specjalnej kombinacji klawiszy

- umożliwia pracę w trybie surowym polegającym na braku przetwarzania znaków specjalnych, takich jak znak ścierający, kasowania wiersza czy końca wiersza.

Interfejs dyscypliny linii jest zdefiniowany w **include/linux/tty_ldisc.h**.
W Linuksie są zdefiniowane następujące dyscypliny linii:

- N_TTY
- N_MOUSE
- N_SLIP
- N_PPP
- N_STRIP

5.3.1 N_TTY

Najczęściej używaną w Linuksie dyscypliną linii jest **N_TTY**. Oto jej niektóre algorytmy (pierwsze dwa – wołane przez użytkownika, następne przez podprogramy obsługi; funkcje te zdefiniowane są w pliku **driver/char/N_tty.c**) :

write_chan - w interfejsie **write**

1. Sprawdź, czy terminal jest w porządku (nie jest np. zastopowany), jeśli nie to koniec.
2. Wstaw się (tzn. proces wywołujący) na kolejkę procesów czekających na zapis do urządzenia i wykonuj w pętli:
   - Sprawdzaj, czy np. połączenie nie jest zerwane lub czy nie zapisaliśmy wszystkiego co chcieliśmy
   - Jeśli jest ustawiona flaga **O_OPOST**, to: w pętli pobieraj pojedyncze znaki i je przetwarzaj (funkcja **opost**) aż do momentu przepełnienia bufora urządzenia wyjściowego lub zapisania wszystkiego co chcieliśmy
   - Jeśli flaga **O_OPOST** nie była ustawiona to wywołaj **driver->write**
3. Usuń się z kolejki procesów czekających na zapis

read_chan - w interfejsie **read**

1. Sprawdź, czy terminal jest w porządku (nie jest np. zastopowany), jeśli nie to koniec.
2. Wstaw się (tzn. proces wywołujący) na kolejkę procesów czekających na zapis do urządzenia i wykonuj w pętli:
   - Sprawdzaj, czy np. połączenie nie jest zerwane lub czy nie odczytaliśmy już wszystkiego co chcieliśmy

  - Jeśli terminal jest w stanie kanonicznym (flaga **L_ICANON**), to w pętli odczytuj pojedyncze znaki i kopiuj je do przestrzeni użytkownika (funkcja **put_user**), aż do opróżnienia bufora urządzenia wejściowego lub odczytania wszystkiego co chcieliśmy
  - W p.p. (to znaczy terminal jest w trybie surowym) prześlij tyle ile się da znaków (minimum z zawartości bufora i liczby znaków do odczytu) do przestrzeni użytkownika (funkcja **copy_from_read_buf**)
  - Jeśli bufor wejściowy jest dostatecznie pusty, powiadom o tym jego podprogram obsługi (funkcja **tty->driver->unthrottle**)

3. Usuń się z kolejki procesów czekających na odczyt.

n_tty_receive_buf - jest wołana przez podprogram obsługi zajmujący się odbiorem znaków (np. z klawiatury)

1. Jeśli terminal pracuje w trybie surowym to przekopiuj znak będący parametrem do bufora użytkownika
2. W p.p. wywołaj funkcję zależną od flagi znaku
3. Jeśli flagą była **TTY_NORMAL**, to jest wołana funkcja **n_tty_receive_char**, która analizuje otrzymane znaki. W przypadku innych flag odpowiednie funkcje wstawiają do bufora pewne ustalone znaki lub przekazują błąd

n_tty_receive_room - daje w wyniku ilość wolnego miejsca w buforze wejściowym tzn. tym do którego pisze sterownik (lub w przypadku pseudoterminali, drugi członek pary)

n_tty_receive_char - dostaje pojedynczy wczytany znak i interpretując go: zmienia stany terminala, wyświetla echo, generuje sygnały itp.

5.4 Podprogramy obsługi terminali

W systemie są różne rodzaje urządzeń terminalowych. Tym co je różni jest to jaki mają podprogram obsługi, tzn. moduł dokonujący końcowego "wyrzucenia" znaków i jednocześnie odczytującego znaki z zewnątrz i za pomocą **ldisc->receive_buf** przekazującym je do dyscypliny linii.

W Linuksie zdefiniowanych jest pięć rodzajów sterowników:

- **TTY_DRIVER_TYPE_SYSTEM**
- **TTY_DRIVER_TYPE_CONSOLE**, czyli konsole
- **TTY_DRIVER_TYPE_SERIAL**, czyli terminale podłączane do linii szeregowych

- **TTY_DRIVER_TYPE_PTY**, czyli pseudoterminale, dzielą się one na dwa podtypy:
  - **PTY_TYPE_MASTER** – nadrzędny
  - **PTY_TYPE_SLAVE** – podległy
- **TTY_DRIVER_TYPE_SCC**

Praktycznie jednak używane są tylko trzy z nich, tzn. podprogramy obsługi konsoli, łącza szeregowego i pseudoterminali. Interfejs podprogramu obsługi tego typu jest podany w pliku **include/linux/tty_driver.h**.

Dwie funkcje podprogramu mogą wydać się nieoczywiste: **throttle** i **unthrottle**. Wywołanie funkcji **throttle** informuje podprogram obsługi, że bufor wejściowy dyscypliny linii jest przepełniony i należy zaczekać z wkładaniem tam nowych znaków, aż do wywołania funkcji **unthrottle**.

5.5 Konsola

Konsola (ang. *console*) jest jednym z 4 rodzajów podprogramów obsługi urządzeń terminalowych zaimplementowanych w Linuksie. Jest to podprogram najbardziej złożony z dwóch powodów:

- Jest on blisko związany ze sprzętem, dlatego też, przy odczytywaniu pojedynczego klawisza musimy wykonać dużo sprawdzeń w rodzaju czy jest naciśnięty Shift, stan klawisza NumLock, CapsLock, w przypadku innych klawiatur również klawisz Meta itp. Podobnie przy wyprowadzaniu znaków na ekran
- Drugi powód to fakt, że twórcy Linuksa chcieli dać użytkownikowi możliwość używania wirtualnych konsol (lub, jak kto woli, wirtualnych terminali). Pojawił się od razu problem, w jaki sposób przełączać się miedzy owymi konsolami tak aby było to niezauważalne gdziekolwiek indziej. To (tzn. rozpoznanie znaków które np. mają spowodować zmianę konsoli) powinno być robione nawet wcześniej niż dane trafiają do dyscypliny linii, bo my właściwie nie wiemy co dyscyplina linii zrobi z odczytanymi znakami

Autorzy Linux'a z drugim problemem poradzili sobie w następujący sposób. Jest zdefiniowana tablica odwzorowań kodów pozycyjnych (ang. *scan code*) w "regularne" kody 2-bajtowe. Ich regularność polega na tym, że najstarsze bity kodu klawisza opisują jego typ. Ponadto jest zdefiniowana 16-elementowa tablica wskaźników do funkcji, każda funkcja w tej tablicy odpowiada typowi klawisza. W ten sposób my możemy dokonać wstępnej obróbki klawiszy zanim staną się one widoczne gdziekolwiek indziej.

Tak więc algorytm odczytywania klawisza jest następujący:

1. Nacisnęliśmy klawisz

2. Zostało zgłoszone przerwanie klawiatury
3. `keysym = key_map[keycode]`
4. Weź typ klawisza o kodzie **keysym**
5. Wywołaj odpowiednią funkcję zależną od typu klawisza podając jej jako parametr **keysym**

Funkcje wejścia-wyjścia dla konsol są zdefiniowane w pliku **drivers/char/console.c**

5.6 Pseudoterminale

Pseudoterminal, to jeden z 4 rodzajów sterowników urządzeń terminalowych zdefiniowanych w Linuksie, wykorzystywany głównie do zdalnego logowania się przez sieć i pracy w X-ach.

Pseudoterminal składa sie z dwóch urządzeń: nadrzędnego i podległego. Te dwie części mogą się ze sobą komunikować: jeśli coś zapiszemy do części nadrzędnej, to będziemy w stanie to odczytać z części podległej i odwrotnie.

Część podległa pseudoterminala udostępnia taki sam interfejs jak urządzenie terminalowe (dyscyplina linii). Dostęp do części nadrzędnej odbywa się natomiast w trybie surowym - bez jakiejkolwiek ingerencji w przesyłane znaki (tak aby proces zajmujący sie komunikacją po prostu przesłał te znaki w sieć w nadziei, ze w zdalnej maszynie zostaną one jakoś przetworzone).

W **tty_struct** znajduje się pole **link**. To pole jest wskaźnikiem na terminal – drugiego członka pary. Funkcje zdefiniowane dla pseudoterminali są dość proste: np. **pty_write** powoduje po prostu natychmiastowe wywołanie funkcji **ldisc->receive_buf** u drugiego członka pary, nie ma kopiowania, gdyż oba urządzenia mają wspólny bufor tymczasowy. Otwarcie pseudoterminala jest też proste: jeśli otwieramy pseudoterminal nadrzędny, to od razu tworzymy strukturę opisującą drugiego członka pary. Natomiast jeśli otwieramy terminal podległy, to czekamy aż ktoś będzie otwierał odpowiadający mu terminal nadrzędny.

Funkcje wejścia-wyjścia pseudoterminali zdefiniowane są w pliku **drivers/char/pty.c**.

Użytkowanie pseudoterminali jest opisane w książce W.R.Stevensa "Programowanie zastosowań sieciowych w systemie Unix" rozdział 15.