Wejście-wyjście w systemie operacyjnym Linux. Wstęp.

Marcin Kowalczyk
Piotr Kryska
Krzysztof Socki
Radosław Szklarczyk
Rafał Wijata

1998/12/29

Streszczenie

W dokumencie tym postaramy się opisać na czym polega komunikacja z urządzeniami zewnętrznymi w systemie Linux, jakie, jako programiści, mamy do dyspozycji mechanizmy i jak one są zaimplementowane w samym systemie.

Spis rzeczy

1 Programy obsługi urządzeń[1]

1.1 Rola programów obsługi urządzeń

Sam system operacyjny nie mogąc się komunikować z urządzeniami zewnętrznymi byłby bezużyteczny: użytkownik nie mógłby ani sterować działaniem systemu, ani w żaden sposób obejrzeć wyników pracy systemu.

Jakie warunki powinniśmy stawiać programom obsługi urządzeń. Przede wszystkim powinny dostarczać interfejs, najlepiej *jednolity dla wszystkich urządzeń*, do komunikacji z fizycznymi urządzeniami. Ponadto powinien zabezpieczać urządzenia przed użytkownikiem dwojako: ograniczać dostęp użytkownikowi niepowołanemu, a także, na co często nie zwraca się uwagi, chronić urządzenie przed fizycznym uszkodzeniem spowodowanym niewłaściwym sterowaniem.

1.2 Przyjęte rozwiązanie w Linuksie

1.2.1 System plików

Linux doskonale spełnia wymagania postawione w poprzednim punkcie. Komunikacja odbywa się przez *system plików* - fizyczna komunikacja z urządzeniem jest ukryta przed użytkownikiem, a cała wymiana informacji ze światem zewnętrznym odbywa się poprzez takie operacje, jak dla systemu plików: przykładowo `read`, `write` czy `lseek`. Jest to wygodny, a przede wszytkim, jednolity[2] interfejs.

1.2.2 Inne udogodnienia

Przed każdą z operacji wykonywana jest funkcja jądra, która wykonuje część operacji, dzięki czemu kod funkcji podprogramu obsługi urządzenia[3] może być b. krótki.

Możliwe jest, że twórca programu obsługi zdecyduje, iż któraś z operacji, którą mógłby zaimplementować, nie jest potrzebna: wtedy, naturalnym byłoby, że użytkownik wywołujący taką funkcję otrzymuje błąd. Tak rzeczywiście jest, choć nie we wszystkich przypadkach. I tak przykładowo nie jesteśmy zobligowani do własnoręcznego implementowania funkcji `lseek` - system oferuje "za darmo" własną, często wystarczającą implementację, nie komunikującą się co prawda z fizycznym urządzeniem, ale uaktualniającą struktury systemowe.

Ochrona przed niepowołanymi użytkownikami realizowana jest na poziomie sytemu plików - użytkownik ma takie prawa wywoływać funkcje podprogramu, jakie prawa dostępu do pliku specjalnego reprezentującego urządzenie: sam podprogram obsługi nie musi tego w żaden sposób kontrolować.

Ponad to wszystko jednemu urządzeniu fizycznemu odpowiadać może wiele plików specjalnych - cecha praktycznie nie wykorzystywana, ale warto o tym wiedzieć.

[1] Autorem tego i następnego rozdziału jest Radosław Szklarczyk

[2] Znaczy to nie tylko, że gdy np. wymienimy kartę muzyczną na najnowszy model, nie zmieni sposób komunikacji programów z rzeczoną kartą, ale także, że kacie muzycznej możemy przesłać dane w ten sam sposób, jak choćby terminalowi.

[3] Terminu "podprogramy obsługi urządzenia" używamy zamiennie z "programy obsługi urządzenia"

1.3 Identyfikacja w systemie

1.3.1 Numery główne i drugorzędne

Poszczególne podprogramy są jednoznacznie identyfikowane w systemie Linux przy pomocy numerów głównych. Poszczególne instancje urządzenia fizycznego w systemie (możemy mieć wiele twardych dysków tego samego typu) identyfikuje numer drugorzędny (w zakresie 0-255). Jako ciekawostkę dodam, że jednoznaczności tych numerów pilnuje (a dokładniej przyznaje numery główne) jedna osoba, ang. *device registrar*.

1.3.2 Urządzenia blokowe a znakowe

W Linuksie mamy doczynienia z dwoma podstawowymi klasami urządzeń: urządzeniami blokowymi oraz urządzeniami znakowymi. Najważniejszą cechą tych pierwszych jest dostęp swobodny. Jak sama nazwa mówi operacje odczytu/zapisu wykonujemy blokami, a nie pojedyńczymi słowami. Jest to absolutnie niezbędne, jeśli chcielibyśmy montować systemy plików. Dodatkowo dostęp do urządzen blokowych jest buforowany: odbywa się to w sposób dla użytkownika przeźroczysty. Przykładem są tu oczywiście wszelkiego rodzaju dyski.

Urządzenia znakowe, to takie, do których dostęp nie jest swobodny. Natura ich jest sekwencyjna - znaki odczytywane są pojedyńczo, poza tym nie są buforowane na poziomie systemu operacyjnego. Przykłady urządzeń znakowych to porty szeregowe (mysz), równoległe, terminale.

Warto tu dodać iż urządzenie znakowe o numerze głównym k to zupełnie inne urządzenie niż blokowe o tym samym numerze (patrz: tablice rozdzielcze).

1.3.3 Pliki specjalne[4]

Numer główny, drugorzędny, a także typ urządzenia zapisane są w pliku specjalnym. plik specjalny tworzymy programem `mknod`. Przykładowo:

```
mknod nic c 3 1
```

stworzy w katalogu bieżącym plik `nic` reprezentujący urządzenie znakowe o numerze głównym 3, drugorzędnym 1 (gwoli dokładności jest to to samo urządzenie co `/dev/null`). Tradycja mówi, iż wszystkie pliki specjalne powinny być w katalogu `/dev`, ale nic nie stoi na przeszkodzie, by pliki specjalne tworzone były w dowolnym katalogu, o ile pozwala na to system plików[5]. Konsekwencją tego jest fakt, o którym już wspomniałem, iż wiele plików specjalnych może odpowiadać jednemu urządzeniu. Plik tego typu stworzyć może tylko superużytkownik.

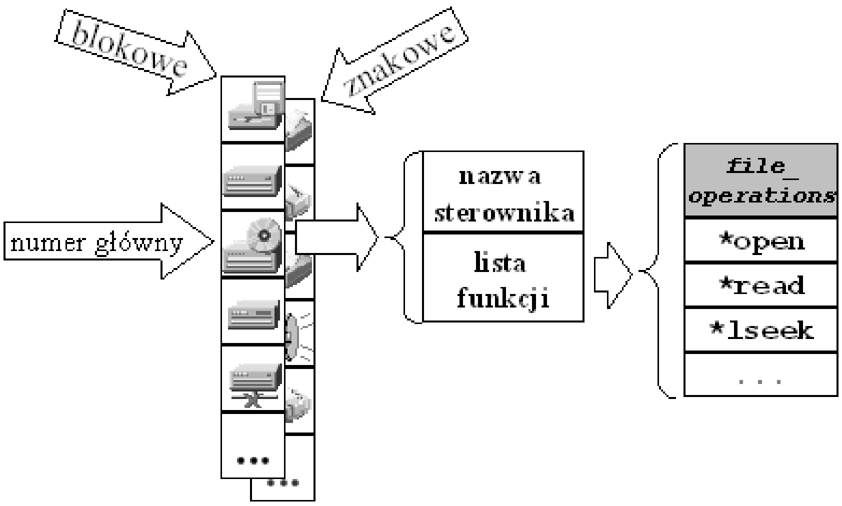
[4] Przez plik specjalny w tym dokumencie rozumiem znakowy/blokowy plik specjalny (character/block special file). Dokumentacja do `mknod` wymienia wśród plików specjalnych jeszcze FIFO.

[5] Nie wszystkie systemy plików zezwalają na zakładanie plików specjalnych: jest to podstawowe wymaganie stawiane systemowi montowanemu jako katalog główny (ext, ext2, umsdos) ale nie np. fat, vfat czy iso9660

1.4 Tablice rozdzielcze

1.4.1 Tablice rozdzielcze

W Linuksie istnieją dwie tablice w których znajują się "opisy" urządzeń: inna tablica dla urządzeń znakowych, inna dla blokowych. Obie indeksowane są numerem drugorzędnym urządzenia. Na każdej pozycji tablicy rozdzielczej zapisana jest nazwa podprogramu, a także wskaźnik na strukturę `file_operations`.



Zaglądnijmy w kod (dla urządzeń znakowych):

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

#define MAX_CHRDEV 128
    static struct device_struct chrdevs[MAX_CHRDEV] = {
    { NULL, NULL },
};
```

1.4.2 Struktura `file_operations`

W pliku fs.h zadeklarowana jest ona tak:

```
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);
    };
```

W każdym polu tej struktury zapamiętany jest adres funkcji bądź NULL, podprogram jej nie obsługuje.

Jak widać file_operations definiuje nam o różnym stopniu integracji ze sprzętem: znaleźć możemy w niej zarówno funkcję `ioctrl`, bardzo ze sprzętem zintegrowaną, jak i `readdir` z mającą z urządzeniem fizycznym zupełnie luźne koneksje.

1.5 Inicjalizowanie urządzeń w jądrze[6]

Każdy podprogram obsługi urządzenia posiada funkcję init_module() bądź xxx_init() (ta pierwsza wywołuje najczęściej tą drugą). W tych właśnie funkcjach odbywa się inicjalizacja procedury obsługi i urządzenia. Inicjalizacją wszystkich podprogramów wkompilowanych w jądro zajmują się funkcje chr_dev_init() oraz blk_dev_init(). Wywołanie ich następuje w funkcji sys_setup() wywoływanej przez proces INIT. Dzięki temu można z nich korzystać dosyć wcześnie. Funkcje te po kolei wywołują wszystkie wkompilowane funkcje init dla każdego urządzenia.[7] . Podprogramy obsługi używane jako moduły inicjalizowane są przez kerneld lub polecenie insmod.

Inicjalizacja podprogramu obsługi urządzenia obejmuje:

- utworzenie wszelkich potrzebnych struktur danych;
- zarejestrowanie urządzenia w jądrze (wstawienie do odpowiedniej tablicy rozdzielczej odpowiednich struktur)
- znalezienia sposobu komunikacji z urządzeniem czyli wykrywanie portów, przerwań i ewentualnie kanałów DMA.

Do rejestrowania urządzeń w jądrze służą funkcje `register_chrdev(unsigned int major, const char name, struct file_operations *fops)` i `register_blkdev(...)`. Działają one prawie tak samo. Jeśli `major` czyli numer główny jest różny od zera to w to miejsce tablicy rozdzielczej wstawia nazwę i strukturę `file_operations`. Wpp szuka ostatniego wolnego miejsca i tam wstawia odpowiednia wartości.

[6] Autorem tego podrozdziału jest Piotr Kryska

[7] Wyjątkiem jest tu tylko urządzenie znakowe o numerze głównym 1 (mem)

2 Operacje na plikach specjalnych

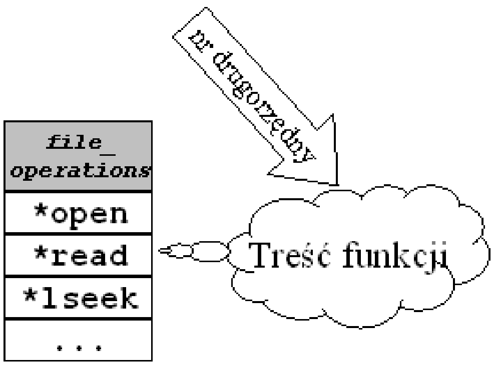
2.1 Parametry funkcji podprogramu obsługi urządzenia

2.1.1 Struktury[8] inode i file jako parametry funkcji

Oczywiście struktura `inode` związana jest z plikiem specjalnym istniejącym fizycznie, a struktura `file` powstaje "na skutek" wykonania przez użytkownika `open` na tymże pliku - jest to wskaźnik wskazujący na pozycję w globalnej tablicy plików. Ogólnie: z `inode` odczytać możemy np. atrybuty pliku, jego wielkość (co jest akurat nieistotne przy plikach specjalnych), a z `file` odczytać możemy tryb otwarcia (do odczytu, wyłączyny itp.) i, najczęściej chyba wykorzystywane, `f_pos` - pozycję w pliku. Szerzej opisano to z pewnością w dokumentach dotyczących systemu plików.

2.1.2 Co z numerem drugorzędnym?

Wiemy już, że typ i numer główny wybiera nam podprogram, ale jak numer drugorzędny wpływa na działanie podprogramu? Jest on po prostu przekazywany jako parametr każdej z funkcji struktury file_operations. Dokonuje się to niejawnie, poprzez pole `r_dev` (typu `kdev_t`) struktury `inode`.



2.2 Wykonanie funkcji

Użytkownik wykonując jedną z funkcji wymienionych w strukturze file_operations, nie powoduje bezpośrednio jej wykonania. Funkcje implementowane w podprogramie obsługi urządzenia działają na innym poziomie abstrakcji, niż funkcje wywoływane przez użytkownika. Wywołanie funkcji `xxx` powoduje uruchomienie funkcji `sys_xxx` z jądra, a ta dopiero (ewentualnie) wywołuje funkcję xxx podprogramu - tak dzieje się w większości przypadków. Przyjżyjmy się jednak bliżej co dzieje się, gdy użytkownik wywołuje poszczególne funkcje - rozpatruję przypadek urządzeń znakowych (chyba, że zaznaczę inaczej) i systemu pilków ext2.

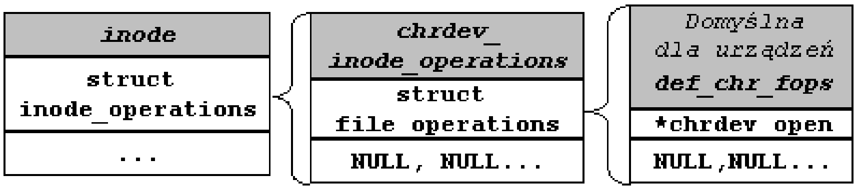
[8] Parametrami funkcji nie są oczywiście struktury, ale wskaźniki do tychże.

2.2.1 open

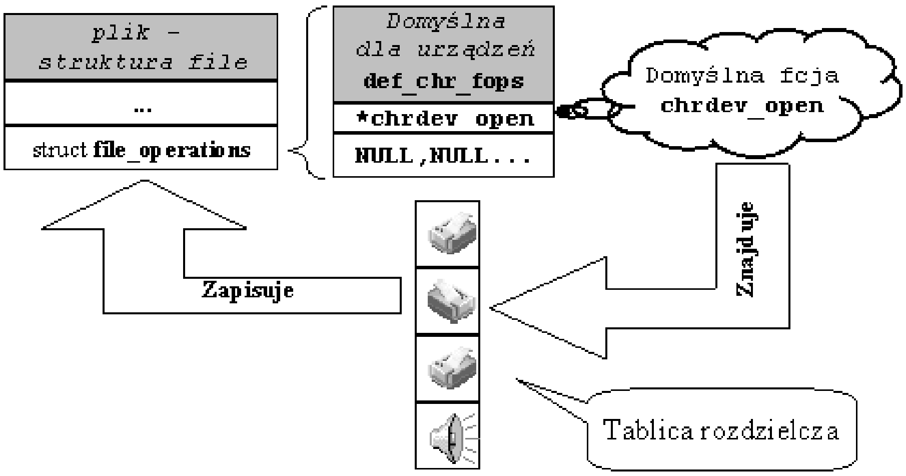
Interfejs na zewnątrz jądra to funkcja

```
int open(const char *pathname, int flags, mode_t mode);
```

Wykonanie jej powoduje wywołanie funkcji `sys_open` (plik `fs/open.c`). Nie wnikając zanadto w szczegóły - stworzony zostaje opis w globalnej tablicy plików (struktura `file`) oraz wczytany (lub znaleziony) i-węzeł pliku. Zostaje wywołana funkcja `open` z pola `default_file_operations` (typu `inode_operations`) i-węzła. Nie jest to funkcja `open` podprogramu! Jest to funkcja `chrdev_open` wspólna dla wszystkich urządzeń znakowych.



Dopiero potem odczytywany jest z tablicy rozdzielczej adres właściwej struktury operacji plikowych (tablica rozdzielcza, jak wiadomo, indeksowana jest numerem głównym, a ten zapisany jest w i-węźle pliku). Znalezione operacje plikowe zapisane zostają w polu `f_op` struktury `file` i dopiero teraz wywoływana jest funkcja `open` podprogramu. Ufff...



Jak widać na powyższym rysunku, po wykonaniu `open` (zakończonym sukcesem) adresy funkcji podprogramu zapisywane są w strukturze `file` - i to z nich system będzie korzystał, gdy użytkownik wykona jedną z funkcji takich jak `read`, `write` czy `close`. Od tego momentu operacje na otwartym pliku specjalnym będą wywołaniami funkcji podprogramu.

Zastanówmy się dlaczego ten schemat wywołań jest tak skomplikowany. Czyż nie prościej byłoby, gdyby od razu wywoływana była funkcja `open` podprogramu, która powinna (?) być zapisana w i-węźle? Przecież w i-węźle znajdują się wszystkie informacje które identyfikują podprogram! I rzeczywiście, być może byłoby to prostsze, a nawet szybsze rozwiązanie, ale w ten sposób pozbylibyśmy sie możliwości instalowania modułów w systemie. W takiej postaci, jak to jest obecnie, gdy podprogram urządzenia nie zostanie znaleziony w tablicy rozdzielczej doczytywany jest odpowiedni moduł[9] - a odbywa się to dopiero po wykonaniu `open` przez użytkownika. Gdybyśmy chcieli mieć wszystkie odwołania do podprogramu od razu w i-węźle musielibyśmy wczytywać wszystkie moduły zaraz po starcie systemu lub po wczytaniu i-węzła do pamięci operacyjnej. Stawiałoby to pod znakiem zapytania sens stosowania modułów.

2.2.2 `release`

Interfejs do tej funkcji na zewnątrz jądra to

```
int close(int fd);
```

Funkcja ta różni się o tyle od pozostałych funkcji, że nie każde wywołanie close z zewnątrz jądra powoduje wywołanie release. Z każdą pozycją w tablicy plików (struktura file) związany jest licznik deskryptorów nań wskazujących. Dopiero, gdy zamykany jest ostatni z deskryptorów wywoływana jest funkcja release. Jest to rozwiązanie o tyle sensowne, że liczba wykonanych open jest równa liczbie wykonanych release (dla systemu z wszystkimi plikami zamkniętymi).

Funkcji release można nie implementować - pomimo to wywołanie close nie będzie kończyło się błędem.

Taka, a nie inna semantyka open i release powoduje, że każdy podprogram musi zliczać na własnym, lokalnym liczniku ile razy plik specjalny został otwarty - wywołanie release nie znaczy, że podprogram obsługi urządzenia nie jest już potrzebny, ale tylko jeden z jego użytkowników "odłącza się". Ponadto takiego licznika nie możemy wiązać ani z deskryptorem pliku (wiele deskryptorów może wskazywać na tą samą pozycję tablicy plików), ani z pozycją w tablicy plików (plik mógł być wiele razy otwarty), ani z i-węzłem (wiele plików może być reprezentacją jednego urządzenia w systemie patrz: 1.2.2).

2.2.3 `lseek`

Wywołujemy

```
off_t lseek(int fildes, off_t offset, int whence);
```

Twórca podprogramu nie chce implementować tej funkcji to... musi ją zaimplementować jako funkcję zawsze zwracającą błąd - jeśli jej po prostu nie zaimplementuje, działać będzie standardowa semantyka funkcji `lseek` zdefiniowana

[9] Oczywiście taki moduł musi istnieć, jądro musi byc skompilowane z możliwością instalacji modułów a program kerneld musi być poprawnie skonfigurowany.

w sys_lseek (fs/read_write.c). Standardowa sys_lseek zmienia pole f_pos odpowiedniej pozycji w tablicy plików, a chcąc odczytać wielkość pliku korzysta z pola i_size zapisaego w i-węźle (co dla plików specjalnych jest bez sensu, ponieważ nie mają one rozmiaru).

2.2.4 read

Użytkownik wywołuje sys_read poprzez

```
ssize_t read(int fd, void *buf, size_t count);
```

sys_read przed wywołaniem funkcji podprogramu wykonuje ważną operację weryfikacji obszaru, który ma być zapisany - po to, by nie pisać do obszaru do którego użytkownik nie ma praw zapisu. Nietrudno bowiem wyobrazić sobie sytuację, że beztroski użytkownik decyduje sie zmazać struktury systemowe wykonując read. Co prawda, nic nie uchroni systemu, jeśli w podprogramie jest błąd (podprogramy działają z uprawnieniami jądra), ale weryfikacja praw zapisu do obszaru zapewnia, że sam podprogram zapisując count bajtów (i ani jednego więcej) może mieć pewność, że nie zmniejsza tym samym stabilności systemu.

2.2.5 write

Z punktu widzenia plików specjalnych jest to pełna analogia do read.

2.2.6 ioctl

Funkcja ioctl służy do sterowania urządzeniami w zakresie, który wykracza ponad to, co mogą wykonywać pozostałe funkcje. Za jej pomocą m.in. możemy formatować dyskietkę, wyjmować dysk cd z napędu, czy pobrać geometrię dysku twardego. Polecenia ioctl nie są ustandaryzowane - każde z urządzeń może mieć inne parametry.

2.2.7 check_media_change, revalidate

Funkcje check_media_change, revalidate dotyczą tylko urządzeń blokowych. Nie mogą być jawnie wywołane przez użytkownika. Są one wywoływane podczas montowania systemu (procedura do_mount), oraz podczas otwierania pliku. Funkcja revalidate wywoływana jest w przypadku stwierdzenia wymiany nośnika (szerszy powinien znaleźć przy urządzeniach blokowych).

Parametrem w/w dwóch funkcji jest zmienna typ kdev_t, która już kilka razy przewinęła się w tym opracowaniu. Definicja, znajdująca się, wraz z towarzyszącymi makrami, w pliku kdev_t.h wygląda następująco:

```
typedef unsigned short kdev_t;
#define MAJOR(dev) ((dev)>>8)
#define MINOR(dev) ((dev) & 0xff)
#define MKDEV(ma,mi) ((ma)<<8 | (mi))
```

Jak widać na tej zmiennej możemy zakodować zarówno numer główny, jak i drugorzędny. Zmienna ta przechowywana jest w polu `i_dev` oraz `i_rdev` i-węzła. Pierwszy wskazuje na jakim urządzeniu zamontowany jest system plików w którym znajduje się dany i-węzeł. `i_rdev` oznacza tyle co "requestet device" i dla plików specjalnych reprezentuje urządzenie stworzone przez `mknod`.

2.2.8 `fsync, async`

`fsync` oraz `async` to również odnoszą się tylko do urządzeń blokowych, a związane są z metodami buforowania (tam też należy szukać informacji na ich temat).

3 Komunikacja z urządzeniem zewnętrznym[10]

Wiadomo już jak odbywa się komunikacja z urządzeniem zewnętrznym, a raczej procedurą jego obsługi, z poziomu użytkownika. Znamy też metody realizacji odwołań do urządzeń z poziomu jądra. Teraz zajmiemy się komunikacją z urządzeniem na poziomie procedury obsługi.

3.1 Porty.

Wszystkie transmisje danych między procesorem lub pamięcią a urządzeniem zewnętrznym odbywają się poprzez porty wejścia-wyjścia. Są to adresy z przestrzeni adresowej jądra (0 - 65536) reprezentujące specjalnym rejestrom odpowiadającym za obsługę urządzeń. Korzystamy z nich za pomocą różnych funkcji in i out (zdefiniowanych w pliku io.h) które odpowiednio odczytują i zapisują odpowiednie wartości w zadanych portach. Dostęp do portów jest też możliwy przez wykonanie read, write i lseek na znakowym pliku specjalnym /dev/port. Znając odpowiedni zestaw portów dla pewnego urządzenia możemy teraz zapisywać lub odczytywać z niego potrzebne dane zapisując bądź odczytując odpowiednie porty. Widać stąd, że komunikacja z urządzeniem zewnętrznym to zestaw zapisów i odczytów portów.

Listę zarezerwowanych portów można zobaczyć w pliku /proc/ioports.

3.2 Przerwania i odpytywanie

Jednak porty i stojące za nimi urządzenia są znacznie wolniejsze niż procesor. Powstaje więc pytanie: Skąd mamy wiedzieć czy zapisany przed chwilą do portu bajt został już wykorzystany na przykład wydrukowany na drukarce? Skąd wiemy, że odczytujemy rzeczywiście kolejny bajt z klawiatury?. Do rozwiązania tego problemu stosuje się dwie techniki: odpytywanie i przerwania.

3.2.1 Odpytywanie

Pierwsza z nich polega na tym, iż procedura obsług i urządzenia w sposób ciągły testuje stan pewnego portu i kontynuuje działanie dopiero wtedy gdy pojawi się na nim odpowiednia wartość. Niejako odpytuje urządzenie czy jest gotowe do dalszej pracy. Ma to swoje wady. Urządzenie jest blokowane na czas

[10] Autorem tego rozdziału jest Piotr Kryska

działania procedury, wykonywanie tylko jednego zadania naraz oraz co chyba najgorsze to blokowanie systemu operacyjnego (aktywne czekanie na możliwość zapisu czy odczytu w czasie operacji wejścia / wyjścia). Zaletą tego rozwiązania jest niewątpliwa prostota rozwiązania a co za tym idzie prostota procedury obsługi, niewielki nakład systemowy na obsługę (nie licząc aktywnego czekania) i nie zajmowanie przerwań. Poza tym w prosty sposób można kontrolować czy urządzenie nadal odpowiada na wysyłane dane (czy nie wystąpił błąd lub minął wyznaczony czas reakcji). Najbardziej istotną jest nieużywanie przerwań gdyż nie wszystkie urządzenia muszą z nich korzystać. Poza tym liczba linii przerwań jest ograniczona, a część z nich jest już zajęta np: 0 czy 1.[11]

Dla przykładu domyślnym sposobem synchronizacji złącza równoległego jest odpytywanie.

3.2.2 Przerwania

Drugą metodą, znacznie częstszą, jest zastosowanie przerwań do komunikacji urządzenia z systemem operacyjnym. Przerwanie to asynchoniczne zdarzenie, które wymaga od procesora podjęcia natychmiastowej akcji. Dzięki zastosowaniu przerwania proces czekający na obsługę żądania wejścia/wyjścia może zostać uśpiony a urządzenie poinformuje procedurę obsługi o konieczności podjęcia pewnej akcji na przykład odczytania kolejnego znaku, czy rozpoczęcie nowej transmisji przerwaniem. Aby tego dokonać, podprogram musi zarejestrować w systemie własną procedurę obsługi przerwania. W tym celu do systemowej tablicy `struct irqaction *irq_action[16]` wstawiana jest odpowiednia struktura:

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

W strukturze tej `handler` oznacza właściwą funkcję obsługi przerwania a pozostałe to flagi określające tryb pracy bądź nazwę (używaną tylko przez system plików proc)czy dodatkowe dane przesyłane dla funkcji obsługi po jej wywołaniu. Wfunkcji `handler` powinno się wykonać wszystkie działania potrzebne do kontynuowania pracy systemu i urządzenia a proces, który śpi w oczekiwaniu na przerwanie może zostać obudzony.

Procedury obsługi przerwań moga pracować w dwóch podstawowych trybach. Przerwania "szybkie" pracują z wyłączonymi przerwaniami dzięki czemu wiemy, że procedura nie zostanie przerwana. Przerwania "wolne" działają przy włączonych przerwaniach co pozwala na ich przerywanie innymi przerwaniami. W Linuxie do obsługi przerwań służą funkcje `do_IRQ(...)` dla przerwań wolnych i `do_fast_IRQ(...)` dla szybkich. Ograniczeniem stosowania przerwań

[11] Straciło to swoje znaczenie w Linuksie 1.3.70, gdyż wprowadzono w nim współdzielenie przerwań, co zniosło to ograniczenie.

do synchronizacji jest wymaganie aby urządzenie było zdolne do wywołania przerwania. Zwykle do komunikacji z urządzeniami używa się przerwań szybkich.

3.2.3 Współdzielenie przerwań.

Niestety, liczba dostępnych linii przerwań na komputerach klasy PC jest ograniczona do 15. Pozwoliłoby to na obsługę tylko 15 urządzeń dlatego też w Linux-ie 1.3.70 wprowadzono współdzielenie przerwań. Dzięki niemu na jednej linii przerwań możemy zainstalować więcej niż jedną procedurę obsługi. Ich rejestracja powoduje wrzucenie kolejnej funkcji obsługi do kolejki. Tutaj właśnie wykorzystuje się pole `next` ze struktury irqaction. Funkcja `request_irq(...)` gdy stwierdzi, że w danym miejscu jest zarejestrowana funkcja wstaw nową na koniec kolejki. Po nastąpieniu przerwania wywoływana jest funkcja `do_IRQ(int irq; struct pt_regs *)` bądź `do_fast_IRQ(...)`, które po kolei wywołują wszystkie fukncje z kolejki dla numeru przerwania `irq`. Funkcja obsługi musi sprawdzić czy przerwanie dotyczy akurat jej i ewentualnie podjąć potrzebne działania. Najczęściej odbywa się to poprzez czytanie odpowiedniego portu.

Uwaga! Jednej linii przerwań nie mogą współdzielić przerwania szybkie i wolne ze względu na to, iż ich obsługą zajmują się inne funkcje.

Listę zarejestrowanych przerwań można obejrzeć w pliku /proc/interrupts.

3.2.4 Dolne połowy

Z obsługa przerwań bezpośrednio wiąże się pojęcie "dolnych połów". Przy obsłudze przerwania czasem ważne jest by część procedury obsługi była wykonana natychmiast z zamaskowanymi przerwaniami, a część czasem bardzo długa może zostac wykonana z odmaskowanymi np: kopiowanie danych do bufora w pamięci, czy uaktualnianie stanu diód na klawiaturze. Poza tym druga część może zostać wykonana później po wykonaniu innych przerwań. Jest to istotne o tyle, że zapobiega zgubieniu przerwania przy zbyt długim czasie pracy w stanie wyłączonych przerwań i jednocześnie jakimś błędom spowodowanym przez przerwanie procedury obsługi przerwania przez inne przerwanie. Mechanizm ten w Linux-ie jest realizowany przez tak zwane "dolne połowy". W systemie istnieje tablica `void (*bh_base[32])(void)` wskaźników do różnych funkcji dolnych połów. W trakcie obsługi przerwania (tzw. górna połowa) zaznacza pewne dolne połowy jako przeznaczone do wykonania przy pomocy funkcji `mark_bh(int nr)`, po czym może robić cokolwiek: sprawdzić, poprawić struktury danych itd. Przy najbliższej okazji tzn. powrót z funkcji systemowej czy funkcja `schedule()` wszystkie zaznaczone dolne połowy zostaną wykonane. Trzeba oczywiście pamiętać aby dolną połowę do tej tablicy wstawić funkcją `void init_bh(int nr, void (*routine)(void));` ale robi się to zwykle przy zajmowaniu przerwania w funkcji inicjującej urządzenie.

Warto wiedzieć, że 12 początkowych pozycji w tej tablicy można używać przy pomocy typu wyliczeniowego, który obejmuje na przykład: `TIMER_BH, CONSOLE_BH` czy `SERIAL_BH.` Są to też dolne połowy obsługiwane przez system. Jednak w Linux-ie wersji 2.x większość tych funkcji uruchamia odpowiednie kolejki zadań (choć nie jest to regułą).

3.2.5 Kolejki zadań

Z dolnymi połowami, podobnie jak z przerwaniami, mamy problem z ograniczeniem ilości dolnych połów. Normalnie w Linux-ie wersji 1.x wolnych było 20 pozycji w tablicy `bh_base`. Aby ten problem rozwiązać w nowszych wersjach zastosowano kolejki zadań (task queues). W tablicy bh_base można umieścić funkcję, która wykona `run_task_queue(...)` (np: `immediate_bh()` czy `cyclades_bh()`) czyli spowoduje wykonanie wszystkich funkcji z kolejki. Alternatywna metoda polecana zresztą przez twórców polega na tym, iż funkcję obsługi kolejki umieszcza się w kolejce TQ_IMMEDIATE, która ma implementować i zastępować stary system obsługi dolnych połów. Na kolejki te można wstawiać dowolne funkcje umieszczając je najpierw w strukturze tq_struct

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

Struktura ta zawiera: wskaźnik do funkcji przeznaczonej do wykonania (routine), wskaźnik do opcjonalnych danych dla tej funkcji (data), a także flagę synchronizacji (sync), która zapobiega wstawianiu tej samej struktury kilka razy. Co ciekawe funkcja będąca składnikiem struktury tq_struct może ją wstawić na jakąś kolejkę co pozwala kilkakrotnie wykonać daną funkcję. Poza tym dana struktura wstawiana jest zawsze na początek kolejki więc nie można w ten sposób wykonać jej dwa razy. Zaznaczanie odbywa się tak jak poprzednio, lecz teraz w odpowiednim miejscu wykonana zostanie cała kolejka dolnych połów. W przypadku stosowania kolejki `TQ_IMMEDIATE` do uruchamiania zdefiniowanej kolejki zadajń, funkcja obsługująca kolejkę zadań wykonywana jest przy wykonywaniu kolejki `TQ_IMMEDIATE`. Dzieje się to zawsze tam gdzie wykonywana jest obsługa dolnych połów (patrz wyżej). Oprócz tych rozwiązań możliwe jest wykonanie kolejki zadań w dowolnym miejscu funkcją `run_task_queue(...)` choć powinno się unikać tego w procedurach obsługi przerwań gdyż po to właśnie są te kolejki by odciążyć przerwania.

3.3 DMA (Direct Memory Acces)

Kolejnym istotnym aspektem komunikacji jest DMA. Jak wiadomo kontrolery DMA pozwalają na transmisję danych z pamięci i do pamięci bez udziału procesora. Zajmują natomiast szynę danych co przy dobrym wykorzystaniu pamięci podręcznej nie powoduje spowolnienia pracy. W architekturze i_86 obsługiwane przez Linuxa są dwa kontrolery z czterema kanałami każdy. Pierwszy kontroler wykonuje transmisje po bajcie drugi po słowie. Jeden z kanałów pierwszego przeznaczony jest na obsługę drugiego stąd mamy efektywnie 7 kanałów DMA. Jednak i tutaj napotykamy pewne problemy. Kontrolery wymagają podania fizycznych adresów w pamięci co może być sprzeczne z zasadami ochrony; wymagają aby pamięć przeznaczona na transmisję znajdowała się poniżej 16M w przestrzeni adresowej. Poza tym istniej duże ograniczenie ilości przesyłanych danych (64 lub 128k w zależności od stosowanego kontrolera 0 czy 1). Aby poradzić sobie z tą sytuacją w Linux-ie wprowadzono kilka ciekawych rozwiązań. Po pierwsze jądro zawsze ma w przestrzeni poniżej 16M zarezerwowane miejsce na pamięć dla transmisji DMA. Wprowadzono specjalny sposób przydzielania pamięci na bufory DMA: jądro dba by obszar ten był spójny, znajdował się w pamięci dostępnej dla kontrolerów i miał odpowiednią wielkość (dla drugiego kontrolera ważne jest poza tym aby startowy adres był parzysty). Dzięki temu, iż kontrolery DMA można połączyć z przerwaniami można poradzić sobie z ograniczeniem wielkości bufora danych. Po prostu w procedurze obsługi przerwania przygotowujemy i inicjujemy kolejną transmisję. Pewną wadą obsługi DMA w Linuxie jest to, że bufory DMA muszą być kopiowane do i z przestrzeni adresowej procesu aby zapewnić pełną ochronę pamięci. Pewne problemy stwarza też przekazywanie danych do kontrolera jednak w Linux-ie dostępne są funkcje, które to ułatwiają:

- `request_dma(int dmanr, char *);` zajmuje kanał DMA o numerze dmanr. Drugi argument jest używany przez system plików proc.
- `free_dma(int dmanr);` zwalnia kanał DMA.
- `disable_dma(int dmanr);` wyłącza kanał.
- `enable_dma(int dmanr);` włącza kanał.
- `set_dma_addr(int dmanr, unsigned int a);` ustawia adres bufora dla transmisji.
- `set_dma_count(int dmanr, unsigned int count);` ustawia długo ść transmitowanych danych.)
- `set_dma_mode(int dmanr, char mode);` ustawia tryb transmisji danych. Linux jak na razie obsługuje tylko trzy: pojedyńczy odczyt, zapis i kaskadowanie na drugi kontroler.
- `clear_dma_ff(int dmanr);` używane do komunikacji 16 bitowych rejestrów procesora z 8 bitowymi rejestrami kontrolerów.

Poza tym dostęp do wszystkich rejestrów DMA realizowany jest poprzez makra zdefiniowane w pliku dma.h.

Zajęte kanały DMA można obejrzeć w pliku /proc/dma.

4 Urządzenia blokowe[12]

4.1 Charakterystyka

Najważniejszą cechą urządzeń blokowych jest możliwość swobodnego dostępu do urządzenia, tzn. dostępu do danych w dowolnym wskazanym miejscu. Odczytywane są całe bloki z urządzenia, a dokonywane jest to z wykorzystaniem podręcznej pamięci buforowej. Dostęp blokowy i znakowy to dwa podstawowe sposoby dostępu do urządzeń w Linuxie.

[12] Autorem tego rozdziału jest Krzysztof Socki

4.2 Podręczna pamięć buforowa

Krótko wspomnę o podręcznej pamięci buforowej, szczegółowo temat ten zostanie opracowany przy omawianiu systemu plików. Bufory to struktury jądra reprezentujące pojedyncze bloki urządzenia (np. bloki dyskowe). Są opisane przez strukturę `buffer_head`, która zawiera informację o tym jaki blok jest w buforze, o jego stanie, liczbie procesów korzystających z bufora, czasie jego zapisu na urządzenie, czasie ostatniego dostepu do bufora, rozmiarze oraz o dowiązaniach do struktur w których bufory są przechowywane. Bufory mogą mieć różne wielkości, tak jak różne mogą być wielkości bloków urządzeń. Jeden blok może być odwzorowany na conajwyżej jeden bufor. Każdy bufor jest jednoznacznie identyfikowany przez numer urządzenia i numer bloku na nim. Działanie podręcznej pamięci buforowej jest dla użytkownika całkowicie niewidoczne, tzn. może nawet nie zdawać sobie sprawy z jej istnienia. Przyspiesza ona wykonanie operacji na urządzeniach, ponieważ np. żądanie odczytu bloku który akurat znajduje się w buforze nie wymaga dostępu do dysku. Podobnie z żądaniami zapisu. Aby zapewnić szybkie wykonanie operacji na buforach są one zorganizowane w tablicy haszującej `hash_table`, zawierającej listy wszystkich buforów w systemie. Tablica ta ma `NR_HASH` pozycji, a przypisanie bufora do odpowiedniej listy odbywa się na podstawie numeru urządzenia i numeru bloku, czyli atrybutów unikalnych dla bufora. Poza tym bufory są jeszcze połączone w niezależne zestawy list:

- tablica list buforów wolnych `free_list` (tablica o tylu pozycjach co ilość rozmiarów buforów w systemie - dla jądra 2.0.32 to 5),
- tablica list buforów zawierających dane `lru_list` (tablica o tylu pozycjach co ilość rodzajów zajętych buforów - dla jądra 2.0.32 to 4).

Jeden bufor nie może jednocześnie znajdować się w obydwu tych zestawach choćby dlatego, że wykorzystują one te same atrybuty nagłówka bufora.

4.3 Dostęp do urządzeń blokowych

Wszystkie urządzenia blokowe korzystają z tych samych funkcji do odczytu i zapisu danych: `block_read` i `block_write.` Na poziomie tablicy rozdzielczej urządzeń blokowych nie ma funkcji specyficznych dla jednego rodzaju urządzenia. To właśnie funkcje `block_*` korzystają bezpośrednio z podręcznej pamięci buforowej.

Funkcja `block_read` otrzymuje jako parametry i-węzeł, bufor na dane w przestrzeni adresowej użytkownika, strukturę `file` pliku specjalnego którego dotyczy i ilość bajtów do przeczytania. Funkcja ta pobiera z i-węzła numer główny urządzenia, ustala wielkość bloku, ilość bloków do odczytania i numer pierwszego bloku. Następnie pobiera bufory dla kolejnych bloków których dotyczy polecenie wczytania i jeśli nie zawierają aktualych danych (jest to rozstrzygane na podstawie atrybutów struktury `buffer_head`) to zgłasza żądanie wczytania brakujących buforów (funkcja `ll_rw_block` opisana dalej). Na koniec kopiuje zawartość buforów do bufora użytkownika (`memcpy_tofs`).

Funkcja `block_write` otrzymuje takie same parametry jak `block_read` tylko, że bufor użytkownika zawiera dane do zapisania. Najpierw działa analogicznie jak `block_read`, tzn. pobiera i weryfikuje wszystkie potrzebne jej wartości, następnie wczytuje te bufory których dotyczy polecenie (wywołanie `ll_rw_block`) i modyfikuje dane w nich (`memcpy_fromfs`). Następnie zaznacza zmodyfikowane bufory jako przeznaczone do zapisu (atrybut `buffer_head`) i zgłasza żądanie ich zapisania (`ll_rw_block`).

Funkcje `block_read` i `block_write` niczego tak naprawde nie zapisują, tylko zlecają wykonanie odpowiednich operacji przez wywołanie funkcji `ll_rw_block`.

W systemie znajduje się tablica `all_requests` która zawiera dowiązania do wszystkich żądań do urządzeń blokowych. Żądanie jest reprezentowane przez strukturę `request`, która między innymi zawiera adres bloku którego dotyczy. Tablica `all_requests` ma `NR_REQUEST` = 64 pozycje. Poza tym żądania są zgrupowane w kolejki dla poszczególnych urządzeń (tzn. numery główne urządzeń których dotyczą żądania w ramach pojedynczej takiej kolejki są takie same).

Funkcja `ll_rw_block` weryfikuje istnienie urządzenia, wielkość bloków i ewentualną dostępność do pisania, następnie wywołuje funkcję `make_request` dla każdego bufora którego żądanie dotyczy. Ta funkcja weryfikuje wielkość urządzenia (czy żądany blok nie jest gdzieś poza urządzeniem) i celowość utworzenia żądania (np. nie tworzy żądania odczytu dla bufora zawierającego aktualne dane). Następnie przechodzi listę już zgłoszonych żądań dla danego urządzenia i sprawdza czy któreś z nich nie dotyczy bloków leżących fizycznie przy bloku który nas interesuje. W razie znalezienia takiego żądania rozszerza jego zakres (tzn. będzie ono polegało na wczytaniu większej ilości bloków). W przeciwnym razie tworzy nowe żądanie i dowiązuje do niego odpowiednie wskaźniki w tablicy `all_request`. Jeśli w tej tablicy nie ma wolnego miejsca to zawiesza się na kolejce `wait_for_request`. Co ważne: w poszukiwaniu miejsca dla żądań odczytu przeszukiwana jest cała tablica `all_request`, natomiast dla żądań zapisu - tylko jej początkowe 2/3. W ten sposób żądania odczytu zyskują priorytet.

Na koniec `make_request` wywołuje funkcję `add_request`, która wstawia żądanie w odpowiednie miejsce na liście żądań dla danego urządzenia i jeśli było to pierwsze żądanie na tej liście to wywołuje funkcję strategii urządzenia, tzn. inicjalizuje niskopoziomową obsługę odczytu lub zapisu na urządzenie.

4.4 Szeregowanie

Za szeregowanie żądań do urządzenia odpowiedzialna jest funkcja `add_request`, a realizuje je przez wstawienie żądania w odpowiednie miejsce w kolejce żądań dla danego urządzenia. Procedura strategi po uruchomieniu pobiera żądania z tej kolejki tak, jak są na niej ułożone i jakiekolwiek szeregowanie może odbywać się tylko przez odpowiednie ułożenie na tej kolejce. W jądrze 2.0.32 funkcja `add_request` umieszcza żądania w kolejce tak, że są posortowane według wzrastającego numeru drugorzędnego urządzenia, którego dotyczą, a następnie według pierwszego bloku którego dotyczą (jest to algorytm C-LOOK). Przegląda ona listę w pętli i znajduje odpowiednie miejsce wykorzystując makro `IN_ORDER` zdefiniowane w pliku `blk.h`:

```
#define IN_ORDER (s1,s2) ((s1->rq_dev < (s2)->rq_dev ||
(((s1)->rq_dev==(s2)->rq_dev && (s1)->sector<(s2)->sector)) )
```

Zaimplementowany sposób może doprowadzić do zagłodzenia w przypadku, gdy stale przychodzą żądania do jednego urządzenia lub bieżącego bloku, ponieważ nie jest spełniony warunek sprawiedliwości - nowoprzybyłe żądanie zostanie wstawione przed wszystkimi już wstawionymi żądaniami, jeśli tylko dotyczy urządzenia lub bloku o niższym numerze. Żadne żądanie nie może być wstawione przed żądanie aktualnie realizowane - w takim przypadku żądanie jest wstawiane na koniec kolejki.

Każde urządzenie blokowe jest opisane przez strukturę:

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

gdzie `request_fn` jest wspomnianą już funkcją strategii urządzenia, a `curret_request` wskaźnikiem na aktualnie realizowane żądanie. Istnieje możliwość wstrzymania realizacji żądań skierowanych do urządzenia. Zaimplementowana jest funkcja `plug_device` która powoduje wstawienie żądania `plug` na pierwsze miejsce w kolejce żądań dla urządzenia. Jest to specjalne żądanie, którego jedynym zadaniem jest powstrzymanie funkcji strategii przed realizacją następnych żądań, a więc po wywołaniu `plug_device` urządzenie praktycznie nie będzie działać (będzie zakorkowane, ang. plug - korek). Aby ponownie umożliwić działanie trzeba wywołać funkcję `unplug_device`, która usuwa żądanie `plug` z kolejki i jeśli nie jest pusta, to wywołuje funkcję strategii.

4.5 Funkcja `fsync` strutktury `file_operations`

Opiszę jeszcze kilka pól struktury `file_operations` (która składa się na strukturę `device_struct` zawartą w tablicach rozdzielczych urządzeń) charakterystycznych dla urządzeń blokowych. Są to:
`int (*fsync)(struct file *, struct dentry *)` - zdefiniowana w pliku `fs/block_dev.c`. Zapisuje i zwalnia bufory przypisane do urządzenia. Jest wywoływana przy demontowaniu urządzenia oraz przez funkcję systemową `sync`, a ta funkcja jest co jakiś czas wywoływana przez demona `cron`, np. do zapisywania buforów dysku twardego.

5 Urządzenia znakowe[13]

5.1 Terminal

Jednym z najważniejszych urządzeń znakowych jest terminal, którym w szczególności może być interfejs do ekranu i klawiatury. Jądro Linuxa zajmuje się buforowaniem i podstawową edycją tekstu wprowadzanego z klawiatury oraz przetwarzaniem tekstu wysyłanego przez proces na ekran. Oprócz wysyłania na terminal i czytania z niego znaków, parametry terminala można odczytywać i zmieniać różnorodnymi `ioctl()`ami.

[13] Autorem tego rozdziału jest Marcin Kowalczyk

Terminal ma przyporządkowaną grupę procesów („chodzących na pierwszym planie” na tym terminalu). Do tych właśnie procesów zostanie wysłany SIGINT, kiedy użytkownik naciśnie ^C, i SIGSTOP przy ^Z. Kiedy lider grupy procesów związanej z terminalem (najczęściej jest tologin shell) kończy działanie, pozostałe procesy tej grupy otrzymują SIGHUP, co domyślnie powoduje zakończenie procesu.

Proces nienależący do grupy związanej z terminalem dostaje SIGTTIN przy próbie odczytu z tego terminala i SIGTTOU przy zapisie. Może jednak ignorować te sygnały - wtedy uda mu się coś wypisać na ekranie, ale odczyt zwróci błąd EIO.

Procesy grupy terminala dostają SIGWINCH w razie zmiany rozmiaru okna. Jest to prośba o ewentualne przerysowanie się, dostosowując się donowego rozmiaru okna.

5.2 Wirtualne konsole

Wśród standardowych terminali ważne są wirtualne konsole. Jedna z nich może być w danej chwili aktywna - jej zawartość jest pokazywana na ekranie i ona odbiera naciskane klawisze. Numer aktywnej wirtualnej konsoli jest pamiętany w zmiennej `fg_console`.

Wirtualnym konsolom odpowiadają pliki specjalne `/dev/tty1`, `/dev/tty2`, `/dev/tty3` itd. Pliki te są z reguły otwierane jako standardowe wejście, wyjście i wyjście błędów przez program `getty` i są dostępne dla procesów-dzieci pod tymi standardowymi strumieniami do czasu przekierowania tych strumieni na coś innego.

`/dev/tty` jest aliasem na konsolę przyporządkowaną danemu procesowi.

Natomiast `/dev/console` to konsola, która jest właśnie aktywna. Funkcja `tty_open()` (do której wskaźnik jest umieszczony w tablicy rozdzielczej dla terminala) sprawdza, który plik specjalny otwieramy: w szczególności jeśli jest nim `/dev/tty`, to otwierany jest bieżący terminal danego procesu, a jeśli `/dev/console`, to aktywna konsola. Wynika z tego, że `/dev/console` to konsola aktywna wmomencie wykonania wykonania `open()` i odnosi się do tej samej konsoli aż do zamknięcia pliku, nawet jeśli w międzyczasie użytkownik przełączy się na inną. Można się też dostać do bezpośredniej zawartości ekranu poszczególnych wirtualnych konsoli.

Pliki specjalne `/dev/vcs1`, `/dev/vcs2` itd. obrazują same znaki, bez atrybutów, natomiast `/dev/vcsa1`, `/dev/vcsa2` itd. to znaki z atrybutami, w takim układzie jak w pamięci karty graficznej na pececie: znak, jego atrybut, następny znak, jego atrybut itd. `/dev/vcs0, /dev/vcsa0` to aliasy na aktywną konsolę. Pliki te można czytać i można do nich pisać; pisanie do `/dev/vcs[0-9]*` pozostawia atrybuty w danym miejscu ekranu bez zmian. Skorzystanie `/dev/vcsa[0-9]*` pozwala zachować zawartość ekranu dla późniejszego jej dokładnego odtworzenia. Nie należy jednak oczekiwać odczytania zrozumiałych numerów znaków, zwłaszcza spoza ASCII, ponieważ są tutaj używane numery znaków w foncie, które nie muszą mieć nic wspólnego z numerami znaków używanymi przez programy; szczegóły na ten temat będą niżej.

`/dev/stdin`, `/dev/stdout` i `/dev/stderr` są po prostu aliasami na deskryptory 0, 1, 2 (z reguły są to symlinki do `/proc/self/fd/[012]`) i nie muszą odnosić się do żadnego terminala, jeśli odpowiednie deskryptory zostały przekierowane.

5.3 Dyscyplina linii

Dyscyplina linii to moduł, który pośredniczy między procesem wykonującym operacje na terminalu a podprogramem niższego poziomu. Funkcja odczytu z terminala (`tty_read()` w tablicy rozdzielczej) po wstępnym sprawdzeniu poprawności stanu terminala przekazuje żądanie odczytu funkcji odczytu dyscypliny linii tego terminala. Analogicznie robi funkcja zapisu (`tty_write()`), z tym że tutaj dodatkowo blok przeznaczony do zapisania jest dzielony na mniejsze kawałki, żeby zapewnić stabilność i płynność funkcjonowania systemu w razie żądania wyświetlenia bardzo dużego bloku. Można ładować własne moduły dyscypliny linii.

Standardowy moduł nazywa się `N_TTY` i zajmuje się m.in.:

- echem wejścia na ekran
- buforowaniem wejścia po wierszu - wczytane z klawiatury znaki są przekazywane procesowi dopiero po naciśnięciu Enter
- obsługą podstawowych klawiszy edycyjnych dotyczących wczytywanego wiersza, pozwalających kasować ostatnio wprowadzone znaki i wstawiać znaki kontrolne, oraz klawiszy wysyłających sygnały do procesów grupy danego terminala, m.in. `^C`, `^\`, `^?`, `^U`, `^D`, `^Z`, `^W`, `^V` (takie są standardowo -można przedefiniować)
- wyświetlaniem tabulacji jako odpowiedniej liczby spacji (do najbliższej wielokrotności 8 znaków)
- obsługą XON/XOFF (`^S`, `^Q`)
- tłumaczeniem CR i LF kończących linie na wejściu i wyjściu nawszelkie możliwe sposoby – różne termimale stosują różne konwencje
- dostosowywaniem działania do bardziej ułomnych terminali, którenp. nie mogą kasować raz wypisanego tekstu, nie mają małych liter itp.

Rozmaite parametry tego zachowania, w szczególności włączanie i wyłączanie wykonywania poszczególnych czynności, można konfigurować `ioctl()`em (`TCSETS`; `struct termios`). Dyscyplinę linii można też przestawić w tryb surowy, w którym przepuszcza ona znaki bez ingerencji.

Dyscypliny linii instalują swojego rodzaju tablicę rozdzielczą (`struct tty_ldisc`), zawierającej wskaźniki na funkcje implementujące inicjalizację, odczyt, zapis, opróżnienie buforów, ładowanie `termios`a itp.

5.4 Pseudoterminale

Zamiast na fizycznej konsoli całkowicie obsługiwanej przez jądro, proces może działać na wirtualnym terminalu, którego wejście i wyjście jest kontrolowane przez inny proces. Tak są zrobione m.in. *xterm*, *telnet*, *screen*.

Proces kontrolujący (master) otwiera `/dev/ptyp*`. Proces kontrolowany (slave) otwiera `/dev/ttyp*`, tzn. z reguły dostaje go jako przekierowane *stdin*, *stdout* i *stderr*, niekoniecznie będąc świadomym, na jakiego rodzaju terminalu działa. To, co master wysyła na swoje urządzenie, slave dostaje na wejściu swojego, i odwrotnie. Dane są buforowane, a kiedy jeden z procesów nie nadąża z przetwarzaniem swojej części, drugi jest tymczasowo blokowany - podobnie jak przy komunikacji przez pipe.

Pseudoterminal ma własną dyscyplinę linii, dzięki czemu master dostaje dane już przetworzone przez nią i może zająć się samym wyświetlaniem znaków oraz dostarczaniem emulowanej klawiatury (w szczególności jeślichce wyświetlić przefiltrowane wyjście na prawdziwym terminalu, to powinien przestawić go w tryb surowy, i analogicznie dla wejścia). Np. echo wpisywanych z klawiatury znaków master dostaje razem z **wyjściem** produkowanym przez proces.

5.5 Mapowanie klawiatury

Przetwarzanie wejścia i wyjścia terminala następuje nie tylko w dyscyplinie linii. Klawiatura generuje sprzętowe przerwania przy każdym naciśnięciu i zwolnieniu klawisza oraz przy przytrzymywaniu klawisza wystarczająco długo, żeby zadziałało automatyczne powtarzanie. W procedurze obsługi przerwania na porcie 60h jest widoczny kod klawisza (przy zwolnieniu - z ustawionym najstarszym bitem).

Niektóre klawisze wysyłają kody złożonez kilku bajtów, generując kilka przerwań klawiatury. Stan Shiftów, CapsLocka itp. nie jest na tym etapie uwzględniany (z wyjątkiem anomalii związanych z niektórymi klawiszami). Na klawiaturze 83-klawiszowej kody były bardziej regularne, a na 101-klawiszowej, dla zapewnienia kompatybilności, bywa to bardziej skomplikowane. Skankody są związane z położeniem klawisza na klawiaturze i nie mają nic wspólnego z ASCII.

Takie kody klawiszy są wstępnie tłumaczone na tzw. *keycodes*, które są bardziej regularne - jeden kod odpowiadajednemu klawiszowi i wygładzone są anomalie. Następnie te kody są przetwarzane na podstawie załadowanej mapy klawiatury. Mapa klawiaturydefiniuje znaczenia każdego *keycode* przy poszczególnych stanach Shifta, Ctrla itp. Klawisze i ich kombinacje są zamieniane na ciągi znaków ASCII (wielobajtowe kody klawiszy specjalnych najczęściej zaczynają się od *ESC*), które są następnie pobierane przez dyscyplinę linii. Niektóre klawisze mają specjalne znaczenie, np. zmianę aktywnej wirtualnej konsoli albo przewinięcie ekranu (Shift-PgUp/PgDn). Interpretowane jest wpisywanie numeru znaku na klawiaturze numerycznej przy wciśniętym Alcie.

Wszystko konfiguruje się `ioctl()`ami. Proces może zażądać pomijania ostatniej fazy tłumaczenia (w praktyce jest to rzadko stosowane) albo nawet obu faz (tryb `VC_RAW`, używany np. w X-Window). Taki stan terminala nie nadaje się do normalnej pracy bez niebanalnego tłumaczenia klawiszy przez proces.

5.6 Mapowanie ekranu

Podobnie znaki wysyłane na konsolę są jeszcze interpretowane poprzejściu przez dyscyplinę linii, która w tym kontekście zajmuje siętylko tabulacjami i końcami linii. Rozmaite sekwencje sterujące pozwalają na standardowej konsoli zmieniać i odczytywać położenie kursora, zmieniać atrybuty, kasować fragmenty ekranu, zmieniać trybypracy, zmieniać paletę kolorów, zmieniać częstotliwość i długość standardowego sygnału dźwiękowego (*beep*), zapamiętywać i odtwarzać te parametry itp. Sekwencje te są wypisane w `man console_codes`. Kiedy proces działa na pseudoterminalu, zamiast jądra za całe to przetwarzanie jest odpowiedzialny proces kontrolujący.

Konsola może być w trybie „bajtowym" albo w trybie UTF-8. W pierwszym trybie znaki do wyświetlenia są wstępnie tłumaczone na Unikod, na podstawie jednej z czterech tablic: ISO-8859-1, grafika VT-100, CP-437 albo mapa załadowana przez użytkownika. Tablice te wybiera się przez przestawienie jednego z zestawów nazwanych G0 i G1 na odpowiednią tablicę i uaktywnienie G0 bądź G1. Jest to takie skomplikowane raczej z przyczyn historycznych, dla kompatybilności z innymi terminalami - dałoby się to zaprojektować prościej. W trybie UTF-8 znaki są od razu podane w Unikodzie, zapisane jako UTF-8. W obu przypadkach znaki są następnie tłumaczone na pozycje w foncie - font powinien mieć zapisane, które rysunki znaków odpowiadają którym znakom w Unikodzie. Dzięki temu zestaw znaków używany przez programy może być niezależny od kolejności znaków w foncie, przez co możemy np. uzyskać na ekranie ramki, których nie ma w ISO-8859-2, a z powodu sprzętowej cechy karty VGA muszą być umieszczone fizycznie w pozycjach, które są w ISO-8859-2 zajęte przez inne znaki. W trybie UTF-8 można uzyskać bardzo dużo różnych znaków; font może fizycznie zawierać do 512 znaków, a pewne rysunki znaków mogą pełnić funkcje kilku znaków podobnie albo tak samo wyglądających.

5.7 GPM

Jądro oferuje funkcje ułatwiające implementację programu GPM, który pozwala kopiować fragmenty ekranu myszą i potem wstawiać je tak, jakby były wpisane z klawiatury. Oprócz wyświetlania wskaźnika myszy i zaznaczonego obszaru (przez zamianę kolorów tektstu i tła), samo kopiowanie obszaru, łącznie z wyborem pełnych słów i linii przy podwójnym albo potrójnym kliknięciu, jest realizowane przez jądro. Interfejs jest w postaci `ioctl()`.

Kody znaków są tłumaczone z numerów pozycji w foncie na załadowany zestaw używany przez aplikacje, co nie zawsze jest dokładne i nie zawsze jest nawet możliwe, jeśli zbiór znaków w foncie nie odpowiada dokładnie zbiorowi znaków w zestawie aplikacji (kopiowane znaki mogły zostać wyświetlone przy innym kodowaniu niż bieżące); w szczególności będą przekłamania, jeśli wiele znaków uwspólnia ten sam rysunek znaku w foncie, co skądinąd bywa przydatne. Wstawianie nie jest też zaimplementowane poprawnie dla trybu UTF-8. Dokładniejsze kopiowanie wymagałoby zmiany sposobu pamiętania zawartości ekranu - musiałby być pamiętamy w Unikodzie.

5.8 Urządzenie mem[14]

W Linux-ie pliki specjalne używane są nie tylko jako reprezentacja urządzeń zewnętrznych. Niektóre z nich reprezentują także urządzenia logiczne. Przykładem może być urządzenie znakowe o numerze głównym 1. Reprezentuje ono zbiór urządzeń połączonych z pamięcią. Są to:

- `/dev/mem` fizyczna pamięć
- `/dev/kmem` wirtualna pamięć jądra
- `/dev/random` generator liczb losowych (działa blokująco gdy system nie ma wystarczającego poziomu entropii)

[14] Autorem tego podrozdziału jest Piotr Kryska

- `/dev/urandom` jw. nieblokujący
- `/dev/null` zawsze puste urządzenie; można pisać bez ograniczeń; czarna dziura.
- `/dev/zero` generator zer
- `/dev/port` reprezentuje porty
- `/dev/full` zawsze pełne urządzenie przy zapisie zwraca `ENOSPACE`

Urządzenie to jest niezwykłe, także ze względu na sposób inicjalizacji. Nie posiada ono oddzielnej funkcji inicjalizującej. Inicjalizowane jest bezpośrednio w funkcji `chr_dev_init`. Poza tym posiada nietypowy sposób wyboru struktury `file_operations`. W tablicy rozdzielczej znajduje się tylko wskaźnik do funkcji `open()`. Odpowiednie struktury wybiera się dopiero przy pierwszym jej wywołaniu. Funkcja `open()` przy pomocy makra `MINOR` sprawdza numer drugorzędny otwieranego pliku i w zależności od niego do deskryptora pliku wstawia odpowiednią strukturę `file_operations`.

6 Jak pisać programy obsługi urządzeń, czyli kilka wiadomości dla programisty[15]

6.1 Jak wykryć sprzęt.

6.1.1 Wprowadzenie.

Wykrywanie sprzętu jest ważną częścią pisania programów obsługi dla urządzeń, które wykorzystują do swego działania porty wejścia/wyjścia, przerwania lub kanały DMA. To właśnie czynność wykrywania jakich zasobów należy użyć do komunikacji z urządzeniem nazywa się wykrywaniem sprzętu. Oczywiście możnaby w czasie kompilacji ustalić zakresy zasobów z jakich będzie korzystał program obsługi urządzenia jak i samo urządzenie, ale co jeśli zmienimy konfiguracje sprzętu? Rekompilacja jądra lub modułu. Podobna sytuacja nastąpi jeśli po prostu usuniemy jakąś kartę z profilu sprzętowego. Co wtedy? Dlatego każdy program obsługi urządzenia, które korzysta z takich zasobów jak porty I/O, przerwania sprzętowe lub kanały DMA, powinny najpierw stwierdzić jakie zasoby należy zająć a potem powiadomić jądro o rezerwacji tychże zasobów.

6.1.2 Porty I/O.

Wykrywanie sprzętu należy zacząć od próby odgadnięcia portów na których odpowiada urządzenie. Najprościej byłoby gdyby można się dowiedzieć jakich portów urządzenie używa poprzez odczytanie tych danych z portów. Ale niestety nie da to żadnych rezultatów dla większości urządzeń. Dane czytane z portów nie muszą pochodzić od naszego urządzenia i mogłyby być mylne dla nas. Nie ma wiec wyjścia, należy spróbować zapisać do portu jakąś wartość i sprawdzić czy urządzenie odpowie na taki sygnał. Ale przecież nie można zapisywać do portów już używanych przez inne urządzenie. Mogłoby to doprowadzić do załamania się systemu. Dlatego jądro udostępnia możliwość

[15] Autorem tego rozdziału jest Rafał Wijata

blokowania portów I/O. Można to zrobić na dwa sposoby. Pierwszy to jako parametr do jądra podąć "***reserve=0x300,0x10***", co oznacza iż jądro już przy starcie rezerwuje porty od 0x300 do 0x30F; drugi parametr to ilość portów. Z punktu widzenia programisty wygląda to trochę inaczej. Jądro udostępnia funkcje:

```
* void request_region(int from, int num, const char name),
```

która rezerwuje paczkę '*num*' portów począwszy od '*from*'. Trzeci parametr jest nazwą programu obsługi. Jest potrzebny tylko do wiadomości systemu. Tę nazwę można potem zobaczyć oglądając plik '/proc/ioports'. Są w nim zapisane aktualnie używane porty wraz z nazwami urządzeń, które je używają. Do zwalniania portów (tylko dla modułów) służy inna funkcja:

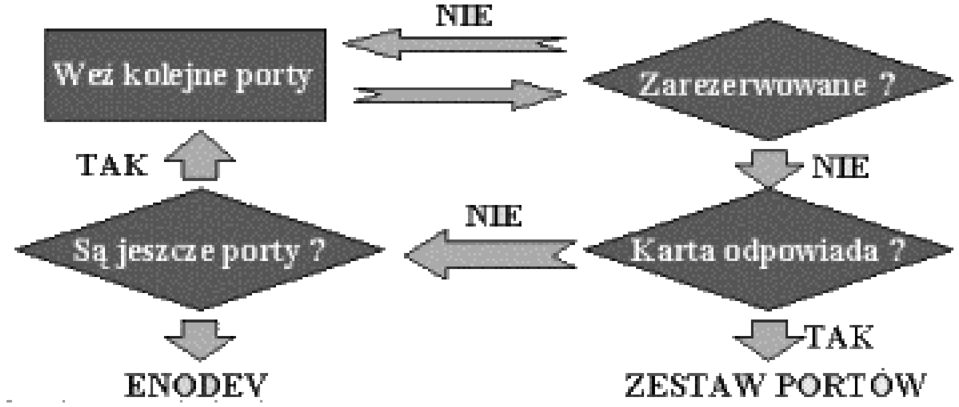
```
* void release_region(int from, int num),
```

parametry są analogiczne do poprzedniej. Żadna z powyższych funkcji nie zwraca informacji o błędach. Należy więc jeszcze móc sprawdzić czy jakiś region jest już używany czy jeszcze nie? Do tego przyda się funkcja:

```
* int check_region(int from, int num).
```

Jeśli ta funkcja zwróci zero to porty są wolne, jeśli nie to błąd *-EBUSY*, i wtedy przynajmniej jeden port z podanego zakresu jest zajęty. Wreszcie jak mógłby wyglądać przykładowy algorytm szukania urządzenia:

```
//card_portlist - tablica przypuszczalnych początków
//dla portów I/O, zakończona zerem.
#include <linux/ioports>         //tu są nasze funkcje
for (i=0; card_portlist[i]; i++) {
int ioaddr = card_portlist[i];
if (check_region(ioaddr, CARD_IO_EXTEND)) continue;
if (card_probe(ioaddr) == 0) return ioaddr;
};
return -ENODEV;
```



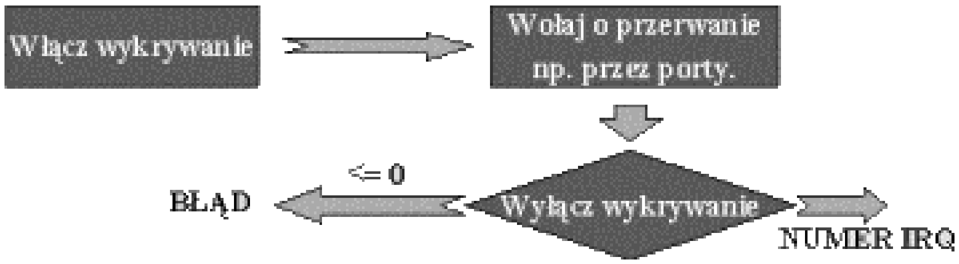
6.1.3 Przerwania.

Stajemy teraz przed problemem jak sprawdzić na którym przerwaniu działa karta? Dla nowszych kart typu Plug&Pray oraz PCI ten problem jest prostszy, a może nawet nie istnieje. Ale o tych rozwiązaniach trochę później. Najpierw zajmiemy się kartami, które konfiguruje się przez zworki. Sposób wykrywania jest zawsze taki sam. Alokujemy wszystkie wolne przerwania, prosimy urządzenie o wykonanie przerwania i sprawdzamy które zostało zgłoszone. Linux jeszcze upraszcza sprawę udostępniając odpowiednie funkcje:

```
* long probe_irq_on()
* int probe_irq_off(long irqs).
```

Pierwsza z nich zwraca nam maskę nieużywanych do tej pory przerwań. Teraz możemy zawołać do urządzenia o przerwanie. Możemy to zrobić zapisując adpowiednie wartości do portów, a te już znamy. Druga funkcja jako parametr bierze maskę, czyli wynik pierwszej i odpowiada jakie przerwanie zostało zgłoszone. Jeśli jednak żadne przerwanie w tym czasie nie uaktywniło się to zwraca zero. Jeśli zaś więcej niż jedno to daje (- najmniejszy numer przerwania). Teraz wykrycie jest proste:

```
unsigned long irqs = probe_irq_on();
outb(CALL_IRQ, DEV_IO_CONTROL);
                             //zawołaliśmy do urządzenia
o przerwanie
udelay(1000);                //czekamy na nie
irq = probe_irq_off(irqs);
if (irq <= 0) error("IRQ probe failed, set IRQ
manually !");
```



Jeszcze dwie sprawy. Po pierwsze skąd jądro wie które przerwania są już zajęte, po drugie jak zainstalować procedurę obsługi przerwania. Na pierwsze pytanie odpowiedz jest taka, że przerwanie jest używane, jeśli ma zainstalowaną procedurę obsługi, w przeciwnym przypadku jest wolne. Zaś do zainstalowania programu obsługi przerwania służy funkcja:

```
*                 int request_irq(int irq, void (*handler)(int,
void *, struct pt_regs *) hand,
                 long irqflags, char * devname, void *dev_id)
```

Parametry to numer przerwania; wskaźnik do procedury obsługi; flagi (np. czy

IRQ może być dzielone, lub czy ma to być przerwanie szybkie itp.); nazwa (widoczna w pliku '/proc/interrupts') i dane przekazywane procedurze obsługi. Funkcja ta dynamicznie alokuje strukturę *irqaction*, więc może wystąpić błąd - *NOMEM*. Jeśli uda się instalacja funkcja zwraca zero, w przeciwnym przypadku kod błędu. Do zwolnienia przerwania służy funkcja:

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

Więcej informacji i funkcji można zobaczyć w pliku 'arch/i386/kernel/irq.c'.

6.1.4 Kanały DMA.

Tu sprawa jest trochę kłopotliwa. Brak dobrych recept na zgadnięcie jaki kanał będzie używany do transmisji. Ale przecież jakoś się próbuje. Są na to trzy sposoby. Po pierwsze można zainicjować transmisje kanałem a potem sprawdzić czy dane dotarły do urządzenia. Niestety jednak niewiele kart potrafi odpowiedzieć na pytanie czy transmisja DMA się udała? Inna możliwością jest ustawienie karcie poprzez porty numeru DMA po którym będzie się odbywała transmisja. Ale tu także niewiele urządzeń ma taka możliwość. Można jeszcze ewentualnie przekazać numer kanału poprzez parametry programu obsługi. Tak jak poprzednio istnieje oczywiście mechanizm rezerwowania i zwalniania kanałów DMA. Służą do tego funkcje:

`* int request_dma(unsigned int dmanr, const char *device_id)`

`* void free_dma(unsigned int dmanr).`

Nazwy urządzeń i numery zarezerwowanych DMA można obejrzeć w pliku '/proc/dma'.

6.2 ISA Plug&Play oraz PCI.

Najpierw co to jest P&P. Otóż kiedyś powstał taki pomysł aby karty były tak produkowane, żeby zaraz po podłączeniu karta od razu zaczęła działać. Czyli tłumacząc z angielskiego włóż i używaj. Oznacza to że karcie można poprzez porty ustawić przerwanie i kanał DMA na których dalej będzie pracować. Dla kart PCI ta filozofia nawet działa, ale dla ISA jest gorzej. Najczęściej po prostu nie wiadomo czy karta jest włożona do gniazda, bo nie dostała sygnału reset i nie ustawiono zasobów z których ma korzystać. A bez tego karta po prostu nie odpowiada. Pod linuxem mamy do dyspozycji dwa programy '**pnpdump**', który skanuje wszystkie sloty ISA i sprawdza czy i jakieś karty są do nich włożone. Jeśli znajdzie jakąś kartę, to zapisuje dane o niej do pliku wynikowego. Teraz administrator musi wyedytować taki plik, przydzielić zasoby kartom, a następnie podesłać go jako parametr drugiego programu '**isapnp**', który dokonuje resetu i uatawienia kart. Teraz możemy już użyć naszych algorytmów wykrywania i/lub ustawiania zasobów. Dla takich kart można też zmienić zasoby odpowiednio zapisując do portów I/O. Dla kart PCI sprawa jest prostsza. W jądrze istnieje specjalny program obsługi dla urządzenia PCI. On załatwia wszystkie sprawy takie jak reset i przydzielenie zasobów dla kart. Można od niego tez przeczytać sporo informacji na temat podłączonych do PCI kart. Szczegółów można się dowiedzieć oglądając plik 'drivers/pci/pci.c'.

6.3 Jak napisać program obsługi urządzenia.

6.3.1 Funkcje rejestrujące.

W kodzie naszego programu obsługi urządzenia musi się gdzieś znaleźć sekwencja, która będzie odpowiedzialna za zarejestrowanie naszego programu w jądrze. Nasz program ma do tego odpowiednią funkcję systemową:

* `int register_blkdev(int Major, char *name, struct file_operations *fops)`,
* *lub* `register_chrdev()`.

Pierwsza rejestruje urządzenie blokowe, druga znakowe. Rejestrują one program obsługi urządzenia w systemie. Parametrami są numer główny urządzenia, nazwa, oraz wskaźnik do struktury operacji plikowych (*file_ operations*). Jeśli jako numer główny podamy zero, to funkcja spróbuje zarejestrować urządzenie na pierwszym wolnym numerze w systemie i zrócić go jako wynik. Jeśli funkcja zwróci zero lub liczbe większą od zera, to w porządku, nasz program został zarejestrowany, w przeciwnym przypadku funkcja zwróci błąd. Jeśli piszemy moduł, to dyspozycji programisty jest także funkcja, która zwalnia urządzenie:

* `int unregister_chrdev(int major, char *name)`,
* `int unregister_blkdev(int major, char *name)`.

Oprócz samego urządzenia i operacji dla niego, możemy także zarejestrować w systemie symbole eksportowane przez nasz program obslugi. Symbolami mogą być zmienne jak i funkcje. Po takim zabiegu są one widoczne dla kodu jądra, i mogą być wykorzystywane przez inne programy obsługi urządzeń. Służy do tego funkcja:

* `void register_symtab(struct symbol_table *stable)`.

Samą zaś strukturę możemy stworzyć w następujący sposób:

```
struct symbol_table *stable
{
#include <linux/symtab_begin.h>
X(symbol),
Xalias(really, sym)
#include <linux/symtab_end.h>
}
```

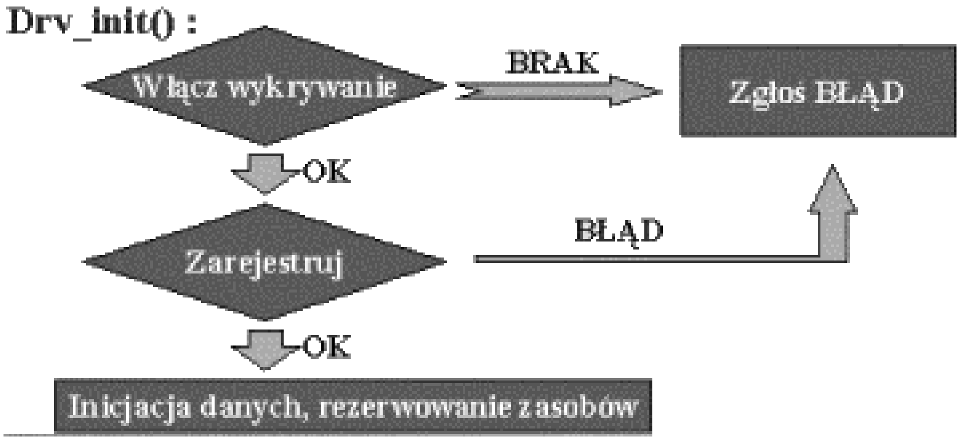
Makra X() i Xalias() są zadeklarowane w pliku 'include/linux/symtab_begin.h'. Pierwsze służy do wstawienia symbolu do tablicy, drugie zaś wstawia symbol do tablicy, ale pod inną nazwą niż występuje rzeczywiście występuje ona w naszym programie. Powyższe funkcje wykorzystuje przy inicjacji programu obsługi.

6.3.2 Inicjacja programu obsługi urządzenia.

Ale żeby inicjacja w ogóle była możliwa musimy w naszym kodzie umieścić funkcję:

* `int init_drv(void)`.

To w niej właśnie spróbujemy wykryć sprzęt, potem zarejestrujemy urządzenie oraz symbole w systemie. Ta funkcja zostanie wywołana podczas startu systemu, lub przy ładowaniu modułu do pamięci. Tak może wyglądać przykładowy schemat takiej funkcji:



6.3.3 Odczytywanie parametrów dla programów obsługi.

We wcześniejszych paragrafach była mowa o tym że można do programu obsługi przesyłać parametry. Z modułami sprawa jest prosta, bo wystarczy zadeklarować zmienną *"static typ nazwa"*, a potem jako parametr *insmod'a* podać "*nazwa=XXX*". *Insmod* sam wstawi w odpowiednie miejsce pamięci wartość. Jednak jeśli wkompilujemy nasz program w jądro, to nie mamy takiej możliwości. Wtedy parametry możemy podawać jako parametry jądra (np. przez LILO). Parametry mają wtedy składnię: '*drvname=param1,..,paramN*'. W pliku 'init/main.c' jest funkcja *parse_ options()*, która zajmuje się analizą wiersza wejściowego dla jądra. Dla każdej grupy parametrów wywołuje ona inną funkcję *checksetup()*, która sprawdza czy w tablicy *bootsetups*[] (też w 'init/main.c') istnieje urządzenie o nazwie takiej jak ciąg znaków przed znakiem równa się. Jeśli tak to próbuje zmienić pierwsze dziesięć parametrów na liczby i wstawić do odpowiedniej tablicy. Resztę ciągu zachowuje niezmienioną. Tak spreparowane dane podsyła jako parametr do funkcji:

```
* void drv_setup(char *s, int *p).
```

Ta funkcja musi się znaleźć w naszym programie obsługi. Parametry tej funnkcji to oczywiście wcześniej przygotowane dane. Przy czym p[0] jest liczbą elementów w tablicy p. Oczywiście nie trzeba implementować tej funkcji jeśli nie chcemy przyjmować żadnych parametrów na wejściu. Ta funkcja będzie wywołana podczas startu systemu, jeszcze przed funkcja *init_ drv()*. Z tego powodu nie da się jej zarejestrować dynamicznie w systemie. Jądro już w czasie startu systemu musi o niej wiedzieć. Dlatego ręcznie należy wyedytować plik 'init/main.c' i do struktury *bootsetups*[] dodać naszą funkcję. Wygląda to tak:

```
struct {
char *str;
void (*setup_func)(char *, int *);
} bootsetups[] = {
...
{"drvname", drv_setup},                //nasza funkcja
{0,0}
};
```

W tej funkcji powinniśmy już przygotować wartości zmiennych do uruchomienia funkcji *init_ drv()*. Przykładem jest:

```
if (!strcmp(s, "off")) disable = 1;          //nie
uruchamiamy usługi
if (p[0]   > 0) count = p[1];                   //ilość
urządzeń logicznych
```

6.3.4 Jakie funkcje są w strukturze *file_ operations*.

Do tej pory opisane zostały funkcje związane z interfejsem jądra. Teraz trochę o funkcjach które związane są ściśle z urządzeniem i interfejsem dla programisty, wykorzystującemu to urządzenie. Jak już wiadomo zestaw funkcji jest zawsze taki sam dla każdego urządzenia. Wskaźniki do nich są zapisane w strukturze *file_ operations*, i ta właśnie struktura jest podawana jako parametr funkcji rejestrującej urządzenie w systemie. Po spodem jest taka przykładowa struktura:

```
static struct file_operations drv_fops = {
drv_lseek,
drv_read,               /* jeśli obsługujemy urządzenie blokowe */
                        /* to ten powinien wołać block_read */
block_write,            /* standardowy dla blokowych */
NULL,                   /* drv_readdir - nie używane */
NULL,                   /* drv_select - nie obsługujemy */
drv_ioctl,
NULL,                   /* drv_mmap */
drv_open,               /* open i release powinny być napisane ! */
drv_release,
blk_fsync               /* fsync() */   };
```

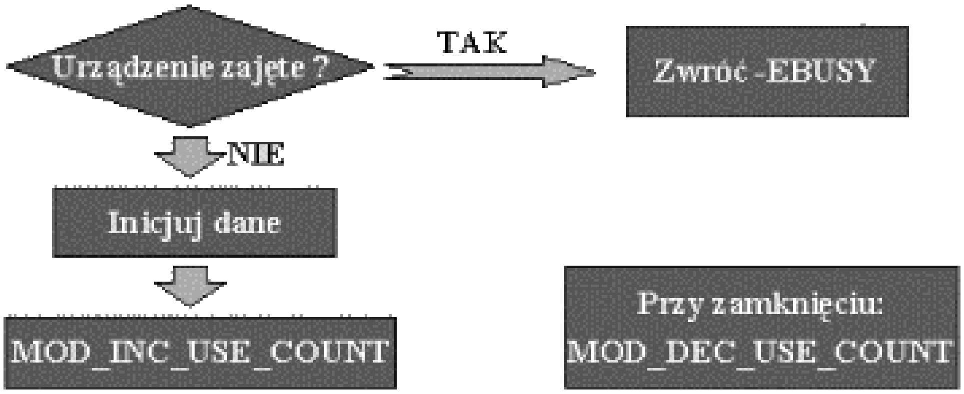
To co najważniejsze to fakt, iż nie trzeba implementować wszystkich funkcji. Tak naprawdę to nie trzeba implementować żadnej z nich, choć mocno zalecane jest napisanie przynajmniej *drv_ open()* i *drv_ close()*. Reszta jest opcjonalna. Warto jeszcze powiedzieć, że dla urządzeń blokowych istnieją standardowe funkcje czytania, pisania i opróżniania buforów *block_fsync()*. Tę trzecią, zazwyczaj wpisuje się po prostu do naszej struktury operacji na urządzeniu. Dwie pozostałe (czytanie i pisanie) też można po prostu wykorzystać i nie pisać własnych funkcji, ale zalecane jest napisanie własnych, które jednak gdzieś w kodzie i tak wywołają standardowe czytanie lub pisanie. Jeśli nie chcemy implementować jakiejś funkcji, to w miejsce jej wskaźnika wstawiamy *NULL*. Wtedy system zadba o to, aby w przypadku jej wołania został zwrócony błąd. Wyjątkiem jest *lseek()*, która standardowo nie zwraca błędu, więc jeśli chcemy jakiś zwracać, to musimy zaimplementować tę funkcję sami.

6.3.5 Otwieranie i zamykanie urządzenia.

```
- int drv_open(struct inode *node, struct file *f);
- int drv_release(struct inode *n, struct file *f);
```

Ta para funkcji będzie wywoływana przy próbie otwarcia i zamknięcia urządzenia. Co ważne dla tych funkcji, to fakt, iż muszą one zliczać liczbę otwartych urządzeń. Jeśli zaś piszemy moduł to powinny w swoim ciele wywoływać makra, odpowiednio *MOD_ INC_ USE_ COUNT* i

MOD_DEC_USE_COUNT. Można jednak je wstawić na stałe do funkcji, ponieważ dla funkcji wkompilowanych w jądro mają one (makra) puste znaczenie. Oto mały schemat procedury otwarcia urządzenia:



6.3.6 Czytanie i pisanie z/na urządzenie.

- `int drv_read(struct inode *n, struct file *f, char *buf, int count);`
- `int drv_write(struct inode *n, struct file *f, char *buf, int count);`

Te funkcje implementują pisanie i czytanie do i z urządzenia. Należy w tej funkcji skopiować *buf* do urządzenia dla pisania, i odwrotnie dla pisania. *Count* oznacza ilość bajtów do skopiowania. Trzeba tu zwrócić uwagę, że pamięć bufora znajduje się w pamięci użytkownika, toteż do pobierania i pisania danych w przestrzeni użytkownika służą dodatkowe funkcje:

* `memcpy_tofs(void *dest, void *source, int count)` - skopiowanie '*count*' bajtów do
przestrzeni użytkownika,
* `memcpy_fromfs(void *dest, void *source, int count)` - jw., ale odwrotnie,
* `put_user(x, void *ptr)` - wpisanie bajtu/słowa/długiego słowa do pamięci użytkownika
(adres w ptr); makrodefinicja działa magicznie - rozmiar określony jest na podstawie
typu parametru 'x',
* `get_user(void *ptr)` - jw, ale odczytanie.

Należy tu jeszcze powiedzieć o funkcjach alokujących i zwalniających pamięć jadra, ponieważ nie można używać *malloc()* itp.

* `void *kmalloc(unsigned int size, int priority);`
* `void kfree(void * obj);`
* `void *vmalloc(unsigned long size);`
* `void vfree(void *addr);`

Funkcja *kmalloc()* pozwala na przydział spójnego obszaru wielkości maksymalnie 32 stron pamięci (dla Intela niecałe 128kb pamięci; część pamięci jądro rezerwuje na nagłówek bloku). Przydział pamięci realizowany jest szybko (algorytm Buddy). Parametr *priority* określa rodzaj pamięci (stałe GFP_* określające typ pamięci są zdefiniowane w pliku 'include/linux/mm.h'). Definicje znajdują się w pliku 'include/linux/malloc.h'. Przy pomocy *vmalloc()* można przydzielić obszar dowolnie duży (pod warunkiem, że jest odpowiednio dużo wolnej pamięci fizycznej), ale już niespójny (pamięć ta przechodzi przez translację adresów). Definicje funkcji znajdują się w pliku 'include/linux/malloc.h'.

Tutaj też należy wspomnieć o sytuacji kiedy np. brak danych na urządzeniu, a jakiś program chce je czytać. Co wtedy? Można zwrócić błąd, ale może program chce czytać dane dopiero jak się pojawią. Rozwiązywanie tego poprzez pętle próbującą czytać aż się uda jest karygodne. Dlatego w jądrze istnieje mechanizm usypiania i budzenia procesów oczekujących na operacje związane z urządzeniami. Są to '*wait queues*'. Proces, który musi oczekiwać na zajście pewnego zdarzenia usuwany jest z kolejki procesów gotowych i umieszczany jest w kolejce oczekiwania (*wait queue*) - stan procesu zmienia się z *TASK_RUNNING* na *TASK_INTERRUPTIBLE* bądź *TASK_UNINTERRUPTIBLE*. Mamy do dyspozycji funkcje:

* `init_waitqueue()` - przed użyciem wait_queue trzeba ją zainicjować,

* `sleep_on()` - umieszczenie procesu w kolejce oczekiwania w stanie odpornym na
sygnały (*TASK_UNINTERRUPTIBLE*),

* `sleep_on_interruptible()` - jw, ale przejście w stan *TASK_INTERRUPTIBLE*
(nadejście sygnału spowoduje przejście w stan *TASK_RUNNING*),

* `wake_up()` - wstawienie wszystkich (znajdujących w stanie TASK_INTERRUPTIBLE
bądź TASK_UNINTERRUPTIBLE) znajdujących się w kolejce oczekiwania do
kolejki procesów gotowych Uwaga: funkcja nie powoduje usunięcia procesu z
kolejki oczekiwania - proces "usunie" się sam, po wznowieniu działania,

* `wake_up_interruptible()` - jw, ale tylko dla procesów w stanie *TASK_INTERRUPTIBLE*

Powyższe funkcje zdefiniowane są w 'include/linux/sched.h' (oprócz pierwszej, która znajduje się w 'include/linux/wait.h'.

6.3.7 Funkcja kontroli urządzenia.

```
- int drv_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
                unsigned long arg)
```

To dość ważna funkcja. Służy ona do zmieniania charakterystyki urządzenia podczas jego pracy. Jest to funkcja przekazująca programowi obsługi polecenia do wykonania. Mogą one być różne, i są inne dla każdego urządzenia. Na przykład do głośnika systemowego można tą drogą przesłać częstotliwość odtwarzanego dźwięku, lub ilość możliwych do otworzenia urządzeń. Implementując tę funkcję powinniśmy się trzymać pewnych standardowych ustaleń co do formatu poleceń. W pliku 'include/linux/ioctl.h' zdefiniowane są makra tworzące polecenia:

* `_IO(dev, cmd)`

```
* _IOW(dev, cmd, type)
* _IOR(dev, cmd, type)
* _IORW(dev, cmd, type)
```

Gdzie '*dev*' jest jedną literą identyfikującą nasze urządzenie, '*cmd*' jest numerem polecenia dla urządzenia, a '*type*' jest rozmiarem danych przesyłanym do lub z funkcji do wołającego. Znaczenia makr są następujące. Pierwsze jest poleceniem bez argumentów, drugie tworzy polecenie które będzie zapisywało do przestrzeni użytkownika dane typu *type*, trzecie będzie czytało, a ostatnie będzie i czytało i pisało. Przykładem niech będzie polecenie

```
#define SNDCTL_DSP_SET_STEREO  _IOW(P, 1, int)
```

A oto jak jest takie polecenie kodowane w pamięci komputera:

Zarezerwowane litery dla urządzeń są w pliku 'Documentation/ioctl_number.txt'. Cała ta zabawa służy jednak czemuś. A mianowicie standaryzacji poleceń dla *ioctl()*, i co ważniejsze dla bezpieczeństwa. Konstruując w ten sposób polecenia unikamy powtórzeń, więc potem nie nasz program nie będzie odbierał pomyłkowo przesłanych poleceń. Będzie rozpoznawał tylko swoje polecenia, pomimo tych samych numerów komend.

6.3.8 Inne, mniej ważne funkcje ze struktury *file_ operations.*

```
- int lseek(struct inode *node, struct file *f, off_t
t, int count)
```

W tej funkcji należy zaimplementować zmienianie aktualnej pozycji czytania/pisania urządzenia. Sposób podawania pozycji jest identyczny jak dla zwykłego pliku. Implementuje się ją właściwie tylko dla urządzeń blokowych.

```
- int select(struct inode *node, struct file *f, int
rw, select_table *table)
```

Ta funkcja powinna odpowiadać czy można czytać lub pisać na urządzenie. Służy to zapobieganiu aktywnemu czekaniu w programie na pierwsze urządzenie, na którym będziemy mogli wykonać operację.

```
- int mmap(struct inode *node, struct file *f, struct
vm_area_struct *buf)
```

Ma sens tylko dla urządzeń stałej wielkości i tylko dla blkowych. Mapuje urządzenie do pamięci. Używane raczej do mapowania plików niż urządzeń.

```
- int (*readdir) (struct inode *, struct file *,
void *, filldir_t);
- int (*fsync) (struct inode *, struct file *);
- int (*fasync) (struct inode *, struct file *, int);
```

Znów sens ma tylko *fsync()*, i to tylko dla urządzeń blokowych. Właściwie to też nie powinno się tego implementować tylko użyć standardowej funkcji *block_fsync()*. Funkcje te są używane w odniesieniu do plików, nie urządzeń.

`- int check_media_change(kdev_t dev)`

Funkcja używana tylko dla urządzeń z wymiennym nośnikiem danych. Powinna odpowiedzieć na pytanie czy zmieniono źródło danych, jeśli odpowiedź będzie pozytywna to wywoływana jest funkcja:

`- int (*revalidate) (kdev_t dev),`

która od nowa wczytuje potrzebne struktury danych. Przykładem implementacji tych funkcji jest program obsługi stacji dyskietek.