Rozdział 5

Podsystem wejścia-wyjścia

5.1 Pojęcia podstawowe

Istnieje dużo różnego rodzaju sprzętu, który można podłączyć do komputerów działających pod kontrolą Linuxa. Każde takie urządzenie wymaga osobnego programu obsługi (ang. *driver* - sterownik), który jest swoistym tłumaczem poleceń systemu na sygnały zrozumiałe dla sprzętu. Ponadto, o ile program taki jest napisany prawidłowo, nie ma możliwości nieprawidłowego korzystania z urządzenia zewnętrznego - wszystkie operacje, jakie na nim wykonujemy, muszą zostać zaaprobowane przez sterownik. Tak więc program obsługi, oprócz swego podstawowego zadania - umożliwienia komunikacji, spełnia także funkcję ochronną.

5.1.1 Numery główne i drugorzędne

Obecność w systemie wielu urządzeń zewnętrznych powoduje, że jednocześnie jest zainstalowanych wiele programów obsługi. Aby mogły bezboleśnie współdziałać, wprowadzono metodę ich jednoznacznego opisu poprzez tzw. numer nadrzędny (ang. major number), określający rodzaj urządzenia, którym 'opiekuje się' dany program. Dodatkowo, ponieważ sterownik może obsługiwać wiele urządzeń (np. dyski twarde), każde z nich identyfikowane jest przez numer podrzędny (ang. minor number), liczbę całkowitą z zakresu 0-255.

Numery nadrzędne przydzielane są przez odpowiednią osobę pracującą nad rozwojem Linuxa. Pełną ich listę można znaleźć w Documentation/devices.txt.

Kombinacja numeru nadrzędnego i podrzędnego w połączeniu z typem sprzętu (blokowy/znakowy) jednoznacznie wskazuje na urządzenie w systemie.

Obsługą numerów urządzeń zajmują się makra zdefiniowane w pliku include/linux/kdev_t.h (numery zapisywane są w strukturze i-węzła w jednym polu typu kdev_t):

```
typedef unsigned short kdev_t;

#define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

5.1.2 Pliki Specjalne

Wyróżniamy dwa podstawowe typy urządzeń: blokowe i znakowe.

- Do urządzeń blokowych mamy dostęp swobodny, tzn. możemy operować (czytać i pisać) na dowolnym bloku. Operacje te są przeprowadzane za pośrednictwem pamięci cache, jednak jest ona dla użytkownika przeźroczysta. Klasycznym przykładem urządzeń blokowych są systemy plików.
- Z kolei do urządzeń znakowych mamy dostęp sekwencyjny, co wymusza konieczność zastosowania bufora. Do tego typu urządzeń należy większość znanego nam sprzetu - kart dźwiękowe, drukarki, skanery.

Powszechna tendencja do ukrywania sprzętu przed użytkownikiem, bez jednoczesnego ograniczania jego możliwości, doprowadziła do pomysłu reprezentowania urządzeń fizycznych przez pliki. Dzięki temu dostęp do sprzętu odbywa się za pomocą wywołań funkcji read/write zastosowanej do pliku specjalnego, co znacznie upraszcza przenoszenie programów pomiędzy różnymi wersjami UNIXa. Pliki reprezentujące urządzenia zwyczajowo umieszczone są w katalogu /dev (ang. devices). O tym, że są to pliki specjalne, możemy dowiedzieć się np. poprzez wykonanie polecenia 'ls -l', które w kolumnie uprawnien pokazuje również typ pliku (- - zwykły plik, d - katalog, l - dowiązanie symboliczne, c - urządzenie znakowe, b - urządzenie blokowe, etc). Informacja ta pochodzi z i-węzła pliku. W przypadku plików urządzeń tam również przechowywane są wartości numerów nadrzędnych i podrzędnych.

5.1.3 Tworzenie i usuwanie urządzeń

Do tworzenia plików specjalnych służy polecenie systemowe `mknod`

`mknod /dev/nazwa typ nr_nadrzędny nr_podrzędny`

przy czym jako typ podajemy 'b' (urządzenie blokowe) lub 'c' (znakowe). W zdecydowanej większości przypadków będziemy używać tego drugiego typu. Czasami konieczne jest stworzenie dwóch rodzajów plików dla jednego urządzenia (oczywiście o różnych, choć podobnych, nazwach). W takich przypadkach plik urządzenia znakowego wykorzystywany jest np. do sterowania urządzeniem, do którego mamy także dostęp bezpośredni. Oczywiście, samo stworzenie pliku nic nam nie da bez dołączenia do jądra Linuxa odpowiedniego programu obsługi. W przeciwnym wypadku, gdy zechcemy skorzystać z urządzenia, dowiemy się tylko, że '/dev/nazwa: device not configured.'

Pliki specjalne usuwane są w tradycyjny sposób komendą `rm`.

5.1.4 Porty wejścia wyjścia

Porty stanowią niskopoziomwy interfejs komunikacji systemu operacyjnego z urządeniami wejścia wyjścia. W architekturze i386 występuje 16 bitowa przestrzeń adresowa dla urządzeń, w związku z czym mamy dostęp do 65536 portów.

W celu uniknięcia konfliktów w obrębie zakresów wejści wyjścia w linuksie wprowdzono mechanizm rejestracji używanych portów (pliki kernel/resource.c oraz include/ioports.h).

`extern int check_region(unsigned long from, unsigned long extent);` Sprawdzenie czy obszar portów od from o długości extent jest już wykorzystywany

`extern void request_region(unsigned long from, unsigned long extent, const char *name);` Zarezerwowanie podanego obszaru pod nazwą name.

`extern void release_region(unsigned long from, unsigned long extent);` Zwolnienie podanego obszaru.

`extern int get_ioport_list(char *);` Pobranie listy zarezerwowanych portów w postaci tekstu.

Listę aktualnie wykorzystywanych portów można pobrać z pseudo pliku `/proc/ioports`, który zawiera wynik funkcji `get_ioports_list`. Przykładowa lista portów:

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0220-022f : soundblaster
0330-0333 : MPU-401 UART
0376-0376 : ide1
0388-038b : Yamaha OPL3
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(auto)
d400-d41f : eth0
e000-e007 : ide0
e008-e00f : ide1
```

Dostęp do portów zapewniają funkcje zadeklarowane w pliku `include/asm/io.h`:

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
Odczyt i zapis bajtów do podanego portu.

unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
Odczyt i zapis portów 16-bitowych.

unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);
Otczyt i zapis portów 32-bitowych.
```

Istnieją również funkcje do odczytu i zapisu ciągów danych do portów, które pobierają bufor do odcztu/zapisu oraz długość danych.

5.1.5 Tablice rozdzielcze urządzeń blokowych i znakowych

Struktury te przechowują adresy funkcji implementujących poszczególne operacje realizowane przez sterownik danego urządzenia, identyfikowany numerem nadrzędnym, który jest równocześnie indeksem tablicy. Deklaracje powyższych struktur znajdują się w pliku `fs/devices.c`.

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

static struct device_struct chrdevs[MAX_CHRDEV] = {
    { NULL, NULL },
};

static struct device_struct blkdevs[MAX_BLKDEV] = {
    { NULL, NULL },
};
```

Gdy korzystamy z systemowej obsługi odczytu/zapisu dla urzadzeń blokowych potrzebna jest dodatkowo funkcja realizujaca kolejkowane zadania wejscia/wyjscia. Ustala sie ja w oddzielnej tablicy zdefiniowanej w `include/linux/blkdev.h`. i zadeklarowanej w `devices/block/ll_rw_blk.c`

```
typedef void (request_fn_proc) (void);
typedef struct request ** (queue_proc) (kdev_t dev);

struct blk_dev_struct {
        request_fn_proc         *request_fn;
        queue_proc               *queue;
        void                      *data;
        struct request            *current_request;
        struct request   plug;
        struct tq_struct plug_tq;
};

struct blk_dev_struct blk_dev[MAX_BLKDEV];
```

5.1.6 Inicjowanie urządzeń w jądrze

Aby system wiedział, że dany program ma pośredniczyć w komunikacji z danym urządzeniem, musimy taki program zarejestrować. Służy do tego jedna z funkcji:

```
int register_chrdev(unsigned int major,
                    const char *name,
                    struct file_operations *fops);
int register_blkdev(unsigned int major,
                    const char *name,
                    struct file_operations *fops);
```

Powoduje ona umieszczenie pod indeksem major w tablicy `chrdevs[]` (tablica rozdzielcza urządzeń znakowych) lub `blkdevs[]` (tablica rozdzielcza urządzeń blokowych) wykazu dozwolonych operacji. Jeżeli program obsługi urządzenia jest już zarejestrowany dla danego numeru nadrzędnego, a operacje dozwolone nie są identyczne z poprzednimi, rejestracja się nie powiedzie (wynik <0). Jeżeli numer nadrzędny wynosi 0, urządzenie rejestrowane jest na pierwszej wolnej pozycji, zaś przydzielony mu numer zwracany jest jako wynik funkcji. Dzięki temu możemy dodawać własne programy obsługi, które nie mają oficjalnie przydzielonego numeru nadrzędnego. Od tej pory wszystkie operacje na urządzeniu (`open/close/read/write/..`) powodują wywołanie procedur obsługi określonych w odpowiedniej tablicy rozdzielczej.

Jesli korzystamy z funkcji `block_read` i `block_write` do obslugi kolejkowania zadan dostepu nalezy dodatkowo zarajestrowac funkcje realizujaca obsluge zadan. Dokonuje sie tego bezposrednio przez wpis do tablicy `blk_dev[]` w pole `request_fn`.

5.1.7 Podprogramy obsługi urządzeń

Program obsługi urządzenia udostępnia zestaw funkcji, za pośrednictwem których system komunikuje się ze sprzętem. Dostęp do nich jest możliwy dzięki strukturze `file_operations`, przekazywanej jako jeden z parametrów podczas inicjalizacji sterownika, a umieszczonej w tablicy rozdzielczej urządzeń (blokowych bądź znakowych).

```
struct file_operations {
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *);
int (*fasync) (int, struct file *, int);
int (*check_media_change) (kdev_t dev);
int (*revalidate) (kdev_t dev);
int (*lock) (struct file *, int, struct file_lock *);
};
```

Zadania poszczególnych funkcji:

`loff_t (*llseek) (struct file *, loff_t, int);`

Funkcja dotyczy tylko urządzeń blokowych i zmienia bieżącą pozycję w pliku na zadaną. O ile nie jest zdefiniowana, system samodzielnie obsłuży żądania przesunięcia względem początku pliku lub bieżącej pozycji (i skorzysta z nowej wartości przy najbliższej operacji read/write), zaś zwróci błąd w przypadku próby ustawienia pozycji względem końca pliku.

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

Funkcja powoduje odczytanie z urządzenia zadanej ilości bajtów i przepisanie ich do bufora wskazanego przez użytkownika. Zwracana jest liczba przepisanych bajtów (w najlepszym przypadku równa zadanej, może byc też równa 0) lub ujemny kod błędu. Możliwe jest też zawieszenie się procesu na tej funkcji w oczekiwaniu na dane z urządzenia, o ile flagi (zawarte w polu f_flags struktury file) nie zabraniają tego.

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

Funkcja przesyła do urządzenia zadaną liczbę bajtów z bufora wskazanego przez użytkownika. Analogicznie do `read()`, zwrócona może zostać liczba bajtów przesłanych (z zakresu 0 <= n <= zadana) lub ujemny kod błędu.

W przypadku dwóch powyższych funkcji, `read()` i `write()`, występuje istotna różnica między urządzeniami znakowymi i blokowymi. Dla urządzeń blokowych istnieją systemowe odpowiedniki obu funkcji, zastępujące je w strukturze `file_operations`, `block_read()` i `block_write()`, korzystające z buforów oraz tzw. procedur strategii, czyli zależnych od typu urządzenia metod dostępu do danych.

`int (*readdir) (struct file *, void *, filldir_t);`

Nie ma zastosowania do urządzeń.

`unsigned int (*poll) (struct file *, struct poll_table_struct *);`

Funkcja dająca możliwość odpytywania (ang. polling) urządzenia, czy możliwe jest przeprowadzenie operacji `read()` bądź `write()`, a także, czy wystąpiły sytuacje specyficzne. Używana do implementacji systemowej funkcji `select()` oraz `poll()`. W linuksie 2.0 wystepowala w tym miejscu funkcja `(*select)()`, która nie umożliwiała implementacji `poll()` (funkcja pochodzaca z Systemu V)

`int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);`

Funkcja używana do zmiany parametrów pracy urządzenia (np. zmiana prędkości przesyłania danych przez terminal) lub wykonania specyficznych dla niego operacji (np. formatowanie ścieżki).

`int (*mmap) (struct file *, struct vm_area_struct *);`

Funkcja odwzorowująca część pliku do pamięci bieżącego procesu. Wszystkie parametry operacji są opisane w strukturze vm_area.

`int (*open) (struct inode *, struct file *);`

Funkcja ustanawia połączenie między procesem a urządzeniem zewnętrznym, dokonując jednocześnie weryfikacji poprawności jego działania, inicjalizacji sprzętu (o ile jest otwierane po raz pierwszy), inicjalizacji prywatnych struktur danych. Jest wywoływana z poziomu systemowej funkcji `open()` po zmodyfikowaniu i-węzła pliku specjalnego (zwiększenie licznika odwołań) i przydzieleniu mu pozycji w tablicy plików oraz deskryptora pliku. Operacja może się nie powieść (niezerowy rezultat) - wtedy funkcja wywołująca wycofuje dokonane wcześniej zmiany i przekazuje zwrócony kod błędu wyżej.

`int (*flush) (struct file *);`

Nowość w wersji 2.2, wywoływana zawsze przez funkcje systemową `sys_close()`, niezależnie od licznika odwołań (`release()` wywoływany jest gdy licznik odwołań spadnie do 0)

`int (*release) (struct inode *, struct file *);`

Funkcja (znana również jako `close()`) usuwa połączenie między procesem a urządzeniem zewnętrznym. Jest wywoływana przez systemową funkcję `close()`, jednak tylko wtedy, gdy uprzednio wywołana była funkcja `open()` sterownika i odpowiedni licznik wykorzystania jest równy 0. Przykładem wywołania systemowej funkcji `close()`, która nie odwoła się do `close()` sterownika, jest zamykanie deskryptora powstałego w wyniku działania funkcji `fork()` czy `dup()` w sytuacji, gdy deskryptor ojca ciągle jest otwarty.

`int (*fsync) (struct file *, struct dentry *);`

Czyści bufory, po uprzednim przesłaniu ich zawartości do urządzenia.

`int (*fasync) (int, struct file *, int);`
Wywołana w przypadku zmiany flagi `O_ASYNC` na podanym pliku.
`int (*check_media_change) (kdev_t dev);`
Funkcja mająca zastosowanie tylko w przypadku urządzeń blokowych, sprawdzająca, czy od czasu wykonania ostatniej operacji read/write nastąpiła zmiana nośnika.
`int (*revalidate) (kdev_t dev);`
Wywoływana w przypadku, gdy `check_media_change()` zwróci true, usuwa zdezaktualizowane bufory urządzenia.
`int (*lock) (struct file *, int, struct file_lock *);`
Nowość w wersji 2.2, wywoływana po pomyślnie zakończonej operacji `posix_make_lock()` w funkcji `fcntl_getlk()`. Stosowana do implementacji blokad plikow na zdalnych systemach (np. nfs)

5.1.8 Obsługa funkcji systemowych

Definicje funkcji systemowych znajduja sie w plikach w katalogu `fs/`.

Algorytmy dzialania funkcji podstawowych funkcji zwiazanych z urzadzeniami:

1. `open()` Realizowana przez wywołanie `sys_open()`, gdzie po pobraniu wolnego deskryptora wykonywane jest `filp_open()`, tam znajduje się interesująca nas część dotycząca inicjalizacji tablicy operacji na pliku `file_operations`.

```
f->f_op = NULL;
/* ustawiamy f_op na default_file_ops, gdzie zdefiniowana jest jedynie
   funkcja chrdev_open() lub blkdev_open(), która ustawia właściwe
   file_operations i wywołuje ponownie f_op->open */
if (inode->i_op)
        f->f_op = inode->i_op->default_file_ops;
if (f->f_op && f->f_op->open) {
        error = f->f_op->open(inode,f);
        if (error)
                goto cleanup_all;
}
```

2. `close()` Funkcja realizowana przez `sys_close`, właściwe zamknięcie pliku wykonuje `filp_close`:

```
int filp_close(struct file *filp, fl_owner_t id)
{
        int retval;
        struct dentry *dentry = filp->f_dentry;

        if (filp->f_count == 0) {
                printk("VFS: Close: file count is 0\n");
                return 0;
        }
        retval = 0;
```

```
/* tu wywolanie f_op->flush  */
        if (filp->f_op && filp->f_op->flush)
                retval = filp->f_op->flush(filp);
        if (dentry->d_inode)
                locks_remove_posix(filp, id);
/* fput zmniejsza licznik odwołań i wywołuje f_op->release jeśli spadł do 0  */
        fput(filp);
        return retval;
}
```

3. `ioctl()` Funkcja ta stanowi interfejs do ustawiania i pobierania paramertów urządzenia. Parametr `cmd` jest numerem funkcji dla danego urządzenia (numerom tym zazwyczaj przypisuje się stałe). Wartości `cmd` dla części urządzeń można uzyskać wywołując `man ioctl_list`.

```
asmlinkage int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
        struct file * filp;
        unsigned int flag;
        int on, error = -EBADF;

        lock_kernel();
        filp = fget(fd);
        if (!filp)
                goto out;
        error = 0;
        switch (cmd) {
                case FIOCLEX:
...
                case FIONCLEX:
...
                case FIONBIO:
...
/* Powyżej występuje obsługa flag close_on_exec oraz O_NONBLOCK
   niezależna od urządzenia */
                case FIOASYNC:
/* Obsługa flagi FASYNC służącej do włączania obsługi wejścia/wyjścia
   przy pomocy sygnału SIGIO */
                        if ((error = get_user(on, (int *)arg)) != 0)
                                break;
                        flag = on ? FASYNC : 0;
/* Powiadamy sterownik o ewentualnej zmianie flagi przez f_op->fasync() */
                        if ((flag ^ filp->f_flags) & FASYNC) {
                                if (filp->f_op && filp->f_op->fasync)
                                        filp->f_op->fasync(fd, filp, on);
                        }
                        if (on)
```

```
                                filp->f_flags |= FASYNC;
                        else
                                filp->f_flags &= ~FASYNC;
                        break;
                default:
/* Parametr cmd nie rozpoznany przez system - wywołujemy file_ioctl
   jeśli zwykły plik lub f_op->ioctl() jeśli plik specjalny */
                        error = -ENOTTY;
                        if (!filp->f_dentry || !filp->f_dentry->d_inode)
                                error = -ENOENT;
                        else if (S_ISREG(filp->f_dentry->d_inode->i_mode))
                                error = file_ioctl(filp, cmd, arg);
                        else if (filp->f_op && filp->f_op->ioctl)
                                error = filp->f_op->ioctl(filp->f_dentry->d_inode, filp, cmd,
        }
        fput(filp);
out:
        unlock_kernel();
        return error;
}
```

4. `read()` i `write()` Funkcje `sys_read` oraz `sys_write` korzystają bezpośrednio z `read` i `write` zdefiniowanych w `f_op`

5. `llseek()` Funkcja do zmiany pozycji w pliku jeśli nie jest zdefiniowana w `f_op->llseek` zostanie wywołana standartowa funkcja systemowa `default_llseek`, która nie obsługuje ustawiania względem końca pliku.

```
static inline loff_t llseek(struct file *file, loff_t offset, int origin)
{
        loff_t (*fn)(struct file *, loff_t, int);

        fn = default_llseek;
        if (file->f_op && file->f_op->llseek)
                fn = file->f_op->llseek;
        return fn(file, offset, origin);
}
```

5.2 Urządzenia blokowe

5.2.1 Wstęp.

Co to są urządzenia blokowe ?

Urządzenia blokowe to np. dysk twardy, czytnik kompaktów, czy stacja dysków. Charakteryzują się one przede wszystkim inną od urządzeń znakowych budową, a co za tym idzie innym działaniem i w związku z tym inną obsługą od strony programowej.

Cechy charakterystyczne urządzeń blokowych.

Są dwie cechy odróżniające urządzenia blokowe od pozostałych urządzeń :

- Po pierwsze urządzenia te, jak sama ich nazwa wskazuje są podzielone na jednostki - bloki. Oznacza to, że nie można przesłać między sterownikiem urządzenia a pamięcią operacyjną informacji o długości nie będącej wielokrotnością wielkości bloku. W praktyce np. gdy chcemy zapisac jeden bajt na urządzenie blokowe, musimy odczytać z niego blok, w którym znajduje się nasz bajt do pamięci, zmodyfikować tą jedną wartość w pamięci i zapisać cały blok na naszym urządzeniu blokowym. Jak widać nie jest to najlepsza cecha urządzeń blokowych, gdyż koszt przy zapisie bajt po bajcie byłby wielkości

  ```
  wielkość_bloku * (średni_czas__odczytu_bajtu + średni_czas_zapisu_bajtu) +
  czas_modyfikacji_komórki_pamięci.
  ```

  Jednak nie ma co się martwić, bo system operacyjny świetnie sobie z tym radzi za pomocą mechanizmu buforowania. Blok dyskowy jest buforowany w pamięci operacyjnej i dopiero po określonym czasie lub też po zmodyfikowaniu całego bloku jest zapisywany na urządzeniu blokowym.

- Drugim bardzo ważnym wyróżnikiem urządzeń blokowych jest swobodny dostęp do całego urządzenia co daje niesamowitą przewagę tych urządzeń w przypadku częstej konieczności korzystania ze zmagazynowanych danych rozrzuconych po całej "przestrzeni adresowej" urządzenia np. w przypadku korzystania z baz danych.

5.2.2 Urządzenia blokowe w jądrze systemu Linux.

Podstawowe struktury.

W jądrze linux'a znajduje się kilka struktur opisujących urządzenia blokowe. Oczywiście gdzieś musi się znajdować informacja o nazwie urządzenia i operacjach plikowych jakie można na nim wykonać. Taka informacja przechowywana jest w tablicy *struct device_ struct blkdevs[MAX_ BLKDEV]*, której pola są postaci:

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

Pola struktury fops są wypełniane przy inicjalizacji, podobnie jak w urządzeniach znakowych z jedną bardzo ważną różnicą. Pola read i write tej struktury wyglądają tak samo dla każdego urządzenia blokowego, tzn. read wskazuje na funkcję *ssize_ t block_ read(struct file * filp, char * buf, size_ t count, loff_ t *ppos)*, a write wskazuje na *ssize_ t block_ write(struct file * filp, const char * buf,size_ t count, loff_ t *ppos)*. Obie te funkcje obsługują mechanizm buforowania i czytania z wyprzedzeniem dla wszystkich urządzeń blokowych, co zwalnia programistę z obowiązku zajęcia się tym samemu.

Żądania

Skoro read i write są jednakowe dla wszystkich urządzeń blokowych, to pojawia się pytanie, gdzie znajduje się właściwa procedura komunikacji z urządzeniem? Okazuje się, że urządzenia blokowe są obsługiwane w nieco inny sposób, otóż każda chęć zapisu lub odczytu z urządzenia blokowego jest zapamiętywana w strukturze request i dodawana do kolejki żądań właściwej danemu urządzeniu. System sprawdza od czasu do czasu czy są żądania dla danego urządzenia i jeśli są to wywołuje jego procedurę strategii, która przegląda kolejkę żądań i komunikuje się z urządzeniem w celu ich zrealizowania. To właśnie procedura strategii jest odpowiedzialna za zapis i odczyt danych "zlecony" przez funkcje *block_read* i *block_write*. Procedura strategii, kolejka żądań i kilka dodatkowych pól znajdują się w tablicy *struct blk_dev_struct blk_dev[MAX_BLKDEV]*.

```
struct blk_dev_struct {
request_fn_proc *request_fn;    // procedura strategii
queue_proc *queue;         // kolejka żądań
...
};
```

Jak dokładnie działają *block_read* i *block_write*.

Block_read - funkcja sprawdza czy żądane bloki dyskowe znajdują się w pamięci buforowej. Jeśli tak to przepisuje zawartość buforów do przestrzeni adresowej uzytkownika. W przeciwnym wypadku generuje żądanie odczytu potrzebnych bloków i dodatkowo kilku bloków znajdujących się bezpośrednio za żądanymi blokami na urządzeniu blokowym. Jest to tzw. czytanie z wyprzedzeniem, a ilość bloków jaka jest odczytywana z wyprzedzeniem jest stała dla każdego urządzenia i jest zapisana w tablicy *int read_ahead[MAX_BLKDEV]*. W momencie wygenerowania żądania proces czytający jest usypiany do czasu zakończenia realizacji żądania. Po obudzeniu procesu odczytane dane są kopiowane do przestrzeni użytkownika.

Block_write - funkcja ta podobnie jak *block_read* sprawdza, czy bloki, które chcemy zmodyfikować znajdują się już w pamięci buforowej. Jeśli żądane bloki są już zbuforowane to zostają zmodyfikowane i oznaczone do zapisu, w przeciwnym wypadku muszą zostać odczytane z dysku, oczywiście tylko przy modyfikacji nie obejmującej całego bloku. Podobnie jak w *block_read*, przy generowaniu żądania odczytu, brana jest pod uwagę wartość tablicy *read_ahead[]*. Proces jest usypiany do momentu gdy bloki znajdą się w pamięci buforowej, po czym bloki są modyfikowane i oznaczane do zapisu.

5.2.3 Podprogramy obsługi dysków

Wiadomości wstępne o sterowniku ide.c

Działanie programów obsługi dysku twardego zostanie omówione na przykładzie sterownika *ide.c*.

Jest to sterownik dla wszelkich urządzeń podłączonych do interfejsu IDE. Dla różnych typów urządzeń istnieją różne „podsterowniki" współdziałające z *ide.c*, na przykład dla dysku twardego jest to *ide-disk.c*. Zarówno *ide.c* jak i jego podsterowniki mogą być ładowane jako moduły. Może on obsłużyć do czterech interejsów IDE, a co za tym idzie, do ośmiu napędów (być może różnego typu) do nich podłąconych.

Ide.c zakłada takie skojarzenie numerów głównych urządzeń z interfejsami IDE jakie przedstawia poniższa tabela.

Tablica 5.1: Przypisanie numerów głónych interfejsom IDE.

Interfejs	Numer urządzenia
ide0	3
ide1	22
ide2	33
ide3	34

Standardowo interfejsy te obsługiwane są przez przerwania numer 14 i 15.

Sterownik *ide-disk.c* częściowo przejął zadania sterownika hd.c, gdyż ten drugi nie radzi sobie z nowszymi typami dysków. Jeśli są one oba wkompilowane w jądro, to hd.c obsługuje dyski podłączone do interfejsu *ide0*, natomiast *ide-disk.c* – pozostałe. W takiej sytuacji starsze dyski mogą być podłączone do *ide0*, podczas gdy te nowsze powinny być podpięte na przykład pod *ide1*.

Konfigurowalne parametry sterownika *ide.c*

Istnieje możliwość ustawienia niektórych parametrów sterownika „ręcznie” w pliku *ide.h*, przed kompilacją. Parametry te to:

`INITIAL_MULT_COUNT` – początkowa wartość dla liczby sektorów transmitowanych podczas jednej operacji wejścia/wyjścia

`SUPPORT_SLOW_DATA_PORTS` – 1 oznacza wsparcie dla powolnych portów, ustawienie 0 powoduje zmniejszenie rozmiaru jądra

`SUPPORT_VLB_SYNC` – 1 oznacza wsparcie dla osobliwych 32-bitowych chipów, 0 redukuje rozmiar jądra

`DISK_RECOVERY_TIME` – 1 daje dyskom, które tego potrzebują pewne opóźnienie po dostępie do nich

`OK_TO_RESET_CONTROLLER` – do prawidłowej obsługi błędów powinno być ustawione na 1

`FAKE_FDISK_FOR_EZDRIVE` – 1 pomaga programowi fdisk przy EZDRIVE, 0 odchudza jądro

`FANCY_STATUS_DUMPS` – 1 sprawia, że wypisywane błędy mają postać czytelnych komunikatów, 0 jak zwykle powoduje zmniejszenie rozmiarów jądra

Są także w *ide.h* definicje stałych, których użytkownik nie powinien zmieniać. Oto kilka z nich:

`ERROR_MAX` – maksymalna dopuszczalna liczba błędów wejścia/wyjścia w jednym sektorze

`ERROR_RESET` – po której próbie odczytu/zapisu należy „zresetować” kontroler

`ERROR_RECAL` – po której próbie odczytu/zapisu należy zrekalibrować dysk

Najważniejsze struktury danych w *ide.c*.

Oto najważniejsze struktury danych, dotyczące urządzeń obsługiwanych przez *ide.c* oraz opis niekórych spośród ich pól. Są one wszystkie zadeklarowane w pliku *ide.h*.

`struct ide_drive_t` – struktura taka trzymana jest dla każdego dysku, jest ona połączona w listę cykliczną z innymi strukturami tego samego typu w obrębie jednej struktury `ide_hwgroup_t` (patrz dalej).
Opis niektórych pól:

- `*queue` – kolejka żądań do dysku
- `*next` – wskaźnik do kolejnej struktury `ide_drive_t`
- `service_start` – czas rozpoczęcia ostatniego żądania
- `service_time` – czas realizacji ostatniego żądania
- `special` – flagi do oznaczania specjalnych operacji na dysku
- `keep_settings` – zachowanie ustawień po „resecie”
- `using_dma` – czy dysk używa DMA
- `present` – czy dysk jest fizycznie obecny
- `media` – rodzaj napędu (disk, cdrom, ...)
- `mult_count` – aktualna liczba sektorów transmitowanych podczas jednej operacji wejścia/wyjścia
- `mult_req` – pożądana wartość dla powyższego parametru
- `usage` – licznik operacji `open()` dla tego dysku
- `head` – „faktyczna” liczba głowic
- `sect` – „faktyczna” liczba sektorów na ścieżce
- `cyl` – „faktyczna” liczba cylindrów
- `bios_head` – liczba głowic podana przez BIOS
- `bios_sect` – liczba sektorów podana przez BIOS
- `bios_cyl` – liczba cylindrów podana przez BIOS
- `*hwif` – wskaźnik do struktury `ide_hwif_t` (patrz dalej)
- `*id` – informacja o modelu napędu (`hd_driveid`)
- `*part` – tablica partycji dysku (`hd_struct`)
- `name[4]` – nazwa dyku, np.: „hda”
- `*driver` – wskaźnik do struktury sterownika `ide_driver_t` (patrz dalej)
- `*proc` – wskaźnik do katalogu /proc/ide/
- `[...]`

`struct ide_hwif_t` – struktura utrzymywana dla grupy urządzeń podłączonych do tego samego interfejsu IDE. Podobnie jak miało to miejsce w poprzedniej strukturze, te także połączone są w listę w obrębie `ide_hwgroup_t` (patrz dalej).
Opis niektórych pól:

*next – wskaźnik do kolejnej struktury ide_hwif_t

*hwgroup – wskaźnik do struktury ide_hwgroup_t (patrz dalej)

drives[MAX_DRIVES] – tablica, której elementami są struktury ide_drive_t (oczywiście MAX_DRIVES=2)

irq – numer przerwania obsługującego dany interfejs

major – numer główny urządzeń podłączonych do interfejsu

name[6] – nazwa interfejsu, np.: „*ide0*”

index – 0 dla *ide0*, 1 dla *ide1*, ...

sharing_irq – 1 oznacza, że interfejs dzieli jedno przerwanie z innym interfejsem

[...]

struct ide_hwgroup_t – struktura wspólna dla tych interfejsów IDE, które obsługiwane są tym samym przerwaniem.
Opis niektórych pól:

*handler – wskaźnik do procedury obsługi najbliższego przerwania

*drive – wskaźnik do struktury ide_drive_t odpowiadającej aktualnie obsługiwanemu dyskowi

*hwif – wskaźnik do struktury ide_hwif_t odpowiadającej aktualnie obsługiwanemu interfejsowi

*rq – aktualnie obsługiwane żądanie

timer – maksymalny czas oczekiwania na reakcję dysku

[...]

struct ide_driver_t – struktura opisująca „podsterownik” obsługujący konkretny typ urządzenia.
Opis niektórych pól:

*name – nazwa „podsterownika”, np.: „ide-disk”

*version – wersja

media – typ obsługiwanego urządzenia (disk, cdrom, ...)

*do_request – wskaźnik do procedury obsługi żądania

*end_request – wskaźnik do procedury kończącej obsługę żądania

*open – wskaźnik do procedury „otwierającej” urządzenie

*release – wskaźnik do procedury zwalniającej urządzenie

*special – wskaźnik do procedury obsługującej polecenia specjalne do dysku

[...]

ide_module_t – struktura opisująca jeden moduł.
Opis pól:

type – typ modułu

*init – wskaźnik do procedury inicjalizującej moduł

*info – dodatkowe informacje o module

*next – wskaźnik do kolejnej struktury ide_module_t

ide_hwifs[] – tablica, której elementami są struktury ide_hwif_t. Jest to główne repozytorium danych o wszystkim co dzieje się w *ide.c*.

*ide_modules – lista modułów.

W przypadku przez nas omawianym, czyli sterownika dysku twardego, struktura ide_driver_t zainicjowana jest następującymi wartościami:

```
name = „ide-disk”
version = IDEDISK_VERSION
media = ide_disk
do_request = do_rw_request
end_request = NULL
open = idedisk_open
release = idedisk_release
special = idedisk_special
[...]
```

NULL w polu end_request w praktyce oznacza, że procedurą kończącą obsługę żądania będzie ide_end_request() z *ide.c*.

Obsługa żądań

Komunikacja pomiędzy dyskiem twardym, a komputerem odbywa się za pomocą portów, którymi przesyłane są dane, oraz przerwań, jakie wysyła dysk na przykład aby powiadomić o zakończeniu wykonywania jakiegoś polecenia.

Funkcja add_request() (ll_rw_blk.c) po dodaniu nowego żądania wywołuje funkcję strategii odpowiednią dla urządzenia, a dokładnie jest to wywołanie request_fn() dla odpowiedniej pozycji w tablicy blk_dev[], której elementami są struktury blk_dev_struct. W przypadku naszego sterownika funkcją tą jest do_ideX_request(), gdzie X jest odpowiednim dla urządzenia numerem interfejsu IDE.

Uruchamiane są kolejno funkcje unlock_do_hwgroup_request(), do_hwgroup_request(), ide_do_request(), w których ustawiane są odpowiednie blokady, oraz wyznaczany jest dysk, do którego skierowane jest żądanie. Teraz wywoływana jest funkcja start_request(), która sprawdza jakiego rodzaju jest aktualnie obsługiwane żądanie i podejmuje odpowiednie kroki.
Możliwości są następujące:

1. żądanie jest poleceniem specjalnym dla dysku (zwykle są to polecenia wywoływane przez ioctl());

2. żądanie dotyczy ustawienia parametrów charakterystycznych dla tego konkretnego dysku (może to być na przykład żądanie sprawdzenia geometrii dysku – WIN_SPECIFY);

3. żądanie jest zwykłym poleceniem zapisu/odczytu.

W przypadku 1 wywoływane są funkcje `execute_drive_cmd()`, a z niej `ide_cmd()`, które zawiadamiają dysk o poleceniu, a następnie przy pomocy funkcji `ide_set_handler()` ustawiana jest procedura obsługi kolejnego przerwania jakie przyjdzie od dysku na funkcję `drive_cmd_intr()`.
W przypadku 2 wywoływana jest funkcja `do_special()`, a z niej `ide_disk_special()`, która jest już częścią „podsterownika" *ide-disk.c*. Dysk jest powiadamiany o poleceniu i tak jak poprzednio ustawiana jest procedura obsługi przerwania przy pomocy funkcji `ide_set_handler()`, lecz tym razem procedura ta jest jedną z trzech: `set_geometry_intr()`, `recal_intr()`, `set_multmode_intr()`.
W przypadku 3 wywoływana jest funkcja `do_rw_disk()` z *ide-disk.c*, która podobnie jak w punktach 1 i 2 używa `ide_set_handler()` do ustawienia procedury obsługi przerwania. Jest to dla polecenia odczytu funkcja `read_intr()`, a dla polecenia zapisu `write_intr()` lub `multwrite_intr()` (zapis więcej niż jednego sektora podczas jednej operacji). Jeśli polecenie nie jest ani komendą odczytu ani komendą zapisu, to `do_rw_disk()` kończy obsługe żądania wywołując funkcję `ide_end_request()`.

Obsługa przerwań

`drive_cmd_intr()`
Sprawdza, czy stan dysku po wykonaniu operacji jest prawidłowy (`OK_STAT()`). Jeśli tak, to kończy obsługe polecenia przez wywołanie funkcji `ide_end_drive_cmd()`, natomiast jeśli wystąpił jakiś błąd, to wywoływana jest `ide_error()`.

`set_multmode_intr()`
Jeżeli stan dysku jest prawidłowy (`OK_STAT()`), to funkcja ustawia żądaną liczbę jako ilość sektorów transmitowanych podczas jednej operacji „multwrite". W przeciwnym przypadku ustawia tę wartość na 0 oraz zaznacza flagę oznaczającą konieczność rekalibracji dysku.

`set_geometry_intr()`
Po ustawieniu geometrii dysku, sprawdza jego stan (`OK_STAT()`). Jeśli coś jest nie tak, to wywołuje funkcję `ide_error()`.

`recal_intr()`
Podobnie jak poprzednia funkcja, wywołuje `ide_error()` jeśli stan dysku po operacji rekalibracji jest nieprawidłowy.

`read_intr(), write_intr(), multwrite_intr()`
Dokonywana jest transmisja porcji danych, a następnie jeśli obsłużono już całe żądanie, to wywoływana jest funkcja `ide_end_request()`, w przeciwnym przypadku funkcja ustawia samą siebie do obsługi kolejnego przerwania. W przypadku wystąpienia jakiegoś błędu wywoływana jest funkcja `ide_error()`.

`reset_pollfunc()`
Funkcja ta po „zresetowaniu" dysku „odpytuje" go co 50ms czy jest już gotowy do pracy. Jest to konieczne, gdyż dysk nie wyśle żadnego przerwania by zasygnalizować swą gotowość.

Po wystąpieniu błędu najczęściej wywoływana jest funkcja `ide_error()`. Sprawdza ona, czy

liczba błędów podczas wykonywania aktualnego żądania przekroczyła dopuszczalny próg (`ERROR_MAX`). Jeśli tak, to obsługa żądania jest kończona (przez `ide_end_request()`), jeśli nie, to dysk jest „resetowany” (zajmuje się tym funkcja `ide_do_reset()`).
Na zakończenie obsługi żądania wywoływana jest funkcja `ide_end_request()`. Uaktualnia ona odpowiednie struktury danych tak, by sterownik mógł przystąpić do obsługi kolejnego żądania.

5.2.4 Szeregowanie zadań

W tym rozdziale zostanie omówione szeregowanie zadań skierowanych do urządzeń blokowych.

Wszystkie opisane funkcje zaimplementowane są w pliku */drivers/block/ll_rw_blk.c*. Oto ich nagłówki:

```
void ll_rw_block(int rw, int nr, struct buffer_head *bh[]) /*obsługa żądań*/
void make_request(int major, int rw, struct buffer_head *bh) /*stworzenie żądania*/
void add_request(struct blk_dev_struct * dev, struct request * req)/*umiejscowanie
 żądania w kolejce*/
```

Wszystkie żądania przechowywane są w statycznie zadeklarowanej tablicy `all_request` o rozmiarze zdeterminowanym przez stałą `NR_REQUST` pochodzącą z pliku *<linux/blk.h>*. W wersji 2.2.12 przyjmuje ona wartość 128. W tym momencie warto wspomnieć o tym, że Linux preferuje zadania dotyczące odczytu. Mogą być umiejscowione w całej tablicy. Zadania pisania mogą zajmować co najwyżej 2/3 zawartości tablicy `all_request`. Jeżeli w tablicy brakuje miejsca na nowe żądanie to proces jest wieszany na kolejce `wait_for_request`. Żądania zgrupowane są według numeru nadrzędnego urządzenia do którego są skierowane. W takich kolejkach posortowane są one według numeru podrzędnego urządzenia oraz sektora, którego dotyczą. Funkcja `void ll_rw_block()` obsługuje wszystkie zadania czytania i pisania z urządzeń blokowych. Sprawdza ona poprawność danych a następnie wywołuje funkcję `make_request()`, której przekazuje numer urządzenia, operację do wykonania, oraz nagłówek bufora. Przed stworzeniem nowego żądania funkcja `make_request()` przeszukuje kolejkę żądań do tego urządzenia. Jeżeli w kolejce znajduje się już żądanie dotyczące operacji czytania bądź pisania z sektorów położonych tuż przed aktualnie żądanym sektorem to żądanie z kolejki jest modyfikowane. Zwiększana jest mu liczba sektorów, które ma odczytać. Modyfikowane są również żądania dotyczące odczytu lub zapisu danych sektorów położonych bezpośrednio za ostatnim sektorem, który aktualnie chcemy odczytać. Jeżeli jednak żądanie dotyczy sektora, oddalonego od tych przechowywanych w kolejce to tworzone jest nowe żądanie. Zawiera ono między innymi informacje o urządzeniu, do którego jest skierowane, liczbie sektorów do odczytu/zapisu oraz dowiązanie do następnego żądania. Aby nowo stworzone żądanie umieścić na właściwym miejscu w kolejce żądań oczekujących wywoływana jest funkcja `add_request()`. Korzysta ona z makra `IN_ORDER` zdefiniowanego w pliku *<linux/blk.h>*.

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

Powyższe makro prawdza dla dwóch żądań czy pierwsze powinno się znaleźć przed drugim. Kolejka powinna więc być posortowana ze względu na numer podrzędny urządzenia, do którego skierowane jest żądanie i numeru sektora, którego dotyczy. Poniższy algorytm znajduje odpowiednie miejsce w kolejce, na które powinno zostać wstawione żądanie. Może się zdarzyć, że zostanie zgłoszone żądanie dotyczące urządzenie o numerze mniejszym od żądania aktualnie obsługiwanego.

Algorytm 1 add_request

Dane wejũciowe: struktura opisująca urządzenie oraz struktura opisująca żądanie
Wynik: umieszcza zgłoszone żądanie w odpowiednim miejscu w kolejce żądań oczekujących

```
tmp = pierwszy element kolejki
dopóki nie koniec kolejki wykonuj
  za_aktualne = Czy żądanie ma być wstawione za tmp ?
  przed_następnym = Czy żądanie ma być wstawione przed następnym żądaniem z kolejki?
  jeũli tmp powinno być przed tmp->next to
    jeũli za_aktualne i przed_nastepnym są prawdziwe to
      wyjdź z pętli
    koniec jeũli
  w przeciwnym przypadku
    jeũli za_aktualne lub przed_nastepnym są prawdziwe to
      wyjdź z pętli
    koniec jeũli
  koniec jeũli
  tmp = następny element z kolejki
koniec dopóki
w znalezione miejsce wstawiamy teraz przekazane żądanie
```

Wówczas takie żądanie wstawiane jest na koniec kolejki. W naszej kolejce powstają wówczas dwa ciągi rosnące: pierwszy o numerach sektorów lub urządzeń większych od aktualnie obsługiwanego urządzenia. Oraz drugi o numerach mniejszych. Jeżeli przyjdzie kolejne żądanie dotyczące sektora lub urządzenia o mniejszym podrzędnym numerze to zostanie ono wsortowane właśnie w tym drugim ciągu.

5.3 Urządzenia znakowe

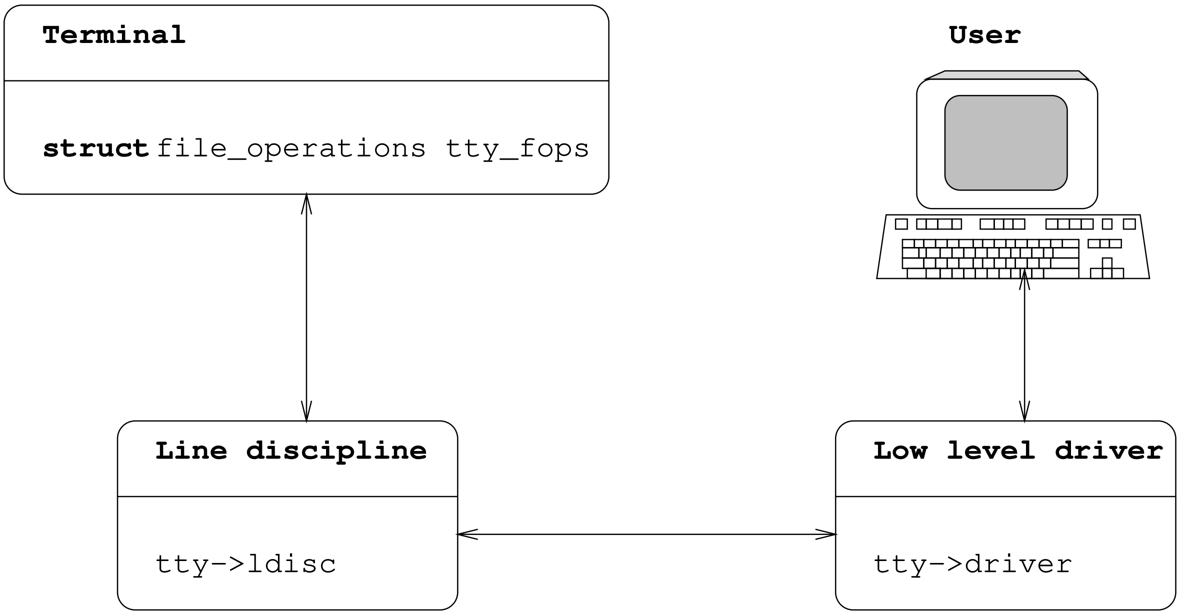
5.3.1 Terminal

Wśród urządzeń znakowych (i w ogóle wśród wszystkich urządzeń) najważniejszą rolę odgrywają terminale. Terminal jest abstrakcyjnym pojęciem. Z jednej strony jest to zwykłe urządzenie znakowe, do którego możemy czytać i pisać, a z drugiej strony jest on odpowiedzialny za komunikację systemu z użytkownikiem lub systemu z innymi systemem. Powszechnie znanym urządzeniem terminalowym jest konsola. Mniej znane natomiast są pseudoterminal oraz terminal szeregowy (mysz, modem, kabel szeregowy).

Działanie terminala

Terminal jest urządzeniem, a więc jest z nim związana struktura `file_operations`. W tej strukturze pamiętane są wskaźniki do funkcji systemowych. Jest to interfejs jądra do urządzenia terminalowego.

W celu porozumiewania się z użytkownikiem (bądź systemem) używamy terminala. Przy nawiązywaniu komunikacji wywołujemy odpowiednią funkcję systemową terminala. Schemat komunikacji terminala jest taki jak na rysunku 5.1. Najpierw terminal komunikuje się z dyscypliną



Rysunek 5.1: Schemat działania terminala

lini, którą możemy traktować jak filter pomiędzy terminalem, a sterownikiem niskopoziomowym. Zadania jakie wykonuje dyscyplina lini jest opisane w odpowiedniej sekcji. Dopiero sterownik niskopoziomowy komunikuje się bezpośrednio ze sprzętem.

Struktura `tty_struct`

W większości funkcji systemowych drugim parametrem jest `file`. Jest to wskaźnik na `struct file`. W polu `private_date` trzymane są prywatne dane dla sterownika urządzenia. W przypadku terminala trzymana tam jest struktura `tty_struct` zdefiniowana w pliku *include/linux/tty.h*. Poniżej przedstawione są niektóre pola tej struktury.

```
struct tty_driver driver;
struct tty_ldisc ldisc;
```

Pole `driver` to interfejs pomiędzy procedurami sterownika niskopoziomowego, a procedurami terminala. Pamiętane są w nim wskaźniki na funkcję sterownika niskopoziomowego. Pole `ldisc` to interfejs pomiędzy implementacją dyscypliny lini, a procedurami terminala. Pamiętane są w nim wskaźninki do funkcji dyscypliny lini.

```
struct termios * termios, * termios_locked;
```

W polu `termios` pamiętane są ustawienia urządzenia terminalowego, takie jak: sposób prztwarzania znaków wejściowych, znaków wyjściowych, definicje znaków specjalnych i inne parametry pracy. Pole `termios_locked` służy do zablokowania niektórych parametrów terminala. Ustawiane są w nim flagi, których zmiana w polu `termios` ma być zabroniona. Definicja struktury `termios` znajduje się w pliku *include/asm/termios.h*.

```
int pgrp;
int session;
```

Z terminalem związana jest **grupa terminalowa**. Są to wszystkie procesy, których numer grupy równa się polu `pgrp`. Terminal odpowiada jednej sesji, której numer wynosi `session`.

Ponadto w strukturze `tty_struct` pamiętane są bufory i inne potrzebne dane dla sterownika niskopoziomowego i dyscypliny lini. Ze względów historycznych w strukturze tej umieszczone są dodatkowo dane dla dyscypliny lini `N_TTY`.

Zadania terminala

Kod urządzenia terminalowego można znaleźć w pliku źródłowym *driver/char/tty_io.c*. Przedstawmy zarys tego co "dzieje się" w terminalu. Otwieranie terminala polega na zarezerwowaniu pamięci i zainicjowaniu wszystkich struktur. Czytanie i pisanie odwołuje się do odpowiednich procedur dyscypliny lini. W przypadku odebrania specjalnych znaków są rozsyłane sygnały do procesów z grupy terminalowej (`SIGINT` dla ^C, `SIGSTOP` dla ^Z). W przypadku zerwania połączenia z terminalem (np. użytkownik wylogował się, nastąpiło zerwanie transmisji przez łącze szeregowe) wysyłany jest sygnał `SIGHUP` do procesów związanych z danym terminalem. Ponadto wysyłane są także sygnały do informowania procesów o zmianie rozmiaru okna z prośbą o przerysowanie. Proces, który nie należy do danego terminala, przy próbie zapisu lub odczytu z niego otrzymuje odpowiednie sygnały o błędzie.

5.3.2 Konsola

Czymże jest konsola?

Jest to urządzenie znakowe odpowiadające za lokalną komunikację z użytkownikiem (na jednym komputerze). Po starcie systemu, jesli nie podłączamy się zdalnie do innego komputera, naszym terminalem staje się właśnie konsola. W Linuksie mamy 63 konsole, o numerze głównym 4 i pobocznych odpowiednio od 1 do 63 (tty1..63). Dzielą one ten sam ekran i dlatego są nazywane wirtualnymi (ang. virtual consoles). Tylko jedna z nich może być jednocześnie widoczna i oczywiscie też nie wszystkie sa włączone (bo niby po co, coś ma zajmować pamieć, jesli nie jest używane?).

O tym jak przełączać konsole i je tworzyć bedzie trochę dalej. Ta aktywna konsola odpowiada urządzeniu tty0. Isnieje jeszcze jedno urządzenie /dev/console.

W jądrach starszych niż 2.1.71 bylo ono po prostu linkiem do /dev/tty0, lub innego urządzenia np cu* albo tty*. Teraz już tak nie jest i /dev/console jest odrębnie obsługiwane przez jądro (nr główny 5, poboczny 1). Uwaga: jeśli nie mamy karty graficznej z frame buforem (nie ma urządzeń fb*), nie powinniśmy używać /dev/console, lecz /dev/tty0 (to już nie link !).

Obsługą konsoli zajmuje się driver urządzeń terminalowych (1 z 4 rodzajów dostępnych w Linuksie). Jest to najbardziej skomplikowany sterownik (z tych 4), ze względu na ilość obsługiwanych funkcji. Charakteryzuje się integracją ze sprzętem (potrafi np. zapalać, lampki na klawiaturze, a w trybie chronionym robi się to przez bezpośrednie programowanie kontrolera klawiatury. Realizacja oglądania tego co było wcześniej wyświetlone jest realizowana też niebanalnie.

Jedną z ważniejszych właściwości driverów konsoli jest emulacja terminali **VT100**. Oto jak to działa:

- jesli poślesz coś na konsolę, zostaje dokonana standartowa analiza charakterystyczna dla tty, następnie wynik jest przekazywany do drivera konsoli.
- driver przetwarza dane i analizuje sekwencje z kodami ESC (ruchy kursorów, czyszczenie ekranu, etc.)
- znaki, które nie są częścią sekwencji sterujących, są konwertowane na **UNICODE** przy użyciu 1 z 4 tablic mapujących (jeśli konsola nie była w trybie **UTF-8**)
- na podstawie uzyskanego kodu określany jest faktyczny kod znaku i przesyłany do pamieci karty graficznej.

Trochę inaczej ma się sprawa z odczytem z konsoli (klawiatura). Dzieje się to mniej więcej w ten sposób:

- po odczytaniu klawisza pojawia się przerwanie klawiatury, **scan code**, który otrzymaliśmy (być może wielobajtowy) jest konwertowany na postać przyjętą w Linuxie (16 bitów, ostatnie odpowiadają za typ klawisza (jest 16 typów))
- bierzemy typ klawisza, który został naciśniety, i wywołujemy jedną z 16 funkcji odpowiedzialnych za interpretację danego typu klawisza.

Za sprawą takiego rozwiązania, klawisze np. pod X-ami zachowują się inaczej niż na "gołej", tekstowej konsoli. Dzieki temu, Linux ukrywa też specjalne działanie niektórych klawiszy np **ALT-Fn** i **ALT-SysRQ-<key>** (o czym dalej).

Kolejna interesująca rzecz dotycząca konsoli, to **Magic SysRQ Key**. Pod tą magiczną nazwą kryje się zestaw dosyć złożonych komend uruchamianych przez kombinacje klawiszy **ALT-SysRQ-<key>** (SysRQ to najczesciej Print Screen na klawiaturze). Magic SysRQ dostępne jest (są??) dopiero w jądrach od 2.y.x gdzie y>=1, a x jest dowolne. Jak bardzo złożone funkcje mogą być wywoływane, możemy przekonać się przegladajac poniższy spis fajniejszych komend (pełny wykaz mozna znaleść w pliku *Documentation/sysrq.txt* - źrodła kernela)

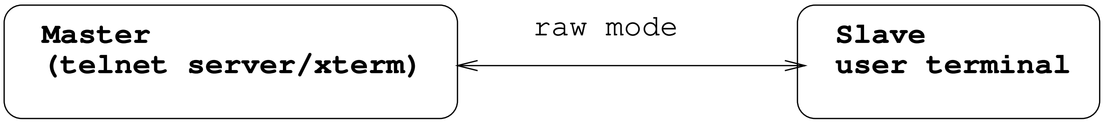
- 'k' - zabija wszystkie procesy na wirtualnej konsoli
- 'u' - remount wszystkich filesystemów
- 'p' - wywala aktualny stan wszystkich rejestrów i flag procesora
- 't' - wyświetla na konsoli wszystkie zadania aktualnie uruchomione w systemie (oj może być tego dużo)
- 'm' - przeróżne informacje o pamieci (naciśnij to zaraz ;-)

Można robić inne ciekawe rzeczy, np. wysłać SIGKILL do wszystkich procesów łącznie z init-em.

Obsługa Magicznych Klawiszy jest zaimplementowana w *linux/drivers/char/sysrq.c*

Jak jest realizowane przeglądanie wcześniejszych ekranów ?

(standartdowe klawisze: **Shift+ PgUp, PgDn**)



Rysunek 5.2: Pseudoterminal

Aby to zrobić, sterownik konsoli wykorzystuje możliwości tzw. **hard scrollingu**, dotępnego na kartach graficznych.. Ekrany widoczne poprzednio na konsoli, są pamiętane tylko w pamieci karty graficznej, a przegladanie ich (Shift +....) to poprostu zmiana adresu początku wyświetlanego obrazu. Z tego też powodu, jeżeli zapragniemy zmienić konsolę na inną, driver obsługujący konsolę wyczyści zawartość karty graficznej (żeby zachować sens działania **Shift+....**) i po powrocie na pierwotną konsole nie będziemy mogli obejrzeć tego, co poprzednio było wyświetlone.

Nie wiem, po co w ten sposób jest to zrobione, ale widocznie są jakieś powody....

Jak zmieniać i tworzyć konsole ?

Zmieniać można na kilka sposobów:

Alt-Fn lub Alt-Ctrl-Fn

Prawy ALT+Fn - zmieniamy na konsole o numerze n+12

Alt + strzałki w bok

programem chvt (**man chvt**)

Podane kombinacje dotyczą domyślnego mapowania klawiszy. Jak ktoś chce zmienić to **man loadkeys**

Teraz o tworzeniu:

tworzymy za pomocą **chvt.** Jesli chcemy usunąć to **deallocvt**. Na utworzonej konsoli można uruchomić program używając polecenia **openvt**. Opisy są w man-ie.

Źródła informacji:

- **Unicode** - jest to "zunifikowany 16-bitowy zestaw znakow". Zawiera znaki z prawie wszystkich języków. Wiecej (naprawde sporo) informacji mozna znaleść na stronach man-a **man unicode**
- **UTF-8** - kodowanie Unikodowe, kompatybilne z ascii. Patrz **man 7 utf-8** zainsteresowanych róznymi sztuczkami z klawiaturą i konsolą odsyłam do *kbd.FAQ.txt*
- *devices.txt* znajduje się lista wszystkich urządzeń w linuxie.
- no i oczywiscie zródła: *console.c*, *devices.txt console.h, consolemap.c, consolemap.h, console_struct.h*

5.3.3 Pseudoterminal

Pseudoterminal to urządzenie terminalowe wykorzystywane na przykład w programie `telnet` i `xterm`. Składa się z pary terminali nadrzędnego i podrzędnego połączonych dwukierunkowym pipem. Każdy z terminali reprezentuje jeden z dwóch końców połączenia. W strukturze `tty_struct` znajduje się pole `link`, które to jest wskaźnikiem na drugiego członka pary. Pseudoterminale są stworzone na przykład po to, aby komunikacja mogła się odbywać przy użyciu dwóch różnych dyscyplin lini, co jest niemożliwe w przypadku zwykłych terminali. Przy operacjach czytania

/ pisania z jednego terminala (np. czytanie z klawiatury / wyświetlanie na ekranie) stosowana jest odpowiadająca mu dyscyplina lini i sterownik niskopoziomowy. Na przykład przy programie `telnet` ze strony użytkownika do komunikacji z nim stosowany jest terminal z dyscypliną lini `N_TTY`, a ze strony połączenia sieciowego (np. PPP) może być stosowany terminal z dyscypliną lini `N_PPP`, wtedy jeden terminal reprezentuje użytkonika, a drugi połączenie sieciowe. Analogicznie w przypadku programu `xterm` jeden terminal reprezentuje użytkownika, a drugi `Xserver`.

5.3.4 Dyscyplina linii

Dyscyplina linii polega na tłumaczeniu znaków przesyłanych przez terminal do procesu. Jest to przydatne, aby mieć pewien standardowy sposób reagowania procesu na wejście (np. Ctrl-Z zatrymanie procesu).

Interfejs dyscypliny linii zdefiniowany jest w pliku *include/linux/tty_ ldisc.h*

Wersja 2.2.x jądra wyróżnia 2 dyscypliny linii, są to:

- **N_TTY**
- **N_HDLC** - używana przez urządzenia komunikacjyjne, np modemy. Jest zorientowana na przesyłanie całych ramek (*frames*), tzn można wysłać lub odebrać pełną ramkę. Jej cechą jest to, że nie wykonuje żadnej konwersji.

Dyscyplina **N_TTY** obsługuje następujące konwersje:

Plik *include/linux/tty_ ldisc.h*
Definicja interfejsu pomiędzy *tty_ discipline* a procedurami obsługi *tty*

```
#ifndef _LINUX_TTY_LDISC_H
#define _LINUX_TTY_LDISC_H
...
#include <linux/fs.h>
#include <linux/wait.h>

struct tty_ldisc {
   int magic;
   char *name;
   int num;
   int flags;
};
```

Funkcje wywoływane "z góry", czyli przez użytkownika

```
int (*open)(struct tty_struct *);
```

/* Funkcja wywoływana przy przypisywaniu dyscypliny linii do urządzenia *tty.* Dyscyplina linii wykorzystuje ją do inicjalizacji stanu. */

```
void (*close)(struct tty_struct *);
```

/* Funkcja wywoływana przy zamykaniu dyscypliny linii */

```
void (*flush_buffer)(struct tty_struct *tty);
```

/* Funkcja czyści bufor wejściowy **tty* */

```
ssize_t (*chars_in_buffer)(struct tty_struct {*}tty);
```

/* zwraca liczbę znaków w buforze */

```
ssize_t (*read)(struct tty_struct * tty, struct file * file,
                unsigned char * buf, size_t nr);
```

/* Funkcja wywoływana przy żądaniu odczytu przez użytkownika. Funkcja odpowiada, czy dyscyplina ma w buforze jakieś znaki. Jeśli nie jest zdefiniowana, to zwraca błąd EIO */

```
ssize_t (*write)(struct tty_struct * tty, struct file * file,
                  const unsigned char * buf, size_t nr);
```

/* Funkcja wywoływana przy przy żądaniu zapisu przez użytkownika. Dostarcza znaki strownikowi niskopoziomowemu, ewentualnie dokonując konwersji. Jeśli nie zdefiniowana to zwraca błąd EIO */

```
int (*ioctl)(struct tty_struct * tty, struct file * file,
             unsigned int cmd, unsigned long arg);
```

/* Funkcja wykonywana przy operacji *ioctl* jeśli nie udało się jej zrealizować w poprzedniej warstwie. Porządek warstw jest następujący (1) *tty* (2) sterownik niskopoziomowy *tty* (3) dyscplina linii */

```
void    (*set_termios)(struct tty_struct *tty,
                       struct termios * old);
```

/* Funkcja powiadamia dyscyplinę linii, że nastąpiła zmiana w strukturze *termios* */

```
unsigned int (*poll)(struct tty_struct *, struct file *,
                     struct poll_table_struct *);
```

/* Funkcja wykonywana, gdy użytkownik przepytuje (praca w trybie *pooling*) urządzenie */

Funkcje wywoływane "z dołu" - przez sterownik niskopoziomowy

```
void (*receive_buf)(struct tty_struct *, const unsigned char *cp,
                    char *fp, int count);
```

/* Funkcja wywoływana przez sterownik niskopoziomowy do wysłania znaków otrzymanych od sprzętu do dyscypliny linii. *cp* jest wskaźnikiem na bufor wejścia, a *fp* wskaźnikiem na wskaźnik bajtów flag, które wskazują na to czy nastąpił błąd parzystości itp. */

```
int    (*receive_room)(struct tty_struct *);
```

/* Funkcja wywoływana przez sterownik niskopoziomowy, do stwierdzenia rozmiaru bufora dyscypliny linii */

```
void (*write_wakeup)(struct tty_struct *);
```

/* Funkcja wywoływana przez sterownik niskopoziomowy, by powiadomić dyscyplinę linii, że ta może wysłać kolejne znaki. Jeśli dyscyplina ma pusty bufor to nic nie robi */

```
};

#define TTY_LDISC_MAGIC 0x5403

#define LDISC_FLAG_DEFINED 0x00000001

#endif /* _LINUX_TTY_LDISC_H */
```

5.3.5 Urządzenie mem

Wstęp

Pamięć jest dostępna jako urzadzenie znakowe o numerze głównym 1.

Wyróżniamy kilka podurządzeń mem:

Podnumer(minor)	lokalizacja w systemie plików	cel
1	/dev/mem	pamięć fizyczna
2	/dev/kmem	pamięć wirtualna widziana przez jądro
3	/dev/null	czarna dziura , wszystko tam wrzucone przepada
4	/dev/port	dostęp do portów
5	/dev/zero	generator zer
6	/dev/core	
7	/dev/full	urzadzenie zwracajace ENOSPC przy próbie zapisu
8	/dev/random	generator liczb pseudolosowych
9	/dev/urandom	j.w. tyle że nie blokujące

mem i kmem

Na /dev/mem i /dev/kmem można wykonywać operacje:

- open
- write
- read
- lseek
- map

w takim razie tablica rozdzielcza dla (k)mem wyglada:

```
static struct file_operations (k)mem_fops = {
           memory_lseek,
           read_(k)mem,
           write_(k)mem,
           NULL,  /* mem_readdir */
           NULL,  /* mem_poll */
           NULL,  /* mem_ioctl */
           mmap_(k)mem,
           open_(k)mem,
           NULL,  /* flush */
           NULL,  /* no special release code */
           NULL  /{*} fsync {*}/
};
```

Funkcje

Funkcje obsługi mem są zaimplementowane w pliku */usr/scr/linux/drivers/char/mem.c.*

Do najważniejszych można zaliczyć :

```
static int open_port(struct inode * inode, struct file * filp)
```

-służy do otwierania zarówno io jaki i mem,kmem jej działanie sprowadza się do wywołania makra

```
capable(CAP_SYS_RAWIO == 17) (defincja w capabilty.h)
```

które sprawdzania czy można skorzystać z IO. Definicja: **capable**(x) w *linux/sched.h*

```
static ssize_t do_write_mem(struct file * file, void *p, unsigned long realp,
                            const char * buf, size_t count, loff_t *ppos)
```

-wykonanie właściwego zapisu do pamięci

```
static ssize_t read_mem(struct file * file, char * buf,
                        size_t count, loff_t *ppos)
```

-czytanie z pamięci

```
static ssize_t write_mem(struct file * file, const char * buf,
                         size_t count, loff_t *ppos)
```

-sprawdza parametry i woła odpowiednio do_write_mem

```
static int mmap_mem(struct file * file, struct vm_area_struct * vma)
```

- zamapowanie urzadzenia do przestrzeni adresowej użytkownika */dev/mem* i */dev/kmem* są wyjatkiem od reguły, że urządzenia znakowe nie potrzebuja *mmap.*

Mogą być mapowane ponieważ są skończone (mają określony rozmiar) i dają się adresować.

W powyższych funkcjach parametr file nie jest wykorzystywany. Funkcje **read_mem** i **write_mem** mają swoje odpowiedniki **read_kmem** i **write_kmem** dla *kmem.*

```
static loff_t memory_lseek(struct file * file, loff_t offset, int orig)
```

uwaga: nie można przesuwać względem końca pamięci

Zmienne globalne i makra

Jedną z zmiennych globalnych często wykorzystywaną w powyższych funkcjach to **high_memory** (wskazuje poczatek pamieci) jest zadeklarowana w *include/linux/mm.h*

```
void * high_memory = NULL;
```

Makra zdefiniowne w *asm/uaccess.h*

```
copy_from_user
```

- kopiowania z obszaru użytkownika do pamieci jądra

```
copy_to_user
```

- na odwrót
oraz

```
__pa(high_memory)
__va(unsigned long)
```

do dostępu do pamieci fizycznej są zdefiniowane w *asm/page.h*

```
#define __PAGE_OFFSET         (PAGE_OFFSET_RAW)
#define PAGE_OFFSET           ((unsigned long)__PAGE_OFFSET)
#define __pa(x)               ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x)               ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

PAGE_OFFSET_RAW (zależne od Max rozmiaru RAM-u) (*asm/page_ offset.h*)

```
#ifdef CONFIG_1GB #define PAGE_OFFSET_RAW 0xC0000000
#elif defined(CONFIG_2GB)
#define PAGE_OFFSET_RAW 0x80000000
#elif defined(CONFIG_3GB)
#define PAGE_OFFSET_RAW 0x40000000
#endif
```

Różnice między mem a kmem

Główna różnice pomiędzy obsługa mem i kmem polega na dodatkowym przeliczanie adresów w przypadku mem przez makra **__pa**,**__va** ;

Przykład write_kmem i write_mem

```
static ssize_t write_kmem(struct file * file, const char * buf,
                          size_t count, loff_t *ppos)
{ unsigned long p = *ppos;
  if (p >= (unsigned long) high_memory)
         return 0;
  if (count > (unsigned long) high_memory - p)
         count = (unsigned long) high_memory - p;
  return do_write_mem(file, (void*)p, p, buf, count, ppos);
}

static ssize_t write_mem(struct file * file, const char * buf,
                         size_t count, loff_t *ppos)
{ unsigned long p = *ppos;
  unsigned long end_mem;
  end_mem = __pa(high_memory);
  if (p >= end_mem)
         return 0;
  if (count > end_mem - p)
         count = end_mem - p;
  return do_write_mem(file, __va(p), p, buf, count, ppos);
}
```

5.4 Jak napisać program obsługi urządzenia

5.4.1 Wiadomości wstępne

Inicjowanie programu obsługi urządzenia

Aby można było korzystać z naszego programu obsługi urządzenia musimy go najpierw zainicjować. Standardowo funkcja inicjująca odpowiada za wykrycie urządzenia, jego rejestrację w jądrze (rejestracja jest omówiona w kolejnych rozdziałach) i stworzenie wewnętrznych struktur danych programu. Jeśli nasz program obsługi ma być wkompilowany w jądro tworzymy funkcję:

```
int init_mydrv (void)
```

i dopisujemy jej wywołanie do jednego z poniżej wypisanych plików w zależności od typu programu obsługi urządzenia:

- `drivers/char/mem.c` - dla znakowych (funkcja `char_dev_init`)
- `drivers/block/ll_rw_bl.c` - dla blokowych (funkcja `blk_dev_init`)

W przypadku gdy program obsługi urządzenia ma być modułem, jego inicjalizacja odbywa się w funkcji inicjującej moduł:

```
int init_module (void)
```

Odczyt parametrów

Czasami inicjalizację programu obsługi urządzenia trzeba sparametryzować dla poprawnego działania na danym komputerze. Dla programu wkompilowanego w jądro możemy podać parametry w czasie startu systemu (np. przez linię poleceń LILO) w formacie `drvname=param1,...,paramN`. Ponieważ analizą linii parametrów dla jądra zajmują się funkcje z pliku `<init/main.c>` (`parse_options`, `checksetup`, `get_options`), program obsługi urządzenia nie musi się już tym zajmować. Chcąc odczytać parametry przesłane do naszego programu musimy napisać funkcję:

```
int mydrv_setup (char *s, int *p)
```

a nastepnie "poinformować" o niej jądro. Funkcja ta będzie wywołana przed inicjalizacją (`init_mydrv()`), więc na podstawie przekazanych do niej parametrów można ustalić późniejsze zachowanie programu obsługi. Parametr `p` to tablica zawierająca parametry przekazane do naszego programu, które udało się zamienić na liczby (próba zamiany dotyczy pierwszych dziesięciu parametrów), a ciąg `s` składa się z pozostałych argumentów. Rozmiar tablicy `p` znajduje się w `p[0]`. Aby jądro "wiedziało" o funkcji `mydrv_setup` musimy ją dopisać do pliku `<init/main.c>` w `cooked_params`:

```
static struct kernel_param cooked_params[] __initdata = {
    ......
    {"mydrv=", mydrv_setup },
    ......
    {0, 0}
};
```

gdzie,

```
struct kernel_param {
    const char *str;
    void (*setup_func)(char *, int *);
};
```

Istnieje jeszcze tablica `raw_params[]` (oprócz `cooked_params[]`) i dla jej programów obsługi funkcja `checksetup` nie próbuje zamieniać początkowych parametrów na liczby i zamiast tablicy `p` przesyła wartość `NULL`.

Ustawienie parametrów dla programu w postaci modułu można wykonać podczas jego ładowania (`insmod moduł nazwa_param1=wartość_param1...`). Zadeklarowanie zmiennej jako parametru odbywa się za pomocą makra:

```
MODULE_PARM(variable, type-description);
```

gdzie `variable` to nazwa zmiennej a `type-description` jej opis. Jak opisać zmienną jest udokumentowane w pliku `<linux/module.h>` (plik nagłówkowy konieczny do dołączenia jeśli piszemy moduł). Przykład zastosowania:

```
#include <linux/module.h>
......
int param1;
MODULE_PARM(param1, "i");        // "i" to liczba całkowita
```

Dostęp do przestrzeni użytkownika

W wielu funkcjach programu obsługi urządzenia musimy mieć dostęp do przestrzeni użytkownika, który wywołał daną funkcję. Tak będzie naprzykład, kiedy program otrzyma zlecenie odczytania danych z urządzenia i umieszczenia ich w buforze użytkownika. Trzeba pamiętać, że wskaźnik do bufora jest wtedy adresem z przestrzeni procesu użytkownika. Ponadto należy także sprawdzić czy proces ma prawa do obszaru pamięci na który wskazuje przekazany do programu obsługi urządzenia wskaźnik. Chcąc korzystać z interfejsu jądra do uzyskiwania dostępu do pamięci użytkownika musimy dołączyć plik `<asm/uaccess.h>`. A oto zbiór dostępnych funkcji:

- `int access_ok (int type, unsigned long addr, unsigned long size)`
  Parametr `type` może przybierać wartości `VERIFY_READ` lub `VERIFY_WRITE`, `addr` to adres początku sprawdzanego bloku pamieci, a `size` jego rozmiar. Funkcja zwraca 1, jeżeli bieżący proces ma dostęp do pamięci i 0 w przeciwnym przypadku. Wykorzystywana przez inne funkcje i dlatego rzadko wołana bezpośrednio.

- `int get_user (lvalue, address)`
  `int __get_user(lvalue, address)`
  Makro. Przypisanie na `lvalue` wartości spod adresu `address`. Zwraca 0 w przypadku powodzenia lub ujemny kod błędu. Druga funkcja nie wywołuje wewnętrznie `access_ok`.

- `int put_user (expression, address)`
  `int __put_user (expression, address)`
  Zapisanie wartości `expression` pod adres `address`.

- `unsigned long copy_from_user (unsigned long to, unsigned long from, unsigned long len)` i odpowiednik `__copy_from_user`
  Kopiowanie danych z przestrzeni uzytkownika do przestrzeni jądra. Jeśli zwróci 0 to się powiodło.

- `unsigned long copy_to_user (unsigned long to, unsigned long from, unsigned long len)` i `__copy_to_user`
  Kopiowanie danych z przestrzeni jądra do przestrzeni użytkownika.

Alokacja pamięci

Do dynamicznej alokacji i dealokacji pamięci w jądrze mamy dostępne funkcje:

```
void *kmalloc (size_t size, int flags)
void kfree (const void *objp)
```

Parametr `size` to rozmiar pamięci. Ponieważ alokowane są całe strony pamięci (w Intelu 4096 bajtów na stronę) to zazwyczaj zwrócony zostanie obszar pamięci o rozmiarze większym niż żądaliśmy. Ponadto jądro z alokowanej przez nas pamięci rezerwuje sobie część na nagłówek i dlatego chcąc zarezerwować 4kb pamięci powinniśmy za parametr `size` podać raczej 4000 niż 4096, gdyż w drugim przypadku zostaną zajęte dwie strony (8kb). Za pomocą `kmalloc` można zaalokować maksymalnie 32 strony pamięci (w rzeczywistości trochę mniej niż 128kb). Drugi parametr, `flags` określa rodzaj rezerwowanej pamięci. Możliwe wartości parametru są opisane w pliku `<linux/mm.h>` (stałe `GFP_*`). Najczęściej wystarczającą wartością będzie `GFP_KERNEL`. Wtedy w przypadku braku pamięci jądro wstrzymuje proces działający aktualnie w przestrzeni jądra i "czeka" na zwolnienie jakiejś strony. Oczywiście inny proces dostaje wtedy czas procesora. Jest to niebezpieczne kiedy alokujemy pamięć podczas obsługi przerwania, gdyż możemy zablokować dowolny proces, który w chwili zgłoszenia przerwania miał procesor. W takim przypadku należy użyć `GFP_ATOMIC` i zbadać czy operacja `kmalloc` się powiodła (oczywiście badanie poprawności wykonania funkcji `kmalloc` jest zawsze dobrym pomysłem). Dodatkowo można jeszcze użyć flagi `GFP_DMA` jeśli dany obszar będzie służył do transmisji DMA.

Dla alokacji większych obszarów pamięci można użyć funkcji:

```
void *vmalloc (unsigned long size)
void vfree (void *addr)
```

Funkcja `vmalloc` alokuje pamięć z *wirtualnej* przestrzeni adresowej i dlatego w szczególności nie nadaje się do alokacji obszarów pamięci służących do transmisji DMA.

5.4.2 Jak napisać program obsługi urządzenia znakowego

Rejestracja

Podczas inicjalizacji naszego programu obsługi urządzenia znakowego powinniśmy przypisać mu numer nadrzędny i zarejestrować go w jądrze (w tablicy rozdzielczej). Funkcją używaną do rejestrowania programu obsługi urządzenia znakowego jest funkcja:

```
int register_chrdev (unsigned int major, const char *name, struct file_operations *fops);
```

zdefiniowana w pliku nagłówkowym `<linux/fs.h>`. Argument `major` jest numerem nadrzędnym (major device number) pod jakim chcemy zarejestrować nasz program, `name` nazwą urządzenia, a `fops` wskaźnikiem do struktury operacji plikowych (`struct file_operations` z `<linux/fs.h>`) omówionej w dalszej części rozdziału. Jeśli funkcja zwróci liczbę równą lub większą od zera to rejestracja się powiodła, w przeciwnym wypadku wystąpił błąd. Dynamiczne przydzielenie numeru nadrzędnego naszemu programowi uzyskamy podając zero jako pierwszy argument funkcji (`major`). Wtedy dodatni wynik funkcji jest przydzielonym numerem nadrzędnym. Jeśli piszemy nasz program obsługi jako moduł mamy także dostępną funkcję zwalniającą numer nadrzędny do puli wolnych numerów:

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

Argumentami są tu numer nadrzędny, który ma być zwolniony i nazwa związanego z nim urządzenia. Jeśli podana nazwa jest różna od nazwy zarejestrowanej pod danym numerem lub podany numer wykracza poza dostępny zakres zwracana jest wartość `EINVAL`.

Operacje plikowe

W funkcji rejestrującej program obsługi urządzenia musieliśmy podać wskaźnik do struktury operacji plikowych `struct file_operations`. Struktura ta zawiera wskaźniki do funkcji jakie może obsługiwać nasz program, bądź wartości NULL dla funkcji które nie są obsługiwane. Przykładowa struktura może wyglądać tak:

```
struct file_operations testdrv_fops {
     testdrv_llseek,     /* llseek */
     testdrv_read,       /* read */
     testdrv_write,      /* write */
     NULL,               /* readdir */
     NULL,               /* poll */
     testdrv_ioctl,      /* ioctl */
     NULL,               /* mmap */
     testdrv_open,       /* open */
     NULL,               /* flush */
     testdrv_release,    /* release */
     NULL,               /* fsync */
     NULL,               /* fasync */
     NULL,               /* check_media_change */
     NULL,               /* revalidate */
     NULL                /* lock */
};
```

A zatem zdecydowaliśmy się zaimplementować funkcje: `testdrv_llseek`, `testdrv_read`, `testdrv_write`, `testdrv_ioctl`, `testdrv_open`, `testdrv_release`.

Struktura file

Struktura `struct file` zdefiniowana w pliku `<linux/fs.h>` jest tworzona za pomoca funkcji `open` jądra i przekazywana do wszelkich operacji działających na pliku. Oto jej najważniejsze elementy (te które mają znaczenie dla programów obsługi urządzeń):

- `struct file_operations *f_op;`
  operacje zwiazane z plikiem; Pole to jest wypelniane w momencie wywołania funkcji urządzenia open. Możemy je modyfikoważ w trakcie pracy urządzenia, powodując tym samym zmianę operacji plikowych. Na przykład funkcja open dla urządzenia o numerze nadrzednym 1 zastępuje `f_op` w zależnosci od numeru drugorzędnego. Dzięki temu jeden program obsługuje kilka urządzeń (`/dev/mem, /dev/null, /dev/zero, /dev/full`)

- `mode_t f_mode;`
  tryb otwarcia pliku; Jest identyfikowany przez bity `FMODE_READ i FMODE_WRITE`

- loff_t f_pos;
  bieżąca pozycja czytania lub pisania

- `unsigned int f_flags;`
  flagi; najczęściej używane to `O_RDONLY, O_NONBLOCK`.

- `void *private_data;`
  Pole to program obsługi może wykorzystać do własnych (specyficznych) celów. Funkcja systemowa open ustawia to pole na `NULL`, więc jeśli chcemy je wykorzystać należy zaalokować pamięć w naszej funkcji open.

Otwieranie i zamykanie urządzenia

```
int (*testdrv_open) (struct inode * inode, struct file *filp);
int (*testdrv_release) (struct inode *inode, struct file *filp);
```

Funkcja open służy programowi obsługi do wstępnego przygotowania dla kolejnych operacji. Zazwyczaj powinna wykonywać następujące działania:

- Sprawdzenie błędów specyficznych dla urządzenia
- Inicjacja urządzenia, jeśli jest otwierane po raz pierwszy
- Alokacja i wypełnienie struktur danych, które mają być umieszczone w `filp->private_data`
- Zwiększenie licznika wykorzystania za pomocą makra `MOD_INC_USE_COUNT`

Funkcja release powinna realizować:

- Zmniejszenie licznika wykorzystania za pomocą makra `MOD_DEC_USE_COUNT`
- Zdealokowanie wszystkiego co zaalokowano w open w `filp->private_data`
- Wyłączyć urządzenie przy ostatnim wywołaniu release

Czytanie i pisanie do/z urządzenia

```
ssize_t (*read) (struct file *filp, char *buf, size_t size, loff_t *loff);
ssize_t (*write) (struct file *filp, const char *buf, size_t size, loff_t *loff);
```

Funkcje powinny implementować odpowiednio czytanie danych z urządzenia do bufora `buf` oraz pisanie z bufora `buf` do urządznia. Funkcje mają zwracać ile danych udało się wczytać/zapisać lub wartość ujemną w przypadku błędu. Zwrócenie zera w przypadku read oznacza że nastapił koniec pliku.

Poniewaz read i write przesyłają dane między urządzeniem (przestrzeń jądra) i aplikacjami (przestrzeń użytkownika) nie można do kopiowania użyć zwykłych funkcji takich jak memcpy. Do tego celu mamy makra zdefiniowane w pliku <asm/uaccess.h>

```
unsigned long copy_from_user(unsigned long to, unsigned long from, unsigned long len);
unsigned long copy_to_user(unsigned long to, unsigned long from, unsigned long len);
```

opisanych wcześniej w wiadomościach wstępnych.

Funkcja kontroli urządzenia

```
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
```

Zaimplementowanie tej metody pozwoli użytkownikowi urządzenia na dodatkowe, specyficzne tylko dla tego urządzenia operacje. Parametr `cmd` jest numerem komendy a `arg` jej argumentem. Należy wybrać unikatowe numery komend aby uniemożliwić wydanie poprawnej komendy błędnemu urządzeniu. W przypadku niepoprawnego numeru `cmd` funkcja powinna zwrócić wartość `EINVAL`. W celu wybrania unikatowych numerów komend należy skorzystac z następujących makrodefinicji:

```
_IO(type, nr)
_IOR(type, nr, size)
_IOW(type, nr, size)
_IOWR(type, nr, size)
```

Pierwszej z nich używamy gdy komenda nie przesyła żadnych danych pomiędzy przestrzeniami jądra i użytkownika. Drugiej jeśli komenda przesyła dane do przestrzeni użytkownika, trzeciej w przeciwnym przypadku, a czwartej gdy wymiana danych następuje w obydwie strony. Kolejne parametry oznaczają:

- `type` - **numer magiczny** (ang. *magic number*), liczba ośmiobitowa którą należy wybrać i używać we wszystkich makrach. Spis zarezerwowanych numerów magicznych (nie należy ich używać) znajduje się w pliku <Documentation/ioctl-number.txt>.
- `nr` - liczba ośmiobitowa (po prostu kolejna liczba naturalna)
- `size` - typ danych, które będą przesyłane

Oto przyklad wycięty z pliku <include/linux/random.h>

```
/* Get the entropy count. */
#define RNDGETENTCNT _IOR( 'R', 0x00, int )

/* Add to (or subtract from) the entropy count.  (Superuser only.) */
#define RNDADDTOENTCNT _IOW( 'R', 0x01, int )

/* Get the contents of the entropy pool.  (Superuser only.) */
#define RNDGETPOOL _IOR( 'R', 0x02, int [2] )
```

W podanym przykładzie numerem magicznym jest 'R'. Nalezy zwrócić uwagę, że jako `size` podajemy typ danych, a nie `sizeof(typ)`.

5.4.3 Jak napisać program obsługi urządzenia blokowego

Rejestracja

Podobnie jak miało to miejsce w przypadku urządzeń znakowych, podczas inicjalizacji programu obsługi urządzenia blokowego powinniśmy przypisać mu numer nadrzędny. Służy do tego funkcja:

```
    int register_blkdev (unsigned int major, const char *name,
struct file_operations *fops)
```

Analogicznie do funkcji `register_chrdev` argument `major` jest numerem nadrzędnym pod jakim chcemy zarejestrować program, `name` nazwą urządzenia, a `fops` wskaźnikiem do struktury operacji plikowych `struct file_operations`. Dynamiczną alokacje numeru nadrzędnego uzyskuje się przez podanie za parametr `major` liczby `0`. Dla programów w postaci modułu dostępna jest także funkcja zwalniającą urządzenie:

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

Operacje plikowe

W funkcji rejestrującej program obsługi urządzenia musieliśmy podać wskaźnik do struktury operacji plikowych. Teraz zajmiemy się wypełnieniem tej struktury dla przykładowego programu obsługi urządzenia blokowego (`mydrv`):

```
struct file_operations mydrv_fops {
    NULL,                      /* llseek */
    block_read,                /* read */
    block_write,               /* write */
    NULL,                      /* readdir */
    NULL,                      /* poll */
    mydrv_ioctl,               /* ioctl */
    NULL,                      /* mmap */
    mydrv_open,                /* open */
    NULL,                      /* flush */
    mydrv_release,             /* release */
    block_fsync,               /* fsync */
    NULL,                      /* fasync */
    mydrv_check_media_change,  /* check_media_change */
    mydrv_revalidate,          /* revalidate */
    NULL                       /* lock */
};
```

W pola wskaźników do funkcji, których nasz program nie obsługuje wstawiamy wartość `NULL`. W powyższym przykładzie funkcje, które sami implementujemy mają w nazwie prefiks `mydrv`. Tak naprawdę możemy nie implementować żadnej z nich, chociaż najczęsciej nie obędziemy się jednak bez własnych implementacji `open` i `release`. Najważniejszą rzeczą w powyższej strukturze operacji plikowych jest jednak to, iż nie podaliśmy w niej własnych funkcji czytania i pisania do urządzenia. Programy obsługi urządzeń blokowych zazwyczaj nie implementują własnych funkcji `read` i `write`. Zamiast tego używają standardowych, wspólnych dla wszystkich urządzeń blokowych funkcji `block_read`, `block_write` (także `block_fsync`). Funkcje te korzystają z mechanizmu podręcznej pamięci buforowej (została ona opisana w rozdziale dotyczącym systemu plików) i nigdy nie odwołują się bezpośrednio do urządzenia, więc nie muszą znać sposobu jego działania. Pozostaje pytanie co zrobić, gdy bufory nie są w stanie obsłużyć żądania. W takim przypadku jądro zleca programowi obsługi operację rzeczywistego przesłania danych. Jak napisać funkcję obsługującą takie zlecenie jest napisane w dalszej części rozdziału.

Dodatkowe struktury danych dla urządzeń blokowych

Każde urządzenie blokowe opisane jest przez strukturę `blk_dev_struct` zdefiniowaną w pliku `<linux/blkdev.h>`:

```
struct blk_dev_struct {
    request_fn_proc        *request_fn;
    queue_proc             *queue;
    void                   *data;
    struct request         *current_request;
```

```
        struct request   plug;
        struct tq_struct plug_tq;
    };
```

gdzie,

```
    typedef void (request_fn_proc) (void);
```

Jądro utrzymuje tablicę tych struktur (indeksowaną numerami nadrzędnymi):

```
    extern struct blk_dev_struct blk_dev[MAX_BLKDEV];
```

Pisząc program obsługi urządzenia blokowego musimy wypełnić pole `request_fn`. Jest to wskaźnik na tak zwaną *procedurę strategii*. To właśnie w niej są zaimplementowane lub zlecane niskopoziomowe operacje specyficzne dla danego urządzenia. Gdy jądro zleca wykonanie operacji czytania/pisania wywołuje po prostu funkcję `blk_dev[major].request_fn`.

Inne dodatkowe struktury danych to:

- `int *blk_size [MAX_BLKDEV]`
  Opisuje rozmiar każdego urządzenia w kilobajtach. Pierwszy indeks to numer nadrzędny urządzenia, a drugi to numer drugorzędny. Jeśli w programie nie ustawimy tej wartości to nie będzie wykonywane przez jądro sprawdzanie rozmiaru urządzenia przy zleceniach.
- `int *blksize_size [MAX_BLKDEV]`
  Rozmiar jednego bloku urządzenia w bajtach. Tablica jest indeksowania analogiczne. Wartością domyślną jest 1024.
- `int *hardsect_size [MAX_BLKDEV]`
  Rozmiar sektora urządzenia (domyślnie 512).
- `int read_ahead [MAX_BLKDEV]`
  Ilość sektorów, które mają zostać przeczytane z wyprzedzeniem, gdy plik będzie czytany sekwencyjnie.

Plik nagłówkowy `<linux/blk.h>`

Dzięki temu, że w jądrze zebrano wspólny kod dla wszystkich programów obsługi urządzeń blokowych, pisząc nasz program możemy implementować tylko to co jest ściśle zależne od sposobu działania naszego urządzenia. Aby móc korzystać z dostarczonego przez jądro interfejsu musimy dołączyć plik nagłówkowy `<linux/blk.h>`. Wcześniej jednak musimy zdefiniować kilka symboli, które są potrzebne makrodefinicjom z tego pliku (makrodefinicje te zakładają z góry ich znajomość). Są to:

- `MAJOR_NR`
  Numer nadrzędny urządzenia (nie musi być zdefinowany jako stała).
- `DEVICE_NAME`
  Nazwa urządzenia.

- `DEVICE_NR(device)`
  Służy do określenia numeru fizycznego urządzenia na podstawie drugorzędnego numeru urządzenia (dla programu obsługi dysków kolejne fizyczne urządzenie to kolejny dysk w komputerze a nie kolejna partycja).

- `DEVICE_ON` i `DEVICE_OFF`
  Wykorzystywane w urządzeniach wyłączanych i włączanych (np. stacja dyskietek). Definiujemy je puste.

Przykładowy kod dla programu obsługi urządzenia `mydrv`:

```
#define MAJOR_NR          mydrv_major
#define DEVICE_NAME       "mydrv"
#define DEVICE_NR(device) MINOR(device)
    /* numer fizycznego urządzenia aktualnie obsługiwanego przez nasz
       program jest równy numerowi drugorzędnemu */
#define DEVICE_ON(device)       // nie obsługiwane
#define DEVICE_OFF(device)      // nie obsługiwane
int mydrv_major;
......
#include <linux/blk.h>
```

Funkcje `open` i `release`

```
int mydrv_open (struct inode *n, struct file *f)
int mydrv_release (struct inode *n, struct file *f)
```

Funkcje analogiczne do funkcji `open` i `release` programów obsługi urządzeń znakowych. Służą do otwierania i zamykania urządzenia. Jeśli piszemy nasz program jako moduł możemy w nich skorzystać z makr `MOD_INC_USE_COUNT` i `MOD_DEC_USE_COUNT`. Dzięki temu uniemożliwimy odładowanie modułu jeśli ktoś korzysta z urządzenia.

Funkcja `ioctl`

```
    int mydrv_ioctl (struct inode *inode, struct file *f, unsigned int cmd,
unsigned long arg)
```

Funkcja `ioctl` służy do sterowania urządzeniem. Dla programów obsługi urządzeń blokowych istnieją wspólne komendy, które program powinien w ramach tej funkcji obsługiwać. Ich deklaracje znajdują się w pliku `<linux/fs.h>`. Niektóre z nich to:

- `BLKROGET` i `BLKROSET`
  Sprawdzenie i utawienie flagi "tylko do odczytu".

- `BLKGETSIZE`
  Pobranie rozmiaru bieżącego urządzenia w sektorach.

- `BLKRAGET` i `BLKRASET`
  Pobranie i ustawienie wartości "read ahead" urządzenia.

A w kodzie programu (w funkcji mydrv_ioctl) obsługa przykładowej komendy może wyglądać tak:

```
switch (cmd) {
    ......
    case BLKGETSIZE:
        if (!arg) return -EINVAL;        // wskaźnik arg jest NULL
        return put_user (mydrv_sizes[MINOR(inode->i_rdev)], (long *) arg);
        /* przypisanie rozmiaru na arg (w przestrzeni użytkownika) */
    ......
}
```

gdzie mydrv_sizes[] to tablica z rozmiarami urządzeń obsługiwanych przez program indeksowana numerami drugorzędnymi.

Funkcje check_media_change i revalidate

```
int mydrv_check_media_change (kdev_t dev)
```

Funkcja sprawdzającą czy w urządzeniu identyfikowanym przez dev został wymieniony nośnik danych.

```
int mydrv_revalidate (kdev_t dev)
```

Wywoływana przez poprzednią funkcję w przypadku zwrócenia przez nią odpowiedzi twierdzącej. Typowo od nowa inicjuje struktury danych programu obsługi.

Funkcja request

```
void mydrv_request (void)
```

To jest nasza funkcja obsługi żądania na którą ustawiliśmy wskaźnik request_fn ze struktury blk_dev_struct.

Dla zwiększenia wydajności systemu jądro kolejkuje żądania przesłania danych do/z urządzenia i później przekazuje do programu obsługi całą listę żądań. Obsługą listy żądań zajmują się funkcje add_request, make_request i ll_rw_block z pliku <drivers/block/ll_rw_bl.c>. Dzięki wspólnemu interfejsowi zawartemu w pliku <linux/blk.h> nasza funkcja obsługi żądań może łatwo odwoływać się do kolejnych żądań na liście i je obsługiwać. Makro INIT_REQUEST sprawdza poprawność bieżącego żądania. Mamy do niego dostęp dzięki makru CURRENT (wskazuje na strukturę request). Kiedy obsłużymy żądanie informujemy o tym jądro wywołując funkcję end_request. Należy ją wywołać z jednym argumentem, który jeśli ma wartość 1 oznacza powodzenie w realizacji żądania, lub 0 w przypadku błędu. Gdy zwrócimy błąd jądro zapisze do logu systemowego informację o niepowodzeniu. Wywołanie end_request inicjuje CURRENT na kolejne żądanie i jeśli nie będzie więcej żądań następne wywołanie INIT_REQUEST automatycznie zakończy wykonywanie naszej funkcji obsługi. Dopóki nie wywołamy end_request jądro nie zleci obsługi kolejnego żądania. Funkcję end_request możemy oczywiście zaimplementować własną ale dzięki dołączeniu pliku nagłówkowego <linux/blk.h> nie jest to konieczne i zazwyczaj zbyteczne. Definicja struktury request znajduje się w pliku <linux/blkdev.h>, a jej pola przydatne dla funkcji żądania to:

- `kdev_t rq_de`
  Urządzenie do którego następuje dostęp przez to żądanie (przydatne do wydzielenia numeru drugorzędnego).

- `int cmd`
  Rodzaj zlecenia. Może przybierać wartości `READ` lub `WRITE`.

- `unsigned long sector`
  Pierwszy sektor, którego dotyczy żądanie.

- `unsigned long current_nr_sectors`
  Liczba sektorów bieżącego żądania.

- `char *buffer`
  Obszar w pamięci podręcznej do którego powinny być zapisane (`READ`) lub z którego powinny być odczytane dane (`WRITE`).

Schemat funkcji obsługi żądania może wyglądać tak:

```
void mydrv_request (void)
{
    while (1) {
    /* w pętli nieskończonej bo INIT_REQUEST sam zakończy
       wykonywanie naszej funkcji jeśli nie będzie już żądań */
        INIT_REQUEST;
        ......
        /* sprawdzenie poprawności */
        ......
        switch (CURRENT->cmd) {
            case READ :
                /* czytanie */
                break;
            case WRITE :
                /* pisanie */
                break;
            default:
                /* oops */
                end_request(0);        /* błąd */
                continue;
        }
        end_request(1);         /* ok, udało się */
    }
}
```

W powyższym przykładzie blokujemy jednak procesor na czas wykonywania operacji czytania/pisania. Jest to skrajnie niewydajną metodą implementacji programu obsługi urządzenia. Jądro dostarcza nam wiele mechanizmów dzięki którym możemy temu zapobiec. W przypadku "prawdziwych" urządzeń blokowych program obsługi będzie najczęściej korzystał z mechanizmu przerwań.

Sterowanie przerwaniami

W przypadku korzystania z mechanizmu przerwań schemat działania związanego z obsługą żądań programu obsługi może wyglądać tak:

- Funkcja obsługi żądania zleca niskopoziomową transmisję danych (np. przy pomocy portów) i następnie kończy się bez wywołania `end_request`.
- W funkcji obsługi przerwania, która zostanie uruchomiona po zgłoszeniu przerwania przez sterownik urządzenia, wykonywane jest sprawdzenie poprawności wykonania zleconej operacji i następnie wołana jest funkcja `end_request` z odpowiednim parametrem. Dalej należy jeszcze sprawdzić czy jest jeszcze w kolejce jakieś nie obsłużone zlecenie (sprawdzić czy `CURRENT != NULL`) i jeśli tak to uruchomić znowu funkcję obsługi żądania.

Oczywiście dobrze jest też stosować mechanizm liczników czasu aby nie czekać w "nieskończoność" na nadejście przerwania, które z różnych przyczyn może nigdy nie zostać zgłoszone. Wtedy funkcja wywoływana przez licznik czasu może wywołać `end_request(0)` i ewentualnie znowu uruchomić funkcję obsługi żądania.

5.4.4 Jak skorzystać z przerwań w moim sterowniku?

Co musisz wiedzieć

Po zleceniu obsługiwanemu urządzeniu wykonania operacji są dwie strategie dalszych postępowań. Pierwsza metoda to aktywne czekanie (odpytywanie). Rozwiązanie to ma poważną wadę, gdyż dużo czasu jądro będzie zajęte przez oczekiwanie. Odpytywanie jest czasami nieuniknione. Istnieje możliwość zatuszowania jego wady: stosować metodę budzenia się co jakiś czas i jeśli nie gotowe zasypianie na nowo. Do tego służą liczniki czasu opisane poniżej. Dużo lepszy jest mechanizm przerwań. Można zlecić wykonanie jakiegoś polecenia i i czekać, aż urządzenie samo powiadomi, wysyłając przerwanie, że już skończyło. Oczywiście urządzenie musi wspomagać taki mechanizm przerwań.

Przerwania wykonują się w kontekście procesu, który się wykonywał w chwili przybycia przerwania, co oznacza, że w procedurze obsługi przerwania niewiele można zrobić. Do zrobienia czegoś ponadto służą dolne cześci przerwań, które wykonują się przy najbliższej okazji po wystąpieniu przerwania.

Ponieważ liczba przerwań jest ograniczona, to Linux umożliwia współdzielenie przerwania przez kilka urządzeń. Oczywiście procedury obsługujące przerwania dzielone muszą być napisane w stosowny sposób. Jądro przy wystąpieniu przerwania dzielonego odpala po kolei wszystkie procedury obsługi tego przerwania. Więc procedura obsługi musi móc sprawdzić, czy to jej urządzenie spowodowało przerwanie i jeśli nie, to zaniechać jego obsługi.

Przerwania w Linux'ie są dwóch typów: szybkie i wolne. Po zakończeniu obsługi przerwania wolnego sprawdza się stan zmiennej globalnej need_resched, która wskazuje planerowi, czy należy wybrać następny proces do wykonania. Po zakończeniu obsługi przerwania szybkiego sterowanie wraca do procesu, który się wykonywał, gdy przerwanie przyszło. Oprócz tego istotną różnicą jest to, że przerwania szybkie obsługują się przy zablokowanych innych przerwaniach, podczas obsługi przerwań wolnych mogą wystąpić inne przerwania. Ogólna zasada jest taka, że jeśli można, to należy korzystać z przerwań wolnych.

Co zrobić, żeby to zaczęło działać?

Przerwania w Linux'ie są bardzo podobne do sygnałów. Na niższym poziomie używają nawet tej samej struktury sigaction.

Załóżmy, że już wiemy jakie przerwanie chcemy obsługiwać. Do przypisania przerwaniu procedury obsługi służy funkcja request_irq:

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

zdefiniowana w kernel/irq.c. Pobiera ona cztery parametry: numer przerwania, które chcemy obsługiwać, wskaźnik do funkcji obsługującej, znaczniki oraz nazwę urządzenia, którą można będzie zobaczyć później w /proc/interrupts. Znaczniki mogą być albo SA_INTERRUPT, jeśli chcemy, by przewanie było szybkim, albo 0. W zależności od typu przerwania, różne wartości otrzymuje funkcja sterująca. Dla przerwań jest to numer przerwania, które ma być obsłużone, dla wolnych zaś jest to wskaźnik na strukturę zawierającą wartości rejestrów. Trudno jest wyobrazić sobie sytuację, w której ten parametr mógłby być przydatny. Jeśli tylko w procedurach obsługujących kilka przerwań.

Zwraca ta funkcja 0, gdy wszystko przejdzie pomyślnie, lub kod błędu w przypadku porażki. Np. -EBUSY, gdy przerwanie jest już zajęte.

Zwalnia się przerwanie wykonyjąc funkcję free_irq:

```
int free_irq(unsigned int irq)
```

zdefiniowaną w kernel/irq.h.

Liczniki czasu (ang. timers)

Jeśli ktoś ma wątpliwości co do słuszności umieszczenia tutaj tego tematu, to może przekona się, gdy powiem, że liczniki czasu są realizowane za pomocą przerwania zegara.

Liczniki czasu (ang. timer) są dwóch typów. Jedne są związane ze starymi urządzeniami, które nie generują przerwań, więc musza być obsługiwane przez odpytywanie. Urządzenia te są obsługiwane przez Linux od zamieszchłych czasów, i pewnie dlatego używają liczników czasu wszytych na sztywno do jądra. Liczniki te są przechowywane w tablicy.

```
struct timer_struct timer_table[32];
```

Tablica ta jest zadeklarowana w kernel/sched.c. Dla jakich celow te liczniki są przeznaczone mozna wyczytać w komentarzach do stałych xx_TIMER w linux/timer.h. Tam są wspomniane np. wygaszacz ekranu do konsoli, sygnałek wydawany przez "dziwne" klawisze, port RS-232 itd. timer_struct jest zdefiniowana w linux/timer.h tak:

```
struct timer_struct {
unsigned long expires;
void (*fn)(void);
};
```

expires jest czasem w jiffies, po którym zostanie wygenerowanie przerwanie i wykona się funkcja fn. Oprócz tego jest maska:

```
unsigned long timer_active;
```

pochodzi z kernel/sched.c. Poszczególne bity odpowiadają elementom timer_table. Jeśli licznik działa, to odpowiedni bit jest ustawiony. Patrz też dalej opis makra SET_INTR(x)

Drugi typ liczników, jest do wykorzystania w życiu codziennym. Taki licznik jest reprezentowany przez strukturę time_list zdefiniowaną w linux/timer.h:

```
struct timer_list {
struct timer_list *next;
struct timer_list *prev;
unsigned long expires;
unsigned long data;
void (*function)(unsigned long);
};
```

next i prev służą do połączenia takich struktur w listę, expires jest czasem w jiffies, po którym ma zostać wygenerowane przerwanie i wykonana funkcja function z parametrem data. Jeśli chcesz w swoim sterowniku skorzystać z takiego licznika, to musisz zaalokowac strukturę timer_list wykonać na niej funkcję

```
void init_timer(struct timer_list *timer);
```

zdefiniowana w <linux/timer.h>, która nie zrobi nic poza przypisaniem NULL na prev i next.

Przypisz odpowiednie wartości polom data i function. Ustaw expires na jiffies + «czas oczekiwanie na przerwanie w i powiadom resztę jądra, że dobrze, jeśli zacznie obsługiwać twój liczniczek. Dokonasz tego wykonując funkcję

```
void add_timer(struct timer_list *timer);
```

zdefiniowana w kernel/sched.c. Teraz można spokojnie czekać, aż zostanie wykonana, przypisana do twojego licznika funkcja po określonym przez Ciebie czasie.

Gdy ta chwila bedzie już w przeszłości i zechcesz skorzystać ze swojego liczniczka jeszcze raz, to pamiętaj, że nie wystarczy ustawić expires na nową wartość. Trzeba najpierw usunąć go ze struktur reszty jądra i stworzyć na nowo. Dokonasz tego wykonując funkcję

```
void del_timer(struct timer_list *timer);
```

zdefiniowana w kernel/sched.c, i przechodzac przez kilka kroków opisanych powyżej raz jeszcze. Jest też prostszy sposób. Wykonaj

```
void mod_timer(struct timer_list *timer, unsigned long expires);
```

zdefiniowana w kernel/sched.c, a zrestartujesz liczniczek jednym poleceniem. expires jest jak zawsze czasem w jiffies, po którym zostanie wykonana przypisana do liczniczka funkcja.

/subsubsectionMakra pomocne przy obsłudze przerwań.

Wszystkie makra i stałe opisane tutaj pochodzą z pliku linux/blk.h i oczywiście mają zastosowanie tylko podczas pisania sterowników do urządzeń blokowych.

Jak zostało powiedziane w punkcie dotyczącym pisania sterowników urządzeń blokowych, chcąc dołączyć swój sterownik do jądra w sposób stały (wkompilować go), to w pliku linux/blk.h trzeba dodać kilka definicji stałych. Między innymi jeśli twój sterownik używa przerwan, to rozsądne jest dodać linie

```
#define DEVICE_INTR do_xxx
```

Po dodaniu tej linii otrzymuje się dostęp do kilku przydatnych makr opisanych poniżej. Oprócz tego można jeszcze zdefiniować

```
#define DEVICE_TIMEOUT xxx_TIMER
#define TIMEOUT_VALUE n
```

DEVICE_TIMEOUT rzadko wymaga ustawienia. Najwyżej chcesz napisać sterownik do ktoregoś ze starych dysków IDE, ktore nie generują przerwań. To jest indeks w tablicy liczników czasu zaszytych na sztywno w jądra.

O ile podaliśmy DEVICE_INTR i oznajmiliśmy przez to, że chcemy korzystać z przerwań, deklaruje się funkcję

```
static void (*DEVICE_INTR)(void) = NULL;
```

By ustawić procedurę obslugujacą przerwanie (właczyć przerwanie) wykonujemy SET_INTR(xxx), gdzie xxx jest zdefiniowaną przez nas procedurą obsługi przerwania. Do zakończenia obsługi tego przerwania wykonujemy SET_INTR(NULL). Jeśli zdefiniowaliśmy DEVICE_TIMEOUT to, SET_INTR(x) jest zdefiniowana następująco:

```
Jeśli ustawiamy zmienną DEVICE_INTR na x,
jeśli x != NULL, to
wykonaj SET_TIMER
wpp.
wykonaj CLEAR_TIMER
wpp.
SET_INTR(x) poprostu ustawia DEVICE_INTR na x
```

SET_TIMER ustawia pole expires w timer_table[DEVICE_TIMEOUT] tak, by po TIMEOUT_VALUE została wywołana funkcja przypisana do licznika timer_table[DEVICE_TIMEOUT] oraz ustawia odpowiedni bit w timer_active sygnalizując przez to, że liczniczek jest aktywny. CLEAR_TIMER zeruje odpowiedni bit w timer_active, przez co wyłącza licznik timer_table[DEVICE_TIMEOUT].

5.5 Zadania

5.5.1 Treści

1. Ile pamięci zajmują tablice rozdzielcze urządzeń? (dla jądra 2.2.12)

2. *Szeregowanie żądań do urządzenia blokowego* Jak zostało napisane w rozdziale *Jak napisać program obsługi urządzenia blokowego*, dla zwiększenia wydajności systemu jądro kolejkuje żądania odczytu/zapisu z/do urządzenia blokowego i później przekazuje do programu obsługi całą listę żądań. Nasza *procedura strategii* (funkcja odpowiedzialna za realizację zleceń), której szkic implementacji także został zaprezentowany w książce, mogła pobierać

sekwencyjnie tylko pierwsze żądanie z kolejki (makra `INIT_REQUEST`, `CURRENT` i funkcja `end_request`). Na sposób w jaki zlecenia były umieszczane w tej kolejce program obsługi urządzenia nie miał żadnego wpływu.

W Linuksie dla urządzeń blokowych istnieje jedna, wspólna funkcja szeregująca żądania, `add_request` z pliku `drivers/block/ll_rw_blk.c`. Żądania są przez nią szeregowane leksykograficznie, najpierw po numerze drugorzędnym urządzenia, a później po numerze sektora (pole `sector` struktury `request`), którego dotyczą. Nowe żądania są wsortowywane do kolejki zgodnie z algorytmem C-LOOK.

Zadanie polega na tym aby:

(a) Tak zmienić funkcję szeregującą aby wśród żądań dotyczących jednego urządzenia (ten sam numer drugorzędny) żądania odczytu miały pierwszeństwo przed żądaniami zapisu, a w przypadku tej samej operacji o pierwszeństwie niech decyduje numer sektora, którego dotyczą (bez zmian).

(b) Zmienić `add_request` tak aby żądania były szeregowane w kolejności "przyjścia" (FCFS).

(c) W funkcji szeregującej zaimplementować algorytm F-C-LOOK. W tym algorytmie tak samo jak w C-LOOK głownica przesuwa się w jednym kierunku i realizuje kolejno napotkane żądania, lecz w F-C-LOOK realizowane są jedynie te, które nadeszły przed rozpoczęciem ruchu głownicy w danym kierunku. Chroni to między innymi przed zagłodzeniem jeśli stale przychodzą żądania do bieżącej ścieżki.

(d) Przeprowadzić symulację szeregowania dla C-LOOK, FCFS, F-C-LOOK, dla żądań dotyczących sektorów: `1,2,5,3,6,4`. Wszystkie żądania nadejdą przed zrealizowaniem pierwszego.

(e) Niech czas przesunięcia głowicy o jeden cylinder wynosi 5, a czas realizacji żądania 4. Przeprowadzić symulację odwołań do bloków znajdujących się na cylindrach: `3(1), 5(2), 4(3), 1(4), 0(7)` (w nawiasach czasy przyjścia żądań), jeśli jest wykorzystywany algorytm C-LOOK (lub F-C-LOOK). Należy podać czas zrealizowania każdego żądania.

3. Zmodyfikować jądro Linuksa tak, aby w katalogu /proc pojawił się plik 'extdevs', którego zawartość będzie rozszerzoną wersja pliku 'devices' z tego samego katalogu. Oprócz numeru nadrzędnego i nazwy sterownika, w każdej linijce powinna się znaleźć lista operacji, jakie udostępnia dany sterownik.

   Pytanie do zastanowienia: czy sytuacja, w której dla danego sterownika lista dostepnych informacji składa się tylko z 'open' oznacza, że nie implementuje on żadnych innych operacji?

4. Zakładając, że kolejkujemy zadania tylko dla jednego urządzenia, podać listę jaką wygeneruje algorytm kolejkowania żądań dla podanych niżej żądań: 16,23,8,20,15,25,4,17

5. Napisz program obsługi urządzenia `mutex` udostępniającego zbiór semaforów binarnych (MUTEXów).

   Dostępne mają być tylko dwie operacje: open - odpowiednik V() i close/release - odpowiednik P(). Urządzenie udostępnia pewną ustaloną liczbę semaforów przez kolejne numery drugorzędne. Semafory są zainicjowane wartością 1.

Urządzenie zapisz w formie modułu. Dla uproszczenia można przyjąć, że moduł będzie działał na maszynie jednoprocesorowej (nie trzeba wtedy martwić się o atomowość operacji w jądrze).

6. W jaki sposób dodać obsługę nowego klawisza Magic SysRQ key?

7. Zadanie polega na napisaniu sterownika dla urządzenia znakowego „ideinfo". Urządzenie to ma służyć do wypisywania informacji o urządzeniach obsługiwanych przez sterownik `ide.c`.

   Urządzenie powinno obsługiwać operacje: `open(), close(), read(), lseek() i ioctl()`.

   Operacja `read()` ma wypisywać do pamięci użytkownika jedną informację o jednym urządzeniu.
   Operacja `lseek()` ma zmieniać urządzenie, które nas interesuje na n-te po aktualnym lub n-te od pierwszego, gdzie n jest parametrem `offset`.
   Operacja `ioctl()` służy do ustawienia rodzaju informacji jaka nas intesuje, przekazywanego przez parametr cmd. Informacje, jakich uzyskanie ma umożliwiać nasze urządzenie to (cmd - opis):
   NAME – nazwa urządzenia, np. "hda"
   INTERF – interfejs, do którego jest ono podłączone
   MAJ – numer główny
   IRQ – numer przerwania obsługującego urządzenie
   HEAD – prawdziwa liczba glowic
   SECT – prawdziwa liczba sektorów
   CYLS – prawdziwa liczba cylindrów
   MULTSECT – maksymalna możliwa do ustawienia liczba sektorów przesyłanych jednorazowo
   DRV – nazwa sterownika obsługującego urządzenie
   DRVVER – wersja tego sterownika
   Aby mieć dostęp do struktur danych sterownika `ide.c` należy do naszego programu dołączyć plik nagłówkowy `ide.h` tego sterownika. W źródłach jądra znajduje się on w katalogu `drivers/block/`.

8. W jaki sposób można użyć dyscypliny linii (być może wraz z innym częściami systemu) do prostego szyfrowania transmisji?

9. Większość sterowników urządzeń znakowych wykorzystuje niewiele numerów podrzędych w ramach przyznanego im numeru głównego. Napisz moduł do jądra implementujący funkcje register_subdev(struct file_operations*) i unregister_subdev(int), które przydzielają sterownikom numery minor w ramach jednego numeru głównego.

10. W jaki sposób na jednym numerze glównym można zainstalować wiele rodzajów urządzeń z różną implementacją procedur read, write, lseek w zależności od numeru podrzędnego?

11. Jak lokalną zmienną lub funkcję z modułu można uczynić widoczną dla innych modułów?

12. Kilka procesów działających w systemie w pewnym odcinku czasu generuje następujące żądania do dysków (major, minor, sektor): (15,0,3800),(3,1,10),(22,1,17),(3,5,2),(15,0,3801),(3,2,1200) Jak one zostaną rozmieszczone w kolejk(ę/i) żądań?

13. W systemie są zainstalowane dwa zupełnie identyczne fizyczne dyski twarde (interfejs IDE) widoczne w systemie jako /dev/hda i /dev/hdb. Czy istnieje możliwość, że proces A czytający z pliku znajdującego się na dysku hda zagłodzi proces B czytający z dysku hdb?

14. Na przykładzie przeanalizować, które procesy mogą zostać powieszone, a które na pewno będą powieszone, jeśli zanotowaliśmy następujące żądania zapisu do dysku przy założeniach:

    - stała NR_REQUEST = 7
    - przed pojawieniem się żądania procesu h nie został utworzony żadaen request
    - Żądanie = nr_główny.nr_drugorzędny.sektor
        - 3.1.7 proces a
        - 3.1.5 proces b
        - 3.1.7 proces c
        - 4.1.7 proces d
        - 3.1.9 proces e
        - 3.1.7 proces f
        - 3.1.7 proces g
        - 3.1.7 proces h
        - 3.1.7 proces i
        - 3.1.5 proces j
        - 3.1.7 proces k

15. Na konsolę możemy oprócz zwykłych znaków wysyłać sekwencje specjalne zaczynające się od Esc. Służą one do wywołania specyficznych komend konsoli do np. zmiany ustawień, czy odczytu statusu. Znajomość tych sekwencji jest bardzo przydatna w posługiwaniu się konsolą. Zadanie ma na celu zaznajomienie się z tymi sekwencjami.

    Należy zajrzeć do pliku drivers/char/console.c

    Należy (np. posługując się komendą echo) zrobić następujące rzeczy:

    (a) ustawić konsole w trybie reverse, co oznacza czarne litery na białym tle
    (b) ustawić konsole z powrotem w zwykły tryb
    (c) zmienić rozmiar kursora na większy
    (d) przywrócić standartowy rozmiar kursora

16. Zmienić jądro linuksa, tak aby były obsługiwna następujące sekwencje znaków wysyłane na konsole:

```
\\033[pitchS
\\033[pitch;dur_mulS
\\033[pitch;dur_mul;dur_divS
```

    Sekwencje te mają służyć do zmieniania wysokości i długości dźwięku dzwonka. Dźwięk dzwonka można uzyskać wypisując na konsolę znak o kodzie 7.

    Poszczególne parametry liczbowe oznaczają:

- pitch - częstotliwość wyskości dźwięku
- dur_mul, dur_div - długość trawania dzwięku w sekundach wyznaczonych wzorem dur_mul/dur_div, w przypadku nie podania dur_div przyjmowane jest dur_div = 1

17. Mając daną wielkość urządzenia blokowego i rozmiar jego bloku oraz ilość przydzielonych buforów i średni koszt wykonania przez urządzenie odczytu/zapisu jednego bajtu, wyznacz średni koszt zapisu jednego bajtu na urządzenie blokowe. Jak wyglądałoby to wyliczenie w systemie bez buforowania?

18. Mając takie same dane jak w zadaniu poprzednim, wyznacz średni koszt zapisu jednego bajtu na urządzenie blokowe wiedząc, że wykonana została operacja : UPDATE tabela SET a='nowa wartosc' WHERE ... .

5.5.2 Rozwiązania

1. Wzór ogólny:

```
  (MAX_BLKDEVS + MAX_CHARDEVS) * sizeof( struct device_struct )
  gdzie
    sizeof( struct device_struct ) = 2 * ( sizeof( char* ))
Dla jądra 2.2.12
  (256 + 256) * 2 * 4 = 4096 (bajtow)
```

2. *Szeregowanie żądań do urządzenia blokowego*:

   (a) Do wyznaczania porządku na żądaniach funkcja szeregująca korzysta z makra `IN_ORDER` z pliku `include/linux/blk.h`:

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

   gdzie `s1` i `s2` będą typu struktury `request`. W naszym rozwiązaniu należy dodać porównywanie ze względu na operację `cmd` (czytanie/pisanie), np:

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

   (b) Za wsortowywanie nowego żądania do kolejki odpowiada kawałek kodu:

   ```
   /* tmp - wskazuje na bieżące żądanie,
      req - wskazuje na nowe, wstawiane żądanie,
            oba wskaźniki na struct request */

   for ( ; tmp->next ; tmp = tmp->next) {
       const int after_current = IN_ORDER(tmp,req);
       const int before_next = IN_ORDER(req,tmp->next);
   ```

```
        if (!IN_ORDER(tmp,tmp->next)) {
            if (after_current || before_next)
                break;
        } else {
            if (after_current && before_next)
                break;
        }
    }
    req->next = tmp->next;
    tmp->next = req;
```

Należy "wyrzucić" ciało powyższej pętli `for`, czyli po zakończeniu jej wykonywania `tmp` będzie wskazywał na ostatni element w kolejce żądań.

(c) Musimy zmienić pokazaną w poprzednim punkcie pętle wsortowującą żądania do kolejki. Przykład zmiany:

```
    /* int wsk - pomocnicza zmienna */
    wsk = 0;
    for ( ; tmp->next ; tmp = tmp->next) {
        const int after_current = IN_ORDER(tmp,req);
        const int before_next = IN_ORDER(req,tmp->next);

        if (!IN_ORDER(tmp,tmp->next)) {
            wsk = 1;
            if (before_next)
                break;
        } else {
            if (after_current && before_next && wsk)
                break;
        }
    }
```

(d) C-LOOK: `1,2,3,4,5,6`, FCFS: `1,2,5,3,6,4`, F-C-LOOK: `1,2,5,3,4,6`.

(e) C-LOOK: `3(5), 4(14), 5(23), 0(52), 1(61)`, F-C-LOOK: `3(5), 5(19), 0(48), 1(57), 4(76)`.

3. Istnieje już plik '/proc/devices', nasze zadanie ogranicza się więc jedynie do odszukania implementacji jego funkcji i dostosowania ich do nowej specyfikacji.

   Odpowiedź na pytanie.

   Niekoniecznie. Funkcja 'open' może podmienić (i czesto tak właśnie postępuje) strukturę file_operations na właściwą dla danego numeru drugorzednego, dzięki czemu wygodna jest obsługa urządzeń różnego typu z poziomu jednego sterownika. Przykładem są tutaj urządzenia o numerze nadrzędnym 10 (misc), czy 14 (sound).

   Przykład rozwiązania.

O ile nie podano dokładnej lokalizacji dodawanego kodu, należy go umieszczać możliwie blisko odpowiednika dla 'devices'.

Istotne jest też zwracanie uwagi na to, aby nie przekroczyć rozmiaru jednej strony przy wypisywaniu statystyk. W zdecydowanej większości przypadków 4096 bajtów wystarczy do wypisania interesujacych nas informacji, jeżeli jednak limit ten miałby zostać przekroczony, należy wypisać tylko pierwsze 4kb informacji. W kodzie poniżej warunek ten sprawdzany jeszcze bardziej restrykcyjnie (3900 bajtow zezwala na dodanie jeszcze jednego wpisu do bufora, wpp koniec). Jest to podyktowane potencjalnie dużą liczbą znaków w linii.

```
a) /usr/src/linux/fs/devices.c
 - dodać funkcje 'get_device_extlist'
   np. taka:

   int get_device_extlist(char * page)
   {
      int i,j,t;
      int len=0;
      struct file_operations * fops;
      struct tabledesc {
         struct device_struct * devs;
         char * desc;
         int max;
      } tables[]={
         {chrdevs,"Character",MAX_CHRDEV},
         {blkdevs,"Block",MAX_BLKDEV}
          };
      char * operations[] = {
         "seek","read","write","readdir","poll","ioctl",
         "mmap","open","flush","release","fsync","fasync",
         "media_change","revalidate","lock"
          };
      int opcnt=15;

      for (t = 0;t < 2;t++) {
        len += sprintf(page+len, "\n%s devices:\n\nmajor name    operations\n",
              tables[t].desc);
        for (i = 0; i < tables[t].max ; i++)
        if (tables[t].devs[i].fops && len < 3900) {
            len += sprintf(page+len, "%5d %8s", i, tables[t].devs[i].name);
            fops=tables[t].devs[i].fops;
            for (j=0;j < opcnt;j++)
               if ( ((char**)fops)[j] )
                  len+=sprintf(page+len," %s",operations[j]);
            len += sprintf(page+len, "\n");
        }
      }
```

```
        return len;
    }

b) /usr/src/linux/include/proc_fs.h
 - dodać do zbioru wyliczeniowego root_directory_inos etykietę PROC_EXTDEVS;

c) /usr/src/linux/fs/proc/array.c
 - w funkcji 'get_root_array' dodać do switch-a obslugę PROC_EXTDEVS;
   [...]
       case PROC_EXTDEVS:
            return get_device_extlist(page);
   [...]

d) /usr/src/linux/fs/proc/root.c
 - dodać strukturę opisującą wpis do /proc:
   [...]
   static struct proc_dir_entry proc_root_extdevs = {
       PROC_EXTDEVS, 7, "extdevs",
       S_IFREG | S_IRUGO, 1, 0, 0,
       0, &proc_array_inode_operations
   };
   [...]

 - dodać instrukcję rejestrującą w funkcji proc_root_init()
   [...]
   proc_register(&proc_root, &proc_root_extdevs);
   [...]
```

4. Algorytm kolejkowania działa prawie jak insertion-sort przy czym nowe żądanie nie może być wstawione jako pierwsze. Istotnie: na końcu listy jest strażnik z największą wartością jeżeli lista jest pusta to wstawiamy nowy element i za nim strażnika
   - przechodzimy listę po kolei
   - jeżeli dwa kolejne elementy są posortowane i nowy jest pomiędzy nimi to go tam wstawiamy
   - jeżeli dwa kolejne elementy nie są posortowane to wstawiamy nowy, jeżeli jest on większy od pierwszego lub mniejszy od drugiego z rozpatrywanych elementów tej listy
   - W wyniku podanych danych dostaniemy: 16,17,20,23,25,4,8,15
5. Istotniejsze fragmenty przykładowego rozwiązania:

```
#define MUTEX_MAJOR 124
#define MUTEX_MINORS 16

#define current_mutex (mutexes[MINOR(iptr->i_rdev)])
```

```
struct mutex {
        int count;
        struct wait_queue *queue;
};

struct mutex mutexes[MUTEX_MINORS];

int mutex_open(struct inode *iptr, struct file *filp)
{
        if (MINOR(iptr->i_rdev) >= MUTEX_MINORS)
                return -ENODEV;
        while ((current_mutex.count) == 0) {
                interruptible_sleep_on(&current_mutex.queue);
        }
        current_mutex.count = 0;
        return 0;
}

int mutex_release(struct inode *iptr, struct file *filp)
{
        current_mutex.count = 1;
        wake_up_interruptible(&current_mutex.queue);
        return 0;
}

int init_module(void)
{
        int i;
        for (i=0; i<MUTEX_MINORS; i++) {
                mutexes[i].count = 1;
                init_waitqueue(&mutexes[i].queue);
        }
        ... /* rejestracja urządzenia */
}

... /* cleanup_module */
```

6. Obsługa SysRq Key-s jest zaimplementowana w pliku drivers/char/sysrq.c. Aby dodać nowy klawisz, trzeba odnaleźć funkcję

```
void handle_sysrq(int key, struct pt_regs *pt_regs,
                  struct kbd_struct *kbd, struct tty_struct *tty)
```

Tam, za pomocą instrukcji case dodawane są kolejne klawisze. Trzeba dodać swój i zdefiniować oddzielną funckję, która będzie obsługiwać klawisz (jeśli akcja jest skomplikowana): np: dodajemy klawisz "h"wyświetlający napis "witam"

```
...
case 'h':
printk("Witam\n");
break;
...
```

7. Plik z definicjami **defs.h**:

```
#include <linux/ioctl.h>

#define NAME _IO( 'd', 0x00)
#define INTERF _IO( 'd', 0x01)
#define MAJ _IO( 'd', 0x02)
#define IRQ _IO( 'd', 0x03)
#define HEAD _IO( 'd', 0x04)
#define SECT _IO( 'd', 0x05)
#define CYLS _IO( 'd', 0x06)
#define MULTSECT _IO( 'd', 0x07)
#define DRV _IO( 'd', 0x08)
#define DRVVER _IO( 'd', 0x09)
```

Plik z kodem sterownika **ideinfo.c**:

```
#include <linux/config.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/major.h>
#include <asm/uaccess.h>
#include <linux/kernel.h>

#include < /*tu powinna być scieżka do pliku nagłówkowego ide.c - ide.h */ >

#include "defs.h"

struct ide_drive_s * naped;
char txt[10];
int hwif = 0;
int dysk = 0;

int major = 42; /* przykładowy numer */

static ssize_t
ii_read(struct file * file, char * buf,size_t count, loff_t *filep)
{
```

```
    int i;

    for (i = 0; i < count; i++) {
put_user(txt[(file->f_pos)++], buf);
buf++;
if (file->f_pos == strlen(txt))
    file->f_pos = 0;
    }
    return count;
}

static int
ii_ioctl(struct inode * inode, struct file * file, unsigned int cmd, unsigned long arg)
{
    switch (cmd) {
case NAME:
strcpy(txt, naped->name);
break;
case INTERF:
strcpy(txt, HWIF(naped)->name);
break;
case MAJ:
sprintf(txt, "%d", HWIF(naped)->major);
break;
case IRQ:
sprintf(txt, "%d", HWIF(naped)->irq);
break;
case HEAD:
sprintf(txt, "%d", naped->head);
break;
case SECT:
sprintf(txt, "%d", naped->sect);
break;
case CYLS:
sprintf(txt, "%d", naped->cyl);
break;
case MULTSECT:
sprintf(txt, "%d", naped->id->max_multsect);
break;
case DRV:
strcpy(txt, DRIVER(naped)->name);
break;
case DRVVER:
strcpy(txt, DRIVER(naped)->version);
break;
default:
```

```
return -EINVAL;
    };
    txt[strlen(txt)] = '\n';
    file->f_pos = 0;
    return 0;
}

static loff_t
ii_lseek(struct file * file, loff_t offset, int orig)
{
    int i;

    if (orig == 0) {
hwif = 0;
dysk = 0;
    };
    for (i = 0; i < offset; i++) {
dysk++;
if (dysk == MAX_DRIVES) {
    dysk = 0;
    hwif++;
};
if (hwif == MAX_HWIFS)
    hwif = 0;
if (ide_hwifs[hwif].drives[dysk].driver == NULL)
    i--;
    };
    naped = &ide_hwifs[hwif].drives[dysk];
    ii_ioctl(file->f_dentry->d_inode, file, NAME, 0);
    return offset;
}

static int
ii_open( struct inode* inode, struct file* filep)
{
    if (naped->driver == NULL)
ii_lseek(filep, 1, 1);
    ii_ioctl(inode, filep, NAME, 0);

    return 0;
}

static int
ii_close( struct inode* ino, struct file* filep)
{
    return 0;
```

```
}

static struct file_operations hw_fops = {
ii_lseek,
ii_read,
NULL,
NULL,
NULL,
ii_ioctl,
NULL,
ii_open,
NULL,
ii_close,
NULL
};

int
init_ideinfo(void)
{
    hwif = 0;
    dysk = 0;
    naped = &ide_hwifs[0].drives[0];

    if (register_chrdev(major, "ideinfo", &hw_fops))
return -EIO;

    return 0;
}
```

8. Można wprowadzić dodatkową tablicę konwersji znaków używaną przez dyscyplinę linii do wstępnej konwersji znaków.

9. Rejestrujemy numer główny z operacjami zawierającymi jedynie implenentację open, która podmienia strukturę file_operations na właściwą i wywołać ją (podobnie jak w systemowym algorytmie dla file_open).

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>

#define MMAJOR 135
#define MNAME "subdevs"
#define MAX_SUBDEV (1<<MINORBITS)

static struct file_operations *subdev[MAX_SUBDEV];
```

```
static int open(struct inode *i, struct file *f) {
struct file_operations *fops = subdev[MINOR(i->i_rdev)];
if(!fops) return -ENODEV;
f->f_op = fops;
return fops->open(i,f);
}
static struct file_operations my_ops = { 0, 0, 0, 0, 0, 0, 0, open, 0, 0 };

int register_subdev(struct file_operations *fops) {
int i;
for(i=0;i<MAX_SUBDEV;i++) if(!i) break;
if(i==MAX_SUBDEV) return -1;
subdev[i] = fops;
MOD_INC_USE_COUNT;
return i;
}
int unregister_subdev(int i) {
if(!subdev[i]) return -1;
subdev[i] = 0;
MOD_DEC_USE_COUNT;
return 0;
}

int init_module(void) {
return register_chrdev(MMAJOR,MNAME,&my_ops);
}
void cleanup_module(void) {
unregister_chrdev(MMAJOR,MNAME);
}
```

10. Jeden ze sposobów to w procedurze open dla urzadzenia przypisać nową strukture file_operation do (struct file)filep->f_op w zależności od MINOR(inode->i_rdev).

11.
```
#define EXPORT_SYMTAB
     #include <linux/module.h>
int coś;
EXPORT_SYMBOL(coś);
```

12.
```
Kolejka do urządzeń o numerze głównym 3 (1,10), (2,1200), (5,2)
-//- 15 (0,3800), (0,3801)
-//- 22 (1,17)
```

13. Teorytycznie taka możliwość istnieje, gdyż żądania do urządzeń blokowych są sortowane leksykograficznie względem par (minor_nr, sector_nr), a więc żądania procesu A dysku hda, będą wstawiane do kolejki przed żądaniami procesu B (hda i hdb mają wspólną kolejkę, gdyż są urządzeniami o tym samym numerze głównym). Praktycznie to jest prawie niemożliwe z tego względu, że trudno, by proces A generowałby żądania z taką samą prędkością z jaką

dysk zacząłby je obsługiwać, a tylko w tym wypadku byłaby możliwość, że ani proces B nie zacznie się wykonywać, ani liczba struktur request w systemie nie zostanie wyczerpana, co spowodowałoby zaśnięcie A i dałoby większe szanse B.

14. Po pierwsze proces d żąda dostępu do urządzenia o innym numerze głównym - nie na nim tłoku, więc jak tylko utworzone zostanie request, zostanie obsłużony, więc na pewno nie zawiśnie

    Po drugie spośród procesów a-h pięć żądało 3.1.7, ale $5/7 > 2/3$, więc procesy h,i,j,k poczekają na opróżnienie się tablicy, przez procesy wcześniejsze. Więc pewne jest, że proces h zawiśnie. O pozostałych procesach nic nie wiemy.

15. (a) echo $'
    033[?5h'

    (b) echo $'
    033[?5l'

    (c) echo $'
    033[?6c'

    (d) echo $'
    033[?0c'

16. Należy zmienić plik drivers/char/console.c

    Dodajemy funkcję:

```
static void bell_mode(int currcons)
{
        if (npar > 0)
                bell_pitch = par[0];
        if (npar > 1)
                bell_duration = par[1] * HZ;
        if (npar > 2)
                bell_duration /= par[2];
}
```

    W funkcji do_con_trol po kawałku:

```
                case 'P':
                        csi_P(currcons,par[0]);
                        return;
```

    dodajemy:

```
                case 'S':
                        bell_mode(currcons);
```

17. Niech

- *b - rozmiar bloku*
- *n - wielkość urządzenia (w blokach)*
- *o - średni koszt wykonania zapisu/odczytu przez urządzenie*
- *x - średnia ilość buforów dla urządzenia (w linux'ie zmienna więc uśredniamy dla uproszczenia obliczeń)*

oraz

- *k - ilość zapisów, które chcemy wykonać*
- *z - co ile zapisów zmodyfikowane bloki są zrzucane w całości na dysk (przyjmijmy dla uproszczenia bo w linux'ie mierzy się odstępy między zrzutami na dysk w jednostkach zegarowych)*

Wyliczmy jeszcze pomocniczą wartość $a = x/n$ - prawdopodobienstwo że żądanie dotyczy zbuforowanego bloku. Koszt zapisu k - bajtow nie koniecznie znajdujących się obok siebie na dysku wyniesie wiec :

```
Z(k) = k*(1 - a)*b*o + koszt_zrzutów
koszt_zrzutów to KZ = k/z * ( z*(1 - a)*b*o ) = k*(1 - a)*b*o
```

zatem średni koszt zapisu jednego bajtu wynosi :

```
Sr = ( Z(k) + KZ ) / k = 2*(1 - x/n)*b*o
```

Oczywiście bez buforowania wynosi :

```
Sr =  2*b*o
```

18. Niech *d - dlugość pola do zmodyfikowania (ilość bajtów do zapisania w każdym modyfikowanym bloku, przy założeniu, że jeden rekord bazy danych mieści się w jednym bloku)* oraz podobnie jak w zadaniu poprzednim : *a* - prawdopodobienstwo że zadanie dotyczy zbuforowanego bloku.

    Jednak w tym przypadku :

    ```
    k/d - co ile zapisow żądanie bedzie dotyczyć innego bloku
    a = (k - k/d + (x/n) * (k/d) ) / k = (n*(d - 1) + x) / (n*d)
    ```

    Co przy *d = 10* i *n =1000* oraz *x = 30* daje *a = 90,3%* !

    Średnia wynosi więc :

    ```
    Sr =  2*(1 - (n*(d - 1) + x) / (n*d) )*b*o
    ```

5.5.3 Autorzy zadań

1. Artur Jaźwiński
2. Michał Górka
3. Michał Szymaniak
4. Artur Jaźwiński
5. Tomasz Zielonka
6. Adam Barwiołek
7. Daniel Grabowski
8. Michał Żebrowski
9. Michał Szafrański
10. Tomasz Mosiej
11. Tomasz Mosiej
12. Walerian Sokołowski
13. Walerian Sokołowski
14. Weronika Krych
15. Weronika Krych
16. Jakub Pawlewicz
17. Piotr Gabryańczyk
18. Piotr Gabryańczyk

5.6 Autorzy

1. Adam Barwiołek (2) 2. Ziemowit Buchalski (5) 3. Piotr Gabryańczyk (5*) 4. Michał Górka (1) 5. Daniel Grabowski (5) 6. Artur Jaźwinski (3*) 7. Weronika Krych (3) 8. Tomasz Mosiej (2) 9. Jakub Pawlewicz (5) 10. Paweł Przeborowski (1) 11. Walerian Sokołowski (2) 12. Michał Szafrański (3) 13. Michał Szymaniak (3) 14. Tomasz Zielonka (1*) 15. Marek Żebrowski