Linux — Podsystem wejścia-wyjścia

7 stycznia 1999

Magdalena Bajtlik

1 Wprowadzenie

1.1 Podstawowe pojęcia

Podsystem wejścia-wyjścia z definicji służy do komunikacji procesów z urządzeniami zewnętrznymi (takimi jak dyski, terminale, sieć itp.). Specyficzne własności podprogramów obsługi urządzeń sprawiają, że ma on też drugie, niejako wtórne zastosowanie — korzysta się z niego dla ułatwionego dostępu do funkcji jądra systemu.Przed zagłębieniem się w szczegóły konieczne jest jednak zdefiniowanie podstawowych w tym rozdziale pojęć.

Plik specjalny to plik, którego i-węzeł:

- w polu typu zapisaną ma literę 'c' lub 'b',
- nie zawiera odwołań do danych,
- zawiera dwie jednobajtowe liczby zwane odpowiednio głównym i drugorzędnym numerem urządzenia.

Jeśli w systemie wymagana jest komunikacja procesów z pewnym urządzeniem zewnętrznym, to urządzeniu temu musi odpowiadać jakiś plik specjalny — przynajmniej jeden. Mogą za to istnieć pliki specjalne, które nie specyfikują żadnego fizycznego urządzenia. Mimo to, związane z nimi funkcje nazywane są podprogramem obsługi urządzenia — tak samo jak w przypadku plików specjalnych wyznaczających konkretne urządzenie zewnętrzne.

Tablice rozdzielcze to dwie statyczne tablice jądra: `chrdevs` i `blkdevs`, indeksowane od 0 do 255. Przechowują wartości typu strukturalngo.

Podprogram obsługi urządzenia to zbiór funkcji identyfikowany parą (typ, numer_główny) urządzenia, któremu przekazuje sterowanie funkcja systemowa wywołana w celu wykonania pewnej operacji na pliku specjalnym. Ważne: funkcje z tego zbioru wykonywane są w trybie jądra.

1.2 Pliki specjalne

Wspomniałam już, że i-węzeł pliku specjalnego zawiera trzy ważne wartości. Pierwsza z nich to litera 'b' lub 'c' w polu typu. Litera 'b' definiuje urządzenie jako blokowe (ang. block device), natomiast 'c' oznacza urządzenie znakowe (ang. character device). Podziału tego dokonuje się ze względu na sposób dostępu do danych (szerzej opisane w punkcie czwartym). Punkty wejścia do podprogramów obsługi urządzeń blokowych zgrupowane są w tablicy rozdzielczej urządzeń blokowych — blkdevs, natomiast urządzeniom znakowym odpowiada tablica chrdevs.

Druga ważna wartość zapisana w i-węźle pliku specjalnego to **numer główny urządzenia** (ang. major device number). Określa on rodzaj urządzenia — np. że jest to terminal określonego typu. Poza tym jest on indeksem, pod którym w odpowiedniej tablicy rozdzielczej zakodowana jest informacja o podprogramie obsługi danego urządzenia. Ostatnia liczba to **numer drugorzędny urządzenia** (ang. minor device number). Określa on numer jednostki danego urządzenia, i jest przekazywany podprogramowi obsługi podczas wszelkich operacji na pliku specjalnym.

Numer drugorzędny, podobnie jak numer główny, jest liczbą ośmiobitową. Oba te numery tworzą łącznie **numer urządzenia**, tzn.

numer_urządzenia = (numer_główny << 8) | numer_drugorzędny

Pliki specjalne tworzy się poleceniem **mknod** (dostępne tylko dla nadzorcy systemu). Przykładowo:

`mnknod jakistam c 67 12`

tworzy plik specjalny `jakistam`, oznacza go jako znakowy (c=znakowy, b=blokowy), pod numer główny urządzenia wpisuje 67, pod numer drugorzędny — 12. Nie musi istnieć dla tego pliku podprogram obsługi, nie musi nawet istnieć urządzenie fizyczne, któremu mógłby odpowiadać. Można taki odizolowany plik utworzyć; dopiero próba odczytu z niego zakończy się błędem.

1.3 Tablice rozdzielcze

Tablice rozdzielcze są to tablice podprogramów obsługi urządzeń — jedna dla urządzeń blokowych, a druga dla znakowych. Podprogram obsługi jest identyfikowany parą (litera, liczba). Litera mówi, w której tablicy należy go szukać, natomiast liczba — którą jest, przypominam, numer główny obsługiwanego urządzenia — pod którym indeksem znajduje się opisująca podprogram struktura device_struct. W strukturze tej znajduje się wskaźnik do nazwy podprogramu obsługi, oraz wskaźnik do odpowiadającej mu struktury file_operations. Od momentu, kiedy coś zostanie wpisane do tych pól, każda operacja na pliku specjalnym, z którego numerem głównym stowarzyszone są owe struktury, obsługiwana będzie przez funkcje, których adresy znajdują się w file_operations.

Podprogram obsługi jest uważany za zainstalowany w systemie tylko wtedy, kiedy zajmuje jakąś pozycję w tablicy rozdzielczej. Dokładniej, pozycja i-ta w którejś z tablic rozdzielczych uważana jest za pustą, jesli zawarty w strukturze chrdevs[i] czy blkdevs[i] wskaźnik na nazwę podprogramu obsługi jest nullem. Pozycja zerowa w obu tablicach nie jest używana i nie można pod nią niczego wpisać.

Każdy podprogram obsługi urządzenia, aby mógł być wykorzystywany, musi się zarejestrować odpowiednią funkcją register_chrdev/blkdev podając jej jako parametr numer główny, pod którym chce być dostępny. Poza tym często podprogram obsługi musi zainicjalizować swoje struktury danych, znaleźć związane z nim urządzenia itp. Wszystkie te czynności są wykonywane w funkcjach xxxx.init(), gdzie xxxx jest "nazwą" urządzenia. Z kolei funkcje xxxx_init są wywoływane przez chr_dev_init oraz blk_dev_init.

1.4 Podział na urządzenia znakowe i blokowe

Urządzenie nazywane jest znakowym, jeśli zezwala na dostęp do danych bajt po bajcie, jeżeli natomiast dane obsługiwane są w większych porcjach — blokach, to urządzenie klasyfikuje się jako blokowe. Przy obsłudze tych ostatnich zazwyczaj dane są buforowane. Podział ten jest silniej związany z plikami specjalnymi i podprogramami obsługi niż z samymi urządzeniami: może istnieć urządzenie fizyczne, z którym związane są dwa pliki specjalne (i, odpowiednio, podprogramy obsługi) — blokowy i znakowy. Fakt ten, w połączeniu z buforowaniem danych przy odczycie/zapisie z urządzenia blokowego, może być źródłem pewnych kłopotów.

Wyobraźmy sobie część dysku, której odpowiadają dwa pliki specjalne — blokowy i znakowy. Jeśli pewien proces zapisze dane do blokowego pliku specjalnego, a inny odczyta je spod tego samego adresu, lecz z surowego pliku specjalnego, to ten drugi proces może nie uzyskać danych zapisanych przez pierwszy. Dane te mogą bowiem znajdować się jeszcze w podręcznej pamięci buforowej, a nie na dysku. Jeśli natomiast drugi proces odczyta dane z blokowego pliku specjalnego, to automatycznie uzyska nowe dane w takiej postaci w jakiej istnieją one w pamięci podręcznej. Standardowo traktowane są jako blokowe

urządzenia takie jak dyski czy pamięć taśmowa, ze znakowych najważniejszą grupę stanowią terminale, potem myszy, modemy itp. Z każdego urządzenia blokowego można korzystać tak, jak ze znakowego — przy odpowiednim zminimalizowaniu wielkości bloku. Urządzenia blokowe są widziane przez resztę systemu jako urządzenia pamięci o dostępie bezpośrednim.

1.5 Funkcje open, close, write, read

W przypadku funkcji systemowych korzystających z deskryptorów plików jądro przechodzi od deskryptora pliku użytkownika do tablicy plików oraz i-węzła. Na podstawie i-węzła sprawdza typ pliku, a następnie odwołuje się do odpowiedniej tablicy rozdzielczej. Jądro pobiera z i-węzła pliku główny oraz drugorzędny numer urządzenia i wywołuje odpowiedni podprogram obsługi urządzenia, przekazując mu jako parametr numer drugorzędny.

Ważne: istotną różnicą pomiędzy wykonywaniem funkcji systemowych w stosunku do plików zwykłych oraz specjalnych jest nie blokowanie i-węzła pliku specjalnego na czas wykonywania przez jądro podprogramu obsługi urządzenia. Ma to za zadanie usunąć niebezpieczeństwo tego, iż zaśnięcie jednego procesu przy zblokowanym i-węźle uśpiłoby inne procesy korzystające z tego i-węzła.

1.5.1 Open

Faza wstępna otwarcia pliku specjalnego przez jądro przypomina operację otwarcia pliku zwykłego. Zostaje przydzielone miejsce w pamięci na i-węzeł, zwiększony jego licznik odwołań, przydziela się pozycję w tablicy plików i deskryptor. Nasępnie, tuż przed przekazaniem sterowania odpowiedniemu podprogramowi obsługi, jądro zapamiętuje kontekst procesu (taka asekuracja, na wypadek gdybyby trzeba było coś zrobić z procesem, zanim sam odda sterowanie — np kiedy zaśnie zbyt długo), i wreszcie wykonuje się podprogram obsługi.

```
algorytm open /* otwieranie plików specjalnych */
wejście: ścieżka, tryb otwarcia
wyjście: deskryptor pliku
{
przekształć nazwę ścieżkową na i-węzeł,
zwiększ licznik odwołań i-węzła,
przydziel pozycję w tablicy plików oraz deskryptor pliku użytkownika,
tak samo jak dla pliku zwykłego


pobierz z i-węzła główny oraz drugorzędny numer urządzenia;
zapamiętaj kontekst ( algorytm setjmp );

if ( urządzenie blokowe )
{
użyj numeru głównego jako indeksu tablicy rozdzielczej
urządzeń blokowych;
wywołaj odpowiednią procedurę otwarcia, przekazując jej
drugorządny numer urządzenia oraz tryb otwarcia;
}
else
{
użyj numeru głównego jako indeksu tablicy rozdzielczej
urządzeń znakowych;
wywołaj odpowiednią procedurę otwarcia, przekazując jej
drugorządny numer urządzenia oraz tryb otwarcia;
}
if ( operacja otwarcia zakończona niepomyślnie )
zmniejsz liczniki odwołań w tablicy plików oraz i-węźle;
}
```

Funkcje open specyficzne dla urządzenia mogą zajmować się rozmaitymi rzeczami, m.in.:

- Zależnie od numeru drugorzędnego wywoływać odpowiednią funkcję,
- Inicjalizować struktury danych związane z urządzeniem,
- Sprawdzać, czy urządzenie istnieje i zdobywać o nim informacje,
- Zliczać procesy korzystające z urządzenia

1.5.2 Close

Proces usuwa połączenie z otwartym urządzeniem zamykając je (close). Algorytm zamykania urządzenia jest podobny do algorytmu zamykania zwykłego pliku. Jednak przed zwolnieniem i-węzła jądro musi wykonać pewne operacje specyficzne dla danego urządzenia. Najpierw jądro przeszukuje tablicę plików, sprawdzając, czy jakiś inny proces ma jeszcze otwarte to urządzenie. Zbadanie licznika odwołań w tablicy plików nie wystarcza, gdyż jednemu urządzeniu może odpowiadać wiele plików specjalnych, oraz wiele i-węzłów.

Jeśli urządzenie jest znakowe, to jądro wywołuje procedurę zamknięcia i wraca do trybu użytkownika. Jeśli urządzenie jest blokowe, to jądro sprawdza, czy na urządzeniu jest zamontowany system plików, i jeśli tak, to procedura zamknięcia nie jest wywoływana, gdyż nie jest to ostatnia operacja zamknięcia wywoływana na tym urządzeniu. Gdy urządzenie nie zawiera zamontowanego systemu plików, to w podręcznej pamięci buforowej mogą nadal znajdować się dane pozostałe z poprzednio zamontowanego systemu plików, a nie zapisane z powodu użycia opcji opóźnionego zapisu. Jeśli takie dane zostaną odnalezione, to jądro przesyła je na urządzenie, i wreszcie wywołuje specyficzna procedurę zamknięcia. Po jej wykonaniu jądro znowu przeszukuje podręczną pamięć buforową, i unieważnia znalezione bufory z danymi zamkniętego urządzenia. Dopiero teraz można zwolnić i-węzeł pliku specjalnego.

```
algorytm   close
wejście: deskryptor pliku
wyjście: brak
{
wykonaj zwykły algorytm close;
if ( licznik odwołań w tablicy plików niezerowy )
goto koniec;
if ( istnieje inny otwarty plik, o tym samym numerze urządzenia )
goto koniec; /* nie jest to jednak ostatnie zamknięcie */
if ( urządzenie znakowe )
{
użyj numeru głównego jako indeksu tablicy rozdzielczej
urządzeń znakowych;
wywołaj odpowiednią procedurę zamknięcia,
przekazując jej drugorządny numer urządzenia
}
if ( urządzenie blokowe )
{
if ( urządzenie zamontowane )
goto koniec;
zapisz bloki urządzenia z buforowej pamięci podręcznej na urządzenie;
użyj numeru głównego jako indeksu tablicy rozdzielczej
urządzeń blokowych;
wywołaj odpowiednią procedurę zamknięcia,
przekazując jej drugorządny numer urządzenia
unieważnij bloki znajdujące się w podręcznej pamięci buforowej;
}
      koniec:
```

```
zwolnij i-węzeł;
}
```

1.5.3 Read / Write

Jeśli proces odczytuje lub zapisuje dane do pliku specjalnego, to jądro wywołuje procedurę odczytu lub zapisu specyficzną dla danego urządzenia, a komunikacja podprogramu obsługi z urządzeniem silnie zależy od sprzętu. Zwykle większość kodu podprogramu obsługi zajmuje się obsługą tych właśnie funkcji.

Popularne mechanizmy to m.in. **odwzorowanie wejścia-wyjścia na pamięć** i **programowane wejście-wyjście**. Pierwszy z nich polega na tym, że określone adresy z przestrzeni adresowej jądra nie są adresami pamięci fizycznej, lecz adresami specjalnych rejestrów służących do sterowania pracą poszczególnych urządzeń. Podczas zapisu do takich rejestrów danych sterujących, zgodnych z wymaganiami sprzętowymi, podprogram obsługi urządzenia steruje jego pracą. Występowanie programowanego wejścia-wyjścia oznacza, że komputer zawiera instrukcje maszynowe służące do sterowania pracą urządzeń. Podprogramy obsługi wykonują tylko odpowiednie instrukcje, a sposób ich komunikacji z konkretnym urządzeniem jest niewidoczny dla użytkownika.

1.6 Przerwania

Jeżeli podprogram obsługi otrzymuje zadanie wykonania pewnej operacji na tym urządzeniu i przekaże to zadanie do urządzenia, to musi kiedyś dowiedzieć się, że zostało wykonane (lub też zakończyło się błędem). Dotyczy to niemal wszystkich podprogramów obsługi urządzeń zewnętrznych, jak też niektórych urządzeń programowych, które np. wykonują read w sposób blokujący. Podrogram osługi

może zdobywać tę informację na dwa sposoby:

- co pewien czas sprawdzać, czy jego zadanie zostało wykonane (odpytywać urządzenie). Taki podprogram obsługi nazywamy odpytującym (ang. polling device driver)
- czekać na przerwanie od urządzenia, które w ten sposób zasygnalizuje wykonanie operacji.

Wystąpienie przerwania powoduje wykonanie przez jądro podprogramu obsługi przerwania, na podstawie korelacji pomiędzy urządzeniem generującym przerwanie a przesunięciem w tablicy wektorów przerwań. Jądro wywołuje podprogram obsługi przerwania związany z danym urządzeniem, przekazując mu numer urządzenia oraz inne parametry służące do identyfikacji urządzenia, które spowodowało przerwanie.

1.7 Jak oszukać Linuxa

Czasami, dla uzyskania nietrywialnych efektów, konieczne wydaje się napisanie nowej funkcji systemowej. Po prostu niektóre rzeczy można zrobić tylko w trybie jądra. W trybie jądra pracują także podprogramy obsługi urządzeń. Stąd metodą na dostęp do dowolnej części jądra systemu, dowolnego rejestru i pamięci operacyjnej komputera może być napisanie podprogramu obsługi specjalnego pseudourządzenia. Wywołanie funkcji systemowej na pliku odpowiadającym temu urządzeniu zaowocuje przekazaniem sterowania podprogramowi obsługi. Jest to nabardziej popularny sposób skłaniania Linuxa do wykonywania operacji, których jego twórcy wcale nie zamierzali realizować.

2 Podprogramy obsługi urządzeń

Marcin Stopyra

2.1 Wstęp

Podprogram obsługi urządzenia, czyli sterownik jest to poprostu zbiór funkcji dający systemowi dostęp do urzadzenia (zazwyczaj urządzenia zewnętrznego, ale nie zawsze np. dostęp do pamięci). Sterowniki udostępniają takie operacje jak: odczyt, zapis, otwarcie, zamknięcie czy zmiana konfiguracji urządzenia. Dostęp do tych funkcji jest realizowany właśnie za pomocą tablic rozdzielczych.

Dzięki sterownikom możemy korzystać z takich urządzeń jak: dyski, drukarka, mysz czy terminal.

2.2 Podział podprogramów obsługi urządzeń

W zależności od tego jak wygąda komunikacja między sterownikiem a urządzeniem rozróżniamy następujące typy sterowników :

- sterowniki odpytujące
- sterowniki przerwaniowe

Sterownik odpytujacy co pewien czas sprawdza (odpytuje) czy jego zadanie zostało wykonane. Realizowane jest to za pomocą aktywnego czekania. Podczas operacji odczytu/zapisu przez jeden proces wszystkie inne procesy są wstrzymywane do czasu zakończenia operacji na urządzeniu przez pierwszy proces. Zaleta tego typu sterowników jest prosta budowa, nie korzystanie z przerwań. Jednak blokowanie systemu decyduje o tym, że w praktyce nie stosuje się tego typu sterowników (poza niektórymi sterownikami np.do drukarek).

Drugim typem sterowników (powszechnie stosowanym) sa sterowniki przerwaniowe. W przeciwieństwie do sterowników odpytujących sterowniki te unikają aktywnego czekania dzięki przerwaniom. W momencie kiedy sterownik wyśle żądanie do urządzenia może zająć się innymi zadaniami. Kiedy urządzenie będzie gotowe wyśle przerwanie, które spowoduje uruchomienie procedury obsługi przerwania. Dzięki przerwaniom można wykorzystać takie mechanizmy pamięci buforowej jak: opóźniony zapis, czy czytanie naprzód.

Ze względu na efektywność działania większość urządzeń w systemie Linux korzysta właśnie z sterowników przerwaniowych, chociaż istnieje też możliwość używania sterowników odpytujących (np drukarki).

2.2.1 Porty

2.2.2 Wstęp

Porty są interfejsem pomiędzy podprogramami obsługi a fizycznymi urządzeniami. Poprzez porty przesyłane są dane miedzy urządzeniem a sterownikiem. Numery portów odpowiadają specjalnym rejestrom w komputerze. Każde urządzenie fizyczne ma przypisane na stałe określone numery portów (od 0 do 65536). Sterownik w momencie inicjalizacji musi sam znaleźc te porty, ewentualnie dowiedzieć się o nich (np. zapytać użytkownika). Może to zrobić np. wpisując pewne kody instrukcji do wybranych portów i czekać (przez pewien ustalony czas) na odpowiedź.

2.2.3 Rezerwacja portów

W celu ułatwienia sterownikom znajdywania portów przypisanych do urządzeń, które obsługują, system Linux udostępnia mechanizm rezerwacji portów. Każdy zarezerwowany obszar opisany jest przez następującą strukture :

```
struct resource_entry_t {
     unsigned long from;         - numer pierwszego zarezerwowanego
                                   portu
     unsigned long num;          - długość zarezerwowanego obszaru
     const char *name;           - nazwa urządzenia
     struct resource_entry_t *next;
}
```

Struktury resource_entry_t zawarte są w tablicy iotable o rozmiarze 128. Dodatkowo połączone są w uporządkowaną (względem pola from) listę iolist. Dostęp do listy iolist jest realizowany za pomocą funkcji :

- check_region
- request_region
- release_region

Funkcja check_region dostaje dwa argumenty from i num. Wynikiem jest odpowiedź czy obszar [from, from+num-1] jest już zarezerwowany.

Druga funkcja request_region po otrzymaniu from, num oraz nazwy urządzenia rezerwuje ten obszar (od from do from+num-1), uprzednio sprawdzając czy jest to możliwe.

Do usunięcia zarezerwowanego wcześniej obszaru portów z listy iolist służy funkcja release_region. Potrzebuje ona dwóch argumentów from i num.

Użytkownik może obejrzeć sobie liste zarezerwowanych obszarów portów w pliku /proc/ioports. Co ciekawe zarezerwowanie portów nie jest konieczne aby z nich korzystać. Podczas operacji odczytu/zapisu system operacyjny nie sprawdza czy dany port (porty) zostały uprzednio zarezerwowane i przez kogo. Jednak każdy sterownik powinien rezerwować swoje porty, na wypadek gdyby inny program (sterownik) chciał użyc tych portów.

2.2.4 Operacje odczytu/zapisu do portów

Do zapisu i odczytu z portów służą następujące funkcje :

- outb, outw, outl - zapisują do podanego portu wartość typu odpowiednio: char, short, int
- inb, inw, inl - jak wyżej tylko czytają z portu

- outsb, outsw, outsl - zapisują ciąg wartości o podanej długości z pod wskazanego adresu do portu bądź portów
- insb, insw,insl - podobnie jak wyżej, tyle że dotyczy odczytu

Dostepne są rownież te same funkcje z przyrostkiem _p (np. outb_p, inb_p itd.). Różnica polega na tym, że przed wyjściem z funkcji robia dodatkowo pause, dając czas sprzętowi na dokonanie odczytu/zapisu.

Dostęp do portów jest możliwy rownież za pośrednictwem pliku specjalnego /dev/port. Można wykonać na nim operacje: lseek, read i write. Jest to jednak wolniejszy sposób (operacje te korzystają z funkcji outb i inb), ale dzięki niemu użytkownik (jeśli ma do tego prawa) może bezpośrednio pisać badź czytać z portów.

2.3 Przerwania

2.3.1 Wstęp

Przerwania sprzętowe, jak zostało to już wcześniej powiedziane, służą do komunikowania się urządzeń z sterownikami programowymi. W momencie kiedy urządzenie jest gotowe do działania, zokończyło operacje wejścia-wyjścia lub wystąpił błąd urzadzenie generuje przerwanie, ktore to nastepnie uruchamia procedure obsługi przerwania.

2.3.2 Rejestracja przerwań

Podobnie jak w przypadku portów sterownik musi zarejestrować swoją procedure obsługi przerwań na odpowiedniej lini przerwań (jest ich 16). W systemie pamiętana jest tablica irq_action (o rozmiarze 16) wszystkich procedur obslugi przerwań. Elementami tej tablicy są listy następujących struktur :

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

Znaczenie poszczególnych pól :

- handler - wskaźnik do funkcji obsługującej przerwanie; jej argumentami są: numer lini przerwania, wskaźnik do struktury opisującej urządzenie, wskaźnik do struktury opisującej rejestry procesora
- flags - flagi (zostaną omówione poźniej): SA_INTERRUPT, SA_SHIRQ, SA_SAMPLE_RANDOM lub 0
- mask - wskazuje, które przerwania są zamaskowane
- name - nazwa urządzenia

- dev_id - wskaźnik do struktury opisującej urządzenie

Do zarejestrowania procedury obsługi przerwania służy funkcja request_irq. Jej argumentami sa pola handler, flags, name i dev_id. W zależności od tego jak ma być wywołana procedura obsługi przerwania stosowane są następujące flagi :

- SA_INTERRUPT - Jeśli ta flaga jest ustawiona to definiujemy tzw. szybkie przerwanie. Oznacza to ,że podczas obsługi przerwania wszystkie inne przerwania zostaną zamaskowane i po wyjściu nie nastąpi szeregowanie. W przeciwnym przypadku będzie zamaskowane tylko to przerwanie, ktore zostało wygenerowane i po wyjściu z niego zostanie wywołana funkcja schedule() (jeśli będzie taka potrzeba).
- SA_SHIRQ - Dzięki tej fladze możliwe jest współdzielenie tej samej linii przerwań przez kilka innych procedur obsługi przerwań. Każda taka procedura podczas rejestracji musi użyć tej flagi
- SA_SAMPLE_RANDOM - Ta flaga informuje tylko, czy to przerwanie może być użyte do generatora liczb losowych (wołana jest wtedy funkcja add_interrupt_randomness)

Do celu odrejetrowania przerwania służy funkcja free_irq z argumentem będącym numerem przerwania. Do szukania linii przerwań uzywanej przez dane urządzenie służą funkcje :

- probe_irq_on
- probe_irq_of

Pierwsza z nich jako wynik (nie ma parametrów) zwraca maskę z ustawionymi bitami odpowiadajacymi aktywnym przerwaniom, dla których nie zarejestrowano procedury obsługi. Druga otrzymując maskę przerwań jako parametr zwraca numer przerwania, które nastąpiło (0 - jeśli nie było żadnego, liczba ujemna - jeśli było ich więcej)

Tak więc przykładowy algorytm szukania przerwania może wyglądać następująco:

- irqs= probe_irq_on
- wyślij prośbę o przerwanie
- czekaj na przerwanie
- irq= probe_irq_of(irqs)
- zarejestruj procedure obsługi przerwania na irq linii

2.3.3 Procedury dolnej połowy

Procedura obsługi przerwania musi wykonać często czasochłonne zadania. Zazwyczaj podczas obsługi przerwania maskowane są inne przerwania. Istnieje wtedy ryzyko, że pewne przerwania nie będą obsłużone. W tym właśnie celu dzieli się procedure obsługi przerwania na dwie części :

- procedura górnej połowy - Jej wskaźnik znajduje się w tablicy przerwań irq_action. Wykonuje ona tylko niezbędne czynności jak sprawdzenie stanu urządzenia, czy kopiowanie danych z urządzenia do bufora w pamięci.

- procedura dolnej połowy - Wykonuje pozostałe czynności, które mogą być wykonane później. Podczas jej wykonywania przerwania są odmaskowane (może wykonywać się dowolnie długo).

Wskaźniki do procedur dolnej połowy trzymane są w tablicy bh_base (o rozmiarze 32). Do wstawienia wskaźnika procedury dolnej połowy do tablicy służy funkcja init_bh. Podczas obsługi przerwania procedura górnej połowy zwiększa licznik ilości wywołań danej procedury dolnej połowy, znajdującej się w tablicy bh_base, używając funkcji mark_bh. Wywołanie zaznaczonych procedur następuje po obsłudze wszystkich przerwań na początku funkcji schedule() lub return_from_sys_call(). Każda procedura, ktorej licznik został zwiększony, wykonuje się odpowiednią ilość razy, w kolejności zależnej od pozycji zajmowanej w tablicy bh_base.

2.3.4 Kolejki zadań

Ilość miejsca w tablicy bh_base, a tym samym ilość procedur dolnej połowy, które mogą być wykonane, jest ograniczona. Rozwiązaniem tego problemu mogą być kolejki zadań. Kalejka zadań jest to poprostu lista następujących struktur:

```
struct tq_struct {
    int sync;              - pomocnicze pole (przy wstawianiu
                             i usuwaniu z kolejek) o wartościach
                             0 lub 1
    void (*routine)(void *);    - wskaźnik do wołanej funkcji
    void *data;                 - wskaźnik do argumentów
    struct tq_struct *next;
}
```

Do obsługi kolejek zadań są dostępne następujące funkcje :

- queue_task - dołącza nową strukture do podanej kolejki

- run_task_queue - Jej argumentem jest kolejka, której procedury zostaną wykonane. Każda procedura jest usuwana z kolejki a następnie wykonywana.

Przykładowy algorytm procedury dolnej połowy, która korzysta z kolejek zadań:

- taskq= NULL // nowa kolejka zadań

- wstaw wskaźniki procedur z których chcesz korzystac do kolejki taskq, za pomocą queue_task

- run_task_queue (taskq)

Można również skorzystać z zdefiniowanej już kolejki tq_immediate, która jest uruchamiana przez procedure na pozycji 7 (chyba) w tablicy bh_base.

3 Urządzenia blokowe

Sławomir Joński

3.1 Wstęp

Urządzenia blokowe charakteryzują się tym, że dostęp do nich sprowadza są do zapisu lub odczytu bloków o stałej wielości . Aby usprawnić komunikację z tymi urządzeniami Linux dostarcza mechanizm pamięci buforowej. Bufory są to struktury w pamięci jądra reprezentujące wybrane bloki z urządzeń blokowych. Jeśli na przykład chcemy odczytać pewien blok z urządzenia, a jego aktualna kopia znajduje się akurat w pamięci to nie musimy wykonywać stosunkowo wolnej operacji odczytu bloku z urządzenia. Pamięć buforowa może również zapewniać strategię opóźnionego zapisu oraz odczytu naprzód.

Przyjrzyjmy się teraz schematowi komunikacji z urządzeniami blokowymi. Żądanie odczytu lub zapisu kierowane jest najpierw do pamięci buforowej. W przypadku żądania odczytu, gdy nie ma w pamięci buforowej odpowiednich bloków, żądanie odkładane jest na pewną kolejkę z której żądania są pobierane przez urządzenie (niekoniecznie po jednym). W oczekiwaniu na dane, system operacyjny może wykonywać inne zadania. Urządzenie po wykonaniu pobranych żądań wysyła przerwanie , które budzi sterownik urządzenia (podprogram obsługi), który zajmuje się odesłaniem danych do pamięci buforowej, skąd mogą być one pobrane przez proces. W przypadku żądania zapisu, system może zdecydować o odłożeniu na kolejkę żądania zapisu albo też może przetrzymać dane żądanie na pewien okres (opóźniony zapis)

I tak każde urządzenie blokowe zdefiniowane jest przez strukturę blk_dev_struct:

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

Request_fn jest wskaźnikiem na procedurę strategii (strategy routine) - najważniejszą część programu obsługi urządzenia blokowego, zajmującą się szeregowaniem żądań. Current_request jest zaś kolejką struktur opisujących żądania wejścia-wyjścia skierowane do urządzenia. Z tej to właśnie kolejki procedura strategii pobiera żądanie i inicjuje jego wykonanie.

Urządzenia blokowe nie implementują własnych funkcji read ani write. Korzystają za to ze wspólnych dla wszystkich urządzeń blokowych funkcji block_read i block_write. Ani block_read ani block_write nic bezpośrednio nie zapisują ani nie czytają, zlecają natomiast odpowiednie operacje funkcji ll_rw_block. Funkcja ta także jest używana przez wszystkie urządzenia blokowe. Zadaniem jej jest wstawianie żądań wczytania i zapisu danych do kolejki żądań wejścia-wyjścia odpowiedniego urządzenia. Obsługą tych żądań zajmuje się już później konkretny sterownik, a właściwie jego funkcja strategii, używając niskopoziomowych funkcji do zapisywania i odczytywania danych.

3.2 Podstawowe funkcje programów obsługi urządzeń

Przyjrzyjmy się najpierw mechanizmowi komunikacji programu obsługi urządzenia z urządzeniem blokowym. Najpierw mamy kolejkę żądań do urządzenia. Program obsługi pobiera pierwsze żądanie z kolejki i informuje dysk o akcji do wykonania. Dopiero gdy dysk jest gotowy przesyła przerwanie i dochodzi do transmisji danych.

3.2.1 Funkcje block_read i block_write

Jak już wcześniej napisałem, programy obsługi urządzeń blokowych nie definiują własnych funkcji do zapisu ani do czytania, korzystają natomiast ze wspólnych funkcji block_read i block_write. Przyjrzyjmy się algorytmowi funkcji block_read.

DEFINICJA:

```
long block\_read
  ( struct inode     *inode   // i-wezeł pliku specjalnego
    struct file      *filp    // opis pliku specjalnego w tablicy
                              // plików
    char             *buf     // bufor w przestrzeni użytkownika
                              // na odczytane dane
    unsigned long    count )  // liczba bajtów do przeczytania
```

WYNIK: ilość przeczytanych bajtów lub kod błędu EIO

```
{
      określamy, które bloki dyskowe mamy przeczytać
      pobieramy algorytmem getblk nagłówki dla każdego bloku

      dopóki nie przeczytano wszystkich bajtów {
        wywołujemy ll_rw_block - wstawienie żądania do kolejki;
        wait_on_buffer - czekaj na wczytanie bufora
        przepisz dane z bufora do przestrzeni użytkownika

      }

      zwolnij przydzielone bufory  wywołując dla każdego brelse
      return  (liczba przeczytanych bajtów);

}
```

Algorytm funkcji block_write jest analogiczny

3.2.2 Szeregowanie żądań

Jak już wcześniej napisałem każde urządzenie blokowe, a właściwie program jego obsługi, ma w opisującej go strukturze blk_dev_struct wskaźnik na posortowaną listę żądań skierowanych do konkretnego urządzenia. Każde żądanie opisane jest przez strukturę request :

```
struct request {
   volatile int rq_status;   -    może przyjmować wartość:
                                       RQ_ACTIVE - zwykłe żądanie,
                                       RQ_INACTIVE
                                         - żądanie już wykonane
                                       lub RQ_SCSI_BUSY,
                                       RQ_SCSI_DONE
                                       RQ_SCSI_DISCONNECTING
   kdev_t rq_dev;            -    określa urządzenie
                                  (numer główny i drugorzędny),
                                  do którego skierowane jest
                                  żądanie
   int cmd;                  -    rodzaj żądania, może być
                                  równy READ albo WRITE;
   int errors;               -    licznik błędów otrzymanych
                                  podczas realizowania żądania
                                  -jeśli przekroczy pewną wartość,
                                  określaną przez program obsługi
                                  urządzenia, to żądanie zostaje
                                  usunięte i nieobsłużone,
   unsigned long sector;     -    początkowy sektor, którego
                                  dotyczy żądanie
   unsigned long nr_sectors; -    długość fragmentu,
                                  którego dotyczy żądanie
   unsigned long current_nr_sectors;
                             -    liczba sektorów, która pozostała
                                  do wczytania/zapisania,
   char * buffer;            -    wskaźnik do danych zawartych
                                  w buforach
   struct semaphore * sem;   -    semafor używany przez programy
                                  obsługi urządzeń SCSI
                                  przy korzystaniu z pliku wymiany;
   struct buffer_head * bh;  -    wskaźnik na początek listy
                                  buforów odpowiadających
                                  fragmentowi urządzenia
   struct buffer_head * bhtail;
                             -    wskaźnik na koniec listy buforów
                                  odpowiadających fragmentowi
                                  urządzenia
   struct request * next;
}
```

Za tworzenie nowego żądania i wstawienie go w kolejkę żądań do urządzenia odpowiada, wspólna dla wszystkich urządzeń blokowych funkcja ll_rw_block.

Najpierw funkcja ll_rw_block wstawia żądanie do tablicy all_requests w której znajdują się żądania wszystkich urządzeń blokowych. Tablica all_requests ma rozmiar wielkości 64 elementów z czego żądania zapisu mogą zawierać co najwyżej 2/3 kolejki. Proces, który chce zlecić urządzeniu operację wejścia-wyjścia, a nie znajdzie wolnej pozycji w tablicy all_requests zawiesza się na kolejce wait_for_request. Po wstawieniu do tablicy, każde nowe żądanie wstawiane jest na listę żądań dla konkretnego urządzenia. Żądania są wsortowywane za pomocą algorytmu windowego (ang. elevator algorithm), który minimalizuje czas ruchu głowicy. Jest to realizowane poprzez utrzymywanie listy posortowanej leksykograficznie po numerach drugorzędnych urządzenia, do którego żądanie jest skierowane (numery główne urządzeń w obrębie kolejki są takie same) i po pierwszym sektorze, którego żądanie dotyczy.

Sama funkcja ll_rw_block dostaje jako parametr listę buforów do których ma wczytać dane, następnie sprawdza, czy wielkość buforów odpowiada wielkości bloków urządzenia, czy urządzenie nie jest tylko do odczytu jeśli żądanie dotyczy zapisu danych. W końcu wywołuje dla każdego bufora funkcję make_request.

Funkcja ll_rw_block korzysta z dwóch funkcji make_request i add_request. Funkcja make_request tworzy odpowiednią strukturę żądania request. Następnie próbuje ją wstawić do tablicy all_requests. Jeśli tablica jest pełna to funkcja zawiesza się na kolejce wait_for_requests aż do zwolnienia się miejsca. Następnie wywoływana jest funkcja add_request, która wstawia odpowiednie żądanie (będące już w tablicy all_requests) w kolejkę żądań urządzenia.

3.2.3 Procedura strategii

Procedura strategii jest chyba najważniejszą częścią programu obsługi urządzenia blokowego. Zapewnia komunikację miedzy kolejką żądań a driverem urządzenia. Ogólnie rzecz biorąc funkcja strategii pobiera pierwsze żądanie z kolejki i inicjuje jego realizacje. Procedura strategii wywoływana jest zawsze po nadejściu żądania do pustej kolejki lub przez funkcję kończącą realizację poprzedniego żądania gdy kolejka nie jest pusta.

Większość programów obsługi urządzeń używa podobnych algorytmów funkcji strategii. Oto schemat działania takiej funkcji.

```
void procedura_strategii( void )
{
    if ( lista żądań pusta )
        wyłącz obsługę przerwań od urządzenia;
        return;
    if( ustawiona obsługa przerwania
        (co znaczy że czekamy na zakończenie obsługi jakiegoś
                                                    żądania) )
        return;
    żądanie - pierwsze żądanie z listy;
    if ( żądanie dotyczy zapisu )
        ustaw procedurę obsługi przerwania;
        wywołaj niskopoziomową procedurę zapisu;
        return;
    if ( żądanie dotyczy odczytu )
        ustaw procedurę obsługi przerwania - inną niż dla zapisu;
        wywołaj niskopoziomową procedurę odczytu;
        return;
    nieznane polecenie - panikuj (panic);
}
```

Wywołanie procedury strategii "opakowane" jest w procedurę, która wyłącza sprzętowe przerwanie od urządzenia przed wywołaniem procedury strategii i włącza je po powrocie z procedury strategii.

3.3 Podprogram obsługi dysku twardego

Opiszemy teraz działanie konkretnego programu obsługi dysku na przykładzie driveru hd.c. Program hd.c narzuca następujące warunki na dyski które obsługuje :

- dyski zarejestrowane są w tablicy urządzeń blokowych pod numerem 3
- max liczba dysków obsługiwanych przez driver wynosi 2
- każdy dysk może posiadać max 64 partycje.
- rozmiar sektora dysku 512 bajtów.
- numer partycji to 6 najmłodszych bitów numeru urządzenia
- 7-my wskazuje fizyczny numer dysku (0 lub 1)
- 8-my jest niewykorzystywany

3.3.1 Struktura file_operations dla dysku twardego

```
static struct file_operations hd_fops = {
        NULL,
        block_read,
        block_write,
        NULL,
        NULL,
        hd_ioctl,
        NULL,
        hd_open,
        hd_release,
        block_fsync

};
```

Oczywiście funkcje block_read i block_write są ogólnymi funkcjami używanymi przy korzystaniu z urządzeń blokowych, w związku z czym driver hd.c już ich nie definiuje.

Za pomocą funkcji hd_ioctl można (w zależności od podanego parametru) pobrać geometrię dysku, odczytać tablice partycji, ustawić ile sektorów pliku należy czytać z wyprzedzeniem, ustawić partycję tylko do czytania itp. Hd_open jest funkcją do otwierania partycji dysku twardego. Do zamykania partycji służy zaś funkcja hd_release.

3.3.2 Struktury danych opisujące dysk i funkcje inicjujące

Program hd.c korzysta z wielu różnych struktur opisujących dysk - jego parametry, typ itp. Przyjrzyjmy się podstawowym strukturom aby lepiej zrozumieć później działanie funkcji driveru.

Strukturą do przechowywania najważniejszych informacji o dysku jest struktura hd_i_struct:

```
struct hd_i_struct {
      unsigned int head     - liczba głowic
                   sect     - liczba sektorów na ścieżce
                    cyl     - liczba cylindrów
                  wpcom
                  lzone
                    ctl
};
```

Są zdefiniowane także tablice struct hd_i_struct hd_info[], struct hd_i_struct bios_info[] przechowujące informacje o dysku. Są one inicjowane w funkcji hd_geninit, która wczytuje dane z BIOS-u.

Struktura hd_struct używana jest do opisu partycji dyskowej :

```
struct hd_struct {
        long start_sect - sektor początkowy partycji
        long nr_sects   - liczba sektorów w partycji
};
```

Dane o typie dysku przechowywane są natomiast w specjalnej strukturze wspólnej dla wszystkich programów obsługi urządzeń blokowych :

```
struct gendisk  {
  int major               - numer główny typu dysku
  char *major_name:       - identyfikator typu dysku
  int minor_shift;        - liczba bitów jakie należy obciąć
                            w numerze drugorzędnym
                            aby otrzymać fizyczny numer dysku
  int max_p;              - maksymalna ilość partycji na jednym dysku
  int max_nr;             - maksymalna liczba dysków obsługiwanych
                            przez driver tego typu dysku

  void (*init)(void)      - funkcja inicjująca dane dysku
  struct hd_struct *part  - tablica partycji dysków obsługiwanych
                            przez driver
  int *sizes              - rozmiary partycji
  int nr_real             - rzeczywista liczba dysków w systemie
  void *real_devices;     - wskaźnik do tablicy z danymi o dyskach
  struct gendisk *next;   - wskaźnik do następnej struktury gendisk
}
```

Dla wszystkich typów dysków w systemie tworzona jest kolejka gendisk_head struktur gendisk.

Dla dysków obsługiwanych przez driver hd.c struktura gendisk wygląda następująco :

```
static struct gendisk hd_gendisk {
  MAJOR_NR   numer główny MAJOR_NR==3
  "hd"       nazwa
  6          należy obciąć 6 najmłodszych bitów w numerze
             drugorzednym by uzyskac fizyczny numer dysku
  1 << 6     jeden dysk moze zawierac max 64 partycje
  MAX_HD     max liczba dyskow 2
  hd_geninit funkcja inicjująca dysk.
  hd         wskaźnik do tablicy partycji dysków
  hd_sizes   tablica rozmiarów poszczegolnych partycji
  0          ilość urządzeń wykrytych w systemie, zostanie
             ustalona w funkcji inicjujacej hd_geninit
  (void *)bios_info
             informacje o dyskach (ile sectorow na sciezke itp.)
  NULL       nastepny geninit dla innych dyskow  zostanie ustalony
             przy rejestracji urzadzenia
};
```

Funkcje hd_request (funkcja startegii driveru hd.c) i hd_interrupt zostaną omówione w następnym rozdziale.

Omówmy teraz funkcje inicjujące dane dysku. Kiedy system startuje, w funkcji main.c wywoływana jest funkcja rejestrująca hd_init. Następnie dla każdego typu dysku (każdej struktury gendisk z kolejki gendisk_head) wywoływana jest funkcja setup_dev, która wczytuje partycje z dysku oraz wywołuje funkcję hd_geninit.

Funkcja inicjująca podstawowe dane dysku :

void hd_geninit(void)

Algorytm :

0. sprawdza ile dysków znajduje się w systemie i zapisuje w hd_gendisk.nr_real
1. pobiera dane z BIOS-u o dyskach i wczytuje do tablic hd_info[], bios_info[]
2. ustawia pierwszą partycję dla każdego dysku. Pierwsza partycja obejmuje cały dysk.
3. ustawia funkcję która ma się wywołać gdy przyjdzie przerwanie dyskowe HD_IRQ
   `request_irq(HD_IRQ, hd_interrupt, SA_INTERRUPT, ''hd'')`
4. rezerwuje porty do komunikacji
   `request_region(HD_DATA, 8, ''hd'')` — zwykłe porty
   `request_region(HD_CMD, 1, ''hd(cmd)'')` — port używany do resetowania
5. ustawia rozmiar bufora w zadaniu do partycji dysku

Rejestracja dysku w tablicy urządzeń blokowych:
unsigned long hd_init (unsigned long mem_start, unsigned long mem_end)
Algorytm :

1. Rejestruje typ dysku w tablicy urządzeń blokowych
   `register_blkdev(MAJOR_NR,''hd'',&hd_fops)`
   (MAJOR_NR - dla omawianych dysków wynosi 3 hd_fops - wskaźnik do struktury file_operation dysku twardego)

2. ustawia funkcję strategii dla dysku
   `blk_dev[MAJOR_NR].request_fn = hd_request`
   (blk_dev[MAJOR_NR] jest strukturą blk_dev_struct, która przechowuje kolejkę żądań do urządzenia i funkcję strategii realizującą żądania)

3. wstawia strukturę hd_gendisk do kolejki gendisk_head danych o dyskach

4. ustawia funkcję hd_times_out by była wywoływana w przypadku gdy dysk zbyt długo nie przysyła przerwań.

3.3.3 Funkcje driveru hd.c

Dysk twardy komunikuje się z komputerem poprzez kontroler przy pomocy przerwań (przerwania powodują wywołanie odpowiedniej funkcji obsługi żądania np. read_intr - czytanie danych z dysku) i przez odpowiednie porty (porty do komunikacji są ustalane w czasie inicjowania danych opisujących dysk). Porty służą do wymiany informacji między dyskiem a komputerem.

Funkcje driveru informują dysk o akcji do wykonania przy pomocy funkcji outb_p(komenda,HD_CMMAND), gdzie HD_COMMAND jest numerem portu.

Częściej używane komendy dla dysku :

- WIN_READ — żądanie czytania z dysku
- WIN_WRITE — żądanie pisania na dysk
- WIN_RESTOR — używane przy recalibrowaniu

Funkcje driveru mogą pobierać przez odpowiednie porty stany dysku lub sprawdzić czy dysk nie zwrócił błędu. Dzięki temu funkcje driveru mogą ustalić kiedy dochodzi do sytuacji nieprawidłowych. W przypadku sytuacji błędnej dysk jest **rekalibrowany** lub **resetowany**.

Driver hd.c definiuje następujące funkcje:

- hd_out
- hd_request - funkcja strategii
- do_special_op
- bad_rw_intr
- reset_hd

Ponadto korzysta z następujących funkcji wywoływanych przez przerwania:

- recal_intr
- set_mult_mode_intr

- identify_intr
- read_intr
- write_intr

Przyjrzyjmy się teraz funkcjom tego programu :
hd_out

Funkcja hd_out informuje dysk o zadaniu do wykonania i ustawia funkcję , która zostanie wywołana w najbliższym przerwaniu.

bad_rw_intr

Funkcja ta jest wywoływana przez funkcje driveru uruchamiane w przerwaniach w przypadku gdy dysk znajduje się w nieprawidłowym stanie. Funkcja bad_rw_intr sprawdza i reaguje na błędy dysku - próbuje im przeciwdziałać. Ustawia rekalibrowanie lub resetowanie, może także usunąć żądanie którego zbyt długo nie zrealizowano.

do_special_op

Funkcja ta jest wywoływana przez funkcję strategii hd_request jeśli wcześniej zostało ustawione wywołanie operacji specjalnych. Funkcja do_special_op przy pomocy hd_out wysyła komendę specjalną dla dysku: recal_intr, identify_intr lub set_multmode_intr.

reset_hd

Funkcja ta restartuje kontroler dysków i ustawia rekalibrowanie wszystkich dysków

Funkcja strategii dysku void hd_request(void)

Funkcja ta pobiera kolejne żądania dla dysku i zawiadamia dysk o żądaniu (funkcja hd_out). Hd_request wywoływana jest gdy przyjdzie żądanie do pustej kolejki. Dopóki kolejka żądań nie będzie pusta funkcje kończące realizację żądania (funkcje wywoływane w przerwaniach) będą wywoływać funkcje strategii

Funkcje wywoływane przez przerwania dysku

Funkcja która ma być wykonana w najbliższym przerwaniu pamiętana jest na zmiennej DEVICE_INTR. Kiedy przychodzi przerwanie od dysku wykonuje się funkcja hd_interrupt

```
static void hd_interrupt() {
  void (*handler)(void) = DEVICE_INTR;
                               /*pobranie funkcji do wykonania*/
  DEVICE_INTR = NULL /*wyczyszczenie zmiennej*/
  timer_active &= ~(1*HD_TIMER)
```

```
  /* zaznaczenie faktu, ze przyszło przerwanie */
  if (!handler)
      handler = unexpected_hd_interrupt;
      /*funkcja która obsługuje nieoczekiwane przerwania */
  handler()
  sti()
  /*przywrócenie obsługi przerwań która została zablokowana
                                      przy wykonywaniu handlera*/
}
```

Funkcje które mogą zostać wywołane przez przerwanie :

read_intr () wczytuje dane z dysku.

write_intr () przekazuje dane do dysku do zapisu lecz tylko po jednym sektorze przy jednym wywołaniu funkcji.

read_intr () podobnie jak write_intr lecz może przekazać max tyle sektorów przy jednym wywołaniu ile jest ustawionych dla urządzenia dysku. Wartość tą można zmienić przy pomocy funkcji hd_ioctl.

recal_intr() funkcja ta jest używana przy recalibrowaniu dysku. Najpierw dysk wykonuje komendę WIN_RESTORE, następnie recal_intr sprawdza stan dysku. Jeśli jest on nie prawidłowy to wywoływana jest funkcja bad_rw_intr, która próbuje cos zaradzić.

set_multimode_intr() ustawia nową liczbę sektorów transmitowanych przy jednej operacji czytania/pisania.

indetify_intr() pobiera dane o dysku.

4 Urządzenia znakowe

Mirosław Makaroś

4.1 Wtęp

Niestety nie ma żadnej definicji urządzeń znakowych, są to po prostu te urządzenia które nie są blokowe a dostęp do danych jest bajt po bajcie. Urządzenia znakowe bowiem nie mają w zasadzie żadnej wspólnej cechy, ani żadnych specjalnych właściwości tak jak urządzenia blokowe. Do urządzeń znakowych zaliczamy więc wszystkie urządzenia terminalowe, urządzenia sieciowe itp. W zasadzie o samych urządzeniach znakowych nie da się powiedzieć nic szczególnego dlatego też przejdę do omawiania najważniejszej grupy tych urządzeń jakimi są urządzenia terminalowe.

4.2 Urządzenia terminalowe

Urządzenia terminalowe są w zasadzie zwykłymi urządzeniami znakowymi, ich zadaniem jest transmisja danych z i do terminala. Z drugiej jednak strony są one dość specyficznymi urządzeniami ponieważ pośredniczą między użytkownikiem a systemem. Ponieważ chcielibyśmy aby system udostępniał nam możliwość interakcyjnego korzystania z niego więc urządzenia terminalowe muszą w pewien specjalny sposób traktować wchodzące i wychodzące dane.

Zazwyczaj użytkownik wprowadza jakieś dane z klawiatury a zadaniem urządzenia terminalowego jest odebranie danych informacji i przekazanie ich do odbierającego je procesu. Wiemy, że przy wpisywaniu danych z klawiatury często popełniamy błędy używamy wówczas klawiszy ścierających takich jak **DEL** lub **BACKSPACE**, nie miałoby więc sensu przekazywanie po kolei wszystkiego do procesu bo tak naprawdę potrzeba przekazać tylko tą końcową postać jaką zatwierdzamy Enterem. Właśnie taką wstępną obróbkę wprowadzanych danych zapewnia urządzenie terminalowe dzieki tak zwanej dyscyplinie linii. Inną rzeczą jaką daje nam dyscyplina linii jest to, aby pewne stałe kombinacje klawiszy były w jakiś standardowy sposób interpretowane przez system. Dyscyplina linii jest odpowiedzialna także za to aby na monitorze zastało wyświetlone echo wprowadzanych znaków. Inną ważną rzeczą przy pracy z systemem jest to, że gdy użytkownik z nim pracuje i zaczyna tworzyć procesy to powinny one być jakoś związane z urządzeniem aby w chwili zakończenia pracy mogły być o tym fakcie jakoś powiadomione, a więc terminal powinien wysyłać sygnały do procesów z nim związanych.

Jest też istotne aby zapewnić obsługę takich kombinacji klawiszy jak **Ctr—Z** czy też **Ctr—C**. A więc podsumowując dyscyplina linii odpowiedzialna jest między innymi za:

- przetwarzanie znaków ścierających
- przetwarzanie znaku kasowania wiersza
- wypisywanie echa odebranych znaków
- dzielenie wejściowego ciągu znaków na wiersze
- zamienianie klawisza tabulacji na ciągi spacji

- wysyłanie sygnałów do procesów informując o zawieszeniu terminala, zerwaniu z nim łącznosci
- zapewnianie obsługi pewnych specjalnych kombinacji klawiszy takich jak **Ctr—Z**, **Ctr—C**

Poza tym, co daje nam zaimplementowana w urządzeniach terminalowych dyscyplina linii, urządzenia te dają nam jeszcze inne rzeczy. Konsole pozwalają na wstępną obróbkę znaków wprowadzanych z klawiatury zanim przejdą one do dyscypliny linii, ponieważ niektórych kombinacji klawiszy np. przy przełączaniu się między sesjami nie potrzebujemy przekazywać a ponadto nie powinny być one widoczne i mieć jakiegokolwiek wpływu na system. Inny rodzaj urządzeń terminalowych, jakim są pseudoterminale, pozwala na zdalne zalogowanie się do systemu.

4.3 Numeracja urządzeń

Wszystkie urządzenia terminalowe mają przyporządkowany numer glówny równy 4. Natomiast numeracja numerami drugorzędnymi jest następujaca:

- konsole numery od 1 do 63
- terminale podłączone do linii szeregowych numery 64 — 127
- pseudoterminale nadrzędne numery 128 — 191
- pseudoterminale podrzędne numery 192 — 255, przy czym pseudoterminalowi nadrzędnemu o numerze n odpowiada pseudoterminal podrzędny o numerze n+64

Jest też urządzenie które określa aktywną (bieżącą) konsole, jego numerem glównym jest 4 natomiast drugorzędnym 0.

4.4 Struktury

4.4.1 TERMIOS

Struktura termios zawiera pewne stałe parametry pracy urządzenia terminalowego, informacje które przechowuje są wykorzystywane przez sterownik urządzeń oraz dyscyplinę linii. W zależności od ustawień poszczególnych flag pamiętanych w tej strukturze można określić w jaki sposób i czy w ogóle przetwarzać odbierane lub przesyłane do urządzenia znaki. Pola tej struktury w zależności od ustawionych lub wyzerowanych poszczególnych bitów opisują poszczególne flagi. Istnieje też prawie identyczna struktura termio która rożni się tylko wielkością poszczególnych pól, ale obecnie w nowych wersjach Linuxa nie jest ona już używana. Struktura termios ma sześć pól:

1. c_iflag są to flagi wejściowe tzn. określają sposób przetwarzania znaków otrzymanych od urządzenia
2. c_oflag są to flagi wyjściowe, określają jak wyglądają znaki wysyłane do urządzenia

3. c_cflag flagi sterujące, definiują parametry pracy urządzenia terminalowego takie jak szybkość transmisji, ilość bitów na jeden znak itp.

4. c_lflag określają sposób działania dyscypliny linii, m.in. określają sposób wyświetlania echa znaków na ekran, definiują jak wygląda znak końca wiersza itp.

5. c_line zapamiętuje jaka dyscyplina linii jest związana z danym urządzeniem terminalowym

6. c_cc[$NCCS$] jest to tablica zawierająca znaki specjalne (sterujące)

4.4.2 TTY_LDISC

Struktura tty_ldisc zawiera wskaźniki do funkcji udostępnianych przez dyscyplinę linii takie jak:

- zamykanie i otwieranie dyscypliny linii
- ustawianie parametrów pracy dyscypliny linii,
- pisanie do i czytanie z dyscypliny linii
- zwracanie liczby znaków w buforze
- opróżnianie bufora po wypisaniu znaków na ekran

Zawiera także wskaźniki do funkcji wykorzystywanych przez sterownik terminala takie jak:

- przyjmowanie bufora od sterownika
- zwracanie sterownikowi ilości wolnego miejsca w buforze

4.4.3 TTY_DRIVER

Struktura tty_driver jest strukturą opisującą sterownik urządzenia terminalowego. Pola tej struktury opisują między innymi:

- nazwę oraz numer główny urządzenia
- najmniejszy numer drugorzędny urządzenia
- typ sterownika , może to być: sterownik konsoli, łącza szeregowego, pseudoterminala lub urządzeń SCC
- podtyp sterownika: sterownik pseudoterminala nadrzędnago i sterownik pseudoterminala podrzednego
- początkowe ustawienia pracy terminala czyli strukturę termios
- flagi
- tablicę wskaźników do struktur tty_struct opisujących aktualnie otwarte urządzenia terminalowe
- definicje funkcji otwierania, zamykania urządzenia
- definicje funkcji pisania, czytania z i do urządzenia

4.4.4 TTY_STRUCT

Podstawową strukturą opisującą urządzenie terminalowe jest struktura tty_struct, jej definicja znajduje się w pliku include/linux/tty.h. Struktura ta tworzona jest dynamicznie wtedy gdy jest otwierane jakieś nowe urządzenie terminalowe. Struktura ta daje dostęp do wszystkich informacji potrzebnych podczas pracy z urządzeniem terminalowym. Zanim bliżej poznamy opis niektórych pól tej struktury potrzebna nam będzie definicja terminala sterującego oraz grupy terminalowej. Jak wiadomo każdy proces może być członkiem grupy terminali, identyfikowanej poprzez dodatnią liczbę całkowitą, zwaną identyfikatorem grupy terminali. Identyfikator grupy terminali jest identyfikatorem przywódcy grupy procesów, który otworzył terminal (zazwyczaj jest to shell). Identyfikator grupy terminali identyfikuje terminal sterujący grupy procesów. Terminal sterujący służy między innymi do wysyłania sygnałów, które są odbierane po naciśnięciu określonych klawiszy na terminalu. Terminal sterujący jest to ten terminal który jest wskazywany przez pole tty w task_struct. Natomiast zbiór procesów mających ten sam terminal sterujący nazywamy grupą terminalową. Wszystkie procesy danej grupy terminalowej dostają sygnał **SIGHUP** w przypadku odłączenia terminala.

Podstawowe pola struktury tty_struct zaiwrają między innymi:

- sterownik urządzenia terminalowego (struct tty_driver)
- dyscyplinę linii (struct tty_ldisc)
- stałe parametry pracy (struct termiops)
- specjalne pole o nazwie termios_locked które zezwala bądź nie na wprowadzanie zmian w strukturze termios, dokładniej, jeśli chcemy za pomocą funkcji ioctl zmienić coś w strukturze termios to odpowiedni bit w strukturze wskazywanej przez termios_locked musi być wyzerowany
- identyfikator grupy terminalowej, czyli grupy procesów mających ten sam terminal sterujący
- identyfikator sesji
- numer urządzenia
- flagi
- liczba otwartych plików specjalnych
- bufor sterownika urządzenia terminalowego
- kolejka procesów czekających na pisanie do urządzenia terminalowego
- kolejka procesów czekających na czytanie z urządzenia terminalowego

Cześć pól struktury tty_struct jest wykorzystywana przez dyscyplinę linii n_tty, pola te zawierają następujące informacje:

- w której kolumnie będziemy wypisywać bieżący znak na terminalu

- czy jesteśmy w trybie ścierania
- czy jesteśmy w trybie kanonicznym
- czy jesteśmy w trakcie zamykania dyscypliny linii
- bufor dyscypliny linii
- ilość znaków w buforze
- pozycja w buforze gdzie znajduje się początek i koniec danych

4.5 Algorytmy

— tty_open
{
wywołaj funkcję init_dev(...);
wykonaj procedurę otwarcia opisaną w driver.open,
natomiast jeżeli funkcja ta
nie jest zdefiniowana to zwróć błąd **ENODEV**
}

— tty_read(...)
{
Sprawdź czy nie zaszły żadne niepożądane sytuacje
Wywołaj funkcje tty.ldisc.read
i-node.i-atime = currenttime
}

— tty_write(...)
{
Sprawdź czy nie zaszły żadne niepożądane sytuacje
Wywołaj funkcje tty.ldisc.write
i-node.i-atime = currenttime
}

—init_dev(...)
//funkcja ta alokuje pamięć i inicjuje struktury
opisujące urządzenie terminalowe//
{
znajdź strukturę tty_driver opisująca ten terminal
ze znalezionej struktury pobierz wskaźniki do struktur:
tty, termios, termios_locked
dla każdej ze znalezionych struktur
jeżeli wskaźnik do niej był **Null** — zaalokuj pamięć
jeśli otwierane urządzenie było pseudoterminalem

to zaalokuj też pamięć dla drugiego członka pary
}

— do_tty_hangup(...)
//zadaniem tej funkcji jest poinformowanie procesów podłączonych
do danego terminala o tym że nastąpiło zarwanie komunikacji //
{
Przeglądaj wszystkie i-węzly plików specjalnych i wybierz te
które są związane ze zrywanym właśnie terminalem
dla znalezionych i-węzlów wykonaj tty_fasunc(...)
opróżnij dyscyplinę linii
opróżnij bufor sterownika
zamknij aktualną dyscyplinę linii i zainicjuj **N-TTY**
dla wszystkich procesów związanych z danym terminalem
wyślij sygnał: SIGHUP, SIGCNT
wywołaj funkcje zerwania dla sterownika
}

4.6 Dyscyplina linii

Jak już wcześniej było powiedziane dyscyplina linii odpowiedzialna jest za wstępne przetwarzanie danych wejściowych, zapewnia też standardowy sposób obsługi niektórych kombinacji klawiszy itp.

W Linuxie mamy zdefiniowanych pięć dyscyplin linii. Trzy z nich (**N_SLIP**, **N_PPP**, **N_STRIP**) są to implementacje protokołów sieciowych, istnieje jeszcze jedna dyscyplina która z naszego punktu widzenie nie jest istotna jest nią dyscyplina (**N_MOUSE**) używana na komputerach SUN. Najbardziej istotna dla nas i taka z którą się spotykamy jest dyscyplin **N_TTY**.

Algorytmy (dla dyscypliny **N_TTY**):

write_chan(...)
{
Sprawdź czy terminal nie jest zastopowany
Jeśli tak to zakończ
Jeśli nie to
{
wstaw się na koniec kolejki procesów czekających na zapis do
urządzenia i wykonaj w pętli
{
Sprawdź czy połączenie nie zostało przerwane
Jeśli flaga **O_POST** = 1
to pobieraj w pętli pojedyncze znaki,
przetwarzaj je aż uzbiera się pełny bufor sterownika,
Wywołaj funkcje flush_chars() o ile istnieje
Jeśli flaga **O_POST** nie była ustawiona
to wywołaj driver.write()
Wybierz proces z kolejki czekających na zapis do urządzenia
}

przenieś się z kolejki procesów czekających na zapis do kolejki procesów gotowych

}

}

```
read_chan()
```

{

Sprawdź ile jest znaków do przeczytania
Jeśli w buforze jest za mało to wstaw się do kolejki
procesów czekających na odczyt
i wybierz nowy proces z tej kolejki.
Sprawdź czy połączenie nie zostało zerwane
Skopiuj wszystkie znaki do przestrzeni użytkownika
Zmniejsz wartość opisującą ile chcemy wczytać znaków

}

```
n_tty_receive_buf()
```

{

//funkcja ta jest wywoływana przez sterownik
odpowiedzialny za odbiór znaków//
Jeśli terminal jest w trybie surowym (bez przetwarzania) to
po prostu przekopiuj znaki z bufora do bufora użytkownika
Jeśli terminal działa w trybie kanonicznym to wywołaj funkcję
zależną od flagi znaku

}

Oczywiście jest też możliwość korzystania z terminala tak aby nie przetwarzać wstępnie wprowadzanych znaków, służy do tego specjalny tryb pracy dyscypliny linii tzw. tryb surowy. Dzięki funkcji ioctl możemy też wpływać na sposób przetwarzania przez dyscyplinę linii, możemy decydować czy echo znaków ma być wyświetlane na ekran (może to być przydatne na przykład przy wprowadzaniu hasła kiedy chcemy aby znaki wpisywane z klawiatury nie pojawiły się na ekranie).

4.7 Interfejs sterownika

W Linuxie istnieje kilka rodzajów urządzeń terminalowych. Linux zapewnia obsługę 63 konsol, 64 terminali podłączonych do portów szeregowych oraz 64 pseudoterminali nadrzędnych i 64 pseudoterminali podrzędnych. Poszczególne urządzenia terminalowe różnią się sterownikiem. W Linuxie mamy zdefiniowanych pięć sterowników urządzeń terminalowych, są to:

- TTY_DRIVER_TYPE_SYSTEM
- TTY_DRIVER_TYPE_CONSOLE
- TTY_DRIVER_TYPE_SERIAL
- TTY_DRIVER_TYPE_PTY

  - PTY_TYPE_MASTER
  - PTY_SLAVE

- TTY_DRIVER_TYPE_SCC

W systemie używa się tylko trzech z nich : sterowników konsoli, pseudoterminali i łącza szeregowgo.

4.8 Konsola

Konsola jest jak już wiadomo jednym z czterech urządzeń terminalowych. Charakterystyczną cechą dla konsoli jest to, że jest ona blisko związana ze sprzętem, dlatego konsola musi wykonać wiele sprawdzeń przy każdym naciśnietym klawiszu, czy jest to klawisz specjalny, funkcyjny itp. Poza tym aby możliwe było przełączanie się pomiędzy wirtualnymi konsolami klawisze odpowiedzialne za takie przełączenie musza być niewidoczne dla reszty systemu. Aby osiągnąć takie możliwości zdefiniowane zostały w Linuxie bardzo regularne kody klawiszy. W kodach tych najstarsze bity kodu klawisza określają jeden z 16 możliwych typów. Następnie gdy znany jest typ klawisza sięga się do specjalnej 16 elementowej tablicy wskaźników do funkcji z których każda jest odpowiedzialna za inny typ klawisza. Istnieje też druga tablica odwzorowań kodów klawisza w kody dwubajtowe co pozwala potem na rożne interpretacje rożnych klawiszy jeśli podmienimy tą tablice. Na przykład do przełączania sesji służą klawisze **Alt+ Fn** natomiast pod systemem X-Windows mamy inną tablice odwzorowań dzięki czemu taka kombinacja klawiszy ma inne zastosowanie.

4.9 Pseudoterminal

Pseudoterminale są specjalnymi urządzeniami zachowującymi się trochę jak łącza, które jednak charakteryzują się tym że z jednej strony są rozpoznawane jak terminale tzn. mają dyscyplinę linii, mogą wysyłać sygnały a z drugiej pracują w trybie surowym . Coś takiego dają nam właśnie pseudoterminale. Pseudoterminal składa się z dwu części nadrzędnej i podrzędnej, części te połączone są przez pole link w tty_struct. Obie te części mają wspólny bufor tymczasowy i mogą się komunikować tzn. pisząc coś do części nadrzędnej jesteśmy w stanie odczytać to w części podrzędnej i odwrotnie. To że pseudoterminal jest zawsze złożony z dwu części zapewnia funkcja otwierania pseudoterminala, znaczy to, że jeżeli otwieramy pseudoterminal nadrzędny to pseudoterminal podrzędny jest tworzony automatycznie, natomiast przy otwieraniu pseudoterminala podrzędnego czekamy aż ktoś otworzy pseudoterminal nadrzędny.

Bibliografia

[1] Pliki źródłowe Linuxa 2.0.32

[2] projekt Linux

[3] projekt LabLinux

Spis rzeczy