helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Linux. Mechanizmy sieciowe



Autor: Christian Benvenuti
Tłumaczenie: Jaromir Senczyk, Grzegorz Werner
ISBN: 83-246-0462-6
Tytuł oryginału: Understanding Linux Network Internals
Format: B5, stron: 1000

Kompletny przewodnik po mechanizmach sieciowych Linuksa

- Inicjalizacja urządzeń sieciowych.
- Interfejsy pomiędzy urządzeniami i protokołami.
- Rozwiązania specyficzne dla protokołów.

Sieci, a szczególnie internet, to jeden z filarów współczesnej informatyki. Niemal każdy elektroniczny gadżet może pracować w sieci za pośrednictwem różnych metod komunikacji. Ogromna ilość produkowanych dziś urządzeń sieciowych opiera się na różnych dystrybucjach systemu operacyjnego Linux. Ten dostępny nieodpłatnie system operacyjny od początku tworzony był z uwzględnieniem roli, jaką mógłby odgrywać w świecie sieci komputerowych, więc zaimplementowano w nim niemal wszystkie możliwe mechanizmy sieciowe. Dodatkowo filozofia, jaką przyjęto przy rozwoju tego systemu operacyjnego, pozwala wszystkim jego użytkownikom na dodawanie do jądra Linuksa własnych modułów zapewniających obsługę niestandardowych urządzeń i protokołów.

Książka „Linux. Mechanizmy sieciowe" to szczegółowe omówienie rozwiązań sieciowych, jakie zostały zastosowane w tym systemie operacyjnym. Opisuje sposoby, w jakie jądro Linuksa realizuje zadania przydzielane mu przez protokoły IP. Czytając ją, można poznać współczesną łączność sieciową na wziętych z życia przykładach. Pozycja ta jest doskonałym przewodnikiem po kodzie źródłowym funkcji sieciowych jądra systemu Linux. Przedstawia kod w języku C z obszernymi komentarzami i wyjaśnieniami zastosowanych mechanizmów.

- Struktury danych
- Rejestracja i inicjalizowanie urządzeń sieciowych
- Powiadamianie jądra o odbiorze ramki
- Obsługa protokołów
- Implementacja mostkowania
- Obsługa IPv4
- Podsystem sąsiedztwa
- Routing

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Wstęp **13**

Część I Podstawy 21

1. Wprowadzenie **23**
Podstawowa terminologia 23
Typowe wzorce kodowania 24
Narzędzia dostępne w przestrzeni użytkownika 36
Przeglądanie kodu źródłowego 37
Opcje oferowane w postaci łat 38

2. Najważniejsze struktury danych **41**
Bufor gniazda: struktura sk_buff 41
Struktura net_device 60
Pliki występujące w tym rozdziale 73

3. Interfejs użytkownik – jądro **75**
Informacje ogólne 75
procfs kontra sysctl 77
Interfejs ioctl 84
Netlink 86
Serializacja zmian konfiguracji 87

Część II Inicjalizacja systemu 89

4. Łańcuchy powiadomień **91**
Przyczyny wprowadzenia łańcuchów powiadomień 91
Informacje ogólne 93
Definiowanie łańcucha 93
Rejestracja w łańcuchu 94

Powiadamianie o zdarzeniach 95
Łańcuchy powiadomień w podsystemach sieciowych 96
Strojenie za pośrednictwem systemu plików /proc 97
Funkcje i zmienne występujące w tym rozdziale 97
Pliki i katalogi występujące w tym rozdziale 98

5. Inicjalizacja urządzeń sieciowych 99
Ogólne informacje na temat inicjalizacji systemu 99
Rejestracja i inicjalizacja urządzeń 101
Podstawowe cele inicjalizacji kart sieciowych 101
Interakcje pomiędzy urządzeniami i jądrem 102
Opcje inicjalizacji 107
Opcje modułów 108
Inicjalizacja warstwy obsługi urządzeń: net_dev_init 109
Kod pomocniczy w przestrzeni użytkownika 111
Urządzenia wirtualne 114
Strojenie za pośrednictwem systemu plików /proc 117
Funkcje i zmienne występujące w tym rozdziale 118
Pliki i katalogi występujące w tym rozdziale 118

6. Warstwa PCI i karty sieciowe 119
Struktury danych występujące w tym rozdziale 119
Rejestracja sterownika karty sieciowej PCI 121
Zarządzanie zasilaniem i Wake-on-LAN 122
Przykład rejestracji sterownika karty sieciowej PCI 123
Ogólny schemat 125
Strojenie za pośrednictwem systemu plików /proc 125
Funkcje i zmienne występujące w tym rozdziale 127
Pliki i katalogi występujące w tym rozdziale 127

7. Infrastruktura jądra związana z inicjacją komponentów 129
Opcje uruchamiania jądra 129
Kod inicjacji modułu 135
Optymalizacja etykiet opartych na makrach 138
Procedury inicjacji wykonywane podczas uruchamiania systemu 140
Optymalizacja pamięci 142
Strojenie za pośrednictwem systemu plików /proc 146
Funkcje i zmienne występujące w tym rozdziale 146
Pliki i katalogi występujące w tym rozdziale 147

8. Rejestracja i inicjacja urządzeń 149
Kiedy urządzenie zostaje zarejestrowane 150
Kiedy urządzenie zostaje wyrejestrowane 151
Przydział struktur net_device 151
Szkielet zarejestrowania i wyrejestrowania karty sieciowej 152
Inicjacja urządzenia 154
Organizacja struktur net_device 158
Stan urządzenia 160
Rejestrowanie i wyrejestrowywanie urządzeń 162
Rejestracja urządzenia 166
Wyrejestrowanie urządzenia 168
Włączanie i wyłączanie urządzenia sieciowego 172
Aktualizacja stanu reguły kolejkowania 173
Konfigurowanie urządzeń z przestrzeni użytkownika 177
Urządzenia wirtualne 180
Blokowanie 182
Strojenie za pośrednictwem systemu plików /proc 183
Funkcje i zmienne występujące w tym rozdziale 183
Pliki i katalogi występujące w tym rozdziale 184

Część III Wysyłanie i odbieranie 185

9. Przerwania i sterowniki sieciowe 187
Decyzje i kierunki ruchu 187
Powiadamianie sterownika o odebraniu ramki 189
Procedury obsługi przerwań 192
Struktura danych softnet_data 213

10. Odbiór ramki 217
Interakcje z innymi opcjami 218
Włączanie i wyłączanie urządzenia 218
Kolejki 219
Powiadamianie jądra o odbiorze ramki: NAPI i netif_rx 219
Stary interfejs pomiędzy sterownikami urządzeń i jądrem: pierwsza część netif_rx 225
Zarządzanie obciążeniem 231
Obsługa przerwania NET_RX_SOFTIRQ: net_rx_action 234

11. Wysyłanie ramki 245
Włączanie i wyłączanie wysyłania 247

12. Informacje o przerwaniach 265
Dane statystyczne 265
Strojenie za pośrednictwem systemów plików /proc i sysfs 266
Funkcje i zmienne występujące w tej części książki 267
Pliki i katalogi występujące w tej części książki 268

13. Procedury obsługi protokołów 271
Przegląd stosu protokołowego 271
Wykonanie odpowiedniej procedury obsługi protokołu 279
Organizacja procedur obsługi protokołów 283
Rejestracja procedury obsługi protokołu 284
Ethernet i ramki IEEE 802.3 286
Strojenie za pośrednictwem systemu plików /proc 296
Funkcje i zmienne występujące w tym rozdziale 297
Pliki i katalogi występujące w tym rozdziale 297

Część IV Mostkowanie 299

14. Mostkowanie: podstawowe koncepcje 301
Wtórniki, mosty i routery 301
Mosty i przełączniki 303
Hosty 304
Łączenie sieci lokalnych za pomocą mostów 304
Mostkowanie różnych technologii sieci lokalnych 305
Uczenie się adresów 306
Sieci z wieloma mostami 308

15. Mostkowanie: protokół drzewa częściowego 315
Podstawowa terminologia 316
Przykład hierarchicznej topologii L2 zawierającej mosty 316
Podstawowe elementy protokołu Spanning Tree Protocol 320
Identyfikatory portów i mostów 325
Ramki BPDU 327
Definiowanie aktywnej topologii 332
Liczniki czasu 339
Zmiany topologii 344
Kapsułkowanie ramek BPDU 348
Wysyłanie konfiguracyjnych ramek BPDU 348
Przetwarzanie ramek wejściowych 351
Czas konwergencji 353
Przegląd nowych wersji protokołu STP 354

16. Mostkowanie: implementacja w Linuksie 359
Abstrakcja urządzenia mostkującego 359
Ważne struktury danych 362
Inicjalizacja kodu mostkowania 364
Tworzenie urządzeń i portów mostkujących 365
Tworzenie nowego urządzenia mostkującego 365
Procedura inicjalizacyjna urządzenia mostkującego 366
Usuwanie mostu 367
Dodawanie portów do mostu 367
Usuwanie portu mostu 370
Włączanie i wyłączanie urządzenia mostkującego 370
Włączanie i wyłączanie portu mostu 371
Zmiana stanu portu 373
Panorama 373
Baza przekazywania 375
Obsługa ruchu wejściowego 378
Wysyłanie danych z urządzenia mostkującego 382
Spanning Tree Protocol (STP) 383
Łańcuch powiadomień netdevice 390

17. Mostkowanie: zagadnienia różne 393
Narzędzia konfiguracyjne działające w przestrzeni użytkownika 393
Dostrajanie za pomocą systemu plików /proc 398
Dostrajanie za pomocą systemu plików /sys 398
Statystyka 399
Struktury danych przedstawione w tej części książki 400
Funkcje i zmienne przedstawione w tej części książki 404
Pliki i katalogi przedstawione w tej części książki 405

Część V Internet Protocol Version 4 (IPv4) 407

18. Internet Protocol Version 4 (IPv4): pojęcia 409
Protokół IP: panorama 409
Nagłówek IP 411
Opcje IP 414
Fragmentacja i defragmentacja pakietów 420
Sumy kontrolne 430

19. Internet Protocol Version 4 (IPv4): funkcje i cechy jądra Linuksa 437
Główne struktury danych IPv4 437
Ogólna obsługa pakietów 441
Opcje IP 450

20. Internet Protocol Version 4 (IPv4): przekazywanie i lokalne dostarczanie 461
Przekazywanie 461
Dostarczanie lokalne 466

21. Internet Protocol Version 4 (IPv4): transmisja 469
Kluczowe funkcje transmisyjne 470
Interfejs do podsystemu sąsiedztwa 504

22. Internet Protocol Version 4 (IPv4): obsługa fragmentacji 505
Fragmentacja IP 506
Defragmentacja IP 514

23. Internet Protocol Version 4 (IPv4): zagadnienia różne 527
Długo przechowywane informacje o partnerze IP 527
Wybór wartości identyfikatora w nagłówku IP 531
Statystyka IP 532
Konfiguracja IP 535
IP-over-IP 540
Protokół IPv4: co z nim jest nie tak? 541
Dostrajanie za pomocą systemu plików /proc 542
Struktury danych opisywane w tej części książki 545
Funkcje i zmienne wspomniane w tej części książki 554
Pliki i katalogi wspomniane w tej części książki 556

24. Protokół warstwy czwartej i obsługa Raw IP 557
Dostępne protokoły L4 557
Rejestracja protokołu L4 558
Dostarczanie danych L3 do L4: ip_local_deliver_finish 562
IPv4 a IPv6 569
Dostrajanie za pomocą systemu plików /proc 569
Funkcje i zmienne przedstawione w tym rozdziale 570
Pliki i katalogi przedstawione w tym rozdziale 570

25. Internet Control Message Protocol (ICMPv4) 571
Nagłówek ICMP 572
Treść ICMP 573
Typy komunikatów ICMP 574
Zastosowania protokołu ICMP 580
Panorama 583
Inicjalizacja protokołu 584
Struktury danych opisywane w tym rozdziale 585
Wysyłanie komunikatów ICMP 587

Odbieranie komunikatów ICMP 594
Statystyka ICMP 601
Przekazywanie powiadomień o błędach do warstwy transportu 603
Dostrajanie za pomocą systemu plików /proc 604
Funkcje i zmienne przedstawione w tym rozdziale 605
Pliki i katalogi przedstawione w tym rozdziale 605

Część VI Podsystem sąsiedztwa 607

26. Podsystem sąsiedztwa: pojęcia 609
Co to jest sąsiad? 609
Do czego potrzebne są protokoły sąsiedztwa? 612
Implementacja w Linuksie 617
Pośredniczenie w protokole sąsiedztwa 619
Wysyłanie i przetwarzanie żądań odwzorowania adresu 622
Stany sąsiadów i wykrywanie nieosiągalności sieci 625

27. Podsystem sąsiedztwa: infrastruktura 633
Główne struktury danych 633
Wspólny interfejs między protokołami L3 a protokołami sąsiedztwa 636
Ogólne zadania infrastruktury sąsiedztwa 646
Liczniki referencji do struktur neighbour 650
Tworzenie wpisu sąsiada 651
Usuwanie sąsiada 653
Działanie w charakterze pośrednika 658
Buforowanie nagłówków L2 662
Inicjalizacja i finalizacja protokołu 666
Interakcja z innymi podsystemami 667
Interakcja między protokołami sąsiedztwa a funkcjami transmisyjnymi L3 670
Kolejkowanie 671

28. Podsystem sąsiedztwa: Address Resolution Protocol (ARP) 677
Format pakietu ARP 678
Przykład transakcji ARP 680
Spontaniczny ARP 681
Odpowiedzi z wielu interfejsów 683
Konfigurowalne opcje ARP 685
Inicjalizacja protokołu ARP 691
Inicjalizacja struktury neighbour 693
Wysyłanie i odbieranie pakietów ARP 698
Przetwarzanie wejściowych pakietów ARP 703
Pośredniczenie ARP 710

Przykłady 715
Zdarzenia zewnętrzne 717
ARPD 719
Reverse Address Resolution Protocol (RARP) 722
Ulepszenia w ND (IPv6) w stosunku do ARP (IPv4) 722

29. Podsystem sąsiedztwa: zagadnienia różne 723
Zarządzanie sąsiadami przez administratora systemu 723
Dostrajanie za pomocą systemu plików /proc 726
Struktury danych przedstawione w tej części książki 731
Funkcje i zmienne przedstawione w tej części książki 744
Pliki i katalogi przedstawione w tej części książki 745

Część VII Routing 747

30. Routing: pojęcia 749
Routery, trasy i tablice tras 750
Podstawowe elementy routingu 754
Tablica tras 764
Wyszukiwania 768
Odbieranie pakietów a wysyłanie pakietów 770

31. Routing: zagadnienia zaawansowane 773
Zasady routingu opartego na polityce 773
Zasady routingu wielościeżkowego 778
Interakcje z innymi podsystemami jądra 784
Demony protokołów routingu 789
Szczegółowe monitorowanie 791
Komunikaty ICMP_REDIRECT 791
Filtrowanie ścieżek odwrotnych 796

32. Routing: implementacja w Linuksie 799
Opcje jądra 799
Główne struktury danych 802
Zasięgi tras i adresów 806
Podstawowe i wtórne adresy IP 808
Uniwersalne procedury pomocnicze i makra 809
Globalne blokady 811
Inicjalizacja podsystemu routingu 811
Zdarzenia zewnętrzne 813
Interakcje z innymi podsystemami 824

33. Routing: bufor tras 827
Inicjalizacja bufora tras 827
Organizacja tablicy tras 828
Podstawowe operacje na buforze 829
Buforowanie wielościeżkowe 838
Interfejs między DST a wywołującymi protokołami 843
Opróżnianie bufora tras 849
Odśmiecanie 850
Ograniczanie częstotliwości wyjściowych komunikatów ICMP 860

34. Routing: tablice tras 861
Organizacja tablic mieszających w podsystemie routingu 861
Inicjalizacja tablicy tras 867
Dodawanie i usuwanie tras 868
Routing oparty na polityce i jego wpływ na definicje tablic tras 873

35. Routing: wyszukiwania 875
Panorama funkcji wyszukiwawczych 875
Procedury pomocnicze 876
Przeszukiwanie tablicy: fn_hash_lookup 877
Funkcja fib_lookup 882
Ustawianie funkcji odbiorczych i transmisyjnych 882
Ogólna struktura procedur routingu wejściowego i wyjściowego 885
Routing wejściowy 887
Routing wyjściowy 895
Wpływ routingu wielościeżkowego na wybór następnego przeskoku 902
Routing oparty na polityce 905
Routing źródłowy 908
Routing oparty na polityce i klasyfikator oparty na tablicy tras 909

36. Routing: zagadnienia różne 913
Narzędzia konfiguracyjne działające w przestrzeni użytkownika 913
Statystyka 919
Dostrajanie za pomocą systemu plików /proc 919
Włączanie i wyłączanie przekazywania 926
Struktury danych przedstawione w tej części książki 928
Funkcje i zmienne przedstawione w tej części książki 944
Pliki i katalogi przedstawione w tej części książki 946

Skorowidz 949



ROZDZIAŁ 5.

Inicjalizacja urządzeń sieciowych

Elastyczność współczesnych systemów operacyjnych komplikuje proces inicjalizacji. Sterownik urządzenia może zostać załadowany jako moduł lub statyczny komponent jądra. Co więcej, urządzenia mogą być obecne podczas uruchamiania systemu albo podłączane (i odłączane) w czasie jego pracy. Do tych ostatnich należą między innymi urządzenia USB, PCI CardBus, IEEE 1394 (przez Apple zwane również FireWire). W tym rozdziale pokażę, w jaki sposób możliwość podłączania urządzeń w czasie pracy systemu wpływa na działanie kodu jądra i przestrzeni użytkownika.

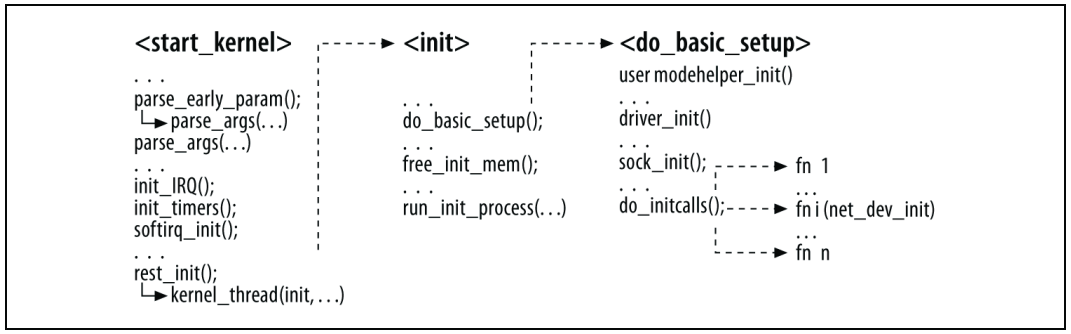
W rozdziale omawiam:

- fragmentu kodu sieciowego odpowiedzialnego za jego inicjalizację;
- inicjalizację karty sieciowej;
- sposób wykorzystania przerwań przez karty sieciowe, sposób przydzielania i zwalniania procedur obsługi przerwań, a także możliwość współdzielenia przerwań przez sterowniki urządzeń;
- sposób określania przez użytkownika parametrów konfiguracyjnych sterowników urządzeń ładowanych jako moduły;
- interakcję pomiędzy przestrzenią użytkownika i jądrem podczas inicjalizacji i konfiguracji urządzeń. Pokażę, w jaki sposób jądro może użyć pomocniczego kodu działającego w przestrzeni użytkownika w celu załadowania właściwego sterownika karty sieciowej lub zastosowania konfiguracji pochodzącej z przestrzeni użytkownika. W szczególności zajmę się też opcją Hotplug;
- różnice pomiędzy wirtualnymi i rzeczywistymi urządzeniami w odniesieniu do ich konfiguracji i interakcji z jądrem.

Ogólne informacje na temat inicjalizacji systemu

Ważne jest, aby wiedzieć dokładnie, gdzie i kiedy następuje inicjalizacja głównych podsystemów związanych z działaniem sieci, włączając w to sterowniki urządzeń. Ponieważ jednak w zakresie tematycznym tej książki leży jedynie sieciowy aspekt inicjalizacji, to nie będziemy się zajmować ogólnym przypadkiem inicjalizacji sterowników urządzeń czy ogólnymi usługami jądra (np. zarządzaniem pamięcią). Zagadnieniom tym poświęcone są książki *Linux Device Drivers* i *Understanding the Linux Kernel*, obie wydane przez O'Reilly.

Rysunek 5.1 przedstawia w skrócie, gdzie i w jakiej kolejności zostają zainicjowane niektóre z podsystemów jądra podczas uruchamiania systemu (*init/main.c*).



Rysunek 5.1. Inicjalizacja jądra

Podczas uruchamiania jądro wywołuje funkcję `start_kernel` inicjującą wiele podsystemów, z których część została przedstawiona na rysunku 5.1. Zanim funkcja `start_kernel` zakończy swoje działanie, wywołuje najpierw wątek `init` jądra, który zajmuje się resztą inicjalizacji. Większość akcji związanych z inicjalizacją omawianą w tym rozdziale wykonywanych jest przez funkcję `do_basic_setup`.

Spośród wielu różnych zadań związanych z inicjalizacją najbardziej będą nas interesować następujące trzy:

Opcje uruchamiania
: Dwa wywołania funkcji `parse_args`, jedno bezpośrednie, a drugie pośrednie przez funkcję `parse_early_param`, obsługują parametry konfiguracyjne przekazane jądru podczas uruchamiania przez procedurę startu LILO lub GRUB. Sposób tej obsługi przedstawiono w podrozdziale „Opcje uruchamiania jądra".

Przerwania i liczniki czasu
: Przerwania sprzętowe i programowe są inicjowane za pomocą funkcji odpowiednio: `init_IRQ` i `softirq_init`. Przerwania zostaną omówione w rozdziale 9. W tym rozdziale pokażę, w jaki sposób sterownik urządzenia rejestruje procedurę obsługi przerwania i w jaki sposób procedury takie są zorganizowane w pamięci. Liczniki czasu zostają zainicjowane we wczesnej fazie uruchamiania systemu, aby mogły zostać użyte przez inne zadania.

Procedury inicjalizacji
: Podsystemy jądra oraz wbudowane sterowniki urządzeń są inicjowane przez `do_initcalls`. Funkcja `free_init_mem` zwalnia fragment pamięci, który zawiera niepotrzebny kod. Optymalizacja taka jest możliwa dzięki zastosowaniu inteligentnych etykiet procedur. Więcej informacji na ten temat w rozdziale 7.

Procedura `run_init_process` określa pierwszy proces wykonywany w systemie, będący procesem nadrzędnym wszystkich innych procesów. Proces ten otrzymuje identyfikator PID równy 1 i działa tak długo jak system. Zwykle wykonuje on program *init* będący częścią pakietu SysVinit. Administrator może jednak podać inny program, używając opcji startowej `init=`. Jeśli opcja taka nie zostanie podana, to jądro próbuje wykonać polecenie *init*, korzystając ze zbioru znanych lokalizacji tego programu. Użytkownik może również podać opcje przekazywane programowi *init* podczas uruchamiania systemu (podrozdział „Opcje uruchamiania jądra").

Rejestracja i inicjalizacja urządzeń

Aby urządzenie sieciowe mogło być używane, musi najpierw być rozpoznane przez jądro i związane z odpowiednim sterownikiem. Sterownik ten przechowuje w swoich prywatnych strukturach wszystkie informacje potrzebne do sterowania urządzeniem oraz do interakcji z innymi komponentami jądra, które żądają urządzenia. Zadania rejestracji i inicjalizacji wykonywane są częściowo przez podstawową część jądra i częściowo przez sterownik urządzenia. A oto kolejne fazy inicjalizacji:

Inicjalizacja sprzętowa
: Wykonywana jest przez sterownik urządzenia we współpracy z warstwą magistrali (PCI lub USB). Sterownik, czasami samodzielnie, a innym razem za pomocą parametrów dostarczonych przez użytkownika, konfiguruje przerwanie oraz adres wejścia i wyjścia pozwalające na komunikację z jądrem. Ponieważ ta część inicjalizacji jest bardziej związana z samym sterownikiem urządzenia niż warstwami protokołowymi, nie będziemy poświęcać jej zbyt wiele uwagi. Ograniczę się do przedstawienia jednego przykładu dla warstwy PCI.

Inicjalizacja programowa
: Zanim urządzenia będzie można użyć, konfiguracja protokołów może wymagać od użytkownika dostarczenia dodatkowych parametrów konfiguracyjnych, takich jak na przykład adres IP. Zadanie to zostanie omówione w innych rozdziałach.

Inicjalizacja opcji sieciowych
: Jądro systemu Linux dostarczane jest z wieloma różnymi opcjami sieciowymi. Ponieważ niektóre z tych opcji wymagają konfiguracji dla poszczególnych urządzeń, to ich istnienie musi zostać uwzględnione podczas inicjalizacji urządzenia. Przykładem może być opcja Traffic Control będąca podsystemem implementującym usługę QoS (*Quality of Service*) decydującą o sposobie umieszczania i usuwania pakietów w kolejce wyjściowej urządzenia (i z pewnymi ograniczeniami podobnie dla kolejki wejściowej).

W rozdziale 2. pokazano już, że struktura danych `net_device` zawiera zbiór wskaźników funkcji używanych przez jądro podczas interakcji ze sterownikiem urządzenia i specjalnymi opcjami jądra. Inicjalizacja tych wskaźników zależy częściowo od typu urządzenia (np. Ethernet) i częściowo od jego modelu. Ze względu na popularność sieci Ethernet w tym rozdziale skoncentrujemy się na inicjalizacji urządzeń Ethernet (inne urządzenia są obsługiwane w bardzo podobny sposób).

W rozdziale 8. zajmiemy się szczegółowo sposobem rejestrowania urządzeń w kodzie sieciowym przez sterowniki urządzeń.

Podstawowe cele inicjalizacji kart sieciowych

Każde urządzenie sieciowe jest reprezentowane w jądrze systemu Linux przez instancję struktury danych `net_device`. W rozdziale 8. pokazano, w jaki sposób struktury te są przydzielane i w jaki sposób są inicjowane ich pola, częściowo przez sterownik urządzenia i częściowo przez podstawowe procedury jądra. W tym rozdziale skoncentrujemy się na sposobie, w jaki sterowniki urządzeń przydzielają zasoby potrzebne do komunikacji pomiędzy jądrem i urządzeniem:

Linii IRQ
: W podrozdziale „Interakcja pomiędzy urządzeniami i jądrem" pokażę, że karty sieciowe muszą otrzymać odpowiednie przerwanie IRQ, którego używają do wywoływania jądra. Nie jest to natomiast potrzebne w przypadku urządzeń wirtualnych: przykładem może być urządzenie pseudosieci, którego działanie odbywa się całkowicie wewnątrz jądra (podrozdział „Urządzenia wirtualne" w dalszej części tego rozdziału).

  Dwie funkcje używane do przydziału i zwalniania linii IRQ zostaną omówione w podrozdziale „Przerwania sprzętowe" zamieszczonym w dalszej części tego rozdziału. W podrozdziale „Strojenie za pośrednictwem systemu plików /proc" omówiony zostanie plik */proc/interrupts* pozwalający sprawdzić aktualny przydział przerwań.

Porty I/O i rejestracja pamięci
: Często sterownik tworzy odwzorowanie obszaru pamięci urządzenia (na przykład jego rejestrów konfiguracyjnych) w pamięci systemu, dzięki czemu operacje odczytu i zapisu przez sterownik mogą odbywać się bezpośrednio przy użyciu adresów pamięci systemowej, co pozwala uprościć kod. Porty I/O i pamięć są przydzielane i zwalniane za pomocą funkcji `request_region` i `release_region`.

Interakcje pomiędzy urządzeniami i jądrem

Interakcje prawie wszystkich urządzeń (włączając w to karty sieciowe) z jądrem odbywają się na dwa sposoby:

Odpytywanie
: Wykonywane po stronie jądra. Jądro sprawdza status urządzenia w regularnych odstępach czasu.

Przerwania
: Wykonywane po stronie urządzenia. Urządzenie wysyła sprzętowy sygnał (generując przerwanie), gdy chce zwrócić uwagę jądra.

W rozdziale 9. można znaleźć szczegółowe omówienie alternatywnych rozwiązań sterowników kart sieciowych oraz przerwań programowych. Pokażę również, w jaki sposób system Linux może używać kombinacji odpytywania i przerwań w celu zwiększenia efektywności. W tym rozdziale natomiast zajmiemy się tylko przypadkiem bazującym wyłącznie na samych przerwaniach.

Nie będę omawiać szczegółów związanych ze zgłaszaniem przerwań na poziomie sprzętu, różnic pomiędzy przerwaniami sprzętowymi i programowymi ani rozwiązań zastosowanych w infrastrukturach jądra związanych ze sterownikiem i magistralą. Wszystkie te zagadnienia można odnaleźć w książkach *Linux Device Drivers* i *Understanding the Linux Kernel*. Tutaj dokonam jedynie krótkiego wprowadzenia w tematykę przerwań, które pomoże zrozumieć, w jaki sposób sterowniki urządzeń inicjują i rejestrują urządzania. Specjalną uwagę poświęcę aspektowi sieciowemu tego zagadnienia.

Przerwania sprzętowe

Znajomość obsługi przerwań sprzętowych na niskim poziomie nie będzie nam potrzebna. Warto jednak wspomnieć o niektórych szczegółach, ponieważ ułatwiają one zrozumienie sposobu implementacji sterowników kart sieciowych i tym samym sposobu ich interakcji z wyższymi warstwami protokołowymi.

Każde przerwanie powoduje wykonanie funkcji zwanej *procedurą obsługi przerwania*, która musi pasować do konkretnego urządzenia i dlatego jest instalowana przez jego sterownik. Zwykle podczas rejestracji urządzenia jego sterownik żąda linii IRQ i przydziela mu ją. Następnie rejestruje i (jeśli sterownik jest wyładowywany) wyrejestrowuje procedurę obsługi danego IRQ za pomocą dwóch funkcji zależnych od architektury systemu. Funkcje te są zdefiniowane w pliku *kernel/irq/manage.c* i są zastępowane funkcjami specyficznymi dla danej architektury umieszczonymi w pliku *arch/XXX/kernel/irq.c*, gdzie *XXX* jest nazwą katalogu dla tej architektury:

`int request_irq(unsigned int irq, void (*handler)(int, void*, struct pt_regs*), unsigned long irqflags, const char * devname, void *dev_id)`
: Funkcja ta rejestruje procedurę obsługi przerwania, upewniając się najpierw, że podane przerwanie jest poprawne i nie jest już przydzielone innemu urządzeniu, chyba że oba urządzenia współdzielą to samo IRQ (podrozdział „Współdzielenie przerwania" w dalszej części tego rozdziału).

`void free_irq(unsigned int irq, void *dev_id)`
: Dla urządzenia identyfikowanego przez `dev_id` funkcja `free_irq` usuwa procedurę obsługi przerwania i wyłącza linię IRQ, jeśli nie używa jej żadne inne urządzenie. Zwróćmy uwagę, że do identyfikacji procedury obsługi jądro wymaga zarówno numeru IRQ, jak i identyfikatora urządzenia. Jest to szczególnie ważne w przypadku współdzielenia jednego IRQ przez kilka urządzeń, co zostanie wytłumaczone w podrozdziale „Współdzielenie przerwania".

Gdy jądro zostaje powiadomione o przerwaniu, używa numeru IRQ w celu znalezienia procedury obsługi związanej z danym sterownikiem i następnie wykonuje ją. Jądro przechowuje związki pomiędzy numerami IRQ i procedurami obsługi w specjalnej, globalnej tabeli. Związki te mogą być typu „jeden do jednego" lub „jeden do wielu", ponieważ jądro systemu Linux pozwala wielu urządzeniom używać tego samego IRQ, co zostanie omówione w podrozdziale „Współdzielenie przerwania".

W kolejnych podrozdziałach pokazano wiele przykładów wymiany informacji pomiędzy urządzeniami i sterownikami za pomocą przerwań, a także sposób współdzielenia jednego IRQ przez wiele urządzeń.

Typy przerwań

Za pomocą przerwania karta sieciowa może poinformować swój sterownik o kilku różnych rzeczach. Wśród nich są:

Odebranie ramki
: Jest to chyba najczęstsza i najbardziej standardowa sytuacja, w której stosowane jest przerwanie.

Błąd transmisji
: Ten rodzaj powiadomienia jest generowany przez urządzenia sieciowe Ethernet tylko w sytuacji, gdy binarne odczekiwanie wykładnicze (zaimplementowane sprzętowo w karcie sieciowej) wykaże błąd. Powiadomienie to nie jest przekazywane przez sterownik do wyższych warstw protokołowych, które dowiedzą się o błędzie w inny sposób (upływ czasu mierzonego przez licznik, negatywne potwierdzenie itd.).

Pomyślne zakończenie transferu DMA
: Bufor, w którym umieszczona została ramka do wysłania, będzie zwolniony przez sterownik, gdy ramka zostanie załadowana do pamięci karty sieciowej. W przypadku transmisji synchronicznej (bez DMA) sterownik „wie" od razu, że ramka została umieszczona w pamięci karty. Natomiast w przypadku transmisji asynchronicznej (z użyciem DMA) sterownik musi zaczekać na przerwanie pochodzące od karty. Przykład obu przypadków można znaleźć w miejscach wywołania funkcji `dev_kfree_skb`[1] wewnątrz kodu sterownika w pliku *drivers/net/3c59x.c* (DMA) i *drivers/net/3c509.c* (bez DMA).

Urządzenie ma wystarczająco pamięci do obsługi nowej transmisji
: Sterownik karty sieciowej blokuje dostęp do kolejki wyjściowej, gdy nie ma ona już miejsca, aby przyjąć ramkę maksymalnego rozmiaru (czyli 1536 bajtów w przypadku karty sieciowej Ethernet). Dostęp do kolejki zostaje przywrócony, gdy dostępny jest odpowiedni obszar pamięci. Omówieniu tego przypadku poświęcimy resztę tego podrozdziału.

Ostatni z przypadków wymienionych na powyższej liście dotyczy zaawansowanego sposobu dławienia transmisji, który jeśli jest prawidłowo zastosowany, może poprawić efektywność. Gdy kolejka jest pełna, sterownik blokuje możliwość transmisji, równocześnie zlecając karcie sieciowej powiadomienie go za pomocą przerwania, gdy zwiększy się obszar dostępnej pamięci (zwykle do wartości odpowiadającej MTU). Gdy przerwanie zostanie wygenerowane, sterownik odblokuje możliwość transmisji.

Sterownik może również wyłączyć kolejkę wyjściową przed transmisją (aby zapobiec wygenerowaniu kolejnego żądania transmisji przez jądro) i włączyć ją z powrotem tylko wtedy, gdy karta sieciowa dysponuje odpowiednio dużym obszarem pamięci. W przeciwnym razie urządzenie wymaga przerwania, które pozwoli mu później wznowić transmisję. Oto przykład takiego działania zaczerpnięty z procedury `el3_start_xmit`, którą sterownik *drivers/net/3c509.c* instaluje jako swoją funkcję `hard_start_xmit`[2] we własnej strukturze `net_device`:

```
static int
el3_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    ... ... ...
    netif_stop_queue (dev);
    ... ... ...
    if (inw(ioaddr + TX_FREE) > 1536)
        netif_start_queue(dev);
    else
        outw(SetTxThreshold + 1536, ioaddr + EL3_CMD);
    ... ... ...
}
```

Sterownik zatrzymuje kolejkę za pomocą funkcji `netif_stop_queue`, uniemożliwiając tym samym jądru generowanie kolejnych żądań transmisji. Następnie sterownik sprawdza, czy wolna pamięć urządzenia pomieści pakiet o rozmiarze 1536 bajtów. Jeśli tak, to sterownik odblokowuje kolejkę, umożliwiając jądru wysyłanie kolejnych żądań transmisji. W przeciwnym razie instruuje urządzenie (poprzez zapis do rejestru konfiguracyjnego za pomocą wywołania `outw`), aby wygenerowało przerwanie, gdy warunek zostanie spełniony. Procedura obsługi przerwania przywróci wtedy działanie kolejki za pomocą funkcji `netif_start_queue` i jądro będzie mogło przywrócić transmisję.

[1] Funkcja ta jest omówiona szczegółowo w rozdziale 11.

[2] Funkcja wirtualna `hard_start_xmit` jest omówiona w rozdziale 11.

Funkcje `netif_xxx_queue` zostaną omówione w podrozdziale „Włączanie i wyłączanie transmisji" zamieszczonym w rozdziale 11.

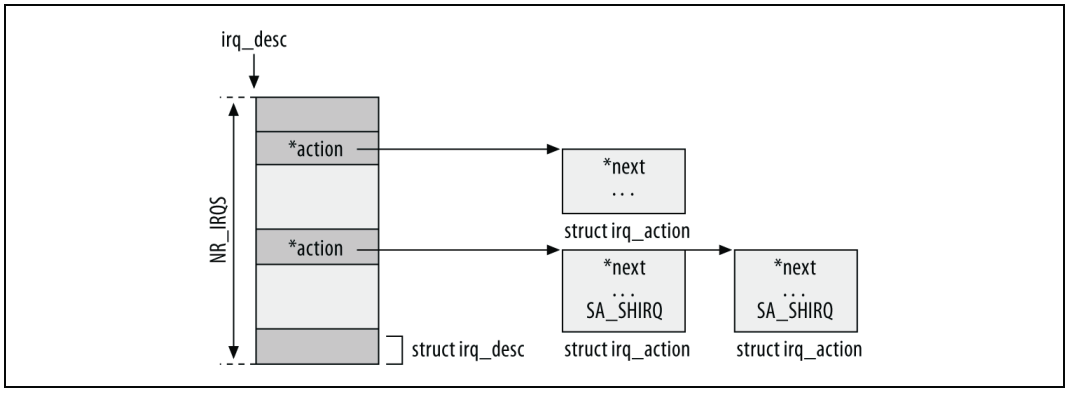
Współdzielenie przerwania

Linie IRQ są zasobem o ograniczonej dostępności. Prosty sposób pozwalający na zwiększenie liczby urządzeń w systemie polega na umożliwieniu wielu urządzeniom wspólnego korzystania z jednego IRQ. Każdy sterownik rejestruje własną procedurę obsługi tego IRQ. Natomiast jądro zamiast odbierać powiadomienie o przerwaniu, odnajdywać właściwe urządzenie i wywoływać należącą do niego procedurę obsługi przerwania, po prostu wywołuje wszystkie procedury obsługi należące do urządzeń, które zarejestrowały się dla tego samego IRQ. Odfiltrowanie niepotrzebnych powiadomień należy już do procedur obsługi i może odbywać się na przykład poprzez odczyt rejestru w urządzeniu.

Aby grupa urządzeń mogła współdzielić linię IRQ, wszystkie należące do niej urządzenia muszą mieć sterowniki urządzeń umożliwiające obsługę współdzielonego IRQ. Innymi słowy, za każdym razem, gdy urządzenie rejestruje się dla danej linii IRQ, musi jawnie określić, czy obsługuje współdzielone przerwanie. Na przykład pierwsze urządzenie rejestrujące dla pewnego IRQ *n* procedurę obsługi *fn* musi również określić, czy może współdzielić to IRQ z innymi urządzeniami. Gdy kolejny sterownik urządzenia próbuje się zarejestrować dla tego samego IRQ, to żądanie rejestracji zostanie odrzucone, jeśli sterownik ten lub sterownik, do którego przypisano IRQ, nie potrafi obsługiwać współdzielenia przerwań.

Organizacja odwzorowania pomiędzy liniami IRQ i procedurami obsługi

Odwzorowanie przerwań IRQ na procedury ich obsługi przechowywane jest w postaci wektora list zawierającego po jednej liście procedur obsługi dla każdej linii IRQ (rysunek 5.2). Lista taka zawiera więcej niż jeden element tylko wtedy, gdy dane przerwanie jest współdzielone przez wiele urządzeń. Rozmiar wektora (czyli liczba linii IRQ) zależy od konkretnej architektury i może wynosić od 15 (w przypadku procesorów rodziny x86) do ponad 200. Wprowadzenie współdzielenia przerwań umożliwia pracę odpowiednio większej liczby urządzeń w jednym systemie.



Rysunek 5.2. Organizacja procedur obsługi IRQ

W podrozdziale „Przerwania sprzętowe" wprowadzone zostały dwie funkcje służące do zarejestrowania i wyrejestrowania procedur obsługi. Teraz przyjrzymy się strukturom danych służącym do reprezentacji odwzorowań pomiędzy przerwaniami i procedurami ich obsługi.

Odwzorowania zostają zdefiniowane za pomocą struktur danych typu `irqaction`. Funkcja `request_irq` wprowadzona we wcześniejszym podrozdziale „Przerwania sprzętowe" obudowuje funkcję `setup_irq`, której parametrem wejściowym jest struktura `irqaction` umieszczana następnie w globalnym wektorze `irq_desc`. Struktura `irq_desc` jest zdefiniowana w pliku *kernel/irq/handler.c*, który może zostać zastąpiony plikiem *arch/XXX/kernel/irq.c* dla konkretnej architektury. Funkcja `setup_irq` jest zdefiniowana w pliku *kernel/irq/manage.c*, który również może zostać zastąpiony plikiem *arch/XXX/kernel/irq.c* dla konkretnej architektury.

Funkcja jądra obsługująca przerwania i przekazująca je sterownikom zależy od konkretnej architektury. W większości przypadków nosi nazwę `handle_IRQ_event`.

Na rysunku 5.2 przedstawiony został sposób przechowywania instancji `irqaction`: dla każdej linii IRQ istnieje instancja struktury `irq_desc`, a dla każdej pomyślnie zarejestrowanej procedury obsługi IRQ istnieje instancja struktury `irqaction`. Wektor instancji `irq_desc` również nosi nazwę `irq_desc`, a jego rozmiar jest określony symbolem `NR_IRQS`, którego wartość zależy od konkretnej architektury.

Zwróćmy uwagę, że gdy dla danego numeru IRQ (czyli danego elementu wektora `irq_desc`) istnieje więcej niż jedna instancja struktury `irqaction`, to wymagana jest obsługa współdzielenia przerwania (każda struktura musi mieć ustawiony znacznik `SA_SHIRQ`).

Przyjrzyjmy się teraz, jakie informacje o procedurach obsługi IRQ są przechowywane w polach struktury `irqaction`:

`void` `(*handler)(`**`int`** `irq, `**`void`** `*dev_id, `**`struct`** `pt_regs *regs)`
: Funkcja dostarczana przez sterownik urządzenia do obsługi powiadomień o przerwaniach: za każdym razem, gdy jądro odbierze przerwanie na linii `irq` wywoła funkcję `handler`. A oto parametry wejściowe tej funkcji:

  `int` `irq`
  : Numer linii IRQ, która wygenerowała powiadomienie. W większości przypadków informacja ta nie jest używana przez sterowniki kart sieciowych, którym wystarcza identyfikator urządzenia.

  `void` `*dev_id`
  : Identyfikator urządzenia. Ten sam sterownik urządzenia może być odpowiedzialny za działanie wielu urządzeń. Poprawna obsługa powiadomienia wymaga więc identyfikatora konkretnego urządzenia.

  `struct` `pt_regs *regs`
  : Struktura używana do przechowania zawartości rejestrów procesora w momencie przerwania bieżącego procesu. Zwykle nie jest używana przez funkcję obsługi przerwania.

`unsigned long` `flags`
: Zbiór znaczników. Wartości `SA_`*`XXX`* są zdefiniowane w pliku nagłówkowym *include/asm-XXX/signal.h*. Oto najważniejsze z nich, dla architektury x86:

  `SA_SHIRQ`
  : Gdy znacznik ten jest ustawiony, to sterownik urządzenia może obsługiwać współdzielone przerwanie.

`SA_SAMPLE_RANDOM`
: Gdy znacznik ten jest ustawiony, to urządzenie może zostać użyte jako źródło zdarzeń losowych. Pomaga to jądru generować losowe wartości przeznaczone do wewnętrznego użytku w celu zwiększenia *entropii systemu*. Zagadnienie to zostanie rozwinięte w podrozdziale „Inicjalizacja warstwy obsługi urządzeń: net_dev_init".

`SA_INTERRUPT`
: Gdy znacznik ten jest ustawiony, to podczas wykonywania procedury obsługi wyłączone są przerwania na lokalnym procesorze. Znacznik ten powinno się ustawiać tylko w przypadku procedur, które bardzo szybko kończą swoje działanie. Warto przeanalizować jedną z instancji `handle_IRQ_event` (na przykład */kernel/irq/handle.c*).

  Istnieją również wartości reprezentujące inne znaczniki, ale są one albo przestarzałe, albo używane wyłącznie przez poszczególne architektury.

`void` `*dev_id`
: Wskaźnik struktury `net_device` związanej z urządzeniem. Wskaźnik ten zadeklarowano jako **`void`** `*`, ponieważ karty sieciowe nie są jedynymi urządzeniami używającymi linii IRQ. Ponieważ różne typy urządzeń używają różnych struktur danych w celu identyfikacji i reprezentacji instancji urządzeń, stąd taki ogólny sposób deklaracji wskaźnika.

`struct` `irqaction *next`
: Wszystkie urządzenia współdzielące to samo IRQ są połączone w listę za pomocą tego wskaźnika.

`const char` `*name`
: Nazwa urządzenia. Można ją odczytać, wyświetlając zawartość pliku */proc/interrupts*.

Opcje inicjalizacji

Zarówno komponenty wbudowane w jądro, jak i ładowane jako moduły mogą otrzymywać parametry wejściowe pozwalające użytkownikom stroić działanie tych komponentów, modyfikować ich domyślne parametry lub zmieniać je podczas każdego uruchamiania systemu. Jądro udostępnia dwa rodzaje makr umożliwiających definiowanie opcji:

Opcje modułów (makra rodziny `module_param`*)*
: Makra te definiują opcje, których można użyć podczas ładowania modułu. Gdy komponent jest wbudowany w jądro, to wartości tych opcji nie mogą zostać użyte podczas uruchamiania jądra. Jednak dzięki wprowadzeniu systemu plików */sys* opcje te można konfigurować za pośrednictwem plików podczas działania systemu. Interfejs */sys* jest stosunkowo nowym rozwiązaniem w porównaniu z interfejsem */proc*. Więcej szczegółów na temat tych opcji można znaleźć w podrozdziale „Opcje modułów" zamieszczonym w dalszej części tego rozdziału.

Opcje uruchamiania jądra (makra rodziny `__setup`*)*
: Makra te definiują opcje, których używamy podczas uruchamiania systemu. Wykorzystywane są głównie przez moduły, które użytkownik może wbudować w jądro systemu, oraz komponenty jądra, których nie można skompilować jako moduły. Zastosowanie tych makr pokazano w podrozdziale „Opcje uruchamiania jądra" zamieszczonym w rozdziale 7.

Warto zauważyć, że moduł może definiować opcje inicjalizacji na dwa sposoby: jeden efektywny w przypadku modułu wbudowanego w jądro i drugi, działający dla modułów ładowanych oddzielnie. Sytuacja taka może być nieco myląca, zwłaszcza że różne moduły mogą definiować przekazywanie parametrów o tej samej nazwie podczas ich ładowania bez jakiegokolwiek ryzyka kolizji nazw (ponieważ parametry są przekazywane właśnie ładowanemu modułowi). Natomiast jeśli przekazujemy te parametry podczas uruchamiania jądra, to musimy upewnić się, że nie ma kolizji nazw pomiędzy opcjami różnych modułów.

Nie będziemy tutaj omawiać zalet i wad obu rozwiązań. Czytelny przykład użycia opcji rodziny `module_param`, jak i `__setup` można znaleźć w pliku sterownika *drivers/block/loop.c*.

Opcje modułów

Moduły jądra definiują swoje parametry za pomocą makr rodziny `module_param`, których listę można znaleźć w pliku nagłówkowym *include/linux/moduleparam.h*. Makro `module_param` wymaga trzech parametrów, co ilustruje poniższy przykład zaczerpnięty z pliku *drivers/net/sis900.c*:

```
...
module_param(multicast_filter_limit, int, 0444);
module_param(max_interrupt_work, int, 0444);
module_param(debug, int, 0444);
...
```

Pierwszy z nich jest nazwą parametru przeznaczoną dla użytkownika. Drugi określa typ parametru (np. **`int`**), a trzeci reprezentuje prawa dostępu do pliku w */sys*, do którego zostanie wyeksportowany parametr.

Na skutek wyświetlenia zawartości katalogu modułów w */sys* uzyskano by w tym przypadku następujące informacje:

```
[root@localhost src]# ls -la /sys/modules/sis900/parameters/
total 0
drwxr-xr-x  2 root root    0 Apr  9 18:31 .
drwxr-xr-x  4 root root    0 Apr  9 18:31 ..
-r--r--r--  1 root root    0 Apr  9 18:31 debug
-r--r--r--  1 root root 4096 Apr  9 18:31 max_interrupt_work
-r--r--r--  1 root root 4096 Apr  9 18:31 multicast_filter_limit
[root@localhost src]#
```

Każdemu modułowi przypisany jest katalog w */sys/modules*. W podkatalogu */sys/modules/module/parameters* znajduje się plik dla każdego z parametrów eksportowanych przez moduł *module*. Ostatni przykład pochodzący z pliku *drivers/net/sis900.c* pokazuje trzy pliki reprezentujące parametry, które mogą być odczytywane przez każdego użytkownika, ale nie mogą być modyfikowane.

Prawa dostępu do plików w katalogu */sys* (a przy okazji również do plików w */proc*) są definiowane w taki sam sposób jak dla zwykłych plików. Możemy więc definiować prawo odczytu, zapisu i wykonania dla właściciela pliku, grupy i wszystkich użytkowników. Na przykład wartość 400 oznacza prawo odczytu pliku przez właściciela pliku (którym jest użytkownik *root*) i żadnego dostępu dla nikogo więcej. Gdy wartość wynosi 0, to nikt nie ma żadnych uprawnień do tego pliku i nie jest on nawet widoczny w */sys*.

Jeśli programista komponentu chce umożliwić użytkownikowi odczyt parametru, to musi przydzielić mu co najmniej prawo odczytu. Może również przydzielić prawo zapisu, jeśli użytkownik ma modyfikować wartość parametru. Jednak należy pamiętać, że moduł, który

wyeksportował parametr, nie jest powiadamiany o zmianach w pliku i musi dysponować własnym mechanizmem ich wykrywania.

Szczegółowy opis interfejsu */sys* można znaleźć w książce *Linux Device Drivers*.

Inicjalizacja warstwy obsługi urządzeń: net_dev_init

Ważna część inicjalizacji kodu sieciowego, w tym sterowania ruchem i kolejek wejściowych dla poszczególnych procesorów, wykonywana jest podczas uruchamiania systemu przez funkcję `net_dev_init` zdefiniowaną w pliku *net/core/dev.c*:

```
static int __init net_dev_init(void)
{
    ...
}
subsys_initcall(net_dev_init);
```

W rozdziale 7. pokazano, w jaki sposób makro `subsys_initcall` gwarantuje wykonanie funkcji `net_dev_init`, zanim jakikolwiek sterownik karty sieciowej zostanie wywołany, oraz wyjaśniono, dlaczego jest to tak ważne. Wyjaśniono również, dlaczego funkcja `net_dev_init` jest oznaczona makrem `__init`.

Przeanalizujmy główne kroki podejmowane przez funkcję `net_dev_init`:

- Funkcja inicjuje struktury danych dla poszczególnych procesorów używane przez dwa sieciowe przerwania programowe. W rozdziale 9. wyjaśniono, czym są przerwania programowe i w jaki sposób są używane przez kod sieciowy.
- Jeśli jądro zostało skompilowane z obsługą systemu plików */proc* (co jest domyślną konfiguracją jądra), to w */proc* zostaje umieszczonych kilka plików przez funkcje `dev_proc_init` i `dev_mcast_init`. Więcej szczegółów można znaleźć w podrozdziale „Strojenie za pośrednictwem systemu plików /proc" zamieszczonym w dalszej części tego rozdziału.
- Funkcja `netdev_sysfs_init` rejestruje klasę `net` w `sysfs`. W ten sposób powstaje katalog */sys/class/net*, w którym znajdzie się osobny podkatalog dla każdego zarejestrowanego urządzenia sieciowego. Katalogi te będą zawierać sporo plików, niektóre z nich znane również z */proc*.
- Funkcja `net_random_init` inicjuje dla każdego procesora wektor posiewów, które będą używane podczas generowania liczb losowych za pomocą funkcji `net_random`. Funkcja `net_random` jest używana w różnych kontekstach omówionych w dalszej części tego podrozdziału.
- Funkcja `dst_init` inicjuje bufor DST omówiony w rozdziale 33.
- Zainicjowany zostaje wektor procedur obsługi protokołów `ptype_base` używany do demultipleksowania ruchu wejściowego. Więcej informacji na ten temat w rozdziale 13.
- Gdy zdefiniowany jest symbol `OFFLINE_SAMPLE`, jądro konfiguruje funkcję wykonywaną w regularnych odstępach czasu w celu zbierania danych statystycznych o długości kolejek urządzeń. Funkcja `net_dev_init` musi utworzyć licznik czasu, który pozwoli regularnie wywoływać wspomnianą funkcję. Więcej informacji na ten temat można znaleźć w podrozdziale „Średnia długość kolejki i wyznaczanie poziomu przeciążenia" zamieszczonym w rozdziale 10.

- Funkcja zwrotna `dev_cpu_callback` zostaje zarejestrowana w łańcuchu powiadomień o zdarzeniach związanych z dołączaniem kolejnych procesorów podczas pracy systemu. Obecnie przetwarzane jest jedynie zdarzenie polegające na zatrzymaniu pracy procesora. Gdy odebrane zostanie powiadomienie o tym zdarzeniu, z kolejki wejściowej procesora usuwane są bufory, które następnie są przekazywane funkcji `netif_rx`. Więcej informacji na temat działania kolejek wejściowych dla poszczególnych procesorów można znaleźć w rozdziale 9.

Generowanie liczb losowych wykorzystywane jest przez jądro w celu nadania losowości niektórym jego działaniom. Podczas lektury tej książki Czytelnik dowie się, że wiele podsystemów sieciowych używa wartości wygenerowanych w sposób losowy. Na przykład często dodają losowo wybrany składnik do wartości liczników czasu, zmniejszając w ten sposób prawdopodobieństwo równoczesnego wykonania zbyt wielu operacji i związanego z tym nadmiernego obciążenia procesora. Zastosowanie wartości losowych pozwala również zapobiegać atakom typu DoS (*Denial of Service*) próbującym odgadnąć sposób organizacji pewnych struktur danych.

Stopień, w jakim wartości używane przez jądro mogą zostać uznane za rzeczywiście losowe, nazywany jest *entropią systemu*. Do jego poprawy wykorzystywane są komponenty jądra, których działanie ma aspekt niedeterministyczny. Do kategorii tej często należą właśnie urządzenia sieciowe. W obecnej wersji systemu tylko kilka sterowników kart sieciowych może być używanych w celu zwiększenia entropii systemu (o czym wspomina wcześniejsze omówienie znacznika `SA_SAMPLE_RANDOM`). Łata jądra 2.4 wprowadza opcję kompilacji, która włącza lub wyłącza wkład kart sieciowych do entropii systemu. Szukając w sieci słowa kluczowego „SA_SAMPLE_NET_RANDOM", można znaleźć informacje na temat aktualnej wersji.

Kod tradycyjny

W poprzednim podrozdziale wspomniałem, że makra `subsys_initcall` gwarantują wykonanie funkcji `net_dev_init`, zanim jakikolwiek sterownik urządzenia zdoła zarejestrować swoje urządzenie. Przed wprowadzeniem tego mechanizmu porządek wykonania był wymuszany w inny sposób, poprzez użycie przestarzałego mechanizmu jednorazowego znacznika.

Globalna zmienna `dev_boot_phase` była używana jako znacznik logiczny informujący o tym, czy funkcja `net_dev_init` ma być wykonana. Znacznik ten był inicjowany wartością 1 (funkcja `net_dev_init` nie była jeszcze wykonana), a następnie był kasowany przez funkcję `net_dev_init`. Za każdym razem, gdy funkcja `register_netdevice` była wywoływana przez sterownik urządzenia, sprawdzała wartość znacznika `dev_boot_phase` i jeśli był on ustawiony, wykonywała funkcję `net_dev_init`.

Mechanizm ten nie jest już potrzebny, ponieważ funkcja `register_netdevice` nie może zostać wywołana przed funkcją `net_dev_init`, jeśli tylko zastosowano właściwe oznaczenie kluczowych procedur sterowników (rozdział 7.). Jednak aby wykryć błędy w oznaczeniach tych procedur lub błędy kodu, funkcja `net_dev_init` nadal kasuje znacznik `dev_boot_phase`, a funkcja `register_netdevice` używa makra `BUG_ON` gwarantującego, że nie zostanie wywołana, gdy znacznik `dev_boot_phase` jest ustawiony[3].

[3] Zastosowanie makr `BUG_ON` i `BUG_TRAP` jest typowym przykładem mechanizmu gwarantującego spełnienie koniecznych warunków w określonych punktach kodu. Mechanizm taki jest przydatny podczas przechodzenia do nowego rozwiązania pewnego problemu.

Kod pomocniczy w przestrzeni użytkownika

Istnieją przypadki, że wywołanie przez jądro aplikacji działających w przestrzeni użytkownika w celu obsługi zdarzeń rzeczywiście ma sens. Szczególnie ważne są dwa z nich:

/sbin/modprobe
: Wywoływany, gdy jądro ładuje moduł. Program ten stanowi część pakietu *module-init-tools*.

/sbin/hotplug
: Wywoływany, gdy jądro wykryje, że nowe urządzenie zostało włączone do systemu (lub wyłączone z niego). Zadaniem tego programu jest załadowanie właściwego sterownika urządzenia na podstawie identyfikatora urządzenia. Urządzenia są identyfikowane na podstawie magistrali, do której są przyłączone (np. PCI) i identyfikatora zdefiniowanego przez specyfikację danej magistrali[4]. Program ten jest częścią pakietu *hotplug*.

Jądro posiada funkcję `call_usermodehelper` pozwalającą wykonywać kod pomocniczy w przestrzeni użytkownika. Funkcja ta umożliwia przekazanie uruchamianej aplikacji zmiennej liczby parametrów w `arg[]` i zmiennych środowiskowych w `env[]`. Na przykład pierwszy parametr `arg[]` informuje funkcję `call_usermodehelper`, jaki program należy uruchomić w przestrzeni użytkownika, a parametr `arg[1]` może zostać użyty do przekazania temu programowi skryptu konfiguracyjnego. Przykład pokazano w podrozdziale „/sbin/hotplug" zamieszczonym w dalszej części tego rozdziału.

Na rysunku 5.3 przedstawiony został sposób, w jaki dwie procedury jądra, `request_module` i `kobject_hotplug`, używają funkcji `call_usermodehelper` w celu wywołania programów pomocniczych */sbin/modprobe* i */sbin/hotplug* działających w przestrzeni użytkownika. Rysunek pokazuje również sposób inicjalizacji tablic `arg[]` i `env[]` w obu przypadkach. W następnych podrozdziałach zajmiemy się nieco bardziej szczegółowo omówieniem obu programów pomocniczych.

kmod

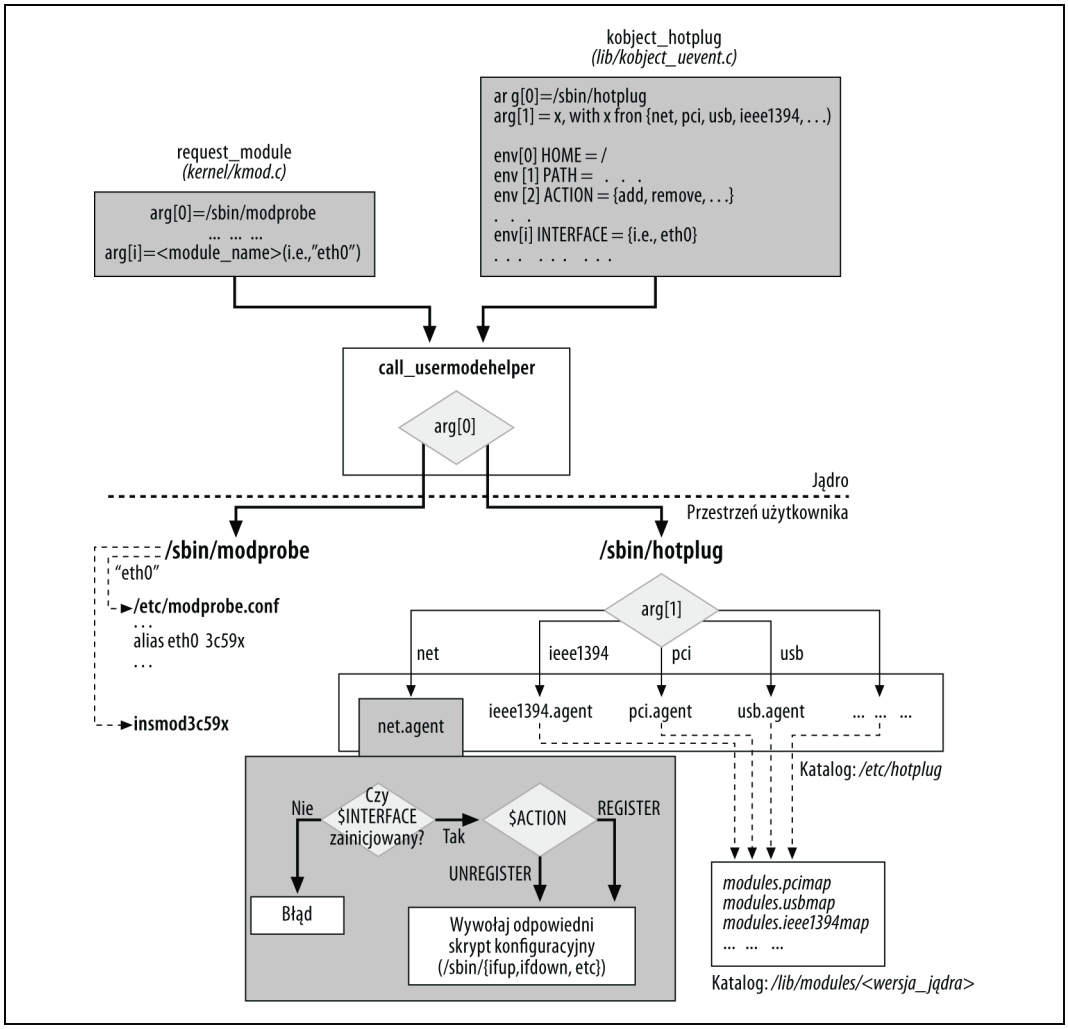
kmod jest procedurą ładującą moduły jądra, pozwalającą komponentom jądra żądać załadowania modułu. Jądro udostępnia w tym celu więcej niż jedną funkcję, ale tutaj zajmiemy się jedynie omówieniem funkcji `request_module`. Funkcja ta inicjuje `arg[1]` nazwą ładowanego modułu. */sbin/modprobe* używa pliku konfiguracyjnego */etc/modprobe.conf*, który pozwala mu na przykład sprawdzić, czy nazwa modułu otrzymana od jądra nie jest w rzeczywistości synonimem czego innego (rysunek 5.3).

Poniżej przedstawiamy dwa przykłady zdarzeń, które spowodują, że jądro zażąda od */sbin/modprobe* załadowania modułu:

- Administrator używa programu *ifconfig* do skonfigurowania karty sieciowej, której sterownik nie został jeszcze załadowany, na przykład *eth0*[5]. Wtedy jądro wysyła żądanie do */sbin/modprobe*, aby załadował moduł, którego nazwą jest `"eth0"`. Jeśli plik konfiguracyjny */etc/modprobe.conf* zawiera pozycję `"alias eth0 3c59x"`, to */sbin/modprobe* próbuje załadować moduł *3c59x.ko*.

[4] Przykład dla magistrali PCI można znaleźć w podrozdziale „Rejestracja sterownika karty sieciowej PCI" zamieszczonym w rozdziale 6.

[5] Zwróćmy uwagę, że *eth0* jeszcze nie istnieje, ponieważ sterownik nie został załadowany.



Rysunek 5.3. Propagacja zdarzeń z jądra do przestrzeni użytkownika

- Administrator konfiguruje sterowanie ruchem dla pewnego urządzenia, używając polecenia *tc* należącego do pakietu *IPROUTE2*. Może odwołać się wtedy do reguły kolejkowania lub klasyfikatora, które nie znajdują się w jądrze. W takim przypadku jądro żąda od */sbin/modprobe* załadowania odpowiedniego modułu.

Więcej informacji na temat modułów i *kmod* można znaleźć w książce *Linux Device Drivers*.

Hotplug

Opcję Hotplug wprowadzono do jądra systemu Linux w celu obsługi coraz popularniejszych urządzeń PnP (*Plug and Play*). Jądro może wykrywać podłączenie lub odłączenie takich urządzeń i powiadamiać o tym aplikacje działające w przestrzeni użytkownika, a także przekazuje im dość szczegółów, aby mogły załadować odpowiedni sterownik i zastosować związaną z nim konfigurację (jeśli taka istnieje).

Hotplug może być również używana do obsługi tradycyjnych urządzeń podczas uruchamiania systemu. Nie ma znaczenia, czy urządzenie zostało podłączone w czasie pracy systemu, czy było już włączone podczas jego uruchamiania. W obu przypadkach powiadomiony zostaje kod pomocniczy działający w przestrzeni użytkownika. Aplikacja działająca w przestrzeni użytkownika decyduje o tym, czy zdarzenie to wymaga z jej strony podjęcia pewnych działań.

System Linux, podobnie jak większość systemów Unix, wykonuje podczas startu zbiór skryptów służących do inicjalizacji urządzeń peryferyjnych, w tym urządzeń sieciowych. Składnia, nazwy i położenie tych skryptów zmienia się dla różnych dystrybucji systemu Linux. (Na przykład dystrybucje używające modelu init pochodzącego z Systemu V mają odpowiednie katalogi w */etc/rc.d/* wraz z plikami konfiguracyjnymi informującymi o tym, co należy uruchomić. Inne dystrybucje są albo oparte na modelu BSD, albo używają go w trybie zgodności z Systemem V.) Dlatego też powiadomienia o urządzeniach obecnych podczas uruchamiania systemu mogą zostać zignorowane, ponieważ skrypty i tak skonfigurują te urządzenia.

Gdy kompilujemy moduły jądra, to pliki wynikowe zostają domyślnie umieszczone w katalogu */lib/modules/wersja_jądra*, gdzie *wersja_jądra* może być na przykład równa 2.6.12. W tym samym katalogu możemy znaleźć dwa interesujące pliki: *modules.pcimap* i *modules.usbmap*. Pliki te zawierają odpowiednio: identyfikatory urządzeń PCI[6] i USB obsługiwanych przez jądro. Te same pliki zawierają dla każdego identyfikatora urządzenia referencję związanego z nim modułu jądra. Gdy program pomocniczy działający w przestrzeni użytkownika otrzyma powiadomienie o włączeniu urządzenia PnP, używa tych plików do odnalezienia odpowiedniego sterownika urządzenia.

Pliki *modules.xxxmap* są wypełniane informacją na podstawie wektorów identyfikatorów dostarczanych przez sterowniki urządzeń. W podrozdziale „Przykład rejestracji sterownika karty sieciowej PCI" zamieszczonym w rozdziale 6. pokażę, w jaki sposób sterownik Vortex inicjuje swoją instancję `pci_device_id`. Ponieważ sterownik ten został napisany dla urządzenia PCI, to zawartość tej tablicy trafi do pliku modules.pcimap.

Czytelnik zainteresowany najnowszą wersją kodu Hotplug znajdzie więcej informacji na stronie *http://linux-hotplug.sourceforge.net*.

/sbin/hotplug

Domyślnym programem pomocniczym, wykonywanym w przestrzeni użytkownika dla opcji Hotplug jest skrypt[7] */sbin/hotplug* należący do pakietu Hotplug. Pakiet ten może zostać skonfigurowany za pomocą plików umieszczonych w domyślnych katalogach */etc/hotplug/* i */etc/hotplug.d/*.

Funkcja `kobject_hotplug` jest wywoływana przez jądro między innymi w odpowiedzi na podłączenie lub odłączenie urządzenia. Funkcja `kobject_hotplug` inicjuje `arg[0]` jako */sbin/hotplug*, a `arg[1]` — odpowiednim agentem. */sbin/hotplug/* jest bowiem prostym skryptem, który przekazuje obsługę zdarzenia innemu skryptowi (agentowi) w oparciu o parametr `arg[1]`.

Agenty kodu pomocniczego wykonywanego w przestrzeni użytkownika mogą być mniej lub bardziej skomplikowane w zależności od tego, na ile „inteligentny" ma być proces autokonfiguracji. Skrypty dostarczane z pakietem Hotplug próbują rozpoznać dystrybucję systemu

[6] Krótki opis identyfikatorów urządzeń PCI znajduje się w podrozdziale „Przykład rejestracji sterownika karty sieciowej PCI" zamieszczonym w rozdziale 6.

[7] Administrator może tworzyć własne skrypty lub wykorzystać dostarczane z większością dystrybucji systemu Linux.

Linux i dopasować swoje działanie do odpowiedniej składni plików konfiguracyjnych i ich położenia.

Przeanalizujmy przykład działania opcji Hotplug dla urządzenia sieciowego. Gdy karta sieciowa zostaje dodana do systemu lub z niego usunięta, funkcja `kobject_hotplug` inicjuje parametr `arg[1]` jako *net*, co prowadzi do wywołania agenta *net.agent* przez */sbin/hotplug*.

W przeciwieństwie do innych agentów przedstawionych na rysunku 5.3, *net.agent* nie reprezentuje ani medium transmisji danych, ani typu magistrali. Podczas gdy inne agenty są używane do ładowania odpowiednich modułów (sterowników urządzeń) na podstawie identyfikatorów urządzeń, *net.agent* jest używany w celu skonfigurowania urządzenia.

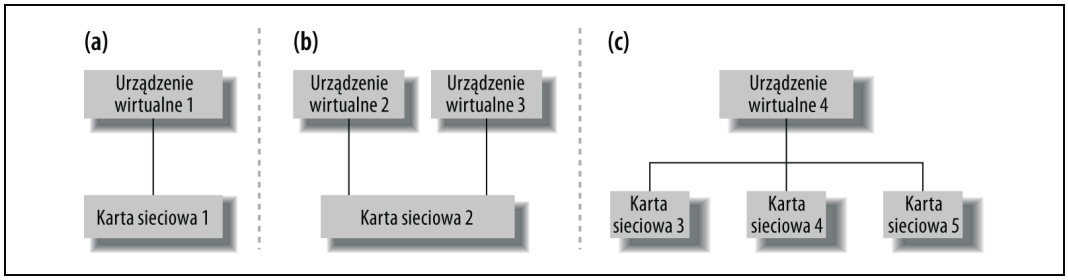
Zadaniem *net.agent* jest zastosowanie konfiguracji związanej z nowym urządzeniem, wobec czego musi on otrzymać od jądra przynajmniej identyfikator tego urządzenia. W przykładzie pokazanym na rysunku 5.3 identyfikator urządzenia zostaje przekazany przez jądro za pomocą zmiennej środowiskowej `INTERFACE`.

Aby urządzenie mogło zostać skonfigurowane, musi najpierw zostać stworzone i zarejestrowane w jądrze. Zadanie to jest zwykle wykonywane przez sterownik urządzenia, który wobec tego musi zostać najpierw załadowany. Na przykład dodanie karty sieciowej PCMCIA Ethernet spowoduje kilka wywołań programu */sbin/hotplug*. Będą wśród nich:

- Wywołanie prowadzące do wykonania programu */sbin/modprobe*[8], który zajmie się załadowaniem odpowiedniego modułu sterownika. W przypadku kart PCMCIA sterownik zostaje załadowany przez agenta *pci.agent* (przy użyciu akcji ADD).
- Wywołanie konfigurujące nowe urządzenie. Zadanie to jest realizowane przez agenta *net.agent* (ponownie przy użyciu akcji ADD).

Urządzenia wirtualne

Urządzenie wirtualne jest abstrakcją zbudowaną w oparciu o jedno lub więcej rzeczywistych urządzeń. Związek pomiędzy urządzeniami wirtualnymi i rzeczywistymi może być typu wiele do wielu, co ilustrują trzy modele przedstawione na rysunku 5.4. Możliwe jest również tworzenie kolejnych urządzeń wirtualnych w oparciu o inne takie urządzenia. Jednak nie wszystkie takie kombinacje są obsługiwane przez jądro.



Rysunek 5.4. Możliwe związki pomiędzy wirtualnymi i rzeczywistymi urządzeniami

[8] W przeciwieństwie do programu */sbin/hotplug* będącego skryptem powłoki program */sbin/modprobe* jest binarnym plikiem wykonywalnym. Jeśli Czytelnik chce przeanalizować jego działanie, powinien pobrać kod źródłowy pakietu *modutil*.

Przykłady urządzeń wirtualnych

System Linux pozwala definiować różne rodzaje urządzeń wirtualnych. Oto kilka przykładów:

Bonding
: Urządzenie wirtualne tego typu sprawia, że grupa rzeczywistych urządzeń zachowuje się jak pojedyncze urządzenie.

802.1Q
: Standard IEEE rozszerzający nagłówek 802.3/Ethernet o tak zwany nagłówek VLAN umożliwiający tworzenie wirtualnych sieci lokalnych.

Most
: Interfejs mostu stanowi wirtualną reprezentację mostu. Szczegóły w części IV książki.

Interfejsy zastępcze
: Początkowo głównym zastosowaniem tego mechanizmu było umożliwienie reprezentowania pojedynczego interfejsu Ethernet przez wiele interfejsów wirtualnych (*eth0:0*, *eth0:1* i tak dalej), z których każdy posiadał własną konfigurację IP. Obecne, ulepszone wersje kodu sieciowego nie potrzebują już interfejsów wirtualnych, aby skonfigurować wiele adresów IP dla tej samej karty sieciowej. Mogą jednak pojawić się sytuacje (związane z routingiem), gdy istnienie wielu interfejsów wirtualnych dla tej samej karty sieciowej upraszcza konfigurację sieci. Szczegóły w rozdziale 30.

TEQL (True equalizer)
: Reguła kolejkowania używana przez sterowanie ruchem. Jej implementacja wymaga utworzenia specjalnego urządzenia. Zasada działania TEQL przypomina nieco Bonding.

Interfejsy tunelujące
: Implementacje tunelowania protokołu IP przez protokół IP (IPIP) oraz protokół GRE (*Generalized Routing Encapsulation*) opierają swoje działanie na tworzeniu urządzeń wirtualnych.

Powyższa lista nie jest kompletna. Biorąc pod uwagę tempo, w jakim nowe opcje wprowadzane są do jądra systemu Linux, można oczekiwać pojawienia się wielu nowych urządzeń wirtualnych.

Urządzenia wirtualne związane z mostkowaniem, opcją Bonding i 802.1Q są przykładami modelu przedstawionego na rysunku 5.4(c). Interfejsy zastępcze są natomiast przykładem modelu przedstawionego na rysunku 5.4(b). Model 5.4(a) można traktować jako specjalny przypadek pozostałych dwóch modeli.

Interakcja ze stosem sieciowym jądra

Interakcje z jądrem różnią się nieco w przypadku urządzeń wirtualnych i rzeczywistych. Dotyczy to przede wszystkim następujących sytuacji:

Inicjalizacja
: Z większością urządzeń wirtualnych związana jest taka sama struktura `net_device` jak w przypadku urządzeń rzeczywistych. Często większość wskaźników funkcji znajdujących się w strukturze `net_device` urządzenia wirtualnego zostaje zainicjowana wskazaniem funkcji, które obudowują w mniej lub bardziej skomplikowany sposób funkcje urządzeń rzeczywistych.

Jednak nie ze wszystkimi urządzeniami wirtualnymi są związane struktury `net_device`. Przykładem mogą być urządzenia zastępcze, które są zaimplementowane jako proste etykiety odpowiednich urządzeń rzeczywistych (podpunkcie „Narzędzia konfiguracyjne starej generacji: interfejsy zastępcze" zamieszczony w rozdziale 30.).

Konfiguracja
Do konfiguracji urządzeń wirtualnych często dostarczane są narzędzia działające w przestrzeni użytkownika, zwłaszcza w przypadku konieczności skonfigurowania pól wysokiego poziomu, które występują tylko w przypadku urządzeń wirtualnych i nie mogą zostać skonfigurowane za pomocą standardowych narzędzi takich jak *ifconfig*.

Interfejsy zewnętrzne
Każde urządzenie wirtualne eksportuje zwykle plik lub katalog zawierający kilka plików w systemie plików */proc*. Stopień skomplikowania i szczegółowości informacji zawartych w tych plikach zależy od konkretnego urządzenia wirtualnego. Dla każdego z urządzeń wirtualnych wymienionych w podrozdziale „Urządzenia wirtualne" zawartość eksportowanych plików zostanie przedstawiona w rozdziałach omawiających poszczególne z tych urządzeń. Pliki związane z urządzeniami wirtualnymi są plikami dodatkowymi i nie zastępują plików związanych z urządzeniami rzeczywistymi. Wyjątkiem po raz kolejny są urządzenia zastępcze, które nie posiadają własnych struktur `net_device`.

Wysyłanie
Jeśli związek pomiędzy urządzeniem wirtualnym i urządzeniem rzeczywistym nie jest jak jeden do jednego, to procedura wysyłająca dane musi, obok innych zadań, wybrać rzeczywiste urządzenie, którego użyje do wysłania danych[9]. Ponieważ QoS jest egzekwowany dla każdego urządzenia z osobna, to wielokrotne powiązania pomiędzy urządzeniami wirtualnymi i rzeczywistymi mają wpływ na konfigurację Traffic Control.

Odbiór
Ponieważ urządzenia wirtualne są obiektami programowymi, to nie wymagają interakcji z rzeczywistymi zasobami systemu takimi jak przerwania czy port I/O. Odbierane dane pochodzą bowiem od rzeczywistych urządzeń, które używają tych zasobów. Odbiór pakietów odbywa się w różny sposób dla różnych urządzeń wirtualnych. Na przykład interfejsy 802.1Q rejestrują się jako typ Ethertype i otrzymują tylko te pakiety odebrane przez rzeczywiste urządzenia, które niosą odpowiedni identyfikator protokołu[10]. Natomiast urządzenia mostkujące otrzymują wszystkie pakiety odebrane przez związane z nimi rzeczywiste urządzenia (rozdział 16.).

Powiadomienia zewnętrzne
Powiadomienia o określonych zdarzeniach pochodzące od innych komponentów jądra[11] są interesujące dla urządzeń wirtualnych w takim samym stopniu jak dla urządzeń rzeczywistych. Ponieważ logika urządzeń wirtualnych jest stworzona ponad urządzeniami rzeczywistymi, te ostatnie nie znają jej i nie mogą przekazywać powiadomień do urządzeń wirtualnych. Z tego powodu powiadomienia muszą trafiać bezpośrednio do urządzeń wirtualnych. Weźmy jako przykład Bonding: jeśli przestanie działać jedno z urządzeń rzeczywistych należących do grupy interfejsu wirtualnego, to algorytm dokonujący rozdziału

[9] Więcej szczegółów na temat wysyłania pakietów, a w szczególności funkcji `dev_queue_init` można znaleźć w rozdziale 11.

[10] Demultipleksację ruchu wejściowego na podstawie identyfikatorów protokołów omówiono w rozdziale 13.

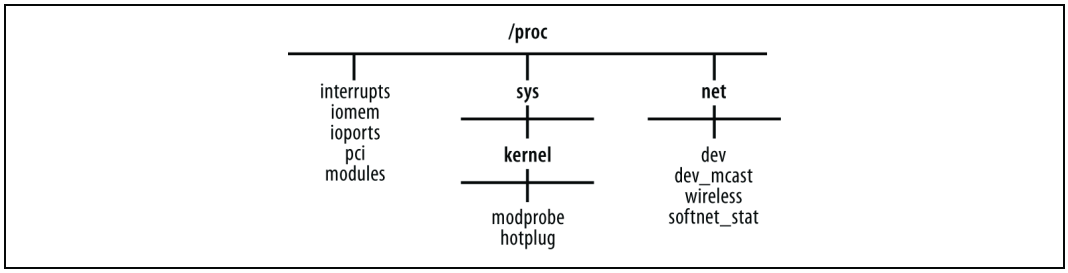
[11] W rozdziale 4. omówiono łańcuchy powiadomień oraz wyjaśniono, dla jakich powiadomień mogą być używane.

ruchu sieciowego pomiędzy interfejsy grupy musi zostać o tym powiadomiony, aby nie wybierał już urządzenia, które nie jest dostępne.

W przeciwieństwie do powiadomień wyzwalanych programowo powiadomienia wyzwalane sprzętowo (np. związane z zarządzaniem zasilaniem urządzeń PCI) nie mogą trafiać do urządzeń wirtualnych, ponieważ urządzenia te nie są związane z żadnym sprzętem.

Strojenie za pośrednictwem systemu plików /proc

Na rysunku 5.5 zostały przedstawione pliki, które mogą być używane albo do strojenia, albo do sprawdzania statusu parametrów konfiguracyjnych związanych z zagadnieniami omawianymi w tym rozdziale.



Rysunek 5.5. Pliki /proc związane z podsystemem routingu

W */proc/sys/kernel* znajdują się pliki *modprobe* i *hotplug*, przy pomocy których można zmieniać ścieżki dostępu i nazwy dwóch programów omówionych wcześniej w podrozdziale „Kod pomocniczy w przestrzeni użytkownika".

Kilka plików w */proc* eksportuje dane pochodzące z wewnętrznych struktur oraz parametry konfiguracyjne, które są przydatne do śledzenia, jakie zasoby zostały przydzielone przez sterowniki urządzeń (wcześniejszy podrozdział „Podstawowe cele inicjalizacji karty sieciowej"). Dla niektórych z tych danych istnieją polecenia wykonywane w przestrzeni użytkownika, pozwalające wyświetlić dane w bardziej przyjaznym formacie. Na przykład *lsmod* wyświetla listę aktualnie załadowanych modułów, używając */proc/modules* jako źródła informacji.

W */proc/net* znajdują się pliki utworzone przez `net_dev_init` za pośrednictwem `dev_proc_init` i `dev_mcast_init` (wcześniejszy podrozdział „Inicjalizacja warstwy obsługi urządzeń: net_dev_ini"):

dev
: Wyświetla dla każdego urządzenia sieciowego zarejestrowanego w jądrze kilka danych statystycznych dotyczących wysyłania i odbierania danych, takich jak na przykład liczba wysłanych i odebranych bajtów, liczba pakietów, liczba błędów itd.

dev_mcast
: Wyświetla dla każdego urządzenia sieciowego zarejestrowanego w jądrze wartości kilku parametrów używanych przez IP multicast.

wireless
: Podobnie jak *dev* dla każdego urządzenia bezprzewodowego wyświetla wartości kilku parametrów zwracane przez funkcję wirtualną `dev->get_wireless_stats`. Zwróćmy uwagę, że funkcja ta zwraca dane tylko w przypadku urządzeń bezprzewodowych, gdyż tylko one tworzą strukturę zawierającą te dane (a */proc/net/wireless* zawiera tylko pliki urządzeń bezprzewodowych).

softnet_stat
: Eksportuje dane statystyczne o przerwaniach programowych używanych przez kod sieciowy. Więcej — rozdział 12.

Istnieją również inne interesujące katalogi, takie jak */proc/drivers*, */proc/bus* i */proc/irq*, których omówienie można znaleźć w książce *Linux Device Drivers*. Co więcej, parametry jądra są stopniowo przemieszczane z */proc* do katalogu */sys*, ale z braku miejsca nie będę omawiać tego nowego systemu.

Funkcje i zmienne występujące w tym rozdziale

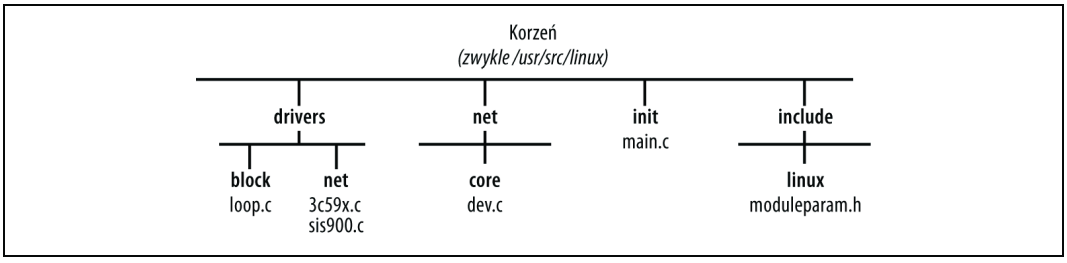
W tabeli 5.1 zostały przedstawione funkcje, makra, zmienne i struktury danych wprowadzone w tym rozdziale.

Tabela 5.1. Funkcje, makra, zmienne i struktury danych związane z inicjalizacją systemu

Nazwa	Opis
Funkcje i makra	
`request_irq` `free_irq`	Rejestrują i zwalniają procedurę obsługi linii IRQ. Rejestracja może być na zasadzie wyłączności lub współdzielenia linii IRQ.
`request_region` `release_region`	Przydzielają i zwalniają porty i pamięć I/O.
`call_usermodehelper`	Wywołuje aplikację pomocniczą działającą w przestrzeni użytkownika.
`module_param`	Makro używane do definiowania parametrów konfiguracji modułów.
`net_dev_init`	Inicjuje element kodu sieciowego podczas uruchamiania systemu.
Zmienne globalne	
`dev_boot_phase`	Znacznik logiczny używany przez tradycyjny kod do wymuszenia wykonania funkcji `net_dev_init` przed zarejestrowaniem sterowników kart sieciowych.
`irq_desc`	Wskaźnik wektora deskryptorów IRQ.
Struktury danych	
`struct` `irq_action`	Każda linia IRQ posiada instancję tej struktury. Wśród innych pól zawiera ona funkcję zwrotną.
`net_device`	Opisuje urządzenie sieciowe.

Pliki i katalogi występujące w tym rozdziale

Na rysunku 5.6 zostały pokazane pliki i katalogi, do których odwoływałem się w tym rozdziale.



Rysunek 5.6. Pliki i katalogi omawiane w tym rozdziale