

Robert Love



Jądro Linuksa

Przewodnik programisty

Wnieś swój wkład w rozwój Linuksa!



Tytuł oryginału: Linux Kernel Development (3rd Edition)

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-246-4273-1

Authorized translation from the English edition, entitled: LINUX KERNEL DEVELOPMENT, Third Edition; ISBN 0672329468; by Robert Love; published by Pearson Education, Inc, publishing as Addison Wesley.

Copyright © 2010 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Polish language edition published by HELION S.A., Copyright © 2014.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/jadlin>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	17
Wstęp	19
Rozdział 1. Jądro systemu Linux — wprowadzenie	25
Historia Uniksa	25
Wprowadzenie do systemu Linux	27
Przegląd systemów operacyjnych	29
Jądro Linuxa a jądro klasycznego systemu uniksowego	31
Oznaczenia wersji jądra Linuxa	34
Społeczność programistów jądra Linuxa	35
Zanim zaczniemy	36
Rozdział 2. Linux — zaczynamy	37
Kod źródłowy jądra	37
Git	37
Instalowanie źródeł jądra z archiwum	38
Aplikowanie łat	39
Struktura katalogów kodu źródłowego jądra	39
Kompilowanie jądra	40
Konfigurowanie jądra	41
Minimalizacja szumu podczas kompilacji	43
Kompilacja na wielu frontach	43
Instalowanie nowego jądra	44
Odmienność jądra	45
Brak implementacji i nagłówek biblioteki standardowej C	45
GNU C	47
Brak mechanizmu ochrony pamięci	49
Niemożność (łatwego) korzystania z operacji zmiennoprzecinkowych	50
Ograniczony co do rozmiaru (i stały) stos	50
Synchronizacja i współbieżność	50
Znaczenie przenośności	51
Podsumowanie	51

Rozdział 3. Zarządzanie procesami	53
Proces	53
Deskryptor procesu i struktura zadania	55
Przydział deskryptora procesu	55
Przechowywanie deskryptora procesu	57
Stan procesu	58
Manipulowanie bieżącym stanem procesu	59
Kontekst procesu	60
Drzewo procesów	60
Tworzenie procesu	62
Kopiowanie przy zapisie	62
Rozwidlanie procesu	63
Wywołanie vfork()	64
Wątki w systemie Linux	65
Tworzenie wątków	66
Wątki jądra	67
Zakończenie procesu	68
Usuwanie deskryptora procesu	70
Problem zadań osieroconych	71
Podsumowanie	73
Rozdział 4. Szeregowanie zadań	75
Wielozadaniowość	75
Planista zadań w Linuksie	77
Strategia postępowania	77
Procesy ograniczone wejściem-wyjściem	
a procesy ograniczone procesorem	78
Priorytet procesu	79
Kwant czasu procesora	80
Strategia szeregowania w działaniu	81
Algorytm szeregowania zadań w Linuksie	82
Klasy szeregowania	82
Szeregowanie procesów w systemach klasy Unix	82
Sprawiedliwe szeregowanie zadań	85
Implementacja szeregowania zadań w Linuksie	87
Zliczanie czasu wykonania	87
Wybór procesu	89
Punkt wejścia do planisty CFS	94
Zawieszanie i wybudzanie procesów	95
Wywłaszczanie i przelączenie kontekstu	99
Wywłaszczanie procesu użytkownika	100
Wywłaszczanie jądra	101
Szeregowanie czasu rzeczywistego	102

Wywołania systemowe związane z szeregowaniem	103
Wywołania wpływające na strategię szeregowania i wartości priorytetów	104
Wywołania systemowe sterujące kojarzeniem procesów z procesorami	105
Odstąpienie procesora	105
Podsumowanie	106
Rozdział 5. Wywołania systemowe	107
Komunikacja z jądrem systemu	107
API, POSIX i biblioteka C	108
Wywołania systemowe	109
Numery wywołań systemowych	111
Wydajność wywołania systemowego	111
Procedura obsługi wywołań systemowych	111
Oznaczenie właściwego wywołania systemowego	112
Przekazywanie argumentów	113
Implementacja wywołania systemowego	113
Implementowanie wywołań systemowych	114
Weryfikacja argumentów	114
Kontekst wywołania systemowego	118
Wiązanie wywołania systemowego	119
Inicjowanie wywołania systemowego z przestrzeni użytkownika	120
Sześć powodów, aby nie implementować wywołań systemowych	121
Podsumowanie	122
Rozdział 6. Struktury danych jądra	123
Listy	123
Listy jedno- i dwukierunkowe	124
Listy cykliczne	124
Implementacja list w jądrze Linuksa	126
Operacje na listach	128
Przeglądanie list	131
Inne operacje na listach	134
Kolejki	135
kfifo	135
Tworzenie kolejki	136
Zakolejkowanie	137
Wyciąganie danych	137
Określanie rozmiaru kolejki	137
Zerowanie i usuwanie kolejki	138
Przykład użycia kolejki kfifo	138

Mapy	139
Inicjalizowanie mapy idr	140
Przydział nowego identyfikatora UID	140
Odwzorowanie UID na wskaźnik	141
Usuwanie UID	142
Usuwanie idr	142
Drzewa binarne	143
Drzewa BST	143
Zrównoważone drzewa BST	144
Drzewa czerwono-czarne	145
rbtree	146
Kiedy i jakiej struktury użyć?	148
Złożoność obliczeniowa	149
Algorytmy	149
Notacja O	149
Notacja theta	150
Złożoność czasowa	150
Podsumowanie	151
Rozdział 7. Przerwania i procedury obsługi przerwania	153
Przerwania	154
Procedury obsługi przerwania	155
Połówki górne i dolne	156
Rejestrowanie procedury obsługi przerwania	157
Znaczniki procedury obsługi przerwania	158
Przykładowe przerwanie	159
Zwalnianie procedury obsługi przerwania	160
Tworzenie procedury obsługi przerwania	160
Procedury obsługi przerwania współużytkowanych	162
Prawdziwa procedura obsługi przerwania	163
Kontekst przerwania	165
Implementacja obsługi przerwania	166
/proc/interrupts	168
Kontrola przerwania	169
Wyłączanie i włączanie przerwania	170
Blokowanie konkretnej linii przerwania	172
Stan systemu przerwania	173
Podsumowanie	174
Rozdział 8. Dolne połówki i czynności odroczone	175
Połówki dolne	176
Po co dolne połówki?	177
Świat dolnych połówek	178

Przerwania programowe	181
Implementacja przerwania programowych	181
Korzystanie z przerwania programowych	183
Tasklety	186
Implementacja taskletów	186
Korzystanie z taskletów	189
Wątek jądra ksoftirqd	191
Dawny mechanizm BH	193
Kolejki prac	194
Implementacja kolejek prac	195
Korzystanie z kolejek prac	199
Dawny mechanizm kolejkowania zadań	201
Jak wybrać implementację dolnej połówki?	202
Blokowanie pomiędzy dolnymi połówkami	204
Wyłączanie dolnych połówek	204
Podsumowanie	206
Rozdział 9. Wprowadzenie do synchronizacji jądra	207
Sekcje krytyczne i przeplot operacji	208
Po co ta ochrona?	208
Zmienna globalna	210
Blokowanie	211
Źródła współbieżności	213
Co wymaga zabezpieczenia?	215
Zakleszczenia	216
Rywalizacja a skalowalność	218
Podsumowanie	220
Rozdział 10. Metody synchronizacji jądra	221
Operacje niepodzielne	221
Niepodzielne operacje na liczbach całkowitych	222
64-bitowe operacje niepodzielne	226
Niepodzielne operacje bitowe	228
Rygle pętlowe	230
Interfejs rygli pętlowych	231
Inne metody blokowania ryglami pętlowymi	234
Rygle pętlowe a dolne połówki	234
Rygle pętlowe R-W	235
Semafony	238
Semafony binarne i zliczające	239
Tworzenie i inicjalizowanie semaforów	240
Korzystanie z semaforów	240
Semafony R-W	241

Muteksy	243
Muteksy czy semafor?	245
Muteksy czy rygle pętlowe?	245
Zmienne sygnałowe	245
Blokada BKL (Big Kernel Lock)	246
Blokady sekwencyjne	247
Blokowanie wyłączenia	249
Bariery pamięciowe	251
Podsumowanie	255
Rozdział 11. Liczniki i zarządzanie czasem	257
Czas z punktu widzenia jądra	258
Częstotliwość taktowania — HZ	259
Optymalna wartość HZ	260
Zalety wysokich wartości HZ	261
Wady wysokich wartości HZ	262
Chwilki	263
Wewnętrzna reprezentacja zmiennej jiffies	264
Zawijanie zmiennej jiffies	266
HZ a przestrzeń użytkownika	267
Zegary i liczniki sprzętowe	268
Zegar czasu rzeczywistego	268
Zegar systemowy	269
Procedura obsługi przerwania zegarowego	269
Data i godzina	272
Liczniki	274
Korzystanie z liczników	275
Liczniki i sytuacje hazardowe	277
Implementacja licznika	277
Opóźnianie wykonania	278
Oczekiwanie w pętli aktywnej	278
Krótkie opóźnienia	280
Funkcja schedule_timeout()	281
Implementacja funkcji schedule_timeout()	282
Podsumowanie	284
Rozdział 12. Zarządzanie pamięcią	285
Strony	285
Strefy	287
Pozyskiwanie stron pamięci	290
Pozyskiwanie czystych stron pamięci	291
Zwalnianie stron	291
Funkcja kmalloc()	292
Znaczniki gfp_mask	293
Funkcja kfree()	298

Funkcja <code>vmalloc()</code>	299
Alokator plastrowy	301
Zadania alokatora plastrowego	302
Interfejs alokatora plastrowego	305
Przydział z pamięci podręcznej	307
Statyczne przydziały na stosie	309
Jednostronicowy stos procesów jądra	309
Ostrożnie ze stosem	310
Odwzorowanie pamięci wysokiej	310
Odwzorowanie trwałe	311
Odwzorowania czasowe	311
Przydziały lokalne względem procesora	312
Interfejs percpu	313
Statyczne dane lokalne względem procesora	313
Dynamiczne dane lokalne względem procesora	314
Przydatność danych lokalnych względem procesora	315
Wybór metody przydziału pamięci w kodzie jądra	316
Podsumowanie	317
Rozdział 13. Wirtualny system plików	319
Wspólny interfejs systemu plików	320
Warstwa abstrakcji systemu plików	320
Uniksowy system plików	322
Obiekty VFS i ich struktury danych	323
Obiekt bloku głównego	325
Operacje bloku głównego	326
Obiekt i-węzła	329
Operacje i-węzła	331
Obiekt wpisu katalogowego	334
Stan wpisu katalogowego	335
Bufor wpisów katalogowych	336
Operacje na wpisach katalogowych	337
Obiekt pliku	339
Operacje na plikach	340
Struktury danych systemu plików	343
Struktury danych procesu	345
Podsumowanie	347
Rozdział 14. Blokowe urządzenia wejścia-wyjścia	349
Anatomia urządzenia blokowego	350
Bufory i nagłówki buforów	351
Struktura bio	354
Wektory wejścia-wyjścia	355
Stare a nowe	357

Kolejki zleceń	357
Zawiadywanie operacjami wejścia-wyjścia	358
Zadania planisty operacji wejścia-wyjścia	359
Winda Linusa	360
Terminowy planista operacji wejścia-wyjścia	361
Przewidujący planista operacji wejścia-wyjścia	363
Sprawiedliwy planista kolejkowania operacji wejścia-wyjścia	364
Nieingerujący planista operacji wejścia-wyjścia	365
Wybór planisty operacji wejścia-wyjścia	366
Podsumowanie	366
Rozdział 15. Przestrzeń adresowa procesu	367
Przestrzeń adresowa	367
Deskryptor pamięci	369
Przydział deskryptora pamięci	371
Zwalnianie deskryptora pamięci	371
Struktura mm_struct i wątki jądra	372
Obszary pamięci wirtualnej	372
Znaczniki VMA	374
Operacje VMA	375
Obszary pamięci na listach i w drzewach	376
Obszary pamięci w praktyce	377
Manipulowanie obszarami pamięci	379
Funkcja find_vma()	379
Funkcja find_vma_prev()	381
Funkcja find_vma_intersection()	381
Tworzenie interwału adresów	
— wywołania mmap() i do_mmap()	381
Usuwanie interwału adresów	
— wywołania munmap() i do_munmap()	383
Tablice stron	384
Podsumowanie	386
Rozdział 16. Pamięć podręczna stron i opóźniony zapis stron w tle	387
Modele pamięci podręcznych	387
Buforowanie zapisów	388
Eksmisja z pamięci podręcznej	389
Pamięć podręczna stron w Linuksie	391
Obiekt address_space	391
Operacje na obiektach address_space	392
Drzewo pozycyjne	395
Tablica skrótów stron	395
Pamięć podręczna buforów	396

Wątki zapisu w tle	396
Tryb laptopowy	398
Rys historyczny — bdflush, kupdated i pdflush	399
Eliminowanie obciążenia operacjami wejścia-wyjścia	400
Podsumowanie	401
Rozdział 17. Sterowniki i moduły	403
Typy urządzeń	403
Moduły	404
Hello, world!	405
Kompilowanie modułów	406
Instalowanie modułów	409
Generowanie zależności międzymodułowych	409
Ładowanie modułów	410
Zarządzanie opcjami konfiguracji jądra	411
Parametry modułów	413
Symbole eksportowane	415
Model sterowników	416
Obiekty jądra	417
Typy obiektów jądra	418
Grupy obiektów jądra	419
Powiązania pomiędzy obiektami, typami i grupami obiektów jądra	419
Zarządzanie i operowanie obiektami jądra	420
Zliczanie odwołań	421
System plików sysfs	423
Dodawanie i usuwanie obiektów jądra w sysfs	426
Dodawanie plików do sysfs	427
Kernel Event Layer	430
Podsumowanie	431
Rozdział 18. Diagnostyka błędów jądra	433
Od czego zacząć?	433
Błędy w jądrze	434
Diagnostyka komunikatami	435
Niezawodność printk()	435
Poziomy diagnostyczne	436
Bufor komunikatów	437
Demony syslogd i klogd	438
printf(), printk() — łatwo o pomyłkę	438
Błąd oops	438
Polecenie ksymbols	440
kallsyms	441
Opcje diagnostyczne jądra	441

Asercje i wypisywanie informacji o błędach	442
Funkcja Magic SysRq Key	443
Saga debugera jądra	444
gdb	445
kgdb	446
Stymulowanie i sondowanie systemu	446
Uzależnianie wykonania kodu od identyfikatora UID	446
Korzystanie ze zmiennych warunkowych	447
Korzystanie ze statystyk	447
Ograniczanie częstotliwości i liczby komunikatów diagnostycznych	448
Szukanie winowajcy — wyszukiwanie binarne	449
Binarne wyszukiwanie wersji za pomocą Gita	450
Koledzy — kiedy wszystko inne zawiedzie	451
Podsumowanie	451
Rozdział 19. Przenośność	453
Przenośne systemy operacyjne	453
Historia przenośności systemu Linux	455
Rozmiar słowa i typy danych	456
Typy nieprzejrzyste	459
Typy specjalne	459
Typy o zadanych rozmiarach	460
Znak typu char	461
Wyrównanie danych	462
Unikanie problemów wyrównywania	462
Wyrównanie typów niestandardowych	463
Dopełnienie struktury	463
Wzajemny porządek bajtów	465
Pomiar upływu czasu	467
Rozmiar strony	468
Kolejność wykonywania instrukcji	469
Tryb SMP, wyłączenie jądra i pamięć wysoka	469
Podsumowanie	470
Rozdział 20. Kodowanie, łąty i społeczność	471
Spółeczność	471
Obowiązujący styl kodowania	472
Wcięcia	472
Instrukcje switch	473
Odstępy	473
Nawiasy klamrowe	475
Długość wiersza kodu	476
Nazewnictwo	476

Funkcje	477
Komentarze	477
Definicje typów	478
Korzystanie z gotowców	478
Unikanie definicji ifdef w ciele funkcji	478
Inicjalizacja struktur	479
Poprawianie złego stylu	479
Łańcuch poleceń	480
Przesyłanie raportów o błędach	480
Łaty	481
Generowanie łat	481
Generowanie łat za pomocą Gita	482
Rozsyłanie łat	483
Podsumowanie	484
Dodatek A Bibliografia	485
Książki o projektowaniu systemów operacyjnych	485
Książki o jądrze systemu Unix	486
Książki o jądrze systemu Linux	486
Książki o jądrach innych systemów operacyjnych	487
Książki o interfejsie programowym Uniksa	487
Książki o programowaniu w języku C	487
Inne książki	488
Witryny WWW	488
Skorowidz	489

Diagnostyka błędów jądra

Jedną z podstawowych miar odróżniających programowanie jądra od programowania przestrzeni użytkownika jest stopień trudności diagnostyki błędów. Diagnostyka jądra nie jest łatwa, zwłaszcza w porównaniu z przestrzenią użytkownika. Problem komplikuje jeszcze wysokość stawki — błąd w jądrze może załamać cały system komputerowy.

Rosnąca umiejętność diagnozowania błędów w jądrze — a co za tym idzie, coraz większa swoboda programowania jądra — jest w dużej mierze funkcją doświadczenia i stopnia zrozumienia systemu operacyjnego. Podziałać mogą zapewne również zaklęcia i czary, ale do udanej diagnostyki jądra konieczna jest jego pełna znajomość. Niniejszy rozdział poświęcony będzie właśnie zagadnieniom związanym z diagnostyką błędów jądra.

Od czego zacząć?

Diagnostyka błędów w jądrze bywa uciążliwym i żmudnym procesem. Niektóre z błędów całymi miesiącami myliły tropy, choć śledzone były przez wielu programistów. Na szczęście na dosłownie każdy tak pracochłonny błąd przypada wiele błędów bardziej oczywistych, z równie oczywistymi poprawkami. Przy odrobinie szczęścia wszystkie błędy, jakie popełnimy, będą błędami prostymi. Tego jednak przed rozpoczęciem poszukiwań nie będzie wiadomo na pewno. Dlatego warto zaopatrzyć się w:

- Błąd. To może brzmieć śmiesznie, ale potrzebny będzie dobrze zdefiniowany i bardzo konkretny błąd. Dobrze, jeżeli jest to błąd powtarzalny, który można sprowokować w określonych warunkach. Niestety, błędy rzadko bywają stabilne i powtarzalne.
- Wersję jądra, w której występuje rzeczony błąd (na przykład najnowszą wersję, bo inaczej nikt nie zawracałby sobie głowy błędem). Najlepiej, jeżeli uda się wytypować najwcześniejszą wersję jądra, w której pojawił się błąd. W rozdziale będzie omawiana technika polowania na popsutą wersję jądra.
- Znajomość odpowiedniej części jądra albo szczęście. Diagnozowanie błędów jądra jest śledztwem, które daje tym lepsze wyniki, im lepsza jest znajomość otoczenia.

Większość bieżącego rozdziału będzie poświęcona technikom reprodukcji błędów. Skuteczność w ich poprawianiu jest bowiem wypadkową umiejętności prowokowania wystąpienia błędu. Niemożność sprowokowania błędu ogranicza możliwości jego usunięcia, zmuszając do jedynie koncepcyjnego określenia problemu i odnalezienia odpowiadającej mu luki w kodzie źródłowym. Często zdarza się zresztą i tak, ale szansa powodzenia jest znacznie większa przy możliwości obserwacji błędu.

Istnienie błędu, którego wpływu na system nie da się ujawnić, może być nieco wątpliwe. W programach przestrzeni użytkownika błędy są najczęściej bardziej dokuczliwe — na przykład „wywołanie funkcji `bla()` w takim a takim kroku pętli powoduje awaryjne zamknięcie programu”. W kodzie jądra sprawy są zazwyczaj znacznie bardziej zagmatwane. Interakcje pomiędzy jądrem, przestrzenią użytkownika i sprzętem potrafią być bardzo delikatne. Niekiedy będące przyczyną błędu sytuacje hazardowe pojawiają się raz na milion iteracji jakiegoś algorytmu. Słabo zaprojektowany albo nawet źle skompilowany kod może dawać w niektórych systemach oczekiwane rezultaty, pogarszając wydajność systemów o innych konfiguracjach. Niejednokrotnie zdarza się wywołać błąd jedynie w określonej konfiguracji jądra, na pewnej przypadkowej maszynie, przy obciążeniu systemu nietypowymi operacjami itd. Dlatego kluczowe znaczenie ma ilość informacji o środowisku, w jakim wystąpił błąd. Zazwyczaj jednak możliwość powtórnego sprowokowania błędu to więcej niż połowa sukcesu.

Błędy w jądrze

Błędy w jądrze bywają bardzo różnorodne. Zdarzają się one z niezliczonych powodów i objawiają się w równie niezliczonych formach. Błędy od jawnie błędnego kodu (na przykład niezachowania poprawnej wartości w odpowiednim miejscu), przez błędy synchronizacji (wynikające z braku właściwego blokowania dostępu do współdzielonej zmiennej), po nieprawidłowe zarządzanie sprzętem (wysyłanie nieprawidłowej wartości do nieodpowiedniego rejestru sterującego) objawiają się pod wszelkimi postaciami, od niezadowolającej wydajności systemu po jego niewłaściwe działanie w skrajnych przypadkach powodujące utratę danych albo zawieszanie systemu.

Błąd w postaci objawiającej się użytkownikowi nieraz może dzielić od jego przyczyny tkwiącej w kodzie jądra długi łańcuch zdarzeń. Na przykład współużytkowana struktura pozbawiona licznika odwołań może prowokować sytuacje hazardowe. Wobec braku poprawnego zliczania odwołań może dojść do zwolnienia przez jeden proces struktury używanej jeszcze przez inny proces. Ów inny proces może próbować odwołać się do zwolnionej już struktury za pośrednictwem przetrzymywanego lokalnie, nieprawidłowego już wskaźnika. Może to zaowocować wyłusaniem wskaźnika pustego, odczytem „śmieci” czy też czymś zupełnie niewinnym (kiedy obszar zwolnionej struktury nie został jeszcze niczym nadpisany). Wyłuskanie wskaźnika pustego daje w efekcie błąd „oops”, podczas gdy odwołanie do przypadkowych danych („śmieci”) w pamięci może prowadzić do naruszenia spójności danych (a w jej wyniku do niewłaściwego zachowania

programu, a w dalekiej konsekwencji do błędu „oops”). Użytkownik zgłasza więc albo błąd „oops”, albo wyłącznie niepoprawne działanie systemu. Programista jądra musi wysledzić przyczynę błędu, wykrzyć, że nastąpił dostęp do zwolnionych wcześniej danych, że wcześniej niewłaściwie współużytkowana była zawierająca te dane struktura i wreszcie zaaplikować łatę w postaci poprawnego zliczania odwołań do współużytkowanej struktury (i być może jakiegoś rygla chroniącego ją przed współbieżnym dostępem).

Diagnostyka błędów jądra może sprawiać wrażenie zadania dla magików, ale w rzeczywistości jądro systemu nie różni się niczym od innych dużych projektów programowych. Co prawda w jądrze należy brać pod uwagę elementy specyficzne, takie jak ograniczenia czasowe wykonania kodu czy sytuacje hazardowe będące konsekwencją działania wielu wątków w ramach jądra.

Diagnostyka komunikatami

Funkcja jądra `printk()` działa niemal identycznie jak podobnie nazwana funkcja biblioteczna języka C, `printf()`. Podobieństwo działania jest tak duże, że jak na razie w książce nie wystąpiło zastosowanie funkcji `printk()` znacząco odbiegające od zastosowań `printf()`. Tak więc `printk()` to nazwa funkcji jądra generującej sformatowany wydruk znakowy. Nie jest to jednak tak do końca zwykła funkcja formatująca.

Niezawodność `printk()`

Jedną z tych właściwości wywołania `printk()`, które szybko przyjmuje się za oczywistość, jest jej *niezawodność* i *wszechstronność*. Funkcja `printk()` daje się wywołać z dowolnego miejsca jądra w dowolnym momencie jego wykonania. Można ją wywołać z kontekstu procesu i kontekstu przerwania. Można ją wywoływać, przetrzymując równocześnie blokadę dowolnego rodzaju. Można też wywoływać ją współbieżnie na wielu procesorach, przy czym wywołujący nie musi pozyskiwać żadnej blokady.

To funkcja naprawdę nie do zdarcia. To bardzo ważne, ponieważ przydatność funkcji `printk()` wynika w dużej mierze właśnie z możliwości jej wywołania dosłownie zewsząd i z gwarancji jej działania w każdych warunkach.

Istnieje jednak luka we wszechstronności wywołania `printk()`. Nie da się jej bowiem zastosować przed określonym momentem rozruchu jądra, a ściślej mówiąc, przed momentem zainicjowania konsoli. To oczywiste, bo gdzie niby miałyby być wcześniej kierowane komunikaty? Zwykle nie jest to problemem, chyba że chodzi o diagnozowanie bardzo wczesnej fazy rozruchu (na przykład diagnostykę błędów działania funkcji `setup_arch()` odpowiedzialnej za operacje inicjalizacji zależne od architektury systemu). Taka diagnostyka to prawdziwe wyzwanie — a brak jakichkolwiek narzędzi wyprowadzania komunikatów dodatkowo je komplikuje.

Nawet diagnostyka wczesnych faz rozruchu daje jednak pewne możliwości. Niektórzy programiści wykorzystują do wyprowadzania komunikatów sprzęt, który działa zawsze,

na przykład port szeregowy. Nie jest to jednak przyjemna zabawa. Rozwiązaniem jest wariant wywołania `printk()` przystosowany do wyprowadzania komunikatów na konsolę we wczesnych fazach rozruchu — `early_printk()`. Działa ona identycznie jak `printk()` — funkcje te różnią się wyłącznie nazwami i możliwością operowania w pierwszych fazach rozruchu. Nie jest to jednak rozwiązanie przenośne, gdyż funkcja `early_printk()` nie jest implementowana we wszystkich architekturach. Dobrze, żeby diagnozowana architektura ją posiadała — a większość (z architekturą x86 na czele) posiada.

Podsumowując, o ile nie zachodzi potrzeba sygnalizacji na bardzo wczesnym etapie rozruchu, można polegać na wszechstronności i niezawodności funkcji `printk()`.

Poziomy diagnostyczne

Główna różnica pomiędzy wywołaniami `printk()` i `printf()` tkwi w zdolności tej pierwszej do określania tak zwanego *poziomu diagnostycznego* (ang. *loglevel*). Jądro określa poziom diagnostyczny, decydując tym samym, czy komunikat powinien zostać wyświetlony na konsoli czy też przekierowany gdzieś indziej. Jądro wyświetla na konsoli wszystkie komunikaty o poziomie diagnostycznym ustalonym na poziomie niższym od pewnego progu.

Poziom diagnostyczny określa się następująco:

```
printk(KERN_WARNING "To jest ostrzezenie!\n");
printk(KERN_DEBUG "To jest komunikat diagnostyczny!\n");
printk("Brak określenia poziomu diagnostycznego!\n");
```

Ciągi `KERN_WARNING` i `KERN_DEBUG` to po prostu definicje występujące w pliku `<linux/printk.h>`. Rozwijane są one odpowiednio do ciągów `<4>` i `<7>` i dołączane na początek komunikatu przekazywanego do `printk()`. Jądro na podstawie tych przedrostków decyduje o tym, które z komunikatów powinny być wyświetlane na konsoli, porównując określony poziom diagnostyczny z bieżącym poziomem diagnostycznym konsoli określonym parametrem `console_loglevel`. Pełny wykaz dostępnych poziomów diagnostycznych funkcji `printk()` został przedstawiony w tabeli 18.1.

Tabela 18.1. Poziomy diagnostyczne funkcji `printk()`

Poziom diagnostyczny	Opis
<code>KERN_EMERG</code>	Sytuacja awaryjna.
<code>KERN_ALERT</code>	Problem wymagający natychmiastowej interwencji.
<code>KERN_CRIT</code>	Sytuacja krytyczna.
<code>KERN_ERR</code>	Błąd.
<code>KERN_WARNING</code>	Ostrzeżenie.
<code>KERN_NOTICE</code>	Sytuacja normalna, ale warta odnotowania.
<code>KERN_INFO</code>	Komunikat informacyjny.
<code>KERN_DEBUG</code>	Komunikat diagnostyczny — najprawdopodobniej zbyteczny.

W przypadku nieokreślenia poziomu rejestrowania jego wartość jest przyjmowana przez domniemanie jako `DEFAULT_MESSAGE_LOGLEVEL`, która to stała jest z kolei rozwijana do stałej `KERN_WARNING`. Jednak z uwagi na to, że ustawienie domyślne może w kolejnych wersjach jądra ulec zmianie, warto jawnie opatrywać komunikaty stałymi poziomów rejestrowania.

Najwyższy poziom rejestrowania definiuje stała `KERN_EMERG` rozwijana do literału znakowego `<0>`. Poziom najniższy określa zaś stała `KERN_DEBUG` rozwijana do ciągu `<7>`. Na przykład po zakończeniu fazy przetwarzania pliku kodu źródłowego przez preprocesor zamieszczone wcześniej wywołania będą miały postać:

```
printk("<4>To jest ostrzezenie!\n");
printk("<7>To jest komunikat diagnostyczny!\n");
printk("<4>Brak okreslenia poziomu diagnostycznego!\n");
```

Strategia przyjmowania poziomów rejestrowania w różnych sytuacjach leży całkowicie w gestii programisty. Rzecz jasna, zwykle komunikaty należałoby opatrywać odpowiednim dla nich poziomem rejestrowania. Ale już komunikaty, którymi szpikuje się kod w poszukiwaniu źródła problemu — wypada się do tego przyznać, wszyscy tak robią — można opatrywać dowolnym poziomem rejestrowania. Jedną z możliwości jest pozostawienie w spokoju domyślnego poziomu rejestrowania konsoli i generowania wszystkich komunikatów diagnostycznych z poziomem `KERN_CRIT` lub wyższym. Można też generować komunikaty na poziomie `KERN_DEBUG` i w zamian zmodyfikować poziom rejestrowania konsoli, tak aby komunikaty były widoczne. Każda z metod ma swoje wady i zalety. Decyzję pozostawiam Czytelnikowi.

Bufor komunikatów

Komunikaty jądra umieszczone są w cyklicznym buforze o rozmiarze `LOG_BUF_LEN`. Rozmiar ten daje się konfigurować na etapie kompilacji przez ustawienie opcji `CONFIG_LOG_BUF_SHIFT`. Domyślny rozmiar bufora dla komputerów jednoprocessorowych to 16 kB. Innymi słowy, jądro może przechowywać do 16 kB komunikatów jądra. Jeżeli kolejka komunikatów jest zapełniona i dojdzie do kolejnego wywołania `printk()`, nowy komunikat nadpisze najstarszy z komunikatów w buforze. Bufor komunikatów nosi nazwę *cyklicznego*, ponieważ jego zapis i odczyt następują w sposób cykliczny.

Korzystanie z bufora cyklicznego ma szereg zalet. Z racji dużej łatwości równoczesnego zapisywania i odczytywania z takiego bufora funkcja `printk()` może być wywoływana nawet w poziomie kontekstu. Co więcej, cykliczność upraszcza zarządzanie buforem. W obliczu zbyt dużej liczby komunikatów najstarsze komunikaty są po prostu zastępowane nowymi. Jeżeli zdarzy się problem, w wyniku którego wygenerowana zostanie większa liczba komunikatów, cykliczność obsługi bufora wyeliminuje część z nich, nie dopuszczając do zwiększenia zapotrzebowania na pamięć. Jedyną wadą takiego rozwiązania jest prawdopodobieństwo utraty komunikatów, ale jest to niewielka cena za wszechstronność bufora.

Demony `syslogd` i `klogd`

W klasycznym systemie Linux za pobieranie komunikatów jądra z bufora odpowiedzialny jest demon przestrzeni użytkownika o nazwie `klogd` — demon ten wypełnia pobieranymi komunikatami systemowy plik dziennika, korzystając przy tym z pomocy demona `syslogd`. W celu odczytu rejestru komunikatów `klogd` może odczytywać plik `/proc/kmsg` bądź korzystać z wywołania systemowego `syslog()`. Domyślnie wykorzystuje jednak system plików `/proc`. Niezależnie od sposobu odczytywania komunikatów `klogd` ulega zawieszeniu aż do momentu pojawienia się nowych komunikatów. W obliczu zgłoszenia nowych komunikatów demon jest pobudzany, odczytuje nowe komunikaty i przetwarza je. Domyślnie przetwarzanie to polega na przekazaniu ich do demona `syslogd`.

Demon `syslogd` dołącza otrzymywane komunikaty do pliku, którym przez domniemanie jest plik `/var/log/messages`. Działanie demona da się konfigurować za pośrednictwem pliku `/etc/syslog.conf`.

Za pośrednictwem demona `klogd` można zmieniać poziom rejestrowania konsoli — wystarczy w wywołaniu demona określić argument `-c`.

`printf()`, `printk()` — łatwo o pomyłkę

Każdemu początkującemu programiście jądra zdarza się mylić wywołania `printk()` z wywołaniami `printf()`. To całkiem naturalne, ponieważ nie sposób zignorować lata doświadczeń i nawyku wykorzystywania funkcji `printf()` w programach przestrzeni użytkownika. Na szczęście takie pomyłki są szybko wychwytywane, jako że powodują przy kompilacji zalew komunikatami z protestami ze strony konsolidatora.

Pewnego dnia być może zdarzy się Czytelnikowi zastosować w wyniku wyniesionych z programowania jądra nawyków wywołanie `printk()` w programie przestrzeni użytkownika. Wtedy będzie on mógł o sobie powiedzieć, że jest prawdziwym hakerem jądra.

Błąd `oops`

Błąd `oops` to normalny sposób powiadamiania użytkownika o nieprawidłowościach działania jądra. Jako że jądro to nadzorca całego systemu, nie ma możliwości samodzielnego usunięcia usterki czy wykonania samounicestwienia, tak jak unicestwia się błędne procesy przestrzeni użytkownika. Jądro w takich sytuacjach zgłasza błąd `oops`. Polega to na wyświetleniu na konsoli komunikatu o błędzie wraz z zawartością rejestrów i śladem wykonania (ang. *back trace*). Błąd jądra jest trudny do obejścia, więc jądro w celu obsłużenia błędu musi się niezłe napracować. Niekiedy po zakończeniu obsługi pojawia się niespójność jądra. Jądro w momencie pojawienia się błędu mogło być na przykład w trakcie przetwarzania istotnych danych. Mogło przetrzymywać blokadę albo prowadzić komunikację ze sprzętem. W obliczu błędu konieczne jest natomiast ostrożne wycofanie się od poprzedniego kontekstu i w miarę możliwości przywrócenie kontroli nad systemem. W wielu przypadkach powrót taki jest niemożliwy. Jeżeli błąd wystąpi w kontekście przerwania, jądro nie może

kontynuować działania i „panikuje”. Błąd „paniczny” (ang. *panic error*) powoduje zaś trwale unieruchomienie systemu. Również pojawienie się błędu oops w trakcie wykonywania zadania jałowego (o numerze pid równym zero) bądź zadania `init` (pid równy jeden) oznacza błąd paniczny — jądro nie może kontynuować działania bez tych dwóch ważnych procesów. Jedynie wystąpienie błędu oops w kontekście jednego ze zwykłych procesów użytkownika daje możliwość unicestwienia tego procesu i kontynuowania działania reszty systemu.

Przyczyny błędu oops mogą być rozmaite, z nieuprawnionym dostępem do pamięci bądź próbą wykonania niedozwolonej instrukcji włącznie. Programista jądra jest skazany na diagnostykę takich błędów (nie mówiąc o tym, że sam przyczynia się do ich powstawania).

Poniższy wydruk to komunikat towarzyszący błędowi oops zaobserwowanemu w komputerze PPC w procedurze obsługi zegara karty sieciowej:

```
Oops: Exception in kernel mode, sig: 4
Unable to handle kernel NULL pointer dereference at virtual address 00000001

NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905d10 TRAP: 0700
Not tainted
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall: 120
GPR00: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F70000 00000032 FFFFFFFF C3F7C1C0
Call trace:
[c013ab30] tulip_timer+0x128/0x1c4
[c0020744] run_timer_softirq+0x10c/0x164
[c001b864] do_softirq+0x88/0x104
[c0007e80] timer_interrupt+0x284/0x298
[c00033c4] ret_from_except+0x0/0x34
[c0007b84] default_idle+0x20/0x60
[c0007bf8] cpu_idle+0x34/0x38
[c0003ae8] rest_init+0x24/0x34
```

Użytkownicy komputerów klasy PC mogą dziwić się znaczną liczbą rejestrów (aż 32!). Błędy oops w znacznie popularniejszych systemach klasy x86-32 generują dużo krótszy wydruk zawartości rejestrów. Najważniejsze informacje są jednak w obu architekturach identyczne — liczy się zawartość rejestrów i ślad wykonania.

Ślad wykonania to łańcuch wywołania funkcji prowadzący do błędu. W prezentowanym przypadku ślad zdradza dokładnie przebieg wykonania aż do momentu wystąpienia błędu — system realizował proces jałowy w pętli `cpu_idle()`, w której wywoływana jest funkcja `default_idle()`. Zgłoszenie przerwania zegarowego spowodowało zainicjowanie obsługi liczników. Uruchomiona wtedy procedura obsługi takiego licznika dla karty sieciowej `tulip_timer()` wykonała w ramach przetwarzania wyłuskanie wskaźnika o wartości NULL. Na podstawie podanych na wydruku offsetów (tutaj `0x128/0x1c4`) można wręcz wytypować konkretny wiersz programu, którego wykonanie spowodowało błąd.

Równie przydatna jak ślad wykonania może być zawartość rejestrów. Dysponując kopią problematycznej funkcji w postaci kodu assemblerowego, można na podstawie zawartości rejestrów odtworzyć dokładnie stan jej wykonania w momencie wystąpienia błędu. Obecność w jednym z rejestrów nieoczekiwanej wartości może często rzucić światło na przyczynę błędu. W analizowanym przypadku widać, które z rejestrów zawierają wartości NULL (zera), i na tej podstawie można wytypować argument wywołania bądź zmienną lokalną funkcji, która przyjęła niewłaściwą wartość. Dla sytuacji podobnych do analizowanej typowym źródłem błędu jest sytuacja hazardowa — w tym konkretnym przypadku chodzi o rywalizację w dostępie do zasobu pomiędzy obsługą licznika a resztą kodu obsługi karty sieciowej. Diagnostyka sytuacji hazardowych jest zresztą zawsze problematyczna.

Polecenie ksymoops

Prezentowany wcześniej wydruk generowany w ramach obsługi błędu oops nosi nazwę *zdekodowanego*, ponieważ adresy pamięci zostały na wydruku przetłumaczone na nazwy funkcji rezydujących pod tymi adresami. Wersja niezdekodowana tego samego błędu wyglądałaby następująco:

```
NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905d10 TRAP: 0700
Not tainted
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last sysca11: 120
GPR00: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F70000 00000032 FFFFFFFF C3F7C1C0
Ca11 trace: [c013ab30] [c0020744] [c001b864] [c0007e80] [c00033c4]
[c0007b84] [c0007bf8] [c0003ae8]
```

Adresy wymienione w ścieżce śladu muszą dopiero zostać przetłumaczone na nazwy symboliczne funkcji. Jest to możliwe za pośrednictwem polecenia ksymoops i pliku *System.map* generowanego w trakcie kompilacji jądra. Jeżeli jądro korzysta z ładowanych dynamicznie modułów, potrzebne będą również informacje o tych modułach. Polecenie ksymoops próbuje samodzielnie pozyskać wszystkie potrzebne do tłumaczenia informacje, stąd przeważnie wystarczy uruchomić je wywołaniem:

```
ksymoops zapis_wydruku_oops.txt
```

Wynikiem działania programu jest wydrukowanie zdekodowanej wersji wydruku błędu oops. Jeżeli domyślne ustawienia programu ksymoops nie spełniają swojej roli i zachodzi potrzeba określenia własnych źródeł informacji pomocnych w translacji adresów, program można konfigurować szeregiem opcji wywołania. Większość informacji potrzebnych do rozpoczęcia korzystania z programu zawiera dokumentacja systemowa man. Program ksymoops wchodzi w skład większości dystrybucji systemu Linux.

kallsyms

Na szczęście korzystanie z programu `ksymoops` nie jest już konieczne. Na szczęście, ponieważ w przeciwieństwie do programistów jądra szeregowi użytkownicy systemu mieli spore kłopoty z wytypowaniem odpowiedniego pliku `System.map` i tym samym zdekodowaniem wydruku błędu `oops`.

W wersji rozwojowej 2.5 jądra systemu Linux pojawiła się funkcja `kallsyms`, aktywowana za pośrednictwem opcji konfiguracyjnej `CONFIG_KALLSYMS`. Uaktywnienie tej opcji powoduje zapisanie w jądrze symbolicznych nazw adresów funkcji skonsolidowanych do obrazu jądra, dzięki czemu jądro może wypisać zdekodowany stos wywołań. Z tego względu dekodowanie błędów `oops` nie wymaga już dostępności pliku `System.map` ani polecenia `ksymoops`. Z drugiej strony powoduje to zwiększenie rozmiaru jądra, jako że odwzorowanie symboli musi być umieszczone w trwale odwzorowanej pamięci jądra. Koszt ten jednak szybko się zwraca, zwłaszcza przy rozwijaniu jądra, ale także podczas próbnych wdrożeń. Opcja konfiguracyjna jądra `CONFIG_KALLSYMS_ALL` dodatkowo zapisuje nazwy symboliczne wszystkich symboli (nie tylko funkcji), choć to akurat przydaje się tylko w specjalistycznych narzędziach diagnostycznych.

Opcja `CONFIG_KALLSYMS_EXTRA_PASS` powoduje, że w ramach kompilacji i konsolidacji jądra zostanie wykonany drugi przebieg po kodzie obiektowym jądra. Ta operacja służy z kolei przede wszystkim do diagnostyki działania samego mechanizmu `kallsyms`.

Opcje diagnostyczne jądra

W testowaniu kodu jądra i diagnostyce ewentualnych błędów pomocne mogą okazać się dostępne liczne opcje konfiguracyjne kompilacji jądra. Opcje te zebrane są w dziale *Kernel Hacking* opcji konfiguracji jądra. Aktywność wszystkich tych opcji uzależniona jest od włączenia ogólnej opcji `CONFIG_DEBUG_KERNEL`. Przy modyfikacjach jądra warto włączyć wszystkie z dostępnych opcji wspomagających testowanie i diagnostykę.

Niektóre z tych opcji pozwalają na wspomaganie diagnostyki błędów alokatora plastrowego, diagnostyki błędów pamięci wysokiej, odwzorowań wejścia-wyjścia, diagnostyki rygli pętlowych i kontroli przepelnienia stosu. Jedną z najprzydatniejszych opcji jest jednak opcja określana w menu konfiguracyjnym jako *sleep-inside-spinlock checking* (kontrola zawieszenia przy przetrzymywaniu rygla pętlowego, ale nie tylko).

Począwszy od wersji 2.5, jądro systemu Linux dysponuje znakomitą infrastrukturą wspomagającą wykrywanie wszelkiego rodzaju błędów niepodzielności operacji. Z rozdziału 9. wiadomo, że *niepodzielność* to zdolność do nieprzerwanego wykonania kodu — kod niepodzielny powinien zostać wykonany bez przerwania albo nie powinien być wykonany wcale. Kodem niepodzielnym jest kod przetrzymujący rygiel pętlowy bądź kod wykonywany przy wyłączeniu wyłączenia jądra. Kod niepodzielny nie może być zawieszany — zawieszenie kodu przetrzymującego rygiel pętlowy prowadzi prosto do zakleszczenia.

Dzięki wywłaszczeniu jądra jądro to dysponuje centralnym licznikiem niepodzielności. Można dzięki niemu skonfigurować jądro tak, aby zawieszenie zadania w sekcji krytycznej albo nawet inicjowanie operacji potencjalnie zawieszającej zadanie spowodowało wyświetlenie komunikatu o błędzie wraz ze śladem wykonania. W ten sposób dają się wykryć takie błędy jak wywołanie funkcji `schedule()` przez proces przetrzymujący blokadę, inicjowanie przez taki proces blokującej operacji przydziału pamięci czy zawieszanie procesu odwołującego się do danych charakterystycznych dla procesora. Owa infrastruktura diagnostyczna pozwala na wychwycenie znacznej części błędów, więc jej wykorzystywanie jest w procesie testowania i diagnozowania gorąco zalecane.

Wspomniana infrastruktura jest najlepiej wykorzystywana przy następujących ustawieniach opcji:

```
CONFIG_PREEMPT=y
CONFIG_DEBUG_KERNEL=y
CONFIG_KALLSYMS=y
CONFIG_DEBUG_SPINLOCK_SLEEP=y
```

Asercje i wypisywanie informacji o błędach

Do oznaczania błędów, wprowadzania asercji i wyprowadzania informacji służy w jądrze szereg wywołań i makrodefinicji. Najczęściej używane spośród nich to `BUG()` i `BUG_ON()`. Ich wywołanie powoduje wyprowadzenie komunikatu o błędzie `oops` wraz z informacjami o stanie systemu i śladzie wykonania. Sposób prowokowania błędu w tych wywołaniach zależy od architektury systemu. W większości architektur makrodefinicje `BUG()` i `BUG_ON()` rozwijane są do postaci instrukcji niedozwolonych, których próba wykonania prowokuje błąd `oops`. Makrodefinicje te wykorzystuje się w roli asercji do oznaczania sytuacji, które nie powinny mieć miejsca:

```
if (bad_thing)
    BUG();
    albo nawet lepiej:
BUG_ON(bad_thing);
```

Większość programistów jądra uważa, że `BUG_ON()` stosuje się i czyta łatwiej niż `BUG()`; ponadto `BUG_ON()` ujmuje asercję w instrukcji `unl i kely()`. Wśród programistów jądra dyskutowano o pomysłe opcji kompilacji wyrażeń z `BUG_ON()` jako pustych, co dawałoby znaczne oszczędności miejsca w jądrach przeznaczonych do systemów wbudowanych. To wymaga jednak, aby asercja umieszczona w `BUG_ON()` nie posiadała żadnych efektów ubocznych. Zaproponowano też makrodefinicję `BUILD_BUG_ON()` jako asercję czasu kompilacji — jeśli znajdujące się w niej wyrażenie daje wartość `true` w czasie kompilacji, kompilacja zostanie przerwana z komunikatem o błędzie.

Błędy bardziej krytyczne mogą być sygnalizowane wywołaniem `panic()`. Wywołanie to powoduje wyświetlenie komunikatu o błędzie i zatrzymanie systemu. Z oczywistych względów należy je stosować wyłącznie w sytuacjach najtrudniejszych:

```
if (terrible_error)
    panic("okropna sytuacja: %ld!\n", terrible_error);
```

Niekiedy testowanie i diagnostykę można sobie znakomicie ułatwić, wyprowadzając na konsolę wydruk śladu stosu. Służy do tego wywołanie `dump_stack()`. Jego działanie ogranicza się do wyprowadzenia na konsolę zawartości rejestrów i śladu wywołania funkcji:

```
if (!debug_check) {
    printk(KERN_DEBUG "tu dodatkowy komunikat ...\n");
    dump_stack();
}
```

Funkcja Magic SysRq Key

Sporo nerwów testerowi jądra zaoszczędzić może funkcja Magic SysRq Key aktywowana za pośrednictwem opcji konfiguracyjnej `CONFIG_MAGIC_SYSRQ`. Klawisz *SysRq* (żądanie systemowe) obecny jest na większości standardowych klawiatur. W architekturach i386 i PowerPC aktywuje się go za pośrednictwem kombinacji klawiszy *Alt+Print Screen*.

Po uaktywnieniu opcji `CONFIG_MAGIC_SYSRQ` można za pośrednictwem specjalnych kombinacji klawiszy komunikować się z jądrem. Możliwość ta pozwala na wykonywanie pewnych operacji nawet w obliczu „martwego” systemu.

Funkcję specjalnych kombinacji klawiszy można też włączyć, zmieniając parametry działającego systemu za pośrednictwem następującego polecenia:

```
echo 1 > /proc/sys/kernel/sysrq
```

Lista dostępnych operacji wyświetlana jest na konsoli po naciśnięciu kombinacji klawiszy *SysRq* i *h*. Naciśnięcie *SysRq* i *s* powoduje utrwalenie brudnych buforów na dysku, *SysRq* i *u* odmontowuje wszystkie zamontowane systemy plików, natomiast kombinacja *SysRq* i *b* powoduje restart systemu. Użycie powyższych trzech kombinacji klawiszy w podanej kolejności to najbezpieczniejszy sposób restartu niestabilnego systemu, nieporównanie bezpieczniejszy od prostego wciśnięcia przycisku *reset* na obudowie komputera.

System, który uległ całkowitemu zawieszeniu, może nie reagować na kombinacje z klawiszem *SysRq*, ewentualnie może niepoprawnie realizować wydawane w ten sposób polecenia. Jednak przy odrobinie szczęścia włączenie opcji może pozwolić na uratowanie danych i znakomicie wspomóc diagnostykę jądra. Lista poleceń obsługiwanych w ramach funkcji Magic SysRq umieszczona została w tabeli 18.2.

Tabela 18.2. Obsługiwane polecenia SysRq

Kombinacja klawiszy	Opis
<i>SysRq i b</i>	Restart komputera.
<i>SysRq i e</i>	Wysłanie sygnału SIGTERM do wszystkich procesów z wyjątkiem procesu <i>init</i> .
<i>SysRq i h</i>	Wyświetlenie listy poleceń SysRq na konsoli.
<i>SysRq i i</i>	Wysłanie sygnału SIGKILL do wszystkich procesów z wyjątkiem procesu <i>init</i> .
<i>SysRq i k</i>	Unicestwienie wszystkich programów uruchomionych z danej konsoli.
<i>SysRq i l</i>	Wysłanie sygnału SIGKILL do wszystkich procesów, również do procesu <i>init</i> .
<i>SysRq i m</i>	Wygenerowanie zrzutu pamięci i przekazanie go na konsolę.
<i>SysRq i o</i>	Zatrzymanie systemu i wyłączenie komputera.
<i>SysRq i p</i>	Przekazanie na konsolę wydruku zawartości rejestrów.
<i>SysRq i r</i>	Wyłączenie trybu RAW klawiatury i przełączenie jej do trybu XLATE.
<i>SysRq i s</i>	Synchronizacja wszystkich systemów plików (utrwalenie zmian buforów dyskowych).
<i>SysRq i t</i>	Przekazanie na konsolę wydruku opisującego proces.
<i>SysRq i u</i>	Odmontowanie wszystkich zamontowanych systemów plików.

Więcej informacji o funkcji Magic SysRq można znaleźć w pliku *Documentation/sysrq.txt*, w drzewie katalogów kodu źródłowego jądra. Faktyczna implementacja funkcji zdefiniowana jest zaś w pliku *drivers/tty/sysrq.c*. Funkcja Magic SysRq to ważne narzędzie diagnostyki i ratowania „padającego” systemu, jednak z uwagi na to, że jej włączenie daje znaczne możliwości każdemu użytkownikowi konsoli, należałoby ograniczyć stosowanie specjalnych kombinacji klawiszy w komputerach, których bezpieczeństwo jest z jakichś względów ważne. Na prywatnym komputerze wykorzystywanym do programowania i testowania jądra nie ma przeciwwskazań co do włączania funkcji Magic SysRq.

Saga debugera jądra

Wielu programistów jądra od dawna domagało się działającego w jądrze debugera. Niestety, Linus nie podzielał tej chęci i blokował próby wprowadzenia debugera do jądra. Uzasadniał to tym, że obecność debuggerów powoduje wprowadzanie błędnych poprawek

jest łąta wyprowadzona z pełnej znajomości i kodu jądra, i objawów błędu. Mimo to potrzeba wyposażenia jądra w debuger wcale nie osłabła. Z racji małego prawdopodobieństwa spełnienia tych postulatów w najbliższej przyszłości zdesperowani programiści opracowali szereg łąt uzupełniających jądro systemu Linux o obsługę debugera. I choć są to łąty nieoficjalne, stanowią zwykle dobrze wyposażone i efektywne narzędzia. Przed ich omówieniem wypadałoby jednak sprawdzić, w jakim zakresie do diagnostyki jądra nie przyda się standardowy linuksowy debuger gdb.

gdb

Zajrzeć w głąb działającego jądra można za pomocą standardowego debugera GNU. Uruchomienie debugera na rzecz jądra nie różni się od inicjowania debugera na rzecz zwykłych procesów:

```
gdb vmlinux /proc/kcore
```

Plik *vmlinux* to nieskompresowany plik obrazu jądra przechowywany w katalogu głównym drzewa katalogów kompilacji jądra.

Opcjonalny parametr wywołania debugera */proc/kcore* pełni w uruchomieniu debugera rolę pliku obrazu pamięci, za pośrednictwem którego gdb może uzyskać podgląd pamięci jądra. Odczyt pliku */proc/kcore* wymaga uprawnień użytkownika uprzywilejowanego (*root*) systemu.

Po uruchomieniu debugera można korzystać z dowolnych implementowanych w nim poleceń podglądu danych. Na przykład w celu podejrzenia wartości zmiennej `global_variable` należy wystosować polecenie:

```
p global_variable
```

Do deasemblacji funkcji służy zaś polecenie `disassemble`:

```
disassemble nazwa-funkcji
```

Jeżeli jądro było kompilowane z opcją `-g` (czyli było kompilowane po dodaniu `-g` do zmiennej środowiskowej `CFLAGS` w pliku *Makefile* kompilacji jądra), debuger gdb może wydobyć z niego znacznie więcej informacji. Możliwe jest wtedy na przykład podglądanie zawartości struktur i wyłuskiwanie wskaźników. Określenie opcji `-g` powoduje jednak znaczny rozrost jądra, więc nie należy automatycznie uaktywniać tej opcji przy kompilacji jądra do celów innych niż testowe i diagnostyczne.

Niestety, na tym wyczerpuje się lista możliwości debugowania jądra za pomocą programu gdb. Nie da się za jego pośrednictwem modyfikować danych działającego jądra. Nie można też przy jego użyciu wykonywać kodu jądra krokowo ani też ustawiać punktów wstrzymania wykonania. Niemożność modyfikacji struktur danych jądra to poważna wada. I choć możliwość deasemblacji funkcji jest od czasu do czasu przydatna, ogólna przydatność debugera gdb jest — w wyniku braku możliwości modyfikacji danych jądra — ograniczona.

kgdb

kgdb to łąta programowa na jądro systemu Linux pozwalająca na pełne zdalne debugowanie jądra realizowane za pośrednictwem łącza szeregowego. Wykorzystanie debugera kgdb wymaga zastosowania dwóch komputerów. Pierwszy z nich wykonuje kod jądra wyposażonego w łątę kgdb, podczas gdy drugi kontroluje wykonanie kodu jądra za pośrednictwem łącza szeregowego (tak zwanego kabla null-modem łączącego oba komputery) i debugera gdb. Dzięki kgdb debugowanie jądra można wzbogacić o cały szereg niezwykle przydatnych operacji implementowanych przez program gdb, jak definiowanie punktów wstrzymania wykonania, krokowe wykonanie kodu jądra czy podgląd i modyfikacja dowolnych danych jądra itd.! Niektóre z wersji kgdb pozwalają nawet na wykonywanie wybranych funkcji.

Konfiguracja dwóch komputerów jest dość złożona, ale włożony w nią wysiłek owocuje późniejszym prostym debugowaniem jądra. Łata oprócz kodu debugera zawiera obszerną dokumentację instalowaną w podkatalogu *Documentation* drzewa katalogów kodu źródłowego jądra. Warto się z nią zapoznać.

Łaty kgdb dostępne są w rozmaitych wersjach dostosowanych do najróżniejszych architektur sprzętowych i wersji jądra. Najlepszym sposobem pozyskania łąty jest więc przeszukiwanie zasobów internetu.

Stymulowanie i sondowanie systemu

W miarę nabywania doświadczenia w diagnostyce błędów jądra poznaje się sztuczki pozwalające na stymulowanie i sondowanie jądra. Debugowanie jądra jest niewątpliwie czynnością skomplikowaną, stąd wszelka pomoc, nawet w postaci prostych sztuczek, jest na wagę złota. Oto kilka z takich trików.

Uzależnianie wykonania kodu od identyfikatora UID

Jeżeli tworzony kod ma związek z obsługą procesów, możliwe jest opracowanie kodu alternatywnego bez eliminowania z jądra dotychczasowej implementacji. Jest to szczególnie przydatne w przypadku tworzenia nowych wersji istotnych wywołań systemowych, kiedy to pożądanym jest w pełni poprawne działanie systemu, uruchamiającego nowy kod tylko na wyraźne żądanie. Weźmy na przykład wywołanie `fork()`. Jeżeli algorytm tego wywołania zostanie zmodyfikowany pod kątem uwzględnienia w nim nowej, fascynującej możliwości, testowanie i diagnostyka nowej wersji wywołania będą w przypadku błędu implementacji bardzo utrudnione — system, w którym nie działa funkcja `fork()`, nie będzie bowiem ani trochę stabilny. Nie można się jednak poddawać.

W takich przypadkach warto pozostawić istniejący algorytm i wstrzymać niekontrolowane wykonywanie jego nowej wersji. Można na przykład uzależnić uruchomienie nowego algorytmu od wartości identyfikatora użytkownika (UID):

```
if (current->uid != 7777) {
    /* dotychczasowy algorytm */
} else {
    /* nowa wersja algorytmu... */
}
```

Dla wszystkich użytkowników o identyfikatorze UID różnym od 7777 realizowana będzie dotychczasowa, na pewno działająca wersja algorytmu. Testowanie nowej wersji odbywa się wtedy przez utworzenie specjalnego użytkownika o identyfikatorze UID 7777. Znakomicie ułatwia to testowanie kodu o krytycznym znaczeniu dla poprawnego działania procesu.

Korzystanie ze zmiennych warunkowych

Jeżeli niepewny kod nie jest realizowany w kontekście procesu albo jeżeli sterowanie jego wykonaniem ma być bardziej globalne, należy skorzystać ze zmiennych warunkowych. Ta metoda jest nawet prostsza od rozróżniania identyfikatora UID. Wystarczy bowiem utworzyć zmienną globalną i jej wartość wykorzystywać jako barierę wykonania eksperymentalnego kodu. Na przykład wartość zero zmiennej powoduje wykonanie jednej ścieżki kodu, a wartość niezerowa inicjuje wykonanie innej ścieżki kodu. Zmienna może być ustawiana za pośrednictwem eksportowanego do przestrzeni użytkownika interfejsu bądź z poziomu debugera.

Korzystanie ze statystyk

Nierzadko diagnostyka niewłaściwego działania jądra opiera się na informacji o częstotliwości zachodzenia określonego zdarzenia. Niekiedy zachodzi konieczność porównania wielu zdarzeń i wygenerowania pewnych wielkości porównawczych. Można to łatwo zrealizować, kompletując odpowiednie statystyki i implementując mechanizm eksportu ich wartości.

Niech porównywanymi wartościami będą na przykład liczby zdarzeń `b1a` i `b1ab1a`. W pliku kodu (najlepiej tym, który obsługuje dane zdarzenia) wystarczy wtedy zdefiniować dwie zmienne globalne:

```
unsigned long b1a_stat = 0;
unsigned long b1ab1a_stat = 0;
```

Przy każdym wystąpieniu obserwowanego zdarzenia należy zwiększyć wartość odpowiedniej zmiennej i opracować dowolny mechanizm eksportu danych. Na przykład można w tym celu utworzyć plik w podsystemie plików `/proc` i w nim umieszczać wartości zmiennych, ewentualnie zaimplementować specjalne wywołanie systemowe. W ostateczności można też podglądać wartości zmiennych za pomocą debugera.

Warto jednak pamiętać, że taka metoda nie sprawdza się najlepiej w środowisku wieloprocesorowym. W takim środowisku manipulowanie wartościami zmiennych wymaga bowiem stosowania operacji niepodzielnych. Jednak w przypadku prostych statystyk wykorzystywanych jedynie do diagnostyki taka ochrona może być zbędna.

Ograniczanie częstotliwości i liczby komunikatów diagnostycznych

Popularną metodą diagnostyki błędów jest szpikowanie podejrzanego kodu instrukcjami testującymi wartości pewnych zmiennych i wyprowadzającymi komunikaty diagnostyczne na konsolę. Należy jednak wziąć pod uwagę fakt, że w jądrze niektóre z funkcji są wywoływane dziesiątki razy w ciągu sekundy. Jeżeli taka funkcja zostanie naszpikowana wywołaniami `printk()`, system zostanie przeładowany komunikatami diagnostycznymi.

Problem natłoku komunikatów można rozwiązać na dwa proste sposoby. Pierwszy z nich to *ograniczanie częstotliwości* diagnostyki przydatne przy obserwacji postępu zdarzeń, które zachodzą z dość dużą częstotliwością. Aby uniknąć zalewu komunikatów diagnostycznych, należy ograniczyć częstotliwość generowania komunikatów, wyprowadzając je, powiedzmy, wyłącznie raz na kilka sekund. Na przykład:

```
static unsigned long prev_jiffy = jiffies /* ograniczanie częstotliwości */

if (time_after(jiffies, prev_jiffy + 2*HZ)) {
    prev_jiffy = jiffies;
    printk(KERN_ERR "ple ple ple\n");
}
```

W powyższym przykładzie komunikat diagnostyczny jest wyświetlany na konsoli co dwie sekundy. Zapobiega to zalewowi informacji i ryzyku zablokowania konsoli. Częstotliwość można ograniczać jeszcze bardziej bądź pozwalać na częstsze generowanie komunikatów, zależnie od potrzeb.

Można też skorzystać ze specjalnej funkcji ograniczającej częstotliwość generowania komunikatów diagnostycznych:

```
if (error && printk_ratelimit())
    printk(KERN_DEBUG "awaria=%d\n", error);
```

Albo z jeszcze wygodniejszego wariantu:

```
printk_ratelimited(KERN_DEBUG "awaria=%d\n", error);
```

Funkcja `printk_ratelimit()` zwraca zero, jeśli wskutek ograniczania częstotliwości komunikatów diagnostycznych wywołanie `printk()` powinno być pominięte, albo wartość niezerową, gdyby należało je wykonać. Domyślnie funkcja ta przepuszcza jeden komunikat co 5 sekund, ale przedtem przepuszcza nielimitowaną serię do 10 komunikatów. Oba parametry (częstotliwość i rozmiar nielimitowanej serii) można ustawić za pośrednictwem wymuszenia `sysctl` dla parametrów (odpowiednio) `printk_ratelimit` i `printk_ratelimit_burst`. Zaprezentowana powyżej makrodefinicja `printk_ratelimited()` realizuje instrukcję warunkową z `printk_ratelimit()` i ewentualne wywołanie `printk()` bez konieczności jawnego stosowania instrukcji `if`.

Osobny problem pojawia się wtedy, kiedy analizowany ma być przebieg wykonania wzdłuż ścieżki kodu — tym razem chodzi nie o doczekanie się pewnego stanu jądra, a po prostu o powiadomienie o wystąpieniu zdarzenia. Przydatne jest pierwsze takie

powiadomienie, ewentualnie pierwszych kilka. Taka sytuacja dotyczy problemów, w których pierwsze zajście określonego zdarzenia pociąga za sobą lawinę takich samych zdarzeń. Tutaj rozwiązaniem nie jest ograniczanie częstotliwości komunikatów, lecz proste *zmniejszenie* ich liczby:

```
static unsigned long limit = 0;

if (limit < 5) {
    limit++;
    printk(KERN_ERR "ple ple ple\n");
}
```

W tym przykładzie liczba komunikatów diagnostycznych ograniczona jest do pięciu. Po wystosowaniu pięciu komunikatów warunek kontynuowania diagnostyki jest już zawsze fałszywy.

W obu ostatnich przykładach zmienne powinny być statyczne i lokalne względem funkcji — gwarantuje to podtrzymanie wartości zmiennej i tym samym kontynuowanie strategii ograniczania w kolejnych wywołaniach funkcji.

Żaden z powyższych przykładów nie jest zbyt dobrze przystosowany do działania w środowisku wieloprocesorowym — niedomaganie to można jednak szybko wyeliminować, wdrażając w nich operacje niepodzielne. Ale prawdę mówiąc, w kodzie diagnostycznym nie warto chyba zawracać sobie tym głowy.

Szukanie winowajcy — wyszukiwanie binarne

Niejednokrotnie znaczną pomocą w wytypowaniu błędu jest wiedza o momencie, w którym błąd pojawił się w jądrze, w sensie znajomości numeru wersji jądra, od której zaczęto obserwować objawy błędu. Jeżeli na przykład wiadomo, że błąd ujawnił się w wersji 2.6.33, ale nie w wersji 2.6.28, to sprawa jest jasna — wytypowanie błędu polega zazwyczaj na przejrzeniu i przeanalizowaniu listy zmian pomiędzy tymi wersjami (niekiedy wystarczy wycofać feralną zmianę).

Wielokrotnie jednak nie wiadomo na pewno, w której wersji pojawił się dany błąd. Wiadomo jedynie, że obecny jest w wersji *bieżącej*, i wydaje się, że był w tej wersji od zawsze. Wtedy określenie momentu wprowadzenia błędu do kodu jądra wymaga przeprowadzenia śledztwa, które przy odrobinie wysiłku zaowocuje jednak wytypowaniem winowajcy. Po określeniu zmiany wprowadzającej błąd jego usunięcie jest zwykle kwestią czasu, i to niezbyt długiego.

W takich poszukiwaniach potrzebny jest błąd, którego wystąpienie można w sposób pewny sprowokować. Najlepiej, jeśli błąd daje się wywołać zaraz po zakończeniu rozruchu systemu. Następnie należy zaopatrzyć się w jądro, co do którego wiadomo na pewno, że jest wolne od rzeczonego błędu. Może to być na przykład jądro wykorzystywane kilka miesięcy temu, na którym nigdy nie udało się sprowokować błędu. Równie dobrze może to być jednak wersja znacznie wcześniejsza. O ile błąd nie czał się w jądrze od zawsze, poszukiwania takie dają zwykle wyniki stosunkowo szybko.

Potrzebne jest też jądro, o którym wiadomo na pewno, że jest obciążone błędem. Najlepiej, jeżeli będzie to jak najwcześniejsza wersja jądra na pewno zawierająca błąd.

Po uzyskaniu jądra poprawnego należy rozpocząć wyszukiwanie binarne od wersji skażonej do wersji dobrej. Załóżmy, że najnowsze jądro, o którym wiadomo, że jest wolne od błędów, to jądro 2.6.11, a najwcześniejsze jądro z błędem to jądro 2.6.20. Poszukiwania należy rozpocząć od wersji w miarę równo odległej od obu wersji skrajnych, na przykład 2.6.15. Jeżeli w wyniku testów okaże się, że wersja ta jest wolna od błędów, to wiadomo, że błąd pojawił się w kodzie następnych wersji. Należy więc zbadać wersję środkową pomiędzy wersjami 2.6.15 i 2.6.20 — na przykład 2.6.17. Jeżeli zaś test wykaże obecność błędów już w wersji 2.6.15, należy zawęzić poszukiwania do wersji 2.6.11 – 2.6.15 i rozpocząć je od na przykład wersji 2.6.13. Poszukiwania trzeba następnie kontynuować w odpowiednich połówkach tak ograniczonego zakresu wersji.

W końcu kolejne testy doprowadzą do zawężenia zakresu poszukiwań do dwóch kolejnych wersji jądra, w których jedna będzie obciążona błędem, a druga będzie od niego wolna. W tym momencie można już w prosty sposób określić pechową zmianę pomiędzy wersjami. Warto zauważyć, że taki sposób przeszukiwania jest znacznie efektywniejszy niż przeszukiwanie wszystkich kolejnych wersji jądra!

Binarne wyszukiwanie wersji za pomocą Gita

System zarządzania zmianami w kodzie źródłowym Git udostępnia użytkownikom bardzo przydatny i wygodny mechanizm przeprowadzania binarnego wyszukiwania wersji. Każdy, kto kontroluje kopię drzewa kodu źródłowego jądra systemu Linux za pośrednictwem Gita (a jest to metoda preferowana i powszechnie stosowana), może zautomatyzować wyszukiwanie feralnej wersji, składając go na samo repozytorium. Co więcej, Git pozwala na przeprowadzenie wyszukiwania binarnego nie według wersji wydawniczych jądra, ale według zmian zatwierdzanych do repozytorium, a więc z dokładnością do jednej *rewizji* (wersje wydawnicze to agregaty nawet setek rewizji kodu jądra). Dodatkowo sama operacja jest naprawdę prosta (co w przypadku Gita nie zawsze jest oczywiste). Na początek wystarczy zażądać rozpoczęcia wyszukiwania binarnego:

```
$ git bisect start
```

Następnie podaje się najwcześniejszą znaną rewizję, w której błąd występował:

```
$ git bisect bad rewizja
```

Jeśli nie wiadomo, w której wersji błąd się ujawnił, nie trzeba w ogóle podawać rewizji — Git domyślnie uzna, że chodzi o rewizję najnowszą:

```
$ git bisect bad
```

Potem podaje się najpóźniejszą rewizję, o której wiadomo, że nie zawierała błędów:

```
$ git bisect good v2.6.28
```


Wtedy Git przystępuje do automatycznego wyciągnięcia drzewa kodu źródłowego Linuksa z rewizji środkowej podanego przedziału. Wystarczy jądro skompilować, uruchomić, no i przetestować. Jeśli błędu nie ma, polecenie:

```
$ git bisect good
```

wymusi wyszukiwanie binarne wśród nowszych rewizji. Jeśli błąd się ujawni (testowana rewizja manifestuje błąd), należy powiadomić Gita, że kod z tej rewizji jest niepoprawny, i tym samym skierować poszukiwania na wcześniejsze rewizje:

```
$ git bisect bad
```

Za każdym razem Git dokona bisekcji drzewa kodu źródłowego w kolejnym zakresie rewizji. Wystarczy uparcie powtarzać bisekcję do momentu, w którym następnej nie da się już wykonać — wtedy Git wypisze feralny numer rewizji, która wprowadziła błąd do kodu jądra.

Szukanie błędnej rewizji może potrwać, ale automatyzm wydatnie upraszcza całą procedurę. Może ona być jeszcze łatwiejsza i precyzyjniejsza, jeśli przybliżymy Gitowi położenie błędnego kodu. Jeśli na przykład podejrzewamy, że pojawił się błąd w kodzie rozruchu systemu dla architektury x86, możemy zawęzić poszukiwania do tych rewizji, które faktycznie zmieniały coś w ścieżce zawierającej podejrzewany kod:

```
$ git bisect start - arch/x86
```

Koledzy — kiedy wszystko inne zawiedzie

Jeżeli wszelkie próby wytypowania momentu i miejsca wprowadzenia błędu do kodu jądra spełzną na niczym i jeżeli godziny — a najpewniej dni — spędzone nad klawiaturą okażą się bezowocne, to gdy błąd dotyczy głównej wersji jądra, można zawsze liczyć na pomoc innych programistów, tworzących społeczność jądra systemu Linux.

Receptą w takich przypadkach jest wysłanie krótkiego, ale treściwego listu, zawierającego możliwie pełny opis błędu, na adres listy dystrybucyjnej poświęconej programowaniu jądra. W końcu nikt nie lubi błędów i prędzej czy później (zwykle prędzej) pojawi się odzew.

Adresy list dystrybucyjnych, z których korzysta społeczność programistów Linuksa, i adres głównego forum tej społeczności, jakim jest lista *Linux Kernel Mailing List* (*LKML*), podane zostały w rozdziale 20.

Podsumowanie

Kończący się rozdział poświęcony był zagadnieniu diagnostyki błędów w kodzie jądra, czyli procesowi ustalania, *dłaczego* implementacja rozjechała się z planowanym zachowaniem. Wśród zaprezentowanych środków diagnostycznych znalazły się zarówno mechanizmy wbudowane w samo jądro, jak i zewnętrzne debugery. Wśród omawianych technik

w pierwszej kolejności uwzględniono diagnostykę przez komunikaty, na koniec zaś udało się zaprezentować skuteczną technikę wyszukiwania binarnego feralnej wersji kodu w repozytorium Git. Diagnozowanie błędów jądra to doprawdy zadanie nietrywialne, znacznie trudniejsze od diagnostyki zachowania programów przestrzeni użytkownika. Dlatego przyswojenie technik i mechanizmów z tego rozdziału ma zasadnicze znaczenie nie tylko dla testerów, ale przede wszystkim dla programistów jądra.

Następny rozdział poświęcimy innemu dość ogólnemu zagadnieniu, a mianowicie przenośności jądra systemu Linux.

Skorowidz

A

action modifier, *Patrz:*
modyfikator czynnościowy
AIX, 26
algorytm, 149
CFS, 82
dwóch list, 390
eksmisji elementów
najdłużej nieużywanych,
389
FIFO, 102
karuzelowy sprawiedliwy, 79
LRU, *Patrz:* LRU
skalowalność, 149
szeregowania
procesów do wykonania,
149
zadań, 82, 84, 102
usuwania węzłów, 145
windowy, 360
wizjonera, 389
wstawiania węzłów, 145
zachowanie asymptotyczne,
149
złożoność, *Patrz:* złożoność
alignment, *Patrz:*
dane:wyrównanie
alokator plastrowy, 55, 301, 302,
305, 307, 317, 372
diagnostyka błędów, 441
interfejs, 305
API, 25, 108, 109
aplikacja
anomalia, 267
matematyczna, 78
przenośna, 51
Application Programming
Interface, *Patrz:* API
archiwum tar, 37

assembler, 28
asercja, 442
associative array, *Patrz:* mapa
atomicity, *Patrz:* niepodzielność

B

backing store, *Patrz:* magazyn
trwały
bajtów porządek
big-endian, 465, 466, 467
little-endian, 465, 466, 467
wzajemny, 465
bariera, 251, 254
Berkeley Software Distributions,
Patrz: BSD
BH, 178, 180, 193, 201
biblioteka, 29
C, 109
dołączanie dynamiczne, 415
glibc, 40, 120
języka C, 28
ld.so, 377
libc.so, 46, 377
ncurses, 42
POSIX, 109
współużytkowana, 27
big-endian, *Patrz:* bajt porządek
big-endian
binary search tree, *Patrz:* BST
BIOS, 269
bitmapa, 148
BKL, 246, 343
block IO layer, *Patrz:* warstwa
blokowych operacji
wejścia-wyjścia
blok, 350, 351, 359
główny, 323
sprzętowy, 351
systemu plików, 351
wejścia-wyjścia, 351
blokada, 212, 215, 223, 230, 233,
235, 238, 435, *Patrz też:* rygiel
pętlowy
Big Kernel Lock, *Patrz:* BKL
dcache_lock, 338
drobnoziarnista, 219
gruboziarnista, 219
kolejność zwalniania, 218
rekurencyjna, 247
rywalizacja, 218
sekwencyjna, 247, 248
semaforowa, 240
wyłączająca, *Patrz:* semafor
binarny
wysoce pożądaną, 218
wzajemnego wyłączenia h, 243
xtime_lock, 270, 272
blokowanie, 226, 470
mechanizm uogólniony, 239
błąd, 110
diagnostyka, 404, 433, 435,
441
komunikat, 448
poziom, *Patrz:* poziom
diagnostyczny
statystyki, 447
kodu, 111, 434
niepodzielności operacji, 441
obsługa warunkowa, 49
oops, 438, 439, 442
oznaczanie, 442
paniczny, 439
raport, 480, 481
reprodukcja, 434
synchronizacji, 434
wydruk, 440
wyszukiwanie binarne, 450
zgłaszanie, 480

BogoMIPS, 280, 281
 bottom half, *Patrz:* BH
 BSD, 26
 BSS, 368
 BST, 139, 140, 143
 porządek, 143
 buffer head, *Patrz:* bufor nagłówek
 bufor, 351
 cykliczny, 437
 dcache, 336, 337, 338
 deskryptor, 351
 dyskowy, 387
 komunikatów, 437, 438
 nagłówek, 351, 353, 357
 operacji blokowych, 396
 sprzętowy, 156
 TLB, *Patrz:* TLB
 translacji adresów, *Patrz:* TLB
 wpisów katalogowych,
 Patrz: bufor dcache
 busy looping, *Patrz:* pętla
 aktywnego oczekiwania

C

cache, *Patrz:* pamięć podręczna
 cache eviction, *Patrz:* pamięć
 podręczna eksmisja
 cache hit, *Patrz:* pamięć
 podręczna trafienie
 cache miss, *Patrz:* pamięć
 podręczna pudło
 Canonical, 41
 CFS, 77, 82, 84, 85, 87
 punkt wejścia do planisty,
 87, 94
 usypianie i wybudzanie
 procesów, 87, 95
 wybór procesu, 87, 89, 90
 zliczanie czasu wykonania,
 87
 child process, *Patrz:* proces
 potomny
 chwilka, 248, 263, 264
 circular list, *Patrz:* lista
 cykliczna
 clairvoyant algorithm, *Patrz:*
 algorytm wizjonera
 complete fair queuing I/O
 scheduler, *Patrz:* planista
 operacji wejścia-wyjścia
 sprawiedliwy

Completely Fair Scheduler,
Patrz: CFS
 completion variable, *Patrz:*
 zmienna sygnałowa
 copy-on-write, *Patrz:*
 kopiowanie przy zapisie
 critical section, *Patrz:* sekcja
 krytyczna
 cylinder, 351
 czas, 258
 bezwzględny, 257
 kwantyzacja, 76
 pomiar upływu, 467
 sprawności systemu, 258
 systemowy bieżący, 269,
 272
 upływ, 258
 względny, 257
 czynność, 198, 199, 200

D

dane

 lokalne względem
 procesora, 312, 313, 314,
 315, 316
 lokalność czasowa, 387
 niezsynchronizowane, 389
 sekcja, 53
 struktura, *Patrz:* struktura
 danych
 typ, 458
 atomic_t, 222, 223, 226
 char, 461
 int, 222
 nieprzejrzysty, 459
 rozmiar zadany, 460
 specjalny, 459
 ze znakiem, 460
 Darwin, 26
 data, 272
 data temporal locality, *Patrz:*
 dane lokalność czasowa
 D-BUS, 431
 deadline I/O scheduler, *Patrz:*
 planista operacji wejścia-
 wyjścia terminowy
 debugger, 444, 445, 446
 definicja, 478
 demand paging, *Patrz:*
 stronicowanie na żądanie

demon
 klogd, 438
 nasłuchujący na gnieździe,
 431
 przestrzeni użytkownika,
 438
 syslogd, 438
 dequeue, *Patrz:* kolejka
 wyciągnięcie
 deskryptor
 pamięci, 369, 371, 372, 376
 plastra, 303, 304
 powiadomień podsystemu
 inotify, 140
 procesu, 372, *Patrz:* proces
 deskryptor
 Dijkstra Edsger, 239
 directory entry, *Patrz:* wpis
 katalogowy
 dirty list, *Patrz:* lista stron
 brudnych
 DMA, 287, 306
 docelowe opóźnienie, 85
 dokumentacja, 40
 double-linked list, *Patrz:* lista
 dwukierunkowa
 drzewo, 322
 binarne, 89, 143, 370
 poszukiwań, *Patrz:* BST
 samorównoważące, 144,
 Patrz: BST
 zrównoważone, 144
 czerwono-czarne, 145,
 Patrz: drzewo R-B
 liść, 144, 145, 377
 pozycyjne, 392, 395
 priorytetowe, 392
 R-B, 89, 90, 92, 145, 146,
 148, 370, 377, 379
 korzeń, 377
 węzeł, 145
 wstawianie, 146, 147
 wyszukiwanie, 146
 rbtree, *Patrz:* drzewo R-B
 trie, 148
 urządzeń, 416
 węzeł, 143
 głębokość, 144
 potomny, 145, 377
 wysokość, 144

względnie zrównoważone, 145
 zależności modułów, 43
 dynamic timer, *Patrz*: licznik dynamiczny
 DYNIX, 26
 dyrektywa
 optymalizująca wykonanie gałęzi kodu, 48
 kompilatora asm, 48
 dysk twardy, 349, 351, 403,
 Patrz też: magazyn trwały cylinder, 351
 głowica, 351
 klaster, 351

E

enqueue, *Patrz*: kolejka zakolejkowanie

F

fair scheduler, *Patrz*: planista sprawiedliwy
 FAT, 320, 323, 338
 file-backed mapping, *Patrz*: odwzorowanie plików
 firmware, 40
 flusher threads, *Patrz*: wątek zapisu w tle
 format
 bzip2, 38
 gzip, 38
 xz, 38
 free list, *Patrz*: lista struktur wolnych
 FreeBSD, 26
 funkcja, 109, 376, 474, 477
 __alloc_percpu, 314
 __clone, 63
 __do_softirq, 182
 __exit_mm, 69
 __exit_signal, 70
 __free_pages, 291
 __get_free_page, 291
 __get_free_pages, 290, 292, 293
 __list_del, 131
 __schedule, 94
 __tasklet_hi_schedule, 187
 __tasklet_schedule, 187

__unhash_process, 70
 access, 376
 account_process_tick, 271
 acct_update_integrals, 69
 add_timer, 276
 aio_fsync, 342
 alloc_inode, 327
 alloc_pages, 290, 291, 310, 317
 alloc_pages_exact_node, 304
 alloc_pid, 63
 atomic_add, 224
 atomic_dec, 224
 atomic_inc, 224
 ATOMIC_INIT, 224
 atomic_set, 224
 atomic_sub, 224
 atomic64_add, 227
 atomic64_dec, 227
 ATOMIC64_INIT, 227
 atomic64_set, 227
 atomic64_sub, 227
 barrier, 253, 254
 biblioteczna, 30
 bread, 396
 calc_global_load, 271
 calibrate_delay, 281
 capable, 116
 change_bit, 229
 check_flags, 343
 clear_bit, 229
 clear_inode, 328
 clear_tsk_need_resched, 100
 clone, 54, 63, 64, 66, 68, 371
 close, 339, 376
 compat_ioctl, 341, 343
 complete, 246
 context_switch, 99
 copy_from_user, 115, 116
 copy_mm, 371
 copy_process, 63, 64
 copy_to_user, 115, 116
 create, 332
 d_compare, 338
 d_delete, 338
 d_dname, 338
 d_hash, 338
 d_iput, 338
 d_manage, 338
 d_prune, 338
 d_release, 338
 d_revalidate, 337
 del_timer, 276
 del_timer_sync, 277
 del_timer_sync, 69, 277
 delete_inode, 328
 destroy_inode, 327
 detach_pid, 70
 dirty_inode, 327, 328
 disable_irq, 169, 172
 disable_irq_nosync, 170, 172
 do_exit, 69, 70, 71
 do_fork, 63, 64
 do_gettimeofday, 274
 do_IRQ, 167
 do_mmap, 381, 382
 do_munmap, 383
 do_softirq, 188, 193
 do_timer, 270
 down, 241
 down_interruptible, 240, 241
 down_read_trylock, 242
 down_trylock, 241
 down_write_trylock, 242
 downgrade_write, 243
 drop_inode, 327
 dump_stack, 443
 dup_task_struct, 63
 early_printk, 436
 enable_irq, 170, 172
 enqueue_entity, 90
 enqueue_task, 98
 exec, 54, 62, 64
 exit, 54, 69
 exit_files, 69
 exit_fs, 69
 exit_notify, 69, 71
 exit_sem, 69
 fasync, 342
 fault, 376
 find_get_page, 393, 395
 find_new_reaper, 71
 find_vma, 379, 381
 find_vma_intersection, 381
 find_vma_prev, 381
 flock, 343
 flush, 342
 flush_scheduled_work, 200
 follow_link, 333
 forget_original_parent, 71
 fork, 54, 62, 63, 64, 66, 386, 446

- funkcja
 - formatująca, 435
 - free_irq, 160
 - free_pages, 291, 305
 - fsync, 342, 397
 - get_bh, 353
 - get_cpu, 250, 312
 - get_cpu_var, 313
 - get_unmapped_area, 343
 - get_zeroed_page, 291
 - getattr, 333
 - gettimeofday, 274
 - handle_irq, 167
 - haszująca, 148
 - idr_destroy, 142
 - idr_find, 142
 - idr_get_new, 141
 - idr_get_new_above, 141
 - idr_init, 140
 - idr_pre_get, 140
 - idr_remove, 142
 - idr_remove_all, 142
 - in_interrupt, 170
 - in_irq, 170
 - init_completion, 246
 - init_timer, 275
 - inotify_read, 97
 - insmod, 410
 - int atomic_add_negative, 224
 - int atomic_add_return, 224
 - int atomic_dec_and_test, 224, 225
 - int atomic_dec_return, 225
 - int atomic_inc_and_test, 225
 - int atomic_inc_return, 225
 - int atomic_read, 224
 - int atomic_sub_and_test, 224
 - int atomic_sub_return, 224
 - int test_and_change_bit, 229
 - int test_and_clear_bit, 229
 - int test_and_set_bit, 229
 - int test_bit, 229
 - ioctl, 343
 - irqs_disabled, 170
 - jiffies_64_to_clock_t, 268
 - kfifo_alloc, 138
 - kfifo_free, 138
 - kfifo_in, 137
 - kfifo_init, 138
 - kfifo_len, 137
 - kfifo_out_peek, 137
 - kfifo_reset, 138
 - kfifo_size, 137
 - kfree, 298
 - kmalloc, 292, 298, 302, 316, 317
 - kmap, 311, 317
 - kmap_atomic, 311
 - kmem_cache_alloc, 307
 - kmem_cache_create, 306
 - kmem_cache_destroy, 307
 - kmem_cache_free, 307
 - kmem_freepages, 305
 - kobject_add, 426
 - kobject_create, 426
 - kobject_create_and_add, 426
 - kobject_init, 421
 - kobject_uevent, 431
 - kref_get, 422
 - kresu górnego, *Patrz:*
 - notacja O
 - kthread_stop, 68
 - kunmap, 311
 - kunmap_atomic, 311
 - link, 332
 - list_add, 129
 - list_del, 129, 131
 - list_del_init, 130
 - list_empty, 130
 - list_entry, 131
 - list_for_each, 131
 - list_for_each_entry, 133, 134
 - list_for_each_entry_reverse, 133
 - list_for_each_entry_safe, 133
 - list_for_each_entry_safe_reverse, 134
 - list_move, 130
 - list_move_tail, 130
 - list_splice, 130
 - list_splice_init, 130
 - local_bh_disable, 205
 - local_bh_enable, 205
 - local_irq_disable, 169, 170, 171
 - local_irq_enable, 169, 170, 171
 - local_irq_save, 169
 - lock, 342
 - loff_t llseek, 341
 - long
 - atomic64_add_negative, 227
 - long atomic64_add_return, 227
 - long
 - atomic64_dec_and_test, 227
 - long atomic64_dec_return, 227
 - long
 - atomic64_inc_and_test, 227
 - long atomic64_inc_return, 227
 - long atomic64_read, 227
 - long
 - atomic64_sub_and_test, 227
 - long atomic64_sub_return, 227
 - lookup, 332
 - madvise, 375
 - Magic SysRq Key, 443, 444
 - malloc, 292
 - mb, 254
 - mdelay, 280, 281
 - mkdir, 332
 - mknod, 332
 - mm_exit, 371
 - mm_release, 65
 - mmap, 342, 375, 384
 - mmap, 371
 - mod_timer, 276, 277
 - modprobe, 410
 - module_exit, 406
 - module_init, 405
 - mount, 344
 - munmap, 384
 - mutex_init, 243
 - mutex_is_locked, 244
 - mutex_lock, 243, 244
 - mutex_trylock, 244
 - mutex_unlock, 243, 244
 - my_timer.function, 275
 - ndelay, 280
 - need_resched, 100
 - nice, 104
 - obsługa licznika, 275
 - obsługa list, 131
 - open, 320, 339, 342, 376

- open_softirq, 185
- page_mkwrite, 376
- panic, 443
- per_cpu, 313
- permission, 333
- pick_next_task, 94
- preempt_count, 250
- preempt_disable, 250
- preempt_enable, 250
- preempt_enable_no_resched, 250
- prepare_to_wait, 96
- prepare_write, 394
- printf, 30, 46, 435, 436, 438
- printk, 46, 435, 436, 438
- printk_ratelimit, 448
- process_timeout, 283
- prototyp, 120
- put_bh, 353
- put_cpu, 312
- put_cpu_var, 313
- put_task_struct, 70
- radix_tree_lookup, 395
- rb_erase, 94
- rb_insert_color, 147
- rb_link_node, 147
- read, 320, 388
- read_barrier_depends, 252, 253, 254
- read_lock, 236, 237
- read_lock_irq, 237
- read_lock_irqsave, 236, 237
- read_unlock, 237
- read_unlock_irq, 237
- read_unlock_irqrestore, 237
- readdir, 341
- readlink, 332
- readpage, 393
- reboot, 117
- release, 342, 423
- release_task, 70
- remount_fs, 328
- rename, 332
- request_irq, 157, 159, 160
- rmb, 252, 254
- rmdir, 332
- rmmmod, 410
- rozwijana w miejscu
 - wywołania, 47, 48
- run_local_timers, 272
- rw_lock_init, 237
- rzędu O(1), 128
- sched_get_priority_max, 104
- sched_get_priority_min, 104
- sched_getparam, 104
- sched_getscheduler, 104
- sched_getaffinity, 104
- sched_rr_get_interval, 104
- sched_setaffinity, 104, 105
- sched_setparam, 104
- sched_setscheduler, 104
- sched_yield, 104
- schedule, 69, 94, 99, 442
- schedule_timeout, 281, 282, 283
- scheduler_tick, 99, 272
- sema_init, 240, 241
- set_bit, 229
- set_tsk_need_resched, 100
- setattr, 333
- settimeofday, 274
- setup_arch, 435
- setxattr, 333
- show_interrupts, 169
- skalująca, 268
- skrót, 336, 338
- smp_mb, 253, 254
- smp_processor_id, 312
- smp_read_barrier_depends, 253, 254
- smp_rmb, 253, 254
- smp_wmb, 253, 254
- softirq_pending, 193
- spin_is_locked, 234
- spin_lock, 234
- spin_lock_bh, 234
- spin_lock_init, 234
- spin_lock_irq, 233, 234
- spin_lock_irqrestore, 232
- spin_lock_irqsave, 232, 234
- spin_trylock, 234
- spin_unlock, 234
- spin_unlock_bh, 234
- spin_unlock_irq, 234
- spin_unlock_irqrestore, 234
- sprzętowa, 454
- ssize_t aio_read, 341
- ssize_t aio_write, 341
- ssize_t read, 341
- ssize_t readv, 342
- ssize_t sendfile, 342
- ssize_t sendpage, 343
- ssize_t write, 341
- ssize_t writev, 342
- stats, 328
- sterowana
 - czasem, 257
 - zdarzeniami, 257
- suser, 116
- switch_mm, 99
- switch_to, 99
- symlink, 332
- sync, 397
- sync_fs, 328
- synchronize_irq, 172
- sys_bar, 110
- sys_foo, 119
- sys_getpid, 110
- syscall, 120
- sysfs_create_file, 428
- sysfs_remove_file, 429
- system_call, 112, 119
- tasklet_action, 188
- tasklet_disable, 190
- tasklet_disable_nosync, 190
- tasklet_enable, 191
- tasklet_hi_action, 188
- tasklet_hi_schedule, 187
- tasklet_kill, 191
- tasklet_schedule, 187, 190
- taskletu, 190
- tick_periodic, 270, 271
- truncate, 333
- try_to_wake_up, 98
- udelay, 280, 281
- umount_begin, 328
- unlikely, 442
- unlink, 332
- unlocked_ioctl, 341, 343
- unlockfs, 328
- up, 241
- update_curr, 88
- update_process_times, 270, 271
- update_wall_time, 271
- vfork, 63, 64, 246
- vfree, 300
- vfsmount_t d_automount, 338
- vmalloc, 299, 300, 317
- void kfifo_init, 136
- wait, 54, 70
- wait_for_completion, 246

funkcja

wait4, 54
 waitpid, 54
 wake_up, 96, 98, 283
 wakeup_bdflush, 399
 wakeup_flusher_threads, 397
 wb_writeback, 397, 399
 wmb, 252, 254
 worker_thread, 196
 write, 320, 321, 388
 write_inode, 327
 write_lock, 237
 write_lock_irqsave, 237
 write_seqlock, 248
 write_sequnlock, 248
 write_super, 328
 write_super_lockfs, 328
 write_trylock, 237
 write_unlock, 237
 write_unlock_irq, 237
 write_unlock_irqrestore, 237
 writepage, 393, 394
 yield, 106
 złożoność obliczeniowa,
Patrz: złożoność
 obliczeniowa

G

gałąź, 322
 generator
 liczb losowych, 158
 pseudolosowy jądra, 404
 Git, 37, 450, 451
 generowanie łąt, 482
 głodzenie zleceń, 361, 362, 400
 głowica, 351
 gniazdo multicastowe, 430
 GNOME, 29
 GNOME-doc, 477
 GNU, 28, 40
 godzina, 258, 272
 GPL, 406
 graf skierowany acykliczny, 143
 GUI, 78

H

hard real-time, *Patrz:* system
 czasu rzeczywistego
 rygorystyczny

hash table, *Patrz:* tablica
 skrótów
 hazard, 208, 211, 215, 276, 277,
 440
 heurystyka przewidywania, 364
 HP-UX, 26
 HZ, 267, 268, 269, 467
 wartość optymalna, 260,
 261, 262

I

I/O scheduler, *Patrz:* planista
 operacji wejścia-wyjścia
 identyfikator licznika, 140
 IEEE, 108
 Implementacja
 process_one_work, 197
 Implementacja
 process_scheduled_works, 197
 index node, *Patrz:* i-węzeł
 inicjalizator desygnowany, 479
 inline functions, *Patrz:* funkcja
 rozwijana w miejscu
 wywołania
 i-node, *Patrz:* i-węzeł
 instrukcja
 assemblerowa, 48, 170
 cli, 170, 171
 compare and exchange, 213
 kolejność wykonywania, 469
 niepodzielna, 213
 sti, 170, 171
 switch, 473
 test and set, 213
 zmiennoprzecinkowa, 50
 interfejs, 107
 alokatora plastowego, 305
 API, *Patrz:* API
 BH, 178, 193
 dostępu do sprzętu, 107
 gniazd, 404
 jądra eksportowany, 415
 kolejek zadań, 201
 kolejek prac, 179
 kryptograficzny, 40
 operacji niepodzielnych,
 222, 223, 228
 percpu, 313, 316
 POSIX, 108
 programistyczny, *Patrz:* API

programowy aplikacji,
Patrz: API
 rygla pętlowego, 231, 232
 STREAMS, 33
 systemu plików, 319, 320
 użytkownika, 29
 graficzny, 28, 78
 interrupt handler, *Patrz:*
 procedura obsługi przerwania,
Patrz: przerwanie obsługa
 interrupt service routine,
Patrz: ISR
 IRIX, 26, 33
 IRQ, 154
 ISR, 155
 i-węzeł, 302, 322, 323, 327, 329,
 330, 362, 392
 operacje, 331

J

jądro, 28, 45
 dane wewnętrzne, 53
 egzojądro, 32
 haker, 480
 inicjalizacja, 435
 instalacja, 44
 interfejs eksportowany, 415
 kod, 216
 długość wiersza, 476
 narzędzia
 samodokumentacji,
 477, 479
 odstępy, 473
 wcięcia, 472
 źródłowy, 37, 38, 39
 kod rozruchu, *Patrz:* kod
 rozruchu
 kompilacja, 40, 441
 czas trwania, 43
 komunikat, 437, 438
 konfiguracja, 41, 215, 411,
 412, 441
 licznik, *Patrz:* licznik jądra
 mikrojądro, *Patrz:*
 mikrojądro
 moduł, *Patrz:* moduł
 modułowe, 33
 monolityczne, 32, 33
 nagłówek, 40
 obiekt, *Patrz:* obiekt jądra

opiekun, 480
 pamięć, *Patrz:* jądra
 przestrzeń adresowa, 287
 pula entropii, 158
 rozwojowe, 35
 sondowanie, 446
 stan zawieszenia, 118
 stos, *Patrz:* stos jądra
 styl kodowania, 472, 473, 475, 476, 477, 479
 stymulowanie, 446
 systemu uniksowego, 31
 wątek, *Patrz:* wątek jądra
 wersja, 20, 433, 449
 wielowątkowe, 33
 wyłączone, 33, 207
 wyłączenie, 101, 102, 118, 214, 312, 469, 470
 blokowanie, 249
 zamrożone, 35
 zdarzenie, *Patrz:* Kernel Event Layer

jednostka
 szeregowania, 87
 zarządzająca pamięcią,
Patrz: MMU

jiffy, *Patrz:* chwilka, zmienna
 jiffies

K

karta
 pamięci, 365
 sieciowa, 156

katalog, 322, 324
 arch, 40
 block, 40
 crypto, 40
 Documentation, 40
 domowy, 39
 drivers, 40
 firmware, 40
 fs, 40
 include, 40, 46
 init, 40
 ipc, 40
 kernel, 40
 lib, 40
 linux-x.y.z, 481
 mm, 40

net, 40
 samples, 40
 scripts, 40
 security, 40
 stron główny, *Patrz:* PGD
 tools, 40
 tworzenie, 332
 usr, 40
 usuwanie, 332
 virt, 40

Kernel Event Layer, 430
 Kernel-doc, 477

kernel, 135, 138
 definiowanie, 136
 rozmiar, 137
 usuwanie, 138
 zakolejkowanie, 135
 zerowanie, 138

kgdb, 446

k-grupa, 419, 420

klasa
 szeregowania, 82, 102
 urządzeń, 33

klaster, 351

klawiatura, 157, 349, 404
 kontroler, 154

klawisz
 Alt+Print Screen, 443
 SysRq, 443, 444

klucz, 139, 140

Knuth Donald, 150

kobjects, *Patrz:* obiekt jądra

kod
 assemblerowy, 453
 blokowanie, 214
 ziarnistość, 219
 gałąź, 48
 jądra, 216
 komunikacji
 międzyprocesowej, 40
 niepodzielność, *Patrz:*
 niepodzielność
 odporny na
 wieloprzetwarzanie, 214
 odporny na wyłączenie,
 215
 programu, 53, 377, 378
 przerywalny, 214
 rozruchu, 40
 synchronizacja, 208

kolejka, 135, 148, *Patrz też:* kfifo

kfifo, *Patrz:* kfifo
 offset, 135
 prac, 179, 180, 194, 199, 202, 203
 implementacja, 195
 opóźnianie, 200
 tworzenie, 201
 wyciągnięcie, 135, 136
 zadań, 178, 180, 194, 201, 202
 zakolejkowanie, 135, 136, 137
 zleceń, 357, 362

komentarz, 477

kompilator, 28
 gcc, 47, 48

komunikat, 32
 znacznik priorytetu, 46

konfigurator wiersza polecenia, 42

konsolidator, 28

konsument, 135, 148, 235, 237, 242

kontekst
 NUMA, 302
 przełączanie, 99
 przerywania, 155, 165, 173
 wyłączny, 155

kontroler
 APIC, 169
 wejścia-wyjścia, 169

kopiowanie przy zapisie, 62

koprocesor, 50

kres
 dolny, 150
 górny, 149, 150

ksets, *Patrz:* k-grupa

k-typ, 418, 419, 420

kwant czasu
 procesora, 76, 77, 80, 83, 84, 86, 87
 procesu, 104

L

least recently used, *Patrz:* LRU

licencja
 GNU, 28, *Patrz:* GNU
 GPL, *Patrz:* GPL
 pliku kodu źródłowego, 406

liczba
 losowa, 158
 typ nieprzejrzysty, 57

- licznik, 224
 - APIC, 269
 - czasu, *Patrz:* zegar systemowy
 - czasu procesora, *Patrz:* TSC
 - dynamiczny, 258, 274, 275, 276, 277
 - implementacja, 277
 - identyfikator, *Patrz:*
 - identyfikator licznika jądra, 180, 184, *Patrz:* licznik dynamiczny
 - jiffies, 467, 468
 - odwołań, 356, 421, 422
 - preempt_count, 101
 - sekwencyjny, 247
 - semafora, 239, 240
 - użycia, 239
 - wywłaszczeniowy, 101, 250
 - linked list, *Patrz:* lista
 - Linux Elevator, *Patrz:* winda Linusa
 - Linus
 - Linux
 - Fedora, *Patrz:* Fedora
 - historia, 27
 - numer
 - podwersji, 34
 - rewizji, 34
 - wersji, 34
 - Ubuntu, *Patrz:* Ubuntu
 - wersja, 34
 - Linux Kernel Mailing List, 451, 483
 - linux-kernel mailing list, *Patrz:* lklm
 - lklm
 - list head, *Patrz:* lista czoło
 - lista, 123, 148, 235
 - cykliczna, 124, 125, 126
 - czoło, 125, 128
 - dwukierunkowa, 124, 125, 126, 357
 - dyskrybucyjna
 - linux-kernel mailing list, *Patrz:* lklm
 - LKML, *Patrz:* LKML
 - implementacja, 126
 - inicjalizowanie, 127
 - jednokierunkowa, 124
 - LRU, 390
 - łączenie, 130
 - modyfikacja współbieżna, 134
 - procesów potomnych, 73
 - przeglądanie, 125, 131, 132, 133
 - pusta, 130
 - stron aktywnych, 390
 - stron brudnych, 389
 - struktur wolnych, 301, 302
 - systemów plików, 346
 - węzeł, 123
 - dodawanie, 129
 - przenoszenie, 130
 - usuwanie, 129
 - wpisów katalogowych, 336
 - zadań, 55, 61
 - little-endian, *Patrz:*
 - bajt:porządek:little-endian
 - lklm, 35
 - LKML, 451, 471
 - lock contention, *Patrz:* blokada rywalizacja
 - loglevel, *Patrz:* poziom diagnostyczny
 - LRU, 389
- Ł**
- łata, 406, 445, 446, 481
 - aktualizująca, 37
 - dzielenie na porcje, 484
 - generowanie, 481, 482
 - przyrostowa, 39
 - rozsyłanie, 483
- M**
- Mac OS X, 32
 - Mach, 32
 - macierz dyskowa RAID, *Patrz:* RAID
 - magazyn trwałe, 388
 - magistrala, *Patrz:* szyna
 - major release, *Patrz:* Linux numer wersji
 - makrodefinicja, 225, 474
 - alloc_percpu, 314
 - allocate_mm, 371
 - BUG, 442
 - BUG_ON, 442
 - current, 57, 60, 165
 - DECLARE_COMPLETION, 246
 - DECLARE_PER_CPU, 313
 - DECLARE_TASKLET_DISABLED, 191
 - DECLARE_WAITQUEUE, 96
 - DECLARE_WORK, 199
 - for_each_process, 61
 - free_mm, 372
 - get_cpu_var, 315
 - in_interrupt, 173
 - in_irq, 173
 - irqs_disabled, 173
 - likely, 48, 49
 - local_softirq_pending, 183
 - MODULE_AUTHOR, 406
 - MODULE_DESCRIPTION, 406
 - MODULE_LICENSE, 406
 - module_param, 413
 - next_task, 61
 - prev_task, 61
 - time_after, 266, 267
 - time_after_eq, 267
 - time_before, 266
 - time_before_eq, 267
 - u32 __be32_to_cpu, 467
 - u32 __cpu_to_be32, 467
 - u32 __cpu_to_le32, 467
 - u32 __le32_to_cpu, 467
 - unlikely, 48, 49
 - mapa, 139, 148
 - idr, 140, 142
 - mapowanie, 139, *Patrz też:* odwzorowanie
 - maska gfp_mask, 140
 - MATLAB, 78
 - mechanizm sysenter, 112
 - Memory Management Unit, *Patrz:* MMU
 - metadane, 323
 - systemu plików, 323, 362
 - metoda, *Patrz:* funkcja mikrojądro, 32
 - funkcja rdzenna, 32
 - minimum granularity, *Patrz:* ziarnistość minimalna
 - Minix, 453
 - minor release, *Patrz:* Linux numer podwersji
 - MMU, 31, 285

model
 obiektowy urządzeń, 33
 producenta i konsumenta, 135
 sterowników, 416

moduł, 404
 instalowanie, 409
 kompilacja, 406, 407, 408
 konsolidacja z jądrem, 415
 ładowanie, 410
 nadzorczy, 29
 parametry, 413
 tworzenie, 405
 usuwanie, 410
 zależności pomiędzy modułami, 409

modyfikator
 asmlinkage, 110
 strefy, 293, 295, 310

modyfikator czynnościowy, 293, 294

muteks, *Patrz:* semafor binarny

mutual exclusion, 239

mysz, 416

N

narzędzie
 csh, 26
 vi, 26

NetBSD, 26, 453

netlink, 430

niepodzielność, 208, 221, 225, 229, 230
 błąd, 441

noop I/O scheduler, *Patrz:* planista operacji wejścia-wyjścia nieingerujący

Norton Andrew, 35

notacja
 O, 149, 150
 omega, 150
 theta, 150

NTFS, 320, 323

NULL, 124, 147, 290, 395

NUMA, 302, 305

numer wywołania, 111, 112, 120

O

obiekt, 302, 303
 jądra, 417, 418, 419, 420, 421, 430
 atrybuty, 428, 429
 grupa, *Patrz:* k-grupa
 rodzina, 418
 typ, *Patrz:* k-typ

kolorowanie, 302
 przypięty, 421
 VHS, *Patrz:* VHS obiekt:

obszar TLS, 67

odpytywanie, 153

odstąpienie, 76

odzworowanie, 310,
Patrz: mapowanie anonimowe, 382
 czasowe, 311
 niepodzielne, *Patrz:* odzworowanie czasowe plikowe, 382
 prywatne, 375, 392
 wejścia-wyjścia diagnostyka błędów, 441
 wspólne, 375
 współdzielone, 392

opcja, 42
 CONFIG_IKCONFIG_PR OC, 43
 CONFIG_SMP, 41
 dwustanowa, 41
 włączona, 41
 wyłączona, 41

OpenBSD, 26

operacja
 atomowa, *Patrz:* operacja niepodzielna
 barier, 225
 błąd niepodzielności, 441
 down, 240
 jednobitowa, 229
 niepodzielna, 221, 222, 239
 64-bitowa, 226
 bitowa, 228
 odczytu, 361
 up, 240
 uszeregowanie, 225
 VMA, 375

wejścia-wyjścia, 350, 352, 353, 354, 387, 400
 blokowa, 355, 357, 396
 bufor, 396
 dyskowa, 358
 głodzenie, 361, 362
 kolejka, 362
 planowanie, 358
 serializacja, 362
 wektorowa, 354
 wielostronicowa, 357
 zapisu, 361

opóźniony zapis stron, 387

ordering, *Patrz:* operacja uszeregowanie

P

page global directory, *Patrz:* PGD

page middle directory, *Patrz:* PMD

page table entries, *Patrz:* PTE

page writeback, *Patrz:* pamięć, 285
 adresowanie, 368, 287
 fizyczne, 287, 299, 300, 317, 384, 386
 wirtualne, 287, 299, 317, 384, 386

bariery, 251

ciągłość
 fizyczna, 299, 300, 317
 wirtualna, 299, 317

CMOS, 269

deskryptor, *Patrz:* deskryptor pamięci

dostęp bezpośredni, *Patrz:* DMA

flash, 349

fragmentacja, 301

jądra, 49

jednostka zarządzająca, *Patrz:* MMU

kolejność operacji, 251

lista struktur wolnych, *Patrz:* lista struktur wolnych

lokalna względem procesora, 312

masowa, *Patrz:* magazyn trwały

- pamięć
 - obszar, 368, 372, 376, 377, 379
 - wirtualny, *Patrz:* VMA
 - obszar niedozwolony, 49
 - podręczna, 302, 303, 390
 - bufora, 396
 - eksmisja, 389
 - kolorowanie, 55
 - plaster, *Patrz:* plaster
 - przydzielanie, 305, 307
 - puddło, 388
 - stron, 387, 391, 395
 - trafienie, 388
 - z zapisem bezzwłocznym, 388
 - z zapisem w tle, 388
 - zwalnianie, 307, 389
 - procesów przestrzeni użytkownika, *Patrz:* przestrzeń adresowa
 - procesu
 - przeźren adresowa z odwzorowaniem wirtualno-fizycznym, 53
 - przydział, 140
 - przydzielanie, 290, 292, 295, 296, 297, 298, 299, 301, 314
 - na stosie, 309
 - słowo, *Patrz:* słowo
 - strona, *Patrz:* strona
 - strona zerowa, 379
 - stronicowanie, 34, 350
 - tablica stron, 384, 385
 - pierwszego poziomu, *Patrz:* PGD
 - wtórna, *Patrz:* PMD
 - wirtualna, 27, 40, 53, 107, 367, 384
 - współużytkowana, 207
 - wysoka, 310, 317, 469, 470
 - diagnostyka błędów, 441
 - zwalnianie, 290, 298, 300, 301, 315
- panic error, *Patrz:* błąd
 - paniczny
 - parent process, *Patrz:* rodzic
 - pętla aktywnego oczekiwania, 278
 - PGD, 385
 - PID, 60, 62, 63, 66, 70
 - pinned, *Patrz:* obiekt:przypięty
 - PIT, 269
 - planista, 29, 33, 50, 75, 77, 149, 184, 219
 - CFQ, *Patrz:* planista
 - operacji wejścia-wyjścia
 - sprawiedliwy
 - CFS, 77, 80, 82, 84, 85, 87
 - czasu rzeczywistego, 102
 - kod, 454
 - O(1), 77
 - operacji wejścia-wyjścia, 358, 359, 360
 - nieingerujący, 365, 366
 - przewidujący, 363, 364, 366
 - sprawiedliwy, 364, 365, 366
 - terminowy, 361, 362, 366
 - wybór, 366
 - procesów, 358
 - RSD, 77
 - sprawiedliwy, 76, 86
 - plaster, 302, 303, 306
 - deskryptor, *Patrz:* deskryptor plastra
 - pełny, 302
 - pusty, 302
 - zajęty częściowo, 302
 - plik, 322, 330
 - .config, 42
 - /etc/syslog.conf, 438
 - /fs/proc, 169
 - /proc/<pid>/maps, 377
 - /proc/interrupts, 158, 168, 169
 - /proc/irq, 158
 - /proc/kcore, 445
 - /proc/kmsg, 438
 - /proc/sys/kernel/pid_max, 57
 - /proc/sys/vm, 398
 - /proc/sys/vm/laptop_mode, 398
 - /var/log/messages, 438
 - asm/atomic.h, 223
 - asm/bitops.h, 228
 - asm/byteorder.h, 466
 - asm/delay.h, 280
 - asm/irq.h, 169
 - asm/mman.h, 382
 - asm/page.h, 385
 - asm/percpu.h, 313
 - asm/semaphore.h, 240
 - asm/spinlock.h, 231
 - asm/system.h, 169
 - asm/types.h, 460
 - asm-generic/param.h, 259
 - block/cfq-iosched.c, 365
 - block/noop-iosched.c, 366
 - COPYING, 40
 - CREDITS, 40, 480
 - Documentation/CodingStyle, 472
 - Documentation/oops-tracing.txt, 481
 - Documentation/sysrq.txt, 444
 - drivers/char/rtc.c, 163
 - fs/fs-writeback.c, 398
 - fs/mount.h, 346
 - kernel/exit.c, 371
 - kernel/fork.c, 63, 246, 308, 371
 - kernel/sched/core.c, 82, 94, 246, 454
 - kernel/sched/fair.c, 87
 - kernel/sched_rt.c, 102
 - kernel/softirq.c, 181
 - kernel/time/timekeeping.c, 272
 - kernel/workqueue.c, 196
 - kodu źródłowego, 384, 398
 - lib/string.c, 46
 - libc.so, 378
 - linux/bio.h, 354
 - linux/blkdev.h, 357
 - linux/buffer_head.h, 351
 - linux/delay.h, 280
 - linux/fdtable.h, 345
 - linux/fs.h, 325, 326, 329, 339, 344, 392
 - linux/fs_struct.h, 346
 - linux/gfp.h, 290, 294
 - linux/highmem.h, 311
 - linux/interrupt.h, 157, 181, 184, 189
 - linux/jiffies.h, 264
 - linux/kfifo.h, 135
 - linux/kobject.h, 417, 431
 - linux/list.h, 134
 - linux/mm.h, 374, 376, 381, 382

- linux/mm_types.h, 285, 372
- linux/mmzone.h, 287
- linux/mount.h, 344
- linux/percpu.h, 313, 314
- linux/rbtree.h, 146
- linux/sched.h, 55, 371
- linux/slab.h, 294, 298
- linux/slab_def.h, 293
- linux/spinlock.h, 231
- linux/timer.h, 275
- linux/types.h, 222, 293, 460
- MAINTAINERS, 40, 480, 481
- Makefile, 40, 44
- mm/backing-dev.c, 398
- mm/mmap.c, 379, 381, 384
- mm/page-writeback.c, 398
- mm/slab.c, 313
- nagłówkowy, 46, 55, 134
- objektowy, 378
- prawa dostępu, 333
- REPORTING-BUGS, 481
- specjalny, 332
- System.map, 45, 440
- tablica operacji, 340
- urządzenia, 332
- vmlinux, 445
- wykonywalny, 368
- wyszukiwanie na podstawie ścieżki, 334
- PMD, 385
- podsystem, 40
 - czasu rzeczywistego
 - tolerancyjny, 103
 - dźwiękowy, 40
 - inotify, 140
 - sieciowy, 40, 184
 - urządzeń blokowych, 184
 - zarządzania pamięcią, 40
- polecenie, *Patrz też:* program
 - git clone, 38
 - git pull, 38
 - ksymoops, 440, 441
 - make, 43
 - make config, 42
 - make defconfig, 42
 - make dep, 43
 - make gconfig, 42
 - make menuconfig, 42
 - make oldconfig, 42
 - patch, 39
 - ps -ef, 67
 - ps -el, 79
 - tar, 38
- poll, 341
- polling, *Patrz:* odpytywanie
- połówka
 - dolna, 156, 165, 176, 177, 178, 179, 180, 234, 274
 - blokowanie, 204
 - implementacja, 178, 202
 - obsługa, 156
 - wyłączanie, 204, 205
 - górną, 156, 176, 177, 178
- port szeregowy, 349, 436
- POSIX, 28, 108, 140
- powłoka, 29
- poziom
 - diagnostyczny, 436, 437
 - uprzejmości, *Patrz:* wartość
 - nice
- PPID, 62
- preprocesor HZ, 259
- private mapping, *Patrz:* odzorowanie prywatne
- procedura
 - obsługi przerwania, 29, 31, 60, 154, 155, 156, 157, 158, 160, 161, 162, 163, 175, 176, 204, 235
 - programowego, 181, 185, 191, 201
 - sprzętowego, 269, 293, 309
 - zegarowego, 269
- obsługi wywołań systemowych, 112
- startowa, 60
- proces, 53, 54, 65, 345
- czas wykonania, 87
 - wirtualny, 88
- debugera, 73
- deskryptor, 55, 58, 60, 70, 346
 - pamięci, 371
 - przechowywanie, 57
 - przydzielanie, 55
- drzewo, 60
- duch, 54, 70, 71
- identyfikator, *Patrz:* PID
- init, 60, 72
- interaktywny, 77, 81, 84
- klasy real-time, 82
- kojarzenie z procesorem, 104, 105
- kontekst, 60
- lekki, 65
- macierzysty, *Patrz:* rodzaj
- ograniczony, 78, 81
- potomny, 54, 60, 63, 66
- priority, 79, 84, 85
 - manipulacja, 103
- przełączanie, 80, 85, 99
- rozwidlanie, 63
- rzeźnik, 72
- TASK_INTERRUPTIBLE, 58
- TASK_RUNNING, 58
- TASK_STOPPED, 59
- TASK_TRACED, 59
- TASK_UNINTERRUPTIBLE, 58
- tworzenie, 54
- wirtualizacja, 53
- wyłączenie, *Patrz:*
 - wyłączenie
 - zakończenie, 68
 - zawieszony, 58, 96
 - zmienna, *Patrz:* zmienna
- procesor
 - kwant czasu, *Patrz:* kwant czasu procesora
 - numer, 250, 312
 - pułapka, 462
 - rejestr, 53
 - RISC, 58
 - tworzenie, 62
 - wielordzeniowy, 75
 - wirtualny, 53
- process identification, *Patrz:* PID
- process scheduler, *Patrz:*
 - planista
- procdfs, 169
- producent, 135, 148, 235, 237, 242
- program, 53, *Patrz też:* polecenie
 - aktywny, 54
 - diff, 481
 - gdb, 445
 - indent, 479
 - insmod, 410
 - ładujący, 44
 - make, 409
 - pmmap, 378

- program
 - sekcja tekstu, *Patrz:* kod programu
 - sysctl, 398
 - programowanie współbieżne, *Patrz:* współbieżność
 - protokół, 404
 - TCP/IP, 27
 - przebieżność, 51, 453, 469
 - historia, 455
 - względna, 454
 - przeplot operacji, *Patrz:* hazard
 - przerwanie
 - programowe
 - wyzwalanie, 185
 - przerwanie, 31, 233
 - asynchroniczne, 51
 - blokowanie, 31, 169, 170, 172
 - implementacja obsługi, 166
 - klawiatury, 154
 - kontekst, *Patrz:* kontekst przerwania
 - kontrola, 169
 - kontroler, 154
 - linia współużytkowana, 158, 160
 - maskowanie linii, 172
 - numer, 154, 157
 - obsługa, 29, 31, 153
 - programowe, 112, 155, 179, 180, 181, 183, 188, 191, 202, 203, 214, 235
 - implementacja, 181
 - indeks, 184
 - priorytet, 184
 - uruchamianie, 182
 - wyzwalanie, 185
 - przydział dynamiczny, 154
 - reaktywowane, 192
 - sprzętowe, 153, 155, 182, 214
 - stan systemu, 173
 - synchroniczne, 155
 - synchronizacja, 170
 - systemowe, 113
 - współużytkowane, 162
 - wyłączanie lokalne, 170
 - zegarowe, 154, 257, 258, 269
 - przestrzeń
 - adresowa, 392, 367
 - interwał, 368, 372, 373, 381, 383
 - nazw, 67, 322
 - segmentowana, 367
 - użytkownika, 29
 - przetwarzanie
 - równoległe, 65
 - współbieżne symetryczne, *Patrz:* SMP
 - pseudorównoległość, 213
 - pseudourządzenie, 404
 - PTE, 385
 - pułapka, 108
 - przełączająca tryb procesora, 113
 - punkt montowania, 322, 334
- ## R
- race condition, *Patrz:* hazard
 - RAID, 356
 - rbtree, *Patrz:* drzewo R-B
 - reader, *Patrz:* konsument
 - reader-writer lock, *Patrz:* rygiel pętlowy R-W
 - real-time clock, *Patrz:* zegar czasu rzeczywistego
 - reaper, *Patrz:* proces rzeźnik
 - Red Hat, 41
 - red-black tree, *Patrz:* drzewo R-B
 - Redundant Array of Inexpensive Disk, *Patrz:* RAID
 - rejestr, 47
 - eax, 112, 113
 - ebx, 113
 - ecx, 113
 - edi, 113
 - edx, 113
 - esi, 113
 - koprocesora, 50
 - revision number, *Patrz:* Linux numer rewizji
 - Ritchie Dennis, 25, 27
 - rodzeństwo, 60
 - rodzic, 54, 60, 62, 63, 66
 - Rotating Staircase Deadline, *Patrz:* RSD
 - rozmnażanie, 62
 - RSD, 77
 - RTC, *Patrz:* zegar czasu rzeczywistego, *Patrz:* zegar:czasu rzeczywistego
 - rygiel pętlowy, 51, 215, 230, 231, 232, 233, 234, 235, 238, 239, 245, 248, 249, 250, 395
 - bez aktywnego oczekiwania, 243
 - diagnostyka błędów, 233, 441
 - kontrola zawieszenia przy przetrzymywaniu, 441
 - rekurencja, 232
 - R-W, 235, 236
 - wspólny-wyłączny, *Patrz:* rygiel pętlowy R-W
 - współbieżny-wyłączny, *Patrz:* rygiel pętlowy R-W
 - rywalizacja o blokadę, *Patrz:* blokada rywalizacja
- ## S
- samozakleszczenie, 216
 - scatter-gather I/O, *Patrz:* technika rozrzucania/zbierania
 - scheduler, *Patrz:* planista
 - scheduler entity structure, *Patrz:* jednostka szeregowania
 - sekcja
 - .bbs, 377, 378
 - .bss, 368
 - danych programu, 368, 377, 378
 - krytyczna, 208, 210, 230, 233, 249
 - identyfikacja, 214
 - tekstu programu, 368, 377, 378
 - sektor, 350, 351, 359
 - semafor, 51, 95, 190, 231, 232, 238, 243, 245
 - binarny, 239, 242, 243, 244, 245
 - implementacja, 240
 - IPC, 69
 - licznik, *Patrz:* licznik semafora
 - opuszczanie, 240
 - R-W, 242
 - zliczający, 239
 - semantyka SEM_UNDO, 67
 - sequential lock, *Patrz:* blokada sekwencyjna

- serwer, 32
- SGI, 26
- shared mapping, *Patrz:*
 - odzworowanie wspólne
 - sibling, *Patrz:* rodzeństwo
- Single Unix Specification, 28
- single-linked list, *Patrz:* lista jednokierunkowa
- skalowalność, 219, 220, *Patrz:*
 - złożoność obliczeniowa
- skoroszyt, *Patrz:* katalog
- skrypt inicjalizacyjny, 60
- slab allocator, *Patrz:* alokator plastrowy, *Patrz:* alokator plastrowy
- sleep-inside-spinlock checking, *Patrz:* rygiel pętlowy kontrola zawieszenia przy przetrzymywaniu
- słownik, *Patrz:* mapa
- słowo kluczowe, 465, 474
 - inline, 48
 - static, 48
 - rozmiar, 456, 457
- SMP, 33, *Patrz:*
 - wieloprocessorowość
- socket API, *Patrz:* interfejs gniazd
- soft real-time, *Patrz:* podsystem czasu rzeczywistego tolerancyjny
- softirqs, *Patrz:* przerwanie programowe
- Solaris, 26, 33
- sound, 40
- spawn, *Patrz:* rozmnażanie
- spin lock, *Patrz:* rygiel pętlowy
- spurious wakeup, *Patrz:* zadanie wybudzenie nieplanowane
- sshkeygen, 78
- stała
 - HZ, *Patrz:* HZ
 - USER_HZ, *Patrz:* USER_HZ
- sterta, 392
- stos, 47, 50, 110, 133, 165, 309
 - diagnostyka błędów, 310, 441
 - jądra, 50, 309, 310
 - przepelnienie, 310
 - procesu, 378
 - protokołu TCP/IP, 27
 - przerwań, 166, 309
- strategia
 - dwóch list, 390
 - SCHED_FIFO, 102
 - SCHED_NORMAL, 102
 - SCHED_RR, 102
 - szeregowania, 102
- strefa, 294
 - ZONE_DMA, 287, 288, 295, 306
 - ZONE_DMA32, 287, 295
 - ZONE_HIGHMEM, 287, 288, 289, 295
 - ZONE_NORMAL, 287, 288, 289, 295
- strona, 391, 456
 - aktywna, 390
 - brudna, 389, 396, 397
 - fizyczna, 285, 286, 385
 - strefa, *Patrz:* strefa nieaktywna, 390
 - rozmiar, 285, 468, 469
 - wirtualna, 286
 - wypełniona zerami, 291
 - zapis opóźniony, 387
 - zwalnianie, 291
- stronicowanie
 - kopiowanie przy zapisie, 62
 - na żądanie, 26, 27
- struktura
 - address_space, 391, 392
 - address_space_operations, 393
 - bio, 354, 356, 357, 358
 - bio_vec, 355, 356
 - buffer_head, 353
 - cpu_workqueue_struct, 196
 - dentry_operations, 337
 - dopelnianie, 463, 464
 - drzewa czerwono-czarnego, 89, 90, 92
 - FIFO, 135
 - file, 324
 - file_struct, 346
 - file_system_type, 344
 - files_struct, 345
 - fs_struct, 324, 345, 346
 - idr, *Patrz:* mapa idr
 - inicjalizacja, 479
 - init_task, 61
 - kmem_cache, 303
 - kobject, 419
 - kref, 422
 - ktype, 418
 - mm_rb, 370
 - mm_struct, 69, 371, 372
 - mmap, 370
 - mnt_namespace, 345, 346, 347
 - rb_root, 146
 - request_queue, 357
 - samorównoważąca, 89
 - sched_entity, 87, 88
 - sched_param, 104
 - softirq_action, 181
 - struct buffer_head, 351, 357
 - struct completion, 246
 - struct dentry, 334
 - struct file, 339
 - struct file_system_type, 344
 - struct inode, 302, 329
 - struct kobject, 417, 419
 - struct mutex, 243
 - struct page, 285, 292, 317
 - struct request, 358
 - struct rw_semaphore, 242
 - struct semaphore, 240
 - struct slab, 303
 - struct super_block, 325
 - struct super_operations, 326
 - struct task_struct, 55, 57, 60
 - struct thread_info, 56, 57
 - struct timekeeper, 272
 - struct timer_list, 275
 - super_operations, 327
 - sysfs_dirent, 417
 - task_struct, 63, 65, 69, 105, 371
 - tasklet_struct, 186
 - thread_info, 63, 101, 310
 - vfsmount, 344
 - vm_area_struct, 372, 374, 375, 376, 391, 392
 - vm_operations_struct, 376
 - VMA, 306
 - wake_queue_head_t, 96
 - work_struct, 199
 - workqueue_structure, 195
 - wyrównanie, 463, 464

- SunOS, 26, 301
 - superblok, 323, 325
 - supervisor, *Patrz:*
 - SUSv3, 109
 - sygnał, 49, 67
 - SIGSEGV, 49
 - SIGSTOP, 59
 - SIGTSTP, 59
 - SIGTTIN, 59
 - zablokowany, 67
 - syscall number, *Patrz:* numer wywołania
 - syscalls, *Patrz:* wywołanie systemowe
 - system, 29
 - bezwładność, 78
 - czas sprawności, *Patrz:* czas sprawności systemu
 - czas rzeczywistego rygorystyczny, 103
 - energochłonność, 263
 - interaktywność, 80, 85
 - kompilacji, *Patrz:* toolchain
 - kbuild, 406
 - kontroli wersji, 37
 - operacyjny, 29
 - bezzegarowy, 263
 - przenośność, *Patrz:* przenośność rdzeń, 29
 - plików, 66, 322, 349, 391, 403
 - blok, *Patrz:* blok ext3, 344
 - ext4, 344, 393
 - FAT, *Patrz:* FAT
 - metadane, *Patrz:* metadane systemu plików
 - montowanie, 322
 - NTFS, NTFS
 - odmontowywanie, 328
 - procfs, *Patrz:* procfs
 - rekordowy, 322
 - sysfs, 325, 423, 424, 426, 427, 428, 429, 430
 - UDF, 344
 - wirtualny, *Patrz:* VFS
 - wirtualny pamięciowy, 325
 - przerwań, 173, 175
 - RISC, 462
 - sysfs, 33
 - wielozadaniowy, *Patrz:*
 - wielozadaniowość
 - wirtualizacja, 107
 - zawieszenie, 76
 - zliczania odwołań, 421, 422
 - system timer, *Patrz:* zegar
 - systemowy
 - szyna
 - ISA, 288
 - PCI, 154, 157
- ## Ś
- ścieżka dostępu, 322, 334
 - rozwiązywanie, 334
- ## T
- tablica
 - asocjacyjna, *Patrz:* mapa dentry_hashtable, 336
 - haszująca, 139
 - implementacja, 148
 - pidhash, 70
 - mieszająca, *Patrz:* tablica skrótów
 - przemieszczania, *Patrz:*
 - tablica skrótów
 - skrótów, 336, 395
 - softirq_vec, 183
 - statyczna, 123
 - stron, 384, 385
 - sys_call_table, 111
 - takt, 258
 - tarball, *Patrz:* archiwum tar
 - targeted latency, *Patrz:* docelowe opóźnienie
 - task, *Patrz:* zadanie
 - task list, *Patrz:* lista zadań
 - task queue, *Patrz:* kolejka zadań
 - tasklet, 179, 180, 184, 185, 186, 188, 189, 191, 202, 203, 204, 214, 235
 - implementacja, 186, 189
 - szeregowanie, 187, 190
 - tworzenie, 189, 190
 - technika rozrzucania/zbierania, 354
 - temporary mapping, *Patrz:* odwzorowanie czasowe
 - Thompson Ken, 25, 27
 - thrashing, *Patrz:* zaśmiecanie
 - tick rate, *Patrz:* zegar systemowy
 - częstotliwość taktowania
 - tickless, *Patrz:* tryb bezzegarowy
 - TID, 67
 - Time Stamp Counter, *Patrz:* TSC
 - timeslice, *Patrz:* kwant czasu procesora
 - TLB, 300, 385
 - toolchain, 28
 - Torvalds Linus, 27, 33, 36, 37, 480
 - translation lookaside buffer, *Patrz:* TLB
 - transmisja sieciowa, 430
 - Tru64, 26
 - tryb
 - bezzegarowy, 263
 - laptopowy, 398, 399
 - nieuprzywilejowany, 29
 - symetrycznego przetwarzania równoległego, 207
 - uprzywilejowany, 29, 32, 112
 - wieloprocessorowy, *Patrz:* wieloprocessorowość
 - TSC, 269
 - typ
 - blkdevs, 403
 - cdevs, 404
 - definicja, 478
 - miscdevs, 404
 - typ, *Patrz:* znacznik typu
- ## U
- Ubuntu, 41
 - UID, 140, 446, 447
 - odzworowanie na wskaźnik, 140, 141, 142, 148
 - Unix historia, 25
 - uptime, *Patrz:* czas sprawności systemu
 - urządzenie
 - blokowe, 349, 403
 - o silnie swobodnym dostępie, 365, 366
 - sektor, *Patrz:* sektor sterownik, 407
 - węzeł, 403
 - ethernetowe, *Patrz:* urządzenie sieciowe

pełne, 404
 plik, *Patrz:* plik urządzenia
 podłączane na gorąco, 404
 puste, 404
 reprezentujące pamięć, 404
 różne, 404
 sieciowe, 403, 404
 sterownik, 155, 157, 278,
 299, 349, 403, 416, 480
 USB, 407
 wirtualne, 404
 zerujące, 404
 znakowe, 349, 403, 404, 407
 sterownik, 407
 węzeł, 404
 USER_HZ, 267, 268
 użytkownik
 root, 39, 116
 uprzywilejowany, 116

V

VFS, 319, 320, 321, 322, 323, 334
 obiekt, 343
 pliku, 339
 podstawowy, 324
 wpisu katalogowego, 335
 VHS
 blok główny, 326
 obiekt
 bloku głównego, 324, 325
 dentry_operations, 324
 file_operations, 324
 inode_operations, 324
 i-węzła, 324, 329, 331
 operacji, 324
 pliku, 324
 super_operations, 324
 wpisu dentry, *Patrz:* VHS
 obiekt wpisu
 katalogowego
 wpisu katalogowego, 324,
 334, 337, 338
 virtual filesystem, *Patrz:* VFS
 virtual runtime, *Patrz:* proces
 czas wykonania wirtualny
 VMA, 306, 372, 374, 375

W

wall time, *Patrz:* godzina
 warstwa
 abstrakcji systemu plików,
 320, 322
 blokowych operacji
 wejścia-wyjścia, 350
 obiektowa, 323
 plastrowa, *Patrz:* alokator
 plastrowy
 wartość
 nice, 79, 80, 83, 84, 85, 104
 NULL, *Patrz:* NULL
 wątek, 53, 65, 371
 aktywne oczekiwanie, 213
 events, 195
 grupa, 67
 jądra, 67, 372, 397
 bdflush, 399, 400
 ksoftirqd, 191, 192
 kupdated, 399
 pdflush, 399, 400
 tworzenie, 68
 keventd, 202
 pętla aktywnego
 oczekiwania, 231, 232
 roboczy, 195, 196, 198
 samozakleszczenie, 216, 232
 wykonania, *Patrz:* wątek
 zapisu w tle, 397, 400
 zawieszenie, 231, 232, 238,
 281
 wektor wejścia-wyjścia, 355, 356
 wielobieżność, 162
 wieloprocesorowość, 51, 60, 65,
 207, 214, 215, 251, 470
 wielowątkowość, 27
 wielozadaniowość, 27, 50, 75, 107
 idealna, 85
 z kooperacją, 76
 z wywłaszczaniem, 76, 80
 wiersz poleceń konfigurator,
Patrz: konfigurator wiersza
 polecenia
 winda Linusa, 360, 362
 Windows 7, 32, 457
 Windows 95, 454
 Windows NT, 32, 457
 Windows Vista, 32

Windows XP, 32
 work queues, *Patrz:* kolejka prac
 wpis
 dentry, *Patrz:* wpis
 katalogowy
 katalogowy, 322
 writeback, 389
 write-back cache, *Patrz:* pamięć
 podręczna z zapisem w tle
 writer, *Patrz:* producent
 write-through cache, *Patrz:*
 pamięć podręczna z zapisem
 bezzwłocznym
 współbieżność, 50, 65, 204, 213
 błędy, 214
 hazard, 51, 96
 prawdziwa, 214
 źródła, 214
 wyjątek, 108, 155
 wyrównanie, 462, 463
 wyścig, 208, *Patrz:* hazard
 wywłaszczenie, 33, 51, 76, 80,
 100, 102, 182, 207, 213, 234
 blokowanie, 249
 wywołanie
 free_percpu, 315
 systemowe, 29, 60, 103, 107, 109,
 115, 120, *Patrz też:* funkcja
 argument, 113, 114
 bezparametrowe, 110
 błęd, 110, 111
 definiowanie, 110
 fork, 54, *Patrz:* funkcja:fork
 implementacja, 113, 114,
 121, 122
 ioctl, 114
 kompilacja, 119, 120
 kontekst, 118
 lista, 111
 nazwa, 111
 niezaimplementowane, 111
 numer wywołania, *Patrz:*
 numer wywołania
 obsługa, 111
 rejestrowanie, 119
 sys_ni_syscall, 111
 tablica, 111, 119, 120
 uprawnienia, 116
 wiązanie, 119
 wielokrotnione, 114

Z

- zadanie, *Patrz też:* proces
 - czasu rzeczywistego, 102
 - flush, 67
 - init, 439
 - jałowe, 66, 439
 - kolejka, 96, *Patrz:* kolejka zadań
 - kolejkowanie, 201
 - ksoftirqd, 67
 - lista, *Patrz:* lista zadań
 - okresowe, 257
 - osierocone, 71
 - priorytet, 103
 - uśpione, 76
 - wybudzenie, 98
 - nieplanowane, 98
 - zablokowane, 75, 95
 - zawieszane, 95
- zakleszczenie, 216, 232, 236, 247
- zaśmiecianie, 316
- zdominowanie odczytu przez zapis, 361
- zegar
 - czasu rzeczywistego, 163, 268
 - przerwań programowalny, *Patrz:* PIT
 - systemowy, 84, 155, 157, 163, 184, 257, 258, 268, 269
 - częstotliwość taktowania, 258, 259, 260
- ziarnistość minimalna, 86
- zlecenie
 - scalanie, 358, 359, 360, 365
 - sortowanie, 358, 359, 360
- złożoność
 - asymptotyczna, 139
 - czasowa, 150
 - kwadratowa, 150
 - liniowa, 144, 150
 - logarytmiczna, 139, 144, 150
 - O, 150, 370, 377
 - obliczeniowa, 128, 149, 150
 - silniowa, 150
 - sześcienna, 150
 - wykładnicza, 150
- zmienna, 53
 - atomic_t, 223
 - globalna, 210
 - jiffies, 263, 264, 267, 269
 - zawijanie, 266
 - lokalna, 215
 - względem procesora, 312, 313, 314, 315
 - loops_per_jiffy, 281
 - niepodzielna, 422
 - sygnalowa, 245, 246
 - warunkowa, 447
 - wyrównanie naturalne, 462, 463
 - xtime, 269
- znacznik
 - bh_state_bits, 352
 - CLONE_CHILD_CLEARTID, 67
 - CLONE_CHILD_SETTID, 67
 - CLONE_FILES, 66, 346
 - CLONE_FS, 66, 346
 - CLONE_IDLETASK, 66
 - CLONE_NEWNS, 67, 347
 - CLONE_PARENT, 67
 - CLONE_PARENT_SETTID, 67
 - CLONE_PTRACE, 67
 - CLONE_SETTID, 67
 - CLONE_SETTLS, 67
 - CLONE_SIGHAND, 67
 - CLONE_STOP, 67
 - CLONE_SYSVSEM, 67
 - CLONE_THREAD, 67
 - CLONE_UNTRACED, 67
 - CLONE_VFORK, 67
 - CLONE_VM, 67, 371
 - GFP, 140
 - gfp_mask, 293
 - IRQF_DISABLED, 158
 - IRQF_SAMPLE_RANDOM, 158
 - IRQF_SHARED, 158
 - IRQF_TIMER, 158
 - MADV_RANDOM, 375
 - MADV_SEQUENTIAL, 375
 - MAP_ANONYMOUS, 383
 - MAP_DENYWRITE, 383
 - MAP_EXECUTABLE, 383
 - MAP_FIXED, 383
 - MAP_GROWSDOWN, 383
 - MAP_LOCKED, 383
 - MAP_NONBLOCK, 383
 - MAP_NORESERVE, 383
 - MAP_POPULATE, 383
 - MAP_PRIVATE, 383
 - MAP_SHARED, 383
 - MNT_NODEV, 345
 - MNT_NOEXEC, 345
 - MNT_NOSUID, 345
 - modyfikator, *Patrz:*
 - modyfikator
 - need_resched, 100, 101, 262
 - PF_FORKNOEXEC, 63
 - PF_SUPERPRIV, 63
 - PROT_EXEC, 382
 - PROT_NONE, 382
 - PROT_READ, 382
 - PROT_WRITE, 382
 - RQF_DISABLED, 177
 - SA_INTERRUPT, 158
 - SLAB_CACHE_DMA, 306
 - SLAB_HWCACHE_ALIGN, 306
 - SLAB_PANIC, 306
 - SLAB_POISON, 306
 - SLAB_RED_ZONE, 306
 - typu, 293, 296
 - VM_EXEC, 375
 - VM_IO, 375
 - VM_READ, 375
 - VM_SEQ_READ, 375
 - VM_SHARED, 375
 - VM_WRITE, 375
 - VMA, 374
- zombie process, *Patrz:* proces duch
- zone, *Patrz:* strefa
- zone modifier, *Patrz:* modyfikator strefy

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>



Przekonaj się, jak działa jądro Linuksa!

Jądro systemu Linux to jedno z największych osiągnięć otwartej społeczności programistów. Projekt ten, zainicjowany przez Linusa Torvaldsa, światło dzienne ujrzał w 1991 roku. Od tamtego czasu coraz więcej ochotników wspiera jego doskonalenie. Pozwoliło to na wprowadzenie do jądra wielu nowatorskich funkcji i wyznaczenie trendów w rozwoju współczesnego oprogramowania oraz systemów operacyjnych. Prace nad tym gigantycznym projektem przyczyniły się też do powstania mnóstwa innych produktów – w tym zdobywającego ogromną popularność rozproszonego systemu kontroli wersji Git.

Tak dynamiczny rozwój ma jeden niedobry skutek uboczny. Początkujący programiści mają problem z opanowaniem ogromu projektu i poznaniem jego architektury. Ten długotrwały proces będzie zdecydowanie krótszy dzięki przewodnikowi, który właśnie trzymasz w rękach. Przeprowadzi Cię on przez tajemnice systemu. Dowiesz się, jak pobrać kod źródłowy jądra, jak go skonfigurować oraz skompilować. W kolejnych rozdziałach poznasz kluczowe mechanizmy: zarządzania procesami, szeregowania zadań, wywołań systemowych oraz przerwań. Ponadto nauczysz się korzystać z urządzeń blokowych, pamięci podręcznej, sterowników i modułów. Ta książka jest obowiązkową lekturą dla wszystkich programistów, którzy chcą mieć swój wkład w rozwój projektu Torvaldsa. Sprawdzi się ona również w rękach osób, które po prostu chcą zrozumieć, jak działa ten niezwykle system.

Dzięki tej książce:

- pobierzesz, skonfigurujesz i skompilujesz źródła jądra
- zrozumiesz zasady zarządzania procesami
- poznasz struktury danych jądra
- wykorzystasz wirtualne systemy plików
- zyskasz możliwość samodzielnego rozwijania jądra systemu Linux

helion.pl
księgarnia
internetowa

Nr katalogowy: 17927



Księgarnia internetowa
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowosci>

Helion SA
ul. Kosciuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Informatyka w najlepszym wydaniu


Addison
Wesley

cena: 79,00 zł

ISBN 978-83-246-4273-1



ślęgnij po **WIĘCEJ**



KOD KORZYŚCI

9 788324 642731