helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

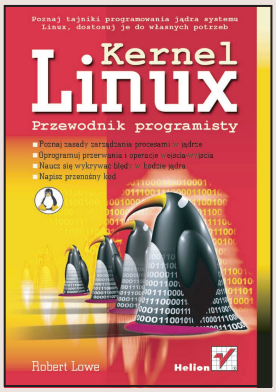
TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Linux Kernel. Przewodnik programisty

Autor: Robert Lowe
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-439-7
Format: B5, stron: 400



Dołącz do grona twórców popularności Linuksa

Fenomen Linuksa polega na tym, że jest on dziełem programistów z całego świata. Każdy może dopisać do niego nową funkcję lub udoskonalić istniejące. Jeśli Linux nie obsługuje urządzenia zainstalowanego w Twoim komputerze – możesz zaimplementować jego obsługę, pisząc własny moduł jądra.

Programowanie jądra systemu Linux nie wymaga żadnych niezwykłych zdolności. Choć jest ono rozległym projektem informatycznym, w żadnej sposób nie różni się od innych projektów tego typu. Oczywiście, samodzielne napisanie choćby kawałka kodu jądra wymaga nauki, ale napisany dotychczas kod nie jest w żaden sposób wyjątkowy ani niezrozumiały. Podstawowym materiałem pomocniczym, niezwykle przydatnym przy opanowywaniu tajników programowania jądra, jest istniejący już kod źródłowy, dostępny dla wszystkich. Jednakże samo poznawanie kodu nie wystarczy – należy również zdobyć wiedzę dotyczącą zasad funkcjonowania systemu operacyjnego i pełnionych przez niego funkcji.

Książka „Linux Kernel. Przewodnik programisty" została napisana po to, aby pomóc programistom w poznaniu zasad tworzenia kodu modułów jądra. Szczegółowo omawia podsystemy i funkcje jądra Linuksa, ich projekt i implementację. Autor książki porusza również zagadnienia związane z projektowaniem systemów operacyjnych.

Książka opiera się na wersji 2.6 jądra systemu Linux i zawiera informacje dotyczące następujących tematów:

- Podstawowe zasady programowania jądra
- Zarządzanie procesami
- Algorytmy szeregowania zadań
- Wywołania systemowe
- Przerwania
- Metody synchronizacji jądra
- Zarządzanie czasem i pamięcią
- Operacje wejścia-wyjścia
- Diagnostyka kodu jądra
- Przenośność kod

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O Autorze **13**

Przedmowa **15**

Wstęp **17**

Słowo od Autora **19**

Rozdział 1. Jądro systemu Linux — wprowadzenie **23**

Wprowadzenie do systemu Linux 25
Przegląd systemów operacyjnych 26
Jądro Linuksa a jądro klasycznego systemu uniksowego 28
Oznaczenia wersji jądra Linuksa 29
Społeczność programistów jądra systemu Linux 31
Odmienność jądra 31
- Brak biblioteki libc 32
- GNU C 32
- Brak mechanizmu ochrony pamięci 34
- Niemożność (łatwego) korzystania z operacji zmiennoprzecinkowych 35
- Ograniczony co do rozmiaru i stały stos 35
- Synchronizacja i współbieżność 35
- Znaczenie przenośności 36

Kompilowanie jądra 36
Zanim zaczniemy 38

Rozdział 2. Zarządzanie procesami **39**

Deskryptor procesu i struktura zadania 40
- Alokacja deskryptora procesu 41
- Przechowywanie deskryptora procesu 42
- Stan procesu 43
- Manipulowanie bieżącym stanem procesu 44
- Kontekst procesu 45

Tworzenie procesu 46
- Kopiowanie przy zapisie 47
- Wywołanie fork() 47
- Wywołanie vfork() 49

Wątki w systemie Linux ... 49
Wątki jądra ... 51
Zakończenie procesu ... 52
Usuwanie deskryptora procesu ... 53
Problem zadań osieroconych ... 54

Rozdział 3. Szeregowanie zadań ... 57
Strategia postępowania ... 58
Procesy ograniczone wejściem-wyjściem a procesy ograniczone procesorem ... 58
Priorytet procesu ... 59
Kwant czasu ... 60
Wywłaszczanie procesu ... 61
Strategia szeregowania w działaniu ... 61
Algorytm szeregujący ... 62
Kolejka procesów gotowych do uruchomienia ... 62
Tablice priorytetów ... 65
Przeliczanie kwantów czasu ... 66
Wywołanie schedule() ... 67
Wyznaczanie nowego priorytetu i kwantu czasu ... 68
Zawieszanie i pobudzanie procesów ... 71
Równoważenie obciążenia ... 73
Wywłaszczanie i przełączanie kontekstu ... 75
Wywłaszczanie procesu użytkownika ... 76
Wywłaszczenie jądra ... 76
Czas rzeczywisty ... 77
Wywołania systemowe związane z szeregowaniem ... 78
Wywołania wpływające na strategię szeregowania i wartości priorytetów ... 79
Wywołania systemowe sterujące kojarzeniem procesów z procesorami ... 80
Odstąpienie procesora ... 80

Rozdział 4. Wywołania systemowe ... 81
API, POSIX i biblioteka C ... 82
Wywołania systemowe ... 83
Numery wywołań systemowych ... 84
Wydajność wywołania systemowego ... 85
Procedura obsługi wywołań systemowych ... 85
Oznaczanie właściwego wywołania systemowego ... 86
Przekazywanie argumentów ... 86
Implementacja wywołania systemowego ... 87
Weryfikacja argumentów ... 87
Kontekst wywołania systemowego ... 89
Wiązanie wywołania systemowego ... 90
Inicjowanie wywołania systemowego z przestrzeni użytkownika ... 92
Cztery powody, aby nie implementować wywołań systemowych ... 93

Rozdział 5. Przerwania i procedury obsługi przerwań ... 95
Przerwania ... 95
Procedury obsługi przerwań ... 96
Połówki górne i dolne ... 97
Rejestrowanie procedury obsługi przerwania ... 98
Zwalnianie procedury obsługi przerwania ... 100
Tworzenie procedury obsługi przerwań ... 100
Procedury obsługi przerwań współużytkowanych ... 102
Prawdziwa procedura obsługi przerwania ... 103
Kontekst przerwania ... 104

Implementacja obsługi przerwań ... 105
/proc/interrupts ... 108
Kontrola przerwań ... 109
Wyłączanie i włączanie przerwań ... 110
Blokowanie konkretnej linii przerwania ... 111
Stan systemu przerwań ... 112

Rozdział 6. Dolne połówki i czynności odroczone ... 115
Połówki dolne ... 116
Po co dolne połówki? ... 117
Świat dolnych połówek ... 117
Przerwania programowe ... 120
Implementacja przerwań programowych ... 120
Korzystanie z przerwań programowych ... 123
Tasklety ... 125
Implementacja taskletów ... 125
Korzystanie z taskletów ... 128
Wątek jądra ksoftirqd ... 130
Dawny mechanizm BH ... 132
Kolejki robót ... 133
Implementacja kolejek robót ... 133
Korzystanie z kolejek robót ... 137
Dawny mechanizm kolejkowania zadań ... 140
Jak wybrać implementację dolnej połówki? ... 140
Blokowanie pomiędzy dolnymi połówkami ... 142
Wyłączanie dolnych połówek ... 142

Rozdział 7. Wprowadzenie do synchronizacji jądra ... 145
Sekcje krytyczne i przeplot operacji ... 146
Po co ta ochrona? ... 146
Blokowanie ... 147
Skąd się bierze współbieżność? ... 149
Co wymaga zabezpieczania? ... 150
Zakleszczenia ... 151
Rywalizacja a skalowalność ... 154
Blokowanie we własnym kodzie ... 155

Rozdział 8. Metody synchronizacji jądra ... 157
Operacje niepodzielne ... 157
Niepodzielne operacje na liczbach całkowitych ... 158
Niepodzielne operacje bitowe ... 160
Rygle pętlowe ... 162
Inne metody blokowania ryglami pętlowymi ... 165
Rygle pętlowe a dolne połówki ... 166
Rygle pętlowe R-W ... 166
Semafory ... 168
Tworzenie i inicjalizowanie semaforów ... 170
Korzystanie z semaforów ... 171
Semafory R-W ... 172
Zmienne sygnałowe ... 174
Blokada BKL (Big Kernel Lock) ... 174
Blokady sekwencyjne ... 176
Blokowanie wywłaszczania ... 177
Bariery ... 178

Rozdział 9. Liczniki i zarządzanie czasem 183
Czas z punktu widzenia jądra 184
Częstotliwość taktowania — HZ 185
Optymalna wartość HZ 186
Chwilki 188
Wewnętrzna reprezentacja zmiennej jiffies 190
Zawijanie zmiennej jiffies 191
HZ a przestrzeń użytkownika 192
Zegary i liczniki sprzętowe 193
Zegar czasu rzeczywistego 193
Zegar systemowy 193
Procedura obsługi przerwania zegarowego 194
Data i godzina 196
Liczniki 198
Korzystanie z liczników 199
Liczniki i sytuacje hazardowe 201
Implementacja licznika 201
Opóźnianie wykonania 202
Oczekiwanie w pętli aktywnej 202
Krótkie opóźnienia 204
Funkcja schedule_timeout() 205

Rozdział 10. Zarządzanie pamięcią 209
Strony 209
Strefy 211
Pozyskiwanie stron pamięci 213
Pozyskiwanie czystych stron pamięci 214
Zwalnianie stron 215
Funkcja kmalloc () 216
Znaczniki gfp_mask 217
Funkcja kfree() 221
Funkcja vmalloc () 222
Alokator plastrowy 223
Zadania alokatora plastrowego 224
Interfejs alokatora plastrowego 227
Statyczne przydziały na stosie 230
Odwzorowanie pamięci wysokiej 231
Odwzorowanie trwałe 231
Odwzorowania czasowe 232
Jak metodę przydziału wybrać? 233

Rozdział 11. Wirtualny system plików 235
Wspólny interfejs systemu plików 235
Warstwa abstrakcji systemu plików 236
Uniksowy system plików 237
Obiekty VFS i ich struktury danych 238
Inne obiekty warstwy VFS 239
Obiekt bloku głównego 240
Operacje bloku głównego 241
Obiekt i-węzła 243
Operacje i-węzła 245
Obiekt wpisu katalogowego 247
Stan wpisu katalogowego 249
Bufor wpisów katalogowych 249
Operacje na wpisach katalogowych 251

Obiekt pliku ... 252
Operacje na plikach ... 253
Struktury danych systemu plików ... 256
Struktury danych procesu ... 257
Systemy plików w Linuksie ... 259

Rozdział 12. Blokowe urządzenia wejścia-wyjścia ... 261
Anatomia urządzenia blokowego ... 262
Bufory i nagłówki buforów ... 263
Struktura bio ... 266
Stare a nowe ... 268
Kolejki zleceń ... 269
Zlecenia ... 269
Zawiadywanie operacjami wejścia-wyjścia ... 269
Zadania planisty operacji wejścia-wyjścia ... 270
Winda Linusa ... 271
Terminowy planista operacji wejścia-wyjścia ... 272
Przewidujący planista operacji wejścia-wyjścia ... 274

Rozdział 13. Przestrzeń adresowa procesu ... 277
Deskryptor pamięci ... 279
Przydział deskryptora pamięci ... 280
Zwalnianie deskryptora pamięci ... 281
Struktura mm_struct i wątki jądra ... 281
Obszary pamięci ... 282
Znaczniki VMA ... 283
Operacje VMA ... 284
Obszary pamięci na listach i w drzewach ... 285
Obszary pamięci w praktyce ... 286
Manipulowanie obszarami pamięci ... 288
Funkcja find_vma() ... 288
Funkcja find_vma_prev() ... 289
Funkcja find_vma_intersection() ... 289
Tworzenie interwału adresów — wywołania mmap() i do_mmap() ... 290
Wywołanie systemowe mmap() ... 292
Usuwanie interwału adresów — wywołania munmap() i do_munmap() ... 292
Wywołanie systemowe munmap() ... 292
Tablice stron ... 293

Rozdział 14. Pamięć podręczna stron i opóźniony zapis stron w tle ... 295
Pamięć podręczna stron ... 296
Obiekt address_space ... 297
Drzewo pozycyjne ... 300
Tablica skrótów stron ... 300
Pamięć podręczna buforów ... 301
Demon pdflush ... 301
bdflush i kupdated ... 303
Eliminowanie przeciążenia, czyli po co w jądrze wiele wątków? ... 303

Rozdział 15. Diagnostyka ... 305
Od czego zacząć? ... 305
Błędy w jądrze ... 306
Funkcja printk() ... 307
Wszechstronność funkcji printk() ... 307
Ograniczenia funkcji printk() ... 307
Poziomy rejestrowania ... 308

Bufor komunikatów ... 309
Demony syslogd i klogd ... 310
Funkcja printk() a hakerzy jądra ... 310
Błąd oops ... 310
Polecenie ksymoops ... 312
kallsyms ... 313
Opcje diagnostyczne jądra ... 313
Diagnostyka niepodzielności operacji ... 313
Prowokowanie błędów i wyprowadzanie informacji ... 314
Funkcja Magic SysRq Key ... 315
Saga debugera jądra ... 315
gdb ... 316
kgdb ... 317
kdb ... 318
Stymulowanie i sondowanie systemu ... 318
Uzależnianie wykonania kodu od identyfikatora UID ... 318
Korzystanie ze zmiennych warunkowych ... 319
Korzystanie ze statystyk ... 319
Ograniczanie częstotliwości komunikatów diagnostycznych ... 319
Szukanie winowajcy — wyszukiwanie binarne ... 321
Koledzy — kiedy wszystko inne zawiedzie ... 322

Rozdział 16. Przenośność ... 323

Historia przenośności systemu Linux ... 325
Rozmiar słowa i typy danych ... 326
Typy nieprzejrzyste ... 328
Typy specjalne ... 329
Typy o zadanych rozmiarach ... 329
Znak typu char ... 330
Wyrównanie danych ... 331
Unikanie problemów wyrównywania ... 331
Wyrównanie typów niestandardowych ... 332
Dopełnienie struktury ... 332
Wzajemny porządek bajtów ... 333
Typy wzajemnego porządku bajtów — rys historyczny ... 335
Wzajemny porządek bajtów w jądrze ... 335
Pomiar upływu czasu ... 336
Rozmiar strony ... 336
Kolejność wykonywania instrukcji ... 337
Tryb SMP, wywłaszczanie jądra i pamięć wysoka ... 338
Przenośność to wyzwanie ... 338

Rozdział 17. Łaty, haking i społeczność ... 339

Społeczność ... 339
Obowiązujący styl kodowania ... 340
Wcięcia ... 341
Nawiasy klamrowe ... 341
Nazewnictwo ... 342
Funkcje ... 342
Komentarze ... 343
Definicje typów ... 344
Korzystanie z zastanego ... 344
Unikanie definicji ifdef w kodzie źródłowym ... 344
Inicjalizacja struktur ... 345
Poprawianie złego stylu ... 345

Łańcuch poleceń ... 346
Przesyłanie raportów o błędach ... 346
Generowanie łat ... 347
Rozsyłanie łat ... 348

Dodatek A Korzystanie z list ... 351
Listy cykliczne ... 351
Poruszanie się pomiędzy elementami listy ... 352
Implementacja listy w jądrze Linuksa ... 353
Struktura listy ... 353
Manipulowanie listami ... 354
Przeglądanie list ... 356

Dodatek B Interfejs procesora ... 359
Nowy interfejs procesora ... 360
Statyczne dane procesora ... 360
Dynamiczne dane procesora ... 361
Po co korzystać z danych procesora? ... 362

Dodatek C Generator liczb losowych jądra ... 365
Projekt i implementacja puli entropii ... 366
Problem rozruchu programu ... 368
Interfejs wejściowy puli entropii ... 368
Interfejs wyjściowy puli entropii ... 369

Dodatek D Złożoność obliczeniowa ... 371
Algorytmy ... 371
Zapis złożoności O(x) ... 372
Notacja duże theta ... 372
Co z tego wynika? ... 373
Pułapki złożoności czasowej ... 373

Dodatek E Bibliografia i lektury dodatkowe ... 375
Książki o projektowaniu systemów operacyjnych ... 375
Książki o jądrze systemu Unix ... 376
Książki o jądrze systemu Linux ... 377
Książki o jądrach innych systemów operacyjnych ... 377
Książki o interfejsie programowym Uniksa ... 377
Inne książki ... 378
Witryny WWW ... 378

Skorowidz ... 381

Rozdział 1.

Jądro systemu Linux — wprowadzenie

Mimo zaawansowanego już wieku (trzech dekad) system Unix wciąż jest uważany za jeden z najefektywniejszych i najlepiej pomyślanych systemów operacyjnych. Powstał w roku 1969 — od tego momentu legendarne już dzieło Dennisa Ritchie i Kena Thompsona wciąż wytrzymuje próbę czasu, który zresztą lekko tylko zaznacza na Uniksie swój ząb.

Unix wyrósł na Mutiksie, zarzuconym przez ośrodek badawczy Bell Laboratories projekcie wielodostępnego systemu operacyjnego. Po zatrzymaniu projektu pracownicy ośrodka badawczego zostali bez interaktywnego systemu operacyjnego. Dopiero latem roku 1969 programiści Bell Labs zakreślili zarysy plikowego systemu operacyjnego, który ostatecznie ewoluował do postaci obecnej w systemie Unix. Projekt został zaimplementowany przez Kena Thompsona na stojącym bezczynnie po zarzuceniu Multiksa komputerze PDP-7. W roku 1971 system został zaadaptowany do specyfiki komputera PDP-7, a w roku 1973 całość została przepisana w języku C, co — choć w owym czasie było przedsięwzięciem bez precedensu — otwarło możliwość przenoszenia systemu na dowolne niemal maszyny. Pierwszym Uniksem wykorzystywanym na szerszą skalę poza ośrodkiem Bell Labs był Unix System Sixth Edition (edycja szósta), znany jako wersja V6.

Również inne firmy podjęły rękawicę i rozpoczęły przenoszenie systemu na nowe platformy. Towarzyszące tym przenosinom ulepszenia zaowocowały powstaniem szeregu odmian systemu operacyjnego. W roku 1977 ośrodek Bell Labs opublikował pod nazwą Unix System III spójną kompilację owych rozszerzeń; w roku 1982 firma AT&T udostępniła zaś słynny System V[1].

Prostota architektury Uniksa oraz fakt, że był on rozprowadzany wraz z kodem źródłowym, umożliwiły rozwój systemu już poza firmą jego twórców. Największy udział w rozwoju systemu miał Uniwersytet Kalifornijski w Berkeley. Odmiany systemu Unix opracowywane na tym uniwersytecie noszą nazwę Berkeley Software Distributions

[1] A co z wersją System IV? Plotka głosi, że wersja ta istniała wyłącznie jako wersja rozwojowo-badawcza.

(BSD). Pierwszą z nich była 3BSD, wydana w roku 1981. Następnymi wersjami były produkty z serii czwartej, a więc 4.0BSD, 4.1BSD, 4.2BSD oraz 4.3BSD. W tych wersjach system Unix został wzbogacony o obsługę pamięci wirtualnej, stronicowanie na żądanie (ang. *demand paging*) oraz stos TCP/IP. W roku 1993 udostępniona została ostateczna wersja systemu z serii 4BSD — 4.4BSD, zawierająca przepisany moduł zarządzania pamięcią wirtualną. Współcześnie rozwój gałęzi BSD jest kontynuowany w ramach systemów FreeBSD, NetBSD oraz OpenBSD. W latach osiemdziesiątych i dziewięćdziesiątych na rynek trafiło też wiele odmian systemu Unix autorstwa różnych firn produkujących systemy serwerowe i stacje robocze. Systemy te bazowały zwykle na implementacji AT&T lub odmiany BSD, rozszerzając je o obsługę cech charakterystycznych dla platform, na które były przenoszone. Do takich właśnie systemów należą: Tru64 firmy Digital, HP-UX firmy Hewlett-Packard, IBM-owski AIX, DYNIX firmy Sequent, IRIX firmy SGI czy słynny Solaris autorstwa Sun Corporation.

Elegancja pierwotnego projektu systemu operacyjnego Unix w połączeniu z latami innowacji i ewolucji dały efekt w postaci wydajnego, niezawodnego i stabilnego systemu operacyjnego. Źródło takiej elastyczności systemu Unix tkwi w szeregu jego cech. Po pierwsze bowiem Unix jest systemem prostym; niektóre inne systemy operacyjne składają się z tysięcy wywołań systemowych, których przeznaczenie nie zawsze jest oczywiste. Systemy z rodziny Uniksa implementują zwykle zaledwie kilkaset wywołań, z których każde jest starannie przemyślane i realizuje ściśle określone funkcje. Po drugie, w systemie Unix *wszystko jest plikiem*[2]. Upraszcza to znakomicie manipulowanie danymi i urządzeniami — dostęp do danych i urządzeń realizowany jest za pośrednictwem zestawu prostych wywołań systemowych: `open()`, `read()`, `write()`, `ioctl()` oraz `close()`. Po trzecie zaś, jądro systemu Unix oraz jego podstawowe elementy zostały napisane w języku C, co daje temu systemowi niezrównaną przenośność i przybliża go szerokiemu gronu programistów.

Dalej, system Unix cechuje się krótkim czasem tworzenia procesu oraz unikalnym wywołaniem `fork()`. Wreszcie system ten udostępnia proste acz przemyślane podstawy komunikacji międzyprocesowej, co w połączeniu z krótkim czasem tworzenia procesów pozwala na konstruowanie prostych narzędzi systemowych, *realizujących jedną funkcję, ale realizujących ją jak najlepiej*. Z owych funkcji mogą korzystać zadania bardziej złożone.

Obecnie Unix jest nowoczesnym systemem operacyjnym obsługującym wielozadaniowość, wielowątkowość, pamięć wirtualną, stronicowanie na żądanie, biblioteki współużytkowane oraz stos protokołu TCP/IP. Istnieją odmiany Uniksa dające się skalować do setek procesorów, ale istnieją również wersje osadzone, przeznaczone do obsługi wyspecjalizowanych platform o minimalnych możliwościach. I choć Unix od dawna nie jest już projektem badawczym, jego kolejne implementacje nadążają za nowymi koncepcjami w dziedzinie systemów operacyjnych, co na szczęście nie odbywa się kosztem przydatności Uniksa jako systemu ogólnego przeznaczenia.

[2] Cóż, może nie wszystko, ale znaczna liczba elementów systemu reprezentowana jest plikiem. W najnowocześniejszych implementacjach systemu Unix i jego pochodnych (jak np. Plan9) pliki reprezentują *niemal wszystko*.

Unix zawdzięcza swój sukces elegancji i prostocie pierwotnego projektu. Jego dzisiejszą siła tkwi w pierwszych decyzjach podejmowanych przez Dennisa Ritchie, Kena Thompsona oraz innych współtwórców pierwotnych wierszy kodu systemu. To właśnie te decyzje są źródłem zdolności systemu operacyjnego Unix do ciągłej ewolucji.

Wprowadzenie do systemu Linux

Linix został opracowany przez Linusa Torvaldsa w roku 1991. Z założenia miał to być system operacyjny dla komputerów wykorzystujących procesor Intel 80386, będący ówcześnie procesorem stosunkowo nowym i niewątpliwie nowoczesnym. Dziś Linux jest systemem operacyjnym działającym na szeregu platform, w tym AMD x86064, ARM, Compaq Alpha, CRIS, DEC VAX, H8/300, Hitachi SuperH, HP PA-RISC, IBM S/390, Intel IA-64, MIPS, Motorola 68000, PowerPC, SPARC, UltraSparc oraz v850. Linux obsługuje zarówno zegarki, jak i klastry superkomputerów. Zwiększyło się też komercyjne zainteresowanie Linuksem. Dziś zarówno firmy związane z Linuksem, jak i pozostali gracze rynkowi, oferują wykorzystujące ten system rozwiązania programowe dla urządzeń wyspecjalizowanych, komputerów biurkowych oraz systemów serwerowych.

Linux jest klonem systemu Unix, ale nie jest Uniksem. Znaczy to, że choć Linux wykorzystuje szereg koncepcji opracowanych pierwotnie na potrzeby Uniksa i implementuje interfejs programowy tego systemu (zdefiniowany specyfikacjami POSIX oraz Single Unix Specification), nie jest bezpośrednią pochodną kodu źródłowego systemu Unix, jak to ma miejsce w przypadku innych systemów uniksopodobnych. Tam, gdzie to potrzebne, Linux jest inny od pozostałych systemów uniksowych, ale odmienność ta nigdy nie naruszała ani podstawowych cech projektowych, ani interfejsu programowego pierwowzoru.

Jedną z najciekawszych cech Linuksa jest fakt, że nie jest to produkt komercyjny — Linux jest owocem współpracy wielu programistów, możliwej dzięki medium, jakim jest internet. I choć Linus Torvalds nigdy nie utraci miana twórcy systemu Linux i wciąż jest opiekunem jądra systemu, jego dzieło jest kontynuowane przez niezliczoną rzeszę programistów. Do rozwoju Linuksa może zresztą przyczynić się dosłownie każdy, kto ma na to ochotę. Jądro systemu Linux, podobnie jak większość jego pozostałej implementacji, jest oprogramowaniem *darmowym* albo inaczej *wolnym*[3]. W szczególności jądro systemu Linux objęte jest Powszechną Licencją Publiczną GNU (ang. *GNU General Public Licence*, *GPL*) w wersji 2.0. Licencja ta gwarantuje możliwość nieodpłatnego pobierania kodu źródłowego i wprowadzania do niego dowolnych modyfikacji. Jedynym wymaganiem jest redystrybucja wersji zmodyfikowanych na tych samych, otwartych zasadach GNU GPL, co oznacza między innymi konieczność dystrybuowania kodu źródłowego wersji zmodyfikowanej[4].

[3] Szkoda tu miejsca na szczegółowe omawianie różnicy pomiędzy oprogramowaniem darmowym i wolnym. Czytelnicy zainteresowani tym tematem powinni zajrzeć pod adresy *http://www.fsf.org* oraz *http://www.opensource.org*.

[4] Warto zapoznać się z treścią licencji GNU GPL. Można ją znaleźć w pliku *COPYING* w drzewie katalogów kodu źródłowego jądra Linuksa oraz na stronie *http://www.fsf.org* (przekład treści licencji na język polski można znaleźć pod adresem *http://www.linux.org.pl/gpl.php — przyp. tłum.*).

Linux dla różnych osób oznacza zupełnie coś innego. Podstawą systemu Linux są jądro, biblioteka języka C, kompilator, system kompilacji (ang. *toolchain*, czyli komplet narzędzi programistycznych, takich jak asembler, konsolidator i inne). Ale system Linux może zawierać również implementację nowoczesnego środowiska graficznego X Window System wraz z kompletnym graficznym interfejsem użytkownika, jakim jest choćby GNOME. Linux ma tysiące komercyjnych i niekomercyjnych zastosowań. W tej książce słowo *Linux* będzie jednak najczęściej oznaczać *jądro systemu Linux*. Tam, gdzie nie będzie to oczywiste, znaczenie tego słowa będzie wskazywane jawnie. Nawiasem mówiąc, termin *Linux* odnosi się właśnie do samego jądra systemu.

Powszechna dostępność kodu źródłowego systemu Linux oznacza możliwość dowolnej konfiguracji jądra przed jego kompilacją. Można więc wybrać do kompilacji jądro zawierające wyłącznie te sterowniki i moduły, które są dla danego zastosowania niezbędne. Elastyczność taka jest zapewniana wielością opcji konfiguracji o nazwach postaci `CONFIG_`*`CECHA`*. Przykładowo, aby w jądrze włączyć obsługę symetrycznego przetwarzania współbieżnego (ang. *symmetrical multiprocessing*, *SMP*), należy do opcji kompilacji dołączyć opcję `CONFIG_SMP`. Jeżeli opcji tej zabraknie, tryb SMP zostanie z jądra usunięty. Opcje kompilacji przechowywane są w pliku *.config* w katalogu głównym drzewa katalogów kodu źródłowego jądra. Plik ów można wypełniać za pośrednictwem jednego z programów konfigurujących proces kompilacji, jak np. `make xconfig`. Opcje konfiguracyjne służą zarówno do włączania do kompilacji kolejnych plików implementacji, jak i do sterowania kompilacją za pośrednictwem dyrektyw preprocesora.

Przegląd systemów operacyjnych

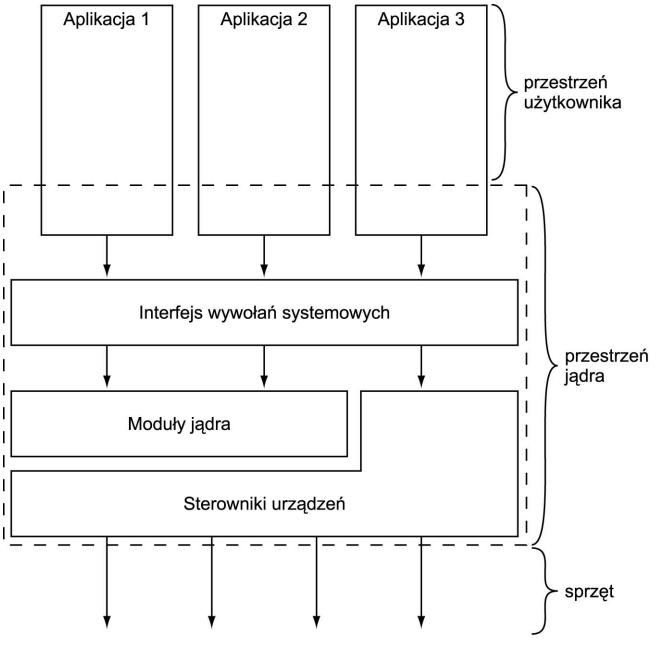
Dzięki niektórym współczesnym systemom operacyjnym pojęcie systemu operacyjnego nie jest już dziś precyzyjne. Użytkownicy często uważają za system operacyjny to, co widzą po uruchomieniu komputera. Zgodnie z ogólnym i wykorzystywanym w tej książce rozumieniem tego pojęcia system operacyjny obejmuje te fragmenty systemu komputerowego, które służą do podstawowej administracji i umożliwiają wykorzystanie systemu. System operacyjny obejmuje więc jądro wraz ze sterownikami urządzeń, moduł ładowania systemu operacyjnego, powłokę (ewentualnie inny interfejs użytkownika) oraz podstawowe pliki konfiguracyjne i narzędzia systemowe. A więc wszystko, co niezbędne do działania systemu. Termin *system* odnosi się przy tym do systemu operacyjnego oraz wszystkich aplikacji, które da się w nim uruchomić.

Książka ta jest rzecz jasna poświęcona *jądru* systemu operacyjnego. Tak jak interfejs użytkownika jest najbardziej zewnętrzną warstwą systemu operacyjnego, tak jego jądro stanowi część najbardziej integralną. Jądro to „samo sedno" systemu — zawiera oprogramowanie implementujące podstawowe usługi dla wszelkich innych elementów systemu, zarządzające sprzętem oraz dystrybuujące zasoby systemowe. Jądro określane jest niekiedy mianem *modułu nadzorczego* (ang. *supervisor*) albo *rdzenia* (ang. *core*) systemu operacyjnego. Najbardziej typowe składniki jądra to: procedury obsługi przerwań, planista (ang. *scheduler*) sterujący podziałem czasu procesora pomiędzy uruchomione w systemie procesy, moduł zarządzania pamięcią zarządzający przestrzeniami adresowymi procesów oraz usługi systemowe, w rodzaju obsługi sieci czy komunikacji

międzyprocesowej. W nowocześniejszych systemach wyposażonych w jednostkę zarządzania pamięcią z jej ochroną jądro działa zazwyczaj w wyodrębnionym od stanu zwykłych aplikacji użytkowych stanie; stan ten obejmuje ochronę pamięci oraz pełen dostęp do zasobów sprzętowych. Ów stan wraz z obszarem pamięci jądra zwany jest *przestrzenią jądra*. Z drugiej strony programy użytkowe wykonywane są w przestrzeni użytkownika. W tej przestrzeni widoczny jest tylko fragment zasobów komputera; z przestrzeni użytkownika nie można też inicjować niektórych funkcji systemowych ani odwoływać się bezpośrednio do sprzętu. W trakcie wykonywania kodu jądra system znajduje się w *przestrzeni jądra* — aplikacje użytkowe wykonywane są zaś w *przestrzeni użytkownika*.

Aplikacje działające w systemie mogą jednak komunikować się z jądrem za pośrednictwem zestawu wywołań systemowych (patrz rysunek 1.1). Aplikacja inicjuje te wywołania zwykle z poziomu pewnej biblioteki (np. biblioteki języka C), która z kolei, wykorzystując interfejs wywołań systemowych, instruuje jądro o potrzebie wykonania funkcji systemowej na rzecz aplikacji. Niektóre wywołania biblioteczne oferują wiele cech, których próżno szukać w implementacji wywołań systemowych — w takim przypadku wywołanie systemowe stanowi zaledwie ułamek operacji realizowanych przez funkcję biblioteczną. Przykładem takiej funkcji jest znana z pewnością Czytelnikowi funkcja `printf()`. Obsługuje ona formatowanie i buforowanie danych znakowych, inicjując wywołanie systemowe `write()` jedynie w celu ostatecznego wyprowadzenia danych na urządzenie zewnętrzne. Z drugiej strony, niektóre funkcje biblioteczne są bezpośrednio odwzorowywane do wywołania systemowego. Przykładem takiej funkcji bibliotecznej jest funkcja `open()`, której działanie sprowadza się do zainicjowania wywołania systemowego `open()`. Istnieje też kategoria funkcji bibliotecznych, jak `strcopy()`, które w ogóle nie korzystają z wywołań systemowych (a przynajmniej nie powinny). Kiedy aplikacja inicjuje wywołanie systemowe, mówi się, że na rzecz tej aplikacji wykonywana jest funkcja jądra. Aplikacje mogą bowiem wykonywać wywołania systemowe w przestrzeni jądra — wtedy jądro działa w *kontekście procesu*. Ten związek pomiędzy aplikacją a jądrem, umożliwiający wywoływanie kodu jądra za pośrednictwem wywołania systemowego, jest podstawą działania wszystkich aplikacji.

Jądro zarządza sprzętem zainstalowanym w systemie komputerowym. Niemal wszystkie znane architektury, włącznie z tymi obsługiwanymi przez system Linux, wykorzystują pojęcie *przerwań*. Przerwanie służy urządzeniu sprzętowemu komunikacji z systemem za pośrednictwem asynchronicznej ingerencji w wykonywanie kodu jądra. Do poszczególnych przerwań przypisane są numery. Na podstawie tych numerów jądro wybiera do wykonania w obliczu przerwania odpowiednią *procedurę obsługi przerwania*. Na przykład, kiedy użytkownik naciśnie klawisz na klawiaturze, kontroler klawiatury inicjuje przerwanie powiadamiające system o obecności nowych danych w buforze klawiatury. Jądro odnotowuje fakt wystąpienia przerwania i uruchamia odpowiednią procedurę obsługi. Procedura ta przetwarza dane i sygnalizuje kontrolerowi gotowość do przyjmowania kolejnych danych z klawiatury. Aby zapewnić odpowiednią synchronizację, jądro ma zwykle możliwość blokowania przerwań (zarówno wszystkich przerwań, jak i przerwań o wybranych numerach). W rzadko którym systemie operacyjnym procedury obsługi przerwań uruchamiane są w kontekście procesu — przeważnie procedury te wykonywane są w *kontekście przerwania* nieskojarzonym z żadnym konkretnym procesem. Kontekst ten istnieje wyłącznie w celu umożliwienia maksymalnie szybkiej reakcji na przerwanie.



Rysunek 1.1. *Zależności pomiędzy aplikacjami, jądrem i sprzętem*

Wymienione konteksty obejmują całość działalności jądra. W rzeczy samej, w przypadku Linuksa można uogólnić to omówienie i stwierdzić, że procesor może w dowolnym momencie realizować jedną z trzech czynności:

- wykonywać kod jądra w kontekście procesu (na rzecz procesu aplikacji użytkowej) w przestrzeni jądra;
- wykonywać kod procedury obsługi przerwania w przestrzeni jądra (w kontekście przerwania);
- wykonywać kod procesu aplikacji użytkowej w przestrzeni użytkownika.

Jądro Linuksa a jądro klasycznego systemu uniksowego

W wyniku wspólnego korzenia i implementowania identycznego interfejsu systemowego jądra współczesnych systemów uniksowych charakteryzują się one podobnymi cechami projektowymi. Z nielicznymi wyjątkami jądra systemów uniksowych to pakiety monolityczne i statyczne. Oznacza to istnienie sporych rozmiarów wykonywalnego obrazu jądra działającego w pojedynczej przestrzeni adresowej. Podstawowym wymaganiem systemów operacyjnych z rodziny Unix jest obecność w komputerze jednostki zarządzania pamięcią ze stronicowaniem; urządzenie to pozwala na wymuszenie przez system operacyjny ochrony systemowego obszaru pamięci i udostępnienie wirtualnej przestrzeni adresowej dla każdego z procesów systemu. Zagadnieniu projektu klasycznych jąder systemów z rodziny Unix poświęcono zresztą wiele osobnych książek.

Wysiłki Linusa Torvaldsa oraz innych programistów uczestniczących w rozwoju jądra systemu Linux skierowane były na umożliwienie ulepszenia architektury Linuksa bez odrzucania jego uniksowych korzeni (i, co ważniejsze, bez odrzucenia interfejsu programowego systemu Unix). W efekcie, ponieważ Linux nie opiera się na żadnym z istniejących Uniksów, Linus i inni mogli w dowolny sposób kształtować rozwiązania poszczególnych zadań projektowych, a w szczególności implementować w jądrze rozwiązania całkiem nowe. Powstałe tak różnice pomiędzy Linuksem a tradycyjnymi odmianami Uniksa to między innymi:

- Linux obsługuje dynamiczne ładowanie modułów jądra. Choć jądro systemu jest monolityczne, ma możliwość dynamicznego ładowania i usuwania z pamięci modułów kodu jądra.
- Linux obsługuje symetryczne przetwarzanie współbieżne (SMP). Współcześnie w obsługę tego trybu wyposażono również wiele komercyjnych odmian Uniksa, ale większość tradycyjnych implementacji systemu Unix jest go pozbawiona.
- Jądro systemu Linux obsługuje wywłaszczanie. W przeciwieństwie do tradycyjnych odmian systemu Unix jądro systemu Linux może wywłaszczyć zadanie, jeżeli działa ono w przestrzeni jądra. Z komercyjnych odmian systemu Unix wywłaszczanie zaimplementowano między innymi w systemach Solaris i IRIX.
- Linux w ciekawy sposób obsługuje wątki — planista nie rozróżnia wątków i procesów. Dla jądra wszystkie procesy są takie same — fakt współużytkowania przez niektóre z nich przestrzeni adresowej nie ma znaczenia.
- Implementacja Linuksa ignoruje te cechy systemu Unix, których implementacja jest powszechnie uznawana za niedomagającą, jak implementacja interfejsu STREAMS; Linux ignoruje też „martwe" standardy.
- Linux jest wolny w każdym znaczeniu tego słowa[5]. Zestaw funkcji zaimplementowany w systemie Linux jest owocem wolności Linuksa i wolnego modelu rozwoju oprogramowania. Funkcje nadmiarowe czy nieprzemyślane nie są implementowane w ogóle. Z drugiej strony wszystkie funkcje umieszczane w jądrze pojawiają się w nim w wyniku opracowywania rozwiązań konkretnych problemów, są rozważnie projektowane i elegancko implementowane. W wyniku takiego modelu rozwoju system operacyjny nie implementuje na przykład obecnego w innych odmianach Uniksa stronicowania pamięci jądra. Mimo to system Linux pozostaje spadkobiercą wszystkich najlepszych cech systemu Unix.

Oznaczenia wersji jądra Linuksa

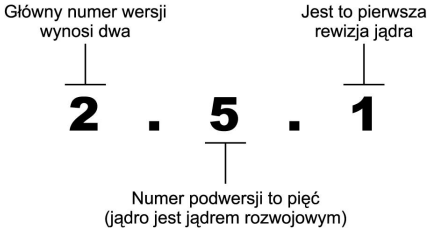
Jądra systemu Linux można podzielić na dwie kategorie — jądra stabilne i rozwojowe. Jądra stabilne to wydania przystosowane do szerokich zastosowań ogólnych. Nowe jądra stabilne publikowane są zwykle wyłącznie w obliczu pojawienia się w jądrze poprawek błędów lub sterowników nowych urządzeń. Tymczasem wersje rozwojowe

[5] Ale nie powolny — *przyp. tłum.*

podlegają niekiedy gwałtownym zmianom — są one wynikiem eksperymentów i nowych pomysłów programistów pracujących nad rozwojem jądra i bywają niekiedy bardzo drastyczne.

Do rozróżniania wersji jądra systemu Linux przyjęto prosty schemat nazewniczy (patrz rysunek 1.2). Mianowicie nazwa (numer) wersji jądra składa się z trzech liczb rozdzielanych kropkami. Pierwsza z tych liczb to główny numer wersji (ang. *major release*), liczba druga to numer podwersji (ang. *minor release*), a liczba trzecia to numer rewizji (ang. *revision number*). Rozróżnienie pomiędzy wersją stabilną a rozwojową możliwe jest na podstawie wartości numeru podwersji: liczby parzyste przypisywane są wersjom stabilnym, numery nieparzyste — wersjom rozwojowym. Na przykład, jądrem stabilnym może być jądro o numerze 2.6.0. Jądro to ma numer wersji dwa, numer podwersji równy sześć, a numer rewizji równy 0. Pierwsze dwie liczby w numerze wersji jądra opisują równocześnie rodzinę jąder — w tym przypadku chodzi o jądro z rodziny 2.6.

Rysunek 1.2. *Numeracja wydań jądra systemu Linux*



Jądra rozwojowe przechodzą szereg faz rozwoju. Początkowo, w wyniku ścierania się różnych pomysłów i koncepcji powstaje swego rodzaju chaos. Z czasem jądro dojrzewa i ostatecznie jest zamrażane — od momentu zamrożenia do jądra nie są dodawane żadne nowe funkcje. Od tej chwili prace mają na celu dopracowanie wprowadzonych funkcji. Kiedy jądro zostanie uznane za wystarczająco stabilne, ogłaszane jest zamrożenie kodu jądra. Od tego czasu akceptowane są jedynie modyfikacje wprowadzające poprawki zauważonych błędów. Wkrótce po zamrożeniu kodu jądro jest publikowane jako pierwsze wydanie nowej rodziny jąder stabilnych (przykładowo, jądro rozwojowe rodziny 2.5 jest stabilizowane do wersji 2.6).

Aktualny kod jądra systemu Linux można zawsze pobrać w postaci zarówno kompletnego archiwum kodu, jak i w postaci łat przyrostowych spod adresu *http://www.kernel.org*.



Instalowanie kodu źródłowego jądra do testów

Kod źródłowy jądra systemu Linux instalowany jest z reguły w katalogu */usr/src/linux*. Nie należy jednak wykorzystywać tej lokalizacji przy próbnych modyfikacjach kodu. Z tymi katalogami powiązana jest zwykle kompilacja biblioteki C dla jądra. Warto też na czas prac nad jądrem zrezygnować z uprawnień użytkownika uprzywilejowanego *root* — najlepiej eksperymentować na kodzie umieszczonym w katalogu domowym, korzystając z konta zwykłego użytkownika systemu, a uprawnienia użytkownika *root* przejmować jedynie na czas instalacji nowego jądra.

Niniejsza książka wykorzystuje jako bazę jądra ze stabilnej rodziny 2.6.

Społeczność programistów jądra systemu Linux

Rozpoczęcie przygody z programowaniem kodu źródłowego jądra oznacza wstąpienie w szeregi globalnej społeczności programistów jądra. Głównym forum wymiany myśli dla tej społeczności jest lista dystrybucyjna *linux-kernel*. Sposób uzyskania subskrypcji tej listy opisany jest pod adresem *http://vger.kernel.org*. Warto pamiętać o tym, że lista ta tętni wprost życiem i dziennie potrafi przyjąć ponad 300 wiadomości; poza tym pozostali subskrybenci listy — w tym ścisła czołówka projektantów jądra z Linusem Torvaldsem na czele — nie akceptują pomysłów nonsensownych. Niemniej jednak owa lista dystrybucyjna stanowi nieocenioną pomoc w procesie rozwoju własnej wersji jądra, gdyż za jej pośrednictwem łatwo o pozyskanie solidnych testerów, otrzymanie recenzji własnych pomysłów i na zadane pytania.

Proces rozwoju jądra zostanie omówiony szerzej w rozdziale 17. Pokazany zostanie w nim również sposób satysfakcjonującego uczestniczenia w społeczności programistów jądra.

Odmienność jądra

Jądro różni się bardzo od zwykłych aplikacji przestrzeni użytkownika, przez co programowanie jądra, choć niekoniecznie trudniejsze niż programowanie aplikacji użytkowych, stawia przed programistą szczególne wyzwania.

Owe różnice czynią jądro wyjątkowym. Programowanie jądra narusza niekiedy zasady przyjęte wśród programistów aplikacji. Niektóre z tych różnic są powszechnie uświadomione (wiadomo, że kod jądra nie podlega takim ograniczeniom jak kod aplikacji), inne nie są już tak oczywiste. Najważniejsze z różnic pomiędzy jądrem a aplikacjami można podsumować następująco:

- Jądro nie ma dostępu do biblioteki C.
- Jądro jest programowane w GNU C.
- Jądro nie podlega ochronie pamięci charakterystycznej dla przestrzeni użytkownika.
- Jądro nie może w prosty sposób realizować operacji zmiennoprzecinkowych.
- Jądro dysponuje niewielkim, ustalonym rozmiarem stosu.
- Jądro, przyjmując asynchroniczne przerwania, podlega wywłaszczaniu i obsługuje tryb SMP, kod jądra musi uwzględniać synchronizację i współbieżność zadań.
- W jądrze ważna jest maksymalna przenośność kodu.

Warto choćby pokrótce omówić wypunktowane różnice, ponieważ powinny one zadomowić się w świadomości każdego programisty jądra.

Brak biblioteki libc

W przeciwieństwie do aplikacji przestrzeni użytkownika kod jądra nie jest konsolidowany ze standardową biblioteką C (ani z żadną inną biblioteką). Ma to wiele przyczyn (obecność bibliotek byłaby między innymi źródłem problemów typu jajka i kury), ale pierwszorzędnymi są dbałość o szybkość działania i minimalny rozmiar jądra. Pełna biblioteka języka C (albo jakikolwiek większy jej podzbiór) jest zdecydowanie zbyt obszerna i zbyt mało wydajna, aby skutecznie wspomagać jądro.

Nie należy jednak rozpaczać, gdyż wiele funkcji charakterystycznych dla biblioteki libc zostało zaimplementowanych w ramach samego jądra. Na przykład, w pliku *lib/string.c* zdefiniowane zostały popularne funkcje manipulujące ciągami znakowymi. Aby z nich skorzystać wystarczy włączyć do kodu nagłówek `<linux/string.h>`. Warto zapamiętać, że wszędzie tam, gdzie w książce będzie mowa o plikach nagłówkowych, chodzić będzie o pliki wchodzące w skład kodu jądra. Do kodu jądra nie można bowiem włączać zewnętrznych plików nagłówkowych, gdyż jądro nie może korzystać z zewnętrznych bibliotek.

Najbardziej daje się we znaki brak funkcji `printf()`. Jądro nie ma dostępu do tej funkcji, udostępniając w zamian wywołanie `printk()`. Funkcja `printk()` kopiuje sformatowany ciąg do wewnętrznego bufora rejestru jądra odczytywanego przez program syslog. Składnia wywołania `printk()` jest zbliżona do tej znanej z funkcji `printf()`:

```
printk("Ahoj, przygodo! Ciąg: %s i liczba: %d\n", a_string, an_integer);
```

Jedna z większych różnic pomiędzy `printf()` i `printk()` polega na tym, że wywołanie `printk()` pozwala na określenie znacznika priorytetu komunikatu, Znacznik ten jest wykorzystywany przez syslog do określania miejsca przeznaczenia komunikatów. Oto przykład określenia priorytetu:

```
printk(KERN_ERR "błąd!\n");
```

Wywołanie `printk()` będzie szeroko wykorzystywane we wszystkich rozdziałach niniejszej książki. Jej szersze omówienie znajduje się natomiast w rozdziale 15.

GNU C

Jak każde szanujące się jądro systemu uniksopodobnego, jądro Linuksa jest programowane w języku C. Jednak, co może być zaskakujące, jądro Linuksa nie jest programowane w ścisłej zgodności ze standardem ANSI C. Tam bowiem, gdzie jest to zasadne, programiści jądra korzystają z rozmaitych rozszerzeń standardu dostępnych w kompilatorze *gcc* (nazwa *gcc* to skrót od *GNU Compiler Collection* i oznacza pakiet zawierający kompilator wykorzystywany do kompilacji jądra).

Programiści jądra wykorzystują rozszerzenia języka C zdefiniowane w standardzie ISO C99[6] oraz rozszerzenia GNU C. Rozszerzenia owe wiązały kod jądra z kompilatorem gcc, choć pojawiające się ostatnio kompilatory, w tym kompilator C firmy Intel, obsługują

[6] ISO C99 to ostatnia poważniejsza rewizja standardu ISO C. Standard C99 został znacznie rozszerzony w porównaniu z wersją C90, wprowadzając między innymi nazwane inicjalizatory struktur oraz typ `complex`.

rozszerzenia gcc w stopniu pozwalającym na wykorzystanie tych kompilatorów do kompilacji jądra. Rozszerzenia ISO C99 wykorzystywane w jądrze nie są żadną rewelacją, a ponieważ C99 to oficjalna rewizja języka C, rozszerzenia te stają się coraz powszechniejsze również w innych projektach. Najważniejszym i najciekawszym odchyleniem od standardu ANSI C w jądrze systemu Linux jest wykorzystywanie rozszerzeń GNU C. Niektóre z tych rozszerzeń, pojawiające się w kodzie źródłowym jądra, zostały omówione poniżej.

Funkcje rozwijane w miejscu wywołania

GNU C obsługuje *funkcje rozwijane w miejscu wywołania* (ang. *inline functions*). Kod ciała funkcji rozwijanej w miejscu wywołania jest, jak sugeruje nazwa rozszerzenia, wstawiany do kodu źródłowego zamiast znajdujących się w nim wywołań funkcji. Pozwala to na wyeliminowanie narzutów związanych z realizacją samego wywołania (a więc zachowywania na stosie zawartości rejestrów) i umożliwia potencjalnie lepszą optymalizację kodu, gdyż kompilator może optymalizować kod funkcji z uwzględnieniem otoczenia jej wywołania. Wadą wprowadzania do kodu funkcji rozwijanych w miejscu wywołania jest zwiększenie rozmiaru kodu, co oznacza zwiększenie zapotrzebowania na pamięć operacyjną i wymagania co do rozmiaru pamięci podręcznej instrukcji. Programiści jądra korzystają z funkcji rozwijanych w miejscu wywołania jedynie w niewielkich funkcjach, w których najważniejszy jest czas wykonania. Rozwijanie w miejscu wywołania większych funkcji, zwłaszcza jeżeli są one wywoływane wielokrotnie, a czas ich wykonania nie jest krytyczny dla działania jądra, nie jest dobrze widziane.

Funkcja rozwijana w miejscu wywołania deklarowana jest za pomocą słów kluczowych `static` i `inline` umieszczonych w definicji funkcji. Oto przykład:

```
static inline void dog(unsigned long tail_size)
```

Definicja funkcji rozwijanej w miejscu wywołania musi znajdować się przed miejscem pierwszego wywołania funkcji. Stąd przyjęło się umieszczać takie funkcje w plikach nagłówkowych. Oznaczenie funkcji słowem kluczowym `static` zapobiega tworzeniu nadmiarowych jednostek kompilacji. Jeżeli funkcja rozwijana w miejscu wywołania wykorzystywana jest tylko w jednym pliku kodu źródłowego jądra, może być równie dobrze umieszczona w tym pliku, w pobliżu jego początku.

W kodzie źródłowym jądra funkcje rozwijane w miejscu wywołania są preferowane przed spełniającymi podobną rolę skomplikowanymi makrodefinicjami.

Wstawki asemblerowe

Kompilator gcc umożliwia osadzanie w kodzie źródłowym funkcji bloków instrukcji asemblerowych. Ta możliwość jest rzecz jasna wykorzystywana jedynie w tych fragmentach jądra, których działanie jest uzależnione od architektury platformy sprzętowej.

Kod jądra systemu Linux jest mieszanką języka C i kodu asemblerowego, przy czym wstawki asemblerowe implementują niskopoziomowe funkcje jądra i te jego elementy, które powinny być wykonywane z maksymalną szybkością. Zdecydowana większość kodu jądra jest jednak pisana w języku C.

Opisywanie gałęzi wykonania kodu

Kompilator gcc udostępnia dyrektywę optymalizującą wykonanie gałęzi kodu, których wykonanie jest albo wielce, albo bardzo mało prawdopodobne. Kompilator wykorzystuje tę dyrektywę do odpowiedniej optymalizacji gałęzi warunkowej. W kodzie jądra dyrektywa ta jest obecna pod postacią wygodnych w użyciu makrodefinicji `likely()` i `unlikely()`.

Na przykład rozważmy następującą instrukcję warunkową:

```
if (foo) {
    /* ... */
}
```

Aby oznaczyć gałąź kodu jako taką, której wykonanie jest mało prawdopodobne, należy wykorzystać makrodefinicję `unlikely()`:

```
if (unlikely(foo)) {
    /* ... */
}
```

a gałęzie, których wykonanie jest niemal pewne, makrodefinicją `likely()`:

```
if (likely(foo)) {
    /* ... */
}
```

Dyrektywy optymalizujące gałęzie należy wykorzystywać jedynie w przypadkach, kiedy wybór jednego z wariantów kodu jest w większości przypadków znany z góry albo kiedy zachodzi potrzeba optymalizacji wykonania jednego z przypadków kosztem przypadków pozostałych. To bardzo ważne — dyrektywy optymalizujące powodują zwiększenie wydajności jedynie w przypadku prawidłowego przewidzenia dużej częstotliwości wykonywania danej gałęzi warunkowej — w przypadkach wyjątkowych, zaburzających przewidywanie, wykonanie kodu gałęzi jest opóźniane. Popularnym zastosowaniem makrodefinicji `likely()` i `unlikely()` są bloki kodu warunkowej obsługi błędów.

Brak mechanizmu ochrony pamięci

Kiedy aplikacja przestrzeni użytkownika próbuje odwołać się do niedozwolonego obszaru pamięci, jądro może ten fakt wykryć i zatrzymać błędny proces. Trudniej kontrolować próby odwołań do niedozwolonego obszaru pamięci z przestrzeni jądra. Błędy odwołań do pamięci w jądrze skutkują tzw. *kernel oops*, a więc ogólnym błędem jądra. Nie trzeba chyba Czytelnikowi przypominać, że nie wolno odwoływać się do niedozwolonych obszarów pamięci, wyłuskując wskaźnik o wartości `NULL` i realizując inne karkołomne operacje — w przestrzeni jądra ich skutki są bardzo poważne!

Dodatkowo pamięć jądra nie podlega stronicowaniu. Stąd każdy bajt pamięci zajmowanej przez jądro oznacza jeden bajt mniej pamięci operacyjnej dostępnej dla systemu. Warto o tym pamiętać, wyposażając jądro w kolejne nowe funkcje.

Niemożność (łatwego) korzystania z operacji zmiennoprzecinkowych

Kiedy proces przestrzeni użytkownika wykonuje instrukcje zmiennoprzecinkowe, jądro zarządza konwersją operandów całkowitych na operandy zmiennoprzecinkowe. Sposób konwersji zależny jest od architektury sprzętu.

W przestrzeni jądra trudno o luksus łatwej obsługi operacji zmiennoprzecinkowych. Korzystanie z takich operacji wewnątrz jądra wymaga między innymi ręcznego zapisywania i późniejszego odtwarzania rejestrów koprocesora. Krótko mówiąc, w kodzie jądra *nie należy* realizować operacji zmiennoprzecinkowych.

Ograniczony co do rozmiaru i stały stos

W przestrzeni użytkownika nie stanowi problemu alokacja na stosie całej masy zmiennych, z obszernymi strukturami i całymi wieloelementowymi tablicami włącznie. Jest to możliwe, ponieważ procesy przestrzeni użytkownika dysponują obszernymi pamięciami stosu, w razie potrzeby dynamicznie powiększanymi.

Stos jądra nie jest ani obszerny, ani dynamiczny — jego niewielki rozmiar jest stały w czasie działania jądra. Stos ten ma rozmiar 8 kB na platformach 32-bitowych oraz 16 kB na większości platform 64-bitowych.

Szersze omówienie stosu jądra znajduje się w podrozdziale „Statyczne przydziały na stosie” w rozdziale 10.

Synchronizacja i współbieżność

Jądro jest podatne na sytuacje hazardowe wynikające ze współbieżności operacji. W przeciwieństwie bowiem do jednowątkowej aplikacji przestrzeni użytkownika wiele funkcji jądra pozwala na współbieżny dostęp do zasobów, wymagając tym samym odpowiedniej synchronizacji eliminującej konflikty. W szczególności:

- Jądro systemu Linux obsługuje wieloprzetwarzanie. Bez odpowiedniej ochrony kod jądra wykonywany na dwóch lub więcej procesorach mógłby doprowadzić do równoczesnych prób dostępu do jednego zasobu.
- Jądro systemu Linux obsługuje asynchroniczne względem aktualnie wykonywanego kodu przerwania. Bez odpowiedniej ochrony mogłoby dojść do sytuacji, w której przerwanie pojawia się w środku operacji dostępu do współużytkowanego zasobu, do którego odwołuje się również procedura obsługi przerwania.
- Jądro systemu Linux podlega wywłaszczaniu, dlatego też bez odpowiedniej ochrony kod jądra mógłby zostać wywłaszczony na rzecz innego kodu odwołującego się do tego samego zasobu.

Typowymi mechanizmami eliminującymi hazard wynikający ze współbieżności są semafory i rygle pętlowe (ang. *spin lock*).

Obszerne omówienie zagadnień synchronizacji i współbieżności znajduje się w rozdziałach 7. oraz 8.

Znaczenie przenośności

Aplikacje użytkowe nie muszą być przenośne. Inaczej jest z samym systemem, który pomyślany został jako przenośny i taki powinien pozostać. Oznacza to, że kod jądra pisany w niezależnym sprzętowo języku C musi dać się poprawnie skompilować i uruchomić na wielu różnych platformach.

Zachowanie przenośności umożliwia szereg reguł, takich jak uniezależnienie kodu od porządku bajtów w słowie, utrzymywanie zgodności z architekturą 64-bitową, niezakładanie rozmiarów słów czy stron i tak dalej. Tematowi przenośności poświęcony został rozdział 16.

Kompilowanie jądra

Kompilacja jądra jest dość prosta. W rzeczy samej czynność ta jest o wiele prostsza niż kompilacja i instalacja innych niskopoziomowych elementów systemu operacyjnego, jak choćby instalacja biblioteki glibc. Jądra z rodziny 2.6 wprowadzają nowy system konfiguracji i kompilacji, które jeszcze bardziej upraszczają cały proces.

Przed przystąpieniem do kompilacji jądra należy je wcześniej odpowiednio skonfigurować. Ponieważ jądro oferuje zestaw niezliczonych funkcji i obsługuje całą masę rozmaitych urządzeń, jest co konfigurować. Opcje konfiguracji reprezentowane są symbolami rozpoczynającymi się ciągiem `CONFIG`, jak w opcji `CONFIG_PREEMPT`, która reprezentuje opcję wywłaszczania. Opcje konfiguracyjne mogą być dwu bądź trzywartościowe. Opcje dwuwartościowe przyjmują wartość włączona (ang. *yes*) albo wyłączona (ang. *no*). Funkcje właściwe jądra takie jak `CONFIG_PREEMPT` są zwykle konfigurowane jako opcje dwuwartościowe. Opcja trzywartościowa może przyjąć wartości włączona, wyłączona oraz moduł (ang. *module*). Ustawienie modułowe oznacza włączenie cechy reprezentowanej opcją, ale skompilowanie jej nie bezpośrednio do jądra, a jedynie do postaci ładowalnego modułu jądra. Do takiej postaci kompilowane są najczęściej sterowniki urządzeń.

Kod jądra obejmuje szereg narzędzi służących do automatyzacji procesu konfiguracji. Najprostszym z tych narzędzi jest konfigurator wiersza polecenia, uruchamiany poleceniem:

```
make config
```

Konfigurator ten wyświetla na konsoli serię pytań o wartości kolejnych opcji, pozwalając wybierać użytkownikowi spośród wartości włączona i wyłączona, ewentualnie (dla opcji trójwartościowych) wybrać kompilację do postaci modułu. Ponieważ określenie

wartości wszystkich kolejnych opcji jest *bardzo* czasochłonne, użytkownicy, których wynagrodzenie jest ustalane nie godzinowo, a akordowo, powinni skorzystać z wersji z interfejsem graficznym wykorzystującym bibliotekę ncurses, uruchamianej poleceniem:

```
make menuconfig
```

W środowisku graficznym X11 można uruchomić w pełni graficzny konfigurator:

```
make xconfig
```

Dwa ostatnie konfiguratory uwzględniają podział opcji konfiguracji na kategorie, na przykład cechy procesora (kategoria *Processor Features*) czy urządzenia sieciowe (*Network Devices*). Można poruszać się pomiędzy kategoriami, przeglądać opcje jądra i oczywiście zmieniać ich wartości.

Opcje konfiguracyjne przechowywane są w katalogu głównym drzewa katalogów kodu źródłowego jądra w pliku o nazwie *.config*. Niekiedy łatwiej jest przeprowadzić konfigurację jądra ręcznie, edytując ten plik bezpośrednio (sposób ten jest preferowany przez większość programistów). W pliku tym można dość łatwo odszukać i zmienić wartość dowolnej z opcji konfiguracyjnych. Po zakończeniu wprowadzania zmian (również po skopiowaniu posiadanego pliku konfiguracyjnego do katalogu nowego jądra) można skontrolować i zaktualizować konfigurację poleceniem:

```
make oldconfig
```

Zawsze warto uruchomić to polecenie przed rozpoczęciem kompilacji jądra. Po ustawieniu konfiguracji jądra można przystąpić do jego kompilowania:

```
make
```

W przeciwieństwie do jąder poprzedzających wersję 2.6 nie trzeba już wykonywać polecenia `make dep` przed kompilacją jądra — drzewo zależności modułów jest tworzone automatycznie. Nie trzeba też już, jak w poprzednich wersjach, określać konkretnego rodzaju kompilacji, jak np. *bzImage*. Wszystkim zajmuje się domyślna reguła pliku *Makefile*!

Aby zredukować natłok komunikatów towarzyszących kompilacji, pozostawiając sobie możliwość oglądania komunikatów ostrzeżeń i błędów, można posłużyć się sztuczką polegającą na przekierowaniu wyjścia programu `make`:

```
make > plik_komunikatow
```

Jeżeli kiedykolwiek po kompilacji zajdzie potrzeba przejrzenia towarzyszących jej komunikatów, wystarczy zajrzeć do wskazanego pliku. Ponieważ jednak przekierowanie komunikatów nie obejmuje komunikatów o błędach ani komunikatów ostrzeżeń, potrzeba taka będzie raczej rzadka.

Skompilowane jądro trzeba zainstalować. Sposób instalacji zależy od architektury sprzętu i konfiguracji programu ładującego — należy przejrzeć wytyczne stosowne dla obecnego w systemie programu ładującego odnośnie położenia obrazu jądra i sposobu uwzględnienia go w konfiguracji startowej. Warto przy tym uwzględnić w tej konfiguracji jądro, co do którego istnieje pewność, że działa poprawnie — umożliwi ono uruchomienie systemu w przypadku problemów z nowym jądrem.

Na przykład, na komputerze o architekturze x86 obsługiwanym przez program ładujący *grub* należy skopiować obraz jądra *arch/i386/boot/bzImage* do katalogu */boot* i zmodyfikować plik */etc/grub/grub.conf* tak, aby zawierał wpis reprezentujący nowe jądro.

Proces kompilacji obejmuje również utworzenie pliku *System.map* umieszczanego w katalogu głównym drzewa katalogów kodu źródłowego jądra. Plik ten zawiera tablicę odwzorowań symboli jądra do ich adresów. Plik ten wykorzystywany jest podczas diagnostyki, kiedy to służy do tłumaczenia adresów na nazwy funkcji i zmiennych.

Zanim zaczniemy

Niniejsza książka poświęcona jest jądru systemu Linux, a w szczególności jego działaniu, przeznaczeniu poszczególnych elementów i ich znaczeniu dla systemu operacyjnego. Omówienie obejmuje projekt i implementację kluczowych podsystemów jądra, jak również ich interfejsów i semantyki programowania. Książka jest podręcznikiem praktycznym i omówienia sposobów działania poszczególnych elementów pozbawione są zbytniego teoretyzowania. To interesujące podejście — w połączeniu z licznymi anegdotami i wskazówkami co do przerabiania jądra gwarantuje wdrożenie Czytelnika do praktycznego programowania jądra.

Mam nadzieję, że Czytelnik dysponuje dostępem do systemu Linux i ma kod źródłowy jego jądra. Najlepiej, aby był on użytkownikiem systemu Linux, który już ściągnął i „dłubie" w jądrze, oczekując pomocy w zrozumieniu znaczenia swoich prób. Równie dobrze jednak Czytelnik może nie znać Linuksa, a jedynie pałać chęcią poznania projektu jego jądra. Każdy jednak, kto chce napisać trochę własnego kodu jądra, musi wcześniej czy później wziąć się za analizę dostępnego kodu źródłowego. Jest on *darmowy* i warto to wykorzystać.

Życzę przyjemnej lektury.