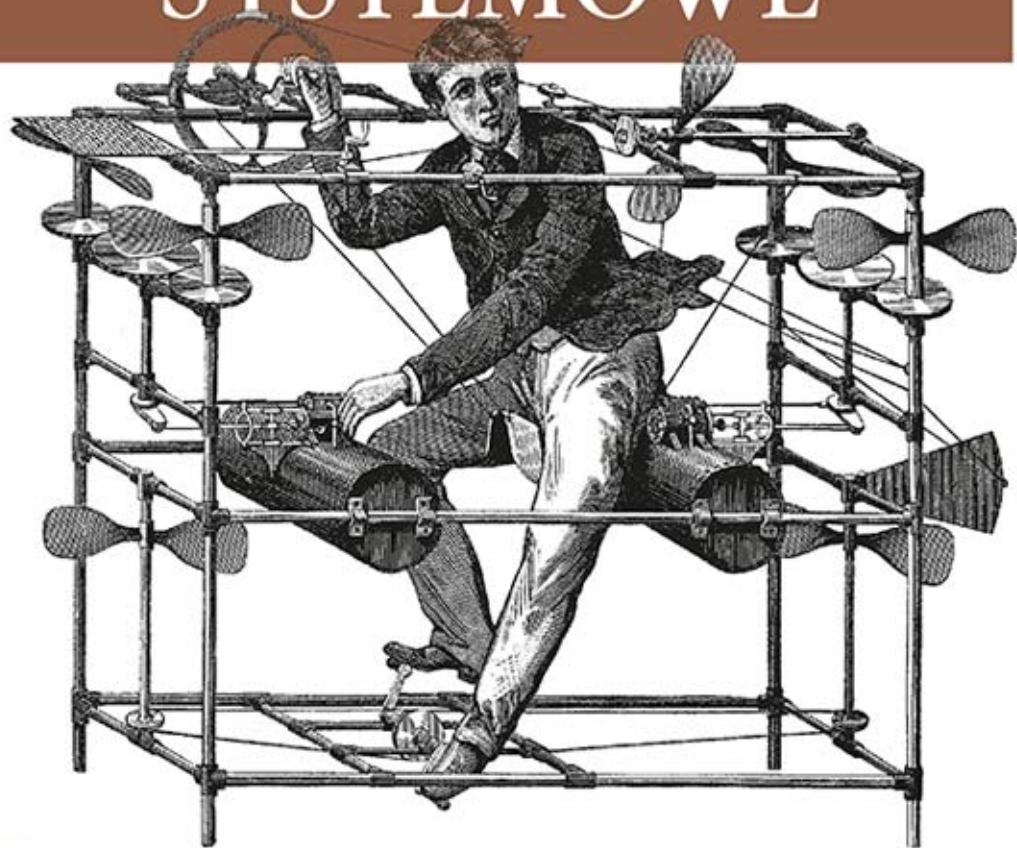


PRZEWODNIK PO JĄDRZE SYSTEMU LINUX!

Wydanie II

LINUX

PROGRAMOWANIE SYSTEMOWE



HELION

O'REILLY®

ROBERT LOVE

Tytuł oryginału: Linux System Programming: Talking Directly to the Kernel and C Library, 2nd Edition

Tłumaczenie: Jacek Janusz

ISBN: 978-83-246-8285-0

© 2014 Helion S.A.

Authorized Polish translation of the English edition Linux System Programming, 2nd Edition, ISBN 9781449339531 © 2013 Robert Love.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

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 the Publisher.

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 dołożyło 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/linps2>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Przedmowa	13
Wstęp	15
1. Wprowadzenie — podstawowe pojęcia	21
Programowanie systemowe	21
Dlaczego warto uczyć się programowania systemowego?	22
Kamienie węgielne programowania systemowego	23
Funkcje systemowe	23
Biblioteka języka C	24
Kompilator języka C	24
API i ABI	25
API	25
ABI	26
Standardy	27
Historia POSIX oraz SUS	27
Standardy języka C	28
Linux i standardy	28
Książka i standardy	29
Pojęcia dotyczące programowania w Linuksie	29
Pliki i system plików	30
Procesy	36
Użytkownicy i grupy	38
Uprawnienia	39
Sygnały	39
Komunikacja międzyprocesowa	40
Pliki nagłówkowe	40
Obsługa błędów	40
Początek programowania systemowego	43

2. Plikowe operacje wejścia i wyjścia	45
Otwieranie plików	46
Funkcja systemowa open()	46
Właściciele nowych plików	49
Uprawnienia nowych plików	49
Funkcja creat()	51
Wartości zwracane i kody błędów	52
Czytanie z pliku przy użyciu funkcji read()	52
Wartości zwracane	53
Czytanie wszystkich bajtów	54
Odczyty nieblokujące	55
Inne wartości błędów	55
Ograniczenia rozmiaru dla funkcji read()	56
Pisanie za pomocą funkcji write()	56
Zapisy częściowe	57
Tryb dopisywania	57
Zapisy nieblokujące	58
Inne kody błędów	58
Ograniczenia rozmiaru dla funkcji write()	59
Sposób działania funkcji write()	59
Zsynchronizowane operacje wejścia i wyjścia	60
Funkcje fsync() i fdatasync()	60
Funkcja sync()	62
Znacznik O_SYNC	63
Znaczniki O_DSYNC i O_RSYNC	63
Bezpośrednie operacje wejścia i wyjścia	64
Zamykanie plików	65
Kody błędów	65
Szukanie za pomocą funkcji lseek()	66
Szukanie poza końcem pliku	67
Kody błędów	67
Ograniczenia	68
Odczyty i zapisy pozycyjne	68
Kody błędów	69
Obcinanie plików	69
Zwielokrotnione operacje wejścia i wyjścia	70
Funkcja select()	71
Funkcja poll()	76
Porównanie funkcji poll() i select()	80
Organizacja wewnętrzna jądra	81
Wirtualny system plików	81
Bufor stron	82
Opóźniony zapis stron	84
Zakończenie	85

3. Buforowane operacje wejścia i wyjścia	87
Operacje wejścia i wyjścia buforowane w przestrzeni użytkownika	87
Rozmiar bloku	89
Typowe operacje wejścia i wyjścia	90
Wskaźniki do plików	90
Otwieranie plików	91
Tryby	91
Otwieranie strumienia poprzez deskryptor pliku	92
Zamykanie strumieni	93
Zamykanie wszystkich strumieni	93
Czytanie ze strumienia	93
Czytanie pojedynczego znaku	93
Czytanie całego wiersza	94
Czytanie danych binarnych	96
Pisanie do strumienia	97
Zapisywanie pojedynczego znaku	97
Zapisywanie łańcucha znaków	98
Zapisywanie danych binarnych	98
Przykładowy program używający buforowanych operacji wejścia i wyjścia	99
Szukanie w strumieniu	100
Otrzymywanie informacji o aktualnym położeniu w strumieniu	101
Opróżnianie strumienia	102
Błędy i koniec pliku	102
Otrzymywanie skojarzonego deskryptora pliku	103
Parametry buforowania	104
Bezpieczeństwo wątków	105
Nieautomatyczne blokowanie plików	106
Nieblokowane operacje na strumieniu	107
Krytyczna analiza biblioteki typowych operacji wejścia i wyjścia	108
Zakończenie	109
4. Zaawansowane operacje plikowe wejścia i wyjścia	111
Rozproszone operacje wejścia i wyjścia	112
Funkcje readv() i writev()	112
Odpytywanie zdarzeń	117
Tworzenie nowego egzemplarza interfejsu odpytywania zdarzeń	117
Sterowanie działaniem interfejsu odpytywania zdarzeń	118
Oczekiwanie na zdarzenie w interfejsie odpytywania zdarzeń	121
Zdarzenia przełączane z boczem a zdarzenia przełączane poziomem	122
Odwzorowywanie plików w pamięci	123
Funkcja mmap()	123
Funkcja munmap()	128

Przykład odwzorowania w pamięci	128
Zalety używania funkcji mmap()	130
Wady używania funkcji mmap()	130
Zmiana rozmiaru odwzorowania	131
Zmiana uprawnień odwzorowania	132
Synchronizacja odwzorowanego pliku	133
Dostarczanie porad dotyczących odwzorowania w pamięci	134
Porady dla standardowych operacji plikowych wejścia i wyjścia	137
Funkcja systemowa posix_fadvise()	137
Funkcja systemowa readahead()	138
Porada jest tania	139
Operacje zsynchronizowane, synchroniczne i asynchroniczne	140
Asynchroniczne operacje wejścia i wyjścia	141
Zarządcy operacji wejścia i wyjścia oraz wydajność operacji wejścia i wyjścia	142
Adresowanie dysku	142
Działanie zarządcy operacji wejścia i wyjścia	143
Wspomaganie odczytów	143
Wybór i konfiguracja zarządcy operacji wejścia i wyjścia	147
Optymalizowanie wydajności operacji wejścia i wyjścia	148
Zakończenie	154
5. Zarządzanie procesami	155
Programy, procesy i wątki	155
Identyfikator procesu	156
Przydział identyfikatorów procesów	156
Hierarchia procesów	157
Typ pid_t	157
Otrzymywanie identyfikatora procesu oraz identyfikatora procesu rodzicielskiego	158
Uruchamianie nowego procesu	158
Rodzina funkcji exec	158
Funkcja systemowa fork()	162
Zakończenie procesu	166
Inne sposoby na zakończenie procesu	167
Funkcja atexit()	167
Funkcja on_exit()	168
Sygnał SIGCHLD	169
Oczekiwanie na zakończone procesy potomka	169
Oczekiwanie na określony proces	171
Jeszcze wszechstronniejsza funkcja oczekiwania	173
BSD wkracza do akcji: funkcje wait3() i wait4()	175
Uruchamianie i oczekiwanie na nowy proces	177
Procesy zombie	179

Użytkownicy i grupy	179
Rzeczywiste, efektywne oraz zapisane identyfikatory użytkownika i grupy	180
Zmiana rzeczywistego lub zapisanego identyfikatora dla użytkownika lub grupy	181
Zmiana efektywnego identyfikatora dla użytkownika lub grupy	182
Zmiana identyfikatora dla użytkownika lub grupy w wersji BSD	182
Zmiana identyfikatora dla użytkownika lub grupy w wersji HP-UX	183
Zalecane modyfikacje identyfikatorów użytkownika i grupy	183
Wsparcie dla zapisanych identyfikatorów użytkownika	184
Otrzymywanie identyfikatorów użytkownika i grupy	184
Grupy sesji i procesów	184
Funkcje systemowe do obsługi sesji	186
Funkcje systemowe do obsługi grup procesów	187
Przestarzałe funkcje do obsługi grupy procesów	188
Demony	189
Zakończenie	191
6. Zaawansowane zarządzanie procesami	193
Szeregowanie procesów	193
Przedziały czasowe	194
Procesy związane z wejściem i wyjściem a procesy związane z procesorem	194
Szeregowanie z wywłaszczaniem	195
Completely Fair Scheduler	196
Udostępnianie czasu procesora	197
Prawidłowe sposoby użycia sched_yield()	198
Priorytety procesu	199
nice()	199
getpriority() i setpriority()	200
Priorytety wejścia i wyjścia	201
Wiązanie procesów do konkretnego procesora	202
sched_getaffinity() i sched_setaffinity()	203
Systemy czasu rzeczywistego	205
Systemy ścisłego oraz zwykłego czasu rzeczywistego	205
Opóźnienie, rozsynchronizowanie oraz parametry graniczne	206
Obsługa czasu rzeczywistego przez system Linux	207
Linuksowe strategie szeregowania i ustalania priorytetów	208
Ustawianie parametrów szeregowania	211
sched_rr_get_interval()	214
Środki ostrożności przy pracy z procesami czasu rzeczywistego	215
Determinizm	216
Ograniczenia zasobów systemowych	218
Ograniczenia	220
Ustawianie i odczytywanie ograniczeń	223

7. Wątkowość	225
Binaria, procesy i wątki	225
Wielowątkowość	226
Koszty wielowątkowości	228
Alternatywy dla wielowątkowości	228
Modele wątkowości	229
Wątkowość na poziomie użytkownika	229
Wątkowość mieszana	230
Współprogramy i włókna	230
Wzorce wątkowości	231
Wątkowość thread-per-connection	231
Wątkowość sterowana zdarzeniami	232
Współbieżność, równoległość i wyścigi	233
Sytuacje wyścigów	233
Synchronizacja	236
Muteksy	236
Zakleszczenia	238
Standard Pthreads	239
Implementacje wątkowości w Linuksie	240
Interfejs programistyczny dla standardu Pthreads	240
Konsolidowanie implementacji Pthreads	241
Tworzenie wątków	241
Identyfikatory wątków	243
Kończenie wątków	243
Łączenie i odłączanie wątków	246
Przykład wątkowości	247
Muteksy standardu Pthreads	248
Dalsze zdobywanie wiedzy	251
8. Zarządzanie plikami i katalogami	253
Pliki i ich metadane	253
Rodzina funkcji stat	253
Uprawnienia	257
Prawa własności	259
Atrybuty rozszerzone	261
Operacje dla atrybutów rozszerzonych	264
Katalogi	269
Aktualny katalog roboczy	270
Tworzenie katalogów	275
Usuwanie katalogów	276
Odczytywanie zawartości katalogu	278

Dowiązania	280
Dowiązania twarde	281
Dowiązania symboliczne	282
Usuwanie elementów z systemu plików	284
Kopiowanie i przenoszenie plików	286
Kopiowanie	286
Przenoszenie	286
Węzły urządzeń	288
Specjalne węzły urządzeń	289
Generator liczb losowych	289
Komunikacja poza kolejką	290
Śledzenie zdarzeń związanych z plikami	292
Inicjalizacja interfejsu inotify	292
Elementy obserwowane	293
Zdarzenia interfejsu inotify	295
Zaawansowane opcje obserwowania	298
Usuwanie elementu obserwowanego z interfejsu inotify	299
Otrzymywanie rozmiaru kolejki zdarzeń	300
Usuwanie egzemplarza interfejsu inotify	300
9. Zarządzanie pamięcią	301
Przestrzeń adresowa procesu	301
Strony i stronicowanie	301
Regiony pamięci	303
Przydzielanie pamięci dynamicznej	304
Przydzielanie pamięci dla tablic	306
Zmiana wielkości obszaru przydzielonej pamięci	307
Zwalnianie pamięci dynamicznej	309
Wyrównanie	310
Zarządzanie segmentem danych	315
Anonimowe odwzorowania w pamięci	315
Tworzenie anonimowych odwzorowań w pamięci	317
Odwzorowanie pliku /dev/zero	318
Zaawansowane operacje przydziału pamięci	319
Dokładne dostrajanie przy użyciu funkcji malloc_usable_size() oraz malloc_trim()	322
Uruchamianie programów używających systemu przydzielania pamięci	323
Otrzymywanie danych statystycznych	323
Przydziały pamięci wykorzystujące stos	324
Powielanie łańcuchów znakowych na stosie	326
Tablice o zmiennej długości	326
Wybór mechanizmu przydzielania pamięci	327

Operacje na pamięci	328
Ustawianie wartości bajtów	329
Porównywanie bajtów	329
Przenoszenie bajtów	330
Wyszukiwanie bajtów	331
Manipulowanie bajtami	332
Blokowanie pamięci	332
Blokowanie fragmentu przestrzeni adresowej	333
Blokowanie całej przestrzeni adresowej	334
Odblokowywanie pamięci	335
Ograniczenia blokowania	335
Czy strona znajduje się w pamięci fizycznej?	336
Przydział oportunistyczny	336
Przekroczenie zakresu zatwierdzenia oraz stan braku pamięci (OOM)	337
10. Sygnały	339
Koncepcja sygnałów	340
Identyfikatory sygnałów	340
Sygnały wspierane przez system Linux	341
Podstawowe zarządzanie sygnałami	346
Oczekiwanie na dowolny sygnał	347
Przykłady	348
Uruchomienie i dziedziczenie	349
Odzworowanie numerów sygnałów na łańcuchy znakowe	350
Wysyłanie sygnału	351
Uprawnienia	352
Przykłady	352
Wysyłanie sygnału do samego siebie	353
Wysyłanie sygnału do całej grupy procesów	353
Współużywalność	354
Funkcje, dla których współużywalność jest zagwarantowana	354
Zbiory sygnałów	356
Inne funkcje obsługujące zbiory sygnałów	356
Blokowanie sygnałów	357
Odzyskiwanie oczekujących sygnałów	358
Oczekiwanie na zbiór sygnałów	358
Zaawansowane zarządzanie sygnałami	359
Struktura siginfo_t	361
Wspaniały świat pola si_code	363
Wysyłanie sygnału z wykorzystaniem pola użytkowego	366
Przykład wykorzystania pola użytkowego	367
Ułomność systemu Unix?	367

11. Czas	369
Struktury danych reprezentujące czas	371
Reprezentacja pierwotna	372
Następna wersja — dokładność na poziomie mikrosekund	372
Kolejna, lepsza wersja — dokładność na poziomie nanosekund	372
Wyłuskiwanie składników czasu	373
Typ danych dla czasu procesu	374
Zegary POSIX	374
Rozdzielczość źródła czasu	375
Pobieranie aktualnego czasu	376
Lepszy interfejs	377
Interfejs zaawansowany	377
Pobieranie czasu procesu	378
Ustawianie aktualnego czasu	379
Precyzyjne ustawianie czasu	379
Zaawansowany interfejs ustawiania czasu	380
Konwersje czasu	381
Dostrajanie zegara systemowego	382
Stan uśpienia i oczekiwania	385
Obsługa stanu uśpienia z dokładnością do mikrosekund	386
Obsługa stanu uśpienia z dokładnością do nanosekund	387
Zaawansowane zarządzanie stanem uśpienia	389
Przenośny sposób wprowadzania w stan uśpienia	390
Przepełnienia	391
Alternatywy stanu uśpienia	391
Liczniki	392
Proste alarmy	392
Liczniki interwałowe	392
Liczniki zaawansowane	394
A Rozszerzenia kompilatora GCC dla języka C	401
B Bibliografia	413
Skorowidz	417

Zaawansowane zarządzanie procesami

W rozdziale 5. wyjaśniono, czym jest proces i jakich elementów systemu dotyczy. Omówiono także interfejsy jądra użyte w celu jego tworzenia, kontroli i usuwania. W niniejszym rozdziale informacje te są wykorzystane podczas rozważań na temat *zarządcy procesów* (ang. *scheduler*) i jego algorytmu szeregowania, by zaprezentować zaawansowane interfejsy dla potrzeb zarządzania procesami. Te funkcje systemowe ustalają zachowanie szeregowania oraz semantykę procesu, wpływając na sposób działania zarządcy procesów w dążeniu do celu określonego przez aplikację lub użytkownika.

Szeregowanie procesów

Zarządca procesów jest składnikiem jądra, który dzieli ograniczony zasób czasu procesora pomiędzy procesy systemowe. Innymi słowy, zarządca procesów (lub mówiąc prościej: *zarządca*) jest podsystemem jądra, który decyduje, jaki proces powinien zostać uruchomiony w następnej kolejności. Podczas podejmowania decyzji, jakie procesy i kiedy mają być uruchomione, zarządca jest odpowiedzialny za maksymalizowanie użycia procesora oraz stwarzanie jednocześnie wrażenia, iż wiele procesów jest wykonywanych współbieżnie i płynnie.

W tym rozdziale będzie mowa o procesach uruchamialnych (ang. *runnable*). Proces uruchamialny to taki proces, który nie jest zablokowany. Proces *blokowany* (ang. *blocked*) jest to taki proces, który znajduje się w stanie uśpienia, czekając, aż jądro rozpocznie wykonywanie operacji wejścia i wyjścia. Procesy współdziałające z użytkownikami, intensywnie zapisujące lub czytające pliki, a także odpowiadające na sieciowe, mają tendencję do zużywania dużej ilości czasu, kiedy są zablokowane podczas oczekiwania na zasoby, które mają stać się dostępne. Nie są one aktywne w czasie tych długich okresów bezczynności. W przypadku tylko jednego procesu uruchamialnego zadanie zarządcy procesów jest trywialne: uruchomić ten właśnie proces! Zarządca udowadnia swoją wartość wówczas, gdy w systemie znajduje się więcej procesów uruchamialnych niż procesorów. W takiej sytuacji niektóre procesy będą działać, a inne muszą czekać na swoją kolej. Zarządca procesów bierze przede wszystkim odpowiedzialność za decyzję, jaki proces, kiedy i na jak długi czas powinien zostać uruchomiony.

System operacyjny zainstalowany w maszynie posiadającej pojedynczy procesor jest *wielozadaniowy*, gdy potrafi na przemian wykonywać wiele procesów, stwarzając złudzenie, iż więcej niż jeden proces działa w tym samym czasie. W maszynach wieloprocessorowych wielozadaniowy system operacyjny pozwala procesom faktycznie działać równolegle na różnych procesorach. System operacyjny, który nie jest wielozadaniowy, taki jak DOS, może uruchamiać wyłącznie jeden program w danym momencie.

Wielozadaniowe systemy operacyjne dzielą się na systemy *kooperacyjne* (ang. *cooperative*) oraz systemy z *wywłaszczeniem* (ang. *preemptive*). Linux implementuje drugą formę wielozadaniowości, w której zarządca decyduje, kiedy należy zatrzymać dany proces i uruchomić inny. Wywłaszczenie zwane jest też zawieszeniem działającego procesu. Jeszcze raz powtórzmy — długość czasu, w którym proces działa, zanim zarządca go wywłaszczy, nazywany jest *przedziałem czasowym* procesu (dosłowne tłumaczenie: „plasterek czasu” — *timeslice* — nazywany tak z powodu przydzielania przez zarządcę każdemu procesowi uruchamialnemu „plasterka” czasu procesora).

W wielozadaniowości kooperatywnej jest odwrotnie — proces nie przestaje działać, dopóki sam nie zdecyduje o swoim zawieszeniu. Samoczynne zawieszenie się procesu nazywa się *udostępnianiem czasu procesora* (ang. *yielding*). W sytuacji idealnej procesy często udostępniają czas procesora, lecz system operacyjny jest niezdolny do wymuszenia tego zachowania. Źle działający lub uszkodzony program może działać tak długo, że potrafi wyeliminować iluzję wielozadaniowości, a nawet zawiesić cały system. Z powodu tych problemów, związanych z zagadnieniem wielozadaniowości, nowoczesne systemy operacyjne są generalnie wielozadaniowe z wywłaszczeniem; Linux nie jest tu wyjątkiem.

Zarządca procesów Linuksa zmienił się na przestrzeni lat. Bieżącym zarządcą procesów, dostępnym od wersji 2.6.23 jądra Linuksa, jest *Completely Fair Scheduler* (w skrócie *CFS*). Jego nazwa pochodzi od słów *fair queuing* (*sprawiedliwe kolejkowanie*) — nazwy algorytmu kolejkowania, który stara się sprawiedliwie udostępniać zasoby rywalizującym ze sobą konsumentom. CFS różni się znacząco od innych uniksowych zarządców procesów, włącznie z jego poprzednikiem, *zarządcą procesów O(1)*. Zarządcę CFS omówimy dokładniej w podrozdziale „Completely Fair Scheduler”, znajdującym się w dalszej części tego rozdziału.

Przedziały czasowe

Przedział czasowy, który zarządca procesów przydziela każdemu procesowi, jest ważną wartością dla ogólnego zachowania i sprawności systemu. Jeśli przedziały czasowe są zbyt duże, procesy muszą oczekiwać przez długi czas pomiędzy uruchomieniami, pogarszając wrażenie równoległego działania. Może to być frustrujące dla użytkownika w przypadku zauważalnych opóźnień. I na odwrót, jeśli przedziały czasowe są zbyt małe, znacząca ilość czasu systemowego jest przeznaczana na przełączanie między aplikacjami i traci się na przykład czasową lokalizację (ang. *temporal locality*) i inne korzyści.

Ustalanie idealnego przedziału czasowego nie jest więc proste. Niektóre systemy operacyjne udzielają procesom dużych przedziałów czasowych, mając nadzieję na zwiększenie przepustowości systemu oraz ogólnej wydajności. Inne systemy operacyjne dają procesom bardzo małe przedziały czasowe w nadziei, iż zapewnią systemowi bardzo dobrą wydajność. Jak się okaże, zarządca CFS rozwiązuje ten problem w bardzo dziwny sposób: eliminuje przedziały czasowe.

Procesy związane z wejściem i wyjściem a procesy związane z procesorem

Procesy, które w ciągły sposób konsumują wszystkie możliwe przedziały czasowe im przydzielone, są określane mianem procesów *związanych z procesorem* (ang. *processor-bound*). Takie procesy ciągle żądają czasu procesora i będą konsumować wszystko, co zarządca im przydzieli. Najprostszym, trywialnym tego przykładem jest pętla nieskończona:

```
// 100% związanie z procesorem
while (1)
;
```

Inne, mniej ekstremalne przykłady to obliczenia naukowe, matematyczne oraz przetwarzanie obrazów.

Z drugiej strony, procesy, które przez większość czasu są zablokowane w oczekiwaniu na jakiś zasób, zamiast normalnie działać, nazywane są procesami *związanymi z wejściem i wyjściem* (ang. *I/O-bound*). Procesy związane z wejściem i wyjściem są często wznawiane i oczekują w plikowych lub sieciowych operacjach zapisu i odczytu, blokują się podczas oczekiwania na dane z klawiatury lub czekają na akcję użytkownika polegającą na ruchu myszką. Przykłady aplikacji związanych z wejściem i wyjściem to programy użytkowe, które robią niewiele poza tym, że generują wywołania systemowe żądające od jądra, aby wykonał operacje wejścia i wyjścia, takie jak `cp` lub `mv`, a także wiele aplikacji GUI, które zużywają dużo czasu oczekując na akcję użytkownika.

Aplikacje związane z procesorem oraz aplikacje związane z wejściem i wyjściem chcą wykorzystywać takie opcje zarządcy procesów, które najbardziej odpowiadają ich sposobowi działania. Aplikacje związane z procesorem wymagają możliwie największych przedziałów czasowych, pozwalających im na poprawienie współczynnika używania pamięci podręcznej (poprzez czasową lokalizację) oraz jak najszybciej kończą swoje działanie. W przeciwieństwie do nich procesy związane z wejściem i wyjściem nie wymagają koniecznie dużych przedziałów czasowych, ponieważ standardowo działają tylko przez krótki czas przed wysłaniem żądań związanych z wejściem i wyjściem oraz blokowaniem się na jakimś zasobie z jądra systemu. Procesy związane z wejściem i wyjściem czerpią jednak korzyści z tego, iż zarządca obsługuje je z wyższym priorytetem. Im szybciej jakaś aplikacja może ponownie uruchomić się po zablokowaniu się i wysłaniu większej liczby żądań wejścia i wyjścia, tym lepiej potrafi ona używać urządzeń systemowych. Co więcej, im szybciej aplikacja czekająca na akcję użytkownika zostanie zaszeregowana, tym bardziej sprawia ona wrażenie płynnego działania dla tego użytkownika.

Dopasowywanie potrzeb procesów związanych z procesorem oraz wejściem i wyjściem nie jest łatwe. W rzeczywistości większość aplikacji stanowi połączenie procesów związanych z procesorem oraz z wejściem i wyjściem. Kodowanie oraz dekodowanie strumieni dźwięku i obrazu jest dobrym przykładem typu aplikacji, który opiera się jednoznacznej kwalifikacji. Wiele gier to również aplikacje o typie mieszanym. Nie zawsze jest możliwa identyfikacja skłonności danej aplikacji; w określonym momencie dany proces może zachowywać się w różny sposób.

Szeregowanie z wyłłaszczaniem

W tradycyjnej metodzie szeregowania procesów w Unikse wszystkim uruchamialnym procesom zostaje przydzielony przedział czasu. Gdy dany proces wykorzysta swój przedział czasowy, jądro zawiesza go, a rozpoczyna wykonywanie innego procesu. Jeśli w systemie nie istnieje więcej procesów uruchamialnych, jądro pobiera grupę procesów z wykorzystanymi przedziałami czasowymi, uzupełnia te przedziały oraz uruchamia procesy ponownie. Cały proces jest powtarzalny: procesy wciąż tworzone pojawiają się na liście procesów uruchamialnych, a procesy kończone są z niej usuwane, blokują się na operacjach wejścia i wyjścia lub są budzone ze stanu uśpienia. W ten sposób wszystkie procesy ostatecznie są uruchamiane, nawet jeśli w systemie istnieją procesy o wyższym priorytecie; procesy niskopriorytetowe muszą czekać, aż procesy

o wysokim priorytecie wyczerpią swoje przedziały czasowe lub zablokują się. To zachowanie formułuje ważną, lecz ukrytą regułę szeregowania w systemach Unix: wszystkie procesy muszą kontynuować swoje działanie.

Completely Fair Scheduler

Zarządca Completely Fair Scheduler (CFS) znacząco różni się od tradycyjnych zarządców procesów spotykanych w systemie Unix. W większości systemów uniksowych, włącznie z Linuksem przed wprowadzeniem CFS, istniały dwie podstawowe zmienne, związane z procesem podczas jego szeregowania: priorytet i przedział czasu. Jak opisano we wcześniejszym podrozdziale w przypadku tradycyjnych zarządców procesom przypisuje się przedziały czasowe, które reprezentują „plasterek” procesora przydzielony dla nich. Procesory mogą działać aż do momentu, gdy wyczerpią ten przedział czasowy. W podobny sposób procesom przypisuje się priorytet. Zarządca uruchamia procesy o wyższym priorytecie przed tymi, które mają niższy. Algorytm ten jest bardzo prosty i sprawdzał się znakomicie w dawnych systemach uniksowych, które wymagały podziału czasu procesora. Nieco gorzej działa on w systemach wymagających wysokiej wydajności interaktywnej oraz sprawiedliwego podziału czasu, takich jak nowoczesne komputery stacjonarne i urządzenia mobilne.

W zarządcy CFS użyto całkiem innego algorytmu, zwanego *sprawiedliwym kolejkowaniem* (ang. *fair scheduling*), który eliminuje przedziały czasu jako jednostki dostępu do procesora. Zamiast nich CFS przypisuje każdemu procesowi część czasu procesora. Algorytm jest prosty: CFS rozpoczyna swoje działanie poprzez przypisanie N procesom po $1/N$ czasu procesora. Następnie modyfikuje to przypisanie poprzez przydzielanie każdemu procesowi odpowiedniej wagi, związanej z jego poziomem uprzejmości. Procesy z domyślną wartością poziomu uprzejmości równą zero mają wagę jeden, dlatego ich proporcja nie zostaje zmieniona. Procesy z mniejszą wartością poziomu uprzejmości (wyższym priorytetem) uzyskują wyższą wagę, wskutek czego zwiększa się ich udział w wykorzystaniu czasu procesora, podczas gdy procesy z większą wartością poziomu uprzejmości (niższym priorytetem) otrzymują niższą wagę, a więc zmniejsza się ich udział w wykorzystaniu czasu procesora.

Zarządca CFS używa obecnie ważonej proporcji czasu procesora przypisanego do każdego procesu. Aby ustalić faktyczną długość odcinka czasu działania dla każdego procesu, CFS musi rozdzielić proporcje w określonym przedziale. Ten przedział jest zwany *docelowym opóźnieniem* (ang. *target latency*), ponieważ reprezentuje opóźnienie szeregowania w systemie. Aby zrozumieć działanie docelowego opóźnienia, załóżmy, że zostało ono zdefiniowane jako 20 milisekund, a w systemie znajdują się dwa uruchamialne procesy o tym samym priorytecie. Wynika z tego, że każdy proces ma tę samą wagę i przypisano mu taką samą część czasu procesora, czyli 10 milisekund. Tak więc CFS uruchomi pierwszy proces na 10 milisekund, drugi również na 10 milisekund, a następnie powtórzy całą operację. Jeśli w systemie byłoby 5 procesów uruchamialnych, CFS przydzielałby im po 4 milisekundy.

Do tej pory wszystko wydaje się proste. Co się jednak stanie, gdy będziemy mieli na przykład 200 procesów? Z docelowym opóźnieniem wynoszącym 20 milisekund CFS będzie mógł uruchamiać każdy z tych procesów na jedynie 100 mikrosekund. Z powodu kosztów przełączania kontekstu (zwanym po prostu *kosztami przełączania*) pomiędzy procesorami oraz ograniczonej czasowej lokalizacji wydajność systemu może się pogorszyć. Aby rozwiązać ten problem,

CFS wprowadza drugą kluczową zmienną: minimalną ziarnistość. *Minimalna ziarnistość* (ang. *minimum granularity*) jest najmniejszym przedziałem czasu, w jakim może działać dowolny proces. Wszystkie procesy, bez względu na przydzieloną im część czasu procesora, będą działać z przynajmniej minimalną ziarnistością (lub do czasu ich zablokowania). Dzięki temu przełączanie nie pochłania nadmiernej ilości całkowitego czasu systemu, dzieje się to jednak kosztem wartości docelowego opóźnienia. Oznacza to, że w przypadku gdy zaczyna być aktywna minimalna ziarnistość, zostaje naruszona zasada sprawiedliwego przydzielania czasu. Jeśli użyte zostaną typowe wartości docelowego opóźnienia i minimalna ziarnistość oraz najczęściej spotykana rozsądna liczba procesów uruchamialnych, minimalna ziarnistość nie zostaje aktywowana, docelowe opóźnienie jest zachowywane, a zasada sprawiedliwego przydzielania czasu realizowana.

Poprzez przydzielanie części procesora, a nie stałych przedziałów czasowych, CFS może stosować zasadę sprawiedliwości: każdy proces otrzymuje *sprawiedliwą część* procesora. Ponadto CFS potrafi używać skonfigurowanego opóźnienia szeregowania, ponieważ *docelowe opóźnienie* może być modyfikowane przez użytkownika. W tradycyjnych zarządzaczach uniksowych procesy z *założenia* działają w stałych przedziałach czasowych, lecz opóźnienie szeregowania (określające, jak często są one uruchamiane) jest nieznaną. W przypadku CFS procesy działają zgodnie z przydzielonymi im częściami i z opóźnieniem, a te wartości z *założenia* są znane. Jednakże przedział czasowy zmienia się, ponieważ jest funkcją liczby uruchamialnych procesów w systemie. Jest to znacząco inna metoda obsługi szeregowania procesów. Rozwiązuje ona wiele problemów dotyczących procesów interaktywnych oraz związanych z wejściem i wyjściem, nękających tradycyjnych zarządzaczy procesów.

Udostępnianie czasu procesora

Chociaż Linux jest wielozadaniowym systemem operacyjnym z wywłaszczaniem, dostarcza również funkcję systemową, która pozwala procesom jawnie zawieszać działanie i informować zarządzacę, by wybrał nowy proces do uruchomienia:

```
#include <sched.h>
int sched_yield (void);
```

Wywołanie funkcji `sched_yield()` skutkuje zawieszeniem aktualnie działającego procesu, po czym zarządzca procesów wybiera nowy proces, aby go uruchomić, w taki sam sposób, jakby jądro samo wywłaszczyło aktualny proces w celu uruchomienia nowego. Warto uwagi jest to, że jeśli nie istnieje żaden inny proces uruchamialny, co jest częstym przypadkiem, wówczas proces udostępniający natychmiast ponowi swoje działanie. Z powodu przypadkowości połączonej z ogólnym przekonaniem, iż zazwyczaj istnieją lepsze metody, użycie tego wywołania systemowego nie jest popularne.

W przypadku sukcesu funkcja zwraca 0; w przypadku porażki zwraca -1 oraz ustawia `errno` na odpowiednią wartość kodu błędu. W Linuksie oraz — bardziej niż prawdopodobnie — w większości systemów Unix wywołanie funkcji `sched_yield()` nie może się nie udać i dlatego też zawsze zwraca wartość zero. Drobiazgowy programista może jednak ciągle próbować sprawdzać zwracaną wartość:

```
if (sched_yield ( ))
    perror ("sched_yield");
```

Prawidłowe sposoby użycia sched_yield()

W praktyce istnieje kilka prawidłowych sposobów użycia funkcji `sched_yield()` w poprawnym wielozadaniowym systemie z wywłaszczeniem, jak na przykład w Linuksie. Jądro jest w pełni zdolne do podjęcia optymalnych oraz najbardziej efektywnych decyzji dotyczących szeregowania — oczywiście, jądro jest lepiej niż sama aplikacja wyposażone w mechanizmy, które pozwalają decydować, co i kiedy należy wywłaszczyć. Oto, dlaczego w systemach operacyjnych odchodzi się od wielozadaniowości kooperatywnej na rzecz wielozadaniowości z wywłaszczeniem.

Dlaczego więc w ogóle istnieje funkcja systemowa „Przeszereguj mnie”? Odpowiedź można znaleźć w aplikacjach oczekujących na zdarzenia zewnętrzne, które mogą być spowodowane przez użytkownika, element sprzętowy albo inny proces. Na przykład, jeśli jeden proces musi czekać na inny, pierwszym narzucającym się rozwiązaniem tego problemu jest: „po prostu udostępnił czas procesora, dopóki inny proces się nie zakończy”. Jako przykład mogłaby wystąpić implementacja prostego konsumenta w parze producent-konsument, która byłaby podobna do następującego kodu:

```
/* konsument... */
do
{
    while (producer_not_ready ())
        sched_yield ();
    process_data ();
} while (!time_to_quit());
```

Na szczęście programiści systemu Unix nie są skłonni do tworzenia kodu tego typu. Programy uniksowe są zwykle sterowane zdarzeniami i zamiast stosować funkcję `sched_yield()` dążą do używania pewnego rodzaju mechanizmu blokowania (takiego jak potok — ang. *pipe*) pomiędzy konsumentem a producentem. W tym przypadku konsument próbuje czytać z potoku, będąc jednak zablokowanym, dopóki nie pojawią się dane. Z drugiej strony, producent zapisuje do potoku, gdy tylko dostępne stają się nowe dane. Przenosi to odpowiedzialność za koordynację z procesu należącego do poziomu użytkownika, na przykład z pętli zajmujących czas procesora do jądra, które może optymalnie zarządzać taką sytuacją przez przełączanie procesów w stan uśpienia oraz budzenie ich tylko w przypadku potrzeby. Ogólnie rzecz biorąc, programy uniksowe powinny dążyć do rozwiązań sterowanych zdarzeniami, które opierają się na blokowanych deskryptorach plików.

Jedna sytuacja do niedawna koniecznie wymagała `sched_yield()` — blokowanie wątku z poziomu użytkownika. Gdy jakiś wątek próbował uzyskać blokadę, która jest już w posiadaniu innego wątku, wówczas udostępniał on czas procesora dopóty, dopóki blokada nie została zwolniona. Bez wsparcia blokad z poziomu użytkownika ze strony jądra to podejście było najprostsze i najbardziej efektywne. Na szczęście nowoczesna implementacja wątków w Linuksie (*Native POSIX Threading Library* — NPTL) wprowadziła optymalne rozwiązanie przy użyciu mechanizmu *futex*, który dostarcza procesorowi wsparcia dla systemu blokowania z poziomu użytkownika.

Jeszcze innym przykładem użycia funkcji `sched_yield()` jest zagadnienie „sympatycznego działania”: program intensywnie obciążający procesor może wywoływać co jakiś czas `sched_yield()`, przez co minimalizuje swoje obciążenie w systemie. Rozwiązanie to ma być może szlachetny cel, lecz niestety posiada dwie wady. Po pierwsze, jądro potrafi podejmować globalne decyzje dotyczące szeregowania dużo lepiej niż indywidualne procesy i w konsekwencji odpowiedzialność za zapewnienie płynności operacji w systemie powinna spoczywać na zarządcy procesów, a nie na samych procesach. Po drugie, za zmniejszenie obciążenia aplikacji

korzystającej intensywnie z procesora z jednoczesnym wyróżnieniem innych programów odpowiada użytkownik, a nie pojedyncza aplikacja. Użytkownik może zmodyfikować swoje preferencje dotyczące wydajności aplikacji poprzez użycie komendy powłoki systemowej *nice*, która będzie omawiana w dalszej części tego rozdziału.

Priorytety procesu



Dyskusja w tym podrozdziale dotyczy zwykłych procesów, niebędących procesami czasu rzeczywistego. Procesy czasu rzeczywistego wymagają odmiennych kryteriów szeregowania oraz oddzielnego systemu priorytetów. Będzie to omówione w dalszej części rozdziału.

Linux nie szereguje procesów w sposób przypadkowy. Zamiast tego procesom przypisywane są *priorytety*, które wpływają na czas ich działania: jak pamiętasz, część procesora przypisana do procesu jest ważona przy użyciu wartości poziomu uprzejmości. Od początku swojego istnienia Unix nazywał te priorytety *poziomymi uprzejmości* (ang. *nice values*), ponieważ ideą ukrytą za tą nazwą było to, aby „być uprzejmym” dla innych procesów znajdujących się w systemie poprzez obniżanie priorytetu aktualnego procesu, zezwalając przez to innym procesom na konsumowanie większej ilości czasu procesora.

Poprawne poziomy uprzejmości zawierają się w granicach od -20 do 19 włącznie, z domyślną wartością równą zero. W pewien sposób dezorientujące jest to, iż im niższy poziom uprzejmości dla danego procesu, tym wyższy jego priorytet oraz większy przedział czasu; odwrotnie, im wyższa wartość, tym niższy priorytet procesu i mniejszy przedział czasu. Dlatego też zwiększanie poziomu uprzejmości dla procesu jest „uprzejme” dla reszty systemu. Odwrócenie wartości liczbowych wprawia niestety w zakłopotanie. Gdy mówi się, że proces ma „wyższy priorytet”, oznacza to, że częściej zostaje wybrany do uruchomienia i może działać dłużej niż procesy niskopriorytetowe. Taki proces posiada jednak niższy poziom uprzejmości.

nice()

Linux dostarcza kilka funkcji systemowych w celu odczytania oraz ustawienia poziomu uprzejmości dla procesu. Najprostszą z nich jest funkcja `nice()`:

```
#include <unistd.h>
int nice (int inc);
```

Udane wywołanie funkcji `nice()` zwiększa poziom uprzejmości procesu o liczbę przekazaną przez parametr `inc` oraz zwraca uaktualnioną wartość. Tylko proces z uprawnieniami `CAP_SYS_NICE` (w rzeczywistości proces, którego właścicielem jest administrator) może przekazywać ujemną wartość w parametrze `inc`, zmniejszając swój poziom uprzejmości i przez to zwiększając swój priorytet. Zgodnie z tym procesy niebędące procesami należącymi do administratora mogą jedynie obniżyć swój priorytet (poprzez zwiększanie swojego poziomu uprzejmości).

W przypadku błędu wykonania `nice()` zwraca wartość -1 . Ponieważ jednak `nice()` zwraca nową wartość poziomu uprzejmości, -1 jest równocześnie poprawną wartością zwrotną. Aby odróżnić poprawne wywołanie od błędnego, można wyzerować wartość `errno` przed wykonaniem wywołania, a następnie ją sprawdzić. Na przykład:

```

int ret;

errno = 0;
ret = nice (10); /* zwiększ nasz poziom uprzejmości o 10 */
if (ret == -1 && errno != 0)
    perror ("nice");
else
    printf ("Poziom uprzejmości wynosi aktualnie %d\n", ret);

```

Linux zwraca tylko jeden kod błędu: EPERM, informując, iż proces wywołujący próbował zwiększyć swój priorytet (poprzez ujemną wartość parametru `inc`), lecz nie posiadał uprawnień `CAP_SYS_NICE`. Inne systemy zwracają również kod błędu `EINVAL`, gdy `inc` próbuje ustalić nową wartość poziomu uprzejmości poza dopuszczalnym zakresem, lecz Linux tego nie robi. Zamiast tego Linux zaokrągla niewłaściwe wartości `inc` w razie potrzeby w górę lub w dół do wielkości będącej odpowiednią granicą dopuszczalnego zakresu.

Przekazanie zera w parametrze `inc` jest prostym sposobem na uzyskanie aktualnej wartości poziomu uprzejmości:

```
printf ("Poziom uprzejmości wynosi obecnie %d\n", nice (0));
```

Często proces chce ustawić bezwzględną wartość poziomu uprzejmości, zamiast przekazywać względną wielkość różnicy. Można to uzyskać poprzez wykonanie poniższego kodu:

```

int ret, val;

/* pobierz aktualną wartość poziomu uprzejmości */
val = nice (0);

/* chcemy ustawić poziom uprzejmości na 10 */
val = 10 - val;
errno = 0;
ret = nice (val);
if (ret == -1 && errno != 0)
    perror ("nice");
else
    printf ("Poziom uprzejmości wynosi aktualnie %d\n", ret);

```

getpriority() i setpriority()

Preferowanym rozwiązaniem jest użycie funkcji systemowych `getpriority()` oraz `setpriority()`, które udostępniają więcej możliwości kontroli, lecz są bardziej złożone w działaniu:

```

#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);

```

Funkcje te działają na procesie, grupie procesów lub użytkowniku, co odpowiednio ustalają parametry `which` oraz `who`. Parametr `which` musi posiadać jedną z następujących wartości: `PRIO_PROCESS`, `PRIO_PGRP` lub `PRIO_USER`; w tym przypadku parametr `who` określa odpowiednio identyfikator procesu, identyfikator grupy procesów lub identyfikator użytkownika.

Wywołanie funkcji `getpriority()` zwraca najwyższy priorytet (najniższą liczbową wartość poziomu uprzejmości) dla każdego z podanych procesów. Wywołanie funkcji `setpriority()` ustawia priorytet wszystkich podanych procesów na wartość `prio`. Podobnie jak w przypadku funkcji `nice()`, tylko proces posiadający uprawnienia `CAP_SYS_NICE` może zwiększyć priorytet

procesu (zmniejszyć liczbową wartość poziomu uprzejmości). Co więcej, tylko proces z tym uprawnieniem może zwiększyć lub zmniejszyć priorytet procesu niebędącego w posiadaniu przez wywołującego go użytkownika.

Podobnie jak `nice()`, również `getpriority()` zwraca `-1` w przypadku błędu. Ponieważ jest to również poprawna wartość zwrotna, programiści powinni zerować `errno` przed wywołaniem funkcji, jeśli zamierzają obsługiwać przypadki błędów. Wywołanie `setpriority()` nie powoduje takich problemów — `setpriority()` zawsze zwraca `0` w przypadku powodzenia, natomiast `-1` w przypadku niepowodzenia.

Następujący kod zwraca aktualną wartość priorytetu procesu:

```
int ret;
ret = getpriority (PRIO_PROCESS, 0);
printf ("Poziom uprzejmości wynosi %d\n", ret);
```

Poniższy kod ustawia priorytet dla wszystkich procesów z aktualnej grupy procesów na wartość `10`:

```
int ret;
ret = setpriority (PRIO_PGRP, 0, 10);
if (ret == -1)
    perror ("setpriority");
```

W przypadku błędu obie funkcje ustawiają `errno` na jedną z poniższych wartości:

EACCESS

Proces zamierzał zwiększyć priorytet, lecz nie posiada uprawnienia `CAP_SYS_NICE` (tylko dla `setpriority()`).

EINVAL

Wartość przekazana przez parametr `which` nie była jedną z następujących: `PRIO_PROCESS`, `PRIO_PGRP` lub `PRIO_USER`.

EPERM

Efektywny identyfikator użytkownika procesu branego pod uwagę nie zgadza się z efektywnym identyfikatorem użytkownika procesu działającego; proces działający dodatkowo nie posiada uprawnienia `CAP_SYS_NICE` (tylko dla `setpriority()`).

ESRCH

Nie znaleziono procesu spełniającego wymagań, które zostały określone przez parametry `which` oraz `who`.

Priorytety wejścia i wyjścia

W ramach dodatku do priorytetów szeregowania Linux pozwala procesom określić *priorytety wejścia i wyjścia*. Ta wartość wpływa na względny priorytet żądań wejścia i wyjścia dla procesów. Zarządca wejścia i wyjścia z jądra systemu (omówiony w rozdziale 4.) obsługuje żądania pochodzące z procesów o wyższym priorytecie wejścia i wyjścia przed żądaniami z procesów o niższym priorytecie wejścia i wyjścia.

Zarządcy wejścia i wyjścia domyślnie wykorzystują poziom uprzejmości procesu, aby ustalić priorytet wejścia i wyjścia. Dlatego też ustawienie poziomu uprzejmości automatycznie zmienia priorytet wejścia i wyjścia. Jądro Linuksa dostarcza jednakże dodatkowo dwóch funkcji systemowych, w celu jawnego ustawienia i uzyskania informacji dotyczących priorytetu wejścia i wyjścia, niezależnie od wartości poziomu uprzejmości:

```
int ioprio_get (int which, int who)
int ioprio_set (int which, int who, int ioprio)
```

Niestety biblioteka *glibc* nie umożliwia żadnego dostępu do tych funkcji z poziomu użytkownika. Bez wsparcia przez *glibc* używanie ich jest w najlepszym przypadku niewygodne. Co więcej, w momencie, gdy biblioteka *glibc* zacznie udzielać wsparcia, może zdarzyć się, iż jej interfejs nie będzie kompatybilny z tymi funkcjami. Dopóki nie ma takiego wsparcia, istnieją dwa przenośne sposoby pozwalające modyfikować priorytety wejścia i wyjścia: poprzez poziomy uprzejmości lub program użytkowy o nazwie *ionice*, który jest składnikiem pakietu *util-linux*¹.

Nie wszyscy zarządcy wejścia i wyjścia wspierają priorytety wejścia i wyjścia. Wyjątkiem jest zarządca wejścia i wyjścia o nazwie *Complete Fair Queuing* (CFQ); inni typowi zarządcy aktualnie nie posiadają wsparcia dla priorytetów wejścia i wyjścia. W przypadku, gdy dany zarządca wejścia i wyjścia nie wspiera priorytetów wejścia i wyjścia, są one przez niego milcząco pomijane.

Wiązanie procesów do konkretnego procesora

Linux wspiera wieloprocessorowość dla pojedynczego systemu. Poza procesem ładowania systemu większość pracy niezbędnej dla zapewnienia poprawnego działania systemu wieloprocessorowego zrzucana jest na zarządcę procesów. Na maszynie wieloprocessorowej zarządca procesów musi decydować, który proces powinien uruchamiać na każdym procesorze.

Z tej odpowiedzialności wynikają dwa wyzwania: zarządca musi dążyć do tego, aby w pełni wykorzystywać wszystkie procesory w systemie, ponieważ sytuacja, gdy procesor jest nieobciążony, a proces oczekuje na uruchomienie, prowadzi do nieefektywnego działania. Gdy jednak proces został zaszerogowany dla określonego procesora, zarządca procesów powinien dążyć, aby szeregować go do tego samego procesora również w przyszłości. Jest to korzystne, gdyż *migracja* procesu z jednego procesora do innego pociąga za sobą pewne koszty.

Największa część tych kosztów dotyczy *skutków buforowania* (ang. *cache effects*) podczas migracji. Z powodu określonych konstrukcji nowoczesnych systemów SMP większość pamięci podręcznych przyłączonych do każdego procesora jest niezależna i oddzielona od siebie. Oznacza to, że dane z jednej pamięci podręcznej nie powielają się w innej. Dlatego też, jeśli proces przenosi się do nowego procesora i zapisuje nowe informacje w pamięci, dane w pamięci podręcznej poprzedniego procesora mogą stracić aktualność. Poleganie na tej pamięci podręcznej może wówczas spowodować uszkodzenie danych. Aby temu przeciwdziałać, pamięci podręczne *unieważniają* każde inne dane, ilekroć buforują nowy fragment pamięci. Zgodnie z tym określony fragment danych znajduje się dokładnie tylko w jednej pamięci podręcznej procesora w danym momencie (zakładając, że dane są w ogóle buforowane). Gdy proces przenosi się z jednego procesora do innego, pociąga to za sobą następujące koszty: dane buforowane nie są już dostępne dla procesu, który się przeniósł, a dane w źródłowym procesorze muszą być unieważnione. Z powodu tych kosztów zarządca procesów próbuje utrzymać proces na określonym procesorze tak długo, jak to jest możliwe.

Dwa cele zarządcy procesów są oczywiście potencjalnie sprzeczne ze sobą. Jeśli jeden procesor jest znacząco bardziej obciążony niż drugi albo, co gorsza, jeśli jeden procesor jest całkowicie zajęty, podczas gdy drugi jest bezczynny, wówczas zaczyna mieć sens przeszerogowanie

¹ Pakiet *util-linux* jest osiągalny pod adresem <http://kernel.org>. Jest on licencjonowany zgodnie z Powszechną Licencją Publiczną GNU w wersji 2.

niektórych procesów do procesora mniej obciążonego. Decyzja, kiedy należy przenieść procesy w przypadku takiego braku równowagi, zwana *wyrównywaniem obciążenia*, odgrywa bardzo ważną rolę w wydajności maszyn SMP.

Wiązanie procesów dla konkretnego procesora odnosi się do prawdopodobieństwa, że proces zostanie konsekwentnie zaszeregowany do tego samego procesora. Termin *miękkie wiązanie* (ang. *soft affinity*) określa naturalną skłonność zarządcy, aby kontynuować szeregowanie procesu na tym samym procesorze. Jak już powiedziano, jest to wartościowa cecha. Zarządca linuksowy szereguje te same procesy na tych samych procesorach przez tak długi czas, jak to jest możliwe, przenosząc proces z jednego procesora na inny jedynie w przypadku wyjątkowego braku równowagi. Pozwala to zarządcy zmniejszać skutki buforowania w przypadku migracji procesów i zapewnia, że wszystkie procesory w systemie są równo obciążone.

Czasami jednak użytkownik lub aplikacja chcą wymusić powiązanie proces-procesor. Dzieje się tak często, gdy sam proces jest wrażliwy na buforowanie i wymaga pozostawienia go na tym samym procesorze. Wiązanie procesu do konkretnego procesora, które zostaje wymuszone przez jądro systemu, nazywa się *twardym wiązaniem* (ang. *hard affinity*).

sched_getaffinity() i sched_setaffinity()

Procesy dziedziczą wiązania do procesora od swych rodziców, a także — domyślnie — mogą działać na dowolnym procesorze. Linux udostępnia dwie funkcje systemowe pozwalające na odczytanie oraz ustawienie twardego wiązania dla procesu:

```
#define _GNU_SOURCE
#include <sched.h>

typedef struct cpu_set_t;

size_t CPU_SETSIZE;

void CPU_SET (unsigned long cpu, cpu_set_t *set);
void CPU_CLR (unsigned long cpu, cpu_set_t *set);
int CPU_ISSET (unsigned long cpu, cpu_set_t *set);
void CPU_ZERO (cpu_set_t *set);

int sched_setaffinity (pid_t pid, size_t setsize, const cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize, cpu_set_t *set);
```

Wywołanie funkcji `sched_getaffinity()` odczytuje wiązanie do procesora dla procesu o identyfikatorze `pid` i zapisuje go w zmiennej specjalnego typu `cpu_set_t`, która jest dostępna poprzez specyficzne makra. Jeśli `pid` jest równe zero, funkcja zwraca wiązanie dla aktualnego procesu. Parametr `setsize` określa rozmiar typu `cpu_set_t`, który może być używany przez *glibc* dla potrzeb kompatybilności z przyszłymi modyfikacjami dotyczącymi rozmiaru tego typu. W przypadku sukcesu `sched_getaffinity()` zwraca 0; w przypadku błędu zwraca `-1` oraz ustawia `errno`. Oto przykład:

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
    perror ("sched_getaffinity");
```

```

for (i = 0; i < CPU_SETSIZE; i++)
{
    int cpu;

    cpu = CPU_ISSET (i, &set);
    printf ("procesor = %i jest %s\n", i, cpu ? "ustawiony" : "wyzerowany");
}

```

Na początku kodu następuje użycie makra `CPU_ZERO`, by wyzerować wszystkie bity w zmiennej `set`. W dalszej części przeprowadzana jest iteracja w pętli po wszystkich elementach tej zmiennej. Ważne jest to, iż `CPU_SETSIZE` nie jest rozmiarem zmiennej `set` (*nigdy* nie należy używać tej wartości w `setsize!`), lecz liczbą procesorów, które mogłyby być potencjalnie reprezentowane przez `set`. Ponieważ aktualna implementacja definiuje przedstawienie każdego procesora jako jeden bit, `CPU_SETSIZE` jest dużo większe niż `sizeof(cpu_set_t)`. `CPU_ISSET` jest używane, aby sprawdzić, czy dany procesor o numerze `i` jest związany (albo nie) z danym procesem. Makro zwraca zero, gdy nie jest związany, lub wartość niezerową, gdy jest związany.

Tylko procesory zainstalowane fizycznie w systemie są rozpoznawane jako związane. Dlatego też powyższy fragment kodu uruchomiony w systemie z dwoma procesorami zwróci następujący wynik:

```

procesor = 0 jest ustawiony
procesor = 1 jest ustawiony
procesor = 2 jest wyzerowany
procesor = 3 jest wyzerowany
...
procesor = 1023 jest wyzerowany

```

Jak wynika z rezultatów zadziałania kodu, wartość `CPU_SETSIZE` (o indeksie rozpoczynającym się od zera) wynosi aktualnie 1024.

W przykładzie mieliśmy do czynienia wyłącznie z procesorem nr 0 i 1, ponieważ są to jedyne obecne fizycznie procesory w systemie. Być może zaistnieje potrzeba, aby zapewnić konfigurację, w której dany proces działa wyłącznie na procesorze o numerze 0, a w ogóle nie działa na procesorze o numerze 1. To zadanie realizuje poniższy kod:

```

cpu_set_t set;
int ret, i;

CPU_ZERO (&set); /* wyzeruj struktury dla wszystkich procesorów */
CPU_SET (0, &set); /* zezwól na procesor 0 */
CPU_CLR (1, &set); /* zablokuj procesor 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
    perror ("sched_setaffinity");

for (i = 0; i < CPU_SETSIZE; i++)
{
    int cpu;

    cpu = CPU_ISSET (i, &set);
    printf ("procesor = %i jest %s\n", i, cpu ? "ustawiony" : "wyzerowany");
}

```

Program rozpoczyna się, jak zawsze, od zerowania zmiennej `set` za pomocą makra `CPU_ZERO`. Następnie ustawia się procesor 0 przy użyciu `CPU_SET` oraz zeruje procesor 1 przy użyciu `CPU_CLR`. Operacja `CPU_CLR` jest nadmiarowa, gdyż wcześniej już wyzerowano całą strukturę `set`; jest ona jednak użyta, aby zapewnić kompletność kodu.

Uruchomienie tego programu na tym samym dwuprocesorowym systemie zakończy się trochę innymi niż poprzednio wynikami:

```
procesor = 0 jest ustawiony
procesor = 1 jest wyzerowany
procesor = 2 jest wyzerowany
...
procesor = 1023 jest wyzerowany
```

Obecnie procesor 1 jest wyzerowany. Proces będzie działać wyłącznie na procesorze 0, bez względu na wszystko!

Możliwe są cztery wartości `errno`:

`EFAULT`

Dostarczony wskaźnik znajdował się poza przestrzenią adresową procesu lub był ogólnie nieprawidłowy.

`EINVAL`

W tym przypadku w systemie nie istniały fizycznie procesory, które zostały odblokowane w zmiennej `set` (tylko dla `sched_setaffinity()`) lub też `setsize` było mniejsze niż rozmiar wewnętrznej struktury danych jądra użytej w celu reprezentacji zbioru procesorów.

`EPERM`

Proces reprezentowany przez `pid` nie zgadza się z aktualnym efektywnym identyfikatorem użytkownika procesu wywołującego; proces dodatkowo nie posiada uprawnień `CAP_SYS_NICE`.

`ESRCH`

Nie znaleziono procesu posiadającego identyfikator `pid`.

Systemy czasu rzeczywistego

W systemach komputerowych termin *czas rzeczywisty* (ang. *real-time*) jest często powodem pewnego zamieszania i niezrozumienia. System jest systemem czasu rzeczywistego, jeśli jego parametry mieszczą się w zakresie *granicznych parametrów operacyjnych* (ang. *operational deadlines*): czasach reakcji — minimalnym oraz narzuconym — pomiędzy zdarzeniem i odpowiedzią na to zdarzenie. Znanym systemem czasu rzeczywistego jest ABS (ang. *antilock braking system*), spotykany obecnie w prawie wszystkich nowoczesnych samochodach. W tym systemie, jeśli nastąpi rozpoczęcie procesu hamowania, komputer zaczyna odpowiednio regulować docisk hamulców, impulsowo zwiększając go do maksimum i następnie zmniejszając wiele razy na sekundę. Chroni to koła przed zablokowaniem, które mogłoby spowodować pogorszenie warunków hamowania lub nawet wprowadzić samochód w niekontrolowany poślizg. W takim układzie granicznymi parametrami operacyjnymi są: szybkość reakcji systemu na stan zablokowania koła oraz szybkość uzyskania przez system określonego nacisku w hamulcach.

Większość nowoczesnych systemów operacyjnych, także Linux, zapewnia pewien poziom wsparcia dla operacji czasu rzeczywistego.

Systemy ścisłego oraz zwykłego czasu rzeczywistego

Systemy czasu rzeczywistego dzielą się na: systemy ścisłego czasu rzeczywistego oraz zwykłego czasu rzeczywistego. *System ścisłego czasu rzeczywistego* (ang. *hard real-time system*) wymaga dokładnego przestrzegania granicznych parametrów operacyjnych. Przekroczenie tych parametrów

prowadzi do błędu i jest bardzo poważnym problemem. Z drugiej strony, dla *systemu zwykłego czasu rzeczywistego* (ang. *soft real-time system*) przekroczenie granicznych parametrów operacyjnych nie jest błędem krytycznym.

Aplikacje działające w systemach ścisłego czasu rzeczywistego są łatwe do wykrycia: należą do nich systemy ABS, wojskowe systemy obronne, urządzenia medyczne oraz generalnie przetwarzanie sygnałów. Aplikacje zwykłego czasu rzeczywistego nie zawsze można łatwo zidentyfikować. Jednym z oczywistych elementów tej grupy są aplikacje przetwarzające dane wideo: użytkownik zauważa chwilowe pogorszenie jakości, gdy graniczne parametry operacyjne są przekroczone, lecz tych kilka pominiętych ramek obrazu może być w tym momencie tolerowane.

Wiele innych aplikacji posiada ograniczenia czasowe, które (jeśli nie są przestrzegane) pogarszają wrażenia użytkownika. Przykładami są tu aplikacje multimedialne, gry oraz programy sieciowe. A co z edytorami tekstu? Jeśli program nie może szybko reagować na naciśnięcie klawiszy, odpowiedź użytkownika na to zjawisko jest negatywna. Czy jest to więc aplikacja zwykłego czasu rzeczywistego? Oczywiście, gdy projektanci tworzyli tę aplikację, zdawali sobie sprawę, iż reakcja na naciskanie klawiszy musi być możliwie szybka i determinowana czasowo. Ale czy jest to już granicznym parametrem operacyjnym? Granica pomiędzy systemami zwykłego czasu rzeczywistego a systemami standardowymi nie jest wyraźna.

W przeciwieństwie do powszechnego przekonania system czasu rzeczywistego nie musi być szybki. Jeśli uwzględni się porównywalny sprzęt, można stwierdzić, że system czasu rzeczywistego jest prawdopodobnie wolniejszy niż zwykły system, co najmniej z powodu zwiększonego nakładu pracy na wsparcie procesów czasu rzeczywistego. Ponadto, podział pomiędzy systemami ścisłego i zwykłego czasu rzeczywistego nie zależy od wielkości granicznych parametrów operacyjnych. Reaktor atomowy przegrzeje się, jeśli system zabezpieczający przed przegrzaniem nie opuści prętów kontrolnych w ciągu kilku sekund po wykryciu nadmiernego strumienia neutronów. Jest to system ścisłego czasu rzeczywistego ze stosunkowo długim czasem (jak na komputery) parametru granicznego. Przeciwnie, odtwarzacz wideo może opuścić ramkę obrazu lub przerwać dźwięk, jeśli bufor odtwarzania nie będzie wypełniony w ciągu 100 milisekund. Jest to system zwykłego czasu rzeczywistego z krytycznym parametrem granicznym.

Opóźnienie, rozsynchronizowanie oraz parametry graniczne

Termin *opóźnienie* (ang. *latency*) odnosi się do okresu między wystąpieniem zdarzenia a momentem reakcji na to zdarzenie. Jeśli opóźnienie jest mniejsze lub równe wartości parametru granicznego, system działa poprawnie. W wielu systemach ścisłego czasu rzeczywistego operacyjny parametr graniczny oraz opóźnienie są sobie równe — system obsługuje zdarzenia w ściśle ustalonych przedziałach czasowych, w określonych momentach. W systemach zwykłego czasu rzeczywistego żądany czas odpowiedzi jest mniej dokładny i opóźnienie wykazuje pewien rodzaj rozbieżności — należy dążyć do tego, aby odpowiedź mieściła się w zakresie założonego parametru granicznego.

Pomiar opóźnienia jest często trudny, gdyż jego obliczenie wymaga danych na temat czasu, kiedy nastąpiło zdarzenie. Zdolność do oznaczenia dokładnego momentu wystąpienia zdarzenia często jednak pogarsza zdolność poprawnej reakcji na niego. Dlatego też wiele metod obliczania opóźnienia nie działa w ten sposób: zamiast tego mierzy się odchylenie synchronizacji pomiędzy odpowiedziami na zdarzenia. Odchylenie synchronizacji pomiędzy kolejnymi zdarzeniami nazywane jest *rozsynchronizowaniem* (ang. *jitter*), a nie opóźnieniem.

Na przykład, można rozważyć zdarzenie występujące co 10 milisekund. Aby uzyskać charakterystykę sprawności w takim systemie, należałoby mierzyć czasy wystąpień odpowiedzi na te zdarzenia, by upewnić się, że występują faktycznie co 10 milisekund. Odchylenie od tej wartości nie jest jednak opóźnieniem, lecz rozsynchronizowaniem. To, co jest mierzone, jest rozbieżnością kolejnych odpowiedzi. Bez wiedzy o tym, kiedy wystąpiło zdarzenie, nie można znać rzeczywistej różnicy czasu pomiędzy zdarzeniem a odpowiedzią na nie. Nawet wiedząc, że zdarzenie występuje co 10 milisekund, nie ma się pewności, kiedy nastąpił *pierwszy* przypadek tego zdarzenia. Być może jest to zaskakujące, ale wiele metod pomiaru opóźnienia działa błędnie i zwraca wartość rozsynchronizowania, a nie opóźnienia. Oczywiście rozsynchronizowanie to też użyteczny parametr i takie pomiary na pewno są przydatne. Mimo to należy nazywać rzeczy po imieniu!

Systemy ścisłego czasu rzeczywistego często wykazują bardzo małe rozsynchronizowanie, ponieważ odpowiadają na zdarzenie po, a *nie w zakresie* określonego przedziału czasowego. Takie systemy dążą do wartości rozsynchronizowania równej zero, a opóźnienie równe jest czasowi granicznego opóźnienia operacyjnego. Jeśli to opóźnienie przekracza graniczną wartość opóźnienia operacyjnego, system przestaje działać.

Systemy zwykłego czasu rzeczywistego są bardziej wrażliwe na opóźnienia. W nich czas odpowiedzi znajduje się dokładnie w zakresie dopuszczalnej wartości opóźnienia operacyjnego — nierzadko jest krótszy, czasami dłuższy. W pomiarach sprawności systemu wartość rozsynchronizowania jest często doskonałym zamiennikiem dla parametru opóźnienia.

Obsługa czasu rzeczywistego przez system Linux

Linux umożliwia aplikacjom korzystanie ze wsparcia dla systemu zwykłego czasu rzeczywistego poprzez rodzinę funkcji systemowych zdefiniowanych przez *IEEE Std 1003.1b-1993* (nazwa często jest skracana do *POSIX 1993* lub *POSIX.1b*).

Mówiąc językiem technicznym, standard POSIX nie wymusza tego, by udostępniany system czasu rzeczywistego był zwykły lub ścisły. W rzeczywistości to, co naprawdę robi standard POSIX, jest opisem kilku strategii szeregowania, które biorą pod uwagę priorytety. Od projektantów systemu operacyjnego zależy to, jakie metody ograniczeń czasowych zostaną wymuszone przez system dla tych strategii.

Przez lata jądro Linuksa ulepszało swoje wsparcie dla systemu czasu rzeczywistego, dostarczając coraz mniejsze opóźnienia oraz bardziej spójną wartość rozsynchronizowania, przy jednocześnie niezmnieszonej wydajności systemu. Zawdzięczamy te zmiany w większości temu, iż poprawienie opóźnienia wpłynęło pozytywnie na działanie wielu innych grup aplikacji, na przykład procesów powiązanych ze środowiskiem graficznym czy też wejściem i wyjściem, a nie wyłącznie samych aplikacji czasu rzeczywistego. Ulepszenia związane są także z sukcesem Linuksa na polu systemów *wbudowanych* (ang. *embedded*) oraz systemów czasu rzeczywistego.

Niestety wiele z tych modyfikacji, które zostały wprowadzone w jądrze Linuksa dla celów systemów wbudowanych oraz czasu rzeczywistego, istnieje tylko w niestandardowych rozwiązaniach, poza podstawową i oficjalną wersją jądra. Niektóre z tych ulepszeń zapewniają dalsze ograniczenia opóźnienia, a nawet zachowanie charakterystyczne dla systemów ścisłego czasu rzeczywistego. W kolejnych podrozdziałach omówione zostaną tylko oficjalne interfejsy oraz zachowanie standardowego jądra. Na szczęście większość zmian dotyczących wspierania czasu rzeczywistego opiera się w dalszym ciągu na interfejsach POSIX. Zatem dalsze dyskusje odnoszą się również do systemów zmodyfikowanych.

Linuksowe strategie szeregowania i ustalania priorytetów

Zachowanie zarządcy Linuksa w odniesieniu do procesu zależy od *strategii szeregowania* (ang. *scheduling policy*) dla tego procesu, zwanej również *klasą szeregowania* (ang. *scheduling class*). Jako uzupełnienie zwykłej, domyślnej strategii Linux dostarcza dwie strategie szeregowania dla czasu rzeczywistego. Makra preprocesora z pliku nagłówka `<sched.h>` reprezentują każdą z tych strategii i zwane są `SCHED_FIFO`, `SCHED_RR` oraz `SCHED_OTHER`.

Każdy proces posiada *priorytet statyczny* (ang. *static priority*), niezależny od wartości poziomu uprzejmości. Dla zwykłych aplikacji priorytet ten jest zawsze równy zeru. Dla procesów czasu rzeczywistego wynosi on od 1 do 99 włącznie. Linuksowy zarządca zawsze wybiera proces o najwyższym priorytecie, aby go uruchomić (tzn. ten, który posiada priorytet statyczny posiadający największą wartość liczbową). Jeśli proces działa z wartością priorytetu statycznego równą 50, a kolejny proces o priorytecie 51 staje się uruchamialny, zarządca natychmiast wykonuje przeseregowanie poprzez przełączenie się na tenże uruchamialny proces. I odwrotnie, jeśli w systemie działa już proces o priorytecie 50, a proces o priorytecie 49 staje się uruchamialny, zarządca nie uruchomi go, dopóki proces o priorytecie 50 nie zablokuje się, czyli przestanie działać. Ponieważ normalne procesy mają priorytet 0, każdy proces czasu rzeczywistego, który jest uruchamialny, zawsze wywłaszczy zwykły proces i zacznie działać.

Strategia FIFO (first in, first out)

Klasa „*pierwszy na wejściu, pierwszy na wyjściu*” (ang. *first in, first out class*), inaczej zwana też *klasą FIFO* (ang. *FIFO class*), jest bardzo prostą strategią czasu rzeczywistego bez przedziałów czasowych. Proces z klasy FIFO będzie działał tak długo, jak długo nie będzie istnieć żaden inny proces o większym priorytecie. Klasa FIFO jest opisywana przez makro `SCHED_FIFO`.

Ponieważ ta strategia nie używa przedziałów czasowych, zasady działań są raczej proste:

- Proces uruchamialny klasy FIFO będzie zawsze działał, jeśli jest procesem o najwyższym priorytecie w systemie. Gdy tylko proces klasy FIFO stanie się uruchamialny, od razu wywłaszczy proces zwykły.
- Proces klasy FIFO będzie kontynuować swoje działanie, dopóki się nie zablokuje lub wywoła `sched_yield()` albo dopóki proces o wyższym priorytecie nie stanie się uruchamialny.
- Gdy proces klasy FIFO zablokuje się, zarządca usunie go z listy procesów uruchamialnych. Gdy znów staje się on uruchamialny, jest umieszczany na końcu listy procesów odpowiadającej jego priorytetowi. Dlatego też nie będzie on uaktywniony, dopóki wszystkie inne procesy o wyższym lub *równym* priorytecie nie przerwą swojego działania.
- Gdy proces klasy FIFO wywoła funkcję `sched_yield()`, zarządca przesunie go na koniec listy procesów odpowiadającej jego priorytetowi. Dlatego też nie będzie on uruchomiony, dopóki wszystkie inne procesy o równym jemu priorytecie nie zakończą swego działania. Jeśli proces wywołujący funkcję `sched_yield()` będzie jedynym procesem posiadającym taki priorytet, wywołanie tej funkcji systemowej nie odniesie żadnego skutku.
- Gdy proces o wyższym priorytecie wywłaszczy proces klasy FIFO, pozostanie on na takiej samej pozycji w liście procesów odpowiadającej jego priorytetowi. Zatem gdy proces o wyższym priorytecie zostanie zawieszony, wywłaszczony proces klasy FIFO będzie mógł w dalszym ciągu kontynuować swoje działanie.

- Gdy proces przystępuje do klasy FIFO lub gdy priorytet statyczny procesu zmienia się, jest on umieszczany na początku listy procesów pasującej do jego priorytetu. W rezultacie proces klasy FIFO otrzymujący nowy priorytet może wywłaszczyć inny działający proces posiadający ten sam priorytet.

Zasadniczo można powiedzieć, iż procesy klasy FIFO działają tak długo, jak chcą, dopóki są procesami posiadającymi najwyższy priorytet w systemie. Ciekawe reguły dotyczą natomiast tych procesów klasy FIFO, które posiadają identyczne priorytety.

Strategia cykliczna

Klasa cykliczna (ang. *round-robin class*) jest identyczną klasą jak FIFO, z wyjątkiem tego, że posiada dodatkowe reguły w przypadku zarządzania procesami o takim samym priorytecie. Klasa cykliczna jest opisywana przez makro `SCHED_RR`.

Zarządca przypisuje każdemu procesowi klasy cyklicznej przedział czasowy. Gdy proces klasy cyklicznej wykorzysta swój przedział czasu, zarządca przesuwając go na koniec listy procesów odpowiadającej jego priorytetowi. W ten sposób procesy klasy cyklicznej, posiadające dany priorytet, są regularnie szeregowane w pętli. Jeśli istnieje tylko jeden proces o danym priorytecie, wówczas klasa cykliczna działa identycznie jak klasa FIFO. W takim przypadku, gdy przedział czasowy ulegnie zużyciu, proces po prostu wznawia swoje działanie.

Można uważać proces klasy cyklicznej za identyczny do klasy FIFO, z wyjątkiem tego, iż dodatkowo wstrzymuje on swoje działanie, gdy zużyje przedział czasowy należący do niego, kiedy jest przenoszony na koniec listy procesów odpowiadającej jego priorytetowi.

Decyzja, czy należy użyć klasy `SCHED_FIFO` lub `SCHED_RR` jest kwestią dotyczącą zachowania się samych priorytetów. Przedziały czasowe klasy cyklicznej mają znaczenie tylko pomiędzy procesami o takim samym priorytecie. Procesy klasy FIFO będą działać niezagrożone; procesy klasy cyklicznej będą wywłaszczać się pomiędzy sobą w przypadku wystąpień takich samych priorytetów. W żadnym przypadku proces o mniejszym priorytecie nie będzie mógł zostać uruchomiony, gdy działa proces o wyższym priorytecie.

Strategia zwykła

Makro `SCHED_OTHER` opisuje standardową strategię szeregowania, będącą domyślną strategią dla klasy niespełniającej wymagań czasu rzeczywistego. Wszystkie procesy *klasy zwykłej* (ang. *normal class*) posiadają priorytet statyczny równy zeru. Dlatego też każdy uruchamialny proces należący do klasy FIFO lub klasy cyklicznej będzie wywłaszczać proces działający w klasie normalnej.

Zarządca używa poziomów uprzejmości, aby ustalać priorytety procesów wewnątrz klasy zwykłej. Poziom uprzejmości nie oddziałuje na priorytet statyczny, który dla tej klasy pozostaje zawsze równy zeru.

Strategia szeregowania wsadowego

Makro `SCHED_BATCH` opisuje *strategię szeregowania wsadowego* (ang. *batch scheduling policy*), inaczej zwaną też *strategią jałowego szeregowania* (ang. *idle scheduling policy*). Jej zachowanie jest jakby przeciwieństwem strategii czasu rzeczywistego: procesy należące do tej klasy mogą działać tylko wtedy, gdy nie istnieją żadne inne uruchamialne procesy w systemie, nawet jeśli te procesy wykorzystywały już swoje przedziały czasowe. Różni się to od zachowania w przypadku procesów

z największymi wartościami poziomów uprzejmości (tzn. procesów o najniższych priorytetach) — te procesy w końcu zaczną działać, gdy procesy o wyższych priorytetach wykorzystają swoje przedziały czasowe.

Ustalanie strategii szeregowania dla systemu Linux

Procesy mogą zmieniać strategię szeregowania Linuksa poprzez wywołanie funkcji systemowych `sched_getscheduler()` oraz `sched_setscheduler()`:

```
#include <sched.h>

struct sched_param
{
    /* ... */
    int sched_priority;
    /* ... */
};
int sched_getscheduler (pid_t pid);
int sched_setscheduler (pid_t pid, int policy, const struct sched_param *sp);
```

Poprawne wywołanie funkcji `sched_getscheduler()` odczytuje strategię szeregowania dla procesu określonego poprzez parametr `pid`. Jeśli `pid` wynosi 0, funkcja zwraca strategię szeregowania dla procesu ją wywołującego. Liczba całkowita zdefiniowana w `<sched.h>` opisuje strategię szeregowania: strategia FIFO określona jest przez `SCHED_FIFO`, strategia cykliczna przez `SCHED_RR`, a strategia normalna przez `SCHED_OTHER`. W przypadku błędu funkcja zwraca `-1` (co nie jest żadną poprawną wartością strategii szeregowania) i odpowiednio ustawia `errno`.

Użycie tej funkcji jest proste:

```
int policy;
/* pobierz strategię szeregowania */
policy = sched_getscheduler (0);
switch (policy)
{
    case SCHED_OTHER:
        printf ("Strategia normalna\n");
        break;
    case SCHED_RR:
        printf ("Strategia cykliczna\n");
        break;
    case SCHED_FIFO:
        printf ("Strategia FIFO\n");
        break;
    case -1:
        perror ("sched_getscheduler");
        break;
    default:
        fprintf (stderr, "Strategia nieznaną!\n");
}
}
```

Wywołanie funkcji `sched_setscheduler()` ustawia strategię szeregowania dla procesu wskazanego przez `pid` na strategię o wartości przekazanej w parametrze `policy`. Dodatkowe parametry związane ze strategią ustawiane są poprzez parametr `sp`. W przypadku poprawnego wywołania funkcja zwraca 0, natomiast w przypadku błędu zwraca `-1` oraz odpowiednio ustawia `errno`.

Poprawność pól wewnątrz struktury `sched_param` zależy od strategii szeregowania zapewnianych przez system operacyjny. Strategie `SCHED_RR` oraz `SCHED_FIFO` wymagają jednego pola — `sched_priority`, które reprezentuje priorytet statyczny. `SCHED_OTHER` nie używa żad-

nego pola, natomiast strategie, które mogą zostać użyte w przyszłości, będą być może wymagać nowych pól. Dlatego też przenośne i poprawne programy nie mogą czynić żadnych założeń dotyczących formatu tej struktury.

Ustawianie strategii oraz parametrów szeregowania dla procesu jest proste:

```
struct sched_param sp = { .sched_priority = 1 };
int ret;

ret = sched_setscheduler (0, SCHED_RR, &sp);
if (ret == -1)
{
    perror ("sched_setscheduler");
    return 1;
}
```

Ten fragment kodu ustawia strategię szeregowania dla procesu wywołującego na strategię cykliczną oraz ustala priorytet statyczny równy 1. Zakłada się w tym momencie, iż wartość 1 jest poprawną wartością priorytetu — z technicznego punktu widzenia nie zawsze tak musi być. W następnym podrozdziale zostanie omówione, jak należy określać prawidłowy zakres priorytetów dla danej strategii.

Ustawianie strategii innej niż SCHED_OTHER wymaga posiadania uprawnień CAP_SYS_NICE. Dlatego też najczęściej administrator ma prawo do uruchamiania procesów czasu rzeczywistego. Od wersji jądra 2.6.12 parametr ograniczeń zasobów RLIMIT_RTPRIO pozwala także innym użytkownikom na ustawianie strategii czasu rzeczywistego aż do pewnej określonej granicy priorytetów.

Kody błędów

W przypadku błędu możliwe są cztery wartości errno:

EFAULT

Zmienna wskaźnikowa sp wskazuje na błędny lub niedostępny obszar pamięci.

EINVAL

Strategia szeregowania określona przez parametr policy jest nieprawidłowa albo też wartość przekazywana przez sp nie ma sensu dla danej strategii (tylko dla sched_setscheduler()).

EPERM

Proces wywołujący nie posiada odpowiednich uprawnień.

ESRCH

Wartość przekazana w pid nie odzwierciedla żadnego istniejącego procesu.

Ustawianie parametrów szeregowania

Funkcje systemowe sched_getparam() oraz sched_setparam(), zdefiniowane przez POSIX, odczytują oraz ustawiają parametry związane z ustaloną już strategią szeregowania:

```
#include <sched.h>
struct sched_param
{
    /* ... */
    int sched_priority;
    /* ... */
};
int sched_getparam (pid_t pid, struct sched_param *sp);
int sched_setparam (pid_t pid, const struct sched_param *sp);
```

Funkcja `sched_getscheduler()` zwraca wyłącznie strategię szeregowania, natomiast nie zwraca związanych z nią parametrów. Wywołanie `sched_getparam()` dla procesu określonego w parametrze `pid` zwraca parametry szeregowania w zmiennej `sp`:

```
struct sched_param sp;
int ret;
ret = sched_getparam (0, &sp);
if (ret == -1)
{
    perror ("sched_getparam");
    return 1;
}
printf ("Nasz priorytet wynosi %d\n", sp.sched_priority);
```

Jeśli `pid` wynosi 0, funkcja podaje parametry dla procesu wywołującego. W przypadku sukcesu funkcja zwraca 0, w przypadku niepowodzenia zwraca -1 oraz odpowiednio ustawia `errno`.

Ponieważ `sched_setscheduler()` również ustawia parametry szeregowania, `sched_setparam()` jest użyteczna tylko w celu późniejszych modyfikacji tych parametrów:

```
struct sched_param sp;
int ret;
sp.sched_priority = 1;
ret = sched_setparam (0, &sp);
if (ret == -1)
{
    perror ("sched_setparam");
    return 1;
}
```

W przypadku sukcesu parametry szeregowania dla procesu `pid` są ustawiane poprzez zmienną `sp`, a funkcja zwraca 0. W przypadku niepowodzenia funkcja zwraca -1 oraz odpowiednio ustawia `errno`.

Po uruchomieniu tych dwóch fragmentów kodu otrzyma się następujący wynik:

```
Nasz priorytet wynosi 1
```

Przykład ten także jest oparty na założeniu, że 1 określa prawidłową wartość priorytetu. Tak przeważnie jest, lecz podczas tworzenia programów przenośnych należy to wcześniej sprawdzić. Za chwilę zostanie pokazane, jak ustalać zakres poprawnych priorytetów.

Kody błędów

W przypadku błędu możliwe są cztery wartości `errno`:

EFAULT

Zmienna wskaźnikowa `sp` wskazuje na błędny lub niedostępny obszar pamięci.

EINVAL

Wartość przekazana przez `sp` nie ma sensu dla danej strategii (tylko `sched_getparam()`).

EPERM

Proces wywołujący nie posiada niezbędnych uprawnień.

ESRCH

Wartość przekazana w `pid` nie odpowiada żadnemu istniejącemu procesowi.

Określanie zakresu poprawnych priorytetów

Poprzednie przykłady kodów przekazywały sztywno ustalone wartości priorytetów do odpowiednich funkcji systemowych. POSIX nie gwarantuje, że dane wartości priorytetów szeregowania istnieją w określonym systemie, z wyjątkiem tego, że muszą istnieć przynajmniej 32 różne priorytety pomiędzy najniższą a najwyższą wartością. Jak wspomniano już wcześniej w punkcie „Linuksowe strategie szeregowania i ustalania priorytetów”, Linux przypisuje zakres wartości od 1 do 99 włącznie dla dwóch strategii czasu rzeczywistego. Poprawny i przenośny program zwykle definiuje swój własny obszar wartości priorytetów i mapuje je w odpowiedni zakres, właściwy dla systemu operacyjnego. Na przykład, jeśli zaistnieje potrzeba uruchomienia procesów o czterech różnych poziomach priorytetów czasu rzeczywistego, można dynamicznie określić zakres priorytetów i wybrać cztery konkretne wartości.

Linux udostępnia dwie funkcje systemowe dla odczytu zakresu poprawnych wartości priorytetów. Jedna z nich zwraca wartość minimalną, a druga maksymalną:

```
#include <sched.h>
int sched_get_priority_min (int policy);
int sched_get_priority_max (int policy);
```

W przypadku sukcesu wywołanie funkcji `sched_get_priority_min()` zwraca wartość minimalną, natomiast wywołanie `sched_get_priority_max()` zwraca maksymalny poprawny priorytet powiązany ze strategią szeregowania, przekazaną w parametrze `policy`. W przypadku niepowodzenia obie funkcje zwracają `-1`. Jedyny możliwy błąd, jaki może wystąpić, dotyczy błędnego parametru `policy` — wówczas `errno` ustawiane jest na `EINVAL`.

Użycie funkcji jest proste:

```
int min, max;
min = sched_get_priority_min (SCHED_RR);
if (min == -1)
{
    perror ("sched_get_priority_min");
    return 1;
}
max = sched_get_priority_max (SCHED_RR);
if (max == -1)
{
    perror ("sched_get_priority_max");
    return 1;
}
printf ("Zakres wartości priorytetów dla strategii SCHED_RR wynosi: %d - %d\n", min,
max);
```

W standardowym systemie Linux powyższy fragment kodu spowoduje wyświetlenie następującego wyniku:

```
Zakres wartości priorytetów dla strategii SCHED_RR wynosi: 1 - 99
```

Jak już wcześniej wspomniano, większe wartości liczbowe oznaczają wyższe priorytety. Aby przypisać procesowi najwyższy priorytet dla jego strategii szeregowania, należy posłużyć się następującym kodem:

```
/*
 * set_highest_priority — dla procesu pid ustawia priorytet
 * szeregowania na największą wartość, dopuszczaną przez jego
 * aktualną strategię szeregowania.
 * Jeśli wartość pid wynosi zero, następuje ustawienie priorytetu
 * dla aktualnego procesu.
```

```

*
* W przypadku sukcesu funkcja zwraca zero.
*/
int set_highest_priority (pid_t pid)
{
    struct sched_param sp;
    int policy, max, ret;

    policy = sched_getscheduler (pid);
    if (policy == -1)
        return -1;

    max = sched_get_priority_max (policy);
    if (max == -1)
        return -1;

    memset (&sp, 0, sizeof (struct sched_param));
    sp.sched_priority = max;
    ret = sched_setparam (pid, &sp);

    return ret;
}

```

Programy zwykle zwracają minimalną lub maksymalną wartość dla systemu, a następnie modyfikują te wielkości o 1 (np. `max - 1`, `max - 2` itd.), aby odpowiednio przypisać priorytety procesom.

sched_rr_get_interval()

Jak już zostało wcześniej powiedziane, procesy klasy `SCHED_RR` zachowują się tak samo jak procesy klasy `SCHED_FIFO`, z wyjątkiem tego, że zarządca przydziela tym procesom przedziały czasowe. Gdy proces należący do klasy `SCHED_RR` wyczerpie swój przedział czasowy, zarządca przesuwa go na koniec listy procesów działających, zawierającej również inne procesy o priorytecie równym jego aktualnemu priorytetowi. W ten oto sposób wszystkie procesy klasy `SCHED_RR` o tym samym priorytecie są wykonywane cyklicznie. Te o wyższym priorytecie (oraz procesy klasy `SCHED_FIFO` o takim samym lub wyższym priorytecie) będą zawsze wywłaszczać działający proces klasy `SCHED_RR`, bez względu na to, czy pozostał mu jakiś przedział czasowy do wykorzystania.

POSIX definiuje interfejs w celu odczytania wielkości przedziału czasowego dla danego procesu:

```

#include <sched.h>
struct timespec
{
    time_t tv_sec; /* liczba sekund */
    long tv_nsec; /* liczba nanosekund */
};
int sched_rr_get_interval (pid_t pid, struct timespec *tp);

```

Poprawne wywołanie funkcji systemowej o skomplikowanej nazwie `sched_rr_get_interval()` zapisuje w strukturze `timespec` wskazywanej przez parametr `tp` czas trwania przedziału czasowego przydzielonego dla procesu `pid` oraz zwraca zero. W przypadku błędu funkcja zwraca `-1` oraz odpowiednio ustawia `errno`.

Zgodnie ze standardem POSIX funkcja ta jest wymagana wyłącznie dla pracy z procesami klasy `SCHED_RR`. Jednak w systemie Linux może ona udostępniać długość przedziału czasowego dowolnego procesu. Programy przenośne powinny zakładać, że funkcja ta działa tylko z proce-

sami cyklicznymi; programy dedykowane dla Linuksa mogą nadużywać tej funkcji. Oto przykład:

```
struct timespec tp;
int ret;
/* pobierz wielkość przedziału czasowego dla aktualnego zadania */
ret = sched_rr_get_interval (0, &tp);
if (ret == -1)
{
    perror ("sched_rr_get_interval");
    return 1;
}
/* zamień liczbę sekund i nanosekund na milisekundy */
printf ("Nasz kwant czasowy wynosi %.21f milisekund\n", (tp.tv_sec * 1000.0f) +
(tp.tv_nsec / 1000000.0f));
```

Jeśli proces działa w klasie FIFO, wartości zmiennych `tv_sec` oraz `tv_nsec` wynoszą 0, co symbolizuje w tym przypadku nieskończoność.

Kody błędów

W przypadku błędów wartości `errno` są następujące:

`EFAULT`

Zmienna wskaźnikowa `tp` wskazuje na błędny lub niedostępny obszar pamięci.

`EINVAL`

Wartość przekazana przez `pid` jest błędna (np. ujemna).

`ESRCH`

Wartość przekazana w `pid` jest poprawna, lecz nie odnosi się do żadnego istniejącego procesu.

Środki ostrożności przy pracy z procesami czasu rzeczywistego

Z powodu samej natury procesów czasu rzeczywistego projektanci powinni być ostrożni podczas tworzenia i uruchamiania takich programów. Gdy program czasu rzeczywistego zaczyna się dziwnie zachowywać, cały system może się zawiesić. Każda pętla związana z procesorem w programie czasu rzeczywistego, to znaczy dowolny kawałek kodu, który nie może się zatrzymać, będzie kontynuować swoje działanie w nieskończoność, tak długo, dopóki procesy czasu rzeczywistego o wyższym priorytecie nie staną się uruchamialne.

Dlatego też projektowanie programów czasu rzeczywistego wymaga ostrożności oraz szczególnej uwagi. Takie programy rządzą w sposób absolutny i mogą w prosty sposób spowodować zawieszenie całego systemu. Oto kilka wskazówek i ostrzeżeń:

- Należy pamiętać, że każda pętla programowa wykonywana przez proces związany z procesorem będzie działać bez jakiegokolwiek przerwania pracy, dopóki się naturalnie nie zakończy, jeśli w systemie nie pojawi się żaden proces o wyższym priorytecie. Jeśli pętla jest nieskończona, system zawiesi się.
- Ponieważ procesy czasu rzeczywistego działają kosztem całego systemu, należy zwrócić szczególną uwagę na sposób ich zaprojektowania. Trzeba zadbać, aby nie dopuścić do trwałego zablokowania systemu z powodu poświęcenia mu zbyt małej ilości czasu procesora.

- Należy być bardzo ostrożnym w przypadku oczekiwania w pętli (ang. *busy-wait*). Jeśli proces czasu rzeczywistego oczekuje w pętli na zasób zajmowany przez proces o niższym priorytecie, będzie niestety czekać w nieskończoność.
- Podczas projektowania procesu czasu rzeczywistego zalecane jest używanie terminala, który posiada wyższy priorytet niż proces projektowany. Dzięki temu w przypadku problemu terminal pozostanie w dalszym ciągu aktywny i umożliwi usunięcie procesu, który wymknął się spod kontroli (dopóki terminal pozostanie w stanie jałowym, oczekując na dane z klawiatury, nie będzie przeszkadzać innym procesom).
- Program użytkowy *chrt*, jeden z elementów pakietu narzędziowego *util-linux*, upraszcza odczytywanie i ustawianie atrybutów czasu rzeczywistego dla innych procesów. Narzędzie to ułatwia uruchamianie dowolnych programów w trybie szeregowania czasu rzeczywistego, na przykład wyżej wspomnianego terminala, jak również pozwala na zmianę priorytetów czasu rzeczywistego dla istniejących aplikacji.

Determinizm

Procesy czasu rzeczywistego są pełne determinizmu. W obliczeniach czasu rzeczywistego czynność jest *deterministyczna* (ang. *deterministic*), jeśli przy takich samych danych wejściowych generuje zawsze ten sam wynik w tym samym przedziale czasu. Nowoczesne komputery są bardzo dobrym przykładem systemów niedeterministycznych: wielopoziomowe pamięci podręczne (poddające się trafieniom i chybieniom bez żadnej przewidywalności), wieloprocesorowość, stronicowanie (ang. *paging*), wymiana danych (ang. *swapping*) oraz wielozadaniowość niszczą każde oszacowanie zmierzające do tego, aby ustalić, jak długo dana czynność będzie się wykonywać. Oczywiście osiągnięto już punkt, w którym praktycznie każda czynność (za wyjątkiem dostępu do twardego dysku) jest „niesamowicie szybka”, ale jednocześnie nowoczesne systemy utrudniły dokładne sprecyzowanie, ile czasu zajmie jej wykonanie.

Aplikacje czasu rzeczywistego często próbują ogólnie ograniczyć nieprzewidywalność, a w szczególności najbardziej niekorzystne opóźnienia. Kolejne podrozdziały omówią dwie metody, które używane są w tym celu.

Wcześniejsze zapisywanie danych oraz blokowanie pamięci

Rozważmy następującą sytuację: zostaje wygenerowane przerwanie sprzętowe, pochodzące od niestandardowego monitora śledzącego międzykontynentalne pociski balistyczne, co powoduje, że sterownik urządzenia chce szybko skopiować dane z układu sprzętowego do jądra systemu. Sterownik zauważa jednak, że proces jest uspiiony, blokując się na elemencie sprzętowym i czekając na dane. Powiadamia jądro, aby obudziło ten proces. Jądro, zauważając, że działa on w strategii czasu rzeczywistego i posiada wysoki priorytet, natychmiast wywłaszcza aktualnie działający proces, decydując o natychmiastowym zaszeregowaniu procesu czasu rzeczywistego. Zarządca uruchamia tenże proces i przełącza kontekst do jego przestrzeni adresowej. Proces zaczyna działać. Cała akcja trwa 0,3 milisekundy, przy zakładanej wartości najgorszego przypadku opóźnienia równej 1 milisekundzie.

Proces, będąc obecnie na poziomie użytkownika, zauważa nadlatujący międzykontynentalny pocisk balistyczny i zaczyna przetwarzać jego trajektorię. Gdy balistyka zostanie wyliczona, proces czasu rzeczywistego uruchamia wystrzelenie pocisku służącego do niszczenia głowic rakiet o dalekim zasięgu. Minęło kolejne 0,1 milisekundy — wystarczające, aby uruchomić

odpowieź systemu obronnego i ocalić życie ludzi. Jednak w tym momencie kod zarządzający wystrzeleniem pocisku zostaje przerzucony na dysk! Następuje błąd dostępu do strony, procesor z powrotem przełącza się w tryb jądra, jądro rozpoczyna operacje dyskowe wejścia i wyjścia w celu przeniesienia zapisanych danych z powrotem do pamięci. Zarządca przełącza proces w tryb uśpienia na czas obsługi błędu dostępu. Mijają kolejne sekundy. Jest już za późno...

Stronicowanie i wymiana danych wprowadzają oczywiście całkiem niedeterministyczne zachowanie, które może spowodować zniszczenia w procesie czasu rzeczywistego. Aby uchronić się przed taką katastrofą, aplikacje czasu rzeczywistego często „blokują” lub „wiążą na stałe” wszystkie strony ze swojej przestrzeni adresowej z pamięcią fizyczną, zapisując je wstępnie do niej i chroniąc przed wymieceniem na dysk. Gdy tylko strony zostaną zablokowane w pamięci, jądro już ich nie zapisze na dysk. Dowolny dostęp do tych stron nie spowoduje żadnych błędów. Większość aplikacji czasu rzeczywistego blokuje niektóre lub nawet wszystkie swoje strony w fizycznej pamięci operacyjnej.

Linux dostarcza funkcji systemowych w celu wcześniejszego zapisywania oraz blokowania danych. W rozdziale 4. omówione zostały interfejsy używane dla zapisywania danych do pamięci fizycznej. W rozdziale 9. przedstawione będą funkcje systemowe przeznaczone dla blokowania danych w fizycznej pamięci operacyjnej.

Wiązanie do procesora a procesy czasu rzeczywistego

Drugim problemem aplikacji czasu rzeczywistego jest wielozadaniowość. Choć jądro Linuksa działa w trybie wyłączenia, jego zarządca nie zawsze potrafi na żądanie przeszerogować jeden proces, by udostępnić czas drugiemu. Czasami aktualnie działający proces wykonuje się wewnątrz regionu krytycznego w jądrze i zarządca nie potrafi wyłączyć go, dopóki nie opuści on tego regionu. Jeśli jest to proces, który oczekuje na uruchomienie w czasie rzeczywistym, to opóźnienie może być nie do zaakceptowania i istnieje niebezpieczeństwo, że szybko przekroczy graniczny parametr operacyjny.

Zatem wielozadaniowość wprowadza niedeterminizm podobny w naturze do nieprzewidywalności spotykanej w stronicowaniu. Rozwiązanie, które bierze pod uwagę wielozadaniowość, jest jedno: należy ją po prostu wyeliminować. Niestety jest prawdopodobne, że nie da się łatwo usunąć wszystkich innych procesów. Gdyby było to możliwe w danym środowisku, prawdopodobnie nie byłoby potrzeby użycia systemu Linux — wystarczyłby dowolny prosty system operacyjny. Jeśli jednak system posiada wiele procesorów, można przeznaczyć jeden lub więcej z tych procesorów dla procesu (lub procesów) czasu rzeczywistego. W praktyce można chronić procesy czasu rzeczywistego przed wielozadaniowością.

Wcześniej w tym rozdziale omówiono funkcje systemowe, używane w celu kontroli wiązania procesora dla danego procesu. Możliwą optymalizacją dla aplikacji czasu rzeczywistego jest zarezerwowanie po jednym procesorze dla każdego z procesów czasu rzeczywistego oraz zezwolenie wszystkim innym procesom na dzielenie swojego czasu na pozostałym procesorze.

Najprostszą metodą, aby to osiągnąć, jest takie zmodyfikowanie jednej z wersji głównego programu inicjalizującego („ojca procesów”) *init* o nazwie *SysVinit*², aby wykonywał coś podobnego do kodu zamieszczonego poniżej, zanim rozpocznie proces startowania systemu:

² Źródła *SysVinit* znajdują się pod adresem <http://freecode.com/projects/sysvinit/>. Program jest licencjonowany zgodnie z Powszechną Licencją Publiczną GNU v2.

```

cpu_set_t set;
int ret;

CPU_ZERO (&set); /* wyzeruj wszystkie procesory */
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
{
    perror ("sched_getaffinity");
    return 1;
}

CPU_CLR (1, &set); /* nie dopuszczaj procesora o numerze 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
{
    perror ("sched_setaffinity");
    return 1;
}

```

Ten fragment kodu pobiera aktualny zestaw dozwolonych procesorów dla programu *init*. Zestaw ten zawiera (jak się tego oczekuje) wszystkie dostępne procesory. W dalszej części kodu następuje usunięcie z tego zbioru procesora o numerze 1 oraz aktualizacja listy dopuszczonych procesorów.

Ponieważ lista dozwolonych procesorów jest dziedziczona przez potomstwo, a program *init* jest patronem wszystkich procesów, wszystkie procesy systemowe będą działać z takim zestawem dozwolonych procesorów, na jaki zezwoli program *init*. Dlatego też w tym przypadku żaden z procesów nie będzie używać procesora o numerze 1.

Następnie należy tak zmodyfikować proces czasu rzeczywistego, aby działał wyłącznie na procesorze nr 1:

```

cpu_set_t set;
int ret;

CPU_ZERO (&set); /* wyzeruj wszystkie procesory */
CPU_SET (1, &set); /* pozwalaj na procesor o numerze 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
{
    perror ("sched_setaffinity");
    return 1;
}

```

Wynikiem użycia powyższych fragmentów kodu jest to, iż procesy czasu rzeczywistego działają tylko na procesorze nr 1, a wszystkie inne procesy używają pozostałych procesorów.

Ograniczenia zasobów systemowych

Jądro Linuksa narzuca na procesy pewne *ograniczenia zasobów systemowych* (ang. *resource limits*). Te ograniczenia ustalają bezwzględne górne granice dotyczące wielkości zasobów jądra, których proces może używać, na przykład liczby otwartych plików, stron pamięci, zawieszonych sygnałów itd. Ograniczenia są stosowane bezwarunkowo; jądro nie zezwoli na działanie, które pozwoliłoby procesowi użyć jakiegos zasobu powyżej dopuszczalnej górnej granicy. Na przykład, jeśli otwarcie nowego pliku pozwoliłoby procesowi posiadać więcej otwartych plików, niż zezwolono

w odpowiednim ograniczeniu zasobów systemowych, wywołanie funkcji `open()` nie powiedzie się³.

Linux udostępnia dwie funkcje systemowe służące kontroli ograniczeń zasobów systemowych. Co prawda POSIX znormalizował oba interfejsy, lecz Linux wspiera dodatkowo jeszcze inne ograniczenia zasobów, uzupełniając te, które istnieją już zgodnie ze standardem. Ograniczenia mogą być odczytane poprzez wywołanie funkcji `getrlimit()`, a ustawione przez funkcję `setrlimit()`:

```
#include <sys/time.h>
#include <sys/resource.h>

struct rlimit
{
    rlim_t rlim_cur; /* ograniczenie miękkie */
    rlim_t rlim_max; /* ograniczenie twarde */
};
int getrlimit (int resource, struct rlimit *rlim);
int setrlimit (int resource, const struct rlimit *rlim);
```

Zasoby reprezentowane są przez stałe całkowitoliczbowe, takie jak na przykład `RLIMIT_CPU`. Struktura `rlimit` opisuje bieżące ograniczenia. Definiuje dwie górne granice: *ograniczenie miękkie* (ang. *soft limit*) oraz *ograniczenie twarde* (ang. *hard limit*). Jądro wymusza miękkie ograniczenie zasobów na procesach, lecz proces może swobodnie zmieniać je na dowolną wartość w zakresie od zera aż do granicy określanej przez ograniczenie twarde. Proces nieposiadający uprawnień `CAP_SYS_RESOURCE` (czyli niebędący procesem administratora) może jedynie obniżyć swoje ograniczenie twarde. Proces bez przywilejów nie może nigdy zwiększyć swoich ograniczeń twardej, nawet do wyższej wartości, którą poprzednio posiadał — obniżanie ograniczeń twardej jest nieodwracalne. Proces uprzywilejowany może ustawić swoje ograniczenie twarde na dowolną poprawną wartość.

Od konkretnego zasobu zależy, co naprawdę reprezentuje jego ograniczenie. Jeśli zasobem jest na przykład `RLIMIT_FSIZE`, wówczas ograniczenie opisuje maksymalny rozmiar pliku w bajtach, który proces może stworzyć. W tym przypadku, jeśli `rlim_cur` wynosi 1024, proces nie może utworzyć ani powiększyć pliku większego niż rozmiar jednego kilobajta.

Wszystkie ograniczenia zasobów systemowych posiadają dwie specjalne wartości: 0 oraz nieskończoność. Pierwsza wartość zupełnie blokuje użycie danego zasobu. Na przykład, jeśli `RLIMIT_CORE` jest równe zero, jądro nigdy nie stworzy pliku zrzutu systemowego. I odwrotnie, druga wartość likwiduje wszelkie ograniczenia dla danego zasobu. Jądro rozpoznaje nieskończoność poprzez specjalną wartość `RLIM_INFINITY`, która wynosi dokładnie `-1` (może to spowodować pewne nieporozumienia, gdyż `-1` jest również wartością zwracaną w przypadku błędu). Jeśli `RLIMIT_CORE` jest równy nieskończoności, jądro będzie tworzyć pliki zrzutu systemowego o dowolnym rozmiarze.

Funkcja `getrlimit()` pobiera aktualne ograniczenia miękkie oraz twarde dla zasobu określonego poprzez parametr `resource` i umieszcza je w strukturze opisywanej przez `rlim`. W przypadku poprawnego wywołania funkcja zwraca 0, natomiast w przypadku błędu zwraca `-1` oraz odpowiednio ustawia `errno`.

³ W tym przypadku wywołana funkcja systemowa ustawi `errno` na wartość `EMFILE`, informując, iż proces osiągnął graniczną dopuszczalną wartość dla liczby otwartych plików. Funkcja systemowa `open()` omówiona jest w rozdziale 2.

Funkcja systemowa `setrlimit()` ustawia odpowiednio ograniczenia miękkie i twarde dla zasobu `resource` na wartości wskazywane przez strukturę `rlim`. W przypadku poprawnego wywołania funkcja zwraca 0, a jądro stosownie uaktualnia ograniczenia zasobów systemowych. W przypadku błędu funkcja zwraca `-1` oraz ustawia `errno` na właściwą wartość.

Ograniczenia

Linux aktualnie udostępnia 16 ograniczeń zasobów systemowych:

RLIMIT_AS

Ogranicza w bajtach maksymalny rozmiar przestrzeni adresowej procesu. Próba zwiększenia rozmiaru przestrzeni adresowej poza to ograniczenie (poprzez funkcje systemowe takie jak `mmap()` oraz `brk()`) nie powiedzie się, a funkcje zwrócą wartość błędu `ENOMEM`. Jeśli stos procesu, który automatycznie rośnie w miarę potrzeby, przekracza to ograniczenie, jądro wysyła temu procesowi sygnał `SIGSEGV`. Limit ten wynosi zwykle `RLIM_INFINITY`.

RLIMIT_CORE

Ustala maksymalny rozmiar w bajtach pliku zrzutu systemowego (`core`). Jeśli ograniczenie jest niezerowe, pliki zrzutu systemowego większe od tej wielkości obcinane są do maksymalnego dopuszczanego rozmiaru. Jeśli ograniczenie wynosi zero, pliki zrzutu systemowego nie są nigdy tworzone.

RLIMIT_CPU

Wyznacza maksymalną ilość czasu procesora w sekundach, która może być zużyta przez proces. Jeśli proces działa dłużej niż to ograniczenie, jądro wysyła procesowi sygnał `SIGXCPU`, który może być jednak przechwycony i obsłużony przez tenże proces. Programy przenośne powinny zakończyć swoje działanie po otrzymaniu takiego sygnału, ponieważ POSIX nie definiuje żadnej akcji dla jądra po jego wysłaniu. Jednak Linux zezwala procesom, by kontynuowały swoje działanie, a następnie wysyła regularnie co jedną sekundę kolejne sygnały `SIGXCPU`. Gdy tylko nastąpi osiągnięcie granicy ograniczenia twardego, do procesu zostaje wysłany sygnał `SIGKILL`, który powoduje zakończenie jego działania.

RLIMIT_DATA

Zarządza maksymalnym rozmiarem w bajtach segmentu danych i stosu dla danego procesu. Próba zwiększenia rozmiaru segmentu danych poza to ograniczenie przy pomocy funkcji systemowej `brk()` kończy się niepowodzeniem i zwraca `ENOMEM`.

RLIMIT_FSIZE

Określa maksymalny rozmiar w bajtach dla pliku, który może być stworzony przez dany proces. Jeśli proces spróbuje rozszerzyć plik ponad tę granicę, jądro wyśle mu sygnał `SIGXFSZ`. Sygnał ten domyślnie kończy działanie procesu. Proces może jednak przechwycić go i obsłużyć, co spowoduje, że kolejne próby wywołania funkcji systemowej poszerzającej rozmiar pliku będą kończyć się niepowodzeniem i zwracać błąd `EFBIG`.

RLIMIT_LOCKS

Zarządza maksymalną liczbą blokad pliku, które mogą być w posiadaniu przez dany proces (w rozdziale 8. można zapoznać się z dyskusją na temat blokad pliku). Gdy tylko ograniczenie zostanie osiągnięte, dalsze próby otrzymania dodatkowych blokad pliku powinny kończyć się niepowodzeniem i zwracać błąd `ENOLCK`. Jądro Linuksa w wersji 2.4.25 likwiduje jednak tę cechę. Obecne wersje jądra pozwalają na ustawienie ograniczenia, lecz nie powoduje to żadnych zmian.

RLIMIT_MEMLOCK

Określa maksymalny rozmiar w bajtach dla pamięci, która może być zablokowana przy pomocy funkcji systemowych `mlock()`, `mlockall()` lub `shmctl()` przez proces nieposiadający uprawnień `CAP_SYS_IPC` (czyli tak naprawdę przez proces niebędący administratorem). Jeśli to ograniczenie zostaje przekroczone, wywołania tych funkcji kończą się błędem i zwracają kod `EPERM`. W praktyce rzeczywiste ograniczenie jest zaokrąglone w dół, do liczby całkowitej będącej wielokrotnością liczby stron. Procesy posiadające uprawnienie `CAP_SYS_IPC` mogą zablokować dowolną liczbę stron w pamięci i ograniczenie to nie powoduje w ich przypadku żadnych zmian. Zanim pojawiła się wersja jądra 2.6.9, limit ten dotyczył procesów z uprawnieniem `CAP_SYS_IPC`, a procesy bez przywilejów w ogóle nie mogły blokować żadnych stron. Ograniczenie to nie jest częścią standardu POSIX, wprowadził je system BSD.

RLIMIT_MSGQUEUE

Określa maksymalną wielkość obszaru w bajtach, który może być przydzielony przez użytkownika dla potrzeb kolejek wiadomości w standardzie POSIX. Jeśli nowo utworzona kolejka wiadomości przekroczy to ograniczenie, funkcja systemowa `mq_open()` wykona się niepoprawnie i zwróci kod błędu `ENOMEM`. Ograniczenie to nie jest częścią standardu POSIX; zostało dodane w jądrze wersji 2.6.8 i jest specyficzne dla Linuksa.

RLIMIT_NICE

Określa maksymalną wartość, do której dany proces może obniżyć swój poziom uprzejmości (czyli podnieść swój priorytet). Jak zostało to już wcześniej omówione w niniejszym rozdziale, procesy zwykle mogą jedynie zwiększać wartość swojego poziomu uprzejmości (zmniejszać swój priorytet). To ograniczenie umożliwia administratorowi ustalenie maksymalnej wartości (czyli dolnej granicy poziomu uprzejmości), do której procesy mogą poprawnie zwiększać swój priorytet. Ponieważ poziom uprzejmości mogą być ujemne, jądro interpretuje wartość jako $20 - rlim_cur$. Zatem jeśli ograniczenie ustawione jest na 40, proces może obniżyć wartość swojego poziomu uprzejmości do minimalnej wartości równej -20 (jest to jednocześnie najwyższy możliwy priorytet). To ograniczenie zostało wprowadzone w jądrze w wersji 2.6.12.

RLIMIT_NOFILE

Ustala liczbę będącą wartością o jeden większą od maksymalnej liczby deskryptorów plików, które dany proces może mieć otwarte. Próba ominięcia tego ograniczenia kończy się błędem i odpowiednia funkcja systemowa zwraca kod `EMFILE`. To ograniczenie jest również znane pod nazwą `RLIMIT_OFIle`, która pochodzi z systemu BSD.

RLIMIT_NPROC

Określa maksymalną liczbę procesów, które mogą działać, podczas gdy są uruchomione w danym momencie przez użytkownika w systemie. Próba przekroczenia tego ograniczenia kończy się niepowodzeniem, a funkcja systemowa `fork()` zwraca kod błędu `EAGAIN`. Ograniczenie to nie jest częścią standardu POSIX, wprowadził je system BSD.

RLIMIT_RSS

Określa maksymalną liczbę stron, które proces może umieścić w pamięci (ta wielkość znana też jest pod nazwą *rozmiaru grupy rezydentnej*, ang. *resident set size* — RSS). Tylko wcześniejsze wersje 2.4 jądra systemu egzekwowały to ograniczenie. Obecne wersje jądra zezwalają co prawda na ustawienie tej wartości, ale nie powoduje to żadnych zmian. To ograniczenie nie jest częścią standardu POSIX; wprowadził je system BSD.

RLIMIT_RTTIME

Określa ograniczenie (w mikrosekundach) czasu procesora, jaki może zostać zużyty przez proces czasu rzeczywistego, niewykonujący blokującego wywołania systemowego. Gdy proces wykonuje blokujące wywołanie systemowe, czas procesora zostaje wyzerowany. Chroni to system operacyjny przed zawieszeniem przez procesy czasu rzeczywistego, które wymknęły się spod kontroli. Ograniczenie zostało dodane w wersji jądra 2.6.25 i jest specyficzne dla Linuksa.

RLIMIT_RTPRIO

Określa maksymalny poziom priorytetu czasu rzeczywistego, który może być zażądany przez proces nieposiadający uprawnień `CAP_SYS_NICE` (czyli w rzeczywistości proces niebędący administratorem). Zwykle procesom bez przywilejów nie wolno żądać żadnej klasy szeregowania czasu rzeczywistego. Ograniczenie to nie jest częścią standardu POSIX; zostało dodane w wersji jądra 2.6.12 i jest specyficzne dla Linuksa.

RLIMIT_SIGPENDING

Ustala maksymalną liczbę sygnałów (standardowych oraz czasu rzeczywistego), które mogą być umieszczone w kolejce dla danego użytkownika. Próba dodania do kolejki następnym sygnałów nie powiedzie się, a funkcje systemowe takie jak `sigqueue()` zwrócą wartość błędę `EAGAIN`. Należy zwrócić uwagę, że bez względu na to ograniczenie zawsze możliwe jest dodanie do kolejki jednego egzemplarza sygnału o typie, który jeszcze się w niej nie znajduje. Dzięki temu zawsze istnieje możliwość wysłania procesowi sygnałów takich jak `SIGKILL` czy `SIGTERM`. Ograniczenie to nie jest częścią standardu POSIX; jest specyficzne dla Linuksa.

RLIMIT_STACK

Ustala maksymalny rozmiar w bajtach stosu dla procesu. Próba ominięcia tego ograniczenia kończy się wysłaniem sygnału `SIGSEGV`.

Jądro zarządza ograniczeniami zasobów, przydzielając je procesom. Proces potomny dziedziczy ograniczenia po swoim rodzicu podczas operacji rozwidlenia. Ograniczenia są zachowywane w trakcie użycia funkcji typu *exec*.

Ograniczenia domyślne

Ograniczenia domyślne dostępne dla procesu zależą od trzech zmiennych: wstępnego ograniczenia miękkiego, wstępnego ograniczenia twardego oraz administratora systemu. Jądro narzuca wstępne ograniczenia miękkie i twarde, pokazane w tabeli 6.1. Jądro ustawia te ograniczenia dla procesu `init`, a ponieważ potomkowie dziedziczą ograniczenia od swoich rodziców, wszystkie dalsze procesy przejmują ograniczenia miękkie i twarde od procesu `init`.

Dwa przypadki mogą zmienić te domyślne wartości ograniczeń:

- Każdy proces może swobodnie zwiększyć ograniczenie miękkie do dowolnej wartości od 0 do wielkości ograniczenia twardego. Potomkowie będą dziedziczyć te uaktualnione ograniczenia podczas rozwidlenia procesu.
- Proces uprzywilejowany może swobodnie ustawiać ograniczenie twarde na dowolną wartość. Potomkowie odziedziczą te uaktualnione ograniczenia podczas rozwidlenia procesu.

Tabela 6.1. Domyślne miękkie i twarde ograniczenia zasobów systemowych

Symbol ograniczenia	Ograniczenie miękkie	Ograniczenie twarde
RLIMIT_AS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_CORE	0	RLIM_INFINITY
RLIMIT_CPU	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_DATA	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_FSIZE	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_LOCKS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_MEMLOCK	8 stron	8 stron
RLIMIT_MSGQUEUE	800 kB	800 kB
RLIMIT_NICE	0	0
RLIMIT_NOFILE	1024	1024
RLIMIT_NPROC	0 (oznacza brak ograniczeń)	0 (oznacza brak ograniczeń)
RLIMIT_RSS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_RTPRIO	0	0
RLIMIT_SIGPENDING	0	0
RLIMIT_STACK	8 MB	RLIM_INFINITY

Jest raczej mało prawdopodobne, że prawidłowo utworzony proces administratora zmieni jakies ograniczenia twarde. Dlatego też pierwszy przypadek jest duzo bardziej prawdopodobnym źródłem modyfikacji ograniczeń niż drugi. Istotnie, faktyczne ograniczenia udostępniane procesowi są w większości ustawiane przez powłokę systemową użytkownika, która może być w taki sposób dopasowana przez administratora, aby ustalać różne restrykcje. Na przykład, w powłoce systemowej Bourne-again shell (*bash*) administrator uzyskuje dostęp do parametrów ograniczeń dzięki komendzie *ulimit*. Należy zwrócić uwagę, że administrator nie potrzebuje niższych wartości; może on również podnieść ograniczenia miękkie do poziomu ograniczeń twardych, dostarczając użytkownikom rozsądniejszych wielkości domyślnych. Jest to często stosowane w przypadku ograniczenia `RLIMIT_STACK`, które w wielu systemach ustawia się na wartość `RLIM_INFINITY`.

Ustawianie i odczytywanie ograniczeń

Definicje różnych ograniczeń systemowych zostały już objaśnione, dlatego też można przystąpić do ich odczytywania i ustawiania. Odczytywanie wartości ograniczenia systemowego jest całkiem proste:

```

struct rlimit rlim;
int ret;

/* pobierz ograniczenie dla rozmiarów pliku zrzutu systemowego */
ret = getrlimit (RLIMIT_CORE, &rlim);
if (ret == -1)
{
    perror ("getrlimit");
    return 1;
}
printf ("Ograniczenie RLIMIT_CORE: miękkie=%ld twarde=%ld\n", rlim.rlim_cur,
rlim.rlim_max);

```

Skompilowanie tego fragmentu kodu i jego uruchomienie udostępnia następujący wynik:

```
Ograniczenie RLIMIT_CORE: miękkie=0 twarde=-1
```

Ograniczenie miękkie ustawione jest na 0, natomiast twarde na nieskończoność (-1 oznacza RLIM_INFINITY). Dlatego też możliwe jest ustawienie limitu miękkiego na dowolną wartość. Poniższy przykład ustawia maksymalny rozmiar pliku zrzutu systemowego na 32 MB:

```
struct rlimit rlim;
int ret;
rlim.rlim_cur = 32 * 1024 * 1024; /* 32 MB */
rlim.rlim_max = RLIM_INFINITY; /* zostawiamy tak, jak jest */
ret = setrlimit (RLIMIT_CORE, &rlim);
if (ret == -1)
{
    perror ("setrlimit");
    return 1;
}
```

Kody błędów

W przypadku błędów możliwe są trzy wartości errno:

EFAULT

Obszar pamięci wskazywany przez rlim jest błędny lub niedostępny.

EINVAL

Wartość określona przez parametr resource jest nieprawidłowa albo wartość rlim.rlim_cur jest większa od rlim.rlim_max (tylko dla setrlimit()).

EPERM

Proces wywołujący nie posiadał uprawnień CAP_SYS_RESOURCE, a próbował zwiększyć ograniczenie twarde.

#define, 41, 42
/bin/init, 156
/bin/sh, 156
/dev/full, 289
/dev/null, 289
/dev/random, 289
/dev/urandom, 289
/dev/zero, 88, 289, 318
/etc/group, 38, 157
/etc/init, 156
/etc/passwd, 38, 157
/sbin/init, 156
__attribute__, 411
__builtin_return_address, 402, 410
_exit(), 165, 166
_IOFBF, 104
_IOLBF, 104
_IONBF, 104
_POSIX_SAVED_IDS, 184
_SC_ATEXIT_MAX, 168
_SC_PAGESIZE, 126
_XOPEN_SOURCE, 386
1:1 threading, 229

A

ABI, 17, 25, 26
abort(), 340, 342
ABS, 205
absolute pathname, 32, 270
absolute section, 36
abstrakcja, 22
abstrakcja programistyczna, 226
access control lists, 39
ACL, 39
addr, 123
adjtime(), 383
adjtimex(), 384, 385
administrator, 38

adresowanie
 bloków logicznych, 143
 CHS, 142
 dysku, 142
advice, 134, 137
aio, 141
aktualny czas, 376
 ustawianie, 379
aktualny katalog roboczy, 32, 270
alarm(), 342, 392, 394
alarm_handler(), 392
alarmy, 392
algorytm cykliczny, 146
aligned, 406
alignment, 311
alignof, 408
alloca(), 324, 325, 326
alternatywny
 stan uśpienia, 391
 stos sygnałowy, 360
American National Standards Institute, 28
anonimowe odwzorowanie w pamięci, 315, 316
ANSI C, 28
Anticipatory I/O Scheduler, 146
antilock braking system, 205
anulowanie, 244
 asynchroniczne, 245
API, 21, 25
API Pthreads, 241
aplikacja, 155
 odpowiedzialna, 179
 sposób działania stronicowania systemowego,
 332
 ściśłego czasu rzeczywistego, 206
 wstrzymanie wykonania, 75
 związana z procesorem, 195
 związana z wejściem i wyjściem, 195
 zwykłego czasu rzeczywistego, 206
aplikacje GUI, 21, 195

- append mode, 47
- Application Binary Interface, 25
- Application Programming Interface, 21, 25
- architektura i386, 24
- arytmetyka wskaźników do funkcji, 411
- asctime(), 381
- asctime_r(), 381
- asynchroniczne operacje, 140, 141
 - aio, 141
- async-safe, 40
- atexit(), 166, 167, 168
- atrybuty autoryzacji, 39
- atrybuty rozszerzone, 261
 - klucze, 262
 - lista atrybutów, 267
 - odczyt, 264
 - operacje, 264
 - przestrzenie nazw, 263
 - security, 263
 - system, 263
 - trusted, 263
 - user, 263
 - ustawianie, 265
 - usuwanie, 268
 - wartości, 262
- atrybuty wątku
 - obiekt pthread_attr_t, 242
- automatyczne przechwytywanie potomków, 360
- automatyczne zmienne, 324

B

- bajty, 301
 - manipulowanie, 332
 - porównywanie, 329
 - przenoszenie, 330
 - ustawianie wartości, 329
- bash, 223
- batch scheduling policy, 209
- bcmp(), 330
- bcopy(), 330
- bdflush, 84
- bezpieczeństwo, 333
 - wątków, 105
- bezpośrednie operacje wejścia i wyjścia, 64
 - O_DIRECT, 64
- biblioteka
 - aio, 141
 - C++, 25
 - GNU C, 17
 - iostream, 90
 - języka C, 24
 - libpthread, 241
 - libstdc++, 25

- typowych operacji wejścia i wyjścia, 90
 - analiza, 108
 - bezpieczeństwo wątków, 105
 - błędy, 102
 - czytanie ze strumienia, 93
 - koniec pliku, 102
 - opróżnianie strumienia, 102
 - otwieranie plików, 91
 - otwieranie strumienia, 92
 - parametry buforowania, 104
 - podwójne kopiowanie, 108
 - skojarzony deskryptor pliku, 103
 - szukanie w strumieniu, 100
 - użycie, 109
 - wskaźniki do plików, 90
 - zamykanie strumienia, 93
 - zapis do strumienia, 97
- wątkowości, 241
- binaria, 225
- bit NX, 124
- bitowy zestaw uprawnień, 49
- bity uprawnień, 39
- block devices, 34
- block started by symbol, 36
- blocked, 193
- blok, 35, 87, 143
 - częściowe operacje, 87
 - rozmiary, 89
 - rozpoczęty od symbolu, 36
 - wpływ wielkości na wydajność, 88
- blok fizyczny, 142
- blok logiczny, 143
- blok systemu plików, 143
- blok urządzenia, 142
- blokada, 236
 - typu mutex, 237
 - w kontekście wątków, 236
- blokada strumienia, 107
- blokady
 - globalne, 251
 - o określonym zakresie widoczności, 250
 - the_mutex, 251
 - zależne, 250
- blokowane operacje wejścia i wyjścia, 227
- blokowanie, 105
 - całej przestrzeni adresowej, 334
 - danych, 237
 - drobnoziarniste, 251
 - fragmentu przestrzeni adresowej, 333
 - muteksów, 249
 - odczytów przez zapisy, 144
 - pamięci, 216, 332, 335
 - plików, 106
 - sygnałów, 357

- błędy, 40
 - opisy, 41, 42
 - segmentacji, 344
 - strony, 302
 - boundary, 312
 - Bourne shell, 156
 - Bourne-again shell, 223
 - break point, 315
 - brk(), 220, 315
 - broken link, 33
 - bss, 303
 - buddy memory allocation scheme, 316
 - buf, 52
 - buffer_head, 84
 - bufor
 - brudny, 59, 84
 - funkcje systemowe, 24
 - maksymalny wiek, 60
 - obsługiwany przez bibliotekę języka C, 102
 - podręczny katalogu, 32
 - rozmiar, 89
 - synchronizacja na dysku, 62
 - zdeaktualizowany, 60
 - bufor stron, 81, 82
 - lokalizacja sekwencyjna, 83
 - obcinanie, 83
 - odczyt z wyprzedzeniem, 83
 - przerzucanie stron, 83
 - rozmiar, 82
 - buforowane operacje wejścia i wyjścia, 87
 - bezpieczeństwo wątków, 105
 - blokowanie plików, 106
 - błędy, 102
 - brak buforowania, 104
 - buforowanie
 - blokowe, 104
 - pełne, 104
 - w przestrzeni użytkownika, 87
 - wierszowe, 104
 - czytanie
 - całego wiersza, 94
 - danych binarnych, 96
 - dowolnych łańcuchów, 95
 - ze strumienia, 93
 - deskryptor pliku, 103
 - informacja o aktualnym położeniu
 - w strumieniu, 101
 - koniec pliku, 102
 - nieblokowane operacje na strumieniu, 107
 - opróżnianie strumienia, 102
 - otwieranie
 - plików, 91
 - strumienia, 92
 - parametry buforowania, 104
 - przykładowy program, 99
 - rozmiar bloku, 88, 89
 - szukanie w strumieniu, 100
 - wycofywanie znaku, 94
 - zamykanie strumieni, 93
 - zapis
 - danych binarnych, 98
 - do strumienia, 97
 - łańcucha znaków, 98
 - pojedynczego znaku, 97
 - buforowanie
 - blokowe, 104
 - pełne, 104
 - przeprowadzane przez jądro, 102
 - tryby, 104
 - w przestrzeni użytkownika, 88
 - wierszowe, 104
 - zapisów, 60
 - BUFSIZ, 105
 - busy-wait, 216
 - bzero(), 329
- ## C
- C, 28
 - C99, 402
 - cache effects, 202
 - call(), 240
 - calloc(), 306, 318
 - całkowity czas oczekiwania dla operacji wejścia i wyjścia, 63
 - cancellation, 244
 - CAP_CHOWN, 259
 - CAP_FOWNER, 258
 - CAP_IPC_LOCK, 335
 - CAP_KILL, 352
 - CAP_SYS_ADMIN, 263
 - CAP_SYS_NICE, 199, 200, 222
 - CAP_SYS_RAWIO, 153
 - CAP_SYS_RESOURCE, 219
 - case, 410
 - CD-ROM, 34
 - CDROMEJECT, 291
 - CFQ, 146, 202
 - cfree(), 310
 - CFS, 194
 - character devices, 34
 - chdir(), 190, 273
 - chmod(), 257, 258
 - chown(), 259
 - chrt, 216
 - CHS, 142, 143
 - CHS addressing, 142

- chwila, 370
- clearerr(), 103
- clock_getres(), 375
- clock_gettime(), 377, 380, 389
- CLOCK_MONOTONIC, 374, 375
- clock_nanosleep(), 389
- CLOCK_PROCESS_CPUTIME_ID, 375
- CLOCK_REALTIME, 374, 389
- clock_settime(), 380
- clock_t, 374
- CLOCK_THREAD_CPUTIME_ID, 375
- clockid_t, 374
- CLOCKS_PER_SEC, 371
- clone(), 229, 240
- close(), 65
 - kody błędów, 65
- closedir(), 279
- close-on-exec, 47, 117, 292
- command, 177
- Complete Fair Queuing I/O Scheduler, 146
- Completely Fair Scheduler, 194, 196
- concurrency, 233
- congestion avoidance, 84
- const, 314, 403
- constant function, 403
- cooperative, 194
- copy-on-write, 164
- core, 220
- coroutines, 230
- count, 56, 114
- COW, 164, 303
- CPU_CLR, 204
- CPU_SET, 204
- CPU_SETSIZE, 204
- CPU_ZERO, 204
- creat(), 51
 - wartości zwracane, 52
- critical region, 233
- ctime(), 381
- ctime_r(), 381
- current working directory, 270
- cwd, 270
- czas, 369
 - absolutny, 370
 - aktualny, 376
 - ustawianie, 379
 - alternatywy stanu uśpienia, 391
 - clock_t, 374
 - dokładność na poziomie mikrosekund, 372
 - dokładność na poziomie nanosekund, 372
 - GMT, 370
 - interfejs ustawiania czasu, 380
 - jądra, 63
 - konwersje, 381
 - licznik chwil, 370
 - liczniki, 392
 - monotoniczny, 369
 - obsługa stanu uśpienia, 387
 - opóźnienie licznika, 392
 - precyzyjne ustawianie czasu, 379
 - procesu, 369, 378
 - przepełnienia, 391
 - rozdzielczość źródła czasu, 375
 - rzeczywisty, 205, 369
 - stan oczekiwania, 385
 - stan uśpienia, 385, 386
 - struktury danych, 371
 - reprezentacja pierwotna, 372
 - systemowy, 369
 - time_t, 372
 - timespec, 372
 - timeval, 372
 - timex, 384
 - tm, 373
 - UTC, 370
 - użytkowy, 369
 - użytkownika, 63
 - wprowadzanie w stan uśpienia, 390
 - wygaśnięcia licznika, 392
 - wyluskiwanie składników, 373
 - względny, 370
 - zarządzanie stanem uśpienia, 389
 - zegar
 - POSIX, 374
 - sprzętowy, 371
 - systemowy, 370, 382
- czasowa lokalizacja, 82
- czytanie
 - całego wiersza, 94
 - danych binarnych, 96
 - pojedynczego znaku, 93
 - z pliku, 52
 - wszystkie bajty, 54
 - ze strumienia, 93
 - całe wiersze, 94
 - dane binarne, 96
 - dowolne łańcuchy, 95
 - katalogu, 278
 - pojedyncze znaki, 93

D

- daemon(), 191
- dane
 - lokalne dla wątku, 105
 - synchronizowanie dostępu, 105

dangling symlink, 283
 data race, 233
 date, 379
 dd, 88
 Deadline I/O Scheduler, 144, 145
 deadlock, 238
 demony, 185, 189
 dentry, 32
 dentry cache, 32
 deprecated, 405
 deskryptor, 31

- elementu obserwowanego, 293
- pliku, 30, 45, 72
 - otwieranie strumienia, 92
 - tablice, 73
- procesu, 36
- wątki, 70

 determinizm, 216, 332
 difftime(), 382
 directory, 269
 directory entry, 269
 directory resolution, 32
 directory stream, 278
 dirfd(), 278
 dnotify, 292
 docelowe opóźnienie, 196
 domyślna powłoka użytkownika, 185
 dostęp

- do pliku, 30
- równoległy, 31

 dostrajanie zegara systemowego, 382
 dowiązania, 31, 269, 280

- licznik użycia, 281
- miękkie, 282
- plik docelowy, 282
- symboliczne, 33, 281, 282
- twarde, 32, 280, 281
- tworzenie, 281, 283
- uszkodzone, 33

 drugoplanowa grupa procesów, 185
 drzewo

- katalogów, 32
- procesów, 37

 duch, 179
 dynamiczne przydzielanie pamięci, 306
 dyski SSD, 147
 dziedziczenie priorytetu, 239

E

E2BIG, 41, 161
 EACCESS, 201, 255
 EACCESS, 41, 127, 132, 161

EAGAIN, 41, 55, 127, 131, 136, 163, 181, 183
 EBADF, 41, 55, 58, 61, 65, 67, 74, 78, 100, 120, 121, 127, 136, 255, 258
 EBUSY, 41
 ebx, 24
 ECHILD, 41, 169, 172
 ECHLD, 175
 ecx, 24
 EDEADLK, 247, 249
 edge-triggered, 122
 edi, 24
 EDOM, 41
 EDQUOT, 266
 edx, 24
 edytor tekstowy

- Unix, 15

 EEXIST, 120, 266
 EEXIT, 41
 EFAULT, 41, 55, 58, 78, 121, 131, 161, 205, 255, 258
 EFBIG, 41, 58
 efektywny

- GID, 38
- identyfikator użytkownika, 38, 180
- UID, 38

 effective UID, 38
 EINTR, 41, 53, 74, 78, 121, 169, 173, 175
 EINVAL, 41, 55, 58, 62, 67, 74, 78, 100, 113, 118, 120, 121, 127, 131, 132, 134, 136, 153, 173, 175, 188, 201, 205, 247, 249, 335
 EIO, 41, 56, 58, 62, 65, 136, 162, 258
 EISDIR, 41, 162
 element katalogu, 269
 ELF, 36

- sekcja bss, 36
- sekcja danych, 36
- sekcja tekstu, 36
- sekcje, 36
- sekcje absolutne, 36
- sekcje niezdefiniowane, 36

 ELOOP, 162, 255, 258
 embedded, 207
 EMFILE, 41, 118, 162
 EMLINK, 41
 ENAMETOOLONG, 255, 258
 ENFILE, 41, 118, 127, 162
 ENODEV, 41, 127
 ENOENT, 41, 120, 162, 255, 258
 ENOEXEC, 41, 162
 ENOMEM, 41, 74, 78, 118, 120, 127, 131, 133, 134, 136, 162, 163, 255, 258, 335
 ENOSPC, 41, 58, 267, 289
 ENOTDIR, 41, 162, 255, 258, 265, 267
 ENOTSUP, 265, 267

ENOTTY, 42
entropy pool, 289
ENXIO, 42
EOF, 34, 53, 103
E_OVERFLOW, 67, 127
E_PERM, 42, 120, 128, 162, 181, 183, 187, 188, 201, 205, 249, 258, 335
EPIPE, 42, 58
epoll, 111, 117
EPOLL_CLOEXEC, 117
epoll_create(), 117
epoll_create1(), 117, 118
epoll_ctl(), 118, 122
EPOLL_CTL_ADD, 119
EPOLL_CTL_DEL, 119, 120
EPOLL_CTL_MOD, 119
epoll_wait(), 121, 122
EPOLLERR, 119
EPOLLET, 119, 122
EPOLLHUP, 119
EPOLLIN, 119
EPOLLONESHOT, 119
EPOLLOUT, 119
EPOLLPRI, 119
ERANGE, 42, 265
EROFS, 42, 258, 282
errno, 40, 43, 52
errno, 42
esi, 24
ESPIPE, 42, 68
ESRCH, 42, 187, 188, 201, 205, 244, 247
ETXTBSY, 42, 162
euid, 182, 183
EUID, 38
event loop, 232
event poll, 117
EXDEV, 42, 282
exec(), 158
execl(), 159, 160
 atrbuty procesu, 159
execle(), 160
execlp(), 160
 problem bezpieczeństwa, 161
executing, 158
execv(), 160
execve(), 160, 161
execvp(), 160, 161
 problem bezpieczeństwa, 161
exit(), 166, 167, 190
EXIT_FAILURE, 166
EXIT_SUCCESS, 166
ext3, 263
ext4, 35, 261

extern, 41
external fragmentation, 316

F

fast bins, 320
FAT, 35
fchdir(), 273
fchmod(), 257, 258
fchown(), 259
fclose(), 93
fcloseall(), 93
fcntl(), 47
fd, 52
FD_CLR, 73
FD_ISSET, 73
FD_SET, 73
FD_SETSIZE, 73
FD_ZERO, 73
fdatasync(), 60, 61, 62
 wartości zwracane, 61
fdopen(), 92
fds, 45
feof(), 96, 103
ferror(), 96, 103
fflush(), 102
fgetc(), 93, 95, 97, 108
fgetpos(), 101
fgets(), 94, 95
fgetxattr(), 264
fibers, 230
FIBMAP, 151, 153
FIFO, 34, 208
FILE, 90, 92
file descriptor, 30
file offset, 30
file pointer, 90
file position, 30
file table, 45
fileno(), 103
filesystem, 35
filesystem UID, 38
find_file_in_dir(), 279
fine-grained locking, 251
first in, first out class, 208
flags, 46
flistxattr(), 267
flockfile(), 106, 107
fopen(), 91, 92
fork(), 37, 158, 162, 190
forking, 158
format wykonywalny, 36
forward compatibility, 29
fputc(), 97, 98

fputs(), 98
 fragmentacja
 wewnętrzna, 316
 zewnętrzna, 316
 fread(), 96
 Free Software Foundation, 27
 Free Standards Group, 29
 free(), 309
 fremovexattr(), 268
 fseek(), 100, 101
 fsetpos(), 100
 fsetxattr(), 265, 266
 fstat(), 254
 fsync(), 60, 62
 wartości zwracane, 61
 ftell(), 101
 ftruncate(), 69
 ftrylockfile(), 107
 full device, 289
 fully qualified, 32
 funkcje
 _exit(), 165, 166
 abort(), 340, 342
 adjtime(), 383
 adjtimex(), 384, 385
 alarm(), 342, 392, 394
 alarm_handler(), 392
 alloca(), 324, 325, 326
 asctime(), 381
 asctime_r(), 381
 atexit(), 166, 167, 168
 bcmp(), 330
 bcopy(), 330
 bezpieczne dla sygnałów, 354
 brk(), 220, 315
 bzero(), 329
 call(), 240
 calloc(), 306, 318
 cfree(), 310
 chdir(), 190, 273
 chmod(), 257, 258
 chown(), 259
 clearerr(), 103
 clock_getres(), 375
 clock_gettime(), 377, 380, 389
 clock_nanosleep(), 389
 clock_settime(), 380
 close(), 65
 closedir(), 279
 creat(), 51, 52
 ctime(), 381
 ctime_r(), 381
 czyste, 403
 daemon(), 191
 difftime(), 382
 dirfd(), 278
 epoll_create(), 117
 epoll_create1(), 117, 118
 epoll_ctl(), 118
 epoll_wait(), 121
 exec(), 158
 execl(), 159
 execle(), 160
 execlp(), 160
 execv(), 160
 execve(), 160, 161
 execvp(), 160, 161
 exit(), 166, 167, 190
 fchdir(), 273
 fchmod(), 257, 258
 fchown(), 259
 fclose(), 93
 fcloseall(), 93
 fcntl(), 47
 fdatasync(), 60, 62
 fdopen(), 92
 feof(), 96, 103
 ferror(), 96, 103
 fflush(), 102
 fgetc(), 93, 95, 97, 108
 fgetpos(), 101
 fgets(), 94, 95
 fgetxattr(), 264
 fileno(), 103
 find_file_in_dir(), 279
 flistxattr(), 267
 flockfile(), 106, 107
 fopen(), 91, 92
 fork(), 37, 158, 162, 190
 fputc(), 97, 98
 fputs(), 98
 fread(), 96
 free(), 309
 fremovexattr(), 268
 fseek(), 100, 101
 fsetpos(), 100
 fsetxattr(), 265, 266
 fstat(), 254
 fsync(), 60, 62
 ftell(), 101
 ftruncate(), 69
 ftrylockfile(), 107
 funlockfile(), 106, 107
 get_block(), 153
 get_current_dir_name(), 272
 get_inode(), 150
 get_nr_blocks(), 153
 get_thread_area(), 23

funkcje

getcwd(), 271, 272
getdents(), 280
getegid(), 184
geteuid(), 184
getgid(), 184
getitimer(), 392, 393
getpagesize(), 126, 312
getpgid(), 188
getpgrp(), 189
getpid(), 158
getpriority(), 200, 201
getrlimit(), 219
gets(), 108
getsid(), 187, 188
gettimeofday(), 377, 380
getuid(), 184
getwd(), 272
getxattr(), 264
gmtime(), 381
gmtime_r(), 381
inline, 401, 402
inotify_add_watch(), 293
inotify_init1(), 292
inotify_rm_watch(), 299
ioctl(), 151, 290
ioprio_get(), 202
ioprio_set(), 202
kill(), 345, 351
killpg(), 353
lchown(), 259
lgetxattr(), 264
link(), 281
listxattr(), 267
llistxattr(), 267
localtime(), 382
localtime_r(), 382
lock(), 237
lremovexattr(), 268
lseek(), 66
lsetxattr(), 265, 266
lstat(), 254, 283
madvise(), 134
mallinfo(), 323
malloc(), 304, 306
malloc_stats(), 324
malloc_trim(), 322
malloc_usable_size(), 322
mallopt(), 319, 321
memalign(), 312
memchr(), 331
memcmp(), 329
memcpy(), 331
memfrob(), 332
memmem(), 332
memmove(), 330
memcpy(), 331
memrchr(), 331
memset(), 307, 329
mincore(), 336
mkdir(), 275
mktime(), 381
mlock(), 221, 333
mlockall(), 221, 334
mmap(), 123, 130, 317
mprotect(), 132
mq_open(), 221
mremap(), 131
msync(), 133
munlock(), 335
munlockall(), 335
munmap(), 128, 317, 318
nanosleep(), 387
nice(), 199
on_exit(), 166, 168
open(), 24, 46, 52
open_sysconf(), 327
opendir(), 47, 278
oznaczenie, 405
pause(), 347
perror(), 42, 54
poll(), 76, 79, 80
posix_fadvise(), 137
posix_memalign(), 311
ppoll(), 80
pread(), 68, 69
przydzielanie pamięci, 404
pselect(), 75
psignal(), 351
pthread_cancel(), 244
pthread_create(), 241
pthread_detach(), 247
pthread_equal(), 243
pthread_exit(), 244
pthread_join(), 246
pthread_mutex_lock(), 249, 251
pthread_mutex_unlock(), 249, 251
pthread_self(), 243
pthread_setcancelstate(), 245
pthread_setcanceltype(), 245
Pthreads, 241
ptrace(), 170
pure, 403
pwrite(), 68, 69
raise(), 353
read(), 23, 52
readahead(), 137, 138
readdir(), 278, 280

readv(), 112, 115
 realloc(), 307
 remove(), 285
 removexattr(), 268
 rename(), 286
 rewind(), 101
 rmdir(), 276, 285
 rodzina exec
 elementy, 160
 kody błędów, 161
 sbrk(), 315
 sched_get_priority_max(), 213
 sched_get_priority_min(), 213
 sched_getaffinity(), 203
 sched_getparam(), 211, 212
 sched_getscheduler(), 210
 sched_rr_get_interval(), 214
 sched_setaffinity(), 203
 sched_setparam(), 211
 sched_setscheduler(), 210
 sched_yield(), 197, 198
 select(), 71, 74, 80, 390
 set_tid_address(), 23
 setegid(), 182
 seteuid(), 182, 183, 184
 setgid(), 181
 setitimer(), 342, 344, 345, 392, 393
 setpgid(), 187
 setpgrp(), 189
 setpriority(), 200, 201
 setregid(), 182
 setresgid(), 183
 setresuid(), 183
 setreuid(), 182
 setrlimit(), 219, 224
 setsid(), 186, 190
 settimeofday(), 379, 380
 setuid(), 181, 184
 setvbuf(), 104, 105
 setxattr(), 265
 shmctl(), 221
 sigaction(), 169, 359
 sigaddset(), 356
 sigandset(), 356
 sigdelset(), 356
 sigemptyset(), 356
 sigfillset(), 356
 sigisemptyset(), 356
 sigismember(), 356
 signal(), 169, 346, 347, 359
 signalstack(), 360
 sigorset(), 356
 sigpending(), 358
 sigprocmask(), 357, 358
 sigqueue(), 366
 sigsuspend(), 358
 sizeof(), 409
 sleep(), 385
 stale, 403
 start_routine(), 242
 stat(), 151, 253, 255
 stime(), 379
 strdupa(), 326
 strerror(), 42
 strerror_r(), 42
 strndupa(), 326
 strsignal(), 351
 symlink(), 283
 sync(), 62
 sysconf(), 126, 168
 system(), 177
 time(), 376
 timer_create(), 394, 395, 397
 timer_delete(), 394, 399
 timer_getoverrun(), 398
 timer_gettime(), 397
 timer_settime(), 394, 396
 times(), 378
 tmpfile(), 166
 tryb uśpienia, 53
 typeof(), 408
 ungetc(), 94
 unlink(), 284
 unlock(), 237
 usleep(), 386
 valloc(), 312
 vfork(), 165
 wait(), 169, 170, 171, 343
 wait3(), 175
 wait4(), 175
 waitid(), 173
 waitpid(), 170, 171, 175, 378
 write(), 23, 56
 writew(), 109, 112, 114
 xmalloc(), 307
 funkcje nieblokujące, 107
 funkcje nieużywane, 405
 funkcje niezalecane, 405
 funkcje obsługi sygnału, 40
 funkcje systemowe, 23
 procesy, 36
 wywoływanie, 23
 funkcje wpłatane, 402
 funkcje współużywalne, 354
 funlockfile(), 106, 107
 futex, 198

G

gcc, 17, 24
GCC, 401
generator liczb losowych, 289
get_block(), 153
get_current_dir_name(), 272
get_inode(), 150
get_nr_blocks(), 153
get_thread_area(), 23
getcwd(), 271, 272
getdents(), 280
getegid(), 184
geteuid(), 184
getgid(), 184
getitimer(), 392, 393
getpagesize(), 126, 312
getpgid(), 188
getpgrp(), 189
getpid(), 158
getpriority(), 200, 201
getrlimit(), 219
gets(), 108
getsid(), 187, 188
gettimeofday(), 377, 380
getuid(), 184
getwd(), 272
getxattr(), 264
GID, 38
 grupy właścicielskie, 49
glibc, 17, 24, 29, 240
global register variables, 407
globalne zmienne rejestrowe, 407
główny system plików, 35
GMT, 370
gmtime(), 381
gmtime_r(), 381
gniazda, 34
GNU C, 401
GNU C Compiler, 24
GNU Compiler Collection, 24, 401
GNU libc, 24
Go-routines, 231
graniczne parametry operacyjne, 205
group ID, 38
groups, 38
grupy, 38, 179
 /etc/group, 38
 dodatkowe, 38
 GID, 38
 identyfikator, 38
 podstawowa, 38
 procesów, 157, 184

 drugoplanowe, 185
 identyfikator, 184, 187
 lider, 184
 obsługa, 187
 pierwszoplanowe, 185
 przestarzałe funkcje obsługi, 188
 sesji, 184
 wheel, 157
 właścicielskie, 49
 procesu, 157
GUI, 21

H

handler, 23
hard affinity, 203
hard limit, 219
hard link, 32, 280
hard real-time system, 205
hasła, 38
hierarchia procesów, 37, 157
hole, 67
hooks, 81
HP-UX, 183
hwclock, 371
hybrid threading, 230

I

I/O control, 290
I/O scheduler, 81
I/O schedulers, 142
I/O-bound, 195
i386, 24
id_t, 174
identyfikator
 grupy, 38, 180
 grupy procesów, 184, 187
 i-węzła, 149
 procesu, 37, 156
 otrzymywanie, 158
 przydział, 156
 rodzicielskiego, 157, 158
 pthread_t, 243
 sesji, 185, 186, 187
 sygnału, 340
 użytkownika, 38, 180
 dla systemu plików, 38
 wątku, 243
 porównywanie, 243
idle scheduling policy, 209
IEEE, 27
IEEE Std 1003.1-1990, 27

- ignorowanie sygnału, 340
- implementacja
 - API, 25
 - NPTL, 240
 - wątków, 37
- implementacje wątkowości w Linuksie, 240
- IN_ACCESS, 294
- IN_ALL_EVENTS, 295
- IN_ATTRIB, 294
- IN_CLOEXEC, 292
- IN_CLOSE, 295
- IN_CLOSE_NOWRITE, 294
- IN_CLOSE_WRITE, 294
- IN_CREATE, 294
- IN_DELETE, 294
- IN_DELETE_SELF, 294
- IN_DONT_FOLLOW, 298
- IN_IGNORED, 297, 299
- IN_ISDIR, 297
- IN_MASK_ADD, 298
- IN_MODIFY, 294
- IN_MOVE, 295
- IN_MOVE_SELF, 294
- IN_MOVED_FROM, 294, 298
- IN_MOVED_TO, 294, 298
- IN_NONBLOCK, 292
- IN_ONESHOT, 298
- IN_ONLYDIR, 299
- IN_OPEN, 294
- IN_Q_OVERFLOW, 297
- IN_UNMOUNT, 297
- informacje
 - o aktualnym położeniu w strumieniu, 101
 - o pliku, 254
- inicjalizowanie
 - licznika, 396
 - muteksów, 248
- inicjowanie przy pozyskaniu zasobu, 250
- init, 156
- init process, 37
- inline function, 402
- ino, 31
- inode, 31
- inode number, 31
- inotify, 292
 - elementy obserwowane, 293
 - deskryptor, 293
 - dodawanie, 293
 - maska, 293, 294
 - usuwanie, 299
 - IN_ACCESS, 294
 - IN_ALL_EVENTS, 295
 - IN_ATTRIB, 294
 - IN_CLOSE, 295
 - IN_CLOSE_NOWRITE, 294
 - IN_CLOSE_WRITE, 294
 - IN_CREATE, 294
 - IN_DELETE, 294
 - IN_DELETE_SELF, 294
 - IN_DONT_FOLLOW, 298
 - IN_IGNORED, 297, 299
 - IN_ISDIR, 297
 - IN_MASK_ADD, 298
 - IN_MODIFY, 294
 - IN_MOVE, 295
 - IN_MOVE_SELF, 294
 - IN_MOVED_FROM, 294, 298
 - IN_MOVED_TO, 294, 298
 - IN_ONESHOT, 298
 - IN_ONLYDIR, 299
 - IN_OPEN, 294
 - IN_Q_OVERFLOW, 297
 - IN_UNMOUNT, 297
 - inicjalizacja interfejsu, 292
 - łączenie zdarzeń przenoszenia, 298
 - odczytywanie zdarzeń, 296
 - rozmiar kolejki zdarzeń, 300
 - usuwanie egzemplarza interfejsu, 300
 - zaawansowane opcje obserwowania, 298
 - zasysanie, 296
 - zdarzenia, 295
 - zaawansowane, 297
- inotify_add_watch(), 293
- inotify_event, 295, 300
- inotify_init1(), 292
- inotify_rm_watch(), 299
- int, 23, 30, 45
- interfejs
 - binarny aplikacji, 17, 25, 26
 - odpytywania zdarzeń, 117
 - epoll_create(), 117
 - epoll_ctl(), 118
 - epoll_wait(), 121, 122
 - oczekiwanie na zdarzenie, 121
 - sterowanie działaniem, 118
 - tworzenie egzemplarza, 117
 - zdarzenia przełączane poziomem, 122
 - zdarzenia przełączane zboczem, 122
- programistyczny dla standardu Pthreads, 240
- programowania aplikacji, 25
- programowy, 24
- systemowy, 29
 - Linuksa, 17
 - standardy, 27
- systemu operacyjnego
 - przenośny, 27
- ustawiania czasu, 380
- źródłowy, 25

- internal fragmentation, 316
- interprocess communication, 34
- interval timers, 392
- intmax_t, 158
- i-number, 31
- invalid page, 302
- ioctl(), 151, 290
 - numer bloku fizycznego, 151
- ionice, 202
- ioprio_get(), 202
- ioprio_set(), 202
- iosched, 147
- IOV_MAX, 113
- IPC, 34, 36, 40
- ISO, 28
- ISO C++, 28
- ISO C11, 28
- ISO C95, 28
- ISO C99, 28
- ISO9660, 35
- ITIMER_PROF, 393
- ITIMER_REAL, 393
- ITIMER_VIRTUAL, 393
- itimerspec, 397
- itimerval, 393
- i-węzły, 31, 253
 - dowiązania symboliczne, 33
 - sortowanie operacji wejścia i wyjścia, 149
 - katalogi, 32

J

- jądro, 13, 17
- jądro Linuksa, 415
- jednostka pracy, 231
- język C, 28
- język C++, 25
- jiffies counter, 370
- jiffy, 370
- jitter, 206
- job, 157

K

- katalogi, 31, 253, 269
 - aktualny katalog roboczy, 270
 - czytanie ze strumienia, 278
 - dwie kropki, 270
 - elementy, 269
 - funkcje systemowe, 280
 - główny, 32, 270
 - kropka, 270
 - nadrzędne, 269

- odczyt zawartości, 278
- podkatalog, 269
- strumień, 278
- ścieżki, 270
- tworzenie, 275
- usuwanie, 276
- zamykanie strumienia, 279
- zmiana aktualnego katalogu roboczego, 272
- kernel-level threading, 229
- kill(), 345, 351
- killpg(), 353
- klasa
 - cykliczna, 209
 - FIFO, 208
 - szeregowania, 208
 - zwykła, 209
- klucze, 262
 - niezdefiniowany, 262
 - wartość niepusta, 262
 - zdefiniowany, 262
- kod przykładowy
 - używanie, 19
- kolejka FIFO
 - dla odczytów, 144
 - dla zapisów, 144
- kolejność zapisów, 59
- kompatybilność
 - binarna, 26
 - w przód, 29
 - źródłowa, 26
- kompilator
 - GNU C, 17
 - GNU C++, 25
 - języka C, 24
- komunikacja międzyprocesowa, 34, 40
 - blokowanie deskryptora, 71
- komunikacja poza kolejką, 290
- konfiguracja zarządcy operacji wejścia i wyjścia, 147
- konsolidowanie implementacji Pthreads, 241
- konwencja wywołań, 26
- konwencje typograficzne, 18
- konwersje czasu, 381
- kończenie wątków, 243
- kopiowanie
 - plików, 286
 - podczas zapisu, 164, 302
- koszty
 - przełączania, 196
 - wielowątkowości, 228

L

- latency, 206
- LBA, 143
- lchown(), 259
- level-triggered, 122
- lgetxattr(), 264
- libc, 24
- libpthread, 241
- libstdcxx, 25
- licznik chwil, 370
- licznik dowiązań, 33
- liczniki, 392
 - alarmy, 392
 - czas wygaśnięcia, 392
 - inicjalizowanie, 396
 - interwałowe, 392
 - itimerspec, 397
 - itimerval, 393
 - odczyt czasu wygaśnięcia, 397
 - odczyt wartości przepelnienia, 398
 - opóźnienie, 392
 - sigevent, 395
 - timespec, 397
 - timeval, 393
- liczniki
 - tworzenie, 395
 - usuwanie, 399
 - zaawansowane, 394
- lider grupy procesów, 184
- lider sesji, 185
- likely(), 407
- LINE_MAX, 95
- link, 31, 280
- link(), 281
- linker, 26
- Linus Elevator, 144
- Linux, 13, 21
 - pliki specjalne, 33
 - standardy, 28
- Linux Foundation, 29
- Linux Standard Base, 29
- LinuxThreads, 240
- lista, 160
 - atrybutów rozszerzonych, 267
- listxattr(), 267
- listy kontroli dostępu, 39
- llistxattr(), 267
- localtime(), 382
- localtime_r(), 382
- lock(), 237
- login, 38
- logowanie użytkownika, 38

- lokalizacja sekwencyjna, 83
- LONG_MAX, 56
- lremovexattr(), 268
- LSB, 29
- lseek(), 66
 - kody błędów, 67
 - ograniczenia, 68
- lsetxattr(), 265, 266
- lstat(), 254, 283
- luka, 67

Ł

- łączenie wątków, 246

M

- M_CHECK_ACTION, 320
- M_MMAP_MAX, 320
- M_MMAP_THRESHOLD, 320
- M_MXFAST, 320
- M_TOP_PAD, 320
- M_TRIM_THRESHOLD, 321
- MADV_DOFORK+, 135
- MADV_DONTFORK, 135
- MADV_DONTNEED, 135
- MADV_NORMAL, 134, 135
- MADV_RANDOM, 134, 135
- MADV_SEQUENTIAL, 135
- MADV_WILLNEED, 135, 136
- madvise(), 134
 - kody błędów, 136
 - wartości powrotne, 136
- major number, 289
- make, 22, 382
- maksymalny wiek bufora, 60
- mallinfo, 323
- malloc, 404
- malloc(), 304, 306
- MALLOC_CHECK_, 323
- malloc_stats(), 324
- malloc_trim(), 322
- malloc_usable_size(), 322
- mallopt(), 319
 - parametry, 321
- man pages, 242
- manipulowanie bajtami, 332
- MAP_ANONYMOUS, 318
- MAP_FAILED, 127, 131
- MAP_FIXED, 124
- MAP_PRIVATE, 125, 318
- MAP_SHARED, 125, 126
- mappings, 303

maska uprawnień tworzonych plików, 50
maximum buffer age, 60
MCL_CURRENT, 334
MCL_FUTURE, 334
mechanizmy
 blokowania, 106
 IPC, 40
 przydzielania pamięci, 327
memalign(), 312
memchr(), 331
memcmp(), 329
memcpy(), 331
memfrob(), 332
mемmem(), 332
memmove(), 330
memory areas, 303
memory leak, 310
memory regions, 303
mемpcpy(), 331
memrchr(), 331
memset(), 307, 329
metadane, 253
 i-węzeł, 31
 pozycja w pliku, 30
miękkie wiązanie, 203
migracja procesu, 202
MIME type sniffing, 262
mincore(), 336
minimalna ziarnistość, 197
minimum granularity, 197
minor number, 289
mkdir(), 275
mktime(), 381
mlock(), 221, 333
mlockall(), 221, 334
mmap(), 123, 317
 kody błędów, 127
 parametry, 317
 strony, 126
 wady używania, 130
 wartości powrotne, 127
 zalety używania, 130
MMU, 126, 164
mode, 49
mode_t, 257
modele wątkowości, 229
 wątkowość mieszana, 230
 wątkowość na poziomie użytkownika, 229
 współprogramy i włókna, 230
monitorowanie plików, 292
monotonic time, 369
montowanie, 35
mounting, 35

mounting point, 35
mprotect(), 132
 kody błędów, 132
 wartości powrotne, 132
mq_open(), 221
mremap(), 131
 kody błędów, 131
 wartości powrotne, 131
MREMAP_MAYMOVE, 131
MS_ASYNC, 133
MS_INVALIDATE, 133
MS_SYNC, 133
msync(), 133
 kody błędów, 134
 wartości powrotne, 134
multiplexed I/O, 71
multithreaded, 37, 225
munlock(), 335
munlockall(), 335
munmap(), 128, 317, 318
muteksy, 236
 blokowanie, 249
 inicjalizowanie, 248
 odblokowywanie, 249
 standardu Pthreads, 248
 użycie, 250
 zakleszczenia, 238
mutex, 237

N

N:1 threading, 229
N:M threading, 230
named pipes, 34
namespace, 34
nanosleep(), 387
Native POSIX Thread Library, 240
Native POSIX Threading Library, 37
nazwane potoki, 34
nazwy użytkowników, 38
Next Generation POSIX Threads, 240
NFS, 35
NGPT, 240
nice, 199
nice values, 199
nice(), 199
nieautomatyczne blokowanie plików, 106
nieblokowane operacje na strumieniu, 107
nieblokujące operacje wejścia i wyjścia, 55
niedeterminizm, 217
no-execute, 124
noinline, 410
nonblocking I/O, 55

- Noop I/O Scheduler, 147
- noreturn, 404
- normal class, 209
- notacja przedziałów, 131
- NPTL, 37, 198, 240
- null device, 289
- numer
 - bloku, 142
 - i-węzła, 31, 253
- NX, 124

O

- O_APPEND, 47, 57
- O_ASYNC, 47
- O_CLOEXEC, 47
- O_CREAT, 47
- O_DIRECT, 47, 64
- O_DIRECTORY, 47
- O_DSYNC, 48, 63
- O_EXCL, 47
- O_LARGEFILE, 48
- O_NOATIME+, 48
- O_NOCTTY, 48
- O_NOFOLLOW, 48
- O_NONBLOCK, 48, 55, 58
- O_RDONLY, 46
- O_RDWR, 46
- O_RSYNC, 48, 63
- O_SYNC, 48, 63, 141
- O_TRUNC, 48
- O_WRONLY, 46
- obcięcie bufora, 82
- obcinanie, 31
 - plików, 69
- obiekt
 - pthread_mutex_t, 248
 - z plikiem zależnym, 250
- obracanie, 383
- obsługa
 - błędów, 40
 - grup procesów, 187
 - procesu zombie, 169
 - sesji, 186
 - stanu uśpienia, 386, 387
 - sygnału, 340
- obsłużenie procesu, 37
- obszary pamięci, 303
- oczekiwanie
 - na określony proces, 171
 - na sygnał, 347

- na zakończone procesy potomka, 169
- na zbiór sygnałów, 358
- w pętli, 216
- odblokowywanie
 - muteksów, 249
 - pamięci, 335
- odczyt
 - aktualnego katalogu roboczego, 271
 - atrybutu rozszerzonego, 264
 - czasu wygaśnięcia licznika, 397
 - plików, 52
 - pozycyjny, 68
 - wartości przepełnienia licznika, 398
 - z wyprzedzeniem, 83, 136
 - zawartości katalogu, 278
- odczyty nieblokujące, 55
- odłączanie wątków, 246, 247
- odłączenie pliku, 33
- odmontowanie, 35
- odpytywanie zdarzeń, 117
- odwrócenie priorytetów, 239
- odwzorowania, 303
 - numerów sygnałów na łańcuchy znakowe, 350
 - pliku na przestrzeń adresową procesu, 125
- odwzorowywanie plików w pamięci, 123
 - dodatkowe sygnały, 128
 - madvise(), 134
 - mmap(), 123, 130
 - mprotect(), 132
 - mremap(), 131
 - msync(), 133
 - munmap(), 128
 - odczyt z wyprzedzeniem, 136
 - porady, 134
 - przykład, 128
 - rozmiar strony, 126
 - synchronizacja odwzorowanego pliku, 133
 - usuwanie odwzorowania, 128
 - zmiana rozmiaru odwzorowania, 131
 - zmiana uprawnień odwzorowania, 132
- odzyskiwanie oczekujących sygnałów, 358
- offsetof(), 409, 410
- ogólny model pliku, 81
- ograniczenia domyślne, 222
- ograniczenia zasobów systemowych, 218
 - miękkie, 219, 223
 - odczytywanie, 223
 - ograniczenia domyślne, 222
 - RLIMIT_AS, 220
 - RLIMIT_CORE, 220
 - RLIMIT_CPU, 220
 - RLIMIT_DATA, 220
 - RLIMIT_FSIZE, 220

- ograniczenia zasobów systemowych
 - RLIMIT_LOCKS, 220
 - RLIMIT_MEMLOCK, 221
 - RLIMIT_MSGQUEUE, 221
 - RLIMIT_NICE, 221
 - RLIMIT_NOFILE, 221
 - RLIMIT_NPROC, 221
 - RLIMIT_RSS, 221
 - RLIMIT_RTPRIO, 222
 - RLIMIT_RTTIME, 222
 - RLIMIT_SIGPENDING, 222
 - RLIMIT_STACK, 222
 - twarde, 219, 223
 - ustawianie, 223
 - oldstate, 245
 - on_exit(), 166, 168
 - OOM, 337
 - OOM killer, 337
 - op, 119
 - Open Software Foundation, 27
 - open(), 24, 46
 - wartości zwracane, 52
 - znaczniki, 46
 - open_sysconf(), 327
 - opendir(), 47, 278
 - operacje
 - asynchroniczne, 140
 - atomowe, 236
 - dla atrybutów rozszerzonych, 264
 - na pamięci, 328
 - niezsynchronizowane, 140
 - synchroniczne, 140
 - zbioru sygnałów, 356
 - zsynchronizowane, 140, 141
 - operacje wejścia i wyjścia
 - asynchroniczne, 111, 141, 228
 - bezpośrednie, 64
 - buforowane, 87
 - w przestrzeni użytkownika, 87
 - całkowity czas oczekiwania, 63
 - dla deskryptorów, 71
 - liniowe, 112
 - nieblokujące, 55, 71
 - odwzorowane w pamięci, 111
 - optymalizowanie wydajności, 148
 - porada dla operacji plikowych, 111
 - przenoszenie, 108
 - rozproszone, 109, 111, 112
 - sortowanie
 - wg numeru fizycznego bloku, 151
 - wg numeru i-węzła, 149
 - wg ścieżki, 149
 - standardowe, 106
 - szeregowanie w przestrzeni użytkownika, 148
 - typowe, 90
 - wektorowe, 112
 - wydajność, 142
 - wykonywanie, 66
 - zaawansowane, 111
 - zarządcy, 142
 - działanie, 143
 - niesortujący, 147
 - przewidujący, 145
 - wspomaganie odczytów, 143
 - wybór i konfiguracja, 147
 - z terminem nieprzekraczalnym, 144
 - ze sprawiedliwym szeregowaniem, 146
 - zsynchronizowane, 48, 60
 - z wielokrotnione, 70, 228
 - operational deadlines, 205
 - operator postinkrementacji, 234
 - opóźnienie, 206
 - odczytu, 143
 - opóźniony zapis, 59
 - stron, 81, 84
 - oprogramowanie systemowe, 15, 21
 - opróżnianie strumienia, 102
 - options, 172
 - optymalizacja
 - gałęzi kodu, 407
 - wartości licznika, 114
 - wydajności operacji wejścia i wyjścia, 148
 - organizacja wewnętrzna jądra, 81
 - origin, 66
 - OSF, 27
 - oszczędzanie pamięci, 227
 - otoczenie linuxowe, 30
 - otrzymywanie
 - identyfikatora grupy, 184
 - identyfikatora użytkownika, 184
 - otwieranie plików, 46, 91
 - tryb dopisywania, 47
 - tryb nieblokujący, 48
 - tryb zapisu, 48
 - zsynchronizowane operacje wejścia i wyjścia, 48
 - otwieranie strumienia poprzez deskryptor pliku, 92
 - overcommitment, 337
- ## P
- P_ALL, 174
 - P_GID, 174
 - P_PID, 174
 - packed, 405
 - page fault, 302

- PAGE_SIZE, 127
- paging, 216
- pakowanie struktury, 405
- pamięć, 301
 - dynamiczna, 304
 - flash, 34
 - wirtualna, 225
- parallelism, 233
- parametry graniczne, 206
- parent directory, 269
- partitionable, 35
- partycjonowane urządzenia, 35
- path injection, 161
- pathname, 270
- pathname resolution, 32
- pathnames, 32
- pause(), 347
- per-process namespaces, 35
- perror(), 42, 54
- pętla nieskończona, 194
- pętla zdarzeń, 232
- PGID, 184
- pid, 172
- PID, 37, 156, 157
- pid_max, 157
- pid_t, 157, 174
- pierwszoplanowa grupa procesów, 185
- pipe, 198
- pliki, 30, 253
 - atrybuty rozszerzone, 261
 - bezpośrednie operacje wejścia i wyjścia, 64
 - binarne, 24, 155
 - blokowanie, 106
 - czytanie, 52
 - wszystkich bajtów, 54
 - deskryptor, 30, 45
 - długość, 31
 - dowiązania, 31, 32, 280
 - fds, 45
 - informacje, 254
 - i-węzły, 31, 253
 - kompilowanie plików źródłowych, 19
 - kopiowanie, 286
 - licznik dowiązań, 33
 - metadane, 253
 - monitorowanie, 292
 - nagłówkowe, 40
 - obcinanie, 31, 69
 - odczyt
 - nieblokujący, 55
 - pozycyjny, 68
 - odłączenie, 33
 - odzworowywanie w pamięci, 123
 - operacje wejścia i wyjścia, 45
 - otwieranie, 46, 91
 - pozycja, 30
 - prawa własności, 259
 - przechowywanie typu MIME, 262
 - przenoszenie, 286
 - wyniki, 288
 - puste, 31
 - rozmiar, 31, 255
 - rzadkie, 67
 - skrót, 35
 - specjalne, 33
 - specjalne FIFO, 34
 - szukanie, 66
 - poza końcem pliku, 67
 - śledzenie zdarzeń, 292
 - tablica plików, 45
 - tryb
 - do odczytu, 47, 92
 - dopisywania, 47, 57, 91
 - dostępu, 46
 - nieblokujący, 48
 - otwarcia, 30
 - zapisu, 48
 - tworzenie, 51
 - uprawnienia, 39, 49, 257
 - urządzenia
 - blokowe, 34
 - znakowe, 34
 - urządzeń, 34
 - usuwanie, 33, 284
 - właściciel, 49
 - wskaźnik, 90
 - xattrs, 261
 - zamykanie, 65
 - zapis, 56
 - pozycyjny, 68
 - zapisy
 - częściowe, 57
 - nieblokujące, 58
 - znacznik końca, 34
 - zsynchronizowane operacje wejścia i wyjścia, 60
 - zwielokrotnione operacje wejścia i wyjścia, 70
 - zwykle, 30
- plikowe operacje wejścia i wyjścia, 45
- pmap, 303
- pobieranie
 - aktualnego czasu, 376
 - czasu procesu, 378
- podajniki szybkie, 320
- podkatalog, 269
- podział, 315

- poll(), 76, 79, 80
 - kody błędów, 78
 - wartości powrotu, 78
- POLLER, 77
- POLLHUP, 77
- POLLIN, 77
- POLLMSG, 77
- POLLNVAL, 77
- POLLOUT, 77
- POLLPRI, 77
- POLLRDBAND, 77
- POLLRDNORM, 77
- POLLWRBAND, 77
- POLLWRNORM, 77
- połączone wątki, 246
- pomnożenie, 158
- poprawa czasu reakcji, 226
- porady, 137, 139
 - odzworowywanie plików w pamięci, 134
 - standardowe plikowe operacje wejścia i wyjścia, 137
- porównywanie
 - bajtów, 329
 - identyfikatorów wątków, 243
- Portable Operating System Interface, 27
- POSIX, 27
 - identyfikator pthread_t, 243
- POSIX 1003.1c, 37
- POSIX 1988, 27
- POSIX 1990, 27
- POSIX 1993, 207
- POSIX 1995, 239
- POSIX 2008, 28
- POSIX threads, 239
- POSIX.1, 27
- POSIX.1b, 28, 207
- POSIX.1c, 28, 239
- POSIX_FADV_DONTNEED, 137, 138, 139
- POSIX_FADV_NOREUSE, 137, 138
- POSIX_FADV_NORMAL, 137, 138
- POSIX_FADV_RANDOM, 137, 138, 139
- POSIX_FADV_SEQUENTIAL, 139
- POSIX_FADV_SEQUENTIAL, 137, 138
- POSIX_FADV_WILLNEED, 137, 138, 139
- posix_fadvise(), 137
 - kody błędów, 138
 - wartości powrotne, 138
- posix_memalign(), 311
- potok, 34, 46, 198
- potomek, 37
- powielanie łańcuchów znakowych na stosie, 326
- powłoka, 21
 - systemowa, 38
- powrotny adres funkcji, 410
- poziomy uprzejmości, 199
- pozycja
 - elementu w strukturze, 409
 - w pliku, 30
- PPID, 157
- ppoll(), 80
- prawa własności, 259
- prawdziwa równoległość, 233
- prawdziwa współbieżność, 226
- pread(), 68, 69
 - kody błędów, 69
- precyzyjne ustawianie czasu, 379
- preemptive, 194
- primary group, 38
- PRIO_PGRP, 200
- PRIO_PROCESS, 200
- PRIO_USER, 200
- priorytet statyczny, 208
- priorytety procesu, 199
 - getpriority(), 200
 - nice(), 199
 - setpriority(), 200
- priorytety wejścia i wyjścia, 201
- process descriptor, 36
- process ID, 37, 156
- process time, 369
- process tree, 37
- processor-bound, 194
- procesy, 36, 155, 225
 - blokowane, 193
 - child, 37, 157
 - demony, 189
 - deskryptor, 36
 - drzewo, 37
 - duch, 179
 - format wykonywalny, 36
 - grupa, 157, 179
 - grupy procesów, 157, 184
 - hierarchia, 37, 157
 - identyfikator, 37, 156
 - procesu rodzicielskiego, 158
 - inicjalizujący, 37, 156
 - jałowy, 156
 - jednowątkowe, 37, 155, 225
 - klasy FIFO, 208
 - kończenie, 166, 167
 - kopiowanie podczas zapisu, 164
 - lekkie, 227
 - migracja, 202
 - obsługa, 37
 - oczekiwanie
 - na określony proces, 171
 - na zakończone procesy potomka, 169
 - ograniczenia zasobów systemowych, 218
 - otrzymywanie identyfikatora procesu, 158

- parent, 37, 157
- PID, 37, 156, 157
- pid_t, 157
- potmnozenie, 158
- potomek, 37, 157, 162
- potomny, 37, 157
- powiazania, 186
- PPID, 157
- priorytety, 199
- przydzial identyfikatorow, 156
- rodzic, 157, 162
- rodzicielski, 37, 157
- rozwidlenie, 158
- strategie szeregowania, 211
- sygnaly, 39
- szeregowanie, 193
 - z wywlaszczaniem, 195
- tworzenie, 37
- uruchamialne, 193
- uruchamianie, 158
 - i oczekiwanie na nowy proces, 177
- uzytkownik, 157, 179
- watki, 37
- wejście do powloki systemowej, 177
- wiazanie do konkretnego procesora, 202
- wielowatkowe, 37, 105, 155, 225
- z N watkami, 229
- zalecane modyfikacje identyfikatorow
 - uzytkownika i grupy, 183
- zarzadzca, 193
- zarzadzanie, 155, 230
 - zaawansowane, 193
- znaczniki kontekstu, 45
- zombie, 37, 169, 179
- związane z procesorem, 194
- związane z wejściem i wyjściem, 1957
- program, 155
- programowanie
 - aplikacji, 22
 - systemowe, 15, 21, 43
 - a programowanie aplikacji, 22
 - obsługa błędów, 40
 - podstawy, 23
 - w Linuksie, 18
 - w języku C, 413
 - w Linuksie, 29, 414
- programy wielowatkowe, 37
- projektowanie
 - systemu operacyjnego, 415
 - wysokopoziomowe, 22
- prot, 124
- PROT_EXEC, 124
- PROT_NONE, 124
- PROT_READ, 124, 318
- PROT_WRITE, 124, 318
- przedzial czasowy, 194
- przekierowania, 46
- przekroczenie zakresu zatwierdzenia, 337
- przelaczanie
 - poziomem, 122
 - wewnatrzprocesowe, 227
 - zbozem, 122, 123
- przelaczanie kontekstu, 227
 - procesy i watki, 227
 - watki na poziomie uzytkownika, 229
- przelacznik pliku wirtualnego, 81
- przenoszenie
 - bajtów, 330
 - plików, 286
- przepelnienia, 391
 - licznika, 391
- przerwania
 - generowane poziomem, 123
 - generowane zbozem, 123
- przerwanie programowe, 23
- przerzucanie stron, 83
- przestrzeń adresowa procesu, 301
- przestrzeń nazw, 34
 - dla atrybutów rozszerzonych, 263
 - dla procesu, 35
 - security, 263
 - system, 263
 - trusted, 263
 - user, 263
- przeszukiwanie pliku, 66, 67
- przydzial identyfikatorow procesow, 156
- przydzial oportunistyczny, 336
- przydzielanie, 301
- przydzielanie pamieci
 - anonimowe odwzorowania, 316
 - dla tablic, 306
 - dynamicznej, 304
 - sposoby, 328
 - wyrownanej, 311
- pselect(), 75
- psignal(), 351
- pthread, 241
- pthread_attr_t, 242
- pthread_cancel(), 244
- PTHREAD_CANCEL_ASYNCHRONOUS, 245
- PTHREAD_CANCEL_DEFERRED, 245
- PTHREAD_CANCEL_DISABLE, 245
- PTHREAD_CANCEL_ENABLE, 245
- pthread_create(), 241
- pthread_detach(), 247
- pthread_equal(), 243
- pthread_exit(), 244

- pthread_join(), 246
- pthread_mutex_lock(), 249, 251
- pthread_mutex_unlock(), 249, 251
- pthread_self(), 243
- pthread_setcancelstate(), 245
- pthread_setcanceltype(), 245
- pthreads, 37
- Pthreads, 240
- ptrace(), 170
- pula entropii, 289
- punkt
 - anulowania, 245
 - montowania, 35
 - podziału, 315
 - wywołania, 81
- pure, 403
- pure function, 403
- pwrite(), 68, 69
 - kody błędów, 69

R

- race condition, 233
- RAII, 250
- raise(), 353
- raport o błędach, 42
- read(), 23, 52
 - czytanie wszystkich bajtów, 54
 - odczyty nieblokujące, 55
 - ograniczenia rozmiaru, 56
 - stany wyjściowe, 53
 - wartości błędów, 55
 - wartości zwracane, 53
- readahead, 83
- readahead(), 137, 138
- readdir(), 278, 280
- readv(), 112
 - implementacja, 116
 - użycie, 115
 - wartości powrotne, 113
- real time, 369
- real UID, 38
- realloc(), 307
 - sprawna obsługa, 132
- real-time, 205
- region krytyczny, 106
- regiony pamięci, 303
- regular files, 30
- rejstry maszynowe, 24
- rejon krytyczny, 233
- rekordy, 30
- relative pathname, 32, 270
- remove(), 285
- removexattr(), 268
- rename(), 286
- resident set size, 221
- Resource Acquisition Is Initialization, 250
- resource limits, 218
- rewind(), 101
- RLIM_INFINITY, 219, 220, 223
- rlimit, 163
- RLIMIT_AS, 220
- RLIMIT_CORE, 219, 220
- RLIMIT_CPU, 219, 220
- RLIMIT_DATA, 220
- RLIMIT_FSIZE, 219, 220
- RLIMIT_LOCKS, 220
- RLIMIT_MEMLOCK, 221
- RLIMIT_MSGQUEUE, 221
- RLIMIT_NICE, 221
- RLIMIT_NOFILE, 78, 221
- RLIMIT_NPROC, 163, 221
- RLIMIT_OFILE, 221
- RLIMIT_RSS, 221
- RLIMIT_RTPRIO, 222
- RLIMIT_RTTIME, 222
- RLIMIT_SIGPENDING, 222
- RLIMIT_STACK, 222, 223
- rmdir(), 276, 285
- root, 38, 179
- root directory, 32, 270
- root filesystem, 35
- round-robin, 146
- round-robin class, 209
- rozdzielczość źródła czasu, 375
- rozliczanie ścisłe, 338
- rozmiar
 - bloku, 88, 89
 - bufora, 89
 - grupy rezydentnej, 221
 - pliku, 31, 255
 - słowa, 68
 - strony, 35, 126, 302
- rozpoznawanie typu MIME, 262
- rozproszone operacje wejścia i wyjścia, 109, 112
 - niepodzielność, 112
 - readv(), 112, 115
 - sprawność, 112
 - writev(), 112
 - wydajność, 112
 - wzorzec kodowania, 112
- rozsynchronizowanie, 206
- rozwidlenie, 158
- równoległość, 233
 - prawdziwa, 233
- RSS, 221
- ruid, 182, 183

runnable, 193
rusage, 176
rzeczywisty GID, 28
rzeczywisty identyfikator użytkownika, 38, 180
rzeczywisty UID, 38

S

S_IRGRP, 50
S_IROTH, 50
S_IRUSR, 50
S_IRWXG, 50
S_IRWXO, 50
S_IRWXU, 50
S_ISREG(), 129
S_ISVTX, 277
S_IWGRP, 50
S_IWOTH, 50
S_IWUSR, 50
S_IXGRP, 50
S_IXOTH, 50
S_IXUSR, 50
SA_NOCLDSTOP, 360
SA_NOCLDWAIT, 360
SA_NODEFER, 360
SA_NOMASK, 360
SA_ONESHOT, 360
SA_ONSTACK, 360
SA_RESETHAND, 361
SA_RESTART, 360
SA_SIGINFO, 360
saved UID, 38
sbrk(), 315
scalanie, 143
scatter-gather I/O, 109
SCHED_BATCH, 209
SCHED_FIFO, 208, 209, 210
sched_get_priority_max(), 213
sched_get_priority_min(), 213
sched_getaffinity(), 203
sched_getparam(), 211, 212
sched_getscheduler(), 210
SCHED_OTHER, 208, 209, 210
SCHED_RR, 208, 209, 210, 214
sched_rr_get_interval(), 214
sched_setaffinity(), 203
sched_setparam(), 211
sched_setscheduler(), 210
sched_yield(), 197, 198
scheduler, 193
Scheduler Activations, 230
scheduling class, 208
scheduling policy, 208
schemat przydziału wspieranej pamięci, 316

ScopedMutex, 250
security, 263
SEEK_CUR, 66, 100
SEEK_END, 66, 100
SEEK_SET, 66, 100, 101
segment przechowywania bloku, 36
segmenty
 bss, 303
 dane, 303
 stos, 303
 tekst, 303
sekcje, 36
 absolutne, 36
 bss, 36
 danych, 36
 niezdefiniowana, 36
 tekstu, 36
sektor, 35
select(), 71, 80, 390
 kody błędów, 73
 przenośny sposób wstrzymania wykonania
 aplikacji, 75
 użycie funkcji, 74
 wartości powrotu, 73
sesja, 185
 identyfikator, 186, 187
 obsługa, 186
 tworzenie, 186
set group ID, 49
set_tid_address(), 23
setegid(), 182
seteuid(), 182, 183, 184
setgid(), 181
setitimer(), 342, 344, 345, 392, 393
setpgid(), 187
setpgrp(), 189
setpriority(), 200, 201
setregid(), 182
setresgid(), 183
setresuid(), 183
setreuid(), 182
setrlimit(), 219, 224
setsid(), 186, 190
settimeofday(), 379, 380
setuid, 180
setuid(), 181, 184
setvbuf(), 104, 105
setxattr(), 265
SGID, 49, 161, 275
shell, 21
shmctl(), 221
shortcut, 33
si_code, 175, 363
si_pid, 174

- si_signo, 175
- si_status, 175
- si_uid, 174
- sieciowy system plików, 35
- SIG_BLOCK, 357
- SIG_DFL, 347
- SIG_IGN, 347
- SIG_SETMASK, 357
- SIG_UNBLOCK, 357
- SIGABRT, 341, 342
- sigaction, 359
- sigaction(), 169, 359
- sigaddset(), 356
- SIGALRM, 341, 342
- sigandset(), 356
- SIGBUS, 128, 341, 343
- SIGCHLD, 169, 177, 341, 343
- SIGCONT, 341, 343
- sigdelset(), 356
- sigemptyset(), 356
- SIGEV_NONE, 396
- SIGEV_SIGNAL, 396
- SIGEV_THREAD, 396
- sigevent, 395
- sigfillset(), 356
- SIGFPE, 341, 343
- sighandler_t, 347
- SIGHUP, 40, 341, 343
- SIGILL, 341, 343
- siginfo_t, 174, 361
- SIGINT, 177, 185, 340, 341, 344
- SIGIO, 47, 341, 344
- sigisemptyset(), 356
- sigismember(), 356
- SIGKILL, 40, 220, 340, 341, 344
 - kończenie procesu, 167
- sigmask, 76
- signal(), 169, 346, 347, 359
- signal-safe, 40
- signalstack(), 360
- sigorset(), 356
- sigpending(), 358
- SIGPIPE, 58, 341, 344
- sigprocmask(), 357, 358
- SIGPROF, 341, 344
- SIGPWR, 342, 344
- sigqueue(), 366
- SIGQUIT, 177, 342, 344
- SIGSEGV, 128, 220, 342, 344
- SIGSTKFLT, 342
- SIGSTOP, 40, 340, 342, 345
- sigsuspend(), 358
- SIGSYS, 342, 345
- SIGTERM, 340, 342, 345
 - kończenie procesu, 167
- SIGTRAP, 342, 345
- SIGTSTP, 342, 345
- SIGTTIN, 342, 345
- SIGTTOU, 342, 345
- SIGURG, 342, 345
- SIGUSR1, 342, 345
- SIGUSR2, 342, 345
- sigval, 366
- SIGVTALRM, 342, 345
- SIGWINCH, 342, 346
- SIGXCPU, 220, 342, 346
- SIGXFSZ, 342, 346
- single threaded, 225
- Single UNIX Specification, 27
- single-threaded, 37
- SIZE_MAX, 56
- size_t, 56
- sizeof(), 409
- skrót, 33
- skutki buforowania, 202
- sleep(), 385
- slurping, 296
- SMT, 227
- sockets, 34
- soft affinity, 203
- soft limit, 219
- soft links, 282
- soft real-time system, 206
- Solid State Drives, 147
- sortowanie, 143, 148
 - wg numeru fizycznego bloku, 151
 - wg numeru i-węzła, 149
 - wg ścieżki, 149
- sparse files, 67
- special files, 33
- specjalne węzły urządzeń, 289
- sposoby przydzielania pamięci, 328
- sprawiedliwe kolejkowanie, 196
- SSIZE_MAX, 56, 59
- ssize_t, 56
- st_atime, 255
- st_blksize, 254
- st_blocks, 255
- st_ctime, 255
- st_dev, 254
- st_gid, 254, 259
- st_ino, 254
- st_mode, 254, 257
- st_mtime, 255
- st_nlink, 254
- st_rdev, 254
- st_size, 254
- st_uid, 254, 259
- stacje CD-ROM, 34

- stack, 37
- stan anulowania, 245
 - state, 245
 - włączony, 245
 - wyłączony, 245
- stan braku pamięci, 337
- stan oczekiwania, 385
- stan uśpienia, 385
- standard error, 45
- standard I/O library, 90
- standard in, 45
- standard out, 45
- standard Pthreads, 239
 - API, 241
 - call(), 240
 - clone(), 240
 - exit_group(), 240
 - futex (), 240
 - identyfikatory wątków, 243
 - implementacje wątkowości w Linuksie, 240
 - interfejs programistyczny, 240
 - konsolidowanie implementacji Pthreada, 241
 - kończenie wątków, 243
 - przez inne wątki, 244
 - łączenie i odłączanie wątków, 246
 - muteksy, 248
 - przykład wątkowości, 247
 - strony z pomocą, 242
 - synchronizacja, 241
 - tworzenie wątków, 241
 - typ pthread_t, 243
 - współdzielenie zasobów przez wątki, 242
 - zarządzanie wątkami, 241
- standardowe operacje wejścia i wyjścia, 137
- standardowe wejście, 45
- standardowe wyjście, 45
- standardowe wyjście błędów, 45
- standardowy strumień błędów, 42
- standardy, 27, 29
 - języka C, 28
 - K&R, 28
- start_routine, 241, 243
- start_routine(), 242
- start_thread(), 247
- stat, 89, 254
- stat(), 151, 253, 255
 - numer i-węzła, 150
- static priority, 208
- stderr, 42, 45
- STDERR_FILENO, 46
- stdin, 45, 70
- STDIN_FILENO, 46
- stdio, 90
- stdout, 45
- STDOUT_FILENO, 46
- sterowanie operacjami wejścia i wyjścia, 290
- sterta, 303
- sticky, 277
- stime(), 379
- stos, 37, 324
- strategia FIFO, 208
- strategia jałowego szeregowania, 209
- strategia szeregowania, 208
- strdupa(), 326
- strerror(), 42
- strerror_r(), 42
- strict accounting, 338
- strndupa(), 326
- strona, 126, 301
 - nieprawidłowa, 302
 - prawidłowa, 302
 - wyrzucanie, 302
 - zerowa, 36
- stronicowanie, 216, 301
 - na żądanie, 332
- strsignal(), 351
- struct, 314
- struktura
 - account, 234
 - buffer_head, 84
 - epoll_event, 119, 121
 - inotify_event, 295, 300
 - itimerspec, 397
 - itimerval, 393
 - pirate, 99
 - pirate_ship, 304
 - pollfd, 77
 - rlimit, 219
 - rowboat, 409
 - rusage, 176
 - sched_param, 210
 - sigaction, 359
 - siginfo_t, 361
 - stat, 254
 - time_t, 372
 - timespec, 76, 372
 - timeval, 72, 372
 - timezone, 377
 - tm, 373
 - tms, 378
- strumienie, 91, 144
 - czytanie, 93
 - deskryptor pliku, 103
 - informacja o aktualnym położeniu, 101
 - nieblokowane operacje, 107
 - opróżnianie, 102
 - otwieranie poprzez deskryptor pliku, 92
 - szukanie, 100

- strumienie
 - wejściowe, 91
 - wejściowo-wyjściowe, 91
 - wyjściowe, 91
 - zamykanie, 93
 - zapis, 97
 - zapis pojedynczego znaku, 97
 - zwracanie znaków, 94
- strumień bajtów, 30
- strumień katalogu, 278
- subdirectory, 269
- suid, 180, 183
- SUID, 161
- supplemental groups, 38
- SUS, 27
- SUSv4, 28
- swapping, 216
- sygnały, 39, 339, 341, 342
 - akcja domyślna, 340
 - blokowanie, 357
 - dziedziczenie, 349
 - funkcje współużywalne, 355
 - identyfikatory, 340
 - ignorowane, 40
 - ignorowanie, 340
 - obsługa, 340
 - oczekiwanie
 - na sygnał, 347
 - na zbiór sygnałów, 358
 - oczekujące, 358
 - odzworowanie numerów na łańcuchy
 - znakowe, 350
 - odzyskiwanie oczekujących sygnałów, 358
 - si_code, 363
 - SIGCHLD, 169
 - SIGHUP, 40
 - siginfo_t, 361
 - SIGINT, 340
 - SIGKILL, 40, 340
 - SIGSTOP, 40, 340
 - SIGTERM, 340
 - uprawnienia, 352
 - uruchamianie, 349
 - wspierane przez system Linux, 341
 - współużywalność, 354
 - wygenerowanie, 340
 - wysyłanie, 340, 351
 - do grupy procesów, 353
 - do samego siebie, 353
 - z wykorzystaniem pola użytkowego, 366
 - zarządzanie, 346, 359
 - zbiory, 356
 - zerowy, 341
 - zgłoszenie, 340
- symbolic links, 33, 282
- symlink(), 283
- symlinks, 33, 282
- sympatyczne działanie, 198
- sync(), 62
- synchroniczne operacje, 140
- synchronizacja, 236
 - buforów na dysku, 62
 - dla operacji odczytu, 141
 - dla operacji zapisu, 140
 - dostępu do danych, 105
 - funkcje Pthreads, 241
 - muteksy, 236
 - odzworowanego pliku, 133
 - operacji wejścia i wyjścia, 60
 - zakleszczenia, 238
- sys_siglist[], 350
- syscalls, 23
- sysconf(), 126, 168
- sysctl, 337
- system, 263
- system calls, 23
- system czasu rzeczywistego, 205
 - blokowanie pamięci, 216
 - determinizm, 216
 - klasa szeregowania, 208
 - określanie zakresu poprawnych priorytetów, 213
 - opóźnienie, 206
 - parametry graniczne, 206
 - priorytet statyczny, 208
 - projektowanie programów, 215
 - rozsynchronizowanie, 206
 - SCHED_BATCH, 209
 - SCHED_OTHER, 209
 - strategia
 - cykliczna, 209
 - FIFO, 208
 - szeregowania, 208
 - szeregowania wsadowego, 209
 - zwykła, 209
 - ścisłego, 205
 - środki ostrożności przy pracy, 215
 - ustalenie
 - priorytetów, 208
 - strategii szeregowania, 210
 - ustawianie parametrów szeregowania, 211
 - wcześniejsze zapisywanie danych, 216
 - wiązanie do procesora, 217
 - wspieranie przez system Linux, 207
 - zwykłego, 206
- system operacyjny
 - definiowanie ABI, 26
 - kooperacyjny, 194

- wielozadaniowy, 193
 - z wywłaszczaniem, 194
- system opóźnionego zapisu, 60
- system plików, 30, 34
 - atributy rozszerzone, 262
 - ext4, 35
 - FAT, 35
 - główny, 35
 - ISO9660, 35
 - montowanie, 35
 - NFS, 35
 - odmontowanie, 35
 - organizacja wewnętrzna jądra, 81
 - przechowywanie typu MIME, 262
 - punkt montowania, 35
 - sektor, 35
 - sieciowy, 35
 - VFS, 81
 - wirtualny, 35, 81
 - XFS, 35
- system programming, 21
- system przydzielania pamięci
 - uruchamianie programów, 323
- system software, 21
- system timer, 370
- system uprawnień, 38
- system(), 177
 - problemy bezpieczeństwa, 178
- systemy wbudowane, 207
- SysVinit, 217
- sytuacja wyścigu, 233
- szeregowanie operacji wejścia i wyjścia
 - w przestrzeni użytkownika, 148
- szeregowanie procesów, 193
 - procesy związane z procesorem, 194
 - procesy związane z wejściem i wyjściem, 194
 - przedział czasowy, 194
 - sched_yield(), 198
 - szeregowanie z wywłaszczaniem, 195
 - udostępnianie czasu procesora, 194, 196, 197
 - wielozadaniowe systemy operacyjne, 194
- szukanie w strumieniu, 100

Ś

- ścieżki, 32, 142, 270
 - bezwzględne, 32, 270
 - pełne, 32
 - względne, 32, 270
- śledzenie zdarzeń związanych z plikami, 292

T

- tablica plików, 45
- tablice, 306
 - o długości zerowej, 296
 - o zmiennej długości, 326
- takt, 370
- target latency, 196
- temporal locality, 194
- terminal, 48
- terminal sterujący, 185
- The Open Group, 27
- thread pool
 - abstrakcja programistyczna, 226
- thread-per-connection
 - abstrakcja programistyczna, 226
- threads of execution, 37
- tick, 370
- TID, 243
- time(), 376
- TIME_BAD, 385
- TIME_DEL, 385
- TIME_INS, 385
- TIME_OK, 385
- TIME_OOP, 385
- time_t, 372, 377
- TIME_WAIT, 385
- timeout, 72
- TIMER_ABSTIME, 389
- timer_create(), 394, 395, 397
- timer_delete(), 394, 399
- timer_getoverrun(), 398
- timer_gettime(), 397
- timer_settime(), 394, 396
- times(), 378
- timeslice, 194
- timespec, 372, 397
- timeval, 372, 393
- timex, 384
- TLB, 227
- TLS, 240
- tłumaczenie
 - katalogu, 32
 - ścieżki, 32
- tm, 373
- tmpfile(), 166
- tms, 378
- toolchain, 26
- translation lookaside buffer, 227
- trap, 23
- truncation, 31
- trusted, 263

- tryb
 - automatycznego zamykania, 117
 - dopisywania, 58
 - otwierania plików, 91
- twarde wiązanie, 203
- tworzenie
 - anonimowe odwzorowanie w pamięci, 317
 - dowiązania, 281
 - dowiązania symboliczne, 283
 - katalogów, 275
 - kontekst interfejsu odpypywania zdarzeń, 117
 - liczniki, 395
 - pliki, 51
 - procesy, 37
 - sesja, 186
 - wątków, 241
- typ
 - long, 68
 - unsigned long, 52
- typ anulowania, 244
 - asynchroniczny, 245
 - opóźniony, 245
- typedef, 157
- typeof(), 408
- typowe operacje wejścia i wyjścia, 90

U

- udostępnianie czasu procesora, 194, 196, 197
 - futex, 198
 - sched_yield(), 197
- UID, 38, 49
- ulimit, 223
- umask, 50
- umieszczanie zmiennych globalnych
 - w rejestrach, 407
- undefined section, 36
- ungetc(), 94
- unikanie przeciążenia, 84
- unikanie zakleszczeń, 239
- union, 314
- Unix, 15, 30
- unlikely(), 407
- unlink(), 284
- unlinking, 33
- unlock(), 237
- unmounting, 35
- unsigned long, 314
- unused, 405
- uprawnienia, 38, 257
 - ACL, 39
 - bity uprawnień, 39
 - CAP_CHOWN, 259, 261
 - CAP_FOWNER, 258, 277
 - CAP_KILL, 352
 - CAP_SYS_ADMIN, 263
 - listy kontroli dostępu, 39
 - nowe pliki, 49
 - standardowe, 39
 - sygnały, 352
 - współdzielonych danych, 302
- uruchamianie procesu, 158
- urządzenia, 34
 - generator liczb losowych, 289
 - partycjonowane, 35
 - puste, 289
 - zapełnione, 289
 - zerowe, 289
- urządzenia blokowe, 34
 - pliki, 34
 - sektor, 35
- urządzenia terminalowe, 48
- urządzenia znakowe, 34
 - pliki, 34
- urządzenie tty, 185
- useconds_t, 386
- user, 263
- user ID, 38
- user-level threading, 229
- usernames, 38
- users, 38
- usleep(), 386
- ustalanie strategii szeregowania, 210
- ustawianie
 - aktualnego czasu, 379
 - atrybutu rozszerzonego, 265
 - parametrów szeregowania, 211
 - wartości bajtów, 329
- usuwanie
 - atrybuty rozszerzone, 268
 - elementu z systemu plików, 284
 - katalogów, 276
 - licznik, 399
 - pliki, 33
- UTC, 370
- util-linux, 216
- uzupełnienie, 321
- uzyskiwanie typu dla wyrażenia, 408
- użytkownicy, 38, 179
 - /etc/passwd, 38
 - administrator, 38
 - efektywny identyfikator, 38
 - EUID, 38
 - grupy, 38
 - hasła, 38
 - identyfikator, 38
 - logowanie, 38

- nazwy, 38
- procesu, 157
- root, 38, 179
- rzeczywisty identyfikator, 38
- UID, 38
- zapisany identyfikator, 38

użytkownik API, 26

używanie zasobów po ich zwolnieniu, 310

V

Valgrind, 310

valid page, 302

valloc(), 312

variable-length arrays, 326

variadic, 159

vfork(), 165

VFS, 81, 82

virtual file switch, 81

VLA, 326

vm.overcommit_memory, 337

vm.overcommit_ratio, 338

VMS, 30

void, 411

volatile, 314

W

wait(), 169, 170, 171, 343

wait3(), 175

wait4(), 175

waitid(), 173

waiting on, 37

waitpid(), 170, 171, 175, 378

wall time, 369

warn_unused_result, 404

wątki, 37, 105, 155, 225

- bezpieczeństwo, 105
- blokowanie
 - plików, 106
- z poziomu użytkownika, 198

brak zsynchronizowania, 228

domyślny, 241

flusher, 84

główny, 241

implementacja, 37

kończenie, 243

- innych wątków, 244
- samego siebie, 244
- wszystkich wątków, 244

łączenie, 246

odłączanie, 246

pdflush, 84

pthread, 37

stan anulowania, 244

stos, 37

typ anulowania, 244

- zmiana, 245

wykonawcze, 37

wątkowość, 225

- 1:1, 229
- implementacja wątkowości w Linuksie, 240
- mieszana, 230
- modele, 229
- N:1, 229
- N:M, 230
- na poziomie jądra, 229
- na poziomie użytkownika, 229
- przydzielania wątku do połączenia, 231
- standard Pthreads, 247
- sterowana zdarzeniami, 232
- thread-per-connection, 231
- współbieżność, 233
- wzorce, 231

WCONTINUED, 172, 174

wejscie do powłoki systemowej, 177

wektor, 113, 160

WEXITED, 174

WEXITSTATUS, 177

węzeł informacji, 31

węzły urządzeń, 288

- generator liczb losowych, 289
- numer drugorzędny, 289
- numer główny, 289
- specjalne węzły, 289
- urządzenie puste, 289

wheel, 157

which, 200

wiązanie procesów do konkretnego procesora, 202

wielkość wyrównania dla danego typu, 408

wieloprocessorowość, 216

wielowątkowość, 226

- alternatywy, 228
- koszty, 228

wielozadaniowe systemy operacyjne, 194

wielozadaniowość kooperatywna, 194

WIFCONTINUED, 170

WIFEXITED, 170

WIFSIGNALED, 170

WIFSTOPPED, 170

wirtualizacja, 36

wirtualna przestrzeń adresowa, 301

wirtualny procesor, 225

wirtualny system plików, 35, 81

- ogólny model pliku, 81
- przełącznik pliku wirtualnego, 81
- punkty wywołania, 81

- właściciel pliku, 49
- właściwości lokalizacji, 82
- włókna, 230
- WNOHANG, 172, 174
- WNOWAIT, 174
- Wpadded, 313
- wprowadzanie w stan uśpienia, 390
- wrapper, 305
- wrappers, 24
- write(), 23, 56
 - kody błędów, 58
 - ograniczenia rozmiaru, 59
 - sposób działania, 59
 - tryb dopisywania, 57
 - zapis nieblokujący, 58
 - zapisy częściowe, 57
- writeback, 59, 84
- writev(), 109, 112
 - implementacja, 116
 - użycie, 114
 - wartości powrotne, 113
- wskaźniki
 - do funkcji, 411
 - do pliku, 90
 - void, 411
- wspomaganie odczytów, 143
- współbieżność, 226, 228, 233
 - działanie, 235
- współdzielenie danych, 302
- współprogramy, 230
- współużywalność, 354
- WSTOPPED, 174
- wstrzykiwanie ścieżki, 161
- WUNTRACED, 172
- wycieki pamięci, 310
- wycofywanie znaku, 94
- wydajność operacji wejścia i wyjścia, 142, 148
 - sortowanie wg numeru fizycznego bloku, 151
 - sortowanie wg numeru i-węzła, 149
 - sortowanie wg ścieżki, 149
 - szeregowanie w przestrzeni użytkownika, 148
- wyłuskiwanie składników czasu, 373
- wymiana danych, 216
- wymuszanie sprawdzania wartości powrotnej dla procedur wywołujących, 404
- wypełnienia, 313
- wyrównanie
 - dla struktury, 313
 - dla tablicy, 313
 - dla unii, 313
 - zasada ścisłego wyrównania, 314
- wyrównanie danych, 97, 310
 - reguły, 313
 - sposób naturalny, 97

- wyrównywanie obciążenia, 203
- wyrzucanie stron, 302
- wysyłanie sygnału, 351
 - do grupy procesów, 353
 - do samego siebie, 353
 - z wykorzystaniem pola użytkowego, 366
- wyszukiwanie, 142
 - bajtów, 331
- wyścig do danych, 233
- wyścigi, 233
- wyłaszczanie, 195
- wywoływanie funkcji systemowych, 23
- wzajemne wykluczanie, 236
- wzorce wątkowości, 231
 - thread-per-connection, 231
 - wątkowość sterowana zdarzeniami, 232
- wzorzec
 - przydzielania wątku do połączenia, 232
 - RAII, 250
 - sterowania zdarzeniami, 232

X

- X/Open, 27
- XATTR_CREATE, 266
- XATTR_REPLACE, 266
- xattrs, 261
- XFS, 35
- xmalloc(), 307
- XOR, 332

Y

- yield, 230
- yielding, 194

Z

- zabójca stanu braku pamięci, 337
- zadania, 157
- zakleszczenie, 238
 - ABBA, 239
 - śmiertelny uścisk, 239
 - unikanie, 239
- zakończenie procesu, 166, 167
- zalecane modyfikacje identyfikatorów użytkownika i grupy, 183
- zamknięcie w wątku, 105
- zamykanie
 - pliku, 65
 - strumieni, 93
 - strumienia katalogu, 279

- zapis do pliku, 56
 - zapisy częściowe, 57
- zapis do strumienia, 97
 - dane binarne, 98
 - łańcuch znaków, 98
 - pojedyncze znaki, 97
- zapis nieblokujący, 58
- zapis pozycyjny, 68
- zapisany GID, 38
- zapisany identyfikator użytkownika, 38, 180, 184
- zapobieganie wplataniu funkcji, 402
- zarządca operacji wejścia i wyjścia, 81, 142
 - Anticipatory I/O Scheduler, 146
 - CFQ, 146
 - Complete Fair Queuing I/O Scheduler, 146
 - Deadline I/O Scheduler, 144, 145
 - działanie, 143
 - konfiguracja, 147
 - Linus Elevator, 144
 - niesortujący, 147
 - Noop I/O Scheduler, 147
 - przewidujący, 145
 - scalanie, 143
 - sortowanie, 143
 - sprawiedliwe szeregowanie, 146
 - termin nieprzekraczalny, 144
 - wspomaganie odczytów, 143
 - wybór, 147
- zarządca procesów, 193
- zarządzanie
 - katalogami, 253
 - obciążeniem, 139
 - plikami, 253
 - procesami, 155, 193, 230
 - segmentem danych, 315
 - stanem uśpienia, 389
 - sygnałami, 346, 359
 - wątkami, 241
 - zadaniami, 184
- zarządzanie pamięcią, 301
 - /dev/zero, 318
 - anonimowe odwzorowanie w pamięci, 315, 316
 - blokowanie
 - całej przestrzeni adresowej, 334
 - fragmentu przestrzeni adresowej, 333
 - pamięci, 332
 - błąd strony, 302
 - dane statystyczne, 323
 - fragmentacja
 - wewnętrzna, 316
 - zewnętrzna, 316
 - kopiowanie podczas zapisu, 302
 - MALLOC_CHECK_, 323
 - niskopoziomowa kontrola działania systemu
 - przydzielania pamięci, 322
 - odblokowywanie pamięci, 335
 - ograniczenia blokowania, 335
 - operacje na pamięci, 328
 - pliki odwzorowane, 303
 - powielanie łańcuchów znakowych na stosie, 326
 - przekroczenie zakresu zatwierdzenia, 337
 - przestrzeń adresowa procesu, 301
 - przydział oportunistyczny, 336
 - przydzielanie pamięci
 - dla tablic, 306
 - dynamicznej, 304
 - wyrównanej, 311
 - regiony pamięci, 303
 - rozliczanie ścisłe, 338
 - schemat przydziału wspieranej pamięci, 316
 - stan braku pamięci, 337
 - stos, 324
 - stronicowanie, 301
 - na żądanie, 332
 - strony, 301
 - tablice o zmiennej długości, 326
 - współdzielenie danych, 302
 - wybór mechanizmu przydzielania pamięci, 327
 - wycieki pamięci, 310
 - wyrównanie danych, 310
 - zaawansowane operacje przydziału pamięci, 319
 - zarządzanie segmentem danych, 315
 - zmiana wielkości obszaru przydzielonej pamięci, 307
 - zwalnianie pamięci dynamicznej, 309
- zasada przydzielania najmniej uprzywilejowanych uprawnień, 179
- zasada ścisłego wyrównania, 314
- zasoby procesu, 36
- zasysanie, 296
- zawieszane dowiązania symboliczne, 283
- zbiory sygnałów, 356
- zdarzenia
 - inotify, 295
 - przełączane poziomem, 122
 - przełączane zboczem, 122
- zegar POSIX, 374
- zegar sprzętowy, 371
- zegar systemowy, 370, 382
- zero device, 289
- zero page, 36
- zestaw narzędzi, 26

- zmiana
 - aktualnego katalogu roboczego, 272
 - identyfikatora dla użytkownika lub grupy
 - efektywnego, 182
 - rzeczywistego, 181
 - w wersji BDS, 182
 - w wersji HP-UX, 183
 - zapisanego, 181
 - wielkości obszaru przydzielonej pamięci, 307
- zmienne naturalnie wyrównane, 311
- znacznik końca pliku, 34
- znacznik zamykania, 47, 292
- znaczniki dostępu, 124
- zombie, 37, 169, 179
- zsynchronizowane operacje, 140
- zsynchronizowane operacje wejścia i wyjścia, 60
 - fdatasync(), 61
 - fsync(), 60
 - kody błędów, 61
 - O_DSYNC, 63
 - O_RSYNC, 63
 - O_SYNC, 63
 - sync(), 62
- zwalnianie, 249
 - pamięci dynamicznej, 309
- związek jeden do jednego, 229
- zwiększone operacje wejścia i wyjścia, 70, 228
 - implementacja, 71
 - poll(), 76, 80
 - ppoll(), 80
 - pselect(), 75
 - select(), 71, 80
 - wątkowość sterowana zdarzeniami, 232
- zwiększanie wartości wyrównania dla zmiennej, 406

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>

GRUPA WYDAWNICZA

 **Helion SA**

LINUX. PROGRAMOWANIE SYSTEMOWE



Jądro systemu Linux to jeden z największych projektów rozwijanych przez ogromną społeczność. Setki wolontariuszy dniami i nocami rozwijają najważniejszy element tego systemu operacyjnego. I robią to naprawdę skutecznie! Każde kolejne wydanie Linuksa zawiera dziesiątki nowinek oraz ulepszeń – jest coraz szybsze, bezpieczniejsze i po prostu lepsze. Jednak początkujący programiści mogą mieć problemy z wykorzystaniem usług dostarczanych przez kernel. Masz obawy, że nie odnajdziesz się w gąszczu możliwości współczesnego jądra systemu Linux?

Ta książka rozwieje je w mig. Jest to wyjątkowa pozycja na rynku wydawniczym. W trakcie lektury nauczysz się tworzyć niskopoziomowe oprogramowanie, które będzie się komunikowało bezpośrednio z jądrem systemu. Operacje wejścia i wyjścia, strumienie, zdarzenia, procesy to tylko część elementów, które błyskawicznie opanujesz. Ponadto nauczysz się zarządzać katalogami i plikami oraz poznasz koncepcję sygnałów. Książka ta jest niezastąpionym źródłem informacji dla wszystkich programistów pracujących z jądrem Linuksa. Docenisz tę lekturę!

Poznaj:

- metody zarządzania procesami
- zastosowanie sygnałów
- zaawansowane interfejsy wejścia i wyjścia
- jądro systemu od podszewki

Jądro systemu Linux od podszewki!

Robert Love – od wielu lat jest użytkownikiem i współtwórcą systemu Linux. Rozwija środowisko graficzne GNOME oraz jądro systemu. Pracuje jako projektant oprogramowania w firmie Google, był też członkiem zespołu projektującego system operacyjny Android. Jest autorem licznych książek poświęconych programowaniu w systemie Linux.

helion.pl
księgarnia
internetowa

Nr katalogowy: 17000



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



ISBN 978-83-246-8285-0



Cena 79,00 zł