Andrew S. Tanenbaum, Herbert Bos

Systemy operacyjne

Wydanie V

Helion

Tytuł oryginału: Modern Operating Systems, 5th Edition

Tłumaczenie: Radosław Meryk

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-289-0289-3

Authorized translation from the English language edition, entitled Modern Operating Systems, 5th Edition by Andrew Tanenbaum; Herbert Bos, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2023, 2014, 2008 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. Polish language edition published by Helion S.A., Copyright © 2024.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

PEARSON is an exclusive trademark owned by Pearson Education, Inc. or its affiliates in the U.S. and/or other countries.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek 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 wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/sysop5
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

SPIS TREŚCI

Przedmowa **21**

O autorach **25**

1. WPROWADZENIE **27**

1.1. CZYM JEST SYSTEM OPERACYJNY? 30
1.1.1. System operacyjny jako rozszerzona maszyna 30
1.1.2. System operacyjny jako menedżer zasobów 31

1.2. HISTORIA SYSTEMÓW OPERACYJNYCH 33
1.2.1. Pierwsza generacja (1945 – 1955) — lampy elektronowe 33
1.2.2. Druga generacja (1955 – 1965) — tranzystory i systemy wsadowe 34
1.2.3. Trzecia generacja (1965 – 1980) — układy scalone i wieloprogramowość 36
1.2.4. Czwarta generacja (1980 — czasy współczesne) 40
1.2.5. Piąta generacja (1990 — czasy współczesne) 44

1.3. SPRZĘT KOMPUTEROWY — PRZEGLĄD 45
1.3.1. Procesory 45
1.3.2. Pamięć 49
1.3.3. Pamięć nieulotna 52
1.3.4. Urządzenia wejścia-wyjścia 53
1.3.5. Magistrale 56
1.3.6. Uruchamianie komputera 58

Kup ksi k Pole ksi k

1.4. PRZEGLĄD SYSTEMÓW OPERACYJNYCH 60
1.4.1. Systemy operacyjne komputerów mainframe 60
1.4.2. Systemy operacyjne serwerów 60
1.4.3. Systemy operacyjne komputerów osobistych 61
1.4.4. Systemy operacyjne smartfonów i komputerów podręcznych 61
1.4.5. Internet rzeczy i wbudowane systemy operacyjne 61
1.4.6. Systemy operacyjne czasu rzeczywistego 62
1.4.7. Systemy operacyjne kart elektronicznych 62

1.5. POJĘCIA DOTYCZĄCE SYSTEMÓW OPERACYJNYCH 63
1.5.1. Procesy 63
1.5.2. Przestrzenie adresowe 65
1.5.3. Pliki 66
1.5.4. Wejście-wyjście 69
1.5.5. Zabezpieczenia 69
1.5.6. Powłoka 69
1.5.7. Ontogeneza jest rekapitulacją filogenezy 70

1.6. WYWOŁANIA SYSTEMOWE 73
1.6.1. Wywołania systemowe do zarządzania procesami 76
1.6.2. Wywołania systemowe do zarządzania plikami 80
1.6.3. Wywołania systemowe do zarządzania katalogami 80
1.6.4. Różne wywołania systemowe 82
1.6.5. Interfejs Windows API 83

1.7. STRUKTURA SYSTEMÓW OPERACYJNYCH 85
1.7.1. Systemy monolityczne 85
1.7.2. Systemy warstwowe 86
1.7.3. Mikrojądra 88
1.7.4. Model klient-serwer 90
1.7.5. Maszyny wirtualne 91
1.7.6. Egzojądra i unijądra 95

1.8. ŚWIAT WEDŁUG JĘZYKA C 96
1.8.1. Język C 96
1.8.2. Pliki nagłówkowe 97
1.8.3. Duże projekty programistyczne 97
1.8.4. Model fazy działania 98

1.9. BADANIA DOTYCZĄCE SYSTEMÓW OPERACYJNYCH 99

1.10. PLAN POZOSTAŁEJ CZĘŚCI KSIĄŻKI 101

1.11. JEDNOSTKI MIAR 101

1.12. PODSUMOWANIE 102

Kup ksi k Pole ksi k

2. PROCESY I WĄTKI 107

2.1. PROCESY 107
2.1.1. Model procesów 108
2.1.2. Tworzenie procesów 110
2.1.3. Kończenie działania procesów 112
2.1.4. Hierarchie procesów 113
2.1.5. Stany procesów 113
2.1.6. Implementacja procesów 115
2.1.7. Modelowanie wieloprogramowości 117

2.2. WĄTKI 118
2.2.1. Wykorzystanie wątków 118
2.2.2. Klasyczny model wątków 123
2.2.3. Wątki POSIX 126
2.2.4. Implementacja wątków w przestrzeni użytkownika 128
2.2.5. Implementacja wątków w jądrze 131
2.2.6. Implementacje hybrydowe 132
2.2.7. Przystosowywanie kodu jednowątkowego do obsługi wielu wątków 133

2.3. SERWERY STEROWANE ZDARZENIAMI 136

2.4. SYNCHRONIZACJA I KOMUNIKACJA MIĘDZYPROCESOWA 138
2.4.1. Wyścig 139
2.4.2. Regiony krytyczne 140
2.4.3. Wzajemne wykluczanie z wykorzystaniem aktywnego oczekiwania 141
2.4.4. Wywołania sleep i wakeup 146
2.4.5. Semafory 149
2.4.6. Muteksy 152
2.4.7. Monitory 157
2.4.8. Przekazywanie komunikatów 163
2.4.9. Bariery 165
2.4.10. Inwersja priorytetów 167
2.4.11. Unikanie blokad: odczyt-kopiowanie-aktualizacja 168

2.5. SZEREGOWANIE 170
2.5.1. Wprowadzenie do szeregowania 170
2.5.2. Szeregowanie w systemach wsadowych 176
2.5.3. Szeregowanie w systemach interaktywnych 178
2.5.4. Szeregowanie w systemach czasu rzeczywistego 184
2.5.5. Oddzielenie strategii od mechanizmu 185
2.5.6. Szeregowanie wątków 186

2.6. PRACE BADAWCZE NAD PROCESAMI I WĄTKAMI 187

2.7. PODSUMOWANIE 188

Kup ksi k Pole ksi k

3. ZARZĄDZANIE PAMIĘCIĄ 197

3.1. BRAK ABSTRAKCJI PAMIĘCI 198
3.1.1. Uruchamianie wielu programów w systemach bez abstrakcji pamięci 199

3.2. ABSTRAKCJA PAMIĘCI: PRZESTRZENIE ADRESOWE 201
3.2.1. Pojęcie przestrzeni adresowej 201
3.2.2. Wymiana pamięci 204
3.2.3. Zarządzanie wolną pamięcią 206

3.3. PAMIĘĆ WIRTUALNA 209
3.3.1. Stronicowanie 210
3.3.2. Tabele stron 214
3.3.3. Przyspieszenie stronicowania 216
3.3.4. Tabele stron dla pamięci o dużej objętości 220

3.4. ALGORYTMY ZASTĘPOWANIA STRON 223
3.4.1. Optymalny algorytm zastępowania stron 224
3.4.2. Algorytm NRU 225
3.4.3. Algorytm FIFO 226
3.4.4. Algorytm drugiej szansy 226
3.4.5. Algorytm zegarowy 227
3.4.6. Algorytm LRU 228
3.4.7. Programowa symulacja algorytmu LRU 228
3.4.8. Algorytm bazujący na zbiorze roboczym 230
3.4.9. Algorytm WSClock 233
3.4.10. Podsumowanie algorytmów zastępowania stron 235

3.5. PROBLEMY PROJEKTOWE SYSTEMÓW STRONICOWANIA 236
3.5.1. Lokalne i globalne strategie alokacji pamięci 237
3.5.2. Zarządzanie obciążeniem 239
3.5.3. Strategia czyszczenia 240
3.5.4. Rozmiar strony 241
3.5.5. Osobne przestrzenie instrukcji i danych 242
3.5.6. Strony współdzielone 243
3.5.7. Biblioteki współdzielone 244
3.5.8. Pliki odwzorowane w pamięci 246

3.6. PROBLEMY IMPLEMENTACJI 247
3.6.1. Zadania systemu operacyjnego w zakresie stronicowania 247
3.6.2. Obsługa błędów braku strony 248
3.6.3. Archiwizowanie instrukcji 249

3.6.4. Blokowanie stron w pamięci 250
3.6.5. Magazyn stron 250
3.6.6. Oddzielenie strategii od mechanizmu 252

3.7. SEGMENTACJA 254
3.7.1. Implementacja klasycznej segmentacji 257
3.7.2. Segmentacja ze stronicowaniem: MULTICS 257
3.7.3. Segmentacja ze stronicowaniem: Intel x86 261

3.8. BADANIA DOTYCZĄCE ZARZĄDZANIA PAMIĘCIĄ 261

3.9. PODSUMOWANIE 262

4. SYSTEMY PLIKÓW 273

4.1. PLIKI 275
4.1.1. Nazwy plików 275
4.1.2. Struktura plików 277
4.1.3. Typy plików 278
4.1.4. Dostęp do plików 280
4.1.5. Atrybuty plików 281
4.1.6. Operacje na plikach 282
4.1.7. Przykładowy program wykorzystujący wywołania obsługi systemu plików 283

4.2. KATALOGI 285
4.2.1. Jednopoziomowe systemy katalogów 285
4.2.2. Hierarchiczne systemy katalogów 286
4.2.3. Nazwy ścieżek 287
4.2.4. Operacje na katalogach 289

4.3. IMPLEMENTACJA SYSTEMU PLIKÓW 290
4.3.1. Układ systemu plików 291
4.3.2. Implementacja plików 292
4.3.3. Implementacja katalogów 297
4.3.4. Pliki współdzielone 299
4.3.5. Systemy plików o strukturze dziennika 302
4.3.6. Księgujące systemy plików 304
4.3.7. Systemy plików na nośnikach typu flash 305
4.3.8. Wirtualne systemy plików 309

Kup ksi k Pole ksi k

4.4. ZARZĄDZANIE SYSTEMEM PLIKÓW I OPTYMALIZACJA 312
4.4.1. Zarządzanie miejscem na dysku 312
4.4.2. Kopie zapasowe systemu plików 318
4.4.3. Spójność systemu plików 323
4.4.4. Wydajność systemu plików 325
4.4.5. Defragmentacja dysków 330
4.4.6. Kompresja i deduplikacja 331
4.4.7. Bezpieczne usuwanie plików i szyfrowanie dysków 332

4.5. PRZYKŁADOWY SYSTEM PLIKÓW 333
4.5.1. System plików MS-DOS 334
4.5.2. System plików V7 systemu UNIX 337

4.6. BADANIA DOTYCZĄCE SYSTEMÓW PLIKÓW 339

4.7. PODSUMOWANIE 340

5. WEJŚCIE-WYJŚCIE 347

5.1. WARUNKI, JAKIE POWINIEN SPEŁNIAĆ SPRZĘT WEJŚCIA-WYJŚCIA 347
5.1.1. Urządzenia wejścia-wyjścia 348
5.1.2. Kontrolery urządzeń 349
5.1.3. Urządzenia wejścia-wyjścia odwzorowane w pamięci 350
5.1.4. Bezpośredni dostęp do pamięci (DMA) 353
5.1.5. O przerwaniach raz jeszcze 356

5.2. WARUNKI, JAKIE POWINNO SPEŁNIAĆ OPROGRAMOWANIE WEJŚCIA-WYJŚCIA 361
5.2.1. Cele oprogramowania wejścia-wyjścia 361
5.2.2. Programowane wejście-wyjście 363
5.2.3. Wejście-wyjście sterowane przerwaniami 364
5.2.4. Wejście-wyjście z wykorzystaniem DMA 365

5.3. WARSTWY OPROGRAMOWANIA WEJŚCIA-WYJŚCIA 366
5.3.1. Procedury obsługi przerwań 366
5.3.2. Sterowniki urządzeń 367
5.3.3. Oprogramowanie wejścia-wyjścia niezależne od urządzeń 371
5.3.4. Oprogramowanie wejścia-wyjścia w przestrzeni użytkownika 376

Kup ksi k Pole ksi k

5.4. PAMIĘĆ MASOWA: DYSKI MAGNETYCZNE I SSD 378
5.4.1. Dyski magnetyczne 378
5.4.2. Dyski SSD 388
5.4.3. RAID 392

5.5. ZEGARY 396
5.5.1. Sprzęt obsługi zegara 396
5.5.2. Oprogramowanie obsługi zegara 398
5.5.3. Zegary programowe 400

5.6. INTERFEJSY UŻYTKOWNIKÓW: KLAWIATURA, MYSZ, MONITOR 402
5.6.1. Oprogramowanie do wprowadzania danych 402
5.6.2. Oprogramowanie do generowania wyjścia 408

5.7. CIENKIE KLIENTY 423

5.8. ZARZĄDZANIE ENERGIĄ 425
5.8.1. Problemy sprzętowe 426
5.8.2. Problemy po stronie systemu operacyjnego 427
5.8.3. Problemy do rozwiązania w programach aplikacyjnych 432

5.9. BADANIA DOTYCZĄCE WEJŚCIA-WYJŚCIA 433

5.10. PODSUMOWANIE 434

6. ZAKLESZCZENIA 443

6.1. ZASOBY 444
6.1.1. Zasoby z możliwością wywłaszczania i bez niej 444
6.1.2. Zdobywanie zasobu 445
6.1.3. Problem pięciu filozofów 446

6.2. WPROWADZENIE W TEMATYKĘ ZAKLESZCZEŃ 449
6.2.1. Warunki powstawania zakleszczeń zasobów 450
6.2.2. Modelowanie zakleszczeń 450

6.3. ALGORYTM STRUSIA 453

6.4. WYKRYWANIE ZAKLESZCZEŃ I ICH USUWANIE 453
6.4.1. Wykrywanie zakleszczeń z jednym zasobem każdego typu 454
6.4.2. Wykrywanie zakleszczeń dla przypadku wielu zasobów każdego typu 456
6.4.3. Usuwanie zakleszczeń 458

Kup ksi k Pole ksi k

6.5. UNIKANIE ZAKLESZCZEŃ 459
6.5.1. Trajektorie zasobów 460
6.5.2. Stany bezpieczne i niebezpieczne 461
6.5.3. Algorytm bankiera dla pojedynczego zasobu 462
6.5.4. Algorytm bankiera dla wielu zasobów 463

6.6. PRZECIWDZIAŁANIE ZAKLESZCZENIOM 465
6.6.1. Atak na warunek wzajemnego wykluczania 465
6.6.2. Atak na warunek wstrzymania i oczekiwania 465
6.6.3. Atak na warunek braku wywłaszczania 466
6.6.4. Atak na warunek cyklicznego oczekiwania 466

6.7. INNE PROBLEMY 467
6.7.1. Blokowanie dwufazowe 467
6.7.2. Zakleszczenia komunikacyjne 468
6.7.3. Uwięzienia 470
6.7.4. Zagłodzenia 471

6.8. BADANIA NA TEMAT ZAKLESZCZEŃ 472

6.9. PODSUMOWANIE 473

7. WIRTUALIZACJA I PRZETWARZANIE W CHMURZE 481

7.1. HISTORIA 484

7.2. WYMAGANIA DOTYCZĄCE WIRTUALIZACJI 485

7.3. HIPERNADZORCY TYPU 1 I TYPU 2 488

7.4. TECHNIKI SKUTECZNEJ WIRTUALIZACJI 489
7.4.1. Wirtualizacja systemów bez obsługi wirtualizacji 490
7.4.2. Koszt wirtualizacji 492

7.5. CZY HIPERNADZORCY SĄ PRAWIDŁOWYMI MIKROJĄDRAMI? 493

7.6. WIRTUALIZACJA PAMIĘCI 496

7.7. WIRTUALIZACJA WEJŚCIA-WYJŚCIA 500

7.8. MASZYNY WIRTUALNE NA PROCESORACH WIELORDZENIOWYCH 503

Kup ksi k Pole ksi k

7.9. CHMURY OBLICZENIOWE 504
7.9.1. Chmury jako usługa 504
7.9.2. Migracje maszyn wirtualnych 505
7.9.3. Punkty kontrolne 506

7.10. WIRTUALIZACJA NA POZIOMIE SYSTEMU OPERACYJNEGO 506

7.11. STUDIUM PRZYPADKU: VMWARE 509
7.11.1. Wczesna historia firmy VMware 509
7.11.2. VMware Workstation 510
7.11.3. Wyzwania podczas opracowywania warstwy wirtualizacji na platformie x86 511
7.11.4. VMware Workstation: przegląd informacji o rozwiązaniu 513
7.11.5. Ewolucja systemu VMware Workstation 521
7.11.6. ESX Server: hipernadzorca typu 1 firmy VMware 522

7.12. BADANIA NAD WIRTUALIZACJĄ I CHMURĄ 523

7.13. PODSUMOWANIE 525

8. SYSTEMY WIELOPROCESOROWE 529

8.1. SYSTEMY WIELOPROCESOROWE 532
8.1.1. Sprzęt wieloprocesorowy 532
8.1.2. Typy wieloprocesorowych systemów operacyjnych 543
8.1.3. Synchronizacja w systemach wieloprocesorowych 547
8.1.4. Szeregowanie w systemach wieloprocesorowych 551

8.2. WIELOKOMPUTERY 558
8.2.1. Sprzęt wielokomputerów 559
8.2.2. Niskopoziomowe oprogramowanie komunikacyjne 563
8.2.3. Oprogramowanie komunikacyjne poziomu użytkownika 565
8.2.4. Zdalne wywołania procedur 568
8.2.5. Rozproszona współdzielona pamięć 571
8.2.6. Szeregowanie systemów wielokomputerowych 575
8.2.7. Równoważenie obciążenia 575

8.3. SYSTEMY ROZPROSZONE 578
8.3.1. Sprzęt sieciowy 580
8.3.2. Usługi i protokoły sieciowe 583
8.3.3. Warstwa middleware bazująca na dokumentach 587

Kup ksi k Pole ksi k

8.3.4. Warstwa middleware bazująca na systemie plików 588
8.3.5. Warstwa middleware bazująca na obiektach 592
8.3.6. Warstwa middleware bazująca na koordynacji 593

8.4. BADANIA DOTYCZĄCE SYSTEMÓW WIELOPROCESOROWYCH 596

8.5. PODSUMOWANIE 597

9. BEZPIECZEŃSTWO 603

9.1. PODSTAWY BEZPIECZEŃSTWA SYSTEMÓW OPERACYJNYCH 605
9.1.1. Triada bezpieczeństwa CIA 606
9.1.2. Zasady bezpieczeństwa 607
9.1.3. Bezpieczeństwo struktury systemu operacyjnego 608
9.1.4. Zaufana baza obliczeniowa 610
9.1.5. Intruzi 611
9.1.6. Czy możemy budować bezpieczne systemy? 614

9.2. KONTROLOWANIE DOSTĘPU DO ZASOBÓW 615
9.2.1. Domeny ochrony 615
9.2.2. Listy kontroli dostępu 618
9.2.3. Uprawnienia 621

9.3. MODELE FORMALNE BEZPIECZNYCH SYSTEMÓW 624
9.3.1. Bezpieczeństwo wielopoziomowe 626
9.3.2. Kryptografia 628
9.3.3. Moduły TPM 632

9.4. UWIERZYTELNIANIE 633
9.4.1. Hasła 634
9.4.2. Uwierzytelnianie z wykorzystaniem obiektu fizycznego 640
9.4.3. Uwierzytelnianie z wykorzystaniem technik biometrycznych 642

9.5. WYKORZYSTYWANIE BŁĘDÓW W KODZIE 643
9.5.1. Ataki z wykorzystaniem przepełnienia bufora 644
9.5.2. Ataki z wykorzystaniem łańcuchów formatujących 653
9.5.3. Ataki typu „użyj po zwolnieniu” 657
9.5.4. Luki typu Type Confusion 657
9.5.5. Ataki bazujące na odwołaniach do pustego wskaźnika 659
9.5.6. Ataki z wykorzystaniem przepełnień liczb całkowitych 660
9.5.7. Ataki polegające na wstrzykiwaniu kodu 660
9.5.8. Ataki TOCTOU 661
9.5.9. Luka oparta na podwójnym pobieraniu 662

Kup ksi k Pole ksi k

9.6. EKSPLOITY SPRZĘTOWE 663
9.6.1. Kanały ukryte 663
9.6.2. Kanały boczne 666
9.6.3. Ataki z wykorzystaniem przejściowego wykonywania 668

9.7. ATAKI OD WEWNĄTRZ 673
9.7.1. Bomby logiczne 673
9.7.2. Tylne drzwi 674
9.7.3. Podszywanie się pod ekran logowania 674

9.8. WZMACNIANIE SYSTEMU OPERACYJNEGO 675
9.8.1. Dokładna randomizacja 676
9.8.2. Ograniczenia przepływu sterowania 677
9.8.3. Ograniczenia dostępu 679
9.8.4. Kontrole integralności kodu i danych 682
9.8.5. Zdalna atestacja przy użyciu modułu TPM 683
9.8.6. Hermetyzacja niezaufanego kodu 684

9.9. BADANIA DOTYCZĄCE BEZPIECZEŃSTWA 687

9.10. PODSUMOWANIE 688

10. PIERWSZE STUDIUM PRZYPADKU: UNIX, LINUX I ANDROID 697

10.1. HISTORIA SYSTEMÓW UNIX I LINUX 698
10.1.1. UNICS 698
10.1.2. PDP-11 UNIX 699
10.1.3. Przenośny UNIX 700
10.1.4. Berkeley UNIX 701
10.1.5. Standard UNIX 702
10.1.6. MINIX 703
10.1.7. Linux 704

10.2. PRZEGLĄD SYSTEMU LINUX 707
10.2.1. Cele Linuksa 707
10.2.2. Interfejsy systemu Linux 708
10.2.3. Powłoka 710
10.2.4. Programy użytkowe systemu Linux 713
10.2.5. Struktura jądra 714

Kup ksi k Pole ksi k

10.3. PROCESY W SYSTEMIE LINUX 717
10.3.1. Podstawowe pojęcia 717
10.3.2. Wywołania systemowe Linuksa związane z zarządzaniem procesami 720
10.3.3. Implementacja procesów i wątków w systemie Linux 724
10.3.4. Szeregowanie w systemie Linux 730
10.3.5. Synchronizacja w Linuksie 734
10.3.6. Uruchamianie systemu Linux 735

10.4. ZARZĄDZANIE PAMIĘCIĄ W SYSTEMIE LINUX 738
10.4.1. Podstawowe pojęcia 738
10.4.2. Wywołania systemowe Linuksa odpowiedzialne za zarządzanie pamięcią 741
10.4.3. Implementacja zarządzania pamięcią w systemie Linux 742
10.4.4. Stronicowanie w systemie Linux 748

10.5. OPERACJE WEJŚCIA-WYJŚCIA W SYSTEMIE LINUX 751
10.5.1. Podstawowe pojęcia 752
10.5.2. Obsługa sieci 753
10.5.3. Wywołania systemowe wejścia-wyjścia w systemie Linux 755
10.5.4. Implementacja wejścia-wyjścia w systemie Linux 756
10.5.5. Moduły w systemie Linux 759

10.6. SYSTEM PLIKÓW LINUKSA 760
10.6.1. Podstawowe pojęcia 760
10.6.2. Wywołania systemu plików w Linuksie 765
10.6.3. Implementacja systemu plików Linuksa 768
10.6.4. NFS — sieciowy system plików 776

10.7. BEZPIECZEŃSTWO W SYSTEMIE LINUX 783
10.7.1. Podstawowe pojęcia 783
10.7.2. Wywołania systemowe Linuksa związane z bezpieczeństwem 785
10.7.3. Implementacja bezpieczeństwa w systemie Linux 786

10.8. ANDROID 786
10.8.1. Android a Google 787
10.8.2. Historia Androida 788
10.8.3. Cele projektowe 792
10.8.4. Architektura Androida 794
10.8.5. Rozszerzenia Linuksa 796
10.8.6. ART 799
10.8.7. Binder IPC 801
10.8.8. Aplikacje Androida 809
10.8.9. Zamiary 819

Kup ksi k Pole ksi k

10.8.10. Model procesów 821
10.8.11. Bezpieczeństwo i prywatność 825
10.8.12. Uruchamianie w tle a nauki społeczne 843

10.9. PODSUMOWANIE 850

11. DRUGIE STUDIUM PRZYPADKU: WINDOWS 11 859

11.1. HISTORIA SYSTEMU WINDOWS DO WYDANIA WINDOWS 11 859
11.1.1. Lata osiemdziesiąte: MS-DOS 860
11.1.2. Lata dziewięćdziesiąte: Windows na bazie MS-DOS-a 861
11.1.3. Lata dwutysięczne: Windows na bazie NT 861
11.1.4. Windows Vista 864
11.1.5. Windows 8 865
11.1.6. Windows 10 866
11.1.7. Windows 11 867

11.2. PROGRAMOWANIE SYSTEMU WINDOWS 868
11.2.1. Platforma programowania UWP 869
11.2.2. Podsystemy Windowsa 870
11.2.3. Rdzenny interfejs programowania aplikacji (API) systemu NT 871
11.2.4. Interfejs programowania aplikacji Win32 875
11.2.5. Rejestr systemu Windows 879

11.3. STRUKTURA SYSTEMU 881
11.3.1. Struktura systemu operacyjnego 881
11.3.2. Uruchamianie systemu Windows 898
11.3.3. Implementacja menedżera obiektów 902
11.3.4. Podsystemy, biblioteki DLL i usługi trybu użytkownika 913

11.4. PROCESY I WĄTKI SYSTEMU WINDOWS 916
11.4.1. Podstawowe pojęcia 916
11.4.2. Wywołania API związane z zarządzaniem zadaniami, procesami, wątkami i włóknami 921
11.4.3. Implementacja procesów i wątków 927
11.4.4. WoW64 i emulacja 937

11.5. ZARZĄDZANIE PAMIĘCIĄ 941
11.5.1. Podstawowe pojęcia 941
11.5.2. Wywołania systemowe związane z zarządzaniem pamięcią 946
11.5.3. Implementacja zarządzania pamięcią 948
11.5.4. Kompresja pamięci 958
11.5.5. Partycje pamięci 961

Kup ksi k Pole ksi k

11.6. PAMIĘĆ PODRĘCZNA SYSTEMU WINDOWS 963

11.7. OPERACJE WEJŚCIA-WYJŚCIA W SYSTEMIE WINDOWS 964
11.7.1. Podstawowe pojęcia 965
11.7.2. Wywołania API związane z operacjami wejścia-wyjścia 967
11.7.3. Implementacja systemu wejścia-wyjścia 969

11.8. SYSTEM PLIKÓW NT SYSTEMU WINDOWS 974
11.8.1. Podstawowe pojęcia 975
11.8.2. Implementacja systemu plików NTFS 976

11.9. ZARZĄDZANIE ENERGIĄ W SYSTEMIE WINDOWS 986

11.10. WIRTUALIZACJA W SYSTEMIE WINDOWS 988
11.10.1. Hyper-V 989
11.10.2. Kontenery 996
11.10.3. Bezpieczeństwo oparte na wirtualizacji 1002

11.11. BEZPIECZEŃSTWO W SYSTEMIE WINDOWS 1004
11.11.1. Podstawowe pojęcia 1005
11.11.2. Wywołania API związane z bezpieczeństwem 1007
11.11.3. Implementacja bezpieczeństwa 1008
11.11.4. Czynniki ograniczające zagrożenia bezpieczeństwa 1011

11.12. PODSUMOWANIE 1020

12. PROJEKT SYSTEMU OPERACYJNEGO 1027

12.1. ISTOTA PROBLEMÓW ZWIĄZANYCH Z PROJEKTOWANIEM SYSTEMÓW 1028
12.1.1. Cele 1028
12.1.2. Dlaczego projektowanie systemów operacyjnych jest takie trudne? 1029

12.2. PROJEKT INTERFEJSU 1031
12.2.1. Zalecenia projektowe 1032
12.2.2. Paradygmaty 1034
12.2.3. Interfejs wywołań systemowych 1037

12.3. IMPLEMENTACJA 1040
12.3.1. Struktura systemu 1040
12.3.2. Mechanizm kontra strategia 1043
12.3.3. Ortogonalność 1044

Kup ksi k Pole ksi k

12.3.4. Nazewnictwo 1045
12.3.5. Czas wiązania nazw 1047
12.3.6. Struktury statyczne kontra struktury dynamiczne 1048
12.3.7. Implementacja góra-dół kontra implementacja dół-góra 1049
12.3.8. Komunikacja synchroniczna kontra asynchroniczna 1050
12.3.9. Przydatne techniki 1051

12.4. WYDAJNOŚĆ 1056
12.4.1. Dlaczego systemy operacyjne są powolne? 1057
12.4.2. Co należy optymalizować? 1057
12.4.3. Dylemat przestrzeń-czas 1058
12.4.4. Buforowanie 1061
12.4.5. Wskazówki 1062
12.4.6. Wykorzystywanie efektu lokalności 1063
12.4.7. Optymalizacja z myślą o typowych przypadkach 1063

12.5. ZARZĄDZANIE PROJEKTEM 1064
12.5.1. Mityczny osobomiesiąc 1064
12.5.2. Struktura zespołu 1066
12.5.3. Znaczenie doświadczenia 1067
12.5.4. Nie istnieje jedno cudowne rozwiązanie 1069

13. LISTA PUBLIKACJI I BIBLIOGRAFIA 1073

13.1. SUGEROWANE PUBLIKACJE DODATKOWE 1073
13.1.1. Publikacje wprowadzające 1074
13.1.2. Procesy i wątki 1074
13.1.3. Zarządzanie pamięcią 1075
13.1.4. Systemy plików 1075
13.1.5. Wejście-wyjście 1076
13.1.6. Zakleszczenia 1077
13.1.7. Wirtualizacja i przetwarzanie w chmurze 1077
13.1.8. Systemy wieloprocesorowe 1078
13.1.9. Bezpieczeństwo 1079
13.1.10. Pierwsze studium przypadku: UNIX, Linux i Android 1080
13.1.11. Drugie studium przypadku: Windows 1080
13.1.12. Projekt systemu operacyjnego 1081

13.2. BIBLIOGRAFIA W PORZĄDKU ALFABETYCZNYM 1082

Skorowidz 1104

Kup ksi k Pole ksi k

Kup książkę Poleć książkę

2.

PROCESY I WĄTKI

Teraz przystąpimy do szczegółowego badania, jak są zaprojektowane i zbudowane systemy operacyjne. Kluczowym pojęciem we wszystkich systemach operacyjnych jest *proces*: abstrakcja działającego programu. Wszystkie pozostałe elementy systemu operacyjnego bazują na pojęciu procesu, dlatego jest bardzo ważne, aby projektant systemu operacyjnego (a także student) jak najszybciej dobrze zapoznał się z pojęciem procesu.

Procesy to jedne z najstarszych i najważniejszych abstrakcji występujących w systemach operacyjnych. Zapewniają one możliwość wykonywania (pseudo-) współbieżnych operacji nawet wtedy, gdy dostępny jest tylko jeden procesor. Przekształcają one pojedynczy procesor CPU w wiele wirtualnych procesorów. Gdy dostępne są cztery, osiem lub więcej procesorów (rdzeni), mogą działać dziesiątki lub setki procesów. Bez abstrakcji procesów istnienie współczesnej techniki komputerowej byłoby niemożliwe. W tym rozdziale przedstawimy szczegółowe informacje na temat tego, czym są procesy oraz ich pierwsi kuzynowie — wątki.

2.1. PROCESY

Wszystkie nowoczesne komputery bardzo często wykonują wiele operacji jednocześnie. Osoby przyzwyczajone do pracy z komputerami osobistymi mogą nie być do końca świadome tego faktu, zatem kilka przykładów pozwoli przybliżyć to zagadnienie. Na początek rozważmy serwer WWW. Żądania stron WWW mogą nadchodzić z wielu miejsc. Kiedy przychodzi żądanie, serwer sprawdza, czy potrzebna strona znajduje się w pamięci podręcznej. Jeśli tak, jest przesyłana do klienta. Jeśli nie, inicjowane jest żądanie dyskowe w celu jej pobrania. Jednak z perspektywy procesora obsługa żądań dyskowych zajmuje wieczność. W czasie oczekiwania na zakończenie obsługi żądania na dysk może nadejść wiele kolejnych żądań. Jeśli w systemie jest wiele dysków niektóre z żądań może być skierowanych na inne dyski na długo przed obsłużeniem pierwszego żądania. Oczywiste, że potrzebny jest sposób zamodelowania i zarządzania tą współbieżnością. Do tego celu można wykorzystać procesy (a w szczególności wątki).

Kup ksi k Pole ksi k

Teraz rozważmy komputer osobisty użytkownika. Podczas rozruchu systemu następuje start wielu procesów. Często użytkownik nie jest tego świadomy; np. może być uruchomiony proces oczekujący na wchodzące wiadomości e-mail. Inny uruchomiony proces może działać w imieniu programu antywirusowego i sprawdzać okresowo, czy są dostępne jakieś nowe definicje wirusów. Dodatkowo mogą działać jawne procesy użytkownika — np. drukujące pliki lub tworzące kopie zapasowe zdjęć na dysku USB — podczas gdy użytkownik przegląda strony WWW. Działaniami tymi trzeba zarządzać. W tym przypadku bardzo przydaje się system z obsługą wieloprogramowości, obsługujący wiele procesów jednocześnie. Wiele procesów mogą obsługiwać nawet proste urządzenia komputerowe, takie jak smartfony i tablety.

W każdym systemie wieloprogramowym procesor szybko przełącza się pomiędzy procesami, poświęcając każdemu z nich po kolei dziesiątki albo setki milisekund. Chociaż, ściśle rzecz biorąc, w dowolnym momencie procesor realizuje tylko jeden proces, w ciągu sekundy może obsłużyć ich wiele, co daje iluzję współbieżności. Czasami w tym kontekście mówi się o **pseudowspółbieżności**, dla odróżnienia jej od rzeczywistej, sprzętowej współbieżności systemów **wieloprocesorowych** (wyposażonych w dwa procesory współdzielące tę samą fizyczną pamięć lub większą liczbę takich procesorów). Śledzenie wielu równoległych działań jest bardzo trudne. Z tego powodu projektanci systemów operacyjnych w ciągu wielu lat opracowali model pojęciowy (procesów sekwencyjnych), które ułatwiają obsługę współbieżności. Ten model, jego zastosowania oraz kilka innych konsekwencji stanowią temat rozdziału.

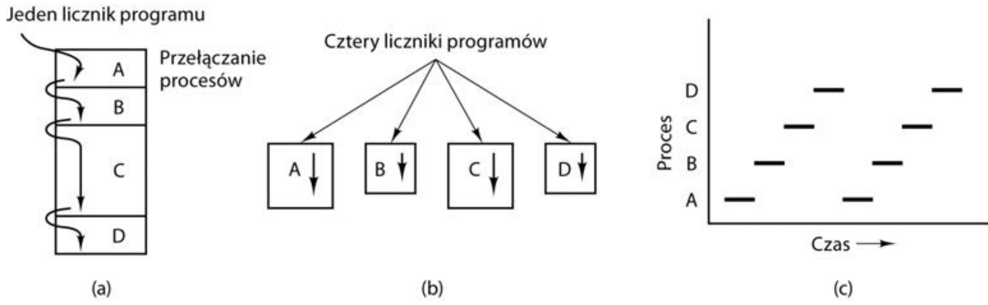
2.1.1. Model procesów

W tym modelu całe oprogramowanie możliwe do uruchomienia w komputerze — czasami włącznie z systemem operacyjnym — jest zorganizowane w postaci zbioru **procesów sekwencyjnych** (lub w skrócie procesów). Proces jest egzemplarzem uruchomionego programu włącznie z bieżącymi wartościami licznika programu, rejestrów i zmiennych. Pojęciowo każdy proces ma własny wirtualny procesor CPU. Oczywiście w rzeczywistości procesor fizyczny przełącza się od procesu do procesu. Aby jednak zrozumieć system, znacznie łatwiej jest myśleć o kolekcji procesów działających (pseudo) współbieżnie, niż próbować śledzić to, jak procesor przełącza się od programu do programu. To szybkie przełączanie się procesora jest określane jako **wieloprogramowość**, o czy mówiliśmy w rozdziale 1.

Na rysunku 2.1(a) pokazaliśmy komputer, w którym w pamięci działają w trybie wieloprogramowym cztery programy. Na rysunku 2.1(b) widać cztery procesy — każdy ma własny przepływ sterowania (tzn. własny logiczny licznik programu) i każdy działa niezależnie od pozostałych. Oczywiście jest tylko jeden fizyczny licznik programu, dlatego kiedy działa wybrany proces, jego logiczny licznik programu jest kopiowany do rzeczywistego licznika programu. Kiedy proces kończy działanie (na pewien czas), jego fizyczny licznik programu jest zapisywany w logicznym liczniku programu umieszczonym w pamięci. Na rysunku 2.1(c) widać, że w dłuższym przedziale czasu nastąpił postęp we wszystkich procesach, jednak w danym momencie działa tylko jeden proces.

W tym rozdziale założymy, że jest tylko jeden procesor CPU. Coraz częściej jednak takie założenie okazuje się nieprawdziwe. Nowe układy często są wielordzeniowe — mają dwa procesory, cztery lub większą ich liczbę. O układach wielordzeniowych i systemach wieloprocesorowych powiemy więcej w rozdziale 8. Na razie będzie prościej, jeśli przyjmiemy, że maszyna wykorzystuje jednorazowo tylko jeden procesor. Jeśli zatem mówimy, że procesor w danym momencie może wykonywać tylko jeden proces, to jeśli zawiera dwa rdzenie (lub dwa procesory), na każdym z nich w określonym momencie może działać jeden proces.

Kup ksi k Pole ksi k



Rysunek 2.1. (a) Cztery programy uruchomione w trybie wieloprogramowym; (b) pojęciowy model czterech niezależnych od siebie procesów sekwencyjnych; (c) w wybranym momencie jest aktywny tylko jeden program

Ze względu na szybkie przełączanie się procesora pomiędzy procesami tempo, w jakim proces wykonuje obliczenia, nie jest jednolite, a nawet trudne do powtórzenia w przypadku ponownego uruchomienia tego samego procesu. A zatem nie można programować procesów z wbudowanymi założeniami dotyczącymi czasu działania. Rozważmy dla przykładu proces obsługi strumienia audio, który odtwarza muzykę będącą akompaniamentem wysokiej jakości wideo uruchomionego przez inne urządzenie. Ponieważ dźwięk powinien rozpocząć się nieco później niż wideo, proces daje sygnał serwerowi wideo do rozpoczęcia odtwarzania, a następnie, zanim rozpocznie odtwarzanie dźwięku, uruchamia 10 tysięcy iteracji pustej pętli. Wszystko pójdzie dobrze, jeśli pustą pętlę można uznać za niezawodny czasomierz. Jeśli jednak procesor zdecyduje się na przełączenie do innego procesu podczas trwania pętli, proces obsługi strumienia audio nie będzie mógł ponownie się uruchomić do momentu, kiedy nie będą gotowe właściwe ramki wideo. W efekcie powstanie bardzo denerwujący efekt braku synchronizacji pomiędzy audio i wideo. Kiedy proces obowiązują tak ścisłe wymagania działania w czasie rzeczywistym — tzn. określone zdarzenia muszą wystąpić w ciągu określonej liczby milisekund — trzeba przedsięwziąć specjalne środki w celu zapewnienia, że tak się stanie. Zazwyczaj jednak większości procesów nie dotyczą ograniczenia wieloprogramowości procesora czy też względne szybkości działania różnych procesów.

Różnica pomiędzy procesem a programem jest subtelna, ale ma kluczowe znaczenie. Do wyjaśnienia tej różnicy posłużymy się analogią. Załóżmy, że pewien informatyk o zdolnościach kulinarnych piecze urodzinowy tort dla swojej córki. Ma do dyspozycji przepis na tort urodzinowy oraz kuchnię dobrze wyposażoną we wszystkie składniki: mąkę, jajka, cukier, aromat waniliowy itp. W tym przykładzie przepis spełnia rolę programu (tzn. algorytmu wyrażonego w odpowiedniej notacji), informatyk jest procesorem (CPU), natomiast składniki ciasta odgrywają rolę danych wejściowych. Proces jest operacją, w której informatyk czyta przepis, dodaje składniki i piecze ciasto.

Wyobraźmy sobie teraz, że z krzykiem wbiega syn informatyka i mówi, że użądliła go pszczoła. Informatyk zapamiętuje, w którym miejscu przepisu się znajdował (zapisuje bieżący stan procesu), bierze książkę o pierwszej pomocy i zaczyna postępować zgodnie z zapisanymi w niej wskazówkami. W tym momencie widzimy przełączenie się procesora z jednego procesu (pieczenie) do procesu o wyższym priorytecie (udzielanie pomocy medycznej). Przy czym każdy z procesów ma inny program (przepis na ciasto, książka pierwszej pomocy medycznej). Kiedy informatyk poradzi sobie z opatrzeniem użądlenia, powraca do pieczenia ciasta i kontynuuje od miejsca, w którym skończył.

Kluczowe znaczenie ma uświadomienie sobie, że proces jest pewnym działaniem. Charakteryzuje się programem, wejściem, wyjściem i stanem. Jeden procesor może być współdzielony przez kilka procesów za pomocą algorytmu szeregowania. Algorytm ten decyduje, kiedy

Kup ksi k Pole ksi k

zatrzymać pracę nad jednym programem i rozpocząć obsługę innego. Natomiast program jest czymś, co może być przechowywane na dysku i niczego nie robić.

Warto zwrócić uwagę na to, że jeśli program uruchomi się dwa razy, liczy się jako dwa procesy. Często np. istnieje możliwość dwukrotnego uruchomienia edytora tekstu lub jednoczesnego drukowania dwóch plików, jeśli system komputerowy jest wyposażony w dwie drukarki. Fakt, że dwa działające procesy korzystają z tego samego programu, nie ma znaczenia — są to oddzielne procesy. System operacyjny może mieć możliwość współdzielenia kodu pomiędzy nimi w taki sposób, że w pamięci znajduje się jedna kopia. Jest to jednak szczegół techniczny, który nie zmienia faktu działania dwóch procesów.

2.1.2. Tworzenie procesów

Systemy operacyjne wymagają sposobu tworzenia procesów. W bardzo prostych systemach lub w systemach zaprojektowanych do uruchamiania tylko jednej aplikacji (np. kontrolera w kuchence mikrofalowej), bywa możliwe zainicjowanie wszystkich potrzebnych procesów natychmiast po uruchomieniu systemu. Jednak w systemach ogólnego przeznaczenia potrzebny jest sposób tworzenia i niszczenia procesów podczas ich działania. W tym punkcie przyjrzymy się niektórym spośród tych mechanizmów.

Są cztery podstawowe zdarzenia, które powodują tworzenie procesów:

1. Inicjalizacja systemu.
2. Uruchomienie wywołania systemowego tworzącego proces przez działający proces.
3. Żądanie użytkownika utworzenia nowego procesu.
4. Zainicjowanie zadania wsadowego.

W momencie rozruchu systemu operacyjnego zwykle tworzonych jest kilka procesów.

Niektóre z nich są procesami pierwszego planu — tzn. są to procesy, które komunikują się z użytkownikami i wykonują dla nich pracę. Inne są procesami drugoplanowymi, które nie są powiązane z określonym użytkownikiem, ale spełniają pewną specyficzną funkcję. I tak jeden proces drugoplanowy może być zaprojektowany do akceptacji wchodzących wiadomości e-mail. Taki proces może być uśpiony przez większość dnia i nagle się uaktywnić, kiedy nadchodzi wiadomość e-mail. Inny proces drugoplanowy może być zaprojektowany do akceptacji wchodzących żądań stron WWW zapisanych na serwerze. Proces ten budzi się w momencie odebrania żądania strony WWW w celu jego obsłużenia. Procesy działające na drugim planie, które są przeznaczone do obsługi pewnych operacji, takich jak odbiór wiadomości e-mail, serwowanie stron WWW, aktualności, drukowanie itp., są określane jako **demony**. W dużych systemach zwykle działają dziesiątki takich procesów. W systemie UNIX[1], aby wyświetlić listę działających procesów, można skorzystać z programu `ps`. W systemie Windows można skorzystać z menedżera zadań.

Procesy mogą być tworzone nie tylko w czasie rozruchu, ale także później. Działający proces często wydaje wywołanie systemowe w celu utworzenia jednego lub kilku nowych procesów mających pomóc w realizacji zadania. Tworzenie nowych procesów jest szczególnie przydatne, kiedy pracę do wykonania można łatwo sformułować w kontekście kilku związanych ze sobą, ale poza tym niezależnych, współdziałających ze sobą procesów. Jeśli np. przez sieć jest pobierana duża ilość danych w celu ich późniejszego przetwarzania, to można utworzyć jeden proces, który pobiera dane i umieszcza je we współdzielonym buforze, oraz drugi proces, który usuwa dane

[1] W tym rozdziale pod pojęciem systemu UNIX będziemy rozumieć prawie wszystkie systemy oparte na POSIX, w tym Linux, FreeBSD, OS X, Solaris itp., a także do pewnego stopnia Android i iOS.

Kup ksi k Pole ksi k

z bufora i je przetwarza. W systemie wieloprocesorowym, w którym każdy z procesów może działać na innym procesorze, zadanie może być wykonane w krótszym czasie.

W systemach interaktywnych użytkownicy mogą uruchomić program poprzez wpisanie polecenia lub kliknięcie (ewentualnie dwukrotne kliknięcie) ikony. Wykonanie dowolnej z tych operacji inicjuje nowy proces i uruchamia w nim wskazany program. W systemach uniksowych bazujących na systemie X Window nowy proces przejmuje okno, w którym został uruchomiony. W systemie Windows po uruchomieniu procesu nie ma on przypisanego okna. Może on jednak stworzyć jedno (lub więcej) okien i większość systemów to robi. W obydwu systemach użytkownicy mają możliwość jednoczesnego otwarcia wielu okien, w których działają jakieś procesy. Za pomocą myszy użytkownik może wybrać okno i komunikować się z procesem, np. podawać dane wejściowe wtedy, kiedy są potrzebne.

Ostatnia sytuacja, w której są tworzone procesy, dotyczy tylko systemów wsadowych w dużych komputerach mainframe. Rozważmy działanie systemu zarządzania stanami magazynowymi sieci sklepów pod koniec dnia — obliczanie, co zamówić, analizowanie popularności produktów w poszczególnych sklepach itp. W systemach tego typu użytkownicy mogą przesyłać do systemu zadania wsadowe (czasami zdalnie). Kiedy system operacyjny zdecyduje, że ma zasoby wystarczające do uruchomienia innego zadania, tworzy nowy proces i uruchamia następne zadanie z kolejki.

Z formalnego punktu widzenia we wszystkich tych sytuacjach proces tworzy się poprzez zlecenie istniejącemu procesowi wykonania wywołania systemowego tworzenia procesów. Może to być działający proces użytkownika, proces systemowy, wywołany z klawiatury lub za pomocą myszy, albo proces zarządzania zadaniami systemowymi. Proces ten wykonuje wywołanie systemowe tworzące nowy proces. To wywołanie systemowe zleca systemowi operacyjnemu utworzenie nowego procesu i wskazuje, w sposób pośredni lub bezpośredni, jaki program należy w nim uruchomić. Aby wszystko przebiegało płynnie, pierwszy proces jest tworzony „na sztywno” podczas rozruchu systemu.

W systemie UNIX istnieje tylko jedno wywołanie systemowe do utworzenia nowego procesu: `fork`. Wywołanie to tworzy dokładny klon procesu wywołującego. Po wykonaniu instrukcji `fork` procesy rodzic i dziecko mają ten sam obraz pamięci, te same zmienne środowiskowe oraz te same otwarte pliki. Po prostu są identyczne. Wtedy zazwyczaj proces dziecko uruchamia wywołanie `execve` lub podobne wywołanie systemowe w celu zmiany obrazu pamięci i uruchomienia nowego programu. Kiedy użytkownik wpisze polecenie w środowisku powłoki, np. `sort`, powłoka najpierw za pomocą wywołania `fork` tworzy proces dziecko, a następnie proces dziecko wykonuje polecenie `sort`. Powodem, dla którego dokonuje się ten dwuetapowy proces, jest umożliwienie procesowi dziecku manipulowania deskryptorami plików po wykonaniu wywołania fork, ale przed wywołaniem execve w celu przekierowania standardowego wejścia, standardowego wyjścia oraz standardowego urządzenia błędów.

Dla odróżnienia w systemie Windows jedna funkcja interfejsu Win32 — `CreateProcess` — jest odpowiedzialna zarówno za utworzenie procesu, jak i załadowanie odpowiedniego programu do nowego procesu. Wywołanie to ma 10 parametrów. Są to program do uruchomienia, parametry wiersza polecenia przekazywane do programu, różne atrybuty zabezpieczeń, bity decydujące o tym, czy otwarte pliki będą dziedziczone, informacje dotyczące priorytetów, specyfikacja okna, jakie ma być utworzone dla procesu (jeśli proces ma mieć okno), oraz wskaźnik do struktury, w której są zwracane do procesu wywołującego informacje o nowo tworzonym procesie. Oprócz wywołania `CreateProcess` interfejs Win32 zawiera około 100 innych funkcji do zarządzania i synchronizowania procesów oraz wykonywania powiązanych z tym operacji.

Kup ksi k Pole ksi k

Zarówno w systemie UNIX, jak i Windows po utworzeniu procesu rodzic i dziecko mają osobne przestrzenie adresowe. Jeśli dowolny z procesów zmieni słowo w swojej przestrzeni adresowej, zmiana nie jest widoczna dla drugiego procesu. W systemie UNIX początkowa przestrzeń adresowa procesu dziecka jest kopią przestrzeni adresowej procesu rodzica. Są to jednak całkowicie odrębne przestrzenie adresowe. Zapisywalna pamięć nie jest współdzielona pomiędzy procesami (w niektórych implementacjach Uniksa tekst programu jest współdzielony pomiędzy procesami rodzica i dziecka, ponieważ nie może on być modyfikowany). Inne rozwiązanie polega na współużytkowaniu przez proces dziecko pamięci procesu rodzica, ale w tym wypadku pamięć jest współdzielona w trybie **kopiuj przy zapisie** (ang. *copy-on-write*). To oznacza, że zawsze, gdy jeden z dwóch procesów chce zmienić część pamięci, najpierw fizycznie ją kopiuje, aby mieć pewność, że modyfikacja następuje w prywatnym obszarze pamięci. Tak jak wcześniej, nie ma współdzielenia zapisywalnych obszarów pamięci. Nowo utworzony proces może jednak współdzielić niektóre inne zasoby procesu swojego twórcy — np. otwarte pliki. W systemie Windows przestrzenie adresowe procesów rodzica i dziecka od samego początku są różne.

2.1.3. Kończenie działania procesów

Po utworzeniu proces zaczyna działanie i wykonuje swoje zadania. Nic jednak nie trwa wiecznie — nawet procesy. Prędzej czy później nowy proces zakończy swoje działanie. Zwykle dzieje się to z powodu jednego z poniższych warunków:

1. Normalne zakończenie pracy (dobrowolnie).
2. Zakończenie pracy w wyniku błędu (dobrowolnie).
3. Błąd krytyczny (przymusowo).
4. Zabicie przez inny proces (przymusowo).

Większość procesów kończy działanie dlatego, że wykonały swoją pracę. Kiedy kompilator skompiluje program, wykonuje wywołanie systemowe, które informuje system operacyjny o zakończeniu pracy. Tym wywołaniem jest `exit` w systemie UNIX oraz `ExitProcess` w systemie Windows. W programach wyposażonych w interfejs ekranowy zwykle są mechanizmy pozwalające na dobrowolne zakończenie działania. W edytorach tekstu, przeglądarkach internetowych i podobnych im programach zawsze jest ikona lub polecenie menu, które użytkownik może kliknąć, aby zlecić procesowi usunięcie otwartych plików tymczasowych i zakończenie działania.

Innym powodem zakończenia pracy jest sytuacja, w której proces wykryje błąd krytyczny. Jeśli np. użytkownik wpisze polecenie:

```
cc foo.c
```

w celu skompilowania programu *foo.c*, a taki plik nie istnieje, to kompilator po prostu skończy działanie. Procesy interaktywne wyposażone w interfejsy ekranowe zwykle nie kończą działania, jeśli zostaną do nich przekazane błędne parametry. Zamiast tego wyświetlają okno dialogowe z prośbą do użytkownika o ponowienie próby.

Trzecim powodem zakończenia pracy jest błąd spowodowany przez proces — często wynikający z błędu w programie. Może to być uruchomienie niedozwolonej instrukcji, odwołanie się do nieistniejącego obszaru pamięci lub dzielenie przez zero. W niektórych systemach (np. w Uniksie) proces może poinformować system operacyjny, że sam chce obsłużyć określone błędy. W takim przypadku, jeśli wystąpi błąd, proces otrzymuje sygnał (przerwanie), zamiast zakończyć pracę.

Czwartym powodem, dla którego proces może zakończyć działanie, jest wykonanie wywołania systemowego, które zleca systemowi operacyjnemu zniszczenie innego procesu. W Uniksie można

Kup ksi k Pole ksi k

to zrobić za pomocą wywołania systemowego `kill`. Odpowiednikiem tego wywołania w interfejsie Win32 API jest `TerminateProcess`. W obu przypadkach proces niszczący musi posiadać odpowiednie uprawnienia do niszczenia innych procesów. W niektórych systemach zakończenie procesu — niezależnie od tego, czy jest wykonywane dobrowolnie, czy przymusowo — wiąże się z zakończeniem wszystkich procesów utworzonych przez ten proces. Jednak w taki sposób nie działa ani UNIX, ani Windows.

2.1.4. Hierarchie procesów

W niektórych systemach, kiedy proces utworzy inny proces, to proces rodzic jest w pewien sposób związany z procesem dzieckiem. Proces dziecko sam może tworzyć kolejne procesy, co formuje hierarchię procesów. Zwróćmy uwagę, że w odróżnieniu od roślin i zwierząt rozmnażających się płciowo proces może mieć tylko jednego rodzica (ale zero, jedno dziecko lub więcej dzieci). Tak więc proces przypomina bardziej hydrę niż, powiedzmy, cielę.

W Uniksie proces wraz z wszystkimi jego dziećmi i dalszymi potomkami tworzy grupę procesów. Kiedy użytkownik wyśle sygnał z klawiatury (np. poprzez wciśnięcie *CTRL-C*), sygnał ten jest dostarczany do wszystkich członków grupy procesów, które w danym momencie są powiązane z klawiaturą (zwykle są to wszystkie aktywne procesy utworzone w bieżącym oknie). Każdy proces może indywidualnie przechwycić sygnał, zignorować go lub podjąć działanie domyślne — tzn. zostać zniszczonym przez sygnał.

W celu przedstawienia innego przykładu sytuacji, w której hierarchia procesów odgrywa rolę, przyjrzyjmy się sposobowi, w jaki system UNIX inicjuje się podczas rozruchu. W obrazie rozruchowym występuje specjalny proces o nazwie `init`. Kiedy rozpoczyna działanie, odczytuje plik i informuje o liczbie dostępnych terminali. Następnie tworzy po jednym nowym procesie na terminal. Procesy te czekają, aż ktoś się zaloguje. Kiedy logowanie zakończy się pomyślnie, proces logowania uruchamia powłokę, która jest gotowa na przyjmowanie poleceń. Polecenia te mogą uruchamiać nowe procesy itd. Tak więc wszystkie procesy w całym systemie należą do tego samego drzewa — jego korzeniem jest proces `init`.

Dla odróżnienia w systemie Windows nie występuje pojęcie hierarchii procesów. Wszystkie procesy są sobie równe. Jedyną oznaką hierarchii procesu jest to, że podczas tworzenia procesu rodzic otrzymuje specjalny znacznik (nazywany **uchwytem** — ang. *handle*), który może wykorzystać do zarządzania dzieckiem. Może jednak swobodnie przekazać ten znacznik do innego procesu i w ten sposób zdezaktualizować hierarchię. Procesy w Uniksie nie mają możliwości „wydziedziczenia" swoich dzieci.

2.1.5. Stany procesów

Chociaż każdy proces jest niezależnym podmiotem, posiadającym własny licznik programu i wewnętrzny stan, procesy często muszą się komunikować z innymi procesami. Jeden proces może generować wyjście, które inny proces wykorzysta jako wejście. W poleceniu:

```
cat rozdzial1 rozdzial2 rozdzial3 | grep drzewo
```

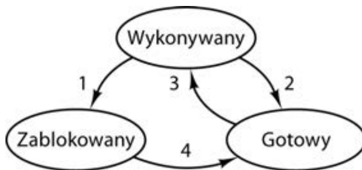
pierwszy proces uruchamia polecenie `cat`, łączy i wyprowadza trzy pliki. Drugi proces uruchamia polecenie `grep`, wybiera wszystkie wiersze zawierające słowo „drzewo". W zależności od względnej szybkości obu procesów (co z kolei zależy zarówno od względnej złożoności programów, jak i tego, ile czasu procesora każdy z nich ma do dyspozycji) może się zdarzyć, że polecenie `grep`

Kup ksi k Pole ksi k

będzie gotowe do działania, ale nie będą na nie czekały żadne dane wejściowe. Proces będzie się musiał zablokować do czasu, aż będą one dostępne.

Proces blokuje się, ponieważ z logicznego punktu widzenia nie może kontynuować działania. Zazwyczaj dzieje się tak dlatego, że oczekuje na dane wejściowe, które jeszcze nie są dostępne. Jest również możliwe, że proces, który jest gotowy i zdolny do działania, zostanie zatrzymany ze względu na to, że system operacyjny zdecydował się przydzielić procesor na pewien czas jakiemuś innemu procesowi. Te dwie sytuacje diametralnie różnią się od siebie. W pierwszym przypadku wstrzymanie pracy jest ściśle związane z charakterem problemu (nie można przetworzyć wiersza poleceń wprowadzanego przez użytkownika do czasu, kiedy użytkownik go nie wprowadzi). W drugim przypadku to techniczne aspekty systemu (niewystarczająca liczba procesorów do tego, aby każdy proces otrzymał swój prywatny procesor). Na rysunku 2.2 pokazano diagram stanów prezentujący trzy stany, w jakich może znajdować się proces:

1. Działanie (rzeczywiste korzystanie z procesora w tym momencie).
2. Gotowość (proces może działać, ale jest tymczasowo wstrzymany, aby inny proces mógł działać).
3. Blokada (proces nie może działać do momentu, w którym wydarzy się jakieś zewnętrzne zdarzenie).



1. Proces blokuje się w oczekiwaniu na dane wejściowe
2. Program szeregujący przydzielił procesor innemu procesowi
3. Program szeregujący przydzielił procesor temu procesowi
4. Dane wejściowe stają się dostępne

Rysunek 2.2. Proces może byś w stanie działania, blokady lub gotowości. Na rysunku pokazano przejścia pomiędzy tymi stanami

Z logicznego punktu widzenia pierwsze dwa stany są do siebie podobne. W obu przypadkach proces chce działać, ale w drugim przypadku chwilowo brakuje dla niego czasu procesora. Trzeci stan różni się od pierwszych dwóch w tym sensie, że proces nie może działać nawet wtedy, gdy procesor w tym czasie nie ma innego zajęcia.

Tak jak pokazano na rysunku, pomiędzy tymi trzema stanami możliwe są cztery przejścia. Przejście nr 1 występuje wtedy, kiedy system operacyjny wykryje, że proces nie może kontynuować działania. W niektórych systemach proces może wykonać wywołanie systemowe, np. `pause`, w celu przejścia do stanu zablokowania. W innych systemach, w tym w Uniksie, kiedy proces czyta dane z potoku lub pliku specjalnego (np. terminala) i dane wejściowe są niedostępne, jest automatycznie blokowany.

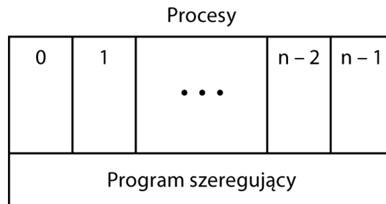
Przejścia nr 2 i nr 3 są realizowane przez **program szeregujący** (ang. *process scheduler*) — część systemu operacyjnego, a procesy nie są o tym nawet informowane. Przejście nr 2 za-chodzi wtedy, gdy program szeregujący zdecyduje, że działający proces działał wystarczająco długo i nadszedł czas, by przydzielić czas procesora jakiemuś innemu procesowi. Przejście nr 3 zachodzi wtedy, gdy wszystkie inne procesy skorzystały ze swojego udziału i nadszedł czas na to, by pierwszy proces otrzymał procesor i wznowił działanie. Zadanie szeregowania procesów — tzn. decydowania o tym, który proces powinien się uruchomić, kiedy i na jak długo — jest bardzo ważne. Opracowano wiele algorytmów mających na celu zapewnienie równowagi pomiędzy wymaganiami wydajności systemu jako całości oraz sprawiedliwego przydziału procesora do indywidualnych procesów. Niektóre z tych algorytmów omówimy w dalszej części rozdziału.

Kup ksi k Pole ksi k

Przejście nr 4 występuje wtedy, gdy zachodzi zewnętrzne zdarzenie, na które proces oczekiwał (np. nadejście danych wejściowych). Jeśli w tym momencie nie działa żaden inny proces, zajdzie przejście nr 3 i proces rozpocznie działanie. W innym przypadku może być zmuszony do oczekiwania w stanie gotowości przez pewien czas, aż procesor stanie się dostępny i nadejdzie jego kolejka.

Wykorzystanie modelu procesów znacznie ułatwia myślenie o tym, co dzieje się wewnątrz systemu. Niektóre procesy uruchamiają programy realizujące polecenia wprowadzane przez użytkownika. Inne procesy są częścią systemu i obsługują takie zadania, jak obsługa żądań usług plikowych lub zarządzanie szczegółami dotyczącymi uruchamiania napędu dysku lub taśm. Kiedy zachodzi przerwanie dyskowe, system podejmuje decyzję o zatrzymaniu działania bieżącego procesu i uruchamia proces dyskowy, który był zablokowany w oczekiwaniu na to przerwanie. Tak więc zamiast myśleć o przerwaniach, możemy myśleć o procesach użytkownika, procesach dysku, procesach terminala itp., które blokują się w czasie oczekiwania, aż coś się wydarzy. Kiedy nastąpi próba czytania danych z dysku albo użytkownik przyciśnie klawisz, proces oczekujący na to zdarzenie jest odblokowywany i może wznowić działanie.

Ten stan rzeczy jest podstawą modelu pokazanego na rysunku 2.3. W tym przypadku na najniższym poziomie systemu operacyjnego znajduje się program szeregujący, zarządzający zbiorem procesów występujących w warstwie nad nim. Cały mechanizm obsługi przerwań i szczegółów związanych z właściwym uruchamianiem i zatrzymywaniem procesów jest ukryty w elemencie nazwanym tu zarządcą procesów. Element ten w rzeczywistości nie zawiera zbyt wiele kodu. Pozostała część systemu operacyjnego ma strukturę procesów. W praktyce jednak istnieje bardzo niewiele systemów operacyjnych, które miałyby tak przejrzystą strukturę.



Rysunek 2.3. Najniższa warstwa systemu operacyjnego o strukturze procesów zarządza przerwaniami i szeregowaniem. Powyżej tej warstwy znajdują się sekwencyjne procesy

2.1.6. Implementacja procesów

W celu zaimplementowania modelu procesów w systemie operacyjnym występuje tabela (tablica struktur), zwana **tabelą procesów**, w której każdemu z procesów odpowiada jedna pozycja — niektórzy autorzy nazywają te pozycje **blokami zarządzania procesami**. W blokach tych są zapisane ważne informacje na temat stanu procesu. Zawierają one wartości licznika programu, wskaźnika stosu, dane dotyczące przydziału pamięci, statusu otwartych procesów, rozliczeń i szeregowania oraz wszystkie inne informacje, które trzeba zapisać w czasie przełączania procesu ze stanu wykonywany do stanu gotowy lub zablokowany. Dzięki nim proces może być później wznowiony, tak jakby nigdy nie został zatrzymany.

Kilka kluczowych pól w typowym systemie pokazano w tabeli 2.1. Pola w pierwszej kolumnie są związane z zarządzaniem procesami. Pozostałe dwa łączą się odpowiednio z zarządzaniem pamięcią oraz zarządzaniem plikami. Należy zwrócić uwagę na to, że obecność poszczególnych pól w tabeli procesów w dużym stopniu zależy od systemu. Poniższa tabela daje jednak ogólny obraz rodzajów potrzebnych informacji.

Kup ksi k Pole ksi k

Tabela 2.1. Przykładowe pola typowego wpisu w tabeli procesów

Zarządzanie procesami	Zarządzanie pamięcią	Zarządzanie plikami
Rejestry	Wskaźnik do informacji segmentu tekstu	Katalog główny
Licznik programu	Wskaźnik do informacji segmentu danych	Katalog roboczy
Słowo stanu programu	Wskaźnik do informacji segmentu stosu	Deskryptory plików
Wskaźnik stosu		Identyfikator użytkownika
Stan procesu		Identyfikator grupy
Priorytet		
Parametry szeregowania		
Identyfikator procesu		
Proces rodzic		
Grupa procesów		
Sygnały		
Czas rozpoczęcia procesu		
Wykorzystany czas CPU		
Czas CPU procesów dzieci		
Godzina następnego alarmu		

Teraz, gdy przyjrzeliśmy się tabeli procesów, możemy wyjaśnić nieco dokładniej to, w jaki sposób iluzja wielu sekwencyjnych procesów jest utrzymywana w jednym procesorze (lub każdym z procesorów), a także opisać przerwania nieco bardziej szczegółowo niż w rozdziale 1. Z każdą klasą wejścia-wyjścia wiąże się lokalizacja (zwykle pod ustalonym adresem w dolnej części pamięci) zwana **wektorem przerwań**. Jest w niej zapisany adres **procedury obsługi przerwania** (ang. *interrupt service routine* — ISR). Załóżmy, że w momencie wystąpienia przerwania związanego z dyskiem ma działać proces użytkownika nr 3. Sprzęt obsługujący przerwania odkłada na stos licznik programu procesu użytkownika nr 3, słowo stanu programu i czasami jeden lub kilka rejestrów. Następnie sterowanie przechodzi pod adres określony w wektorze przerwań. To jest wszystko, co robi sprzęt. Odtąd wszystko zależy od procedur ISR w oprogramowaniu.

Obsługa każdego przerwania rozpoczyna się od zapisania rejestrów — często pod pozycją tabeli procesów odpowiadającą bieżącemu procesowi. Następnie informacje odłożone na stos przez mechanizm obsługi przerwania są z niego zdejmowane, a wskaźnik stosu jest ustawiany na adres tymczasowego stosu używanego przez procedurę obsługi procesu. Takich działań, jak zapisanie rejestrów i ustawienie wskaźnika stosu, nawet nie można wyrazić w językach wysokopoziomowych, np. w C. W związku z tym operacje te są wykonywane przez niewielką procedurę w języku asemblera. Zazwyczaj jest to ta sama procedura dla wszystkich przerwań, ponieważ zadanie zapisania rejestrów jest identyczne, niezależnie od tego, co było przyczyną przerwania.

Kiedy ta procedura zakończy działanie, wywołuje procedurę w języku C, która wykonuje resztę pracy dla tego konkretnego typu przerwania (zakładamy, że system operacyjny został napisany w języku C — w tym języku napisana jest większość systemów operacyjnych). Kiedy procedura ta wykona swoje zadanie (co może spowodować, że pewne procesy uzyskają gotowość do działania), wywoływany jest program szeregujący, który ma sprawdzić, jaki proces powinien zostać uruchomiony w następnej kolejności. Następnie sterowanie jest przekazywane z powrotem do kodu w asemblerze, który ładuje rejestry i mapę pamięci nowego bieżącego procesu oraz rozpoczyna jego działanie. Obsługę przerwań i szeregowanie podsumowano w tabeli 2.2. Warto zwrócić uwagę, że różne systemy nieco się różnią pewnymi szczegółami.

Kup ksi k Pole ksi k

Tabela 2.2. Szkielet działań wykonywanych przez najniższy poziom systemu operacyjnego w momencie wystąpienia przerwania. Poszczególne systemy operacyjne mogą się różnić pewnymi szczegółami

1.	Sprzęt odkłada na stos licznik programu itp.
2.	Sprzęt ładuje nowy licznik programu z wektora przerwań
3.	Procedura w języku asemblera zapisuje rejestry
4.	Procedura w języku asemblera ustawia nowy stos
5.	Uruchamia się procedura obsługi przerwania w C (zazwyczaj czyta i buforuje dane wejściowe)
6.	Program szeregujący decyduje o tym, który proces ma być uruchomiony w następnej kolejności
7.	Procedura w języku C zwraca sterowanie do kodu w asemblerze
8.	Procedura w języku asemblera uruchamia nowy bieżący proces

Proces może być przerwany tysiące razy w trakcie działania, ale kluczową ideą jest to, że po każdym przerwaniu proces powraca dokładnie do tego stanu, w jakim był przed wystąpieniem przerwania.

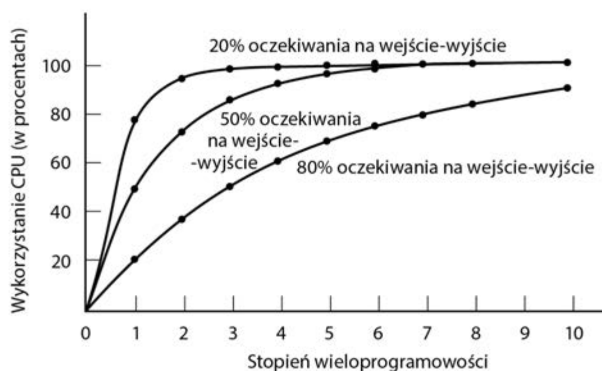
2.1.7. Modelowanie wieloprogramowości

Zastosowanie wieloprogramowości pozwala na poprawę wykorzystania procesora. Z grubsza rzecz biorąc, jeśli przeciętny proces jest przetwarzany przez 20% czasu rezydowania w pamięci, to w przypadku gdy w pamięci jest jednocześnie pięć procesów, procesor powinien być zajęty przez cały czas. Ten model jest jednak nierealistycznie optymistyczny, ponieważ zakłada, że w żadnym momencie nie zdarzy się sytuacja, w której wszystkie pięć procesów będzie jednocześnie oczekiwało na operację wejścia-wyjścia.

Lepszym modelem jest spojrzenie na wykorzystanie procesora z probabilistycznego punktu widzenia. Załóżmy, że proces spędza fragment p swojego czasu na zakończeniu operacji wejścia-wyjścia. Przy n procesach znajdujących się jednocześnie w pamięci prawdopodobieństwo tego, że wszystkie n procesów będzie jednocześnie oczekiwało na obsługę wejścia-wyjścia (wtedy procesor pozostanie bezczynny), wynosi p^n. W takim przypadku wykorzystanie procesora można opisać za pomocą wzoru:

$$\text{Wykorzystanie procesora} = 1 - p^n$$

Na rysunku 2.4 pokazano procent wykorzystania procesora w funkcji n — co określa się jako **stopień wieloprogramowości**.



Rysunek 2.4. Wykorzystanie procesora w funkcji liczby procesów w pamięci

Z rysunku jasno wynika, że jeśli procesy spędzają 80% czasu w oczekiwaniu na operacje wejścia-wyjścia, to aby współczynnik marnotrawienia procesora utrzymać na poziomie poniżej 10%, w pamięci musi być jednocześnie co najmniej 10 procesów. Kiedy zdamy sobie sprawę ze stanu, w którym proces interaktywny oczekuje, aż użytkownik wpisze na terminalu jakieś dane, stanie się oczywiste, że czasy oczekiwania na wejścia-wyjścia rzędu 80% i więcej nie są niczym niezwykłym. Nawet na serwerach procesy wykonujące wiele dyskowych operacji wejścia-wyjścia często charakteryzują się tak wysokim procentem.

Dla ścisłości należy dodać, że model probabilistyczny opisany przed chwilą jest tylko przybliżeniem. Zakłada on niejawnie, że wszystkie *n* procesów jest niezależnych. Oznacza to, że w przypadku systemu z pięcioma procesami w pamięci dopuszczalnym stanem jest to, aby trzy z nich działały, a dwa czekały. W związku z tym proces, który osiąga gotowość w czasie, gdy procesor jest zajęty, będzie musiał czekać. Tak więc procesy nie są niezależne. Dokładniejszy model można stworzyć z wykorzystaniem teorii kolejkowania, jednak teza, którą sformułowaliśmy — wieloprogramowość pozwala procesom wykorzystywać procesor w czasie, gdy w innej sytuacji byłby on bezczynny — jest oczywiście w dalszym ciągu prawdziwa. Faktu tego nie zmieniłaby nawet sytuacja, w której rzeczywiste krzywe stopnia wieloprogramowości nieco odbiegałyby od tych pokazanych na rysunku 2.4.

Mimo że model z rysunku 2.4 jest uproszczony, można go wykorzystywać w celu tworzenia specyficznych, jednak przybliżonych prognoz dotyczących wydajności procesora. Przypuśćmy, że komputer ma 8 GB pamięci, przy czym system operacyjny zajmuje 2 GB, a każdy program użytkownika również zajmuje do 2 GB. Te rozmiary pozwalają na to, aby w pamięci jednocześnie znajdowały się trzy programy użytkownika. Przy średnim czasie oczekiwania na operacje wejścia-wyjścia wynoszącym 80% mamy procent wykorzystania procesora (pomijając narzut systemu operacyjnego) na poziomie $1 - 0{,}8^3$ czyli około 49%. Dodanie kolejnych 8 GB pamięci operacyjnej umożliwia przejście systemu z trójstopniowej wieloprogramowości do siedmiostopniowej, co przyczyni się do wzrostu wykorzystania procesora do 79%. Mówiąc inaczej, dodatkowe 8 GB pamięci podniesie przepustowość o 30%.

Dodanie kolejnych 8 GB spowodowałoby zwiększenie stopnia wykorzystania procesora z 79% do 91%, a zatem podniosłoby przepustowość tylko o 12%. Korzystając z tego modelu, właściciel komputera może zadecydować, że pierwsza rozbudowa systemu jest dobrą inwestycją, natomiast druga nie.

2.2. WĄTKI

W tradycyjnych systemach operacyjnych każdy proces ma przestrzeń adresową i jeden wątek sterowania. W rzeczywistości prawie tak wygląda definicja procesu. Niemniej jednak często występują sytuacje, w których korzystne jest posiadanie wielu wątków sterowania w tej samej przestrzeni adresowej, działających quasi-równolegle — tak jakby były (niemal) oddzielnymi procesami (z wyjątkiem współdzielonej przestrzeni adresowej). Sytuacje te oraz wynikające z tego implikacje omówiono w kolejnych punktach. Później przeanalizujemy inne rozwiązanie.

2.2.1. Wykorzystanie wątków

Do czego może służyć rodzaj procesu wewnątrz innego procesu? Okazuje się, że istnieją powody istnienia tych miniprocesów zwanych **wątkami**. Spróbujmy przyjrzeć się kilku z nich. Głównym powodem występowania wątków jest to, że w wielu aplikacjach jednocześnie wykonywanych

Kup ksi k Pole ksi k

jest wiele działań. Niektóre z nich mogą być zablokowane od czasu do czasu. Dzięki dekompozycji takiej aplikacji na wiele sekwencyjnych wątków działających quasi-równolegle model programowania staje się prostszy.

Taką samą dyskusję przedstawiliśmy już wcześniej. Dokładnie te same argumenty przemawiają za istnieniem procesów. Zamiast myśleć o przerwaniach, licznikach czasu i przełączaniu kontekstu, możemy myśleć o równoległych procesach. Tyle że teraz, przy pojęciu wątków, dodajemy nowy element: zdolność równoległych podmiotów do współdzielenia pomiędzy sobą przestrzeni adresowej oraz wszystkich swoich danych. Zdolność ta ma kluczowe znaczenie dla niektórych aplikacji, dlatego właśnie obecność wielu procesów (z oddzielnymi przestrzeniami adresowymi) w tym przypadku nie wystarczy.

Drugi argument, który przemawia za istnieniem wątków, jest taki, że — ponieważ są one mniejsze od procesów — w porównaniu z procesami łatwiej (tzn. szybciej) się je tworzy i niszczy. W wielu systemach tworzenie wątku trwa 10 – 100 razy krócej od tworzenia procesu. Ponieważ liczba potrzebnych wątków zmienia się dynamicznie i gwałtownie, szybkość nabiera dużego znaczenia.

Trzecim powodem istnienia wątków są względy wydajności. Istnienie wątków nie poprawi wydajności, jeśli wszystkie one będą związane z procesorem. Jednak w przypadku wykonywania intensywnych obliczeń i jednocześnie znaczącej liczby operacji wejścia-wyjścia występowanie wątków pozwala na nakładanie się na siebie tych działań, co w efekcie końcowym przyczynia się do przyspieszenia aplikacji.

Na koniec — wątki przydają się w systemach wyposażonych w wiele procesorów, gdzie możliwa jest rzeczywista współbieżność. Do tego zagadnienia powrócimy w rozdziale 8.

Najłatwiej przekonać się o przydatności wątków, analizując konkretne przykłady. W roli pierwszego przykładu rozważmy edytor tekstu. Edytory tekstu zazwyczaj wyświetlają na ekranie tworzony dokument sformatowany dokładnie w takiej postaci, w jakiej będzie on wyglądał na drukowanej stronie. Zwłaszcza wszystkie znaki podziału wierszy i stron znajdują się na prawidłowych i ostatecznych pozycjach. Dzięki temu użytkownik ma możliwość przeglądania i poprawienia dokumentu, jeśli zajdzie taka potrzeba (np. w celu wyeliminowania sierot i wdów — niekompletnych wierszy na początku i na końcu strony, które uważa się za nieestetyczne).

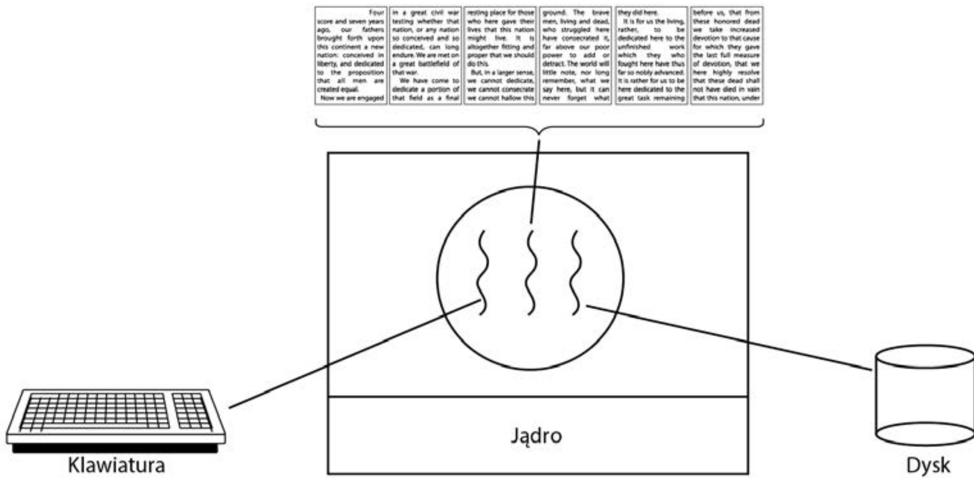
Załóżmy, że użytkownik pisze książkę. Z punktu widzenia autora najłatwiej umieścić całą książkę w pojedynczym pliku, tak by łatwiej było wyszukiwać tematy, wykonywać globalne operacje zastępowania itp. Można również umieścić każdy z rozdziałów w osobnym pliku. Jednak umieszczenie każdego podrozdziału i punktu w osobnym pliku, np. gdyby zaszła potrzeba globalnego zastąpienia jakiegoś terminu w całej książce, byłoby prawdziwym utrapieniem. W takim przypadku trzeba by było bowiem indywidualnie edytować każdy z kilkuset plików. Jeśli np. zaproponowany termin „standard xxxx" zostałby zatwierdzony tuż przed oddaniem książki do druku, trzeba by było w ostatniej chwili zastąpić wszystkie wystąpienia terminu „roboczo: standard xxxx" na „standard xxxx". Jeśli książka znajduje się w jednym pliku, taką operację można wykonać za pomocą jednego polecenia. Dla odróżnienia, gdyby książka składała się z 300 plików, każdy z nich trzeba by osobno otworzyć w edytorze.

Rozważmy teraz, co się zdarzy, kiedy użytkownik nagle usunie jedno zdanie z pierwszej strony 800-stronicowego dokumentu. Po sprawdzeniu poprawności zmodyfikowanej strony zdecydował, że chce wykonać inną zmianę na stronie 600 i wpisuje polecenie zlecające edytorowi przejście do tej strony (np. poprzez wyszukanie frazy, która znajduje się tylko tam). Edytor tekstu jest w tej sytuacji zmuszony do natychmiastowego przeformatowania całej książki do strony 600, ponieważ nie będzie wiedział, jaką treść ma pierwszy wiersz na stronie 600, dopóki nie przetworzy wszystkich poprzednich stron. Zanim będzie można wyświetlić stronę 600, może powstać znaczące opóźnienie, co doprowadzi do niezadowolenia użytkownika. W takim przypadku może pomóc wykorzystanie wątków.

Kup ksi k Pole ksi k

Załóżmy, że edytor tekstu jest napisany jako program składający się z dwóch wątków. Jeden wątek zajmuje się komunikacją z użytkownikiem, a drugi przeprowadza w tle korektę formatowania. Natychmiast po usunięciu zdania ze strony 1 wątek komunikacji z użytkownikiem informuje wątek formatujący o konieczności przeformatowania całej książki. Tymczasem wątek komunikacji z użytkownikiem kontynuuje nasłuchiwanie klawiatury i myszy i odpowiada na proste polecenia, takie jak przeglądanie strony 1. W tym samym czasie drugi z wątków w tle wykonuje intensywne obliczenia. Przy odrobinie szczęścia zmiana formatu zakończy się, zanim użytkownik poprosi o przejście na stronę 600. Jeśli tak się stanie, przejście na stronę 600 będzie mogło się odbyć bezzwłocznie.

Kiedy już jesteśmy przy edytorach, odpowiedzmy sobie na pytanie, dlaczego by nie dodać trzeciego wątku. Wiele edytorów tekstu jest wyposażonych w mechanizm automatycznego zapisywania całego pliku na dysk co kilka minut. Ma to zapobiec utracie całodniowej pracy w przypadku awarii programu, awarii systemu lub problemów z zasilaniem. Trzeci wątek może obsługiwać wykonywanie kopii zapasowych na dysku, nie przeszkadzając w działaniu pozostałym dwóm. Sytuację z trzema wątkami pokazano na rysunku 2.5.



Rysunek 2.5. Edytor tekstu składający się z trzech wątków

Gdyby program zawierał jeden wątek, to każde rozpoczęcie wykonywania kopii zapasowej na dysk powodowałoby, że polecenia z klawiatury i myszy byłyby ignorowane do czasu zakończenia wykonywania kopii zapasowej. Użytkownik z pewnością by to zauważył jako obniżoną wydajność. W innym rozwiązaniu zdarzenia związane z klawiaturą i myszą mogłyby przerwać wykonywanie kopii zapasowej na dysk, co pozwoliłoby na zachowanie dobrej wydajności, ale prowadziłoby do skomplikowanego modelu programowania opartego na przerwaniach. W przypadku zastosowania trzech wątków model programowania jest znacznie prostszy. Pierwszy wątek zajmuje się jedynie interakcjami z użytkownikiem. Drugi wątek przeformatowuje dokument, kiedy otrzyma takie zlecenie. Trzeci wątek okresowo zapisuje zawartość pamięci RAM na dysk.

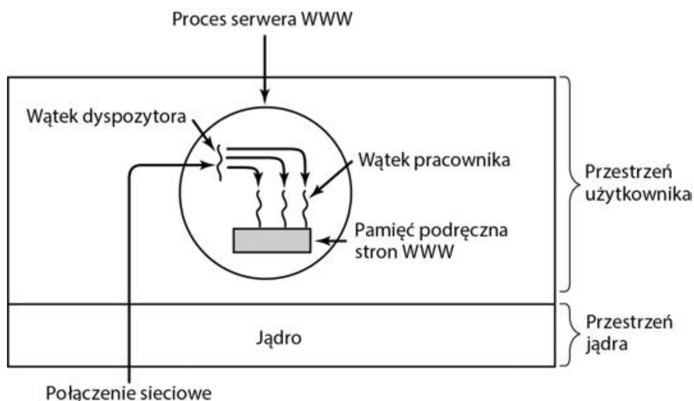
W tym przypadku powinno być jasne, że istnienie trzech oddzielnych procesów w tej sytuacji się nie sprawdzi, ponieważ wszystkie trzy wątki muszą operować na tym samym dokumencie. Dzięki występowaniu trzech wątków zamiast trzech procesów wątki współdzielą pamięć i w efekcie wszystkie mają dostęp do edytowanego dokumentu. Przy trzech procesach byłoby to niemożliwe.

Kup ksi k Pole ksi k

Analogiczna sytuacja występuje w przypadku wielu innych interaktywnych programów. I tak elektroniczny arkusz kalkulacyjny jest programem umożliwiającym użytkownikowi obsługę macierzy — niektóre z jej elementów są danymi wprowadzanymi przez użytkownika. Inne elementy są wyliczane na podstawie wprowadzonych danych i z wykorzystaniem potencjalnie skomplikowanych wzorów. Arkusz kalkulacyjny, który oblicza przewidywany roczny zysk dość dużej firmy, może mieć setki stron i tysiące złożonych formuł opartych na setkach zmiennych wejściowych. Zmiana wartości jednej zmiennej może powodować konieczność ponownego obliczenia wielu komórek. Dzięki zdefiniowaniu działającego w tle wątku zajmującego się przeliczaniem wątek interaktywny pozwala użytkownikowi na wprowadzanie zmian w czasie, gdy są wykonywane obliczenia. Na podobnej zasadzie trzeci wątek może samodzielnie obsługiwać kopie zapasowe wykonywane na dysku.

Rozważmy teraz jeszcze jeden przykład zastosowania wątków: serwer ośrodka WWW. Przychodzą żądania stron, a w odpowiedzi żądane strony są przesyłane do klienta. W większości witryn internetowych niektóre strony są częściej odwiedzane niż inne. Przykładowo główna strona serwisu Samsunga jest odwiedzana znacznie częściej niż strona w głębi drzewa zawierająca szczegółowe specyfikacje techniczne dowolnego modelu smartfona. Serwery WWW wykorzystują ten fakt do poprawy wydajności. Utrzymują kolekcję często używanych stron w pamięci głównej, aby wyeliminować potrzebę odwoływania się do dysku w celu ich pobrania. Taka kolekcja jest nazywana **pamięcią podręczną** (ang. *cache*) i wykorzystuje się ją również w wielu innych kontekstach; np. w rozdziale 1. zetknęliśmy się z pamięciami podręcznymi procesora.

Jeden ze sposobów organizacji serwera WWW pokazano na rysunku 2.6(a). W tym przypadku jeden z wątków — **dyspozytor** — odczytuje z sieci przychodzące żądania. Po przeanalizowaniu żądania wybiera bezczynny (tzn. zablokowany) **wątek pracownika** i przekazuje mu żądanie — np. poprzez zapisanie wskaźnika do komunikatu w specjalnym słowie powiązanym z każdym wątkiem. Następnie dyspozytor budzi uśpiony wątek pracownika — tzn. zmienia jego stan z „zablokowany" na „gotowy".



Rysunek 2.6. Serwer WWW z obsługą wielu wątków

Kiedy wątek się obudzi, sprawdza, czy jest w stanie spełnić żądanie z pamięci podręcznej strony WWW, do której mają dostęp wszystkie wątki. Jeśli tak nie jest, rozpoczyna operację odczytu w celu pobrania strony z dysku i przechodzi do stanu „zablokowany", trwającego do chwili zakończenia operacji dyskowej. Kiedy wątek zablokuje się na operacji dyskowej, inny wątek zaczyna działanie, np. dyspozytor, którego zadaniem jest przyjęcie jak największej liczby żądań, albo inny pracownik, który jest gotowy do działania.

Kup ksi k Pole ksi k

W tym modelu serwer może być zapisany w postaci kolekcji sekwencyjnych wątków. Program dyspozytora zawiera pętlę nieskończoną, w której jest pobierane żądanie pracy, później wręczane pracownikowi. Kod każdego pracownika zawiera pętlę nieskończoną, w której jest akceptowane żądanie od dyspozytora i następuje sprawdzenie, czy żądana strona jest dostępna w pamięci podręcznej serwera WWW. Jeśli tak, strona jest zwracana do klienta, a pracownik blokuje się w oczekiwaniu na nowe żądanie. Jeśli nie, pracownik pobiera stronę z dysku, zwraca ją do klienta i blokuje się w oczekiwaniu na nowe żądanie.

W uproszczonej formie kod przedstawiono na listingu 2.1. W tym przypadku, podobnie jak w pozostałej części tej książki, założono, że `TRUE` odpowiada stałej o wartości 1. Natomiast `buf` i `strona` są strukturami do przechowywania odpowiednio żądania pracy i strony WWW.

Listing 2.1. Uproszczona postać kodu dla struktury serwera z rysunku 2.6: (a) wątek dyspozytora, (b) wątek pracownika

(a)

```
while (TRUE){
    pobierz_nast_zadanie(&buf);
    przekaz_prace(&buf);
}
```

(b)

```
while (TRUE){
    czekaj_na_prace(&buf)
    szukaj_strony_w_pamieci_cache(&buf,
&strona);
    if ((&strona))
        czytaj_strone_z_dysku(&buf,
&strona);
    zwroc_strone(&strona);
}
```

Zastanówmy się, jak mógłby być napisany serwer WWW, gdyby nie było wątków. Jedna z możliwości polega na zaimplementowaniu go jako pojedynczego wątku. W głównej pętli serwera WWW następowałyby pobieranie żądania, jego analiza i realizacja. Dopiero potem serwer WWW mógłby pobrać następne żądanie. Podczas oczekiwania na zakończenie operacji dyskowej serwer byłby bezczynny i nie przetwarzałby żadnych innych przychodzących żądań. Jeśli serwer WWW działa na dedykowanej maszynie, tak jak to zwykle bywa, w czasie oczekiwania serwera WWW na dysk procesor pozostałby bezczynny. W efekcie końcowym można by było przetworzyć znacznie mniej żądań na sekundę. A zatem skorzystanie z wątków pozwala na uzyskanie znaczącego zysku wydajności, ale każdy z wątków jest programowany sekwencyjnie — w standardowy sposób. Alternatywą — podejściem opartym na zdarzeniach — zajmiemy się później.

Trzecim przykładem zastosowania wątków są aplikacje, które muszą przetwarzać duże ilości danych. Normalne podejście polega na przeczytaniu bloku danych, przetworzeniu go, a następnie ponownym zapisaniu. Problem w takim przypadku polega na tym, że jeśli dostępne są tylko blokujące wywołania systemowe, proces blokuje się, kiedy dane przychodzą oraz kiedy są wysyłane na zewnątrz. Doprowadzenie do sytuacji, w której procesor jest bezczynny w czasie, gdy jest wiele obliczeń do wykonania, to oczywiste marnotrawstwo i w miarę możliwości należy unikać takiej sytuacji.

Rozwiązaniem problemu jest wykorzystanie wątków. Wewnątrz procesu można wydzielić wątek wejściowy, wątek przetwarzania danych i wątek wyprowadzania danych. Wątek wejściowy czyta dane do bufora wejściowego. Wątek przetwarzania danych pobiera dane z bufora wejściowego, przetwarza je i umieszcza wyniki w buforze wyjściowym. Wątek wyprowadzania danych zapisuje wyniki z bufora wyjściowego na dysk. W ten sposób wprowadzanie danych, ich wyprowadzanie i przetwarzanie mogą być realizowane w tym samym czasie. Oczywiście model ten działa tylko wtedy, kiedy wywołanie systemowe blokuje wyłącznie wątek wywołujący, a nie cały proces.

Kup ksi k Pole ksi k

2.2.2. Klasyczny model wątków

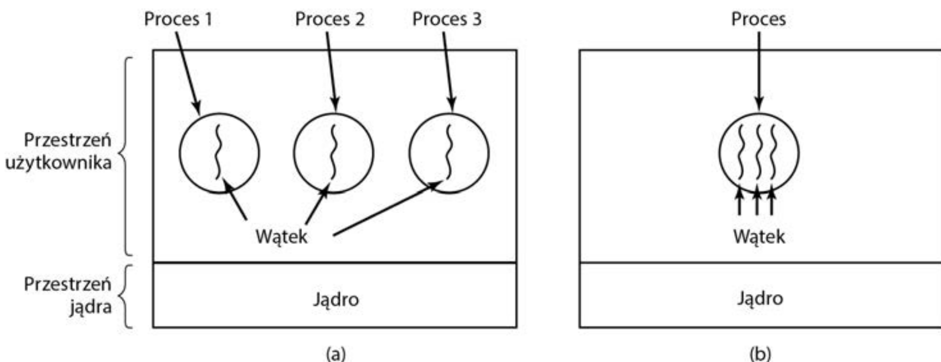
Teraz, kiedy pokazaliśmy, do czego mogą się przydać wątki i jak ich można używać, spróbujmy przeanalizować to zagadnienie nieco dokładniej. Model procesów bazuje na dwóch niezależnych pojęciach: grupowaniu zasobów i uruchamianiu. Czasami wygodnie jest je rozdzielić — wtedy można skorzystać z wątków. Najpierw przyjrzymy się klasycznemu modelowi wątków. Następnie omówimy model wątków Linuksa, w którym linia pomiędzy wątkami i procesami jest rozmyta.

Jednym ze sposobów patrzenia na proces jest postrzeganie go jako sposobu grupowania powiązanych ze sobą zasobów. Proces dysponuje przestrzenią adresową zawierającą tekst programu i dane, a także inne zasoby. Do zasobów tych można zaliczyć otwarte pliki, procesy dzieci, nieobsłużone alarmy, procedury obsługi sygnałów, informacje rozliczeniowe i wiele innych. Dzięki pogrupowaniu ich w formie procesu można nimi łatwiej zarządzać.

W innym pojęciu proces zawiera wykonywany wątek — zwykle w skrócie używa się samego pojęcia **wątku**. Wątek zawiera licznik programu, który śledzi to, jaka instrukcja będzie wykonywana w następnej kolejności. Posiada rejestry zawierające jego bieżące robocze zmienne. Ma do dyspozycji stos zawierający historię działania — po jednej ramce dla każdej procedury, której wykonywanie się rozpoczęło, ale jeszcze się nie zakończyło. Chociaż wątek musi realizować jakiś proces, wątek i jego proces są pojęciami odrębnymi i można je traktować osobno. Procesy są wykorzystywane do grupowania zasobów, wątki są podmiotami zaplanowanymi do wykonania przez procesor.

Wątki dodają do modelu procesu możliwość realizacji wielu wykonań w tym samym środowisku procesu (i tej samej przestrzeni adresowej), w dużym stopniu w sposób wzajemnie od siebie niezależny. Równoległe działanie wielu wątków w obrębie jednego procesu jest analogiczne do równoległego działania wielu procesów w jednym komputerze. W pierwszym z tych przypadków, wątki współdzielą przestrzeń adresową i inne zasoby. W drugim przypadku procesy współdzielą pamięć fizyczną, dyski, drukarki i inne zasoby. Ponieważ wątki mają pewne właściwości procesów, czasami nazywa się je **lekkimi procesami**. Do opisania sytuacji, w której w tym samym procesie może działać wiele wątków używa się także terminu **wielowątkowość**. Jak widzieliśmy w rozdziale 1., niektóre procesory mają bezpośrednią obsługę sprzętową wielowątkowości i pozwalają na przełączanie wątków w skali czasowej rzędu nanosekund.

Na rysunku 2.7(a) widać trzy tradycyjne procesy. Każdy proces ma swoją własną przestrzeń adresową oraz pojedynczy wątek sterowania. Dla odmiany w układzie z rysunku 2.7(b) widzimy jeden proces z trzema wątkami sterowania. Chociaż w obu przypadkach mamy trzy wątki, w sytuacji z rysunku 2.7(a) każdy z nich działa w innej przestrzeni adresowej, podczas gdy w sytuacji z rysunku 2.7(b) wszystkie współdzielą tę samą przestrzeń adresową.



Rysunek 2.7. (a) Trzy procesy, z których każdy posiada jeden wątek; (b) jeden proces z trzema wątkami

Kiedy wielowątkowy proces działa w jednoprocesorowym systemie, wątki działają po kolei. Na rysunku 2.1 widzieliśmy, jak działa wieloprogramowość procesów. Dzięki przełączaniu pomiędzy wieloma procesami system daje iluzję oddzielnych procesów sekwencyjnych działających współbieżnie. Wielowątkowość działa w taki sam sposób. Procesor przełącza się w szybkim tempie pomiędzy wątkami, dając iluzję, że wątki działają współbieżnie — chociaż na wolniejszym procesorze od fizycznego. Przy trzech wątkach obliczeniowych w procesie wątki będą sprawiały wrażenie równoległego działania, ale tak, jakby każdy z nich działał na procesorze o szybkości równej jednej trzeciej szybkości fizycznego procesora.

Różne wątki procesu nie są tak niezależne, jak różne procesy. Wszystkie wątki posługują się dokładnie tą samą przestrzenią adresową, co również oznacza, że współdzielą one te same zmienne globalne. Ponieważ każdy wątek może uzyskać dostęp do każdego adresu pamięci w obrębie przestrzeni adresowej procesu, jeden wątek może odczytać, zapisać, a nawet wyczyścić stos innego wątku. Pomiędzy wątkami nie ma zabezpieczeń, ponieważ (1) byłyby one niemożliwe do realizacji, a (2) nie powinny być potrzebne. W odróżnieniu od różnych procesów, które potencjalnie należą do różnych użytkowników i które mogą być dla siebie wrogie, proces zawsze należy do jednego użytkownika, który przypuszczalnie utworzył wiele wątków, a zatem powinny one współpracować, a nie walczyć ze sobą. Oprócz przestrzeni adresowej wszystkie wątki mogą współdzielić ten sam zbiór otwartych plików, procesów dzieci, alarmów, sygnałów itp., tak jak pokazano w tabeli 2.3. Tak więc organizacja pokazana na rysunku 2.7(a) mogłaby zostać użyta, jeśli trzy procesy są ze sobą niezwiązane, natomiast organizacja z rysunku 2.7(b) byłaby właściwa w przypadku, gdyby trzy wątki były częścią tego samego zadania i gdyby aktywnie i ściśle ze sobą współpracowały.

Tabela 2.3. W pierwszej kolumnie wyszczególniono cechy wspólne dla wszystkich wątków w procesie. W drugiej kolumnie zamieszczone niektóre elementy prywatne dla każdego wątku

Komponenty procesu	**Komponenty wątku**
Przestrzeń adresowa	Licznik programu
Zmienne globalne	Rejestry
Otwarte pliki	Stos
Procesy dzieci	Stan
Zaległe alarmy	
Sygnały i procedury obsługi sygnałów	
Informacje dotyczące statystyk	

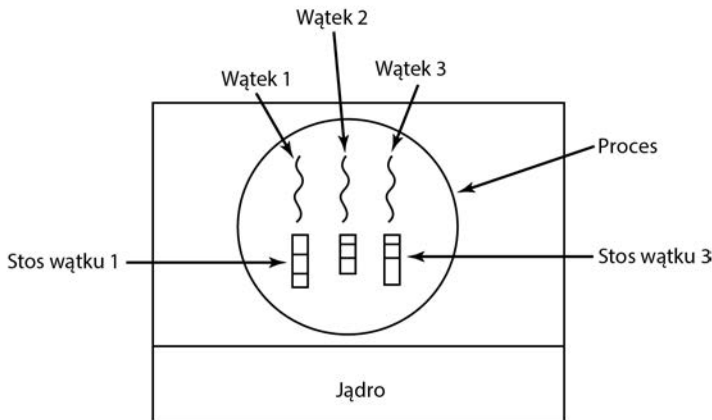
Elementy w pierwszej kolumnie są właściwościami procesu, a nie wątku. Jeśli np. jeden wątek otworzy plik, będzie on widoczny dla innych wątków w procesie. Wątki te będą mogły czytać dane z pliku i je zapisywać. To logiczne, ponieważ właśnie proces, a nie wątek jest jednostką zarządzania zasobami. Gdyby każdy wątek miał własną przestrzeń adresową, otwarte pliki, nieobsłużone alarmy itd., byłby osobnym procesem. Wykorzystując pojęcie wątków, chcemy, aby wiele wątków mogło współdzielić zbiór zasobów. Dzięki temu mogą one ze sobą ściśle współpracować w celu wykonania określonego zadania.

Podobnie jak tradycyjny proces (czyli taki, który zawiera tylko jeden wątek), wątek może znajdować się w jednym z kilku stanów: „działający", „zablokowany", „gotowy" lub „zakończony". Działający wątek posiada dostęp do procesora i jest aktywny. Zablokowany wątek oczekuje na jakieś zdarzenie, by mógł się odblokować. Kiedy np. wątek realizuje wywołanie systemowe odczytujące dane z klawiatury, jest zablokowany do czasu, kiedy użytkownik wpisze dane wejściowe. Wątek może się blokować w oczekiwaniu na wystąpienie zdarzenia zewnętrznego lub może

Kup ksi k Pole ksi k

oczekiwać, aż odblokuje go inny wątek. Wątek gotowy jest zaplanowany do uruchomienia i zostanie uruchomiony, kiedy nadejdzie jego kolej. Przejścia pomiędzy stanami wątków są identyczne jak przejścia pomiędzy stanami procesów. Zilustrowano je na rysunku 2.2.

Istotne znaczenie ma zdanie sobie sprawy, że każdy wątek posiada własny stos, co zilustrowano na rysunku 2.8. Stos każdego wątku zawiera po jednej ramce dla każdej procedury, która została wywołana, a z której jeszcze nie nastąpił powrót. Ramka ta zawiera zmienne lokalne procedury oraz adres powrotu, który będzie wykorzystany po zakończeniu obsługi wywołania procedury. Jeśli np. procedura X wywoła procedurę Y, a procedura Y wywoła procedurę Z, to w czasie, kiedy działa procedura Z, na stosie będą ramki dla procedur X, Y i Z. Każdy wątek, ogólnie rzecz biorąc, będzie wywoływał inne procedury, a zatem będzie miał inną historię wywołań. Dlatego właśnie każdy wątek potrzebuje własnego stosu.



Rysunek 2.8. Każdy wątek ma własny stos

W przypadku gdy system obsługuje wielowątkowość, procesy zazwyczaj rozpoczynają działanie z jednym wątkiem. Wątek ten posiada zdolność do tworzenia nowych wątków za pomocą wywołania procedury, np. `thread_create`. Parametr procedury `thread_create` zwykle określa nazwę procedury, która ma się uruchomić dla nowego wątku. Nie jest konieczne (ani nawet możliwe) ustalenie czegokolwiek na temat przestrzeni adresowej nowego wątku, ponieważ wątek automatycznie działa w przestrzeni adresowej wątku tworzącego. Czasami wątki są hierarchiczne i zachodzą pomiędzy nimi relacje rodzic-dziecko, często jednak takie relacje nie występują, a wszystkie wątki są sobie równe. Niezależnie od tego, czy pomiędzy wątkami zachodzi relacja hierarchii, wątek tworzący zwykle zwraca identyfikator wątku zawierający nazwę nowego wątku.

Kiedy wątek zakończy swoją pracę, może zakończyć działanie poprzez wywołanie procedury bibliotecznej, np. `thread_exit`. W tym momencie wątek znika i nie może być więcej zarządzany. W niektórych systemach obsługi wątków jeden wątek może czekać na zakończenie innego wątku poprzez wywołanie procedury, np. `thread_join`. Procedura ta blokuje wątek wywołujący do czasu zakończenia (specyficznego) wątku. Pod tym względem tworzenie i kończenie wątków przypomina tworzenie i kończenie procesów i wymaga w przybliżeniu tych samych opcji.

Innym popularnym wywołaniem dotyczącym wątków jest `thread_yield`. Umożliwia ono wątkowi dobrowolną rezygnację z procesora w celu umożliwienia działania innemu wątkowi. Takie wywołanie ma istotne znaczenie, ponieważ nie istnieje przerwanie zegara, które wymuszałoby wieloprogramowość, tak jak w przypadku procesów. W związku z tym istotne znaczenie ma to, aby wątki były „uprzejme" i od czasu do czasu dobrowolnie rezygnowały z procesora, tak by inne

Kup ksi k Pole ksi k

wątki miały szanse na działanie. Są również inne wywołania — np. pozwalające na to, aby jeden wątek poczekał, aż następny zakończy jakąś pracę, lub by ogłosił, że właśnie zakończył jakąś pracę itd.

Chociaż wątki często się przydają, wprowadzają także szereg komplikacji do modelu programowania. Na początek przeanalizujmy efekty na uniksowe wywołanie systemowej `fork`. Jeśli proces rodzic ma wiele wątków, to czy proces dziecko również powinien je mieć? Jeśli nie, to proces może nie działać prawidłowo, ponieważ wszystkie wątki mogą mieć istotne znaczenie. Tymczasem gdy proces dziecko otrzyma tyle samo wątków co rodzic, to co się stanie, jeśli wątek należący do rodzica zostanie zablokowany przez wywołanie `read`, powiedzmy, z klawiatury? Czy teraz dwa wątki są zablokowane przez klawiaturę — jeden w procesie rodzicu i drugi w dziecku? Kiedy użytkownik wpisze wiersz, to czy kopia pojawi się w obu wątkach? A może tylko w wątku rodzica? Lub tylko w wątku dziecka? Ten sam problem występuje dla otwartych połączeń sieciowych. Projektanci systemu operacyjnego muszą dokonywać czytelnych wyborów i dokładnie definiować semantykę, tak aby użytkownicy właściwie rozumieli zachowanie wątków.

Przyjrzymy się *niektórym* z tych problemów i pokażemy, że stosowane rozwiązania są często pragmatyczne. Przykładowo w takim systemie jak Linux rozwidlenie wielowątkowego procesu spowoduje utworzenie tylko jednego wątku w procesie potomnym. Jeśli jednak program używa wątków POSIX, do zarejestrowania procedur obsługi rozwidlenia (procedur, które są wywoływane w odpowiedzi na wywołanie `fork`) może skorzystać z wywołania `pthread_atfork()`, więc może uruchamiać dodatkowe wątki i robić wszystko, co jest potrzebne do poprawnego działania rozwidlonego procesu. Mimo to należy zauważyć, że wiele ze wspomnianych problemów to wybory projektowe, a w różnych systemach mogą być wybierane różne rozwiązania. Na razie należy zapamiętać, że związek między wątkami a rozwidleniami może być dość złożony.

Inna klasa problemów wiąże się z faktem współdzielenia przez wątki wielu struktur danych. Co się dzieje, jeśli jeden wątek zamyka plik, podczas gdy inny ciągle z niego czyta? Przypuśćmy, że jeden z wątków zauważa, że jest za mało pamięci, i rozpoczyna alokowanie większej ilości pamięci. W trakcie tego działania następuje przełączenie wątku. Nowy wątek również zauważa, że jest za mało pamięci i także rozpoczyna alokowanie dodatkowej pamięci. Pamięć prawdopodobnie będzie alokowana dwukrotnie. Przy odrobinie wysiłku można rozwiązać te problemy, jednak poprawna praca programów wykorzystujących wielowątkowość wymaga dokładnych przemyśleń i dokładnego projektowania.

2.2.3. Wątki POSIX

Aby było możliwe napisanie przenośnego programu z obsługą wielu wątków, organizacja IEEE zdefiniowała standard 1003.1c. Pakiet obsługi wątków, który tam zdefiniowano, nosi nazwę `Pthreads`. Jest on obsługiwany przez większość systemów uniksowych. W standardzie zdefiniowano ponad 60 wywołań funkcji. To o wiele za dużo, by można je było dokładnie omówić w tej książce. Omówimy zatem kilka najważniejszych. Dzięki temu Czytelnik uzyska obraz ich działania. Wywołania, które opiszemy, zostały wyszczególnione w tabeli 2.4.

Wszystkie wątki pakietu `Pthreads` mają określone właściwości. Każdy z nich posiada identyfikator, zbiór rejestrów (łącznie z licznikiem programu) oraz zbiór atrybutów zapisanych w pewnej strukturze. Do atrybutów tych należy rozmiar stosu, parametry szeregowania oraz inne elementy potrzebne do korzystania z wątku.

Kup ksi k Pole ksi k

Tabela 2.4. Niektóre wywołania funkcji należące do pakietu Pthreads

Wywołanie obsługi wątku	Opis
`pthread_create`	Utworzenie nowego wątku
`pthread_exit`	Zakończenie wątku wywołującego
`pthread_join`	Oczekiwanie na zakończenie specyficznego wątku
`pthread_yield`	Zwolnienie procesora w celu umożliwienia działania innemu wątkowi
`pthread_attr_init`	Utworzenie i zainicjowanie struktury atrybutów wątku
`pthread_attr_destroy`	Usunięcie struktury atrybutów wątku

Nowy wątek tworzy się za pomocą wywołania `pthread_create`. Jako wartość funkcji zwracany jest identyfikator nowo utworzonego wątku. Wywołanie to nieprzypadkowo przypomina wywołanie systemowe `fork` (z wyjątkiem parametrów). W tym przypadku identyfikator wątku spełnia rolę identyfikatora PID, głównie do celów identyfikacji wątków w innych wywołaniach. Kiedy wątek zakończy pracę, która została do niego przydzielona, może zakończyć swoje działanie poprzez wywołanie funkcji `pthread_exit`. Wywołanie to zatrzymuje wątek i zwalnia jego stos.

Często wątek musi czekać, aż inny wątek zakończy swoją pracę. Dopiero później może kontynuować działanie. Wątek oczekujący na zakończenie specyficznego innego wątku wywołuje funkcję `pthread_join`. Identyfikator wątku, który ma się zakończyć, jest przekazywany jako parametr.

Czasami się zdarza, że wątek nie jest logicznie zablokowany, ale czuje, że działa już dość długo, i chce dać innemu wątkowi szansę działania. Cel ten można osiągnąć za pomocą wywołania `pthread_yield`. Nie ma takiego wywołania w wypadku procesów, ponieważ zakłada się, że procesy ze sobą rywalizują i każdy z nich chce uzyskać maksymalnie dużo czasu procesora (chociaż publiczny proces, aby na krótko zwolnić procesor, może wywołać procedurę `sleep`). Ponieważ jednak wątki procesu współdziałają ze sobą, a ich kod jest pisany przez tego samego programistę, czasami programista chce, aby każdy z wątków otrzymał swoją szansę.

Następne dwa wywołania obsługi wątków dotyczą atrybutów wątku. Wywołanie `Pthread_attr_init` tworzy strukturę atrybutów powiązaną z wątkiem i inicjuje ją do wartości domyślnych. Wartości te (takie jak priorytet) można zmieniać poprzez modyfikowanie pól w strukturze atrybutów.

Na koniec — wywołanie `pthread_attr_destroy` usuwa strukturę atrybutów wątku i zwalnia pamięć. Wywołanie to nie ma wpływu na wątki korzystające z atrybutów. Wątki te w dalszym ciągu istnieją.

Aby uzyskać lepszy obraz tego, jak działa pakiet Pthreads, rozważmy prosty przykład z listingu 2.2. Główny program wykonuje się w pętli `NUMBER_OF_THREADS` razy. W każdej iteracji program wyświetla komunikat i tworzy nowy wątek. Jeśli tworzenie wątku nie powiedzie się, program wyświetla komunikat o błędzie i kończy działanie. Po utworzeniu wszystkich wątków program główny kończy działanie.

Listing 2.2. Przykładowy program wykorzystujący wątki

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER OF THREADS 10

void *pr int hello world(void *tid)
{
```

Kup ksi k Pole ksi k

```
/* Funkcja wyświetla identyfikator wątku i kończy działanie */ */
    printf("Witaj, Świecie. Pozdrowienia od wątku %d\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* Program główny tworzy 10 wątków i kończy działanie */ */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER OF THREADS; i++)
    {
        printf("Tu program główny. Tworzenie wątku %d\n", i);
        status = pthread_create(&threads[i], NULL, print hello world, (void *)i);

        if (status != 0) {
            printf("Oops. Funkcja pthread_create zwróciła kod błędu %d\n, status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

Podczas tworzenia wątek wyświetla jednowierszowy komunikat, w którym się przedstawia, a następnie kończy działanie. Kolejność, w jakiej będą się pojawiały poszczególne komunikaty, nie jest określona i może być różna w kolejnych uruchomieniach programu.

Pakiet `Pthreads` w żadnym razie nie ogranicza się do funkcji opisanych powyżej. Jest ich znacznie więcej. Niektóre z kolejnych wywołań opiszemy później, po omówieniu zagadnienia synchronizacji procesów i wątków.

2.2.4. Implementacja wątków w przestrzeni użytkownika

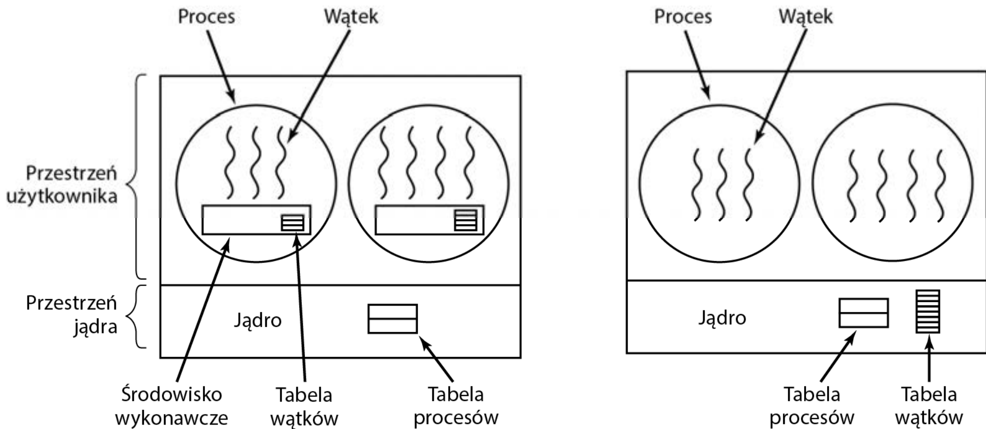
Wątki można implementować w dwóch różnych miejscach: w przestrzeni użytkownika i w jądrze. Podział ten jest dość płynny. Możliwe są również implementacje hybrydowe. Poniżej opiszemy obie metody razem z ich zaletami i wadami.

Pierwsza metoda polega na umieszczeniu pakietu wątków w całości w przestrzeni użytkownika. Jądro nic o nich nie wie. Z jego punktu widzenia procesy, którymi zarządza, są standardowe — jednowątkowe. Pierwsza i najbardziej oczywista zaleta tego rozwiązania polega na tym, że pakiet obsługi wątków na poziomie przestrzeni użytkownika można zaimplementować w systemie operacyjnym, który nie obsługuje wątków. Do tej kategorii w przeszłości należały wszystkie systemy operacyjne i nawet dziś niektóre do niej należą. Przy takim podejściu wątki są implementowane za pomocą biblioteki.

Wszystkie tego rodzaju implementacje mają taką samą ogólną strukturę, co zilustrowano na rysunku 2.9(a). Wątki działają na bazie środowiska wykonawczego — kolekcji procedur, które nimi zarządzają. Dotąd wspominaliśmy o czterech z nich: `pthread_create`, `pthread_exit`, `pthread_join` i `pthread_yield`, ale zwykle jest ich więcej.

Jeśli wątki są zarządzane w przestrzeni użytkownika, każdy proces potrzebuje swojej prywatnej tabeli wątków, która ma na celu śledzenie wątków w tym procesie. Tabela ta jest analogiczna do tabeli procesów w jądrze. Różnica polega na tym, że śledzi ona właściwości tylko na poziomie wątku — np. licznik programu każdego z wątków, wskaźnik stosu, rejestry, stan itp. Tabela wątków jest zarządzana przez środowisko wykonawcze. Kiedy wątek przechodzi do stanu gotowości lub zablokowania, informacje potrzebne do jego wznowienia są zapisywane w tabeli wątków, dokładnie w taki sam sposób, w jaki jądro zapisuje informacje o procesach w tabeli procesów.

Kup ksi k Pole ksi k



Rysunek 2.9. (a) Pakiet obsługi wątków na poziomie użytkownika. (b) pakiet obsługi wątków zarządzany przez jądro

Kiedy wątek wykona operację, która może spowodować jego lokalne zablokowanie, np. oczekuje, aż inny wątek w tym samym procesie wykona jakąś pracę, wykonuje procedurę ze środowiska wykonawczego. Procedura ta sprawdza, czy wątek musi być przełączony do stanu zablokowania. Jeśli tak, to zapisuje rejestry wątku (tzn. własne) w tabeli wątków, szuka w tabeli wątku gotowego do działania i ładuje rejestry maszyny zapisanymi wartościami odpowiadającymi nowemu wątkowi. Po przełączeniu wskaźnika stosu i licznika programu nowy wątek automatycznie powraca do życia.

Jeśli maszyna posiada instrukcję zapisującą wszystkie rejestry oraz inną instrukcję, która je wszystkie ładuje, przełączenie wątku można przeprowadzić za pomocą zaledwie kilku instrukcji. Przeprowadzenie przełączania wątków w taki sposób jest co najmniej o jeden rząd wielkości szybsze od wykonywania rozkazu pułapki do jądra. To silny argument przemawiający za implementacją pakietu zarządzania wątkami na poziomie przestrzeni użytkownika.

Dodatkowo, kiedy wątek zakończy na chwilę działanie, np. gdy wywoła funkcję `thread_yield`, kod funkcji `thread_yield` może zapisać informacje dotyczące wątku w samej tabeli wątków. Następnie wywołuje program szeregowania wątków, który ma wybrać inny wątek do uruchomienia. Procedura zapisująca stan wątku oraz program szeregujący są po prostu lokalnymi procedurami, zatem wywołanie ich jest znacznie bardziej wydajne od wykonania wywołania jądra; m.in. nie jest potrzebny rozkaz pułapki, nie trzeba przełączać kontekstu, nie trzeba opróżniać pamięci podręcznej. W związku z tym zarządzanie wątkami odbywa się bardzo szybko.

Implementacja wątków na poziomie przestrzeni użytkownika ma także inne zalety. Dzięki temu każdemu procesowi można przypisać własny, spersonalizowany algorytm szeregowania. W przypadku niektórych aplikacji, np. zawierających wątek mechanizmu odśmiecania, brak konieczności przejmowania się możliwością zatrzymania się wątku w nieodpowiednim momencie jest zaletą. Takie rozwiązanie okazuje się również łatwiejsze do skalowania, ponieważ wątki zarządzane na poziomie jądra niewątpliwie wymagają przestrzeni na tabelę i stos w jądrze, a to, w przypadku dużej liczby wątków, może być problemem.

Pomimo lepszej wydajności implementacja wątków na poziomie przestrzeni użytkownika ma również istotne wady. Pierwsza z nich dotyczy sposobu implementacji blokujących wywołań systemowych. Przypuśćmy, że wątek czyta z klawiatury, zanim zostanie wciśnięty jakikolwiek klawisz. Zezwolenie wątkowi na wykonanie wywołania systemowego jest niedopuszczalne, ponieważ spowoduje to zatrzymanie wszystkich wątków. Trzeba pamiętać, że jednym z podstawowych

Kup ksi k Pole ksi k

celów korzystania z wątków jest umożliwienie wszystkim wątkom używania wywołań blokujących, a przy tym niedopuszczenie do tego, by zablokowany wątek miał wpływ na inne. W przypadku blokujących wywołań systemowych trudno znaleźć łatwe rozwiązanie pozwalające na spełnienie tego celu.

Wszystkie wywołania systemowe można zmienić na nieblokujące (np. odczyt z klawiatury zwróciłby 0 bajtów, gdyby znaki nie były wcześniej zbuforowane), ale wymaganie zmian w systemie operacyjnym jest nieatrakcyjne. Poza tym jednym z argumentów przemawiających za obsługą wątków na poziomie użytkownika była możliwość wykorzystania takiego mechanizmu w *istniejących* systemach operacyjnych. Co więcej, zmiana semantyki wywołania `read` wymagałaby modyfikacji wielu programów użytkowych.

Jedną z możliwych alternatyw można zastosować w przypadku, gdy można z góry powiedzieć, czy wywołanie jest blokujące. W niektórych wersjach Uniksa istnieje wywołanie systemowe `select`, które pozwala procesowi wywołującemu na sprawdzenie, czy wywołanie `read` będzie blokujące. Jeśli jest dostępne to wywołanie, można zastąpić procedurę biblioteczną `read` nową wersją, która najpierw wykonuje wywołanie `select`, a następnie wywołuje `read` tylko wtedy, gdy jest to bezpieczne (tzn. nie spowoduje zablokowania). Jeżeli wywołanie `read` ma doprowadzić do zablokowania, nie jest wykonywane. Zamiast wywołania `read` uruchamiany jest inny wątek. Następnym razem, kiedy środowisko wykonawcze otrzyma sterowanie, może sprawdzić ponownie, czy wykonanie wywołania `read` jest bezpieczne. Takie podejście wymaga zmiany implementacji części biblioteki wywołań systemowych, jest niewydajne i nieeleganckie, ale możliwości wyboru są ograniczone. Kod wokół wywołania systemowego, który wykonuje test, określa się **osłoną** lub **opakowaniem** — ang. *wrapper* (wkrótce opowiemy o jeszcze wydajniejszych mechanizmach asynchronicznego wejścia-wyjścia, np. `epoll` w Linuksie lub `kqueue` w FreeBSD).

W pewnym sensie podobnym problemem do blokujących wywołań systemowych jest problem braku stron w pamięci (ang. *page faults*). Zagadnienie to omówimy w rozdziale 3. Na razie wystarczy, jeśli powiemy, że komputery można skonfigurować w taki sposób, aby w danym momencie w głównej pamięci znajdowała się tylko część programu. Jeżeli program wywoła instrukcję, której nie ma w pamięci lub skoczy do takiej instrukcji, wystąpi warunek braku strony. Wtedy system operacyjny jest zmuszony do pobrania brakującej instrukcji (wraz z jej sąsiadami) z dysku. Na tym właśnie polega warunek braku strony. Podczas gdy potrzebna instrukcja jest wyszukiwana i wczytywana, proces pozostaje zablokowany. Jeśli wątek spowoduje warunek braku strony, jądro, które nawet nie wie o istnieniu wątków, blokuje cały proces do czasu zakończenia dyskowej operacji wejścia-wyjścia. Robi to, mimo że nie ma przeszkód, by inne wątki działały.

Inny problem z pakietami obsługi wątków na poziomie użytkownika polega na tym, że jeśli wątek zacznie działać, to żaden inny wątek w tym procesie nigdy nie zacznie działać, o ile pierwszy wątek dobrowolnie nie zrezygnuje z procesora. W obrębie pojedynczego procesu nie ma przerwań zegara, dlatego nie ma możliwości szeregowania procesów w trybie cyklicznym (tzn. po kolei). Jeśli wątek z własnej woli nie przekaże sterowania do środowiska wykonawczego, program szeregujący nigdy nie będzie miał szansy działania.

Jednym z możliwych rozwiązań problemu wątków działających bez przerwy jest zlecenie środowisku wykonawczemu żądania sygnału zegara (przerwania) co sekundę w celu przekazania mu kontroli. Takie rozwiązanie okazuje się jednak toporne i trudne do zaprogramowania. Okresowe przerwania zegara z wyższą częstotliwością nie zawsze są możliwe, a nawet jeśli tak jest, koszt obliczeniowy takiej operacji może być wysoki. Co więcej, wątek również może potrzebować przerwania zegara, co przeszkadza wykorzystaniu zegara przez środowisko wykonawcze.

Innym, rzeczywiście druzgocącym argumentem przeciwko wątkom zarządzanym na poziomie przestrzeni użytkownika jest to, że programiści, ogólnie rzecz biorąc, potrzebują wątków

Kup książkę Poleć książkę

w aplikacjach — np. w wielowątkowym serwerze WWW — w których wątki blokują się często. Wątki te bezustannie wykonują wywołania systemowe. Kiedy zostanie wykonany rozkaz pułapki do jądra w celu realizacji wywołania systemowego, jądro nie ma nic więcej do roboty przy przełączaniu wątków, w przypadku gdy stary wątek się zablokował, a zlecenie jądru wykonania tej czynności eliminuje potrzebę ciągłego wykonywania wywołań systemowych `select` sprawdzających, czy wywołania systemowe `read` są bezpieczne. Jaki jest sens istnienia wątków w aplikacjach całkowicie powiązanych z procesorem, które rzadko się blokują? Nikt nie jest w stanie zaproponować sensownego rozwiązania problemu wyliczania liczb pierwszych lub grania w szachy z wykorzystaniem wątków, ponieważ realizacja tych programów w ten sposób nie przynosi istotnych korzyści.

2.2.5. Implementacja wątków w jądrze

Rozważmy teraz sytuację, w której jądro wie o istnieniu wątków i to ono nimi zarządza. Środowisko wykonawcze w każdym z procesów nie jest wymagane, co pokazano na rysunku 2.9(b). Zamiast tego jądro dysponuje tabelą wątków, która śledzi wszystkie wątki w systemie. Kiedy wątek chce utworzyć nowy wątek lub zniszczyć istniejący, wykonuje wywołanie systemowe, które następnie realizuje utworzenie lub zniszczenie wątku poprzez aktualizację tabeli wątków na poziomie jądra.

W tabeli wątków w jądrze są zapisane rejestry, stan oraz inne informacje dla każdego wątku. Informacje są takie same, jak w przypadku wątków zarządzanych na poziomie użytkownika, z tą różnicą, że są one umieszczone w jądrze, a nie w przestrzeni użytkownika (wewnątrz środowiska wykonawczego). Informacje te stanowią podzbiór tradycyjnie utrzymywanych przez jądro informacji na temat jednowątkowych procesów — czyli stanu procesów. Oprócz tego jądro utrzymuje również tradycyjną tabelę procesów, która służy do śledzenia procesów.

Wszystkie wywołania, które mogą zablokować wątek, są implementowane jako wywołania systemowe znacząco większym kosztem niż wywołanie procedury środowiska wykonawczego. Kiedy wątek się zablokuje, jądro może uruchomić wątek z tego samego procesu (jeśli jakiś jest gotowy) lub wątek z innego procesu. W przypadku wątków zarządzanych na poziomie przestrzeni użytkownika środowisko wykonawcze uruchamia wątki z własnego procesu do czasu, aż jądro zabierze mu procesor (lub nie będzie wątków gotowych do działania).

Ze względu na relatywnie większy koszt tworzenia i niszczenia wątków na poziomie jądra niektóre systemy przyjmują rozwiązanie „ekologiczne” i ponownie wykorzystują swoje wątki. W momencie niszczenia wątku jest on oznaczany jako niemożliwy do uruchomienia, ale poza tym struktury danych jądra pozostają bez zmian. Kiedy później trzeba utworzyć nowy wątek, stary wątek jest reaktywowany, co eliminuje konieczność wykonywania pewnych obliczeń. Recykling wątków jest również możliwy w przypadku wątków zarządzanych w przestrzeni użytkownika, ale ponieważ koszty zarządzania wątkami są znacznie niższe, motywacja do korzystania z tego mechanizmu jest mniejsza.

Wątki jądra nie wymagają żadnych nowych nieblokujących wywołań systemowych. Co więcej, jeśli jeden z wątków w procesie spowoduje warunek braku strony, jądro może łatwo sprawdzić, czy proces zawiera inne wątki możliwe do uruchomienia. Jeśli tak, uruchamia jeden z nich w oczekiwaniu na przesłanie z dysku wymaganej strony. Główną wadą tego rozwiązania jest fakt, że koszty wywołania systemowego są znaczące. W związku z tym, w przypadku dużej liczby operacji zarządzania wątkami (tworzenia, niszczenia itp.), ponoszone koszty obliczeniowe okazują się wysokie.

O ile wykorzystanie zarządzania wątkami na poziomie jądra rozwiązuje niektóre problemy, o tyle nie rozwiązuje ich wszystkich. Co się np. stanie, jeśli wielowątkowy proces wykona wywołanie

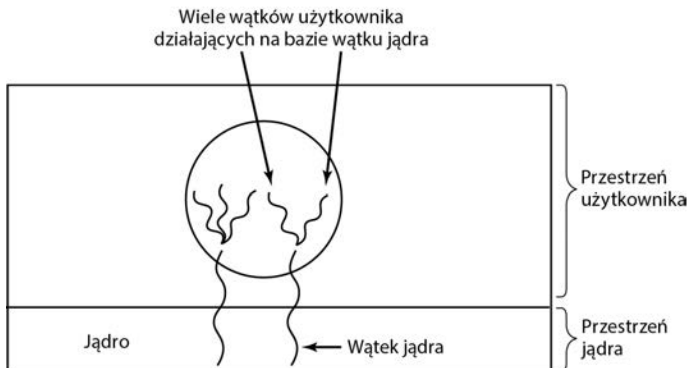
Kup ksi k Pole ksi k

`fork`? Czy nowy proces będzie miał tyle wątków, ile ma stary, czy tylko jeden? W wielu przypadkach najlepszy wybór zależy od tego, do czego będzie służył nowy proces. Jeśli zamierza skorzystać z wywołania `exec` w celu uruchomienia nowego programu, prawdopodobnie właściwe będzie stworzenie procesu z jednym wątkiem, jeśli jednak ma on kontynuować działanie, reprodukcja wszystkich wątków wydaje się właściwsza.

Innym problemem związanym z wątkami są sygnały. Jak pamiętamy, sygnały są przesyłane do procesów, a nie do wątków (przynajmniej w modelu klasycznym). Który wątek ma obsłużyć nadchodzący sygnał? Można wyobrazić sobie rozwiązanie, w którym wątki rejestrują swoje zainteresowanie określonymi sygnałami. Dzięki temu w przypadku nadejścia sygnału mógłby on być skierowany do wątku, który na ten sygnał oczekuje. W Linuksie np. sygnał może być obsłużony przez dowolny wątek, a szczęśliwego zwycięzcę wybiera system operacyjny. Można jednak po prostu zablokować sygnał we wszystkich wątkach z wyjątkiem jednego. Jeśli dwa lub większa liczba wątków zarejestruje swoje zainteresowanie tym samym sygnałem, system operacyjny wybiera wątek (np. losowo) i pozwala mu obsłużyć sygnał. To tylko dwa problemy, jakie stwarzają wątki. Jest ich jednak więcej. Jeśli programista nie zachowa należytej ostrożności, może łatwo popełnić błąd.

2.2.6. Implementacje hybrydowe

Próbowano różnych rozwiązań mających na celu połączenie zalet zarządzania wątkami na poziomie użytkownika oraz zarządzania nimi na poziomie jądra. Jednym ze sposobów jest użycie wątków na poziomie jądra, a następnie zwielokrotnienie niektórych lub wszystkich wątków jądra na wątki na poziomie użytkownika. Sposób ten pokazano na rysunku 2.10. W przypadku skorzystania z takiego podejścia programista może określić, ile wątków jądra chce wykorzystać oraz na ile wątków poziomu użytkownika ma być zwielokrotniony każdy z nich. Taki model daje największą elastyczność.



Rysunek 2.10. Zwielokrotnianie wątków użytkownika na bazie wątków jądra

Przy tym podejściu jądro jest świadome istnienia *wyłącznie* wątków poziomu jądra i tylko nimi zarządza. Niektóre spośród tych wątków mogą zawierać wiele wątków poziomu użytkownika, stworzonych na bazie wątków jądra. Wątki poziomu użytkownika są tworzone, niszczone i zarządzane identycznie, jak wątki na poziomie użytkownika działające w systemie operacyjnym bez obsługi wielowątkowości. W tym modelu każdy wątek poziomu jądra posiada pewien zbiór wątków na poziomie użytkownika. Wątki poziomu użytkownika po kolei korzystają z wątku poziomu jądra.

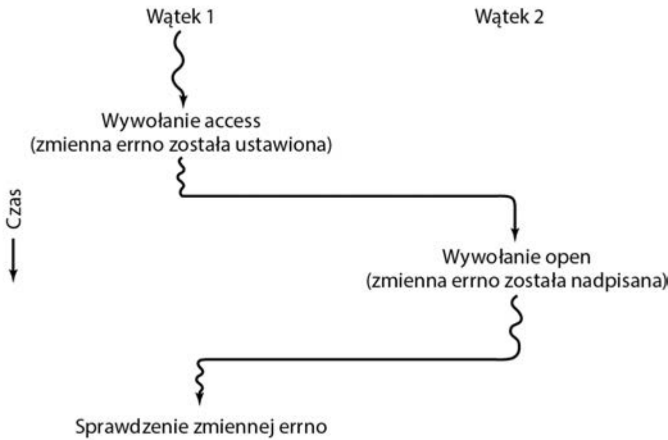
Kup książkę Poleć książkę

2.2.7. Przystosowywanie kodu jednowątkowego do obsługi wielu wątków

Dla procesów jednowątkowych napisano wiele programów. Ich konwersja na postać wielowątkową jest znacznie trudniejsza, niż mogłoby się wydawać na pierwszy rzut oka. Poniżej zaprezentujemy kilka problemów, które mogą wystąpić podczas takiej konwersji.

Na początek należy sobie uświadomić, że wątek, tak jak proces, zwykle składa się z wielu procedur. Mogą one mieć zmienne lokalne, zmienne globalne i parametry. Zmienne lokalne i parametry nie powodują żadnych problemów, tymczasem zmienne, które są globalne dla wątku, ale nie są globalne dla całego programu, sprawiają problem. Są to zmienne, które są globalne w tym sensie, że używa ich wiele procedur w obrębie wątku (ponieważ mogą one wykorzystywać dowolne zmienne globalne), ale inne wątki nie powinny z nich korzystać.

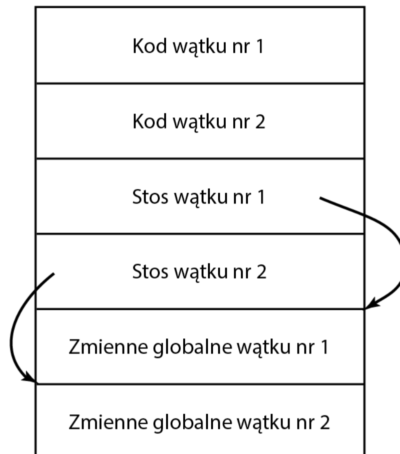
Dla przykładu przeanalizujmy zmienną `errno` występującą w systemie UNIX: kiedy proces (lub wątek) wykonuje wywołanie systemowe, które kończy się niepowodzeniem, do zmiennej `errno` jest zapisywany kod błędu. Na rysunku 2.11 wątek nr 1 wykonuje wywołanie systemowe `access` po to, aby się dowiedzieć, czy ma uprawnienia dostępu do określonego pliku. System operacyjny zwraca odpowiedź w zmiennej globalnej `errno`. Po zwróceniu sterowania do wątku 1., ale jeszcze przed przeczytaniem przez niego zmiennej `errno`, program szeregujący zdecydował, że wątek nr 1 miał przydzielony procesor wystarczająco długo i zdecydował przełączyć go do wątku 2. Kiedy wątek 1. później się uruchomi, przeczyta nieprawidłową wartość i będzie działał nieprawidłowo.



Rysunek 2.11. Konflikty pomiędzy wątkami spowodowane użyciem zmiennej globalnej

Możliwych jest wiele rozwiązań tego problemu. Jedno z nich polega na całkowitym wyłączeniu zmiennych globalnych. Choć mogłoby się wydawać, że jest to rozwiązanie idealne, koliduje ono z większością istniejących programów. Inne rozwiązanie to przypisanie każdemu wątkowi własnych, prywatnych zmiennych globalnych, tak jak pokazano na rysunku 2.12. W ten sposób każdy wątek będzie miał własną, prywatną kopię zmiennej `errno` i innych zmiennych globalnych, co pozwoli na uniknięcie konfliktów. Przyjęcie tego rozwiązania tworzy nowy poziom zasięgu: zmienne widoczne dla wszystkich procedur wątku. Poziom ten występuje obok istniejących poziomów: zmienne widoczne tylko dla jednej procedury oraz zmienne widoczne w każdym punkcie programu.

Kup ksi k Pole ksi k



Rysunek 2.12. Wątki mogą mieć prywatne zmienne globalne

Dostęp do prywatnych zmiennych globalnych jest jednak nieco utrudniony, ponieważ większość języków programowania zapewnia sposób wyrażania zmiennych lokalnych i zmiennych globalnych, ale nie ma form pośrednich. Można zaalokować fragment pamięci na zmienne globalne i przekazać go do każdej procedury w wątku w postaci dodatkowego parametru. Chociaż nie jest to zbyt eleganckie rozwiązanie, okazuje się skuteczne.

Innym rozwiązaniem może być utworzenie nowych procedur bibliotecznych do tworzenia, ustawiania i czytania tych zmiennych globalnych na poziomie wątku. Pierwsze wywołanie może mieć postać:

```
create_global("bufptr");
```

Wywołanie to alokuje pamięć dla wskaźnika o nazwie `bufptr` na stercie lub w specjalnym obszarze pamięci zarezerwowanym dla wywołującego wątku. Niezależnie od tego, gdzie jest zaalokowana pamięć, tylko wywołujący wątek ma dostęp do zmiennej globalnej. Jeśli inny wątek utworzy zmienną globalną o tej samej nazwie, otrzyma inną lokalizację w pamięci — taką, która nie koliduje z istniejącą.

Do dostępu do zmiennych globalnych potrzebne są dwa wywołania: jedno do ich zapisywania i drugie do odczytu. Do zapisywania potrzebne jest wywołanie postaci:

```
set_global("bufptr", &buf);
```

Wywołanie to zapisuje wartość wskaźnika w lokalizacji pamięci utworzonej wcześniej przez wywołanie do procedury `create_global`. Wywołanie do przeczytania zmiennej globalnej może mieć następującą postać:

```
bufptr = read_global("bufptr");
```

Zwraca ono adres zapisany w zmiennej globalnej. Dzięki temu można uzyskać dostęp do jej danych.

Następny problem podczas przekształcania programu jednowątkowego na wielowątkowy polega na tym, że wiele procedur bibliotecznych nie pozwala na tzw. wielobieżność. Oznacza to, że nie ma możliwości wywołania innej procedury, jeśli poprzednie wywołanie się nie zakończyło. I tak wysyłanie wiadomości w sieci można by z powodzeniem zaprogramować w taki sposób, aby wiadomość była tworzona w ustalonym buforze w obrębie biblioteki, a następnie był wykonywany

Kup ksi k Pole ksi k

rozkaz pułapki do jądra w celu jej wysłania. Co się stanie, jeśli jeden wątek utworzył swoją wiadomość w buforze, a następnie przerwanie zegara wymusiło przełączenie do drugiego wątku, który natychmiast nadpisze bufor własną wiadomością.

Podobnie procedury alokacji pamięci, jak `malloc` w Uniksie, utrzymują kluczowe tabele dotyczące wykorzystania pamięci — np. powiązaną listę dostępnych fragmentów pamięci. Podczas gdy procedura `malloc` jest zajęta aktualizacją tych list, mogą one czasowo być w niespójnym stanie — zawierać wskaźniki donikąd. Jeśli nastąpi przełączenie wątku w chwili, gdy tabele będą niespójne i nadejdzie nowe wywołanie z innego wątku, może dojść do użycia nieprawidłowego wskaźnika, co w efekcie może doprowadzić do awarii programu. Skuteczne wyeliminowanie wszystkich tych problemów oznacza konieczność przepisania od nowa całej biblioteki. Wykonanie takiego zadania nie jest proste. Istnieje realna możliwość popełnienia subtelnych błędów.

Innym rozwiązaniem jest wyposażenie każdej procedury w kod opakowujący, który ustawia bit do oznaczenia biblioteki tak, jakby była używana. Każda próba innego wątku skorzystania z procedury bibliotecznej, podczas gdy poprzednie wywołanie nie zostało zakończone, jest blokowana. Chociaż takie rozwiązanie jest wykonalne, w dużym stopniu eliminuje ono możliwość wykorzystania współbieżności.

Inną opcją jest wykorzystanie sygnałów. Niektóre sygnały z logicznego punktu widzenia są specyficzne dla wątku, a inne nie. Jeśli np. wątek wykonuje wywołanie alarm, logiczne jest, aby wynikowy sygnał został przesłany do wątku, który wykonał wywołanie. Jeśli jednak wątki są zaimplementowane w całości w przestrzeni użytkownika, jądro nie wie nawet o istnieniu wątków, a zatem trudno mu skierować sygnał do właściwego wątku. Dodatkowe komplikacje występują w przypadku, gdy proces pozwala na występowanie tylko jednego nieobsłużonego alarmu w danym momencie, a kilka wątków niezależnie wykonuje wywołanie alarm.

Inne sygnały, np. przerwanie klawiatury, nie są specyficzne dla wątku. Co powinno je przechwycić? Wyznaczony wątek? Wszystkie wątki? Ponadto co się stanie, jeśli jeden wątek zmieni procedury obsługi sygnałów bez informowania pozostałych wątków? A co się wydarzy, kiedy jeden wątek będzie chciał przechwycić określony sygnał (np. wciśnięcie przez użytkownika kombinacji *Ctrl+C*), a inny wątek będzie potrzebował tego sygnału do zakończenia procesu? Taka sytuacja może wystąpić, jeśli jeden wątek lub kilka wątków korzysta ze standardowych procedur bibliotecznych, a inne są napisane przez użytkownika. Jest oczywiste, że życzenia tych wątków kolidują ze sobą. Ogólnie rzecz biorąc, sygnały są trudne do zarządzania w środowisku jednowątkowym. Przejście do środowiska wielowątkowego w żaden sposób nie ułatwia zarządzania nimi.

Ostatnim problemem związanym z wątkami jest zarządzanie stosem. W wielu systemach, w przypadku wystąpienia przepełnienia stosu, jądro automatycznie dostarcza takiemu procesowi więcej miejsca na stosie. Jeśli proces ma wiele wątków, musi również mieć wiele stosów. Jeśli jądro nie posiada informacji o wszystkich tych stosach, nie może ich automatycznie rozszerzać, gdy wyczerpie się na nich miejsce. W rzeczywistości jądro może nawet nie wiedzieć, że brak strony w pamięci jest związany z rozszerzeniem się stosu jakiegoś wątku.

Problemy te nie są oczywiście nie do rozwiązania, ale pokazują, że wprowadzenie wątków do istniejącego systemu bez znaczącej jego przebudowy nie zadziała. Trzeba co najmniej zmodyfikować definicję semantyki wywołań systemowych oraz biblioteki. Wszystkie te czynności trzeba dodatkowo wykonać tak, aby zachować wsteczną zgodność z istniejącymi programami, przy założeniu, że wykorzystują one procesy zawierające po jednym wątku. Więcej informacji na temat wątków można znaleźć w tych pozycjach: [Cook, 2008] i [Rodrigues et al., 2010].

Kup ksi k Pole ksi k

2.3. SERWERY STEROWANE ZDARZENIAMI

W poprzednim podrozdziale omówiliśmy dwa możliwe projekty serwera WWW: szybki wielowątkowy i wolny jednowątkowy. Załóżmy, że wątki nie są dostępne lub niepożądane, ale projektanci systemu uznali obniżenie wydajności spowodowane istnieniem pojedynczego wątku za niedopuszczalne. Jeśli jest dostępna nieblokująca wersja wywołania systemowego `read`, możliwe staje się trzecie podejście. Kiedy przychodzi żądanie, analizuje go jeden i tylko jeden wątek. Jeżeli żądanie może być obsłużone z pamięci podręcznej, to dobrze, ale jeśli nie, inicjowana jest nieblokująca operacja dyskowa.

Serwer rejestruje stan bieżącego żądania w tabeli, a następnie pobiera następne zdarzenie do obsługi. Może to być żądanie nowej pracy albo odpowiedź dysku dotycząca poprzedniej operacji. Jeśli jest to żądanie nowej pracy, rozpoczyna się jego obsługa. Jeśli jest to odpowiedź z dysku, właściwe informacje są pobierane z tabeli i następuje przetwarzanie odpowiedzi. W przypadku nieblokujących dyskowych operacji wejścia-wyjścia odpowiedź zwykle ma postać sygnału lub przerwania.

W tym projekcie model „procesów sekwencyjnych" omawiany w pierwszych dwóch przypadkach nie występuje. Stan obliczeń musi być jawnie zapisany i odtworzony z tabeli, za każdym razem, kiedy serwer przełącza się z pracy nad jednym żądaniem do pracy nad kolejnym żądaniem. W rezultacie wątki i ich stosy są symulowane w trudniejszy sposób. W projektach takich jak ten wszystkie obliczenia mają zapisany stan. Ponadto istnieje zbiór zdarzeń, których wystąpienie może zmieniać określone stany. Takie systemy nazywa się **automatami o skończonej liczbie stanów** — pojęcie to jest powszechnie używane w branży komputerowej.

Systemy tego rodzaju są bardzo popularne na serwerach o dużej przepustowości, gdzie nawet wątki są uważane za zbyt kosztowne. Zamiast nich stosowany jest **paradygmat programowania sterowanego zdarzeniami**. Dzięki zaimplementowaniu serwera jako automatu o skończonej liczbie stanów, która reaguje na zdarzenia (np. dostępność danych w gnieździe) i komunikuje się z systemem operacyjnym za pomocą nieblokujących (lub **asynchronicznych**) wywołań systemowych, można uzyskać bardzo wydajne rozwiązanie. Każde zdarzenie prowadzi do zainicjowania sekwencji działań, ale nigdy nie są to operacje blokujące.

Na listingu 2.3 przedstawiono przykład pseudokodu sterowanego zdarzeniami serwera podziękowań (serwer dziękuje każdemu klientowi, który wysłał do niego wiadomość). Serwer używa wywołania `select` do monitorowania wielu połączeń sieciowych (wiersz 17.). Instrukcja `select` pozwala określić deskryptory plików gotowe do odbioru lub wysłania danych. Przeglądanie ich w pętli umożliwia odebranie wszystkich możliwych komunikatów. Następnie serwer próbuje wysłać komunikaty z podziękowaniami z użyciem wszystkich połączeń, które są gotowe do odbioru danych. W sytuacji gdy serwer nie może wysłać pełnej wiadomości z podziękowaniami, pamięta, które bajty musi jeszcze wysłać, więc może ponowić próbę później. Program został uproszczony tak, aby był stosunkowo krótki. Napisaliśmy go za pomocą pseudokodu i bez obsługi błędów ani zamykania połączeń. Jednak na podstawie tego przykładu widać, że jednowątkowy serwer sterowany zdarzeniami może jednocześnie obsługiwać wielu klientów.

Listing 2.3. Sterowany zdarzeniami serwer podziękowań (pseudokod)

```
0. /* Założenia :
1.         svrSock   : gniazdo głównego serwera powiązane z portem TCP 12345
2.         toSend    : baza danych wykorzystywana do śledzenia danych, które powinny być jeszcze wysłane
                       do klienta
```

```
3.                         - wywołanie toSend.put (fd, msg) zarejestruje, że musimy wysłać wiadomość
                             do deskryptora fd
4.                         - toSend.get (fd) zwraca ciąg znaków, który musimy wysłać do fd
5.                         - toSend.destroy (fd) usuwa wszystkie informacje o fd z toSend */
6.
7. inFds     = { svrSock }     /* deskryptory plików do obserwowania przychodzących danych */
8. outFds    = { }       /* deskryptory plików do obserwacji możliwości wysyłania */
9. exceptFds = { }       /* deskryptory plików do obserwacji wyjątków (nieużywane) */
10.
11. char msgBuf [MAX  MSG SIZE]      /* bufor, w którym będą odbierane wiadomości */
12. char *thankYouMsg = "Dziękuję!" /* odpowiedź do odesłania */
13.
14. while (TRUE)
15. {
16.    /* oczekiwanie na dostępne gotowe do użycia deskryptory plików */
17.    rdyIns, rdyOuts, rdyExcepts = select (inFds, outFds, exceptFds, NO TIMEOUT)
18.
19.    for (fd in rdyIns) /* iterowanie po wszystkich połączeniach, które mają dla nas wiadomość */
20.    {
21.        if (fd == svrSock) /* nowe połączenie od klienta */
22.        {
23.            newSock = accept (svrSock)   /* utwórz nowe gniazdo dla klienta */
24.            inFds = inFds ∪ {newSock}   /* też trzeba je monitorować */
25.        }
26.        else
27.        {     /* odebranie wiadomości od klienta */
28.             n = receive (fd, msgBuf, MAX MSG SIZE)
29.             printf ("Odebrano: %s.0, msgBuf)
30.
31.             toSend.put (fd, thankYouMsg) /* trzeba wysłać ThankYouMsg na ten deskryptor fd */
32.             outFds = outFds ∪ { fd }      /* więc trzeba go monitorować */
33.        }
34.    }
35.    for (fd in rdyOuts) /* iteracja po wszystkich połączeniach, dla których można wysłać podziękowania */
36.    {
37         msg = toSend.get (fd)      /* zobacz, co trzeba wysłać za pomocą tego połączenia */
38         n = send (fd, msg, strlen(msg))
39         if (n < strlen (thankYouMsg)
40         {
41             toSend.put (fd, msg+n) /* pozostałe znaki do wysłania następnym razem */
42         } else
43.        {
44.            toSend.destroy (fd)
45.            outFds = outFds \ { fd } /* już przesłaliśmy podziękowania do tego deskryptora */
46         }
47.     }
47. }
```

Większość popularnych systemów operacyjnych oferuje specjalne, zoptymalizowane interfejsy powiadomień o zdarzeniach dla asynchronicznych operacji wejścia-wyjścia, znacznie bardziej wydajne niż polecenie `select`. Dobrze znanymi przykładami są wywołanie systemowe `epoll` w Linuksie i podobny interfejs `kqueue` w systemie FreeBSD. W systemach Windows i Solaris dostępne są nieco inne rozwiązania. Wszystkie one pozwalają serwerowi monitorować wiele połączeń sieciowych jednocześnie, bez blokowania żadnego z nich. Dzięki temu serwery WWW, takie jak `nginx`, mogą swobodnie obsłużyć 10 tysięcy jednoczesnych połączeń. Nie jest to trywialne osiągnięcie. Problem uzyskał nawet unikatową nazwę: **C10k**.

Kup ksi k Pole ksi k

Serwery jednowątkowe i wielowątkowe a serwery sterowane zdarzeniami

Na koniec porównajmy trzy różne sposoby konstruowania serwerów. Teraz powinno być jasne, co oferują wątki. Pozwalają na utrzymanie idei procesów sekwencyjnych wykonujących blokujące wywołania systemowe (np. dotyczące dyskowych operacji wejścia-wyjścia) z jednoczesnym uzyskaniem efektu współbieżności. Blokujące wywołania systemowe ułatwiają programowanie, a współbieżność poprawia wydajność. Jednowątkowy serwer zachowuje prostotę blokujących wywołań systemowych, ale gwarantuje wydajność.

Trzecie podejście pozwala na osiągnięcie wysokiej wydajności dzięki współbieżności, ale wykorzystuje nieblokujące wywołania i przerwania. Uważa się je za trudniejsze do zaprogramowania. Dostępne modele zestawiono w tabeli 2.5.

Tabela 2.5. Trzy sposoby konstrukcji serwera

Model	Charakterystyka
Wątki	Współbieżność, blokujące wywołania systemowe
Proces jednowątkowy	Brak współbieżności, blokujące wywołania systemowe
Automat o skończonej liczbie stanów	Współbieżność, nieblokujące wywołania systemowe, przerwania

Te trzy podejścia do obsługi żądań klienta mają zastosowanie nie tylko do programów użytkownika, ale także do samego jądra, w którym z perspektywy poprawy wydajności współbieżność jest równie ważna. Nadszedł dobry moment, aby zwrócić uwagę na to, że ta książka wprowadza wiele pojęć systemu operacyjnego z naciskiem na ich znaczenie dla programów użytkownika. Jest jednak oczywiste, że sam system operacyjny wewnętrznie również wykorzystuje te pojęcia (niektóre z nich są nawet bardziej istotne dla systemu operacyjnego niż programów użytkownika). Zatem samo jądro systemu operacyjnego może się składać z oprogramowania wielowątkowego lub sterowanego zdarzeniami. Przykładowo jądro Linuksa na nowoczesnych procesorach Intela jest wielowątkowe. Dla odróżnienia MINIX 3 składa się z wielu serwerów zaimplementowanych zgodnie z modelem automatów o skończonej liczbie stanów i korzystających ze zdarzeń.

2.4. SYNCHRONIZACJA I KOMUNIKACJA MIĘDZYPROCESOWA

Procesy często muszą się komunikować z innymi procesami. Przykładowo w przypadku potoku w powłoce wyjście pierwszego procesu musi być przekazane do drugiego procesu, i tak dalej, do niższych warstw. Tak więc występuje potrzeba komunikacji między procesami. Najlepiej, gdyby miała ona czytelną strukturę i gdyby nie korzystano w niej z przerwań. W poniższych punktach przyjrzymy się niektórym problemom związanym z **komunikacją międzyprocesową** (ang. *inter-process communication* — IPC).

Mówiąc w skrócie: wiążą się z tym trzy problemy. O pierwszym była mowa już wcześniej: w jaki sposób jeden proces może przekazywać informacje do innego? Drugi polega na zapobieganiu sytuacji, w której dwa procesy (lub większa liczba procesów) wchodzą sobie wzajemnie w drogę; np. dwa procesy w systemie rezerwacji biletów jednocześnie próbują przydzielić ostatnie miejsce w samolocie — każdy innemu klientowi. Trzeci wiąże się z odpowiednim kolejkowaniem, w przypadku gdy występują zależności: jeśli proces A generuje dane, a proces B je drukuje, przed rozpoczęciem drukowania proces B musi czekać, aż proces A wygeneruje jakieś dane. Wszystkie trzy wymienione problemy omówimy, począwszy od następnego punktu.

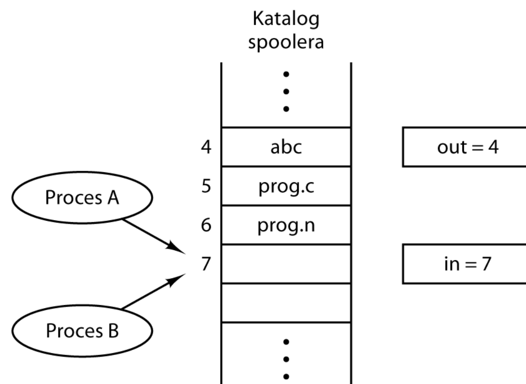
Kup ksi k Pole ksi k

Warto również wspomnieć o tym, że dwa spośród tych problemów mają w równym stopniu zastosowanie do wątków, jak do procesów ze współdzieloną pamięcią. Pierwszy z nich — przekazywanie informacji — jest łatwy w odniesieniu do wątków, ponieważ z natury wykorzystują one wspólną przestrzeń adresową. Jednak pozostałe dwa — trzymanie się z dala od szczegółów drugiego i prawidłowe sekwencjonowanie — są skomplikowane również w odniesieniu do wątków. Poniżej omówimy te problemy w kontekście procesów. Pamiętajmy jednak o tym, że te same problemy i rozwiązania mają zastosowanie także do wątków.

2.4.1. Wyścig

W niektórych systemach operacyjnych procesy, które ze sobą pracują, mogą wykorzystywać pewien wspólny obszar pamięci, do którego wszystkie mogą zapisywać i z którego wszystkie mogą czytać dane. Wspólne miejsce może znajdować się w pamięci głównej (np. w strukturze danych jądra) lub we współdzielonym pliku. Lokalizacja wspólnej pamięci nie zmienia natury komunikacji ani występujących problemów. Aby zobaczyć, jak wygląda komunikacja między procesami w praktyce, rozważmy prosty, ale klasyczny przykład: spooler drukarki. Kiedy proces chce wydrukować plik, wpisuje nazwę pliku do specjalnego **katalogu spoolera**. Inny proces, **demon drukarki**, okresowo sprawdza, czy są jakieś pliki do wydrukowania. Jeśli są, drukuje je, a następnie usuwa ich nazwy z katalogu.

Wyobraźmy sobie, że katalog spoolera ma bardzo dużą liczbę gniazd ponumerowanych 0, 1, 2, … Każde z nich może przechowywać nazwę pliku. Wyobraźmy sobie również, że istnieją dwie zmienne współdzielone: `out` — wskazująca na następny plik do wydrukowania oraz `in` — wskazująca na następne wolne gniazdo w katalogu. Te dwie zmienne równie dobrze mogą być przechowywane w pliku o objętości dwóch słów, który byłby dostępny dla wszystkich procesów. W określonym momencie gniazda 0 – 3 są puste (te pliki zostały już wydrukowane), natomiast gniazda 4 – 6 są zajęte (nazwy plików zostały umieszczone w kolejce do wydruku). Mniej więcej w tym samym czasie procesy A i B zdecydowały, że chcą umieścić plik w kolejce do wydruku. Sytuację tę pokazano na rysunku 2.13.



Rysunek 2.13. Dwa procesy w tym samym czasie chcą uzyskać dostęp do wspólnej pamięci

W przypadkach, w których mają zastosowanie prawa Murphy'ego[2], może się zdarzyć opisana poniżej sytuacja. Proces A czyta zmienną `in` i zapisuje wartość 7 w zmiennej lokalnej `next_free_slot`. W tym momencie zachodzi przerwanie zegara, a procesor decyduje, że proces

[2] Jeśli coś może się zepsuć, to na pewno się zepsuje.

Kup ksi k Pole ksi k

A działał wystarczająco długo, dlatego przełącza się do procesu B. Proces B również czyta zmienną `in` i także uzyskuje wartość 7. On też zapisuje ją w lokalnej zmiennej `next_free_slot`. W tym momencie oba procesy uważają, że następne wolne gniazdo ma numer 7.

Proces B kontynuuje działanie. Zapisuje nazwę swojego pliku w gnieździe nr 7 i aktualizuje zmienną `in` na 8. Następnie wykonuje inne czynności.

W końcu znów uruchamia się proces A, zaczynając w miejscu, w którym przerwał działanie. Odczytuje zmienną `next_free_slot`, znajduje tam wartość 7 i zapisuje swój plik w gnieździe nr 7, usuwając nazwę, którą przed chwilą umieścił tam proces B. Następnie oblicza wartość `next_free_slot+1`, co wynosi 8 i ustawia zmienną in na 8. Katalog spoolera jest teraz wewnętrznie spójny, dlatego demon drukarki nie zauważy niczego złego. Jednak proces B nigdy nie otrzyma żadnych wyników. Użytkownik B będzie się kręcił w pobliżu pokoju drukarek przez lata, bezskutecznie czekając na wydruk, który nigdy nie nadejdzie. Taka sytuacja, kiedy dwa procesy (lub większa liczba procesów) czytają lub zapisują współdzielone dane, a rezultat zależy od tego, który proces i kiedy będzie działał, jest nazywana **wyścigiem** (ang. *race condition*). Debugowanie programów, w których występują sytuacje wyścigu, w ogóle nie jest zabawne. Wyniki większości testów wychodzą poprawnie, ale od czasu do czasu zdarza się coś dziwnego i trudnego do wyjaśnienia. Niestety, wraz ze wzrostem wykorzystania współbieżności, ze względu na rosnącą liczbę rdzeni instalowanych w komputerach, sytuacje wyścigu są coraz bardziej powszechne.

2.4.2. Regiony krytyczne

W jaki sposób uniknąć sytuacji wyścigu? Kluczem do zapobiegania kłopotom w tej sytuacji, a także w wielu innych sytuacjach dotyczących współdzielonej pamięci, współdzielonych plików oraz innych współdzielonych zasobów jest znalezienie sposobu na niedopuszczenie do tego, by więcej procesów niż jeden czytało lub zapisywało współdzielone dane w tym samym czasie. Inaczej mówiąc, potrzebujemy wzajemnego wykluczenia, czyli sposobu na to, by zapewnić wyłączność korzystania ze współdzielonego zasobu — jeśli jeden proces go używa, to inny proces jest wykluczony z wykonywania tej samej operacji. Trudność w przykładzie przytoczonym powyżej wystąpiła dlatego, że proces B zaczął używać jednej ze współdzielonych zmiennych, zanim proces A przestał z niej korzystać. Wybór odpowiednich prymitywnych operacji do tego, aby osiągnąć warunki wzajemnego wykluczenia, jest jednym z głównych problemów projektowych w każdym systemie operacyjnym. Problem ten będziemy dokładnie analizować w kolejnych punktach.

Problem unikania sytuacji wyścigu można również sformułować w sposób abstrakcyjny. Przez część czasu proces jest zajęty wykonywaniem wewnętrznych obliczeń oraz innymi operacjami, które nie prowadzą do sytuacji wyścigu. Czasami jednak proces musi skorzystać ze współdzielonej pamięci lub z plików, albo wykonać inne kluczowe operacje prowadzące do wyścigu. Część programu, w której proces korzysta ze współdzielonej pamięci, nazywa się **regionem krytycznym** lub **sekcją krytyczną**. Gdyby można było tak zaprojektować operacje, aby dwa procesy nigdy nie znalazły się w krytycznych regionach w tym samym czasie, problem wyścigu byłby rozwiązany.

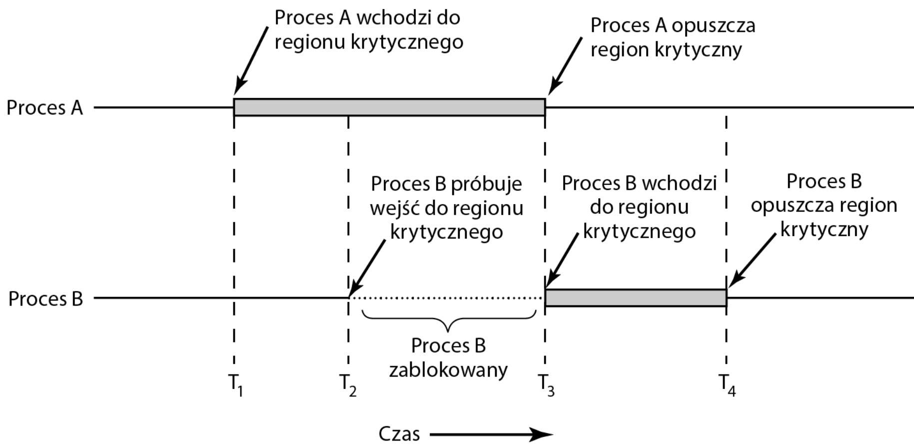
Chociaż spełnienie tego wymagania zabezpiecza przed sytuacjami wyścigu, nie wystarcza do tego, by procesy współbieżne prawidłowo i wydajnie ze sobą współpracowały, wykorzystując współdzielone dane. Dobre rozwiązanie wymaga spełnienia czterech warunków:

1. Żadne dwa procesy nie mogą jednocześnie przebywać wewnątrz swoich regionów krytycznych.
2. Nie można przyjmować żadnych założeń dotyczących szybkości lub liczby procesorów.
3. Proces działający wewnątrz swojego regionu krytycznego nie może blokować innych procesów.

Kup ksi k Pole ksi k

4. Żaden proces nie powinien oczekiwać w nieskończoność na dostęp do swojego regionu krytycznego.

W sensie abstrakcyjnym właściwości, które nas interesują, pokazano na rysunku 2.14. W tym przypadku proces *A* wchodzi do swojego regionu krytycznego w czasie T_1. Nieco później, w czasie T_2, proces *B* próbuje uzyskać dostęp do swojego regionu krytycznego, ale mu się to nie udaje, ponieważ inny proces już znajduje się w swojej sekcji krytycznej, a w danym momencie czasu zezwalamy tylko jednemu procesowi na korzystanie ze swojej sekcji krytycznej. W konsekwencji proces *B* jest czasowo zawieszony do czasu T_3, kiedy proces *A* opuści swój region krytyczny. W tym momencie proces *B* może wejść do swojego regionu krytycznego. Wreszcie proces *B* opuszcza swój region krytyczny (w momencie T_4) i z powrotem mamy sytuację, w której żaden z procesów nie znajduje się w swoim regionie krytycznym.



Rysunek 2.14. Wzajemne wykluczanie z wykorzystaniem regionów krytycznych

2.4.3. Wzajemne wykluczanie z wykorzystaniem aktywnego oczekiwania

W tym punkcie przeanalizujemy kilka propozycji osiągnięcia warunków wzajemnego wykluczania. Chcemy doprowadzić do sytuacji, w której gdy jeden proces jest zajęty aktualizacją współdzielonej pamięci w swoim regionie krytycznym, żaden inny proces nie może wejść do swojego regionu krytycznego.

Wyłączanie przerwań

W systemie jednoprocesorowym najprostszym rozwiązaniem jest spowodowanie, aby każdy z procesów zablokował wszystkie przerwania natychmiast po wejściu do swojego regionu krytycznego i ponownie je włączył bezpośrednio przed opuszczeniem regionu krytycznego. Jeśli przerwania są zablokowane, nie można wygenerować przerwania zegara.

W końcu procesor jest przełączany od procesu do procesu w wyniku przerwania zegara lub innych przerwań. Przy wyłączonych przerwaniach procesor nie może się przełączyć do innego procesu. Tak więc, jeśli proces zablokuje przerwania, może czytać i aktualizować współdzieloną pamięć bez obawy o to, że inny proces ją zmieni.

Kup ksi k Pole ksi k

Takie podejście jest, ogólnie rzecz biorąc, nieatrakcyjne, ponieważ udzielenie procesom użytkownika prawa do wyłączania przerwań nie jest zbyt rozsądne. Przypuśćmy, że jakiś proces wyłączył przerwania i nigdy ich nie włączył. To byłby koniec systemu. Co więcej, w systemie wieloprocesorowym (z dwoma procesorami lub ewentualnie większą ich liczbą) wyłączenie przerwań dotyczy tylko tego procesora, który uruchomił instrukcję `disable`. Inne procesory będą kontynuowały działanie i mogą skorzystać ze współdzielonej pamięci.

Z drugiej strony zablokowanie przerwań na czas wykonywania kilku instrukcji — np. aktualizacji zmiennych lub list — jest często wygodne dla samego jądra. Gdyby przerwanie wystąpiło w czasie, gdy lista gotowych procesów znajduje się w stanie niespójnym, mogłoby dojść do sytuacji wyścigu. Konkluzja jest następująca: zablokowanie przerwań często jest przydatną techniką wewnątrz samego systemu operacyjnego, ale nie nadaje się jako mechanizm wzajemnego wykluczania ogólnego przeznaczenia dla procesów użytkownika. Aby nie przeoczyć przerwań, jądro nie powinno ich wyłączać na więcej niż kilka instrukcji.

Prawdopodobieństwo osiągnięcia warunków wzajemnego wykluczania za pomocą blokowania przerwań — nawet w obrębie jądra — staje się coraz mniejsze ze względu na rosnącą liczbę wielordzeniowych układów nawet w tanich komputerach PC. Dwa rdzenie występują już powszechnie, cztery instaluje się w bardzo wielu maszynach, a w niedalekiej przyszłości można się spodziewać maszyn z 8, 16 lub 32 rdzeniami. W systemie wielordzeniowym (tzn. wieloprocesorowym) wyłączenie przerwań w jednym procesorze nie uniemożliwia innym procesorom przeszkadzania w operacjach, które wykonuje pierwszy procesor. W konsekwencji wymagane jest stosowanie bardziej zaawansowanych mechanizmów.

Zmienne blokujące

W drugiej kolejności przeanalizujmy rozwiązanie programowe. Rozważmy sytuację, w której mamy pojedynczą, współdzieloną zmienną (`blokada`), która początkowo ma wartość 0. Jeśli `blokada` ma wartość 0, proces ustawia ją na 1 i wchodzi do regionu krytycznego. Jeśli blokada ma wartość 1, proces czeka do chwili, kiedy będzie ona miała wartość 0. Tak więc wartość 0 oznacza, że żaden proces nie znajduje się w swoim regionie krytycznym, natomiast wartość 1 oznacza, że niektóre procesy są w swoich regionach krytycznych.

Niestety, ten pomysł ma tę samą krytyczną wadę, jaką miał katalog spoolera. Załóżmy, że proces przeczytał zmienną `blokada` i zauważył, że ma ona wartość 0. Zanim ustawił zmienną na 1, zaczął działać inny proces i ustawił zmienną `blokada` na 1. Kiedy pierwszy proces wznowi działanie, również ustawi zmienną `blokada` na 1 i dwa procesy znajdą się w swoich regionach krytycznych w tym samym czasie.

Można by sądzić, że problem da się obejść poprzez odczytanie wartości zmiennej `blokada`, a następnie ponowne sprawdzenie jej wartości bezpośrednio przed modyfikacją, ale w rzeczywistości to nie pomaga. Znów występuje sytuacja wyścigu, jeśli drugi proces zmodyfikuje zmienną bezpośrednio po tym, jak pierwszy proces zakończył drugi test.

Ścisła naprzemienność

Trzecie podejście do problemu wzajemnego wykluczania zaprezentowano na listingu 2.4. Fragment tego programu, podobnie jak prawie wszystkie w tej książce, został napisany w języku C. Wybrano go, ponieważ rzeczywiste systemy operacyjne zwykle są napisane w języku C (lub czasami w C++), a nader rzadko w takich językach jak Java, Python czy Haskell. Język C ma rozbudowane możliwości, jest wydajny i przewidywalny — są to cechy o kluczowym znaczeniu

Kup ksi k Pole ksi k

dla pisania systemów operacyjnych. Java nie jest przewidywalna. Może jej bowiem zabraknąć pamięci w kluczowym momencie, co spowoduje konieczność wywołania procesu odśmiecania w celu odzyskania pamięci w najmniej odpowiednim czasie. Nie może się to zdarzyć w języku C, ponieważ proces odśmiecania w języku C nie występuje. Porównanie ilościowe języków C, C++, Java i czterech innych można znaleźć w [Prechelt, 2000].

Listing 2.4. Proponowane rozwiązanie dla problemu regionów krytycznych: (a) proces 0, (b) proces 1. W obu przypadkach należy zwrócić uwagę na średniki kończące instrukcje while

(a)

```
while (TRUE){
    while (turn != 0)     /* pętla */;
    region_krytyczny( );
    turn = 1;
    region_niekrytyczny();
}
```

(b)

```
while (TRUE) {
    while (turn != 1)     /* pętla */;
    region_krytyczny( );
    turn = 0;
    region_niekrytyczny();
}
```

W kodzie na listingu 2.4 o możliwości wejścia procesu do regionu krytycznego w celu odczytania lub aktualizacji współdzielonej pamięci decyduje zmienna `turn`, która początkowo ma wartość 0. Najpierw proces 0 bada zmienną `turn`, odczytuje, że ma ona wartość 0 i wchodzi do regionu krytycznego. Proces 1 również odczytuje, że ma ona wartość 0, dlatego pozostaje w pętli i co jakiś czas bada zmienną `turn`, aby trafić na moment, w którym osiągnie ona wartość 1. Należy raczej unikać stosowania tej techniki, ponieważ jest ona marnotrawstwem czasu procesora. Stosuje się ją tylko wtedy, kiedy można się spodziewać, że oczekiwanie nie będzie trwało zbyt długo. Blokadę wykorzystującą aktywne oczekiwanie określa się terminem **blokady pętlowej** (ang. *spin lock*).

Kiedy proces 0 opuszcza region krytyczny, ustawia zmienną `turn` na 1. Dzięki temu proces 1 może wejść do swojego regionu krytycznego. Załóżmy, że proces 1 szybko opuścił swój region krytyczny, tak że oba procesy znajdują się teraz w regionach niekrytycznych, a zmienna `turn` ma wartość 0. Teraz proces 0 szybko uruchamia swoją pętlę, opuszcza swój region krytyczny i ustawia zmienną `turn` na 1. Od tej chwili oba procesy działają poza regionami krytycznymi.

Nagle proces 0 kończy działanie w swoim regionie niekrytycznym i powraca na początek pętli. Niestety, w tym momencie nie jest uprawniony do wejścia do regionu krytycznego, ponieważ zmienna `turn` ma wartość 1, a proces 1 jest zajęty działaniem w regionie niekrytycznym. Proces 0 oczekuje zatem w pętli `while` do czasu, aż proces 1 ustawi zmienną `turn` na 0. Mówiąc inaczej, działanie po kolei nie jest dobrym pomysłem, jeśli jeden z procesów jest znacznie wolniejszy niż drugi.

Sytuacja ta narusza warunek nr 3 sformułowany powyżej: proces 0 jest blokowany przez proces, który nie znajduje się w swoim regionie krytycznym. Wróćmy do katalogu spoolera omówionego powyżej — jeśli teraz powiązalibyśmy region krytyczny z czytaniem i zapisywaniem katalogu spoolera, proces 0 nie mógłby drukować innego pliku, ponieważ proces 1 jest zajęty czymś innym.

W rzeczywistości rozwiązanie to wymaga, aby dwa procesy ściśle naprzemiennie wchodziły do swoich regionów krytycznych, np. plików w spoolerze. Żaden z procesów nie ma prawa do skorzystania ze spoolera dwa razy z rzędu. Podczas gdy ten algorytm pozwala na uniknięcie wszystkich sytuacji wyścigu, nie jest to poważne rozwiązanie, ponieważ narusza ono warunek 3.

Kup ksi k Pole ksi k

Rozwiązanie Petersona

Dzięki połączeniu idei kolejki ze zmiennymi blokującymi i ostrzegawczymi holenderski matematyk Thomas Dekker po raz pierwszy opracował programowe rozwiązanie wzajemnego wykluczania, niewymagające ścisłej naprzemienności. Opis algorytmu Dekkera można znaleźć w [Dijkstra, 1965].

W 1981 roku Gary L. Peterson znalazł znacznie prostszy sposób osiągnięcia wzajemnego wykluczania. Dzięki temu rozwiązanie Dekkera stało się przestarzałe. Algorytm Petersona pokazano na listingu 2.5. Algorytm ten składa się z dwóch procedur napisanych w ANSI C. Oznacza to, że dla wszystkich zdefiniowanych i używanych funkcji muszą być dostarczone prototypy funkcji. Jednak dla zaoszczędzenia miejsca w tym i w kolejnych przykładach nie pokażemy prototypów.

Listing 2.5. Rozwiązanie problemu wzajemnego wykluczania zaproponowane przez Petersona

```
#define FALSE 0
#define TRUE 1
#define N    2                       /* Liczba procesów */
int turn;                            /* Czyja jest kolej? */
int interested[N];                   /* Wszystkie zmienne mają początkowo wartość 0
                                        (FALSE) */
void enter_region(int process);      /* Argument process ma wartość 0 lub 1 */
{
    int other;                       /* Liczba innych procesów */
    other = 1 - process;             /* Przeciwieństwo argumentu process */
    interested[process] = TRUE;      /* Proces pokazuje, że jest zainteresowany */
    turn = process;                  /* Ustawienie flagi */
    while (turn == process && interested[other] == TRUE) /* Instrukcja null */;
}
void leave_region(int process)      /* Argument process oznacza proces, który opuszcza
                                        region krytyczny */
{
    interested[process] = FALSE;     /* Oznacza wyjście z regionu krytycznego */
}
```

Przed skorzystaniem ze zmiennych współdzielonych (tzn. przed wejściem do swojego regionu krytycznego) każdy z procesów wywołuje funkcję `enter_region` i przekazuje do niej parametr oznaczający własny numer procesu (0 lub 1). Wywołanie to wymusza oczekiwanie, jeśli jest taka potrzeba, do momentu, aż wejście do regionu krytycznego będzie bezpieczne. Po zakończeniu korzystania ze zmiennych współdzielonych proces wywołuje funkcję `leave_region`, by w ten sposób zaznaczyć, że zakończył korzystanie z regionu krytycznego i inny proces może wejść do niego, jeśli jest taka potrzeba.

Przyjrzyjmy się, w jaki sposób działa to rozwiązanie. Początkowo żaden z procesów nie znajduje się w swoim regionie krytycznym. Teraz proces 0 wywołuje funkcję `enter_region`. Oznacza on swoje zainteresowanie skorzystaniem z regionu krytycznego poprzez ustawienie swojego elementu tablicy, a następnie ustawia zmienną `turn` na 0. Ponieważ proces 1 nie jest zainteresowany korzystaniem z regionu, funkcja `enter_region` natychmiast zwraca sterowanie. Jeśli proces 1 wykona teraz wywołanie funkcji `enter_region`, zawiesi się do czasu, aż element `interested[0]` będzie miał wartość `FALSE` — zdarzenie to zajdzie tylko wtedy, gdy proces 0 wywoła funkcję `leave_region` w celu opuszczenia regionu krytycznego.

Rozważmy teraz przypadek, w którym oba procesy wywołują funkcję `enter_region` prawie jednocześnie. Oba zapiszą numer swojego procesu w zmiennej `turn`. Zawsze liczył się będzie

Kup ksi k Pole ksi k

ten zapis, który został wykonany jako drugi. Pierwszy zostanie nadpisany i będzie utracony. Załóżmy, że proces 1 zapisał wartość jako drugi, zatem zmienna `turn` ma wartość 1. Kiedy obydwa procesy dojdą do instrukcji `while`, proces 0 wykona ją zero razy i wejdzie do swojego regionu krytycznego. Proces 1 będzie wykonywał pętlę i nie będzie mógł wejść do swojego regionu krytycznego, dopóki proces 0 nie opuści swojego regionu krytycznego.

Instrukcja TSL

Teraz przyjrzyjmy się rozwiązaniu wymagającemu trochę pomocy ze strony sprzętu. Niektóre komputery, zwłaszcza te, które zaprojektowano do pracy z wieloma procesorami, mają instrukcję następującej postaci:

```
TSL RX,LOCK
```

Instrukcja `TSL` (*Test and Set Lock* — testuj i ustaw blokadę) działa w następujący sposób: odczytuje zawartość słowa pamięci `lock` do rejestru `RX`, a następnie zapisuje niezerową wartość pod adresem pamięci `lock`. Dla operacji czytania słowa i zapisywania go jest zagwarantowana niepodzielność — do zakończenia instrukcji żaden z procesorów nie może uzyskać dostępu do słowa pamięci. W ten sposób uniemożliwia innym procesorom korzystanie z pamięci, dopóki sam nie zakończy z nią operacji.

Warto zwrócić uwagę na fakt, że zablokowanie magistrali pamięci bardzo się różni od wyłączenia przerwań. W przypadku zablokowania przerwań, jeśli po wykonaniu operacji odczytu na słowie pamięci będzie wykonany zapis, drugi procesor korzystający z magistrali w dalszym ciągu ma możliwość dostępu do słowa pamięci pomiędzy odczytem a zapisem. Zablokowanie przerwań w procesorze 1 nie ma żadnego wpływu na procesor 2. Jedynym sposobem na to, by zablokować procesorowi 2 dostęp do pamięci do chwili zakończenia pracy przez procesor 1, jest zablokowanie magistrali. To wymaga specjalnego mechanizmu sprzętowego (dokładniej ustawienia linii informującej o tym, że magistrala jest zablokowana i nie jest dostępna dla pro-cesorów, poza tym, który ją zablokował).

Aby skorzystać z instrukcji `TSL`, użyjemy współdzielonej zmiennej `lock`, pozwalającej na koordynację dostępu do współdzielonej pamięci. Jeśli zmienna `lock` ma wartość 0, dowolny proces może ustawić ją na 1 za pomocą instrukcji `TSL`, a następnie czytać lub zapisywać współdzieloną pamięć. Po zakończeniu operacji proces ustawia zmienną `lock` z powrotem na 0, korzystając ze standardowej instrukcji `move`.

W jaki sposób można skorzystać z tej instrukcji w celu uniemożliwienia dwóm procesom jednoczesnego dostępu do swoich regionów krytycznych? Rozwiązanie pokazano na listingu 2.6. Pokazano tam procedurę składającą się z czterech instrukcji w fikcyjnym (ale typowym) języku asemblera. Pierwsza instrukcja kopiuje starą wartość zmiennej `lock` do rejestru, po czym ustawia zmienną `lock` na 1. Następnie stara wartość jest porównywana z wartością 0. Wartość różna od zera oznacza, że wcześniej ustawiono blokadę, dlatego program wraca do początku i testuje zmienną jeszcze raz. Prędzej czy później zmienna przyjmie wartość 0 (kiedy proces znajdujący się w danej chwili w regionie krytycznym zakończy w nim pracę), a procedura zwróci sterowanie, wcześniej ustawiwszy blokadę. Usuwanie blokady jest bardzo proste. Program po prostu zapisuje 0 w zmiennej `lock`. Nie są potrzebne żadne specjalne instrukcje synchronizacji.

Kup ksi k Pole ksi k

Listing 2.6. Wchodzenie i opuszczanie regionu krytycznego z wykorzystaniem instrukcji TSL

```
enter_region:
    TSL REGISTER,LOCK      | Skopiowanie zmiennej lock do rejestru i ustawienie jej na 1
    CMP REGISTER,#0        | Czy zmienna lock miała wartość zero?
    JNE enter_region       | Wartość różna od zera oznacza, że była blokada, zatem
                           | wracamy na początek pętli
    RET                    | Zwrócenie sterowania do wywołującego. Wejście do regionu
                           | krytycznego

leave_region:
    MOVE LOCK,#0           | Zapisanie 0 w zmiennej lock
    RET                    | Zwrócenie sterowania do wywołującego
```

Jedno z rozwiązań problemu regionu krytycznego jest teraz proste. Funkcja ta realizuje aktywne oczekiwanie do chwili, kiedy blokada będzie zwolniona. Następnie ustawia blokadę i zwraca sterowanie. Po opuszczeniu regionu krytycznego proces wywołuje procedurę `leave_region`, która zapisuje 0 w zmiennej `lock`. Podobnie jak w przypadku wszystkich rozwiązań, które bazują na regionach krytycznych, aby metoda mogła działać, procesy muszą w odpowiednich momentach wywołać instrukcje `enter_region` i `leave_region`. Jeśli jakiś proces będzie „oszukiwał", warunek wzajemnego wykluczania nie będzie mógł być spełniony. Inaczej mówiąc, regiony krytyczne działają tylko wtedy, gdy procesy współpracują.

Alternatywą dla instrukcji `TSL` jest `XCHG`. Jej działanie polega na zamianie zawartości dwóch lokalizacji — np. rejestru i słowa pamięci. Kod oparty na rozkazie `XCHG` zaprezentowano na listingu 2.7. Jak można zauważyć, zasadniczo jest on identyczny jak rozwiązanie z instrukcją `TSL`. Z niskopoziomowej synchronizacji opartej na o rozkazie `XCHG` korzystają wszystkie procesory x86 firmy Intel.

Listing 2.7. Wchodzenie i opuszczanie regionu krytycznego z wykorzystaniem instrukcji XCHG

```
enter_region:
    MOVE REGISTER,#1        | Umieszczenie 1 w rejestrze
    XCHG REGISTER,LOCK      | Wymiana zawartości pomiędzy rejestrem a zmienną lock
    CMP REGISTER,#0         | Czy zmienna lock miała wartość zero?
    JNE enter_region        | Wartość różna od zera oznacza, że była blokada, zatem
                            | wracamy na początek pętli
     RET                    | Zwrócenie sterowania do wywołującego. Wejście do regionu
                            | krytycznego
leave_region:
    MOVE LOCK,#0            | Zapisanie 0 w zmiennej lock
    RET                     | Zwrócenie sterowania do wywołującego
```

2.4.4. Wywołania sleep i wakeup

Zarówno rozwiązanie Petersona, jak i rozwiązanie oparte na rozkazach `TSL` lub `XCHG` są poprawne, ale oba są obarczone defektem polegającym na konieczności korzystania z aktywnego oczekiwania. W skrócie działanie tych rozwiązań można ująć następująco: jeśli proces chce wejść do swojego regionu krytycznego, sprawdza, czy wejście jest dozwolone. Jeśli nie, proces pozostaje w pętli w oczekiwaniu na to, aż region stanie się dostępny.

Przy takim podejściu nie tylko jest marnotrawiony czas procesora, ale dodatkowo może ono przynosić nieoczekiwane efekty. Rozważmy przykład komputera z dwoma procesami — H o wysokim priorytecie i L o niskim priorytecie. Reguły szeregowania są takie, że proces H działa zawsze, kiedy jest w stanie gotowości. W pewnym momencie, kiedy proces L znajduje się w swoim

Kup ksi k Pole ksi k

regionie krytycznym, proces H zyskuje gotowość (np. kończy wykonywanie operacji wejścia-wyjścia). W tym momencie H rozpoczyna aktywne oczekiwanie, ale ponieważ proces L nigdy nie będzie zaplanowany w czasie, gdy działa proces H, proces L nigdy nie otrzyma szansy opuszczenia swojego regionu krytycznego. Sytuację tę czasami określa się jako **problem inwersji priorytetów**.

Przyjrzyjmy się teraz pewnym prymitywom komunikacji międzyprocesorowej — operacjom, które w momentach, kiedy procesy nie mogą wejść do swoich regionów, blokują je, zamiast marnotrawić czas procesora. Do najprostszych należy para `sleep` i `wakeup`. `sleep` to wywołanie systemowe, które powoduje zablokowanie procesu wywołującego — tzn. zawieszenie go do czasu, kiedy inny proces go obudzi. Wywołanie `wakeup` ma jeden parametr — identyfikator procesu, który ma być obudzony. W innych implementacjach zarówno operacja `sleep`, jak i `wakeup` mają po jednym parametrze — adresie pamięci stosowanym w celu dopasowania operacji `sleep` do operacji `wakeup`.

Problem producent-konsument

W celu zaprezentowania przykładu użycia tych prymitywów rozważmy **problem producent-konsument** (znany także jako **problem ograniczonego bufora** — ang. *bounded-buffer*). Dwa procesy współdzielą bufor o stałym rozmiarze. Jeden z nich, producent, umieszcza informacje w buforze, natomiast drugi, konsument, je z niego pobiera (można również uogólnić problem dla m producentów i n konsumentów; my jednak będziemy rozważać przypadek tylko jednego producenta i jednego konsumenta, ponieważ to założenie upraszcza rozwiązania).

Problemy powstają w przypadku, kiedy producent chce umieścić nowy element w buforze, który jest już pełny. Rozwiązaniem dla producenta jest przejście do stanu uśpienia i zamówienie „budzenia” w momencie, kiedy konsument usunie z bufora jeden lub kilka elementów. Na podobnej zasadzie, jeśli konsument zechce usunąć element z bufora i zobaczy, że bufor jest pusty, przechodzi do stanu uśpienia i pozostaje w nim dopóty, dopóki producent nie umieści jakichś elementów w buforze i nie obudzi konsumenta.

To podejście wydaje się dość proste, ale prowadzi do sytuacji wyścigu, podobnej do tych, z jakimi mieliśmy do czynienia wcześniej, podczas omawiania katalogu spoolera. Do śledzenia liczby elementów w buforze potrzebna będzie zmienna `count`. Jeśli maksymalna liczba elementów, jakie mogą się zmieścić w buforze, wynosi N, w kodzie producenta trzeba będzie najpierw sprawdzić, czy `count` równa się N. Jeśli tak, to producent przechodzi do stanu uśpienia.

Kod konsumenta jest podobny: najpierw testowana jest zmienna count w celu sprawdzenia, czy ma wartość 0. Jeśli tak, przechodzi do stanu uśpienia. Jeśli ma wartość niezerową, usuwa element z bufora i dekrementuje licznik. Każdy z procesów sprawdza również, czy należy obudzić inny proces. Jeśli tak, to go budzi. Kod dla producenta i konsumenta zaprezentowano na listingu 2.8.

Listing 2.8. Problem producent-konsument z krytyczną sytuacją wyścigu

```
#define N 100                         /* liczba miejsc w buforze */
int count = 0;                        /* liczba elementów w buforze */
void producer(void)
{
     int item;
    while (TRUE) {                    /* pętla nieskończona */
        item = produce_item( );       /* wygenerowanie następnego elementu */
        if (count == N) sleep( );     /* jeśli bufor jest pełny, przejście do uśpienia */
```

```
        insert_item(item);           /* umieszczenie elementu w buforze */
        count = count + 1;           /* inkrementacja licznika elementów w buforze */
        if (count == 1) wakeup(consumer);          /* czy bufor był pusty? */
    }
}
void consumer(void)
{
    int item;
    while (TRUE) {                   /* pętla nieskończona */
        if (count == 0) sleep( );    /* jeśli bufor jest pusty, przejście do uśpienia */
        item = remove_item( );       /* pobranie elementu z bufora */
        count = count - 1;           /* dekrementacja licznika elementów w buforze */
        if (count == N - 1) wakeup(producer);     /* czy bufor był pełny? */
        consume_item(item);          /* wyświetlenie elementu */
    }
}
```

W celu wyrażenia wywołań systemowych, takich jak `sleep` i `wakeup` w języku C, pokażemy je jako wywołania do procedur bibliotecznych. Nie są one częścią standardowej biblioteki C, ale przypuszczalnie będą dostępne w każdym systemie, w którym są wykorzystywane wspomniane wywołania systemowe. Procedury `insert_item` i `remove_item`, których nie pokazano, obsługują operacje umieszczania elementów w buforze i pobierania elementów z bufora.

Teraz powróćmy na chwilę do sytuacji wyścigu. Może się ona zdarzyć ze względu na to, że dostęp do zmiennej `count` jest nieograniczony. W konsekwencji prawdopodobna wydaje się następująca sytuacja: bufor jest pusty, a konsument właśnie przeczytał zmienną `count` i dowiedział się, że ma ona wartość 0. W tym momencie program szeregujący zadecydował, że czasowo przerwie działanie konsumenta i uruchomi producenta. Producent wstawił element do bufora, przeprowadził inkrementację zmiennej `count` i zauważył, że teraz ma ona wartość 1. Na podstawie tego, że zmienna `count` wcześniej miała wartość 0, producent sądzi, że konsument jest uśpiony, a w związku z tym wywołuje `wakeup` w celu zbudzenia go.

Niestety, konsument nie jest jeszcze logicznie uśpiony, zatem sygnał pobudki nie zadziała. Kiedy konsument ponownie zadziała, sprawdzi wartość zmiennej `count`, którą przeczytał wcześniej, dowie się, że ma ona wartość 0 i przejdzie do stanu uśpienia. Prędzej czy później producent wypełni bufor i również przejdzie do uśpienia. Oba procesy będą spały na zawsze.

Sedno tego problemu polega na tym, że sygnał `wakeup` wysłany do procesu, który jeszcze nie spał, został utracony. Gdyby nie został utracony, wszystko działałoby jak należy. Szybkim rozwiązaniem problemu jest modyfikacja reguł polegająca na dodaniu **bitu oczekiwania na sygnał wakeup**. Bit ten jest ustawiany w przypadku, gdy sygnał `wakeup` zostanie wysłany do procesu, który nie jest uśpiony.

Kiedy proces spróbuje później przejść do stanu uśpienia, to w przypadku gdy jest ustawiony bit oczekiwania na sygnał `wakeup`, zostanie on wyłączony, ale proces nie przejdzie do stanu uśpienia. Bit oczekiwania na sygnał `wakeup` jest skarbonką pozwalającą na przechowywanie sygnałów `wakeup`. Konsument zeruje bit oczekiwania na sygnał `wakeup` w każdej iteracji pętli.

O ile pojedynczy bit oczekiwania na sygnał `wakeup` rozwiązuje problem w tym prostym przykładzie, o tyle łatwo skonstruować przykłady z trzema procesami lub większą ich liczbą, w których jeden bit oczekiwania na sygnał `wakeup` nie wystarczy. Można by stworzyć kolejną łatkę i dodać jeszcze jeden bit oczekiwania na sygnał `wakeup` lub stworzyć ich 32, albo nawet 64, ale zasadniczy problem i tak pozostanie.

Kup ksi k Pole ksi k

2.4.5. Semafory

Taka była sytuacja w 1965 roku, kiedy Dijkstra zaproponował użycie zmiennej całkowitej do zliczania liczby zapisanych sygnałów `wakeup`. W swojej propozycji przedstawił nowy typ zmiennej, którą nazwał **semaforem**. Semafor może mieć wartość 0, co wskazuje na brak zapisanych sygnałów `wakeup`, lub jakąś wartość dodatnią, gdyby istniał jeden zaległy sygnał `wakeup` lub więcej takich sygnałów.

Dijkstra zaproponował dwie operacje: `down` i `up` (odpowiednio uogólnienia operacji `sleep` i `wakeup`). Operacja `down` na semaforze sprawdza, czy wartość zmiennej jest większa od 0. Jeśli tak, dekrementuje tę wartość (tzn. wykonuje operację `up` z argumentem 1 dla zapisanych sygnałów `wakeup`) i kontynuuje. Jeśli wartość wynosi 0, proces jest przełączany na chwilę w stan uśpienia bez wykonywania operacji `down`. Sprawdzanie wartości, modyfikowanie jej i ewentualnie przechodzenie do stanu uśpienia jest wykonywane w pojedynczej i niepodzielnej akcji. Istnieje gwarancja, że kiedy rozpocznie się operacja na semaforze, żaden inny proces nie będzie mógł uzyskać do niego dostępu, aż operacja zakończy się lub zostanie zablokowana. Ta niepodzielność ma absolutnie kluczowe znaczenie dla rozwiązywania problemów synchronizacji i unikania sytuacji wyścigu. Niepodzielne akcje, w których grupa powiązanych operacji albo jest wykonywana bez przerwy, albo nie jest wykonywana wcale, są niezwykle ważne w wielu obszarach informatyki.

Operacja `up` inkrementuje wartość wskazanego semafora. Jeśli na tym semaforze był uśpiony jeden proces lub więcej procesów, które nie mogły wykonać wcześniejszej operacji `down`, to system wybiera jeden z nich (np. losowo) i zezwala na dokończenie operacji `down`. Tak więc po wykonaniu operacji `up` na semaforze, na którym były uśpione procesy, semafor w dalszym ciągu będzie miał wartość 0, ale będzie na nim uśpiony o jeden proces mniej. Operacja inkrementacji semafora i budzenia jednego procesu również jest niepodzielna. Żaden proces nigdy nie blokuje wykonania operacji `up`, podobnie jak w poprzednim modelu żaden proces nie mógł blokować operacji `wakeup`.

Tak na marginesie — w oryginalnym artykule Dijkstra zamiast nazw operacji `down` i `up` użył odpowiednio nazw `P` i `V`. Ponieważ nie mają one znaczenia mnemonicznego dla ludzi nieznających języka holenderskiego i niewielkie znaczenie dla tych, którzy go znają — *Proberen* (próbuj) i *Verhogen* (podnieś) — zamiast nich będziemy używać nazw `down` i `up`. Po raz pierwszy operacje te wprowadzono w języku programowania Algol 68.

Rozwiązanie problemu producent-konsument z wykorzystaniem semaforów

Semafory rozwiązują problem utraconych sygnałów `wakeup`, co zaprezentowano na listingu 2.9. Aby działały prawidłowo, istotne znaczenie ma zaimplementowanie ich w sposób niepodzielny. System operacyjny na czas sprawdzania semafora powinien zablokować przerwania, zaktualizować semafor i jeśli trzeba — przełączyć proces do stanu uśpienia. Ponieważ wszystkie te działania zajmują tylko kilka instrukcji, zablokowanie przerwań nie przynosi szkody. W przypadku użycia wielu procesorów każdy semafor powinien być chroniony przez zmienną blokady. W celu sprawdzenia, że tylko jeden procesor w danym momencie bada semafor, można użyć instrukcji `TSL` lub `XCHG`.

Listing 2.9. Rozwiązanie problemu producent-konsument z wykorzystaniem semaforów

```
#define N 100                    /* liczba miejsc w buforze */
typedef int semaphore;           /* semafory to specjalny rodzaj danych typu int */
semaphore muteks = 1;            /* zarządza dostępem do regionu krytycznego */
semaphore empty = N;             /* zlicza puste miejsca w buforze */
semaphore full = 0;              /* zlicza zajęte miejsca w buforze */
```

Kup ksi k Pole ksi k

```
void producer(void)
{
    int item;
    while (TRUE) {              /* TRUE jest stałą o wartości 1 */
        item = produce_item( ); /* wygenerowanie wartości do umieszczenia w buforze */
        down(&empty);           /* dekrementacja licznika pustych */
        down(&muteks);          /* wejście do regionu krytycznego */
        insert_item(item);      /* umieszczenie nowego elementu w buforze */
        up(&muteks);            /* opuszczenie regionu krytycznego */
        up(&full);              /* inkrementacja licznika zajętych miejsc */
    }
}
void consumer(void)
{
    int item;
    while (TRUE) {              /* pętla nieskończona */
        down(&full);            /* dekrementacja licznika zajętych */
        down(&muteks);          /* wejście do regionu krytycznego */
        item = remove_item( );  /* pobranie elementu z bufora */
        up(&muteks);            /* opuszczenie regionu krytycznego */
        up(&empty);             /* inkrementacja licznika pustych miejsc */
        consume_item(item);     /* wykonanie operacji z elementem */
    }
}
```

Należy zdać sobie sprawę z tego, że użycie instrukcji `TSL` lub `XCHG` w celu uniemożliwienia kilku procesorom korzystania z semafora w tym samym czasie różni się od aktywnego oczekiwania producenta lub konsumenta na opróżnienie lub wypełnienie bufora. Operacja na semaforze zajmuje tylko kilka mikrosekund, podczas gdy oczekiwanie producenta lub konsumenta mogło trwać dowolnie długo.

W pokazanym rozwiązaniu użyto trzech semaforów: semafor `full` służy do zliczania gniazd, które są zajęte, semafor `empty` służy do zliczania gniazd, które są puste, natomiast semafor `mutex` zapewnia, aby producent i konsument nie korzystali z bufora jednocześnie. Semafor `full` początkowo ma wartość 0, `empty` ma początkową wartość równą liczbie gniazd w buforze, natomiast `mutex` początkowo ma wartość 1. Semafory inicjowane wartością 1 i używane przez dwa procesy lub większą ich liczbę po to, by zyskać pewność, że tylko jeden z nich może wejść do swojego regionu krytycznego w tym samym czasie, nazywają się **semaforami binarnymi**. Jeśli proces wykona operację `down` bezpośrednio przed wejściem do swojego regionu krytycznego i `up` bezpośrednio po jego opuszczeniu, wzajemne wykluczanie jest zapewnione.

Teraz, kiedy dysponujemy dobrymi prymitywami komunikacji między procesami, powróćmy na chwilę do sekwencji przerwań pokazanej na rysunku 2.5. W systemie, który używa semaforów, naturalnym sposobem ukrycia przerwań jest powiązanie semafora, początkowo ustawionego na 0, z każdym urządzeniem wejścia-wyjścia. Bezpośrednio po uruchomieniu urządzenia wejścia-wyjścia, proces zarządzający wykonuje operację `down` na powiązanym z nim semaforze, a tym samym natychmiast się blokuje. Kiedy nadejdzie przerwanie, procedura obsługi przerwania wykonuje operację `up` na powiązanym semaforze. Dzięki temu proces jest gotowy do ponownego uruchomienia. W tym modelu krok 5. z rysunku 2.5 składa się z wykonania operacji `up` na semaforze powiązanym z urządzeniem. Dzięki temu w kroku 6. program szeregujący może uruchomić menedżera urządzeń. Oczywiście w przypadku, gdy kilka procesów będzie gotowych, program szeregujący będzie mógł uruchomić w następnej kolejności ważniejszy proces. Niektóre z wykorzystanych algorytmów szeregowania omówimy w dalszej części rozdziału.

Kup ksi k Pole ksi k

W przykładzie z listingu 2.9 użyliśmy semaforów na dwa sposoby. Różnica pomiędzy nimi jest na tyle ważna, że należy ją wyjaśnić. Semafor `mutex` jest wykorzystywany do wzajemnego wykluczania. Służy do tego, by można było zagwarantować, że tylko jeden proces w danym czasie odczytuje bufor i powiązane z nim zmienne. To wzajemne wykluczanie jest wymagane w celu przeciwdziałania chaosowi. Zagadnienie wzajemnego wykluczania oraz sposobów osiągnięcia tego stanu omówimy w następnym punkcie.

Poza wzajemnym wykluczaniem semafory wykorzystuje się do synchronizacji. Semafory `full` i `empty` są potrzebne do tego, by zagwarantować, że określone sekwencje zdarzeń wystąpią lub nie. W tym przypadku zapewniają one, że producent przestanie działać, kiedy bufor będzie pełny, oraz że konsument przestanie działać, kiedy bufor będzie pusty. To zastosowanie różni się od realizacji wzajemnego wykluczania.

Problem czytelników i pisarzy

Problem producent-konsument przydaje się do modelowania dwóch procesów (lub wątków), które wymieniają bloki danych podczas współdzielenia bufora. Innym znanym problemem jest problem czytelników i pisarzy [Courtois et al., 1971], który modeluje dostęp do bazy danych. Dla przykładu wyobraźmy sobie system rezerwacji lotniczej zawierający wiele rywalizujących ze sobą procesów, które chcą czytać go i zapisywać. Dopuszczalna jest sytuacja, w której wiele procesów jednocześnie czyta bazę danych, ale jeśli jeden proces aktualizuje (zapisuje) bazę danych, żaden inny proces — nawet czytelnicy — nie może uzyskać dostępu do bazy danych. Problem polega na tym, w jaki sposób zaprogramować procesy czytelników i pisarzy? Jedno z rozwiązań przedstawiono na listingu 2.10.

Listing 2.10. Rozwiązanie problemu czytelników i pisarzy

```
typedef int semaphore;            /* użyjmy swojej wyobraźni */
semaphore mutex = 1;              /* zarządza dostępem do zmiennej 'rc' */
semaphore db = 1;                 /* zarządza dostępem do bazy danych */
int rc = 0;                       /* liczba procesów, które czytają lub chcą czytać */
void reader(void)
{
    while (TRUE) {                /* pętla nieskończona */
        down(&mutex);             /* uzyskanie wyłącznego dostępu do zmiennej 'rc' */
        rc = rc + 1;              /* teraz jest o jednego czytelnika więcej */
        if (rc == 1) down(&db);   /* jeśli to był pierwszy czytelnik... */
        up(&mutex);               /* zwolnienie wyłącznego dostępu do 'rc' */
        read_data_base( );        /* dostęp do danych */
        down(&mutex);             /* uzyskanie wyłącznego dostępu do zmiennej 'rc' */
        rc = rc - 1;              /* teraz jest o jednego czytelnika mniej */
        if (rc == 0) up(&db);     /* jeśli to jest ostatni czytelnik... */
        up(&mutex);               /* zwolnienie wyłącznego dostępu do 'rc' */
        use_data_read( );         /* region niekrytyczny */
    }
}
void writer(void)
{
    while (TRUE) {                 /* pętla nieskończona */
        think_up_data( );          /* region niekrytyczny */
        down(&db);                 /* uzyskanie wyłącznego dostępu */
        write_data_base( );        /* aktualizacja danych */
        up(&db);                   /* zwolnienie wyłącznego dostępu */
    }
}
```

Kup ksi k Pole ksi k

W pokazanym rozwiązaniu pierwszy czytelnik, który chce uzyskać dostęp do bazy danych, wykonuje operację `down` na semaforze `db`. Kolejni czytelnicy jedynie inkrementują licznik `rc`. Kiedy czytelnicy przestają korzystać z bazy danych, dekrementują licznik. Ostatni z nich wykonuje operację `up` na semaforze, pozwalając na skorzystanie z bazy danych zablokowanemu pisarzowi, jeśli taki jest.

Zaprezentowane tutaj rozwiązanie niejawnie zawiera subtelną decyzję, na którą warto zwrócić uwagę. Załóżmy, że podczas gdy czytelnik korzysta z bazy danych, inny czytelnik zgłasza chęć dostępu do bazy danych. Ponieważ dwóch czytelników w tym samym czasie nie jest problemem, drugi czytelnik uzyskuje prawo dostępu. Następni czytelnicy również mogą uzyskać dostęp, jeśli zgłoszą taką chęć.

Załóżmy teraz, że pojawia się pisarz. Nie może on uzyskać dostępu do bazy danych, ponieważ pisarze muszą mieć dostęp na wyłączność, zatem pisarz jest zawieszany.

Po pewnym czasie pojawiają się dodatkowi czytelnicy. Tak długo, jak co najmniej jeden czytelnik jest aktywny, kolejni czytelnicy uzyskują dostęp. Konsekwencja stosowania tej strategii będzie taka, że jeśli wystąpi stały dopływ czytelników, każdy z nich otrzyma dostęp natychmiast po przybyciu. Pisarz będzie zawieszony dopóty, dopóki nie będzie żadnego czytelnika. Jeśli nowy czytelnik będzie zgłaszał chęć dostępu, np. co 2 s, a wykonanie jego pracy zajmie 5 s, to pisarz nigdy nie uzyska dostępu. Oczywiście ta sytuacja nie jest zadowalająca.

Aby jej zapobiec, można by napisać program nieco inaczej: kiedy czytelnik zgłasza chęć skorzystania z bazy danych, a pisarz czeka, nie otrzymuje dostępu natychmiast, tylko jest zawieszany do czasu obsłużenia pisarza. W ten sposób pisarz musi czekać na czytelników, którzy byli aktywni w momencie jego zgłoszenia, ale nie musi czekać na czytelników, którzy zgłosili się po nim. Wada tego rozwiązania polega na tym, że zapewnia ono niższy stopień współbieżności, a tym samym niższą wydajność. Courtois i współpracownicy zaprezentowali rozwiązanie, które nadaje priorytet pisarzom. Szczegółowe informacje można znaleźć w ich artykule.

2.4.6. Muteksy

Jeśli nie jest potrzebna właściwość semafora do zliczania, czasami używa się uproszczonej wersji semaforów, zwanej muteksami (ang. *mutex*). Muteksy nadają się wyłącznie do zarządzania wzajemnym wykluczaniem niektórych współdzielonych zasobów lub fragmentu kodu. Dzięki temu okazują się szczególnie przydatne w pakietach obsługi wątków, które w całości są implementowane w przestrzeni użytkownika.

Muteks jest zmienną, która może znajdować się w jednym z dwóch stanów: „odblokowany" lub „zablokowany". W efekcie do jego zaprezentowania jest potrzebny tylko 1 bit. W praktyce w tej roli często wykorzystuje się dane `integer`, przy czym wartość 0 oznacza „odblokowany", natomiast wszystkie inne wartości oznaczają „zablokowany". Z muteksami wykorzystuje się dwie procedury. Kiedy wątek (lub proces) potrzebuje dostępu do regionu krytycznego, wywołuje funkcję `mutex_lock`. Jeśli muteks jest już odblokowany (co oznacza, że jest dostępny region krytyczny), wywołanie kończy się sukcesem i wątek wywołujący może wejść do regionu krytycznego.

Z drugiej strony, jeśli muteks jest już zablokowany, wątek wywołujący zablokuje się do czasu, kiedy wątek znajdujący się w regionie krytycznym zakończy w nim działania i wywoła funkcję `mutex_unlock`. Jeśli na muteksie jest zablokowanych wiele wątków, losowo wybierany jest jeden z nich i otrzymuje zgodę na założenie blokady.

Ponieważ muteksy są tak proste, można je z łatwością zaimplementować w przestrzeni użytkownika, pod warunkiem że będą dostępne instrukcje `TSL` lub `XCHG`. Kod operacji `mutex_lock` i `mutex_unlock`, których można użyć z pakietem obsługi wątków poziomu użytkownika, pokazano na listingu 2.11.

Listing 2.11. Implementacja operacji muteks_lock i muteks_unlock

```
muteks_lock:
    TSL REGISTER,MUTEKS        | skopiowanie muteksa do rejestru i ustawienie go na 1
    CMP REGISTER,#0            | Czy muteks miał wartość zero?
    JZE ok                     | Jeśli miał wartość zero, był odblokowany, zatem
                                 funkcja kończy działanie
    CALL thread_yield          | Muteks jest zajęty — zaplanowanie innego wątku
    JMP muteks_lock            | ponowienie próby
ok: RET                        | Zwrócenie sterowania do procesu wywołującego.
                               | Wejście do regionu krytycznego
muteks_unlock:
    MOVE MUTEKS,#0             | Zapisanie 0 w muteksie
    RET                        | Zwrócenie sterowania do procesu wywołującego
```

Kod operacji `mutex_lock` jest podobny do kodu operacji `enter_region` z listingu 2.5 z jedną zasadniczą różnicą. Kiedy funkcja `enter_region` nie zdoła wejść do regionu krytycznego, wielokrotnie powtarza testowanie blokady (aktywne oczekiwanie). Kiedy skończy się przydzielony czas, zaczyna działać inny proces. Prędzej czy później proces utrzymujący blokadę zacznie działać i ją zwolni.

W przypadku zastosowania wątków (użytkownika) sytuacja jest inna, ponieważ nie ma zegara, który zatrzymuje zbyt długo działające wątki. W konsekwencji wątek chcący uzyskać blokadę poprzez aktywne oczekiwanie będzie wykonywał się w pętli nieskończonej. W związku z tym nigdy nie uzyska blokady, ponieważ nigdy nie pozwoli żadnemu innemu wątkowi na uruchomienie się i zwolnienie blokady.

W tym miejscu ujawnia się różnica pomiędzy funkcjami `enter_region` i `mutex_lock`. Kiedy tej drugiej nie uda się ustawić blokady, wywołuje funkcję `thread_yield` po to, by przekazać procesor do innego wątku. W konsekwencji nie ma aktywnego oczekiwania. Kiedy wątek uruchomi się następnym razem, ponownie analizuje blokadę.

Ponieważ `thread_yield` to wywołanie do procesu zarządzającego wątkami w przestrzeni użytkownika, jest ono bardzo szybkie. W konsekwencji ani wywołanie `mutex_lock`, ani `mutex_unlock` nie wymagają żadnych wywołań jądra. Dzięki ich wykorzystaniu wątki poziomu użytkownika mogą się synchronizować w całości w przestrzeni użytkownika, z wykorzystaniem procedur wymagających zaledwie kilku instrukcji.

Opisany powyżej system muteksa jest prymitywnym zbiorem wywołań. W przypadku każdego oprogramowania zawsze występuje potrzeba dodatkowych własności. Prymitywy synchronizacji nie są tu wyjątkiem — np. czasami w pakiecie obsługi wątków jest wywołanie `mutex_trylock`, które albo ustanawia blokadę, albo zwraca kod błędu, ale nie blokuje się. Wywołanie to daje wątkowi możliwość decydowania o tym, co zrobić w następnej kolejności, jeśli istnieją jakieś alternatywy do oczekiwania.

Istnieje pewien subtelny problem, który na razie przemilczeliśmy, a który warto jawnie przedstawić. W przypadku pakietu obsługi wątków działającego w przestrzeni użytkownika nie ma problemu z tym, że do tego samego muteksa ma dostęp wiele wątków, ponieważ wszystkie wątki działają we wspólnej przestrzeni adresowej. Jednak w przypadku większości wcześniej omawianych rozwiązań takich problemów — np. algorytmu Petersona i semaforów — przyjmuje się założenie, że przynajmniej do fragmentu współdzielonej pamięci (np. do określonego słowa) ma dostęp wiele procesów. Jeśli procesy posługują się rozdzielnymi przestrzeniami adresowymi, tak jak powiedzieliśmy, to w jaki sposób mogą one współdzielić zmienną `turn` z algorytmu Petersona, semafory albo wspólny bufor?

Kup ksi k Pole ksi k

Są dwie odpowiedzi. Po pierwsze niektóre ze współdzielonych struktur danych, np. semafory, mogą być przechowywane w jądrze, a dostęp do nich jest możliwy tylko za pomocą wywołań systemowych. Takie podejście eliminuje problem. Po drugie w większości systemów operacyjnych (włącznie z systemami UNIX i Windows) istnieje mechanizm, który pozwala procesom współdzielić pewną część swojej przestrzeni adresowej z innymi procesami. W ten sposób bufory i inne struktury danych mogą być współdzielone. W najgorszej sytuacji, kiedy nie jest możliwe nic innego, można wykorzystać współdzielony plik.

Jeśli dwa procesy lub większa ich liczba współdzielą większość lub całość swoich przestrzeni adresowych, różnica pomiędzy procesami a wątkami staje się w pewnym stopniu rozmyta, niemniej jednak istnieje. Dwa procesy, które współdzielą przestrzeń adresową, posługują się różnymi otwartymi plikami, licznikami czasu alarmów i innymi właściwościami procesów, podczas gdy wątki w obrębie pojedynczego procesu je współdzielą. Ponadto wiele procesów współdzielących przestrzeń adresową nie dorównuje wydajnością wielu wątkom działającym w przestrzeni użytkownika, ponieważ w ich zarządzaniu aktywny udział bierze jądro.

Futeksy

Wraz ze wzrostem znaczenia współbieżności istotne stają się skuteczne mechanizmy synchronizacji i blokowania, ponieważ zapewniają wydajność. Blokady pętlowe (ang. *spin locks*) są szybkie, jeśli czas oczekiwania jest krótki, w przeciwnym razie powodują marnotrawienie cykli procesora. Z tego powodu, w przypadku gdy rywalizacja jest duża, bardziej wydajne jest zablokowanie procesu i zlecenie jądru, aby odblokowanie go nastąpiło dopiero wtedy, gdy blokada zostanie zwolniona. Niestety, to powoduje odwrotny problem: sprawdza się w przypadku dużej rywalizacji, ale ciągłe przełączanie do jądra jest kosztowne, gdy rywalizacji nie ma zbyt wiele. Co gorsza, to, ile będzie rywalizacji o blokady, nie jest łatwe do przewidzenia. Ciekawym rozwiązaniem, które stara się połączyć najlepsze cechy z obu światów, są tzw. **futeksy** czyli szybkie muteksy w przestrzeni użytkownika (ang. *fast user space muteks*).

Futeks jest własnością Linuksa, która implementuje podstawowe blokowanie (podobnie jak muteks), ale unika odwoływania się do jądra, jeśli nie jest to bezwzględnie konieczne. Ponieważ przełączanie się do jądra i z powrotem jest dość kosztowne, zastosowanie futeksów znacznie poprawia wydajność. Chociaż w tej książce skupimy uwagę na blokowaniu w stylu muteksów, futeksy są uniwersalne i służą do implementacji szeregu prymitywów synchronizacji — od muteksów po zmienne warunkowe. Futeksy stanowią również niskopoziomową funkcję jądra, której większość użytkowników nigdy nie użyje bezpośrednio — zamiast tego są opakowane w standardowe biblioteki oferujące prymitywy wyższego poziomu. Dopiero po podniesieniu maski widać mechanizm futeksów, który napędza wiele różnych rodzajów synchronizacji.

Futeksy to konstrukcja obsługiwana przez jądro, która umożliwia synchronizację procesów w przestrzeni użytkownika dla współdzielonych zdarzeń. Futeks składa się z dwóch części: usługi jądra i biblioteki użytkownika. Usługa jądra zapewnia „kolejkę oczekiwania”, która umożliwia oczekiwanie na blokadę wielu procesom. Procesy nie będą działać, jeśli jądro wyraźnie ich nie odblokuje. Umieszczenie procesu w kolejce oczekiwania wymaga (kosztownego) wywołania systemowego, dlatego należy go unikać. Z tego powodu, w przypadku braku rywalizacji, futeks działa w całości w przestrzeni użytkownika. W szczególności procesy współdzielą zmienną blokady — to wyszukana nazwa dla 32-bitowej liczby `integer`, spełniającej rolę blokady. Załóżmy, że mamy program wielowątkowy, a blokada początkowo ma wartość 1, co oznacza, że blokada jest zwolniona. Wątek przechwytuje blokadę przez wykonanie atomowej operacji „dekrementacji ze sprawdzeniem” (atomowe funkcje w Linuksie składają się z wywołania asemblerowego inline wewnątrz

funkcji C i są zdefiniowane w plikach nagłówkowych). Następnie wątek sprawdza wynik, aby przekonać się, czy blokada jest wolna. Jeśli nie była w stanie zablokowanym, nie ma problemu — wątek z powodzeniem przechwycił blokadę.

Jeśli jednak blokada jest utrzymywana przez inny wątek, to wątek starający się o blokadę musi czekać. W tym przypadku biblioteka obsługi futeksu nie wykonuje pętli, ale używa wywołania systemowego w celu umieszczenia wątku w kolejce oczekiwania w jądrze. W tej sytuacji koszt przełączenia do jądra jest uzasadniony, ponieważ wątek i tak był zablokowany. Gdy wątek zakończy operację wymagającą blokady, zwalnia ją, wykonując atomową operację „inkrementacji ze sprawdzaniem”. Następnie sprawdza wynik, aby zobaczyć, czy jakieś procesy nadal są zablokowane w kolejce oczekiwania w jądrze. Jeśli tak, informuje jądro, że może ono teraz odblokować jeden lub więcej spośród tych procesów. Jeśli nie ma rywalizacji, jądro w ogóle nie wykonuje żadnych operacji.

Muteksy w pakiecie Pthreads

W pakiecie `Pthreads` dostępnych jest kilka funkcji, które można wykorzystać do synchronizacji wątków. Podstawowy mechanizm wykorzystuje zmienną muteksa, który można zablokować lub odblokować. Muteks strzeże dostępu do każdego regionu krytycznego. Implementacje muteksów różnią się w zależności od systemu operacyjnego. W systemie Linux są one zbudowane na bazie futeksów. Wątek, który zamierza wejść do regionu krytycznego, najpierw próbuje zablokować skojarzony z nim muteks. Jeśli muteks jest odblokowany, wątek może od razu wejść do regionu krytycznego. W niepodzielnej operacji jest ustawiana blokada, dzięki czemu inne wątki nie mogą wejść do regionów krytycznych. Jeśli muteks jest już zablokowany, wątek wywołujący blokuje się do czasu, kiedy muteks zostanie odblokowany. Jeśli na ten sam muteks czeka wiele wątków, to kiedy zostanie on odblokowany, tylko jeden wątek może działać. Wątek ten ponownie blokuje muteks. Blokady te nie są obowiązkowe. Obowiązek zapewnienia poprawnego ich używania przez wątki spoczywa na programiście.

Najważniejsze wywołania związane z muteksami pokazano w tabeli 2.6. Jak można było oczekiwać, możliwe jest ich tworzenie i usuwanie. Wywołania służące do wykonania tych operacji to odpowiednio `pthread_mutex_init` i `pthread_mutex_destroy`. Można je również zablokować — za pomocą wywołania `pthread_mutex_lock`, które próbuje ustanowić blokadę i zatrzymuje swoje działanie, jeśli muteks jest już zablokowany. Istnieje również taka możliwość, że próba zablokowania muteksa się nie powiedzie i wywołanie zwróci kod o błędzie. Dzieje się tak, jeśli muteks był wcześniej zablokowany. Do tego celu służy wywołanie `pthread_mutex_trylock`. Wywołanie to pozwala wątkowi na skuteczną realizację aktywnego oczekiwania, jeśli jest ono potrzebne. Na koniec wywołanie `pthread_mutex_unlock` odblokowuje muteksa i zwalnia dokładnie jeden wątek, jeśli istnieje jeden wątek oczekujący lub większa liczba takich wątków. Muteksy mogą również mieć atrybuty, ale są one używane tylko w specjalistycznych zastosowaniach.

Tabela 2.6. Niektóre wywołania pakietu Pthreads dotyczące muteksów

Wywołanie obsługi wątku	Opis
`Pthread_muteks_init`	Tworzy muteks
`Pthread_muteks_destroy`	Niszczy istniejący muteks
`Pthread_muteks_lock`	Ustanawia blokadę muteksa lub zatrzymuje działanie wątku
`Pthread_muteks_trylock`	Ustanawia blokadę muteksa lub zwraca błąd
`Pthread_muteks_unlock`	Zwalnia blokadę

Kup ksi k Pole ksi k

Oprócz muteksów pakiet Pthreads oferuje inny mechanizm synchronizacji: **zmienne warunkowe**, które opiszemy później. Muteksy są dobre do zezwalania na dostęp lub blokowania dostępu do regionu krytycznego. Zmienne warunkowe pozwalają wątkom blokować się z powodu niespełnienia określonego warunku. Prawie zawsze te dwie metody są wykorzystywane razem. Spróbujmy teraz przyjrzeć się interakcjom pomiędzy wątkami, muteksami i zmiennymi warunkowymi.

W roli prostego przykładu ponownie rozważmy scenariusz producent-konsument: jeden wątek umieszcza elementy w buforze, a drugi je z niego pobiera. Jeśli producent odkryje, że w buforze nie ma więcej pustych miejsc, musi zablokować się do czasu, aż jakieś będą wolne. Muteksy pozwalają na wykonywanie sprawdzenia w sposób niepodzielny, tak aby inne wątki nie przeszkadzały, jednak kiedy producent odkryje, że bufor jest pełny, potrzebuje sposobu na zablokowanie się w sposób umożliwiający późniejsze przebudzenie. Można to zapewnić za pomocą zmiennych warunkowych.

Większość wywołań związanych ze zmiennymi warunkowymi pokazano w tabeli 2.7. Jak można oczekiwać, istnieją wywołania do tworzenia i usuwania zmiennych warunkowych. Mogą one mieć atrybuty — istnieją różne wywołania pozwalające na ich zarządzanie (nie pokazano ich w tabeli). Najważniejsze operacje na zmiennych warunkowych to pthread_cond_wait i pthread_cond_signal. Pierwsze blokuje wątek wywołujący do chwili, kiedy jakiś inny wątek wyśle do niego sygnał (używając drugiego z wywołań). Powody blokowania i oczekiwania nie są oczywiście częścią protokołu oczekiwania i sygnalizacji. Wątek blokujący często oczekuje, aż wątek sygnalizujący wykona jakąś pracę, zwolni jakieś zasoby lub przeprowadzi jakąś inną operację. Tylko wtedy wątek blokujący może kontynuować swoje działanie. Zmienne warunkowe pozwalają na realizację oczekiwania i blokowania w sposób niepodzielny. Wywołanie pthread_cond_broadcast jest wykorzystywane w przypadku, gdy istnieje wiele wątków, które potencjalnie wszystkie są zablokowane i oczekują na ten sam sygnał.

Tabela 2.7. Niektóre wywołania pakietu Pthreads dotyczące zmiennych warunkowych

Wywołanie obsługi wątku	Opis
Pthread_cond_init	Utworzenie zmiennej warunkowej
Pthread_cond_destroy	Zniszczenie zmiennej warunkowej
Pthread_cond_wait	Zablokowanie w oczekiwaniu na sygnał
Pthread_cond_signal	Przesłanie sygnału do innego wątku i obudzenie go
Pthread_cond_broadcast	Przesłanie sygnału do wielu wątków i obudzenie ich wszystkich

Zmienne warunkowe i muteksy zawsze są wykorzystywane wspólnie. Stosowany schemat polega na tym, że jeden wątek blokuje muteks, a kiedy nie może uzyskać tego, co potrzebuje, oczekuje na zmienną warunkową. Ostatecznie inny wątek przesyła sygnał i wątek może kontynuować działanie. Wywołanie pthread_cond_wait w niepodzielny sposób odblokowuje muteks wstrzymywany przez wątek. Następnie, po pomyślnym zwrocie sterowania, muteks zostanie ponownie zablokowany i będzie własnością wątku wywołującego. Z tego powodu muteks jest jednym z parametrów wywołania.

Warto również zwrócić uwagę, że zmienne warunkowe (w odróżnieniu od semaforów) nie mają pamięci. W przypadku wysłania sygnału do zmiennej warunkowej, na którą nie oczekuje żaden wątek, sygnał jest tracony. Programiści muszą zwracać baczną uwagę na to, aby sygnały nie były tracone.

Aby zaprezentować przykład użycia muteksów razem ze zmiennymi warunkowymi, na listingu 2.12 pokazano proste rozwiązanie problemu producent-konsument z pojedynczym buforem. Kiedy

Kup ksi k Pole ksi k

producent wypełni bufor, przed wygenerowaniem nowego elementu musi czekać do czasu, aż konsument go opróżni. Podobnie kiedy konsument usunie element, musi czekać, aż producent wygeneruje jakiś inny. Choć pokazany przykład jest bardzo prosty, ilustruje podstawowe mechanizmy. Instrukcja, która chce przenieść wątek w stan uśpienia, zawsze powinna sprawdzać, czy został spełniony warunek, ponieważ wątek może być budzony za pomocą sygnału Uniksa lub z innych powodów.

2.4.7. Monitory

W przypadku użycia semaforów i muteksów komunikacja między procesami wydaje się łatwa. Nic bardziej mylnego. Przyjrzyjmy się dokładniej kolejności operacji down przed wstawieniem lub usunięciem elementów z bufora w kodzie na listingu 2.12.

Listing 2.12. Wykorzystanie wątków w celu rozwiązania problemu producent-konsument

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000          /* ile liczb generujemy */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                 /* bufor używany pomiędzy producentem a konsumentem */
void *producer(void *ptr)       /* generowanie danych */
{ int i;
    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);       /* uzyskanie wyłącznego dostępu
                                                 do bufora */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i;                           /* umieszczenie elementu w buforze */
        pthread_cond_signal(&condc);          /* obudzenie konsumenta */
        pthread_mutex_unlock(&the_mutex);     /* zwolnienie blokady bufora */
    }
    pthread_exit(0);
}
void *consumer(void *ptr)                      /* konsumpcja danych */
{ int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);        /* uzyskanie wyłącznego dostępu
                                                  do bufora */
        while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0;                            /* pobranie elementu z bufora */
        pthread_cond_signal(&condp);           /* obudzenie producenta */
        pthread_mutex_unlock(&the_mutex);      /* zwolnienie blokady bufora */
    }
    pthread_exit(0);
}
int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
```

Kup ksi k Pole ksi k

```
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Załóżmy, że dwie operacje `down` w kodzie producenta zamieniono miejscami. W związku z tym zmienna `mutex` została poddana dekrementacji przed wykonaniem operacji `empty`, a nie po niej. Gdyby bufor był w całości wypełniony, producent by się zablokował, ustawiając zmienną `mutex` na 0. W konsekwencji przy następnej próbie dostępu konsumenta do bufora, wykonałby on operację `down` w odniesieniu do zmiennej `mutex` (teraz o wartości 0) i też by się zablokował. Oba procesy pozostałyby zablokowane na zawsze i nigdy nie wykonałyby żadnej pracy. Ta niefortunna sytuacja nazywa się **zakleszczeniem** (ang. *deadlock*). Zakleszczenia będziemy omawiać bardziej szczegółowo w rozdziale 6.

Problem ten wskazano po to, by pokazać, jak bardzo trzeba być ostrożnym podczas pracy z semaforami. Wystarczy popełnić jeden subtelny błąd i wszystko się zatrzymuje. To tak jak programowanie w języku asemblera, tylko że jeszcze trudniejsze, ponieważ błędami są sytuacje wyścigu, zakleszczenia i inne formy nieprzewidywalnych i trudnych do powtórzenia zachowań.

Aby pisanie prawidłowych programów było łatwiejsze, [Brinch Hansen, 1973] i [Hoare, 1974] zaproponowali prymityw synchronizacji wyższego poziomu, zwany **monitorem**. Ich propozycje nieco się różniły, co opisano poniżej. Monitor jest kolekcją procedur, zmiennych i struktur danych pogrupowanych ze sobą w specjalnym rodzaju modułu lub pakietu. Procesy mogą wywoływać procedury w monitorze, kiedy tylko tego chcą, ale z poziomu procedur zadeklarowanych poza monitorem nie mogą bezpośrednio korzystać z wewnętrznych struktur danych monitora. Na listingu 2.13 zilustrowano monitor napisany w wymyślonym języku Pidgin Pascal. Nie można tu użyć języka C, ponieważ monitory są konstrukcjami języka, a C ich nie posiada.

Listing 2.13. Monitor

```
monitor example
    integer i;
    condition c;
    procedure producer();
    .
    .
    .
    end;
    procedure consumer();
    .   .   .
    end;
end monitor;
```

Monitory mają ważną właściwość, dzięki której przydają się jako mechanizm implementacji wzajemnego wykluczania: w dowolnym momencie w monitorze może być aktywny tylko jeden proces. Monitory są konstrukcją języka programowania. Dzięki temu kompilator wie, że mają one specjalny charakter, i wywołania do procedur monitora może obsługiwać inaczej niż wywołania innych procedur. Zazwyczaj kiedy proces wywoła procedurę monitora, w kilku pierwszych instrukcjach procedury następuje sprawdzenie, czy w obrębie monitora jest aktywny jakiś inny proces. Jeśli tak, to proces wywołujący zostanie zawieszony do czasu opuszczenia monitora przez inny proces. Jeżeli żaden inny proces nie korzysta z monitora, proces wywołujący może do niego wejść.

Implementacja wzajemnego wykluczania dla procedur monitora leży w gestii kompilatora, ale powszechnie stosowanym sposobem jest użycie muteksa lub semafora binarnego. Ponieważ

Kup ksi k Pole ksi k

to kompilator, a nie programista zapewnia wzajemne wykluczanie, istnieje znacznie mniejsze ryzyko wystąpienia problemów. Osoba pisząca monitor nie musi wiedzieć, w jaki sposób kompilator zapewnia wzajemne wykluczanie. Wystarczy wiedzieć, że dzięki przekształceniu wszystkich regionów krytycznych w procedury monitora żadne dwa procesy nigdy jednocześnie nie wejdą do swoich regionów krytycznych.

Chociaż, jak widzieliśmy powyżej, monitory zapewniają łatwy sposób osiągnięcia wzajemnego wykluczania, to nie wystarcza. Potrzebny jest również sposób na to, by procesy się blokowały w czasie, gdy nie mogą kontynuować działania. W przypadku problemu producent-konsument można łatwo umieścić wszystkie testy sprawdzające, czy bufor jest pełny lub czy jest on pusty w procedurach monitora. Jak jednak powinien zablokować się producent, jeśli się okaże, że bufor jest pełny?

Rozwiązaniem jest wprowadzenie zmiennych warunkowych razem z dwiema operacjami, które są na nich wykonywane: `wait` i `signal`. Kiedy procedura monitora wykryje, że nie może kontynuować działania (np. producent odkryje, że bufor jest pełny), wykonuje operację `wait` na wybranej zmiennej warunkowej, np. `full`. Operacja ta powoduje zablokowanie procesu wywołującego. Pozwala ona również innemu procesowi, który wcześniej nie mógł wejść do monitora, aby teraz do niego wszedł. Zmienne warunkowe oraz wspomniane operacje omawialiśmy wcześniej, w kontekście pakietu `Pthreads`.

Inny proces, np. konsument, może obudzić swojego uśpionego partnera poprzez przesłanie sygnału z wykorzystaniem zmiennej warunkowej, na którą jego partner oczekuje. Aby uniknąć jednoczesnego występowania dwóch aktywnych procesów w monitorze, potrzebna jest reguła, która informuje o tym, co się dzieje po wykonaniu operacji `signal`. Hoare zaproponował umożliwienie działania przebudzonemu procesowi i zawieszenie drugiego z nich. Brinch Hansen zaproponował uściślenie problemu poprzez wymaganie od procesu wykonującego operację `signal` natychmiastowego opuszczenia monitora. Inaczej mówiąc, instrukcja `signal` może występować w procedurze monitora tylko jako ostatnia. My skorzystamy z propozycji Brincha Hansena, ponieważ jest ona pojęciowo prostsza, a poza tym łatwiejsza do zaimplementowania. Jeśli operacja `signal` zostanie wykonana na zmiennej warunkowej, na którą oczekuje kilka procesów, tylko jeden z nich — określony przez systemowego zarządcę procesów — zostanie wznowiony.

Na marginesie warto dodać, że istnieje trzecie rozwiązanie, którego nie zaproponował ani Hoare, ani Brinch Hansen. Polega ono na umożliwieniu procesowi wysyłającemu sygnał kontynuowania działania i pozwolenie procesowi oczekującemu na rozpoczęcie działania dopiero wtedy, gdy proces wysyłający sygnał opuści monitor.

Zmienne warunkowe nie są licznikami. Nie akumulują one sygnałów do późniejszego wykorzystania tak, jak to robią semafory. W związku z tym, jeśli zostanie wysłany sygnał do zmiennej warunkowej, na który nikt nie czeka, zostanie on utracony na zawsze. Inaczej mówiąc, operacja `wait` musi być wykonana przed operacją `signal`. Dzięki tej regule implementacja staje się znacznie prostsza. W praktyce nie jest to problem, ponieważ jeśli jest taka potrzeba, można z łatwością śledzić stan wszystkich procesów z wykorzystaniem zmiennych. Proces, który chce wysłać sygnał, może sprawdzić zmienne i zobaczyć, że ta operacja nie jest konieczna.

Szkielet problemu producent-konsument z wykorzystaniem monitorów pokazano na listingu 2.14. Rozwiązanie zaprezentowano w wymyślonym języku Pidgin Pascal. Zaleta zastosowania go w tym przypadku polega na tym, że jest on prosty i dokładnie odzwierciedla model Hoare'a i Brincha Hansena.

Kup ksi k Pole ksi k

Listing 2.14. Szkielet rozwiązania problemu producent-konsument z wykorzystaniem monitorów. Tylko jedna procedura monitora jest aktywna w danym momencie. Bufor zawiera N gniazd

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;

    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume item(item)
    end
end;
```

Można by sądzić, że operacje `wait` i `signal` są podobne do operacji `sleep` i `wakeup`, które omawialiśmy wcześniej i które powodowały sytuację wyścigu. To prawda, one są bardzo podobne, ale z jedną zasadniczą różnicą: operacje `sleep` i `wakeup` zawodzą, kiedy jeden proces próbuje przejść w stan uśpienia, natomiast drugi próbuje go obudzić. W przypadku monitorów to nie może się zdarzyć. Automatyczne wzajemne wykluczanie procedur monitora gwarantuje, że jeśli np. producent wewnątrz monitora odkryje, że bufor jest pełny, to będzie mógł wykonać operację `wait` bez obawy o to, że program szeregujący zechce przełączyć się do konsumenta bezpośrednio przed zakończeniem wykonywania operacji `wait`. Konsument nie zostanie nawet wpuszczony do monitora, zanim operacja `wait` się nie zakończy, a producent zostanie oznaczony jako niezdolny do działania.

Chociaż Pidgin Pascal jest językiem wymyślonym, istnieją rzeczywiste języki programowania obsługujące monitory. Nie zawsze jednak są one zaimplementowane w takiej formie, jaką zaproponowali Hoare i Brinch Hansen. Jednym z takich języków jest Java. To język obiektowy obsługujący

Kup ksi k Pole ksi k

wątki na poziomie użytkownika. Pozwala również na grupowanie metod (procedur) w klasy. Dzięki dodaniu słowa kluczowego `synchronized` w deklaracji metody Java gwarantuje, że kiedy dowolny wątek zacznie uruchamiać tę metodę, żaden inny wątek nie będzie mógł uruchomić żadnej innej metody tego obiektu zadeklarowanej ze słowem kluczowym `synchronized`. Bez słowa kluczowego `synchronized` nie ma gwarancji przeplatania.

Rozwiązanie problemu producent-konsument z wykorzystaniem monitorów w Javie pokazano na listingu 2.15. Rozwiązanie składa się z czterech klas. Klasa zewnętrzna — `ProducerConsumer` — tworzy i uruchamia dwa wątki — `p` i `c`. Druga i trzecia klasa, odpowiednio `producer` i `consumer`, zawierają kod producenta i konsumenta. Wreszcie — klasa `our_monitor` jest monitorem. Zawiera dwa zsynchronizowane wątki wykorzystywane do wstawiania elementów do współdzielonego bufora i do pobierania ich z niego. W odróżnieniu od poprzednich przykładów na listingu pokazano kompletny kod operacji `insert` i `remove`.

Listing 2.15. Rozwiązanie problemu producent-konsument w Javie

```
public class ProducerConsumer {
    static final int N = 100;                 // stała określająca rozmiar bufora
    static producer p = new producer( );      // utworzenie egzemplarza nowego
                                              // wątku producenta
    static consumer c = new consumer( );      // utworzenie egzemplarza nowego
                                              // wątku producenta
    static our_monitor mon = new our_monitor( ); // utworzenie egzemplarza nowego
                                                 // monitora
    public static void main(String args[ ]) {
        p.start( );                           // rozpoczęcie wątku producenta
        c.start( );                           // rozpoczęcie wątku konsumenta
    }
    static class producer extends Thread {
        public void run( ) {                  // metoda run zawiera kod wątku
            int item;
            while (true) {                    // pętla producenta
            item = produce_item();
            mon.insert(item);
        }
      }
      private int produce_item( ) { ... } // tworzenie elementu
    }
    static class consumer extends Thread {
      public void run( ) {                    // metoda run zawiera kod wątku
        int item;
        while (true) {                        // pętla konsumenta
            item = mon.remove( );
            consume_item(item);
        }
      }
      private void consume_item(int item) { ... } // skonsumowanie elementu
    }
    static class our_monitor {                        // to jest monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0;        // liczniki i indeksy
        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( ); // jeśli bufor jest pełny, wątek przechodzi
                                            // w stan uśpienia
            buffer [hi] = val; // wstawienie elementu do bufora
            hi = (hi + 1) % N; // miejsce, w którym będzie umieszczony następny element
```

Kup książkę Poleć książkę

```
            count = count + 1; // teraz w buforze znajduje się o jeden element więcej
            if (count == 1) notify( ); // obudzenie konsumenta, jeśli był uśpiony
        }
        public synchronized int remove( ) {
            int val;
            if (count == N) go_to_sleep( ); // jeśli bufor jest pusty, wątek przechodzi
                                            // w stan uśpienia
            val = buffer [lo];   // pobranie elementu z bufora
            lo = (lo + 1) % N;   // miejsce, z którego będzie pobrany następny element
            count = count - 1;   // teraz w buforze znajduje się o jeden element mniej
            if (count == N - 1) notify( ); // obudzenie producenta, jeśli był uśpiony
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Wątki producenta i konsumenta są funkcjonalnie identyczne do ich odpowiedników we wszystkich naszych poprzednich przykładach. Producent zawiera pętlę nieskończoną, w której są generowane dane umieszczane później we wspólnym buforze. Konsument również zawiera pętlę nieskończoną, w której są pobierane dane ze wspólnego bufora i wykonywane na nich pewne operacje.

Interesującym fragmentem tego programu jest klasa `our_monitor`, która zawiera bufor, zmienne administracyjne oraz dwie zsynchronizowane metody. Kiedy producent jest aktywny wewnątrz metody `insert`, wie na pewno, że konsument nie może być aktywny wewnątrz metody `remove`. W związku z tym można bezpiecznie zaktualizować zmienne i bufor bez obaw o wystąpienie sytuacji wyścigu. Zmienna `count` kontroluje liczbę elementów znajdujących się w buforze. Może ona przyjąć dowolną wartość od 0 do wartości `N-1` włącznie. Zmienna `lo` jest indeksem gniazda bufora, z którego ma być pobrany następny element. Na podobnej zasadzie zmienna `hi` jest indeksem gniazda bufora, gdzie ma być umieszczony następny element. Dozwolona jest sytuacja, w której `lo = hi`. Oznacza to, że w buforze znajduje się 0 lub `N` elementów. Wartość zmiennej `count` mówi o tym, który przypadek zachodzi.

Metody zsynchronizowane w Javie różnią się od klasycznych monitorów w zasadniczy sposób: w Javie nie ma wbudowanych zmiennych warunkowych. Zamiast nich są dwie procedury: `wait` i `notify`, które stanowią odpowiedniki operacji `sleep` i `wakeup`. Różnica polega na tym, że kiedy są używane wewnątrz metod zsynchronizowanych, nie są przedmiotem wyścigu. Teoretycznie metodę `wait` można przerwać. Do tego właśnie służy otaczający ją kod. W języku Java obsługa wyjątków musi być jawna. Dla naszych celów wyobraźmy sobie, że metoda `go_to_sleep` przenosi wątek do stanu uśpienia.

Dzięki temu, że wzajemne wykluczanie regionów krytycznych w przypadku zastosowania monitorów jest automatyczne, możliwość popełnienia błędów w programowaniu współbieżnym jest znacznie mniejsza niż w przypadku wykorzystania semaforów. Pomimo to monitory także mają pewne wady. Nie bez powodu nasze dwa przykłady monitorów napisano w języku Pidgin Pascal, a nie w języku C, jak inne przykłady w tej książce.

Jak powiedzieliśmy wcześniej, monitory są konstrukcją języka programowania. Kompilator musi je rozpoznać i w jakiś sposób zorganizować wzajemne wykluczanie. W językach C, Pascalu i większości innych języków nie ma monitorów, zatem nie można oczekiwać od kompilatorów tych języków wymuszania reguł wzajemnego wykluczania. W rzeczywistości kompilator nie ma możliwości stwierdzenia, które procedury były w monitorach, a które nie.

Kup ksi k Pole ksi k

W wymienionych językach nie ma również semaforów, ale dodanie semaforów jest łatwe: wystarczy dodać do biblioteki dwie krótkie procedury asemblerowe służące do wydawania wywołań systemowych `up` i `down`. Kompilatory nie muszą nawet wiedzieć, że takie wywołania istnieją. Oczywiście systemy operacyjne muszą mieć informacje o semaforach. Jeśli jednak dysponujemy systemem operacyjnym bazującym na semaforach, to możemy dla nich napisać programy użytkowe w językach C i C++ (lub nawet w języku asemblera, jeśli ktoś ma skłonności do masochizmu). W przypadku monitorów potrzebujemy języka, który ma tę konstrukcję wbudowaną.

Innym problemem dotyczącym monitorów, a także semaforów, jest to, że zostały one zaprojektowane do rozwiązywania problemu wzajemnego wykluczania dla jednego lub kilku procesorów mających dostęp do wspólnej pamięci. Dzięki umieszczeniu semaforów we wspólnej pamięci i zabezpieczeniu ich za pomocą instrukcji `TSL` lub `XCHG` możemy uniknąć wyścigu. W przypadku systemu rozproszonego, składającego się z wielu procesorów połączonych w sieci lokalnej, gdzie każdy dysponuje prywatną pamięcią, prymitywy te stają się nieodpowiednie. Wniosek jest następujący: semafory są zbyt niskopoziomowe, a z monitorów, z wyjątkiem kilku języków programowania, nie można korzystać. Żaden z prymitywów nie ze-zwala również na wymianę informacji pomiędzy maszynami. Potrzebne jest inne rozwiązanie.

2.4.8. Przekazywanie komunikatów

Tym innym rozwiązaniem jest **przekazywanie komunikatów**. W tej metodzie komunikacji między procesami wykorzystywane są dwa prymitywy: `send` i `receive`, które podobnie do semaforów i w odróżnieniu od monitorów są wywołaniami systemowymi, a nie konstrukcjami języka. W związku z tym można je łatwo zaimplementować w postaci procedur bibliotecznych następującej postaci:

```
send(destination, &message);
```

oraz:

```
receive(source, &message);
```

Pierwsza wysyła komunikat do określonej lokalizacji docelowej, natomiast druga odbiera komunikat z określonego źródła (lub z dowolnego, jeśli odbiorcy jest wszystko jedno). Jeśli nie jest dostępny żaden komunikat, odbiorca może się zablokować do czasu nadejścia jakiegoś komunikatu. Może on również natychmiast zwrócić sterowanie i przekazać kod błędu.

Problemy projektowe systemów przekazywania komunikatów

Z systemami przekazywania komunikatów związanych jest wiele istotnych problemów projektowych, które nie występują w przypadku semaforów albo monitorów, zwłaszcza jeśli komunikujące się ze sobą procesy działają na różnych maszynach połączonych przez sieć. Przykładowo podczas przesyłania przez sieć komunikaty mogą być utracone. W celu zabezpieczenia się przed utratą komunikatów nadawca i odbiorca mogą ustalić, że natychmiast po odebraniu komunikatu odbiorca prześle specjalny **komunikat potwierdzający**. Jeśli odbiorca nie odbierze potwierdzenia w ciągu określonego przedziału czasu, ponawia transmisję komunikatu.

Rozważmy teraz, co się stanie, jeśli komunikat zostanie odebrany prawidłowo, ale potwierdzenie wysłane do nadawcy zostanie utracone. Nadawca ponowi transmisję komunikatu, w związku z czym odbiorca otrzyma go dwukrotnie. Istotne znaczenie ma to, aby odbiorca potrafił odróżnić nowy komunikat od ponownej transmisji starego. Zazwyczaj problem jest rozwiązywany poprzez

Kup ksi k Pole ksi k

umieszczenie kolejnego numeru porządkowego w każdym nowym komunikacie. Jeśli odbiorca otrzyma komunikat o takim samym numerze porządkowym, jaki miał poprzedni komunikat, będzie wiedział, że komunikat jest duplikatem, który można zignorować. Pomyślna komunikacja w warunkach zawodnego przekazywania komunikatów stanowi zasadniczą część badań nad sieciami komputerowymi. Więcej informacji na ten temat można znaleźć w [Tanenbaum et al., 2020].

Systemy komunikatów muszą również rozwiązać problem nadawania nazw procesom. Powinny one być takie, aby specyfikacja procesów w wywołaniach `send` i `receive` była jednoznaczna. W systemach komunikatów problemem jest również uwierzytelnianie: w jaki sposób klient może stwierdzić, że komunikuje się z rzeczywistym serwerem plików, a nie z oszustem?

Na drugim końcu spektrum są problemy projektowe, które mają znaczenie w przypadku, gdy nadawca i odbiorca działają na tej samej maszynie. Jednym z takich problemów jest wydajność. Kopiowanie komunikatów z jednego procesu do innego zawsze jest wolniejsze niż wykonywanie operacji na semaforach lub wejście do monitora. Przeprowadzono wiele prac mających na celu zapewnienie odpowiedniej wydajności przekazywania komunikatów.

Rozwiązanie problem producent-konsument za pomocą przekazywania komunikatów

Spróbujmy teraz przyjrzeć się temu, w jaki sposób można rozwiązać problem producent-konsument z wykorzystaniem przekazywania komunikatów i bez współdzielonej pamięci. Rozwiązanie zaprezentowano na listingu 2.16. Zakładamy, że wszystkie komunikaty są tego samego rozmiaru, a komunikaty przesłane, ale jeszcze nie odebrane, są automatycznie buforowane przez system operacyjny. W tym rozwiązaniu wykorzystywana jest całkowita liczba *N* komunikatów. To analogia do *N* miejsc w buforze umieszczonym we współdzielonej pamięci. Konsument rozpoczyna działanie poprzez wysłanie *N* pustych komunikatów do producenta. Za każdym razem, kiedy producent ma element do przekazania konsumentowi, pobiera pusty komunikat i przesyła pełny. W ten sposób całkowita liczba komunikatów w systemie pozostaje stała w czasie. W związku z tym można je zapisać w określonej ilości pamięci, która jest z góry znana.

Listing 2.16. Rozwiązanie problemu producent-konsument z wykorzystaniem N komunikatów

```
#define N 100                       /* liczba miejsc w buforze */
void producer(void)
{
    int item;
    message m;                      /* bufor komunikatów */
    while (TRUE) {
        item = produce_item( );     /* wygenerowanie wartości do umieszczenia
                                       w buforze */
        receive(consumer, &m);      /* oczekiwanie na nadejście pustego komunikatu */
        build_message(&m, item);    /* skonstruowanie komunikatu do wysłania */
        send(consumer, &m);         /* wysłanie elementu do konsumenta */
    }
}
void consumer(void)
{
    int item, i;
    message m;
    for (i = 0; i < N; i++) send(producer, &m); /* wysłanie N pustych komunikatów */
    while (TRUE) {
        receive(producer, &m);      /* pobranie komunikatu zawierającego element */
```

Kup ksi k Pole ksi k

```
        item = extract_item(&m);    /* wyodrębnienie elementu z komunikatu */
        send(producer, &m);         /* wysłanie pustej odpowiedzi */
        consume_item(item);         /* wykonanie operacji z elementem */
    }
}
```

Jeśli producent pracuje szybciej niż konsument, to wszystkie komunikaty się zapełnią. W oczekiwaniu na konsumenta producent będzie zablokowany do momentu, kiedy nadejdzie pusty komunikat. Jeśli konsument pracuje szybciej, zachodzi sytuacja odwrotna: wszystkie komunikaty zostają opróżnione w oczekiwaniu, aż producent je zapełni. Konsument będzie zablokowany do momentu, kiedy nadejdzie pełny komunikat.

Przy przekazywaniu komunikatów jest możliwych wiele wariantów. Na początek przyjrzyjmy się sposobowi adresowania komunikatów. Jednym ze sposobów jest przypisanie każdemu procesowi unikatowego adresu i adresowanie komunikatów za pomocą procesów. Innym sposobem jest utworzenie nowej struktury danych, zwanej **skrzynką pocztową**. Skrzynka pocztowa jest miejscem przeznaczonym na buforowanie określonej liczby komunikatów, zwykle określonych w momencie tworzenia skrzynki.

W przypadku użycia skrzynek pocztowych parametrami adresowymi w wywołaniach `send` i `receive` są skrzynki pocztowe, a nie procesy. Kiedy proces podejmuje próbę wysłania komunikatu do pustej skrzynki pocztowej, jest zawieszany do momentu, kiedy komunikat zostanie pobrany ze skrzynki i powstanie w niej miejsce na nowy.

W przypadku problemu producent-konsument, zarówno producent, jak i konsument tworzą skrzynki pocztowe wystarczająco duże, by pomieścić N komunikatów. Producent wysyła komunikaty zawierające dane do skrzynki pocztowej konsumenta, a konsument wysyła puste komunikaty do skrzynki pocztowej producenta. W przypadku użycia skrzynek pocztowych mechanizm buforowania jest czytelny: docelowa skrzynka pocztowa zawiera komunikaty, które zostały wysłane do procesu docelowego, ale jeszcze nie zostały zaakceptowane.

Przeciwległym ekstremum do posiadania skrzynek pocztowych jest całkowite wyeliminowanie buforowania. W przypadku zastosowania tego podejścia, jeśli zostanie wykonana operacja `send` przed wykonaniem operacji `receive`, proces wysyłający będzie zablokowany do chwili wykonania operacji `receive`. W tym momencie komunikat może być skopiowany bezpośrednio od nadawcy do odbiorcy bez buforowania. Na podobnej zasadzie, jeśli najpierw zostanie wykonana operacja `receive`, odbiorca jest blokowany do momentu wykonania operacji `send`. Strategię tę często określa się terminem **rendezvous** (z fr. spotkanie). Jest ona łatwiejsza do zaimplementowania od mechanizmu z buforowaniem, ale mniej elastyczna, ponieważ na-dawca i odbiorca są zmuszeni do działania w trybie naprzemiennym.

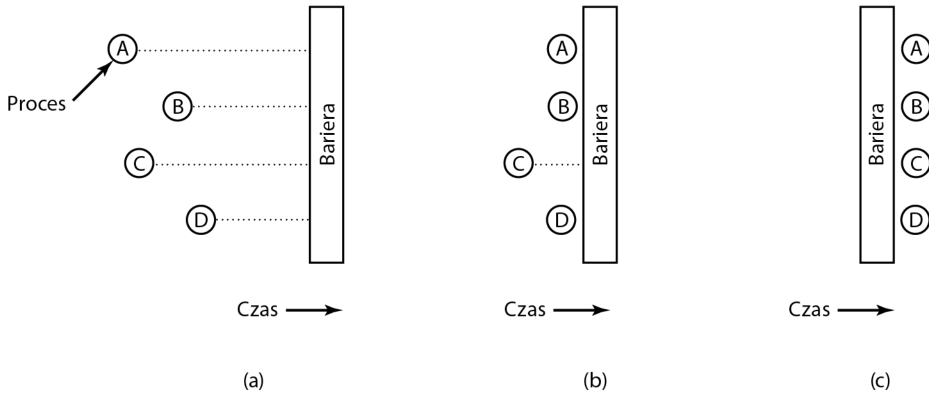
Przekazywanie komunikatów jest mechanizmem powszechnie stosowanym w systemach programowania równoległego; np. jednym ze znanych systemów przekazywania komunikatów jest **MPI** (*Message-Passing Interface*). Jest on powszechnie wykorzystywany do obliczeń naukowych. Więcej informacji na ten temat można znaleźć w następujących pozycjach: [Gropp et al., 1994], [Snir et al., 1996].

2.4.9. Bariery

Ostatni mechanizm synchronizacji, który omówimy, jest przeznaczony w większym stopniu dla grup procesów niż dla sytuacji dwóch procesów typu producent-konsument. Niektóre aplikacje są podzielone na fazy i przestrzegają reguły, według której proces nie może przejść do następnej fazy, jeśli wszystkie procesy nie są gotowe do przejścia do następnej fazy. Takie działanie można uzyskać dzięki umieszczeniu **bariery** na końcu każdej fazy. Kiedy proces osiągnie barierę, jest

Kup ksi k Pole ksi k

blokowany do czasu, aż wszystkie procesy osiągną barierę. Pozwala to grupom procesów na synchronizację. Działanie bariery zilustrowano na rysunku 2.15.



Rysunek 2.15. Wykorzystanie bariery: (a) procesy zbliżające się do bariery; (b) wszystkie procesy oprócz jednego zablokowane na barierze; (c) kiedy ostatni proces dotrze do bariery, wszystkie są przepuszczane

Na rysunku 2.15(a) widać cztery procesy zbliżające się do bariery. Oznacza to, że procesy te wykonują obliczenia i jeszcze nie osiągnęły końca bieżącej fazy. Po pewnym czasie pierwszy proces kończy obliczenia pierwszej fazy. Następnie uruchamia prymityw bariery — ogólnie rzecz biorąc, poprzez wywołanie procedury bibliotecznej. Następnie proces jest zawieszany. Nieco później drugi, a następnie trzeci proces kończą pierwszą fazę i także uruchamiają prymityw bariery. Sytuację tę pokazano na rysunku 2.15(b). Na koniec, kiedy ostatni proces — C — dotrze do bariery, wszystkie procesy są zwalniane, tak jak pokazano na rysunku 2.15(c).

Jako przykład problemu wymagającego barier rozważmy typowy problem relaksacji znany z fizyki lub inżynierii. Zwykle mamy macierz, która zawiera pewne wartości początkowe. Wartości te mogą reprezentować np. temperatury w różnych punktach arkusza metalu. Przypuśćmy, że chcemy obliczyć, ile czasu upłynie, aż efekt podgrzewania płomieniem jednego narożnika arkusza rozprzestrzeni się na cały arkusz.

Począwszy od bieżących wartości macierzy wartości, wykonywane jest przekształcenie, w wyniku którego otrzymujemy drugą wersję macierzy. Stosując np. prawa termodynamiki, obliczamy temperatury punktów po upływie czasu ΔT. Następnie proces jest powtarzany i w ten sposób, w miarę nagrzewania się arkusza, są obliczane temperatury w punktach próbnych. Wraz z upływem czasu algorytm generuje serię macierzy — każda odpowiada określonemu punktowi w czasie.

Wyobraźmy sobie teraz, że macierz jest bardzo duża (np. 1 milion$\times$1 milion). W związku z tym do przyspieszenia obliczeń są potrzebne procesy współbieżne (ewentualnie w systemie wieloprocesorowym). Różne procesy działają na różnych częściach macierzy. W ich wyniku są obliczane nowe elementy na podstawie starych, zgodnie z prawami fizyki. Jednak żaden proces nie może rozpocząć się w iteracji $n + 1$ do czasu, aż zakończy się iteracja n — tzn. do czasu, aż wszystkie procesy zakończą swoje bieżące operacje. Sposobem na osiągnięcie tego celu jest zaprogramowanie każdego procesu w taki sposób, by wykonał operację bariery po zakończeniu swojej części bieżącej operacji. Kiedy wszystkie procesy zakończą działanie, będzie gotowa nowa macierz (stanowiąca dane wejściowe do kolejnej iteracji), a wszystkie procesy zostaną jednocześnie zwolnione i będą mogły rozpocząć nową iterację.

Kup ksi k Pole ksi k

Warto wspomnieć, że specjalne bariery niskiego poziomu są powszechnie używane również do synchronizacji operacji na pamięci. Takie bariery, nazywane **barierami pamięci** lub **ogrodzeniami pamięci**, wymuszają kolejność dającą gwarancję zakończenia wszystkich operacji na pamięci (odczytu lub zapisu) rozpoczętych przed instrukcją bariery przed operacjami na pamięci wykonanymi po instrukcji bariery. Ma to istotne znaczenie, ponieważ nowoczesne procesory wykonują instrukcje nie po kolei, co może powodować problemy. Jeśli np. instrukcja 2. nie zależy od wyniku instrukcji 1., procesor może rozpocząć jej wykonywanie z wyprzedzeniem. Nowoczesne procesory bowiem są superskalarne i składają się z wielu jednostek wykonawczych do wykonywania obliczeń i równoległego dostępu do pamięci. W gruncie rzeczy, jeśli instrukcja 1. zajmuje dużo czasu, instrukcja 2. może zostać zakończona przed nią, a następnie procesor może rozpocząć wykonywanie instrukcji 3. Rozważmy teraz sytuację, w której jeden wątek czeka na inny z wykorzystaniem aktywnego oczekiwania:

WĄTEK 1.

```
while (turn != 1) { } /* pętla*/
printf("%d\n", x);
```

WĄTEK 2.

```
x = 100;
turn = 1;
```

Gdyby początkowo zachodził warunek `turn == 0` i wszystkie instrukcje były wykonywane po kolei, program wypisałby wartość 100. Gdyby jednak instrukcje w wątku 2. zostały wykonane nie po kolei, aktualizacja zmiennej `turn` nastąpiłaby przed zmienną `x`, a wyświetlona wartość mogłaby być starszą wersją zmiennej `x`. Na podobnej zasadzie mogłaby się zmienić kolejność instrukcji z wątku 1., co spowodowałoby odczytanie zmiennej `x` przed wykonaniem testu w wierszu powyżej. Rozwiązaniem w obu przypadkach jest umieszczenie pomiędzy dwoma wierszami wątku instrukcji bariery.

Nawiasem mówiąc, bariery pamięci często odgrywają ważną rolę w łagodzeniu luk w zabezpieczeniach procesora, które są powszechnie określane jako luki typu *transient execution*. Napastnik może w tym wypadku wykorzystać to, że procesory wykonują instrukcje nie po kolei. Od ujawnienia problemów Meltdown i Spectre, co miało miejsce w 2018 roku, pojawiły się informacje o wielu tego rodzaju lukach. Ponieważ luki *transient execution*, ogólnie rzecz biorąc, wpływają również na system operacyjny, w rozdziale 9. pokrótce przyjrzymy się atakom z ich wykorzystaniem.

2.4.10. Inwersja priorytetów

Wcześniej w tym rozdziale wspomnieliśmy o problemie inwersji priorytetów, klasycznym problemie znanym już w latach siedemdziesiątych ubiegłego wieku. Teraz przyjrzymy się mu bardziej szczegółowo.

Słynny przypadek wystąpienia problemu inwersji priorytetów miał miejsce w projekcie Mars w 1997 roku. Dzięki imponującym wysiłkom inżynierów NASA zdołała wylądować na Czerwonej Planecie niewielkim robotem-łazikiem, którego zadaniem było przesłanie na Ziemię wielu interesujących informacji. Pojawił się jednak pewien problem. Transmisje radiowe Pathfindera przestały przesyłać dane w trybie ciągłym, a ich ponowne uruchomienie wymagało zrestartowania systemu. Okazało się, że trzy wątki wzajemnie sobie przeszkadzały. Do przekazywania informacji między różnymi komponentami w Pathfinderze użyto formy pamięci współdzielonej zwanej „szyną informacyjną". Wątek o niskim priorytecie okresowo wykorzystywał tę szynę do przekazywania zebranych danych meteorologicznych (rodzaj marsjańskiego raportu pogodowego). W tym czasie, również okresowo, sięgał do niej wątek o wysokim priorytecie odpowiedzialny za zarządzanie szyną. Aby zapobiec jednoczesnemu dostępowi obu wątków do współdzielonej pamięci, dostęp do niej był kontrolowany w oprogramowaniu łazika przez muteks. Trzeci wątek o średnim priorytecie był odpowiedzialny za komunikację i nie potrzebował muteksa.

Kup ksi k Pole ksi k

Inwersja priorytetów następowała, gdy wątek o niskim priorytecie zajmujący się gromadzeniem danych meteorologicznych w czasie, gdy posiadał muteks, został wywłaszczony przez wątek komunikacyjny o średnim priorytecie. Po pewnym czasie musiał się uruchomić wątek o wysokim priorytecie, ale natychmiast się blokował, ponieważ nie mógł zdobyć muteksa. Długo działający wątek o średnim priorytecie wykonywał się nadal, tak jakby miał wyższy priorytet niż wątek magistrali informacyjnej.

Istnieją różne sposoby rozwiązania problemu inwersji priorytetów. Najprostszym jest wyłączenie w regionie krytycznym wszystkich przerwań. Jak wspomniano wcześniej, w wypadku programów użytkownika takie działanie nie jest pożądane: co się stanie, jeśli zapomną je ponownie włączyć?

Innym rozwiązaniem, znanym jako **pułap priorytetu** (ang. *priority ceiling*), jest powiązanie priorytetu z muteksem i przypisanie do wartości pułapu priorytetu procesu, który posiada muteks. Dopóki żaden proces, który musi przechwycić muteks, nie ma wyższego priorytetu niż pułap priorytetu, inwersja nie jest możliwa.

Trzecim sposobem jest **dziedziczenie priorytetów**. W tym rozwiązaniu zadanie o niskim priorytecie, posiadające muteks, tymczasowo dziedziczy priorytet zadania o wysokim priorytecie, które próbuje go uzyskać. W tym wypadku również żadne zadanie o średnim priorytecie nie będzie w stanie wywłaszczyć zadania posiadającego muteks. Właśnie tę ostatnią technikę ostatecznie zastosowano do rozwiązania problemów Pathfindera.

Nowoczesne systemy operacyjne, takie jak Microsoft Windows, korzystają z techniki *random boosting* (dosłownie: losowe zwiększanie), co w gruncie rzeczy sprowadza się do okresowego „rzutu kostką" i nadaniu losowym wątkom utrzymującym muteks wysokiego priorytetu, dopóki nie opuszczą krytycznego regionu.

2.4.11. Unikanie blokad: odczyt-kopiowanie-aktualizacja

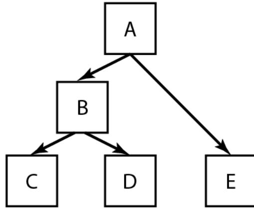
Najszybsze blokady to całkowity brak blokad. A ten oznacza również brak ryzyka inwersji priorytetów. Pytanie brzmi: czy bez blokowania możemy pozwolić na równoczesny dostęp do odczytu i zapisu wspólnej struktury danych? Ogólnie odpowiedź jest oczywiście przecząca. Wyobraźmy sobie wątek A sortujący tablicę liczb w czasie, kiedy wątek B oblicza średnią. Ponieważ A przemieszcza wartości wewnątrz tablicy, B może napotkać pewne wartości wielokrotnie, natomiast innych nie napotyka wcale. Wynik takiej operacji jest nieprzewidywalny, ale prawie na pewno będzie błędny.

Jednak w niektórych przypadkach możemy pozwolić procesowi piszącemu na zaktualizowanie struktury danych, mimo że inne procesy nadal jej używają. Sztuka polega na zagwarantowaniu, by każdy proces czytający mógł odczytać starą lub nową wersję danych, ale nigdy nie odczytywał jakiegoś dziwnego połączenia starej i nowej wersji. Jako przykład rozważmy drzewo pokazane na rysunku 2.16.

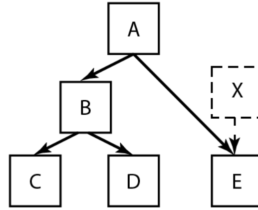
Procesy czytające (czytelnicy) przeglądają drzewo od korzenia do liści. W górnej połowie rysunku dodajemy nowy węzeł X. Aby to zrobić, tworzymy węzeł tuż przed tym, zanim stanie się on widoczny w drzewie: inicjujemy wszystkie wartości w węźle X, włącznie ze wskaźnikami do jego potomków. Następnie za pomocą atomowej operacji zapisu ustalamy, że węzeł X jest potomkiem węzła A. Żaden czytelnik nigdy nie odczyta niespójnej wersji. Następnie, w dolnej części rysunku, usuwamy węzły B i D. Najpierw ustalamy, że lewostronnym potomkiem węzła A jest C. Wszystkie procesy-czytelnicy, które były w węźle A, przejdą do węzła C i nigdy nie zobaczą węzłów B lub D. Innymi słowy, będą widzieć wyłącznie nową wersję. Na podobnej zasadzie wszystkie procesy-czytelnicy, które w tym momencie są w węźle B lub D, będą przeglądały wcześniejsze wskaźniki struktury danych i będą widziały tylko starą wersję. Wszystko działa poprawnie.

Kup ksi k Pole ksi k

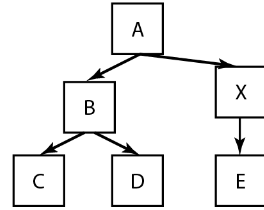
Dodanie węzła:



(a) Drzewo w postaci początkowej

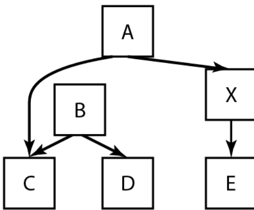


(b) Zainicjowanie węzła X o podłączenie węzła E do węzła X. Operacja nie ma wpływu na czytelników węzłów A i E

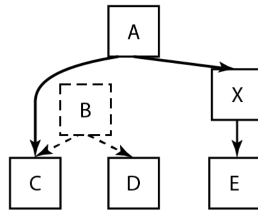


(c) Po całkowitym zainicjowaniu węzła X podłączamy X do A. Czytelnicy, którzy aktualnie są w węźle E, muszą odczytać starą wersję, natomiast czytelnicy w węźle A pobiorą nową wersję drzewa

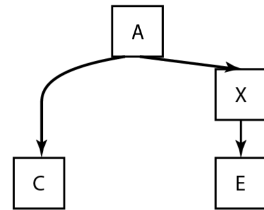
Usuwanie węzłów:



(d) Rozdzielenie B od A. Zwróćmy uwagę, że w B nadal mogą być czytelnicy. Wszystkie procesy-czytelnicy w B będą widziały starą wersję drzewa, natomiast wszystkie procesy-czytelnicy w A będą widziały nową wersję



(e) Oczekiwanie do uzyskania pewności, że wszystkie procesy-czytelnicy opuściły węzły B i C. Do tych węzłów nie będzie już dostępu



(f) Teraz można bezpiecznie usunąć węzły B i D

Rysunek 2.16. Operacja odczyt-kopiowanie-aktualizacja: wstawienie węzła do drzewa, a następnie usunięcie gałęzi — bez blokad

Nigdy nie ma potrzeby, aby cokolwiek blokować. Usunięcie węzłów *B* i *D* działa bez blokowania struktury danych dlatego, że operacja **RCU** (odczyt-kopiowanie-aktualizacja — ang. *Read-Copy-Update*) oddziela od siebie dwie fazy aktualizacji: usunięcie i odtworzenie (ang. *reclamation*).

Jest jednak pewien problem. Tak długo, jak nie mamy pewności, że nie ma więcej czytelników węzłów *B* lub *D*, nie możemy ich usunąć. Ile zatem powinniśmy czekać? Minutę? Dziesięć minut? Musimy czekać, aż ostatni czytelnik opuści te węzły. W operacjach RCU dokładnie określa się maksymalny czas, przez który czytelnik może utrzymywać referencję do struktury danych. Po upływie tego okresu możemy bezpiecznie odzyskać pamięć. W szczególności procesy czytelnicy uzyskują dostęp do struktury danych w tzw. **sekcji krytycznej strony odczytu** (ang. *read-side critical section*), która może zawierać dowolny kod, pod warunkiem że nie wykonuje on blokady (ang. *lock*) lub uśpienia (ang. *sleep*). W takim przypadku dokładnie znamy maksymalny czas oczekiwania. Ustalamy zwłaszcza **okres karencji** (ang. *grace period*) jako dowolny czas, o którym wiemy, że każdy wątek jest poza sekcją krytyczną strony odczytu co najmniej raz. Wszystko będzie dobrze, jeśli przed odzyskaniem pamięci będziemy czekać przez okres równy co najmniej karencji. Ponieważ kod w sekcji krytycznej strony odczytu nie może wykonywać operacji `lock` ani `sleep`, prosty warunek polega na poczekaniu tak długo, aż wszystkie wątki zrealizują przełączenie kontekstu.

Kup ksi k Pole ksi k

Struktury danych RCU nie są zbyt powszechne w procesach użytkownika, ale dość popularne w jądrach systemów operacyjnych dla struktur danych, do których dostęp uzyskuje wiele wątków i które wymagają wysokiej wydajności. W jądrze Linuksa interfejs API RCU wykorzystano wiele tysięcy razy, w większości jego podsystemów. Stos sieciowy, system plików, sterowniki i mechanizmy zarządzania pamięcią korzystają z RCU do równoległego odczytu i zapisu.

2.5. SZEREGOWANIE

Kiedy w komputerze jest wykorzystywana wieloprogramowość, często wiele procesów lub wątków jednocześnie rywalizuje o procesor. Sytuacja taka występuje w przypadku, kiedy dwa lub większa liczba procesów jednocześnie znajdują się w stanie gotowości. Jeśli tylko jeden procesor jest dostępny, trzeba dokonać wyboru, który proces ma się uruchomić w następnej kolejności. Ta część systemu operacyjnego, która dokonuje wyboru, nazywa się **programem szeregującym** (ang. *scheduler*), a algorytm, który ona wykorzystuje, nazywa się **algorytmem szeregowania**. Tematy te będą przedmiotem kolejnych podrozdziałów.

Wiele problemów, które dotyczą szeregowania procesów, dotyczy również szeregowania wątków, choć niektóre różnią się pomiędzy sobą. Jeśli jądro zarządza wątkami, szeregowanie zwykle jest wykonywane na poziomie wątków. W tym przypadku nie ma wielkiego znaczenia lub zupełnie nie ma znaczenia to, do którego procesu należy określony wątek. Najpierw skoncentrujemy się na problemach szeregowania, które dotyczą zarówno procesów, jak i wątków. Później jawnie zajmiemy się szeregowaniem wątków oraz pewnymi unikatowymi problemami, jakie są z tym związane. W rozdziale 8. zajmiemy się układami wielordzeniowymi.

2.5.1. Wprowadzenie do szeregowania

W starych czasach systemów wsadowych, kiedy dane wejściowe miały formę obrazów kart na taśmie magnetycznej, algorytm szeregowania był prosty: polegał na uruchomieniu następnego zadania na taśmie. W przypadku systemów wieloprogramowych algorytm szeregowania był bardziej złożony, ponieważ na obsługę oczekiwało wielu użytkowników. Niektóre komputery mainframe w dalszym ciągu łączą usługi wsadowe z systemami z podziałem czasu. W związku z tym program szeregujący musi zdecydować, czy w następnej kolejności powinien być obsłużony interaktywny użytkownik przy terminalu, czy zadanie wsadowe (nawiasem mówiąc, zadanie wsadowe może być żądaniem uruchomienia wielu programów po kolei, ale dla potrzeb tego podrozdziału przyjmiemy, że jest to po prostu żądanie uruchomienia pojedynczego programu). Ponieważ czas procesora jest deficytowym zasobem na tych maszynach, dobry program szeregujący może znacząco poprawić postrzeganą wydajność systemu, a tym samym satysfakcję użytkownika. W konsekwencji podejmowano wiele wysiłków w celu opracowania inteligentnych i wydajnych algorytmów szeregowania.

Powstanie komputerów osobistych zmieniło sytuację na dwa sposoby. Po pierwsze przez większość czasu jest tylko jeden aktywny proces. Użytkownik rozpoczynający edycję dokumentu w edytorze tekstów zwykle jednocześnie nie kompiluje w tle programu. Kiedy użytkownik wpisuje polecenie w edytorze, program szeregujący nie ma zbyt wiele pracy z wyznaczeniem procesu, który należy uruchomić — edytor tekstu jest jedynym kandydatem.

Po drugie komputery stały się przez lata o tyle szybsze, że czas procesora nie jest już dla nich deficytowym zasobem. W większości programów dla komputerów osobistych ograniczeniem jest tempo, w jakim użytkownik może dostarczać dane wejściowe (poprzez wpisywanie lub klikanie), a nie tempo, w jakim procesor je przetwarza. Nawet kompilacje — najważniejszy pożeracz cykli procesora w przeszłości — obecnie w większości przypadków zajmują zaledwie kilka sekund.

Kup ksi k Pole ksi k

Gdyby nawet dwa programy działały jednocześnie — np. edytor tekstu i arkusz kalkulacyjny — nie ma wielkiego znaczenia, który z nich uruchomi się w pierwszej kolejności, ponieważ użytkownik najprawdopodobniej oczekuje na zakończenie obydwóch (z wyjątkiem tego, że zwykle wykonują one swoje zadania tak szybko, że użytkownik i tak nie czeka długo). W konsekwencji szeregowanie nie ma wielkiego znaczenia na prostych komputerach osobistych. Oczywiście istnieją aplikacje, które praktycznie zjadają procesor żywcem — np. renderowanie jednogodzinnego filmu wideo w wysokiej rozdzielczości z jednoczesnym modyfikowaniem kolorów w każdej ze 108 tysięcy ramek (w systemie NTSC) lub 90 tysięcy ramek (w systemie PAL) wymaga ogromnej mocy obliczeniowej. Podobne aplikacje są jednak raczej wyjątkiem niż regułą.

W przypadku serwerów sieciowych sytuacja znacząco się zmienia. Wówczas wiele procesów zwykle rywalizuje o procesor, zatem szeregowanie odgrywa istotną rolę. Kiedy np. procesor wybiera pomiędzy uruchomieniem procesu zbierającego dzienne statystyki a takim, który obsługuje żądania użytkowników, użytkownik będzie o wiele bardziej zadowolony, jeśli to ten drugi uzyska przydział procesora w pierwszej kolejności.

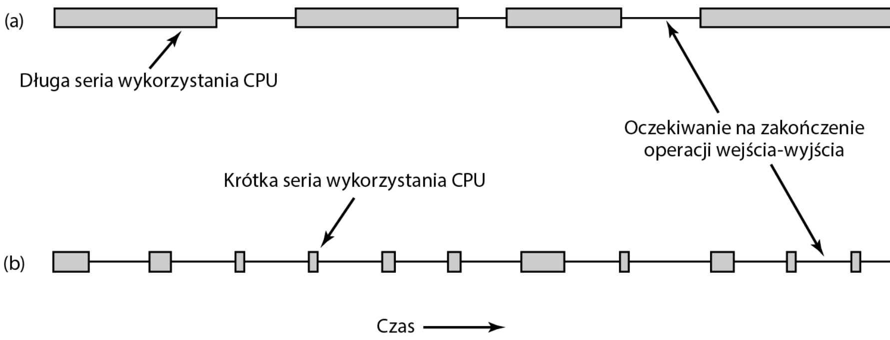
Cecha „obfitości zasobów" nie dotyczy również urządzeń IoT, węzłów w sieciach sensorowych, a także większości smartfonów. Nawet jeśli procesory w telefonach stają się coraz mocniejsze i instaluje się w nich coraz więcej pamięci, żywotność baterii nie ulega poprawie. Ponieważ w tych urządzeniach żywotność baterii jest jednym z najważniejszych ograniczeń, niektóre programy szeregujące dążą do optymalizacji zużycia energii.

Oprócz wyboru właściwego procesu do uruchomienia program szeregujący musi również dbać o wydajne wykorzystanie procesora, ponieważ przełączanie procesów jest kosztowną operacją. Na początek musi nastąpić przełączenie z trybu użytkownika do trybu jądra. Następnie należy zapisać stan bieżącego procesu, włącznie z zapisaniem jego rejestrów w tabeli procesów, tak by mogły być ponownie załadowane później. W wielu systemach trzeba zapisać także mapę pamięci (np. bity odwołań do pamięci w tabeli stron). Nazywa się to **przełączaniem kontekstu**, chociaż czasami używa się tego terminu w odniesieniu do pełnego **przełączania procesu**. Następnie trzeba wybrać nowy proces poprzez uruchomienie algorytmu szeregującego. Potem należy ponownie załadować moduł MMU z wykorzystaniem mapy pamięci nowego procesu. Na koniec trzeba uruchomić nowy proces. Oprócz tego wszystkiego przełączenie procesu zazwyczaj dezaktualizuje całą pamięć cache, co wymusza jej dynamiczne ładowanie z pamięci głównej. Operacja ta musi być wykonana dwukrotnie (przy wejściu do trybu jądra i podczas jego opuszczania). Podsumujmy: wykonywanie zbyt wielu operacji przełączania procesów w ciągu sekundy może doprowadzić do zużycia znaczącej ilości czasu procesora. W związku z tym zalecana jest ostrożność.

Zachowanie procesów

Niemal wszystkie procesy naprzemiennie wykonują obliczenia z (dyskowymi lub sieciowymi) żądaniami wejścia-wyjścia, co pokazano na rysunku 2.17. Zwykle procesor działa nieprzerwanie przez jakiś czas, a następnie wykonywane jest wywołanie systemowe do odczytania danych z pliku lub zapisania danych do pliku. Kiedy obsługa wywołania systemowego się zakończy, procesor ponownie wykonuje obliczenia do czasu, aż będzie potrzebował więcej danych lub będzie musiał zapisać więcej danych itd. Warto zwrócić uwagę, że niektóre operacje wejścia-wyjścia liczą się jako obliczenia. Kiedy np. procesor kopiuje fragmenty do pamięci wideo w celu aktualizacji ekranu, to wykonuje obliczenia, a nie operacje wejścia-wyjścia, ponieważ wykorzystuje do tego procesor. Operacja wejścia-wyjścia w sensie, w jakim rozumiemy to w naszym przykładzie, zachodzi wtedy, gdy proces wchodzi do stanu zablokowania w oczekiwaniu na to, aż urządzenie zewnętrzne zakończy pracę.

Kup ksi k Pole ksi k



Rysunek 2.17. Serie wykorzystania procesora CPU przeplatają się z okresami oczekiwania na zakończenie operacji wejścia-wyjścia. (a) Proces zorientowany na obliczenia; (b) proces zorientowany na operacje wejścia-wyjścia

Ważną rzeczą, na którą należy zwrócić uwagę na rysunku 2.17, jest to, że niektóre procesy, np. ten z rysunku 2.17(a), poświęcają większość czasu na obliczenia, podczas gdy inne, jak ten z rysunku 2.17(b), przez większość czasu oczekują na zakończenie operacji wejścia-wyjścia. Pierwsze określa się jako **zorientowane na obliczenia**, drugie to procesy **zorientowane na wejście-wyjście**. Procesy zorientowane na obliczenia zazwyczaj mają długie serie wykorzystania procesora, a w związku z tym rzadko oczekują na operacje wejścia-wyjścia, natomiast procesy zorientowane na wejście-wyjście mają krótkie serie wykorzystania procesora, a zatem często oczekują na zakończenie operacji wejścia-wyjścia. Zwróćmy uwagę, że kluczowym czynnikiem jest długość trwania serii wykorzystania procesora, a nie serii wykorzystania wejścia-wyjścia. Procesy zorientowane na wejście-wyjście są takie dlatego, że pomiędzy żądaniami wejścia-wyjścia nie wykonują zbyt wielu obliczeń, a nie dlatego, że ich żądania wejścia-wyjścia są szczególnie długotrwałe. Wydanie sprzętowego żądania odczytania bloku dysku zajmuje tyle samo czasu niezależnie od tego, jak dużo lub jak mało czasu zajmie przetworzenie danych, kiedy nadejdą.

Warto zwrócić uwagę na to, że w miarę jak procesory stają się coraz szybsze, procesy w coraz większym stopniu są zorientowane na wejście-wyjście. Efekt ten występuje dlatego, że postęp w dziedzinie procesorów jest znacznie szybszy niż w dziedzinie dysków. W rezultacie w przyszłości ważniejszym tematem może być szeregowanie procesów związanych z wejściem-wyjściem. Podstawowa idea w tym przypadku polega na tym, że jeśli proces zorientowany na wejście-wyjście chce działać, powinien szybko otrzymać swoją szansę. Jak widzieliśmy na rysunku 2.4, kiedy procesy są zorientowane na wejście-wyjście, potrzeba ich dość dużo, aby procesor był przez cały czas zajęty.

Z drugiej strony procesory nie stają się obecnie znacznie szybsze, ponieważ przyspieszanie ich powoduje wytwarzanie zbyt dużej ilości ciepła. Dyski twarde również nie stają się szybsze, ale dyski magnetyczne w komputerach stacjonarnych i notebookach są zastępowane przez dyski SSD. Przy czym w dużych centrach danych, ze względu na niższy koszt pamięci w przeliczeniu na bit, nadal powszechnie stosowane są dyski magnetyczne. W konsekwencji tego wszystkiego szeregowanie w dużej mierze zależy od kontekstu, a algorytm, który sprawdza się na notebooku, może się nie sprawdzić w centrum danych. Za 10 lat zresztą wszystko może się zmienić.

Kiedy wykonywać szeregowanie?

Kluczowym problemem związanym z szeregowaniem jest odpowiedź na pytanie o to, kiedy należy podejmować decyzje dotyczące szeregowania. Okazuje się, że istnieje wiele sytuacji, w których jest potrzebne szeregowanie. Po pierwsze w momencie tworzenia nowego procesu trzeba

Kup ksi k Pole ksi k

podjąć decyzję o tym, czy ma być uruchomiony proces rodzic, czy proces dziecko. Ponieważ oba procesy są w stanie gotowości, jest to normalna decyzja związana z szeregowaniem i może być podjęta w dowolny sposób — co oznacza, że program szeregujący może zdecydować o uruchomieniu w następnej kolejności rodzica lub dziecka.

Po drugie decyzję dotyczącą szeregowania należy podjąć w momencie, gdy proces kończy działanie. Proces, który się zakończył, nie może dłużej działać (ponieważ już nie istnieje), dlatego trzeba wybrać jakiś inny proces ze zbioru gotowych procesów. Jeśli żaden z procesów nie jest gotowy, w normalnych warunkach zaczyna działać systemowy proces bezczynności.

Po trzecie, kiedy proces blokuje się na operacji wejścia-wyjścia z powodu stanu semafora (lub z jakiegoś innego powodu), trzeba wybrać inny proces do uruchomienia. Czasami w wyborze może odgrywać rolę powód blokowania. Jeśli np. A jest ważnym procesem i oczekuje na to, aż B wyjdzie ze swojego regionu krytycznego, zezwolenie procesowi B na działanie w następnej kolejności pozwoli mu na opuszczenie swojego regionu krytycznego, a tym samym umożliwi działanie procesowi A. Problem polega jednak na tym, że program szeregujący zwykle nie posiada informacji pozwalających na wzięcie pod uwagę tej zależności.

Po czwarte decyzję szeregowania procesów trzeba podjąć w momencie wystąpienia przerwania wejścia-wyjścia. Jeśli przerwanie pochodzi od urządzenia wejścia-wyjścia, które zakończyło pracę, niektóre procesy zablokowane w oczekiwaniu na zakończenie operacji wejścia-wyjścia mogą być teraz gotowe do działania. Do kompetencji programu szeregującego należy decyzja o tym, czy należy uruchomić proces, który właśnie uzyskał gotowość, ten, który działał w czasie wystąpienia przerwania, czy jakiś inny.

Jeśli zegar sprzętowy dostarcza okresowych przerwań z częstotliwością 50 lub 60 Hz lub jakąś inną, decyzje szeregowania mogą być podejmowane z każdym przerwaniem zegara lub co k-te przerwanie zegara. Algorytmy szeregowania można podzielić na dwie kategorie, w zależności od sposobu postępowania z przerwaniami zegara. Algorytm szeregowania **bez wywłaszczania** (ang. *nonpreemptive*) wybiera proces do uruchomienia, a następnie pozwala mu działać do czasu zablokowania (na operacji wejścia-wyjścia lub w oczekiwaniu na inny proces) albo do momentu, kiedy proces z własnej woli zwolni CPU. Nawet jeśli proces będzie działał przez wiele godzin, nie będzie zmuszony do zawieszenia. W rezultacie podczas przerwań zegara nie są podejmowane decyzje dotyczące szeregowania. Po zakończeniu przetwarzania przerwania zegarowego wznawiany jest proces działający przed wystąpieniem przerwania, chyba że właśnie upłynął wymagany czas oczekiwania procesu o wyższym priorytecie.

Dla odróżnienia algorytm szeregowania **z wywłaszczaniem** (ang. *preemptive*) wybiera proces i pozwala mu działać maksymalnie przez ustalony czas. Jeśli po upływie przedziału czasu proces nadal działa, zostaje zawieszony, a program szeregujący wybiera do uruchomienia inny proces (jeśli jest dostępny). Aby była możliwa realizacja szeregowania z wywłaszczaniem, na końcu przedziału czasowego musi nastąpić przerwanie zegara. Jeśli nie jest dostępne przerwanie zegara, jedyną opcją okazuje się szeregowanie bez wywłaszczania.

Wywłaszczanie dotyczy nie tylko aplikacji, ale także jąder systemów operacyjnych, zwłaszcza monolitycznych. Współcześnie wiele z nich stosuje wywłaszczanie. Gdyby tak nie było, źle zaimplementowany sterownik lub bardzo powolne wywołanie systemowe mogłoby zablokować procesor. Zamiast tego w jądrze z wywłaszczaniem program szeregujący może wymusić na długo działającym sterowniku lub wywołaniu systemowym przełączenie kontekstu.

Kategorie algorytmów szeregowania

Nie powinno być zaskoczeniem, że w różnych środowiskach potrzebne są różne algorytmy szeregowania. Sytuacja ta występuje dlatego, że różne obszary aplikacji (i różne rodzaje systemów

Kup ksi k Pole ksi k

operacyjnych) realizują różne cele. Inaczej mówiąc, programy szeregujące działające w różnych systemach powinny stosować inne kryteria optymalizacji. Warto wyróżnić trzy środowiska:

1. Wsadowe.
2. Interaktywne.
3. Czasu rzeczywistego.

Systemy wsadowe są ciągle powszechnie używane w biznesie do wykonywania takich zadań jak generowanie list płac, inwentaryzacje, obliczanie uznań i obciążeń, naliczanie odsetek (w bankach), przetwarzanie roszczeń o odszkodowania (w firmach ubezpieczeniowych) oraz innych okresowych zadań. W systemach wsadowych nie ma użytkowników, którzy niecierpliwie oczekują przy terminalach na szybką odpowiedź na krótkie żądanie. W konsekwencji w tych systemach akceptowalne są algorytmy bez wywłaszczania lub algorytmy z wywłaszczaniem o długich przedziałach czasu dla każdego procesu. Algorytmy wsadowe są w zasadzie dość ogólne i często stosuje się je również w innych sytuacjach. W związku z tym warto, by przestudiowały je także te osoby, które nie są związane z obliczeniami przemysłowymi i komputerami typu mainframe.

W środowiskach, w których są interaktywni użytkownicy, wywłaszczanie ma kluczowe znaczenie, aby nie dopuścić do tego, by jeden proces okupował procesor i blokował innym dostęp do niego. Nawet jeśli nie ma takiego procesu, który celowo działa w nieskończoność, jeden proces może zablokować możliwość działania innym niechcący — z powodu błędu w programie. Wywłaszczanie jest potrzebne w celu zapobiegania takim zachowaniom. Do tej kategorii należą również serwery, ponieważ standardowo obsługują one wielu (zdalnych) użytkowników, którzy — wszyscy — bardzo się spieszą. Użytkownicy komputerów zawsze się spieszą.

W systemach z ograniczeniami czasu rzeczywistego, choć może się to wydawać dziwne, wywłaszczanie czasami nie jest potrzebne. Procesy wiedzą bowiem, że nie mogą działać przez długi czas, dlatego zwykle wykonują swoją pracę i szybko się blokują. Różnica w porównaniu z systemami interaktywnymi polega na tym, że w systemach czasu rzeczywistego działają wyłącznie takie programy, których celem jest wspomaganie jednej aplikacji. Systemy interaktywne są ogólnego przeznaczenia i mogą w nich działać dowolne programy, które nie tylko ze sobą nie współpracują, ale nawet są wobec siebie złośliwe.

Cele algorytmów szeregowania

Aby zaprojektować algorytm szeregowania, trzeba wiedzieć, jakie cele powinien on spełniać. Niektóre cele zależą od środowiska (wsadowe, interaktywne, czasu rzeczywistego), ale niektóre są pożądane we wszystkich przypadkach. Wybrane cele przedstawiono w tabeli 2.8. Omówimy je po kolei w dalszej części rozdziału.

W każdych okolicznościach sprawiedliwość ma znaczenie. Porównywalne procesy powinny uzyskiwać porównywalną obsługę. Przydzielanie jednemu procesowi znacznie więcej czasu procesora niż innym nie jest sprawiedliwe. Oczywiście różne kategorie procesów mogą być traktowane różnie. Rozważmy procesy kontroli bezpieczeństwa oraz tworzenia listy płac w centrum obliczeniowym reaktora nuklearnego.

W pewnym stopniu ze sprawiedliwością wiąże się dbałość o przestrzeganie przyjętych zasad w systemie. Jeśli lokalna strategia mówi, że procesy kontroli bezpieczeństwa mogą działać wtedy, kiedy chcą, nawet jeśli lista płac będzie przygotowana 30 s później, program szeregujący musi zapewnić, aby ta zasada była przestrzegana. Może to wymagać dodatkowego wysiłku.

Kup ksi k Pole ksi k

Tabela 2.8. Wybrane cele algorytmów szeregowania w różnych okolicznościach

Wszystkie systemy
Sprawiedliwość — przydzielanie każdemu procesowi odpowiedniego czasu procesora
Wymuszanie strategii — sprawdzanie, czy jest przestrzegana zamierzona strategia
Równowaga — dbanie o to, by wszystkie części systemu były zajęte
Systemy wsadowe
Przepustowość — maksymalizacja liczby wykonywanych zadań na godzinę
Czas cyklu przetwarzania — minimalizacja czasu pomiędzy rozpoczęciem pracy procesu, a jej zakończeniem
Wykorzystanie procesora — dbanie o ciągłą zajętość procesora
Systemy interaktywne
Czas odpowiedzi — szybka odpowiedź na żądania
Proporcjonalność — spełnianie oczekiwań użytkowników
Systemy czasu rzeczywistego
Dotrzymywanie terminów — unikanie utraty danych
Przewidywalność — unikanie degradacji jakości w systemach multimedialnych

Innym ogólnym celem jest dbanie o to, aby wszystkie elementy systemu były zajęte zawsze, kiedy to możliwe. Jeśli procesor i wszystkie urządzenia wejścia-wyjścia będą działać przez cały czas, system wykona więcej pracy na sekundę w porównaniu z sytuacją, kiedy niektóre z komponentów pozostają bezczynne. Przykładowo w systemie wsadowym program szeregujący ma kontrolę nad tym, które zadania będą przesłane do pamięci w celu uruchomienia. Załadowanie do pamięci kilku procesów zorientowanych na procesor razem z kilkoma zorientowanymi na operacje wejścia-wyjścia jest lepszym pomysłem niż załadowanie najpierw wszystkich zadań zorientowanych na procesor, a następnie, kiedy zostaną one zakończone, załadowanie i uruchomienie wszystkich zadań zorientowanych na operacje wejścia-wyjścia. W razie zastosowania tej drugiej strategii, jeśli będą działać procesy zorientowane na procesor, wszystkie one będą walczyły o procesor. W tej sytuacji dysk będzie bezczynny. Kiedy później zostaną załadowane zadania zorientowane na operacje wejścia-wyjścia, będą one walczyły o dysk i procesor pozostanie bezczynny. Lepszym rozwiązaniem jest uważne dobranie procesów, tak by działał cały system.

Menedżerowie dużych centrów obliczeniowych, w których uruchamianych jest wiele zadań wsadowych, oceniając wydajność swoich systemów, zazwyczaj biorą pod uwagę trzy metryki: przepustowość, czas cyklu przetwarzania oraz wykorzystanie procesora. **Przepustowość** określa liczbę zadań zrealizowanych przez system w ciągu godziny. W końcu wykonanie 50 zadań w ciągu godziny jest lepsze od wykonania 40 zadań w ciągu godziny. **Czas cyklu przetwarzania** to statystycznie średni czas od momentu, kiedy zadanie wsadowe zostanie przekazane do realizacji, do chwili, kiedy zostanie ono zakończone. Parametr ten mierzy, jak długo przeciętny użytkownik musi czekać na wyniki. W tym przypadku reguła brzmi: małe jest piękne.

Algorytm szeregowania, który maksymalizuje przepustowość, niekoniecznie musi minimalizować czas cyklu przetwarzania. I tak w przypadku gdy w systemie występują zadania krótkotrwałe i długotrwałe, program szeregujący, który zawsze uruchamia krótkotrwałe zadania i unika uruchamiania długotrwałych, może osiągnąć doskonałą przepustowość (wiele krótkotrwałych zadań na godzinę), ale kosztem bardzo wysokiego czasu cyklu przetwarzania zadań długotrwałych. Jeśli zadania krótkotrwałe będą napływać w stałym tempie, zadania długotrwałe mogą nie dostać szansy na uruchomienie. W ten sposób średni czas cyklu przetwarzania będzie nieskończony, a przepustowość wysoka.

Kup ksi k Pole ksi k

W systemach wsadowych w roli metryki często używa się zajętości procesora. W rzeczywistości jednak nie jest to zbyt dobra metryka. Prawdziwe znaczenie ma to, ile zadań w systemie będzie wykonanych (przepustowość) oraz ile czasu zajmie wykonanie zadania przekazanego do obliczeń (czas cyklu przetwarzania). Użycie wskaźnika zajętości procesora jako metryki przypomina ocenę samochodów na podstawie tego, ile obrotów na godzinę wykona silnik. Z drugiej strony, jeśli wiadomo, kiedy wykorzystanie procesora zbliża się do 100%, wiadomo też, kiedy należy pomyśleć o dodatkowej mocy obliczeniowej.

Dla systemów interaktywnych stosuje się inne cele. Najważniejszym jest minimalizacja **czasu odpowiedzi** — czyli czasu od wydania polecenia do otrzymania wyników. W komputerze osobistym, w którym działa proces drugoplanowy (np. czytający i zapisujący wiadomości e-mail z sieci), żądanie użytkownika uruchomienia programu lub otwarcia pliku powinno mieć pierwszeństwo przed zadaniem drugoplanowym. Udzielenie pierwszeństwa wszystkim interaktywnym żądaniom będzie postrzegane jako dobra obsługa.

W pewnym stopniu powiązana z czasem odpowiedzi jest metryka, którą można by nazwać **proporcjonalnością**. Użytkownicy mają wewnętrzne poczucie (często nieprawidłowe) tego, ile powinna zająć określona operacja. Kiedy żądanie postrzegane jako złożone zajmuje dużo czasu, użytkownicy to akceptują, ale jeśli zadanie uważane za proste zajmuje dużo czasu, irytują się. Jeśli np. po kliknięciu ikony, która uruchamia operację wgrania pliku wideo o rozmiarze 5 GB na serwer w chmurze, zadanie zostaje wykonane po 60 s, użytkownik najprawdopodobniej zaakceptuje to jako obowiązujący fakt, ponieważ nie spodziewa się, że operacja przesyłania na serwer zajmie 5 s. Wie, że to musi potrwać.

Z drugiej strony, jeśli użytkownik klika ikonę operacji przerwania połączenia z chmurą po przesłaniu pliku wideo, ma zupełnie odmienne oczekiwania. Jeżeli operacja nie zakończy się po 30 s, użytkownik będzie coś mruczał pod nosem, natomiast po 60 s będzie miał pianę na ustach. Takie zachowanie wynika z powszechnej opinii użytkowników, że wysyłanie dużej ilości danych powinno zająć więcej czasu niż zwykłe przerwanie połączenia. W niektórych przypadkach (takich jak ten) program szeregujący nie może nic zrobić z czasem odpowiedzi. Czasami jednak może, zwłaszcza kiedy opóźnienie wynika z przyjęcia niewłaściwej kolejności procesów.

Systemy czasu rzeczywistego charakteryzują się innymi właściwościami niż systemy interaktywne, dlatego program szeregujący musi spełniać inne cele. Często są one charakteryzowane przez ścisłe terminy, które muszą, albo co najmniej powinny, być dotrzymane. Jeśli np. komputer steruje urządzeniem, które generuje dane w stałym tempie, to niepowodzenie uruchomienia procesu zbierania danych na czas może skutkować utratą danych. Tak więc najważniejszym wymaganiem w systemach czasu rzeczywistego jest dotrzymanie wszystkich (lub większości) terminów.

W niektórych systemach czasu rzeczywistego, zwłaszcza tych, które wykorzystują multimedia, ważna jest przewidywalność Niedotrzymanie jednego z terminów nie ma kluczowego znaczenia, ale jeśli proces obsługi dźwięku działa nieprawidłowo, jakość dźwięku gwałtownie się pogorszy. Wideo również jest problemem, ale ucho jest znacznie czulsze na zniekształcenia niż oko. Aby uniknąć tego problemu, szeregowanie procesów musi być przewidywalne i regularne. Algorytmy szeregowania w systemach wsadowych i interaktywnych przeanalizujemy w tym rozdziale.

2.5.2. Szeregowanie w systemach wsadowych

Teraz nadszedł czas, by przejść od ogólnych problemów szeregowania do konkretnych algorytmów. W tym punkcie zajmiemy się algorytmami wykorzystywanymi w systemach wsadowych. W dalszych punktach omówimy systemy interaktywne i systemy czasu rzeczywistego. Warto zwrócić uwagę na to, że niektóre algorytmy są wykorzystywane zarówno w systemach wsadowych, jak i w systemach interaktywnych. Algorytmy te przeanalizujemy później.

Kup ksi k Pole ksi k

Pierwszy zgłoszony, pierwszy obsłużony

Najprostszym ze wszystkich algorytmów szeregowania jest algorytm bez wywłaszczania **pierwszy zgłoszony, pierwszy obsłużony** (ang. *first come, first served*). W przypadku zastosowania tego algorytmu procesy otrzymują procesor w kolejności, w jakiej go żądają. Ogólnie rzecz biorąc, jest jedna kolejka gotowych procesów. Kiedy pierwsze zadanie nadejdzie do systemu z zewnątrz, jest natychmiast uruchamiane i może działać tak długo, jak chce. Nie zostanie przerwane dlatego, że działało zbyt długo. W miarę jak nadchodzą kolejne zadania, są one umieszczane na końcu kolejki. Kiedy działający proces się zablokuje, w następnej kolejności uruchamiany jest pierwszy proces z kolejki. Kiedy zablokowany proces uzyska gotowość, jest on umieszczany na końcu kolejki, tak jak zadanie, które dopiero nadeszło — za wszystkimi oczekującymi procesami.

Wielką zaletą tego algorytmu jest to, że łatwo go zrozumieć i równie łatwo zaprogramować. Jest on również sprawiedliwy w takim samym sensie, jak sprawiedliwa jest sprzedaż nowiutkich iPhone'ów osobom, które chcą stać w kolejce od drugiej w nocy. W przypadku zastosowania tego algorytmu do przechowywania wszystkich gotowych procesów wykorzystywana jest jednokierunkowa lista. Wybranie procesu do uruchomienia wymaga usunięcia jednego procesu z początku kolejki. Dodanie nowego procesu lub niezablokowanego procesu wymaga dołączenia go na koniec kolejki. Czy może być coś prostszego do zrozumienia i zaimplementowania?

Niestety, algorytm pierwszy zgłoszony, pierwszy obsłużony ma również istotną wadę. Przypuśćmy, że w systemie jest jeden proces zorientowany na procesor, który jednorazowo działa przez 1 s, oraz wiele procesów zorientowanych na operacje wejścia-wyjścia, które zużywają mało czasu procesora, ale każdy z nich podczas realizacji musi wykonać 1000 odczytów dysku. Proces zorientowany na obliczenia działa przez 1 s, a następnie czyta blok danych z dysku. Teraz zaczynają po kolei działać wszystkie procesy wejścia-wyjścia, odczytując dane z dysku. Kiedy proces zorientowany na obliczenia otrzyma żądany blok danych z dysku, zostanie uruchomiony na kolejną sekundę, a za nim, w bezpośrednim następstwie, zostaną uruchomione wszystkie procesy zorientowane na operacje wejścia-wyjścia.

W efekcie końcowym każdy z procesów zorientowanych na wejścia-wyjścia będzie czytał 1 blok na sekundę, a zatem jego wykonanie zajmie 1000 s. W przypadku zastosowania algorytmu szeregowania, który wywłaszczałby proces zorientowany na procesor co 10 ms, realizacja procesów zorientowanych na wejścia-wyjścia zajęłaby 10 s zamiast 1000 s, a spowolnienie procesu zorientowanego na obliczenia nie byłoby zbyt duże.

Najpierw najkrótsze zadanie

Przyjrzyjmy się teraz innemu algorytmowi bez wywłaszczania, stosowanemu w systemach wsadowych, w którym przyjmuje się, że czasy działania procesów są z góry znane. Przykładowo w firmie ubezpieczeniowej można dosyć dokładnie przewidzieć, ile czasu zajmie przetworzenie paczki 1000 żądań odszkodowania, ponieważ podobne operacje są wykonywane codziennie. Kiedy w kolejce wejściowej jest do uruchomienia kilka zadań równych co do ważności, program szeregujący **najpierw wybiera zadanie krótsze**. Spójrzmy na rysunek 2.18. Mamy na nim zadania A, B, C i D o czasach działania odpowiednio 8, 4, 4 i 4 min. Przy uruchomieniu ich w tej kolejności czas cyklu przetwarzania dla procesu A wynosi 8 min, dla procesu B — 12 min, dla procesu C — 16 min, a dla procesu D — 20 min, co daje średnią 14 min.

Kup ksi k Pole ksi k

Rysunek 2.18. Przykład algorytmu szeregowania: najpierw krótsze zadania; (a) uruchamianie zadań w kolejności pierwotnej; (b) uruchamianie zadań według zasady „najpierw krótsze zadanie"

Rozważmy teraz uruchomienie tych czterech zadań z wykorzystaniem algorytmu „najpierw najkrótsze zadanie", tak jak pokazano na rysunku 2.18(b). Czasy cyklu przetwarzania wynoszą teraz 4, 8, 12 i 20 min, co daje średnią 11 min. Optymalność algorytmu „najpierw najkrótsze zadanie" można udowodnić. Rozważmy przypadek czterech zadań o czasach działania odpowiednio a, b, c i d. Pierwsze zadanie kończy się w czasie a, drugie w czasie $a + b$ itd. Średni czas cyklu przetwarzania wynosi $(4a + 3b + 2c + d) : 4$. Jest oczywiste, że składnik a ma większy udział w średniej niż pozostałe czasy, zatem powinno to być najkrótsze zadanie, później b, następnie c i na koniec d — zadanie najdłuższe, które ma wpływ tylko na własny czas cyklu przetwarzania. To samo rozumowanie można zastosować do dowolnej liczby zadań.

Warto dodać, że algorytm „najpierw najkrótsze zadanie" jest optymalny tylko wtedy, kiedy wszystkie zadania są dostępne jednocześnie. W roli kontrprzykładu rozważmy pięć zadań, od A do E, o czasach działania odpowiednio 2, 4, 1, 1 i 1. Ich czasy nadejścia to 0, 0, 3, 3 i 3. Początkowo mogą być wybrane tylko zadania A lub B, ponieważ inne zadania jeszcze nie dotarły. Przy użyciu algorytmu „najpierw najkrótsze zadanie" będziemy uruchamiać zadania w kolejności A, B, C, D, E — co daje średnią oczekiwania 4,6 s. Natomiast uruchomienie ich w kolejności B, C, D, E, A daje średnią oczekiwania wynoszącą 4,4 s.

Następny proces o najkrótszym pozostałym czasie działania

Odmianą algorytmu „najpierw najkrótsze zadanie" z wywłaszczaniem jest algorytm „następny proces o najkrótszym pozostałym czasie działania". W przypadku użycia tego algorytmu program szeregujący zawsze wybiera proces, którego pozostały czas działania jest najkrótszy. W tym przypadku czas działania również musi być znany z góry. Kiedy nadejdzie następne zadanie, całkowity czas jego działania jest porównywany z pozostałym czasem działania bieżącego procesu. Jeśli nowe zadanie wymaga mniej czasu do zakończenia niż bieżący proces, jest on zawieszany, a program szeregujący uruchamia nowe zadanie. Ten schemat umożliwia uzyskanie dobrej obsługi przez nowe, krótkie zadania.

2.5.3. Szeregowanie w systemach interaktywnych

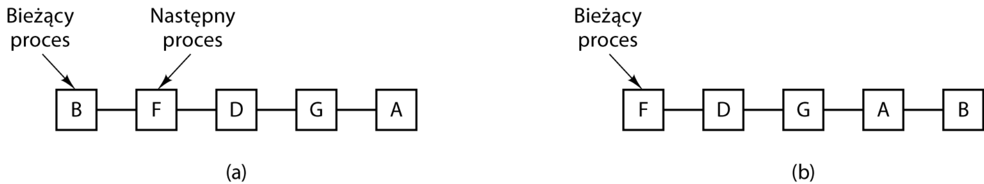
W tym punkcie przyjrzymy się wybranym algorytmom, które można wykorzystać w systemach interaktywnych. Są one powszechne w komputerach osobistych, serwerach, a także innych rodzajach komputerów.

Szeregowanie cykliczne

Jednym z najstarszych, najprostszych, najbardziej sprawiedliwych i najczęściej używanych algorytmów szeregowania jest **szeregowanie cykliczne**. Każdemu procesowi jest przydzielany przedział czasu, nazywany **kwantem**, podczas którego proces może działać. Jeśli po zakończeniu kwantu

Kup ksi k Pole ksi k

proces dalej działa, procesor jest wywłaszczany i przekazywany do innego procesu. Jeżeli proces zablokował się lub zakończył, zanim upłynął kwant, następuje przełączenie procesora. Cykliczny algorytm szeregowania jest łatwy do zaimplementowania. Program szeregujący musi jedynie utrzymywać listę procesów do uruchomienia, podobną do pokazanej na rysunku 2.19(a). Kiedy proces wykorzysta swój kwant, jest umieszczany na końcu listy, co pokazano na rysunku 2.19(b).



Rysunek 2.19. Szeregowanie cykliczne: (a) lista procesów do uruchomienia; (b) lista procesów do uruchomienia po tym, jak proces B wykorzystał swój kwant

Jedynym interesującym problemem w cyklicznym algorytmie szeregowania jest długość kwantu. Przełączenie z jednego procesu do innego wymaga określonego czasu na wykonanie zadań administracyjnych — zapisania i załadowania rejestrów i mapy pamięci, aktualizacji różnych tabel i list, opróżnienia i ponownego załadowania pamięci podręcznej itp. Załóżmy, że to przełączenie kontekstu zajmuje 1 ms i obejmuje takie zadania jak przełączenie map pamięci, opróżnienie i ponowne załadowanie pamięci cache itp. Załóżmy także, że długość kwantu ustawiono na 4 ms. Przy tych parametrach po 4 ms użytecznej pracy procesor będzie musiał poświęcić (a tym samym zmarnować) 1 ms na przełączanie procesów. Tak więc 20% czasu procesora zostanie teraz zmarnowanych na zadania administracyjne. Wartość ta jest oczywiście zbyt duża.

W celu poprawy wydajności procesora możemy ustawić kwant na przykładowo 100 ms. Teraz zmarnotrawiony czas wynosi tylko 1%. Zastanówmy się jednak, co się stanie w systemie serwera, jeśli 50 żądań nadejdzie w ciągu bardzo krótkiego czasu i będą one miały bardzo różne wymagania w zakresie procesora. Na liście procesów do uruchomienia zostanie umieszczonych pięćdziesiąt procesów. Jeśli procesor będzie bezczynny, pierwszy proces uruchomi się natychmiast, drugi nie będzie mógł się uruchomić wcześniej niż za 100 ms itd. Ostatni, przy założeniu, że wszystkie poprzednie w pełni wykorzystały swoje kwanty, może być zmuszony do oczekiwania na swoją szansę przez 5 s. Większość użytkowników odczuje 5-sekundową odpowiedź na krótkie polecenie jako bardzo powolną. Sytuacja ta jest szczególnie zła, jeśli niektóre żądania umieszczone w pobliżu końca kolejki wymagają zaledwie kilku milisekund czasu procesora. Przy krótkim czasie kwantu otrzymałyby one lepszą obsługę.

Jeśli z kolei kwant zostanie ustawiony na dłuższą wartość od średniego czasu wykorzystania procesora, wywłaszczanie nie będzie wykonywane zbyt często. Zamiast tego większość procesów będzie wykonywała operację blokowania, zanim upłynie kwant, co spowoduje przełączenie procesu. Wyeliminowanie wywłaszczania poprawia wydajność, ponieważ przełączanie procesów zachodzi tylko wtedy, gdy jest logicznie konieczne — czyli kiedy proces się zablokuje i nie może kontynuować działania.

Konkluzję można sformułować w następujący sposób: ustawienie kwantu na zbyt niską wartość powoduje zbyt wiele przełączeń procesów i obniża wydajność procesora, ale ustawienie go na zbyt wysoką wartość może przyczynić się do wydłużenia odpowiedzi na krótkie, interaktywne żądania. Rozsądnym kompromisem jest często kwant o czasie trwania 20 – 50 ms.

Kup ksi k Pole ksi k

Szeregowanie bazujące na priorytetach

Przy szeregowaniu cyklicznym przyjmuje się niejawne założenie, że wszystkie procesy są jednakowo ważne. Osoby, które posiadają i wykorzystują komputery wielodostępne, często mają odmienne poglądy na tę kwestię. Przykładowo na wyższej uczelni może obowiązywać hierarchia, według której najpierw są obsługiwane żądania dziekana, później profesorów, następnie sekretarek, woźnych i na końcu studentów. Konieczność brania pod uwagę czynników zewnętrznych prowadzi do **szeregowania według priorytetów**. Podstawowa idea jest prosta: każdemu procesowi jest przydzielany priorytet, a program szeregujący zezwala na działanie procesowi o najwyższym priorytecie.

Nawet w komputerze PC, który ma jednego właściciela, może być wiele procesów ważniejszych niż inne. I tak procesowi demonowi, który w tle wysyła pocztę elektroniczną, powinien być przydzielony niższy priorytet niż procesowi wyświetlającemu w czasie rzeczywistym film.

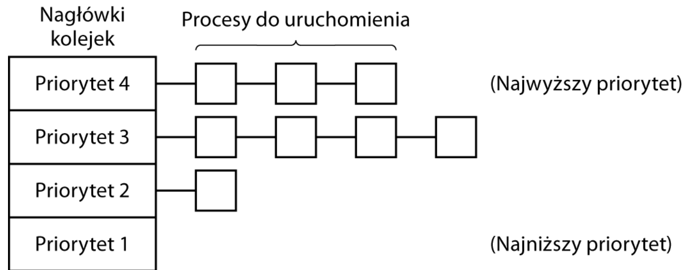
Aby nie dopuścić do tego, by procesy o wysokich priorytetach działały w nieskończoność, program szeregujący może zmniejszać priorytet działających procesów wraz z każdym cyklem zegara. Jeśli działanie to spowoduje obniżenie priorytetu poniżej priorytetu następnego w kolejności procesu, następuje przełączenie procesów. Można również przydzielić każdemu procesowi maksymalny kwant czasu, przez który może działać. Kiedy ten kwant zostanie wykorzystany, szansę na działanie otrzymuje następny proces w kolejności priorytetów. Po upływie określonego czasu priorytet procesu powinien zostać podniesiony zgodnie z jakimś algorytmem, tak aby proces mógł ponownie działać. W przeciwnym razie wszystkie procesy ostatecznie osiągnęłyby priorytet 0.

Priorytety mogą być przypisywane procesom w sposób statyczny lub dynamiczny. W komputerze wojskowym procesy uruchamiane przez generałów mogą mieć początkowy priorytet 100, procesy uruchamiane przez pułkowników — 90, majorów — 80, kapitanów — 70, poruczników — 60 itd. Z kolei w komercyjnym centrum obliczeniowym zadania o wysokim priorytecie mogą kosztować 100 dolarów na godzinę, o średnim priorytecie — 75 dolarów na godzinę, a zadania o niskim priorytecie — 50 dolarów na godzinę. W systemie UNIX istnieje polecenie `nice`, które pozwala użytkownikowi dobrowolnie obniżyć priorytet swojego procesu, aby wykazać się uprzejmością w odniesieniu do innych użytkowników. Nic dziwnego, że nikt go nie używa.

System może także przydzielać priorytety dynamicznie w celu osiągnięcia określonych celów. Niektóre procesy np. są ściśle zorientowane na operacje wejścia-wyjścia i przez większość czasu oczekują na zakończenie wykonywania operacji wejścia-wyjścia. Za każdym razem, kiedy taki proces chce uzyskać dostęp do procesora, powinien go otrzymać natychmiast. Dzięki temu będzie on mógł uruchomić swoje następne żądanie wejścia-wyjścia, które będzie realizowane równolegle z innym procesem wykonującym obliczenia. Zmuszanie procesu zorientowanego na operacje wejścia-wyjścia do długotrwałego oczekiwania na procesor będzie oznaczało, że niepotrzebnie zajmie on pamięć przez długi czas. Prosty algorytm zapewniający dobrą obsługę dla procesów zorientowanych na operacje wejścia-wyjścia polega na ustawieniu priorytetu na wartość $1/f$, gdzie f oznacza fragment ostatniego kwantu wykorzystanego przez proces. Proces, który wykorzystał tylko 1 ms z kwantu o długości 50 ms, otrzymuje priorytet 50, proces, który przed zablokowaniem działał 25 ms, otrzymałby priorytet 2, natomiast proces, który wykorzystał cały kwant, otrzymuje priorytet 1.

Często wygodne jest pogrupowanie procesów na klasy priorytetów i wykorzystanie szeregowania bazującego na priorytetach pomiędzy klasami przy zastosowaniu szeregowania cyklicznego w obrębie każdej z klas. Na rysunku 2.20 pokazano system z czterema klasami priorytetów. Algorytm szeregowania jest następujący: o ile istnieją procesy możliwe do uruchomienia w 4. klasie priorytetów, należy uruchomić po jednym w każdym kwancie w sposób cykliczny i nie

Kup ksi k Pole ksi k



Rysunek 2.20. Algorytm szeregowania z czterema klasami priorytetów

przejmować się niższymi klasami priorytetów. Jeśli 4. klasa priorytetów jest pusta, uruchamiamy cyklicznie procesy klasy 3. Jeśli zarówno klasa 4., jak i 3. są puste, to cyklicznie są uruchamiane procesy klasy 2. itd. Jeśli priorytety nie będą czasami korygowane, procesy o niższych priorytetach mogą nie dostać szansy na działanie.

Wielokrotne kolejki

Jednym z pierwszych systemów, w których zastosowano program szeregujący z wykorzystaniem priorytetów, był zbudowany w MIT system CTSS (*Compatible Time Sharing System*) działający na komputerze IBM 7094 [Corbató et al., 1962]. W systemie CTSS problemem było bardzo powolne przełączanie procesów, ponieważ komputer 7094 mógł przechowywać w pamięci tylko jeden proces. Każde przełączenie oznaczało zapisanie bieżącego procesu na dysk i odczytanie nowego z dysku. Projektanci systemu CTSS szybko doszli do wniosku, że wydajniejszym rozwiązaniem będzie przydzielenie procesom zorientowanym na obliczenia większego kwantu co jakiś czas niż częste przydzielanie im krótkich kwantów (w celu ograniczenia przełączania). Z drugiej strony przydzielenie dużych kwantów wszystkim procesom oznaczałoby długie czasy odpowiedzi (o czym przekonaliśmy się wcześniej). Przyjęto rozwiązanie polegające na skonfigurowaniu klas priorytetów. Procesy należące do najwyższej klasy działały przez jeden kwant. Procesy należące do kolejnej klasy w hierarchii działały przez dwa kwanty. Procesy należące do kolejnej klasy działały przez cztery kwanty itd. Zawsze, gdy proces wykorzystał wszystkie kwanty, które zostały do niego przydzielone, był przenoszony w dół o jedną klasę. Dzięki temu procesy w najwyższej klasie byłyby uruchamiane częściej i z wysokim priorytetem, ale przez krótszy czas — to idealne rozwiązanie w wypadku procesów interaktywnych.

Dla przykładu rozważmy proces, który musiał realizować obliczenia przez 100 kwantów. Początkowo otrzyma jeden kwant, a następnie zostanie przeniesiony na dysk. Następnym razem otrzyma dwa kwanty, po których zostanie przeniesiony na dysk. W kolejnych uruchomieniach uzyska 4, 8, 16, 32 i 64 kwanty, chociaż do zakończenia pracy potrzeba będzie tylko 37 z przydzielonych 64 kwantów. Potrzebne byłoby tylko 7 przesunięć procesu pomiędzy pamięcią a dyskiem (włącznie z początkowym załadowaniem) zamiast 100 w przypadku klasycznego algorytmu cyklicznego. Co więcej, w miarę jak proces wchodzi coraz głębiej w kolejki priorytetów, działa coraz rzadziej. Dzięki temu procesor może być przydzielany krótkim, interaktywnym procesom.

Aby proces, który potrzebuje działać przez długi czas przy pierwszym uruchomieniu, a potem zmienia się w proces interaktywny, nie był zablokowany na zawsze, zastosowano strategię opisaną poniżej. Każdorazowe wciśnięcie na terminalu znaku powrotu karetki (klawisza *Enter*) powoduje przeniesienie procesu należącego do tego terminala do najwyższej klasy priorytetów z założeniem, że proces ten przekształci się w interaktywny. Pewnego dnia użytkownik procesu mocno

Kup ksi k Pole ksi k

zorientowanego na obliczenia odkrył, że siedzenie przy terminalu i losowe wciskanie klawisza *Enter* znacząco poprawia czasy odpowiedzi. O swoim odkryciu opowiedział kolegom. Oni z kolei opowiedzieli swoim kolegom. Jaki jest morał tej historii? Rozwiązanie problemu w praktyce jest znacznie trudniejsze od opracowania zasady jego rozwiązania.

Następny najkrótszy proces

Ponieważ algorytm „najpierw najkrótsze zadanie" zawsze generuje minimalny czas odpowiedzi dla systemów wsadowych, byłoby dobrze, gdyby można go było również wykorzystać w systemach interaktywnych. Do pewnego stopnia można to zrobić. Procesy interaktywne, ogólnie rzecz biorąc, działają według schematu: oczekiwanie na polecenie, wykonanie polecenia, oczekiwanie na polecenie, wykonanie polecenia itd. Jeśli uznamy wykonywanie każdego zadania za oddzielne „zadanie", to będziemy mogli zminimalizować ogólny czas odpowiedzi poprzez uruchomienie najkrótszego zadania w pierwszej kolejności. Jedynym problemem jest określenie, który z procesów do uruchomienia jest tym najkrótszym.

Jedno z podejść polega na oszacowaniu na podstawie działania w przeszłości i uruchomieniu procesu o najkrótszym szacowanym czasie działania. Załóżmy, że szacowany czas na wykonanie polecenia dla pewnego terminala wynosi T_0. Przypuśćmy także, że czas następnego uruchomienia zmierzono jako T_1. Możemy zaktualizować naszą ocenę poprzez obliczenie sumy ważonej tych dwóch liczb — tzn. $aT_0 + (1 - a)T_1$. Dzięki odpowiedniemu wybraniu parametru a możemy zdecydować, czy proces szacowania powinien szybko zapomnieć przeszłe uruchomienia, czy ma je pamiętać przez długi czas. Przy $a = \frac{1}{2}$ otrzymujemy następujące kolejne oszacowania:

```
T0,T0/2 + T1/2,T0/4 + T1/4 + T2/2,T0/8 + T1/8 + T2/4 + T3/2
```

Po trzech nowych uruchomieniach waga T_0 w nowym oszacowaniu spadła do $\frac{1}{8}$.

Technikę szacowania następnej wartości w szeregu na podstawie średniej ważonej bieżącej zmierzonej wartości i poprzedniego oszacowania czasami określa się terminem **starzenie**. Tę technikę stosuje się w wielu sytuacjach, w których należy przewidzieć wynik na podstawie poprzednich wartości. Starzenie jest szczególnie łatwe do zaimplementowania, kiedy $a = \frac{1}{2}$. Trzeba jedynie dodać nową wartość do bieżącego oszacowania i podzielić sumę przez 2 (poprzez przesunięcie w prawo o 1 bit).

Szeregowanie gwarantowane

Całkowicie inne podejście do szeregowania polega na złożeniu użytkownikom obietnic dotyczących wydajności, a następnie spełnienie ich. Jedna z obietnic, którą można realistycznie złożyć i łatwo dotrzymać, jest następująca: jeśli jest n użytkowników zalogowanych podczas pracy, każdy z nich otrzyma $1/n$ mocy procesora. Na podobnej zasadzie w systemie z jednym użytkownikiem, gdy działa n równoprawnych procesów, każdy z nich powinien otrzymać $1/n$ cykli procesora. Algorytm ten wydaje się sprawiedliwy.

Aby dotrzymać tej obietnicy, system musi śledzić, ile czasu procesora miał każdy z procesów od momentu utworzenia. Następnie oblicza czas procesora, do jakiego każdy z procesów jest uprawniony — w tym celu dzieli czas, jaki upłynął od utworzenia przez n. Ponieważ czas procesora, jaki faktycznie miał każdy proces, również jest znany, dość łatwo można obliczyć stosunek rzeczywistego czasu procesora do czasu, przez jaki proces był uprawniony do korzystania procesora. Współczynnik 0,5 oznacza, że proces otrzymał tylko połowę z tego, co powinien był dostać, natomiast współczynnik 2,0 oznacza, że proces otrzymał dwa razy więcej niż to, do czego był

Kup ksi k Pole ksi k

uprawniony. Następnie program szeregujący uruchamia proces z najniższym współczynnikiem do czasu, kiedy współczynnik wzrośnie powyżej jego najbliższego konkurenta. Proces spełniający ten warunek jest uruchamiany jako następny.

Odmianę takiego algorytmu szeregowania zastosowano w algorytmie **CFS** (*Completely Fair Scheduling* — z ang. dosłownie: całkowicie sprawiedliwe szeregowanie) stosowanym w Linuksie. Działanie algorytmu opiera się na śledzeniu „realnego czasu wykonywania" procesów za pomocą drzewa czerwony-czarny. Skrajny lewy węzeł drzewa odpowiada procesowi o najmniejszym realnym czasie wykonywania. Program szeregujący indeksuje drzewo według czasu wykonania i wybiera do uruchomienia węzeł z lewej strony. Kiedy proces przestanie działać (z powodu wykorzystania swojego przedziału czasu, zablokowania lub przerwania), program szeregujący ponownie umieszcza go w drzewie na podstawie nowej wartości realnego czasu wykonywania.

Szeregowanie loteryjne

O ile składanie obietnic użytkownikom, a następnie ich dotrzymywanie jest dobrym pomysłem, o tyle odpowiadający temu algorytm jest trudny do zaimplementowania. Można jednak użyć innego algorytmu i uzyskać podobnie przewidywalne wyniki przy znacznie prostszej implementacji. Algorytm ten nazywa się **szeregowaniem loteryjnym** [Waldspurger i Weihl, 1994].

Podstawowa idea polega na przydzieleniu procesom biletów loteryjnych na różne zasoby systemowe, takie jak czas procesora. Zawsze, kiedy ma być podjęta decyzja dotycząca szeregowania, wybierany jest losowo żeton loteryjny, a zasób otrzymuje proces będący w posiadaniu tego żetonu. W przypadku szeregowania procesora system może przeprowadzać losowanie 50 razy na sekundę i w nagrodę przydzielać zwycięzcy 20 ms czasu procesora.

Sparafrazujmy powiedzenie George'a Orwella: „Wszystkie procesy są równe, ale niektóre procesy są bardziej równe". Ważniejszym procesom można przydzielić dodatkowe żetony i w ten sposób zwiększać ich szanse na zwycięstwo. Jeśli w grze jest 100 żetonów, a jeden proces ma ich 20, to ma 20% szans zwycięstwa w każdej loterii. W dłuższej perspektywie proces ten otrzyma około 20% czasu procesora. W odróżnieniu od szeregowania opartego na priorytetach, gdy bardzo trudno stwierdzić, co właściwie oznacza priorytet 40, w tym wypadku reguła jest czytelna: proces posiadający procent f żetonów otrzyma mniej więcej procent f wybranego zasobu.

Szeregowanie loteryjne ma kilka interesujących właściwości. Jeśli np. w grze pojawi się nowy proces, któremu zostanie przydzielona pewna pula żetonów, to w następnej loterii uzyska on szanse zwycięstwa proporcjonalnie do liczby posiadanych przez siebie żetonów. Inaczej mówiąc, szeregowanie loteryjne jest bardzo czułe.

Współpracujące ze sobą procesy mogą wymieniać między sobą żetony. Jeśli np. proces klienta wyśle komunikat do procesu serwera, a następnie się zablokuje, może przekazać wszystkie swoje żetony serwerowi i w ten sposób zwiększyć szanse na to, by serwer uruchomił się jako następny. Kiedy serwer zakończy pracę, zwraca żetony, dzięki czemu klient może wznowić działanie. W rzeczywistości, jeśli nie ma klientów, serwery w ogóle nie potrzebują żetonów.

Szeregowanie loteryjne można wykorzystać do rozwiązywania problemów, które trudno rozwiązać innymi metodami. Jednym z przykładów jest serwer wideo, w którym kilka procesów dostarcza strumienie wideo swoim klientom, ale z różnymi szybkościami odświeżania. Załóżmy, że procesy potrzebują ramek z szybkością 10, 20 i 25 ramek/s. Dzięki przydzieleniu tym procesom odpowiednio 10, 20 i 25 biletów automatycznie uzyskamy podział procesora w przybliżeniu we właściwej proporcji, tzn. 10:20:25.

Kup książkę Poleć książkę

Sprawiedliwe szeregowanie

Do tej pory zakładaliśmy, że każdy proces jest szeregowany „na własny rachunek”, bez względu na to, kto jest jego właścicielem. W rezultacie, jeśli użytkownik nr 1 uruchomił 9 procesów, a użytkownik nr 2 tylko 1 proces, to przy szeregowaniu cyklicznym lub przy równych priorytetach, użytkownik 1 otrzymałby 90% czasu procesora, a użytkownik 2 tylko 10%.

Aby zabezpieczyć się przed taką sytuacją, niektóre algorytmy szeregowania przed dokonaniem przydziału uwzględniają, do kogo należy proces. W tym modelu każdemu użytkownikowi przydzielany jest pewien fragment czasu procesora, a program szeregujący wybiera procesy w taki sposób, aby ten podział został uwzględniony. Tak więc, jeśli każdemu z dwóch użytkowników obiecano po 50% czasu procesora, to każdy po tyle otrzyma, niezależnie od te-go, ile uruchomili procesów.

Dla przykładu rozważmy system z dwoma użytkownikami, z których każdemu obiecano po 50% czasu procesora. Użytkownik nr 1 ma 4 procesy: A, B, C i D, a użytkownik 2 ma tylko 1 proces — E. Gdyby zastosowano szeregowanie cykliczne, to możliwa sekwencja szeregowania, która spełniłaby wszystkie ograniczenia, mogłaby mieć następującą postać:

```
A E B E C E D E A E B E C E D E ...
```

Jeśli natomiast użytkownik nr 1 byłby uprawniony do uzyskania dwa razy tyle czasu procesora co użytkownik 2, moglibyśmy otrzymać następującą sekwencję:

```
A B E C D E A B E C D E ...
```

Oczywiście istnieje wiele innych możliwości, które można wykorzystać. Wszystko zależy od tego, co rozumiemy pod pojęciem sprawiedliwości.

2.5.4. Szeregowanie w systemach czasu rzeczywistego

System **czasu rzeczywistego** to taki system, w którym czas odgrywa kluczową rolę. Zazwyczaj jedno lub kilka fizycznych urządzeń zewnętrznych generuje bodźce, a komputer musi na nie właściwie reagować w ciągu ustalonego czasu. Przykładowo komputer w odtwarzaczu płyt kompaktowych otrzymuje bity w miarę uzyskiwania ich z napędu i musi przetworzyć je na muzykę w ciągu bardzo krótkiego odcinka czasu. Jeśli obliczenia będą trwały zbyt długo, muzyka zabrzmi dziwnie. Innym przykładem systemów czasu rzeczywistego są systemy monitorujące pacjentów w szpitalach na oddziałach intensywnej terapii, systemy automatycznego pilotażu w samolotach oraz sterowania robotami w zautomatyzowanej fabryce. We wszystkich tych przypadkach otrzymanie prawidłowej odpowiedzi zbyt późno często jest tak samo złe, jak całkowity brak odpowiedzi.

Systemy czasu rzeczywistego ogólnie można podzielić na dwie kategorie: **twarde systemy czasu rzeczywistego**, gdzie występują ścisłe terminy, które koniecznie muszą być dotrzymane, oraz **miękkie systemy czasu rzeczywistego**, gdzie sporadyczne niedotrzymanie terminu jest niepożądane, niemniej jednak może być tolerowane. W obu przypadkach działanie w czasie rzeczywistym osiąga się poprzez podzielenie programu na szereg procesów. Działanie każdego z nich jest przewidywalne i z góry znane. Procesy te są, ogólnie rzecz biorąc, krótkotrwałe, a ich realizacja często zajmuje poniżej sekundy. W przypadku wykrycia zdarzenia zewnętrz-nego zadaniem programu szeregującego jest uszeregowanie procesów w taki sposób, aby były spełnione wszystkie terminy.

Zdarzenia, na które system czasu rzeczywistego musi odpowiadać, można podzielić na okresowe (występujące w regularnych odstępach czasu) lub nieokresowe (występujące w sposób nieprzewidywalny). System może być zmuszony do udzielania odpowiedzi na wiele okresowych strumieni

Kup ksi k Pole ksi k

zdarzeń. W zależności od tego, ile czasu potrzeba na przetwarzanie każdego zdarzenia, system może mieć trudności w obsłużeniu wszystkich zdarzeń. Jeśli np. jest m okresowych zdarzeń, a zdarzenie i występuje okresowo co P_i i wymaga C_i sekund procesora na obsługę, to obciążenie może być obsłużone tylko wtedy, gdy:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

System czasu rzeczywistego, spełniający to kryterium, określa się jako **szeregowalny** (ang. *schedulable*). Oznacza to, że może być praktycznie zaimplementowany. Proces, który nie przejdzie tego testu, nie może być zrealizowany, ponieważ całkowity czas procesora, którego procesy łącznie potrzebują, wynosi więcej, niż procesor CPU może dostarczyć.

Dla przykładu rozważmy miękki system czasu rzeczywistego z trzema okresowymi zdarzeniami, o okresach odpowiednio 100, 200 i 500 ms. Jeśli zdarzenia te wymagają odpowiednio 50, 30 i 100 ms czasu procesora na zdarzenie, to system jest szeregowalny, ponieważ $0{,}5 + 0{,}15 + 0{,}2 < 1$. Jeśli zostanie dodane czwarte zdarze-nie o okresie 1 s, to system pozostanie szeregowalny, o ile zdarzenie to nie będzie wymagało więcej niż 150 ms czasu procesora na zdarzenie. W tym obliczeniu przyjmuje się niejawne założenie, że koszt przełączania kontekstu jest tak niewielki, że można go pominąć.

Algorytmy szeregowania w systemach czasu rzeczywistego mogą być statyczne lub dynamiczne. Pierwsze z nich podejmują decyzje dotyczące szeregowania, zanim system rozpocznie działanie. Drugie podejmują decyzje o szeregowaniu podczas działania systemu. Statyczne szeregowanie działa tylko wtedy, gdy z góry istnieją dokładne informacje o tym, jakie prace są do wykonania oraz jakich terminów należy dotrzymać. Dynamiczne algorytmy szeregowania nie mają takich ograniczeń.

2.5.5. Oddzielenie strategii od mechanizmu

Do tej pory zakładaliśmy, że wszystkie procesy w systemie należą do różnych użytkowników, a w związku z tym rywalizują pomiędzy sobą o procesor. Choć często jest to prawda, czasami się zdarza, że jeden proces ma wiele dzieci działających pod jego kontrolą. Proces zarządzania bazą danych może mieć wiele dzieci. Każde dziecko może obsługiwać inne żądanie lub każde może mieć specyficzną funkcję do wykonania (parsowanie kwerend, dostęp do dysku itp.). Istnieje możliwość, że główny proces dokładnie wie, które z jego dzieci są najważniejsze (mają najbardziej ścisłe ograniczenia czasowe), a które najmniej ważne. Niestety, żaden z algorytmów szeregowania omówionych wcześniej nie uwzględnia informacji od procesów użytkownika podczas podejmowania decyzji związanych z szeregowaniem. W rezultacie programy szeregujące rzadko dokonują najlepszego wyboru.

Rozwiązaniem tego problemu jest oddzielenie **mechanizmu szeregowania** od **strategii szeregowania**. Zasada ta ma ugruntowaną pozycję od wielu lat [Levin et al., 1975]. Oznacza to, że algorytm szeregowania jest w pewien sposób sparametryzowany, ale parametry mogą być podawane przez procesy użytkownika. Rozważmy ponownie przykład z bazą danych. Przypuśćmy, że jądro używa algorytmu szeregowania z wykorzystaniem priorytetów, ale udostępnia wywołanie systemowe, dzięki któremu proces może ustawić (i zmienić) priorytety swoich dzieci. W ten sposób proces rodzic może szczegółowo kontrolować sposób szeregowania swoich dzieci, nawet jeśli sam nie realizuje szeregowania. W tym przypadku mechanizm znajduje się w jądrze, ale strategię ustalają procesy użytkownika. Kluczową koncepcją jest oddzielenie strategii od mechanizmu.

Kup ksi k Pole ksi k

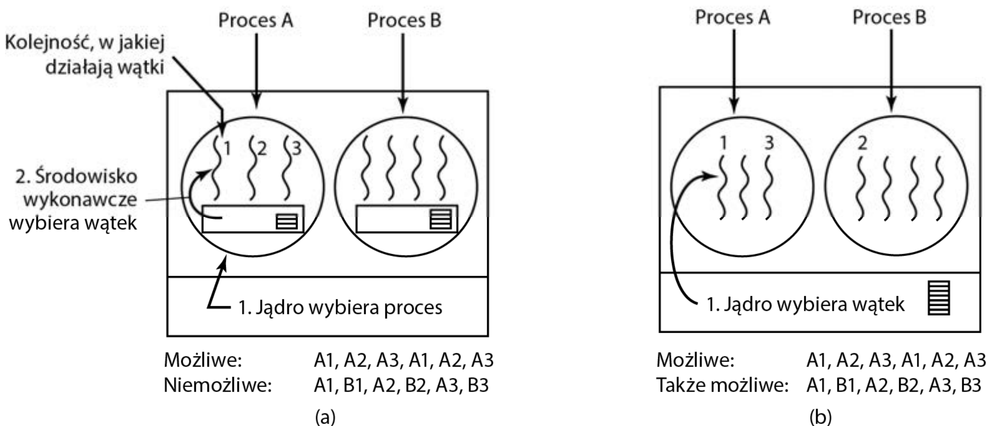
2.5.6. Szeregowanie wątków

Jeśli każdy z kilku procesów składa się z kilku wątków, mamy do czynienia z dwoma poziomami współbieżności: procesami i wątkami. Szeregowanie w takich systemach różni się znacząco w zależności od tego, czy są wykorzystywane wątki na poziomie użytkownika, wątki na poziomie jądra (czy oba rodzaje).

Rozważmy najpierw sytuację wątków na poziomie użytkownika. Ponieważ jądro nie jest świadome istnienia wątków, działa tak jak zawsze — wybiera proces, np. *A*, i przydziela mu sterowanie na ustalony kwant czasu. Program szeregowania wątków wewnątrz procesu A decyduje o tym, który wątek ma być uruchomiony, np. *A1*. Ponieważ nie ma przerwań zegara do wieloprogramowości wątków, wątek ten może działać tak długo, jak będzie chciał. Jeśli zużyje cały kwant, jądro wybierze inny proces do uruchomienia.

Kiedy proces *A* uruchomi się następnym razem, wątek *A1* wznowi swoje działanie. Będzie kontynuował korzystanie z czasu procesora *A* do momentu swojego zakończenia. Jego antyspołeczne zachowanie nie będzie jednak miało wpływu na pozostałe procesy. Procesy te otrzymają tyle, ile program szeregujący uzna za właściwe, niezależnie od tego, czy coś się będzie działo wewnątrz procesu *A*.

Rozważmy teraz przypadek, w którym wątki procesu *A* mają stosunkowo niewiele zadań do wykonania w ciągu jednego przydziału procesora — np. 5 ms pracy w czasie kwantu trwającego 50 ms. W konsekwencji każdy będzie działał przez chwilę, a następnie zwróci procesor do programu szeregującego wątki. Może to doprowadzić do sekwencji *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, po której jądro przełącza się do procesu *B*. Sytuację tę pokazano na rysunku 2.21(a).



Rysunek 2.21. (a) Możliwe uszeregowanie wątków zarządzanych na poziomie użytkownika w przypadku kwantu o czasie trwania 50 ms i wątkach działających przez 5 ms na jeden przydział procesora; (b) możliwe uszeregowanie wątków zarządzanych przez jądro przy tych samych parametrach co w przypadku (a)

Środowisko wykonawcze może wykorzystać dowolny z algorytmów szeregowania opisanych powyżej. W praktyce najczęściej stosowanymi algorytmami są szeregowanie cykliczne oraz szeregowanie oparte na priorytetach. Jedynym ograniczeniem jest brak przerwania zegarowego, które mogłoby wstrzymać wątek działający zbyt długo. Ponieważ wątki współpracują ze sobą, zazwyczaj taki problem nie występuje.

Kup ksi k Pole ksi k

Rozważmy teraz przypadek wątków zarządzanych na poziomie jądra. W tej sytuacji jądro wybiera określony wątek do uruchomienia. Nie musi przy tym brać pod uwagę, do jakiego procesu należy ten wątek, ale może to zrobić, jeśli tego chce. Wątek otrzymuje kwant czasu, a jeśli go przekroczy, jest przymusowo zawieszany. Przy kwancie o długości 50 ms i wątkach blokujących się po 5 ms kolejność wątków dla okresu 30 ms może być następująca: *A1*, *B1*, *A2*, *B2*, *A3*, *B3*. Taka kolejność nie jest możliwa przy tych samych parametrach i wątkach zarządzanych na poziomie użytkownika. Sytuację tę częściowo pokazano na rysunku 2.21(b).

Najważniejszą różnicą pomiędzy wątkami na poziomie użytkownika a wątkami na poziomie jądra jest wydajność. Wykonanie przełączania wątków w przypadku wątków zarządzanych na poziomie użytkownika zajmuje kilka instrukcji maszynowych. W przypadku wątków na poziomie jądra wymagane jest pełne przełączenie kontekstu, zmiana mapy pamięci i dezaktualizacja pamięci cache, co przebiega o kilka rzędów wielkości wolniej. Z drugiej strony, w przypadku wątków zarządzanych na poziomie jądra, blokada wątku na operacji wejścia-wyjścia nie powoduje zawieszenia całego procesu, jak to ma miejsce w przypadku wątków zarządzanych na poziomie użytkownika.

Ponieważ jądro wie, że przełączenie z wątku w procesie *A* do wątku w procesie *B* jest bardziej kosztowne niż uruchomienie drugiego wątku w procesie *A* (z uwagi na konieczność zmiany mapy pamięci oraz dezaktualizacji pamięci cache), przy podejmowaniu decyzji może wziąć pod uwagę te informacje. Jeśli np. istnieją dwa tak samo ważne wątki, przy czym jeden z nich należy do tego samego procesu co wątek, który się zablokował, a drugi należy do innego procesu, to pierwszeństwo może być udzielone temu pierwszemu.

Istotną rolę odgrywa również to, że wątki zarządzane na poziomie użytkownika mogą wykorzystywać mechanizm szeregowania specyficzny dla aplikacji. Rozważmy dla przykładu serwer WWW z rysunku 2.6. Załóżmy, że wątek pracownika właśnie się zablokował, a wątek dyspozytora i dwa wątki pracowników są gotowe. Który powinien zadziałać jako następny? Środowisko wykonawcze, wiedząc o tym, co robią wszystkie wątki, może z łatwością wybrać wątek dyspozytora, tak aby mógł uruchomić następnego pracownika. Taka strategia maksymalizuje współczynnik współbieżności w środowisku, w którym wątki pracowników często blokują się na dyskowych operacjach wejścia-wyjścia. Gdyby zostały zastosowane wątki na poziomie jądra, jądro nigdy nie wiedziałoby, co robi każdy z wątków (chociaż można by im było przypisać różne priorytety). Jednak ogólnie rzecz biorąc, mechanizmy szeregowania wątków na poziomie aplikacji potrafią dostroić aplikację lepiej, niż potrafi to zrobić jądro.

2.6. PRACE BADAWCZE NAD PROCESAMI I WĄTKAMI

W rozdziale 1. przyjrzeliśmy się wybranym pracom badawczym dotyczącym struktury systemów operacyjnych. W tym i w kolejnych rozdziałach przyjrzymy się bardziej ukierunkowanym badaniom, rozpoczniemy od procesów. Jak się okaże z czasem, niektóre zagadnienia mają bardziej ugruntowaną pozycję od innych. Znacznie więcej badań dotyczy nowych zagadnień. Zagadnienia obecne od dziesięcioleci są przedmiotem badań znacznie rzadziej.

Przykładem dość dobrze ugruntowanego tematu jest pojęcie procesu. Niemal w każdym systemie występuje pojęcie procesu rozumianego jako kontener pozwalający na grupowanie powiązanych ze sobą zasobów, takich jak przestrzeń adresowa, wątki, otwarte pliki, uprawnienia dostępu itp. W innych systemach grupowanie jest wykonywane nieco inaczej, ale są to jedynie różnice inżynierskie. Podstawowa idea nie jest zbyt kontrowersyjna, a tematowi procesów nie poświęca się zbyt wielu nowych badań.

Kup ksi k Pole ksi k

Wątki są nowszym mechanizmem niż procesy, ale i one są obecne już od dość dawna. Pomimo to od czasu do czasu pojawia się artykuł poświęcony wątkom — np. na temat klasteryzacji wątków w systemach wieloprocesorowych [Quin et al., 2019] lub skalowania w nowoczesnych systemach operacyjnych z obsługą wielu wątków i wielu rdzeni, takich jak Linux [Boyd-Wickizer, 2010].

Ponadto pojawiło się wiele publikacji zajmujących się udowodnieniem braku ujemnego wpływu współbieżności na działanie algorytmów, np. w systemach plików [Chajed et al., 2019] i [Zou et al., 2019], a także innych usługach [Setty et al., 2018] i [Li et al., 2019]. Są to ważne publikacje, ponieważ badacze wykazali, że błędy współbieżności są niestety niezwykle powszechne [Li et al., 2019]. Jak się przekonaliśmy, blokowanie jest nie tylko trudne, ale także kosztowne, a w systemach operacyjnych zastosowano struktury danych RCU, aby całkowicie uniknąć blokowania [McKenney et al., 2013].

Aktywnym obszarem badań jest rejestrowanie i odtwarzanie przebiegu procesu [Viennot et al., 2013]. Odtwarzanie pomaga programistom wyśledzić trudne do znalezienia błędy, a ekspertom w dziedzinie bezpieczeństwa badać incydenty.

Jeśli chodzi o bezpieczeństwo, ważnym wydarzeniem w 2018 roku było ujawnienie szeregu bardzo poważnych luk w zabezpieczeniach nowoczesnych procesorów. Ze względu na te luki wystąpiła potrzeba wprowadzenia zmian w wielu miejscach: w sprzęcie, oprogramowaniu firmware, systemach operacyjnych, a nawet w aplikacjach. W tym rozdziale szczególną uwagę zwrócono na implikacje dotyczące szeregowania. Przykładowo w systemie Windows zastosowano algorytm szeregowania, którego celem jest zapobieżenie współdzielenia przez kod w różnych domenach bezpieczeństwa tego samego rdzenia procesora [Microsoft, 2018].

Szeregowanie (zarówno w systemach jednoprocesorowych, jak i wieloprocesorowych) w dalszym ciągu jest zagadnieniem znajdującym się w kręgu zainteresowania badaczy. Niektóre badane dotyczą szeregowania w klastrach na potrzeby uczenia głębokiego [Xiao et al., 2018], szeregowania w kontekście mikrousług [Sriraman, 2018] i szeregowalności [Yang et al., 2018]. Ogólnie rzecz biorąc, procesy, wątki i szeregowanie nie są już tak gorącymi tematami badań, jak w przeszłości. Przedmiot badań zmienił się w kierunku takich zagadnień jak zarządzanie energią, wirtualizacja, przetwarzanie w chmurze i zabezpieczenia.

2.7. PODSUMOWANIE

W celu ukrycia efektu przerwań systemy operacyjne dostarczają pojęciowego modelu składającego się z sekwencyjnych procesów działających współbieżnie. Procesy można tworzyć i niszczyć dynamicznie. Każdy proces ma własną przestrzeń adresową.

W wypadku niektórych aplikacji przydatne jest istnienie w obrębie pojedynczego procesu wielu wątków sterowania. Wątki te są szeregowane niezależnie, a każdy z nich ma własny stos, choć wszystkie wątki w procesie współdzielą wspólną przestrzeń adresową. Wątki mogą być implementowane na poziomie przestrzeni użytkownika lub na poziomie jądra.

Alternatywą dla wątków, stosowaną w odniesieniu do serwerów o dużej przepustowości, są systemy oparte na modelu sterowanym zdarzeniami. W tym wypadku serwer działa jako automat o skończonej liczbie stanów, który reaguje na zdarzenia i wchodzi w interakcje z systemem operacyjnym za pomocą nieblokujących wywołań systemowych.

Procesy mogą synchronizować się ze sobą z wykorzystaniem prymitywów synchronizacji i komunikacji między procesami, takich jak semafory, monitory lub komunikaty. Prymitywy te wykorzystuje się po to, by zapewnić, że żadne dwa procesy nigdy nie znajdą się w swoich regionach krytycznych w tym samym czasie — taka sytuacja prowadzi bowiem do chaosu. Proces może

Kup ksi k Pole ksi k

działać, być w stanie gotowości do działania lub zablokowania. Status procesu może się zmienić, kiedy ten proces lub jakiś inny proces wykonają jeden z prymitywów komunikacji między procesami. Na podobnej zasadzie działa komunikacja między wątkami.

W tym rozdziale przeanalizowaliśmy wiele algorytmów szeregowania. Niektóre z nich są używane głównie w systemach wsadowych — np. szeregowanie w pierwszej kolejności najkrótszego zadania. Inne wykorzystuje się powszechnie zarówno w systemach wsadowych, jak i w interaktywnych. Do tej grupy należy szeregowanie cykliczne, szeregowanie oparte na priorytetach, wielopoziomowe kolejki, szeregowanie gwarantowane i szeregowanie według sprawiedliwego przydziału. W niektórych systemach istnieje czytelna granica pomiędzy mechanizmami szeregowania a strategią szeregowania. Dzięki temu podziałowi użytkownicy mogą kontrolować algorytm szeregowania.

PYTANIA

1. Na rysunku 2.2 pokazano trzy stany procesu. Teoretycznie przy trzech stanach może być sześć przejść — po dwa dla każdego ze stanów. Pokazano jednak tylko cztery przejścia. Czy istnieją jakieś okoliczności, w których może wystąpić jedno brakujące przejście lub oba takie przejścia?
2. Przypuśćmy, że masz zaprojektować zaawansowaną architekturę komputerową, w której przełączanie procesów jest realizowane na poziomie sprzętu, a nie przerwań. Jakich informacji będzie potrzebował procesor? Opisz, w jaki sposób może działać sprzętowe przełączanie procesów.
3. We wszystkich współczesnych komputerach przynajmniej pewna część procedur obsługi przerwań jest napisana w języku asemblera. Dlaczego?
4. Kiedy przerwanie lub wywołanie systemowe przekazują sterowanie do systemu operacyjnego, zazwyczaj używany jest obszar stosu jądra oddzielny od stosu przerwanego procesu. Dlaczego?
5. System komputerowy ma wystarczająco dużo miejsca w pamięci głównej, aby pomieścić cztery programy. Przez połowę czasu programy te są w stanie oczekiwania na dostępność urządzeń wejścia-wyjścia. Jaki ułamek czasu procesora jest marnotrawiony?
6. Komputer jest wyposażony w 2 GB pamięci RAM, z której system operacyjny zajmuje 256 MB. Każdy proces zajmuje 128 MB (dla uproszczenia). Wszystkie procesy mają te same własności. Jaki jest maksymalny czas oczekiwania na urządzenia wejścia-wyjścia, jeśli celem jest 99% wykorzystania procesora?
7. Jeśli wiele zadań działa współbieżnie, ich realizacja może zakończyć się szybciej w porównaniu z sytuacją, kiedy działałyby one sekwencyjnie. Przypuśćmy, że dwa zadania, z których każde wymaga 10 min czasu procesora, rozpoczyna się równocześnie. Ile czasu zajmie wykonanie ostatniego, jeśli będą działały sekwencyjnie? A ile, jeśli będą działały współbieżnie? Zakładany czas oczekiwania na urządzenia wejścia-wyjścia wynosi 50%.
8. Rozważmy system wieloprogramowy 5. stopnia (tzn. w tym samym czasie w pamięci jest pięć programów). Załóżmy, że każdy proces spędza 40% swojego czasu w oczekiwaniu na urządzenia wejścia-wyjścia. Ile wynosi procent wykorzystania procesora?
9. Wyjaśnij, w jaki sposób przeglądarka internetowa może skorzystać z pojęcia wątków w celu poprawy wydajności.

Kup ksi k Pole ksi k

10. Załóżmy, że próbujesz pobrać z internetu duży plik o rozmiarze 2 GB. Plik jest dostępny z kilku serwerów lustrzanych, z których każdy może dostarczyć podzbiór bajtów pliku. Zakładamy, że w określonym żądaniu są określone początkowe i końcowe bajty pliku. Wyjaśnij, w jaki sposób można użyć wątków do poprawy czasu pobierania.
11. W tekście rozdziału powiedziano, że model z rysunku 2.7(a) nie był odpowiedni dla serwera plików wykorzystującego buforowanie w pamięci. Dlaczego nie? Czy każdy proces mógłby mieć własną pamięć cache?
12. Na rysunku 2.6 pokazano serwer WWW z obsługą wielu wątków. Jeśli jedynym sposobem czytania z pliku jest normalne blokujące wywołanie systemowe `read`, to jak sądzisz, czy dla serwera WWW są wykorzystywane wątki zarządzane na poziomie użytkownika, czy na poziomie jądra? Dlaczego?
13. W tekście rozdziału opisaliśmy wielowątkowy serwer WWW i pokazaliśmy, dlaczego jest on lepszy od jednowątkowego serwera oraz serwera działającego na zasadzie automatu o skończonej liczbie stanów. Czy istnieją jakieś okoliczności, w których jednowątkowy serwer może być lepszy? Podaj przykład.
14. W tabeli 2.3 zbiór rejestrów wyszczególniono jako komponent wątku, a nie procesu. Dlaczego? W końcu maszyna ma tylko jeden zbiór rejestrów.
15. Dlaczego wątek miałby kiedykolwiek dobrowolnie oddać procesor za pomocą wywołania `thread_yield`? Przecież skoro nie ma okresowych przerwań zegara, to może się zdarzyć, że nigdy nie odzyska procesora.
16. Twoim zadaniem jest porównanie operacji czytania z pliku z wykorzystaniem jednowątkowego serwera plików oraz serwera wielowątkowego. Pobranie żądania pracy, przydzielenie go i wykonanie reszty obliczeń, przy założeniu, że potrzebne dane znajdują się w bloku pamięci cache, zajmuje 15 ms. Jeśli jest potrzebna operacja dyskowa, co zdarza się w jednej trzeciej przypadków, potrzeba kolejnych 75 ms, w ciągu których wątek jest uśpiony. Ile żądań na sekundę może obsłużyć serwer, jeśli jest jednowątkowy? A ile, jeśli jest wielowątkowy?
17. Jaka jest największa zaleta implementacji wątków w przestrzeni użytkownika? A jaka jest największa wada tego sposobu implementacji?
18. W kodzie na listingu 2.2 operacje tworzenia wątków i wyświetlania komunikatów przez wątki losowo przeplatają się. Czy istnieje sposób wymuszenia następującej sekwencji operacji: utworzenie wątku 1, wątek 1 wyświetla komunikat, wątek 1 kończy działanie, utworzenie wątku 2, wątek 2 wyświetla komunikat, wątek 2 kończy działanie itd.? Jeśli tak, to jak to można zrobić? Jeśli nie, to dlaczego?
19. Załóżmy, że program ma dwa wątki, z których każdy wykonuje pokazaną poniżej funkcję `get_account`. Zidentyfikuj sytuację wyścigu w tym kodzie.

```
int accounts[LIMIT]; int account  count = 0;

void *get_account(void *tid) {
    char *lineptr = NULL;
    size t len = 0;

    while (account  count < LIMIT)
    {
        // Odczytaj dane wprowadzone przez użytkownika z terminala i zapisz je w lineptr
        getline(&lineptr, &len, stdin);
```

Kup ksi k Pole ksi k

```
    // Konwertuj dane wejściowe użytkownika na liczbę całkowitą
    // Załóżmy, że użytkownik wprowadził prawidłową liczbę całkowitą
        int entered_account = atoi(lineptr);

        accounts[account_count] = entered_account;
        account_count++;
    }
  // Zwolnij pamięć, która została przydzielona przez wywołanie getline
    free(lineptr);
    retur n NULL; }
```

20. Podczas omawiania zmiennych globalnych w wątkach użyliśmy procedury `create_global` w celu zaalokowania pamięci na wskaźnik do zmiennej zamiast do samej zmiennej. Czy ma to istotne znaczenie, czy też procedury mogą równie dobrze działać na samych wartościach?
21. Rozważmy system, w którym wątki są implementowane w całości w przestrzeni użytkownika, gdzie środowisko wykonawcze obsługuje przerwanie zegara co sekundę. Przypuśćmy, że przerwanie zegarowe występuje w momencie, kiedy jakiś wątek w środowisku wykonawczym jest w stanie, w którym chce się zablokować lub odblokować. Jaki problem może się zdarzyć? Czy możesz zaproponować sposób jego rozwiązania?
22. Przypuśćmy, że w systemie operacyjnym nie ma czegoś takiego, jak wywołanie systemowe `select`, które może sprawdzić, czy odczyt z pliku, potoku lub urządzenia jest bezpieczny, ale istnieje możliwość ustawiania zegarów alarmowych, które przerywają zablokowane wywołania systemowe. Czy możliwe jest zaimplementowanie w przestrzeni użytkownika pakietu wątków, który nie będzie blokował wszystkich wątków, gdy jeden wątek wykona wywołanie systemowe, które może się zablokować? Uzasadnij odpowiedź.
23. Czy rozwiązanie problemu wzajemnego wykluczania Petersona, które pokazano na listingu 2.4, działa w przypadku wykorzystania szeregowania procesów z wywłaszczaniem? A co w przypadku zastosowania szeregowania bez wywłaszczania?
24. Czy problem inwersji priorytetów omówiony w punkcie 2.3.4 może się zdarzyć w przypadku wątków zarządzanych na poziomie użytkownika? Dlaczego tak lub dlaczego nie?
25. W punkcie 2.3.4 opisano sytuację z procesem o wysokim priorytecie H oraz niskim priorytecie L. Doprowadziło to do tego, że proces H wykonywał się w pętli nieskończonej. Czy taki sam problem występuje wtedy, gdy zamiast szeregowania opartego na priorytetach jest stosowane szeregowanie cykliczne? Uzasadnij.
26. Czy w systemie z wątkami zarządzanymi na poziomie użytkownika występuje jeden stos na wątek, czy jeden stos na proces? A jak wygląda sytuacja, jeśli wątki są zarządzane na poziomie jądra? Wyjaśnij.
27. Co to jest wyścig?
28. Kiedy projektuje się komputer, zazwyczaj najpierw przeprowadza się jego symulację za pomocą programu działającego po jednej instrukcji na raz. Nawet systemy wieloprocesorowe są symulowane w ten sposób, ściśle sekwencyjnie. Czy istnieje możliwość wystąpienia wyścigu, jeśli nie ma jednoczesnych zdarzeń, tak jak to ma miejsce w tym wypadku? Wyjaśnij.
29. Problem producent-konsument może być rozszerzony do systemu z wieloma producentami i konsumentami, które zapisują (lub odczytują) do (z) wspólnego bufora. Załóżmy, że każdy producent i konsument działa we własnym wątku. Czy rozwiązanie przedstawione na listingu 2.8 z wykorzystaniem semaforów sprawdzi się w tym systemie?

Kup ksi k Pole ksi k

30. Rozważmy następujące rozwiązanie problemu wzajemnego wykluczania z udziałem dwóch procesów *P0* i *P1*. Załóżmy, że zmienna `turn` jest inicjowana wartością 0. Kod procesu *P0* zamieszczono poniżej.

```
/* inny kod */

while (turn != 0) { } /* Nic nie rób i czekaj.*/
Sekcja krytyczna /*... */
turn = 0;
/* inny kod */
```

   W procesie *P1* w powyższym kodzie należy zastąpić 0 wartością 1. Ustal, czy rozwiązanie spełnia wszystkie wymagane warunki do prawidłowego rozwiązania problemu wzajemnego wykluczania.

31. Pokaż sposób implementacji semaforów zliczających (tzn. semaforów zdolnych do przechowywania dowolnych wartości) z wykorzystaniem semaforów binarnych oraz standardowych instrukcji maszynowych.

32. Jeśli w systemie działają tylko dwa procesy, to czy jest sens, aby wykorzystywać barierę do ich synchronizacji? Dlaczego tak lub dlaczego nie?

33. Czy dwa wątki w tym samym procesie można zsynchronizować z wykorzystaniem semafora w jądrze, jeśli wątki są zarządzane na poziomie jądra? A co w przypadku zaimplementowania ich w przestrzeni użytkownika? Załóżmy, że żaden wątek należący do innego procesu nie ma dostępu do semafora. Uzasadnij swoje odpowiedzi.

34. Przypuśćmy, że mamy system przekazywania wiadomości korzystający ze skrzynek pocztowych. Podczas wysyłania wiadomości do pełnej skrzynki pocztowej lub przy próbie odbierania wiadomości z pustej skrzynki pocztowej proces się nie blokuje. Za-miast tego otrzymuje kod błędu. Proces odpowiada na błąd poprzez wielokrotne ponawianie próby — tak długo, aż się powiedzie. Czy taki schemat prowadzi do sytuacji wyścigu?

35. Komputery CDC 6600 mogą obsłużyć jednocześnie do 10 procesów wejścia-wyjścia z wykorzystaniem interesującej formy szeregowania cyklicznego, zwanej współdzieleniem procesora. Po każdej instrukcji wystąpiło przełączenie procesów, zatem instrukcja 1 pochodziła z procesu 1, instrukcja 2 pochodziła z procesu 2 tp.. Przełączanie procesów było realizowane za pomocą specjalnego sprzętu, a koszt obliczeniowy tej operacji był zerowy. Jeśli w warunkach braku rywalizacji wykonanie procesu wymagało T sekund, to ile czasu wymagałoby, gdyby wykorzystano współdzielenie procesora z n procesami?

36. Rozważmy następujący fragment kodu w języku C:

```
void main( ) {
    fork();
    fork();
    exit();
}
```

   Ile procesów potomnych zostanie stworzonych w wyniku uruchomienia tego programu?

37. Cykliczne programy szeregujące zwykle utrzymują listę wszystkich procesów zdolnych do uruchomienia, przy czym każdy proces występuje na liście tylko raz. Co by się stało, gdyby proces występował na liście dwukrotnie? Czy potrafisz wskazać jakiekolwiek powody, aby na to pozwolić?

Kup ksi k Pole ksi k

38. Czy na podstawie analizy kodu źródłowego można stwierdzić, czy proces jest zorientowany na procesor, czy na operacje wejścia-wyjścia? W jaki sposób można to sprawdzić na etapie działania programu?
39. W podpunkcie „Kiedy wykonywać szeregowanie" wspomniano, że czasami, kiedy ważny proces, gdy się zablokuje, może odgrywać rolę w wyborze następnego procesu do uruchomienia, można poprawić szeregowanie. Podaj sytuację, w której można z tego skorzystać, i wyjaśnij, w jaki sposób.
40. Wyjaśnij, jaki wpływ mają na siebie wartość kwantu czasu i czas przełączania kontekstu w cyklicznym algorytmie szeregowania.
41. Pomiary wykonane w pewnym systemie pokazały, że przeciętny proces działa przez czas T, a następnie blokuje się na operacji wejścia-wyjścia. Przełączenie procesu wymaga czasu S, który jest tracony (koszty obliczeniowe). Dla szeregowania cyklicznego o kwancie Q podaj wzór na wydajność procesora dla każdej z poniższych sytuacji:

    (a) $Q = \infty$
    (b) $Q > T$
    (c) $S < Q < T$
    (d) $Q = S$
    (e) Q jest bliskie zeru

42. Na uruchomienie oczekuje pięć zadań. Ich spodziewane czasy działania wynoszą 9, 6, 3, 5 i X. W jakiej kolejności powinny one działać, aby zminimalizować średni czas odpowiedzi (odpowiedź zależy od X)?
43. Pięć zadań wsadowych od A do E wpłynęło do ośrodka obliczeniowego niemal w tym samym czasie. Szacowany czas ich działania wynosi odpowiednio 10, 6, 2, 4 i 8 min. Ich priorytety (określane zewnętrznie) wynoszą odpowiednio 3, 5, 2, 1 i 4, przy czym 5 oznacza najwyższy priorytet. Dla każdego z poniższych algorytmów szeregowania określ średni czas przełączania cyklu procesu. Zignoruj koszty obliczeniowe związane z przełączaniem procesów.

    (a) Szeregowanie cykliczne
    (b) Szeregowanie oparte na priorytetach
    (c) Pierwszy zgłoszony — pierwszy obsłużony (uruchamianie w porządku 10, 6, 2, 4, 8)
    (d) Najpierw najkrótsze zadanie

    Dla przypadku (a) załóż, że system jest wieloprogramowy oraz że każde zadanie otrzymuje sprawiedliwy przydział procesora. Dla przypadków od (b) do (d) załóż, że w określonym czasie działa tylko jedno zadanie do momentu, aż się zakończy. Wszystkie zadania są całkowicie zorientowane na obliczenia.

44. Proces działający w systemie CTSS wymaga do realizacji 30 kwantów. Ile razy będzie musiał być wymieniany pomiędzy dyskiem a pamięcią, jeśli uwzględnić pierwszą wymianę (jeszcze przed uruchomieniem)?
45. Czy potrafisz wskazać sposób, jak uniemożliwić oszukanie systemu priorytetów CTSS przez losowe znaki powrotu karetki?
46. Rozważmy system czasu rzeczywistego z dwoma połączeniami głosowymi co 5 ms każde z czasem procesora na połączenie wynoszącym 1 ms i jednym strumieniem wideo co 33 ms z czasem procesora na wywołanie wynoszącym 11 ms. Czy ten system jest szeregowalny? Wyjaśnij, w jaki sposób uzyskałeś tę odpowiedź.

Kup ksi k Pole ksi k

47. Czy w systemie opisanym powyżej można dodać kolejny strumień wideo, jeśli system nadal ma być szeregowalny?
48. Do przewidywania czasów działania procesów wykorzystywany jest algorytm starzenia z $a = 1/2$. Czasy poprzednich czterech uruchomień — od najstarszego do najświeższego — to odpowiednio 40, 20, 40 i 15 ms. Jaka jest prognoza następnego czasu uruchomienia?
49. W miękkim systemie czasu rzeczywistego występują cztery okresowe zdarzenia o okresach 50, 100, 200 i 250 ms każdy. Przypuśćmy, że cztery zdarzenia wymagają odpowiednio 35, 20, 10 i x ms czasu procesora. Jaka jest największa wartość x dla systemu szeregowalnego?
50. Wyjaśnij, dlaczego powszechnie stosuje się szeregowanie dwupoziomowe. Jakie ma ono zalety w porównaniu z planowaniem jednopoziomowym?
51. System czasu rzeczywistego musi obsługiwać dwa połączenia głosowe, z których każde jest uruchamiane co 5 ms i zużywa 1 ms czasu procesora, a także jeden strumień wideo o szybkości 25 klatek/s, przy czym każda klatka wymaga 20 ms czasu procesora. Czy ten system jest szeregowalny? Wyjaśnij, dlaczego jest szeregowalny lub dlaczego nie jest szeregowany i jak doszedłeś do tego wniosku.
52. Rozważ system, w którym pożądane jest oddzielenie strategii od mechanizmu szeregowania wątków zarządzanych na poziomie jądra. Zaproponuj sposoby osiągnięcia tego celu.
53. Problem czytelników i pisarzy można sformułować na kilka sposobów w zależności od tego, kiedy powinny się uruchomić poszczególne kategorie procesów. Uważnie opisz trzy różne odmiany problemu dla przypadków faworyzowania poszczególnych kategorii procesów (tzn. czytelników lub pisarzy). Dla każdej odmiany określ, co się stanie, kiedy czytelnik lub pisarz osiągnie gotowość dostępu do bazy danych, a co się stanie, kiedy proces zakończy korzystanie z bazy danych.
54. Napisz skrypt powłoki, który generuje plik sekwencyjnych liczb poprzez odczytanie ostatniej liczby w pliku, dodanie do niej jedynki, a następnie dołączenie jej do pliku. Uruchom jeden egzemplarz skryptu w tle i jeden na pierwszym planie, tak aby każdy z nich korzystał z tego samego pliku. Ile czasu upłynie, zanim da o sobie znać sytuacja wyścigu? Co jest regionem krytycznym? Zmodyfikuj skrypt w taki sposób, aby zapobiec wyścigowi. (Wskazówka: skorzystaj z polecenia

   ```
   ln file file.lock
   ```

   w celu zablokowania pliku danych).
55. Przypuśćmy, że mamy do czynienia z systemem operacyjnym, który udostępnia semafory. Zaimplementuj system komunikatów. Napisz procedury wysyłania i odbierania komunikatów.
56. Przepisz program z listingu 2.3 w taki sposób, aby obsługiwał więcej niż dwa procesy.
57. Napisz program rozwiązujący problem producent-konsument. Program powinien wykorzystywać wątki i wspólny bufor. Do kontrolowania współdzielonych danych nie korzystaj z semaforów ani innych prymitywów synchronizacji. Po prostu pozwól wątkom na korzystanie z tych danych, jeśli tego zażądają. Wykorzystaj operacje `sleep` i `wakeup` do obsługi warunków „pełny” i „pusty”. Zobacz, ile czasu upłynie, zanim wystąpi sytuacja wyścigu. Możesz np. polecić producentowi, by co jakiś czas wyświetlał liczbę. Nie wyświetlaj więcej niż jednej liczby co minutę, ponieważ operacje wejścia-wyjścia mogą wpłynąć na sytuację wyścigu.

Kup ksi k Pole ksi k

58. Proces może być wprowadzony do kolejki cyklicznej więcej niż jeden raz w celu nadania mu wyższego priorytetu. Taki sam skutek może mieć uruchomienie wielu wystąpień programu, z których każde pracuje na innej części puli danych. Najpierw napisz program, który sprawdza, czy wartości z listy są liczbami pierwszymi. Następnie opracuj metodę, która umożliwia wielu instancjom programu na jednoczesne działanie w taki sposób, aby żadne dwa wystąpienia programu nie sprawdzały tej samej wartości. Czy równoległe uruchomienie wielu kopii programu pozwala na szybsze przetwarzanie listy? Zwróćmy uwagę, że wyniki będą zależały od tego, jakie inne operacje wykonuje komputer. W komputerze osobistym, na którym działa tylko jedno wystąpienie programu, nie należy spodziewać się poprawy, ale w systemie z innymi procesami w ten sposób powinno udać się uzyskać większy udział czasu procesora.
59. Napisz program zliczający częstość słów w pliku tekstowym. Plik tekstowy jest podzielony na N segmentów. Każdy segment jest przetwarzany przez oddzielny wątek, który zwraca pośrednie wartości liczby częstości dla segmentu. Główny proces czeka na zakończenie wszystkich wątków. Następnie oblicza łączną częstość danego słowa na podstawie wyników poszczególnych wątków.

Kup ksi k Pole ksi k

SKOROWIDZ

A

abstrakcja pamięci, 198
  przestrzenie adresowe, 201
ACE, Access Control Entries, 1006
ACL, Access Control List, 618, 878
ACPI, Advanced Configuration and Power Interface, 432
adres
  IP, 585
  URL, 587
  wirtualny, 211
AIDL, Android Interface Definition Language, 807
aktywność, activity, 811
  ResolverActivity, 820
algorytm
  alokacji procesorów, 295, 575
  bankiera, 462, 463
  bazujący na zbiorze roboczym, 230
  bliźniaków, 746
  częstości błędów braku stron, 238
  Dekkera, 144
  drugiej szansy, 226
  FIFO, 226
  lokalny zastępowania stron, 237
  LRU, 228
  NRU, 225
  odbierania ramek stron, 748
  Petersona, 144
  postarzania, 229
  SSF, 383
  strusia, 453
  szeregowania ramienia dysku, 383
  szeregujący CFS, 732, 733
  unikania zakleszczeń, 460, 461
  windy, 384
  WSClock, 233
  współdzielenia przestrzeni, 554
  wykrywania zakleszczeń, 456, 457
  wymiany stron, 749
  zastępowania ramek stron, 751
  zastępowania stron, 224, 235, 953
  zegarowy, 227
  zrzutu logicznego, 322
algorytmy szeregowania, 170, 553
  bez wywłaszczania, 173
  całkowicie sprawiedliwe szeregowanie, 183
  cele, 174
  kategorie, 173
  najpierw najkrótsze zadanie, 177
  następny najkrótszy proces, 182
  pierwszy zgłoszony, pierwszy obsłużony, 177
  z klasami priorytetów, 181
  z wywłaszczaniem, 173
alokacja
  pamięci, 204, 237
  przestrzeni dyskowej, 980

Kup książkę
Poleć książkę

Android, 786
  aktywność, activity, 811
  aplikacje, 809
  architektura, 794
  ART, 799
  bezpieczeństwo, 825
  Binder IPC, 801
  blokady WakeLock, 796
  cele projektowe, 792
  cykl życia procesu, 822
  dostawcy zawartości, 818
  drzemka, 848
  ewolucja uprawnień, 839
  lista uprawnień, 835, 837
  menedżer pakietów, 810
  odbiorcy, 817
  okna konserwacji, 848
  piaskownice aplikacji, 825
  prywatność, 825, 833
  publikacje, 1080
  SELinux, 832
  stany istotności procesu, 824
  uprawnienia, 827, 833
  uruchamianie procesów, 821
  uruchamianie w tle, 843
  usługi, 815
  wersje systemu, 791
  zależności pomiędzy procesami, 823
  zamiary, 819
API, Application Program Interface, 83, 493
aplikacje Win32, 875
architektura, 30, 46
  Androida, 794
  magistrali, 532
    równoległej, 57
    współdzielonej, 57
  oznaczona, tagged architecture, 622
archiwizowanie instrukcji, 249
ART, Android RunTime, 799
ASLR, Address Space Layout Randomization, 650
asynchroniczny transfer, 362
ATA, AT Attachment, 54
atak typu
  „użyj po zwolnieniu”, 657
  DoS, 606
  łańcuch formatujący, 653
  Meltdown, 670
  odwołanie do pustego wskaźnika, 659
  powrotu do biblioteki libc, 649
  przejściowe wykonywanie, 668
  przepełnienie bufora, 644, 652
  przepełnienie liczb całkowitych, 660
  TOCTOU, 661
  wstrzyknięcie kodu, 648, 660
atrybuty plików, 281
automat o skończonej liczbie stanów, 136

B

balonikowanie, balooning, 499
bariera pamięci, 165, 167
BCD, Boot Configuration Database, 900
Berkeley UNIX, 701
bezpieczeństwo, 603
  ataki, 644, 652, 653, 657–661, 668
  ataki z wewnątrz, 673
  badania, 687
  bezpieczny system, 614
  bomby logiczne, 673
  czynniki ograniczające zagrożenia, 1011
  domeny ochrony, 615
  dostępność, availability, 606
  eksploity sprzętowe, 663
  eliminowanie luk w zabezpieczeniach, 1011
  fałszywy ekran logowania, 674
  hasła, 634
    jednorazowe, 638
    w systemie UNIX, 636
  hermetyzacja niezaufanego kodu, 684
  integralność, integrity, 606
  intruzi, 611
  kontrola integralności kodu, 682
  kryptografia, 628
  listy kontroli dostępu, 618, 878
  luki
    typu Type Confusion, 657
    w oprogramowaniu, 643
  macierze ochrony, 624
  model
    Bella-La Paduli, 626
    Biby, 628
  moduł TPM, 632, 683
  ograniczenia
    dostępu, 679
    przepływu sterowania, 677
  oparte na wirtualizacji, 1002
  oprogramowanie Windows Defender, 1020
  piaskownice, 481, 685, 825, 832, 1001

bezpieczeństwo
- podpisy cyfrowe, 631
- poufność, confidentiality, 606
- publikacje, 1079
- struktury systemu operacyjnego, 608
- systemy zaufane, 610
- technika ASLR, 650, 677
- tylne drzwi, 674
- ukryte kanały komunikacyjne, 663
- uprawnienia, 621
- uwierzytelnianie, 633, 639–642
- w systemie
  - Android, 825
  - Linux, 783
  - Windows, 1004, 1008, 1016
- wielopoziomowe, 626
- wywołania
  - API, 1007
  - systemowe Linuksa, 785
- zasady, 607

bezpieczny rozruch, Secure Boot, 60
bezpośredni dostęp do pamięci, DMA, 354
biblioteki
- ładowane dynamicznie, DLL, 86, 244
- współdzielone, shared libraries, 86, 244

Binder
- hierarchia dziedziczenia interfejsu, 807
- interfejs API, 806
- IPC, 801
- komunikacja IPC, 802
- mapowanie obiektów, 804
- moduł jądra, 802
- podstawowa transakcja, 803
- transfer obiektów, 804

BIOS, Basic Input/Output System, 51, 58, 198, 291
bit zabrudzenia, 215
bity rwx, 69
blok
- dwupośredni, 338
- jednopośredni, 338
- PEB, 916
- TEB, 916

blokada, 145, 168
- PushLock, 927
- SRW, 926
- WakeLock, 796

blokady
- dostawcy, 505
- pętlowe, spin locks, 143, 548
- współdzielone, shared locks, 764
- wyłączne, exclusive locks, 764

blokowanie, locking, 763
- dwufazowe, two-phase locking, 468
- strony, 250

błędy, 357
- braku strony, 213, 238, 248, 949
- w kodzie, 643
- wejścia-wyjścia, 375

bomba logiczna, logic bomb, 673
bufor
- cykliczny, 373
- TLB, 217, 247, 543, 553, 557, 948
  - programowe zarządzanie, 219

buforowanie, caching, 326, 362, 372, 1061
- podwójne, 373

C

certyfikat główny, root certificate, 900
CFI, Control-Flow Integrity, 678
chmura, 481
- badania, 523
- jako usługa, 504
- obliczeniowa, 483, 504
- publikacje, 1077

cienka alokacja, thin provisioning, 966
cienki klient, 424
CMP, chip multiprocessors, 539
COM, Component Object-Model, 879
czas
- cyklu przetwarzania, 175
- odpowiedzi, 176

czcionka, 421
czyszczenie, 240
- adresów, address sanitizers, 1011

D

DAX, Direct Access for files, 758
deduplikacja, 240, 331, 503
definiowanie obiektów, 906
defragmentacja, 330
demon, 110, 377
- stron, page daemon, 240, 749

deskryptor
adresu wirtualnego, 948
bezpieczeństwa, 872, 1006
pliku, 67
DLL, Dynamic Link Libraries, 86, 244
DMA, Direct Memory Access, 56, 354
DMI, Direct Media Interface, 58
dokument, 587
domeny
ochrony, protection domain, 616
urządzeń, 502
DOS, Disk Operating System, 41
dostawca zawartości, content provider, 818
dostęp bezpośredni do plików, 758
dostępność, availability, 606
dowiązanie, links, 300, 761
symboliczne, 290, 300
twarde, 290, 301
drzewo
katalogów, 289
procesów, 64
DSM, Distributed Shared Memory, 571
dysk, 72
magnetyczny, 52, 378
SATA, 30
SSD, 32, 52, 306, 388
VHD, 999
dyski
dynamiczne, 965
formatowanie, 379
obsługa błędów, 386
wirtualne, 965
dyspozytor płytowy, 746
dyspozytor stron, page allocator, 746
dziedziczenie priorytetów, 168

E

eBPF, extended Berkeley Packet Filter, 684
egzojądro, 95, 1041
ekran
dotykowy, 422
pojemnościowy, 423
rezystywny, 422
eksploit sprzętowy, 663
EPT, Extended Page Tables, 498
Ethernet, 581

F

FAAS, function as a service, 505
FCFS, first-come, first-served, 383
FIFO, first-in, first-out, 226
firmware, 899
fleszowanie, 898
folder, 66
formatowanie dysku, 379
fragmentacja
wewnętrzna, 241
zewnętrzna, 257
FreeBSD, 43
FTL, Flash Translation Layer, 306
funkcja jako usługa, FAAS, 505
futeks, 154

G

gadżet, 650, 1013
GDI, Graphics Device Interface, 418
główny rekord rozruchowy, MBR, 382, 735
gniazdo, socket, 753, 924
GPL, GNU Public License, 705
GPT, GUID Partition Table, 292, 382, 899
GPU, Graphics Processing Unit, 414, 541
graf
skierowany acykliczny, 300
zasobów, 454
graficzny interfejs użytkownika, GUI, 27, 413
grafy alokacji zasobów, 451
grupa kontrolna, control group, 507
grupy szeregowania, scheduling groups, 936

H

hasło, 634
jednorazowe, 638
HiberBoot, 987
hibernacja, 986
hierarchia
katalogów, 589
pamięci, 197
hipernadzorca, hypervisor, 482, 883, 989
Hyper-V, 883
na hoście, 489
typu 1, 93, 488, 522
typu 2, 94, 488
hiperwątek, 48, 542

host, 582
hotpatching, 1018
Hyper-V, 989
    odizolowane kontenery, 1000
    stos wirtualizacji, 991
    wejście-wyjście urządzeń, 992

I

IAAS, infrastructure as a service, 505
identyfikator
    GID, 65, 783
    GUID, 899
    SID poziomu integralności, 1009
    UID, 65, 783, 827
implementacja
    bezpieczeństwa, 786
    katalogów, 297
    menedżera obiektów, 902
    plików, 292
    procesów, 724, 927
    systemu plików, 768, 780, 976
    wątków, 724, 927
    wejścia-wyjścia, 756
    zarządzania pamięcią, 742, 948
infrastruktura jako usługa, IAAS, 505
instrukcja TSL, 145
instrukcje
    uprzywilejowane, 486
    wrażliwe, 486
integralność, integrity, 606
interfejs
    API, 493, 805, 871
    NT API, 873
    sterownik-jądro, 756
    urządzenia graficznego, GDI, 418
    Win32 API, 875
        funkcje, 947
    Windows API, 83
    wywołań systemowych, 1037
internet, 582
internet rzeczy, IoT, 61
interpreter poleceń, 64
intruzi, 611
inwersja priorytetów, 167
IoT, Internet of Things, 61
IP, Internet Protocol, 754
IPC, interprocess communication, 138, 801,
ISA, Industry Standard Architecture, 57
izolacja
    pamięci, 961
    urządzeń, device isolation, 501

J

jądro systemu Linux, 715
JBD, Journaling Block Device, 775
jednostka zarządzania pamięcią, MMU, 52, 211, 501
jednostki miar, 101
język
    AIDL, 807
    C, 96
        kompilacja, 99
        linker, 98
        pliki nagłówkowe, 97
        pliki obiektowe, 97
        preprocesor C, 98
        tworzenie pliku wykonywalnego, 99
        wskaźnik, 96
jitting, 939
JVM, Java Virtual Machine, 94

K

kanał
    boczny, side channel, 666
    ukryty, covert channel, 664
kanarki, stack canaries, 647
karta graficzna, 414
katalog, 66, 285
    główny, 67, 285
    roboczy, working directory, 67, 761
    spoolera, 377
katalogi
    implementacja, 297
    operacje, 289
    system hierarchiczny, 286
    system jednopoziomowy, 285
    ścieżka bezwzględna, 287
    w systemie UNIX, 289
KCET, Kernel-mode CET, 1015
KCFG, Kernel CFG, 1014
klawiatura, 404
KMDF, Kernel-Mode Driver Framework, 970
kompaktowanie pamięci, 205
kompresja, 331
    pamięci, 958

komputer
osobisty, 45
uruchamianie, 58
komunikacja
asynchroniczna, 1050
międzyprocesowa, IPC, 64, 138, 801, 923
synchroniczna, 1050
komunikat potwierdzający, 163
kontener, 95, 483, 996
hosta, 998
serwerowy, 998
odizolowany Hyper-V, 1000
kontrolery urządzeń, 55, 349
koordynacja, 593
koordynator wejścia-wyjścia, 716, 758
kopia przy zapisie, copy on write, 506, 726, 946
krotka, 593
kryptografia, 628
z kluczem publicznym, 630
z kluczem tajnym, 629
kwant, 178

L

LFS, Log-structured File System, 302
LibOS, Library Operating System, 95
licencja publiczna GPL, 705
licznik programu, 46
liczniki dozorujące, watchdog timers, 400
Linux, 43, 704
bezpieczeństwo, 783
cele Linuksa, 707
implementacja
bezpieczeństwa, 786
systemu plików, 768
wejścia-wyjścia, 756
zarządzania pamięcią, 742
interfejsy, 708
moduły, 759
obsługa sieci, 753
powłoka, 710
procesy, 717, 724
programy użytkowe, 713
przydzielanie pamięci, 746
publikacje, 1080
sieciowy system plików, 776
stronicowanie, 748
struktura jądra, 714
synchronizacja, 734
system plików, 760
system wejścia-wyjścia, 751
szeregowanie, 730
tryby ochrony pliku, 783
uruchamianie systemu, 735
warstwy systemu, 708
wątki, 727
wywołania systemowe, 720, 741
plików, 765
wejścia-wyjścia, 755
zarządzanie pamięcią, 738
lista
jednokierunkowa, 207
kontroli dostępu, ACL, 618, 878
domyślna, DACL, 1005
systemowa, SACL, 1007
uprawnień, 621
logiczne adresowanie bloków, 379
logowanie
fałszywy ekran, 674
LRU, Least-Recently Used, 228, 952
luka
Rowhammer, 676
Type Confusion, 657
luki
w oprogramowaniu, 643
podwójne pobieranie, 662
w zabezpieczeniach
eliminowanie, 1011

M

macierz ochrony, 624
macOS, 42
magazyn
stron, 250
systemowy, 960
magistrala, 56
DMI, 58
ISA, 57
NVMe, 388
PCIe, 57
USB, 58
mainframe, 34
mapa, 66
bitowa, 206, 420
strony poziomu 4, 222
maszyna wirtualna, 91, 503, 994
Javy, 94
migracje, 505
VMware, 509

MBR, Master Boot Record, 59, 291, 382, 735
menedżer
  konfiguracji, 896
  magazynu, memory manager, 958
  obiektów, 893, 902
  pamięci, 197, 895
  pamięci podręcznej, 895
  plug and play, 965
  procesów, 895
  zasilania, 986
metoda, 416, 592
mikroarchitektura, 46
mikrojądro, 88, 493
MINIX, 703
MMU, Memory Management Unit, 52, 211, 501
model
  bezpieczeństwa, 627
  klient-serwer, 90
  zbioru roboczego, 231
moduł TPM, 632, 683
moduły
  ładowalne, loadable modules, 759
  w systemie Linux, 759
monitor, 157
  kontroli bezpieczeństwa odwołań, 895
  maszyny wirtualnej, 91
  odwołań, reference monitor, 610, 686
  VMM, 482
most, bridge, 581
MS-DOS, MicroSoft Disk Operating System, 41
MULTICS, 257
muteks, 152, 155, 925
mysz, 407

N

naruszenie dostępu, access violation, 949
NFU, Not Frequently Used, 228
notacja polska, 417
NPU, Neural Processing Unit, 541
NUMA, Non-Uniform Memory Access, 532

O

obiekt, 592
  dyspozytora, 891
  powiadomienia, 892
  sterownika, 874
  synchronizacji, 892
  urządzenia, 909
obraz rdzenia, 64
obsługa
  błędów, 362
  przerwań, 366
  zegara, 396
obszar wymiany, swap area, 749
ochrona stosu egzekwowana sprzętowo, HSP, 1015
oczekiwanie aktywne, busy waiting, 55
odbiorca, receiver, 817
odczyt-kopiowanie-aktualizacja, 168
odczyt z wyprzedzeniem, read ahead, 781
odpytywanie, 364
odśmiecanie, garbage collection, 307
ograniczenia dostępu, 679
okno tekstowe, 408
OOM, out of memory killer, 239
opakowanie, wrapper, 130
operacje
  na katalogach, 289
  na plikach, 282
oprogramowanie
  do generowania wyjścia, 408
  do wprowadzania danych, 402
  jako usługa, SAAS, 505
  klawiatury, 404
  myszy, 407
  obsługi zegara, 398
  wejścia-wyjścia, 361
    niezależne od urządzeń, 371
    w przestrzeni użytkownika, 376
    warstwy, 366
ortogonalność, orthogonality, 1045

P

PAAS, platform as a service, 505
pakiet
  Pthreads, 127
    funkcje, 127, 155, 156
    muteksy, 155
    zmienne warunkowe, 156
  termcap, 408
pakiety MSIX, 869
pamięć, 49, *Patrz także* zarządzanie pamięcią
  dostęp bezpośredni, 353
  dostęp zdalny, 565
  izolacja, 961
  kompresja, 958
  rozproszona współdzielona, 571

pamięć
  EEPROM, 51
  fizyczna, 743, 954
  lokalna wątku, TLS, 916
  masowa
    dyski magnetyczne, 378
    dyski SSD, 388
    RAID, 392
    stabilna, 389
  nieulotna, 52
  o dużej pojemności, 71
  podręczna, cache, 50, 121, 757
    buforowa, 326
    L1, 51
    L2, 51
    obiektu, 747
    stron, 328
    systemu Windows, 963
    z natychmiastowym zapisem, 327
  RAM, 51, 197, 391
  ROM, 51
  trwała, 53
  w systemie Linux, 738
  wirtualna, 52, 73, 204, 209, 496
parawirtualizacja, 94, 487, 494
partycja, 82
  główna, root partition, 989
  rozruchowa, 899
  systemowa, 961
  systemowa EFI, 899
  wymiany, 251
partycje pamięci, 961
paskowanie, striping, 393
pasmo, lane, 57
PCIe, Peripheral Component Interconnect Express, 57
PCR, Platform Configuration Register, 900
PDA, Personal Digital Assistant, 61
PDP-11 UNIX, 699
PE, Portable Executable, 59
perspektywa, view, 963
PF, Physical Functions, 503
PFF, Page Fault Frequency, 238
piaskownica, sandbox, 481, 685, 825
  UID, 832
  Windows, 1001
pierścień ochrony, protection ring, 490
platforma jako usługa, PAAS, 505
plik, 66, 274
  AndroidManifest.xml, 809
  stron, pagefile, 944
  wirtualny stronicowania, 959
pliki
  alokacja
    algorytm, 295
    ciągła, 292
    i-węzły, 296
    lista jednokierunkowa, 294
  atrybuty, 281
  bezpieczne usuwanie, 332
  bezpośredni dostęp, 758
  blokowe surowe, 758
  dostęp, 280
  implementacja, 292
  kopiowanie, 283
  nazwy, 275
  odwzorowywane w pamięci, 246, 740
  operacje, 282
  rozrzedzone, sparse files, 980
  rozszerzenia, 276
  specjalne, 68, 752
    blokowe, 68, 752
    znakowe, 68, 752
  stronicowania, 945
  struktura, 277
  typy, 278
  współdzielone, 299
podpis cyfrowy, 631
  sterowników, 682
podwyższanie uprawnień, 1009
podział czasu, timesharing, 38
potok, pipeline, 46, 68, 712, 719
  trójfazowy, 47
poufność, confidentiality, 606
powłoka, shell, 27, 64, 69, 78, 710
  Bourne'a, Bourne shell, 710
prawa ogólne, generic rights, 623
priorytety podsystemu Win32, 932
problem
  czytelników i pisarzy, 151
  inwersji priorytetów, 147
  pięciu filozofów, 446
  producent-konsument, 147, 149, 157, 160, 161, 164
  relokacji, 200
  zamknięcia, confinement problem, 664
procedura obsługi przerwania, ISR, 116

proces, 63, 107
potomny, 64
publikacje, 1074
wymiany, swapper process, 748
procesor, 45
GPU, 541
graficzny, GPU, 49, 414
superskalarny, 47
system operacyjny, 543
wielordzeniowy heterogeniczny, 542
procesy
aktywne oczekiwanie, 141
bariery, 165
brokerów, broker processes, 870
dziedziczenie, 168
hierarchie, 113
implementacja, 115
inwersja priorytetów, 167
komponenty, 124
kończenie działania, 112
monitory, 157
muteksy, 152
odizolowane sprzętowo, 1001
przekazywanie komunikatów, 163
regiony krytyczne, 140
sekwencyjne, 108
semafory, 149
stany, 113
systemowe, 921
systemu Windows, 917
tworzenie, 110
unikanie blokad, 168
w systemie Android, 821
w systemie Linux, 717, 724
w systemie Windows, 917
WoW64, 938
wyścig, 139
wywołanie sleep, 146
wywołanie wakeup, 146
wzajemne wykluczanie, 141
zabijanie, 459
zorientowane na obliczenia, 172
zorientowane na wejście-wyjście, 172
program szeregujący, process scheduler, 114, 170
programowanie
sterowane zdarzeniami, 136
zorientowane na powrót, ROP, 1013
projektowanie systemu operacyjnego
buforowanie, 1061
cele, 1028
czas wiązania nazw, 1047
doświadczenie, 1067
efekt lokalności, 1063
implementacja, 1040
implementacja dół-góra, 1049
implementacja góra-dół, 1049
interfejs, 1031
interfejs wywołań systemowych, 1037
komunikacja asynchroniczna, 1050
komunikacja synchroniczna, 1050
mityczny osobomiesiąc, 1064
nazewnictwo, 1045
optymalizacja, 1063
paradygmaty, 1034
publikacje, 1081
struktura systemu, 1040
struktura zespołu, 1066
struktury dynamiczne, 1048
struktury statyczne, 1048
wydajność, 1056
zalecenia, 1032
proporcjonalność, 176
protokoły
obsługi, 758
sieciowe, 585
protokół
IP, 754
TCP, 586, 754
UDP, 754
prywatność, 833
przeglądy kodu, code reviews, 674
przekazywanie
komunikatów, 163
urządzeń, device pass through, 501
przekos
cylindrów, cylinder skew, 380
głowic, head skew, 381
przeładowanie, thrashing, 230
przełączanie
kontekstu, 52, 171
obwodów, 561
procesu, 171
światów, world switch, 492
przełączanie typu przechowaj i prześlij, 560

Kup książkę Poleć książkę

przełącznik
krzyżowy, 533
krzyżowy wielostopniowy, 535
przeplot
podwójny, 382
pojedynczy, 381
przepływ sterowania, 677
przepustowość, 175
przerwania, 55, 141, 356
nieprecyzyjne, 360
precyzyjne, 358
przestrzenie adresowe, 65
przestrzeń
adresowa, 63, 201
jądra, 943
jednowymiarowa, 254
danych, 242
dyskowa, storage spaces, 965
instrukcji, 242
nazw obiektów, 906, 908
przetaczanie, marshaling, 569
przetwarzanie w chmurze, 39
publikacje, 1074
pula
pamięci masowej, 965
wątków, thread pool, 919
pułap priorytetu, priority ceiling, 168
pułapka, 357
punkt
kontrolny, checkpoint , 459, 506
krzyżowy, crosspoint, 534
przyłączenia, reparse point, 983

Q

QoS, quality-of-service, 935

R

RAID, 392
RAM, Random-Access Memory, 51, 197
ramka strony, 211
randomizacja, 676
układu przestrzeni adresowej, 1013
raportowanie błędów, 375
RCU, Read-Copy-Update, 169
RDMA, Remote Direct Memory Access, 565
ReFS, Resilient File System, 275
region krytyczny, 140
rejestr
bazy, 202
limitu, 202
systemu Windows, 879
rejestry urządzeń, 55
relokacja, 200
dynamiczna, 202
statyczna, 200
remapowanie przerwań, interrupt remapping, 501
rezerwowanie przepustowości, bandwidth reservation, 967
ROM, Read-Only Memory, 51
router, 582
routing kanalikowy, wormhole routing, 561
rozmiar strony, 241
rozpoznawanie zamiaru, intent resolution, 820
RPC, Remote Procedure Call, 569

S

SAAS, software as a service, 505
SAM, Security Access Manager, 879
SATA, Serial ATA, 54
segment
danych, 79, 738
stosu, 79
tekstu, 79, 738
współdzielony, 740
segmentacja, 254, 256
implementacja, 257
ze stronicowaniem, 257, 261
sekwencje sterujące, escape sequences, 408
SELinux, 832
semafor, 149, 925
binarny, 150
Service Pack, 42
serwer reinkarnacji, 89
serwery
jednowątkowe, 138
sposoby konstrukcji, 138
sterowane zdarzeniami, 136
wielowątkowe, 138
sieć
ARPANET, 582
botnet, 612
Internet, 583
LAN, 580
WAN, 580
WWW, 587

silos aplikacji, 997
silos serwerowy, 997
skrypt powłoki, shell script, 712
skrytka pocztowa, mailslot, 923
skrzynka pocztowa, 165
słowo stanu programu, PSW, 46
SMAP, Supervisor Mode Access Protection, 680
SMEP, Supervisor Mode Execution Protection, 680
SMP, symmetric multiprocessor, 545
SMT, simultaneous multithreading, 542
spooling, 38, 377
SR-IOV, Single Root I/O Virtualization, 502
SSD, solid-state drives, 52
standard POSIX, 720, 741, 755
standard UNIX, 702
standardowe
- wejście, standard input, 711
- wyjście, standard output, 711

standardowy błąd, standard error, 711
sterownik
- dysku, 30
- interfejsu hipernadzorcy, 991
- urządzenia, 54, 752, 897
  - interfejs, 371

sterowniki
- filtrów, 973
- urządzeń, 367, 970

sterta, heap, 739
stos
- urządzeń, 898
- wirtualizacji, 989, 991

strażnik
- dowolnego kodu, ACG, 1015
- integralności kodu, CIG, 1015
- przepływu powrotu, RFG, 1014
- przepływu sterowania, CFG, 1013
- rozszerzony przepływu sterowania, 1014

strona, 210
- adresów wirtualnych, 943
- duża, large page, 941
- ogromna, huge page, 941
- WWW, 587
- zerowa, 739

stronicowanie, 210, 247, 251, 256
- LRU, 952
- na żądanie, 230
- projektowanie systemu, 236
- przyspieszanie, 216
- w systemie Linux, 748
- wstępne, prepaging, 231

strony współdzielone, 243
struktura
- danych CONTEXT, 920
- systemu x86, 57

struktury danych partycji pamięci, 962
superużytkownik, 784
SVM, Secure Virtual Machine, 486
swapping, 65
sygnał, 720
- alarmowy, 64

Symbian OS, 44
symbol
- potoku, pipe symbol, 712
- wieloznaczny, wild card, 710
- zachęty, prompt, 70, 710

symulatory maszyn, 93
synchroniczny transfer, 362
synchronizacja, 138, 924
- rywalizacji, 468
- w Linuksie, 734

syntetyczna obsługa przerwań, 992
system operacyjny
- Android, 786
- DOS, 41
- Blackberry OS, 44
- Linux, 704, 707
- MS-DOS, 41, 860
- UNIX, 698
- Windows 10, 866
- Windows 11, 859, 867
- Windows 7, 865
- Windows 8, 865
- Windows NT 4.0, 863
- Windows Vista, 864

system plików, 66, 274
- /proc, 776
- exFAT, 275
- ext2, 769
- ext4, 775
- FAT-32, 275
- NFS, Network File System, 310, 776
  - architektura systemu, 777
  - implementacja systemu, 780
  - protokoły systemu, 778
  - wersja NFSv4, 782

NTFS, NT File System, 974
kompresja plików, 984
księgowanie, 985
struktura, 976
szyfrowanie, 985
ReFS, 275, 974
V7, 337
VFS, 309, 715, 760, 768
system rozproszony, 578
protokoły sieciowe, 585
sprzęt sieciowy, 580
usługi sieciowe, 583
warstwa middleware
bazująca na dokumentach, 587
bazująca na koordynacji, 593
bazująca na obiektach, 592
bazująca na systemie plików, 588
system wielokomputerowy, 558, 579
interfejs sieciowy, 561
oprogramowanie komunikacyjne
niskopoziomowy, 563
poziom użytkownika, 565
równoważenie obciążenia, 575
sprzęt, 559
szeregowanie, 575
topologie połączeń, 559
współdzielona pamięć, 571
zdalne wywołania procedur, 568
system wieloprocesorowy, 529, 579
badania, 596
NUMA, 536
partycjonowanie pamięci, 543
przełączanie, 550
publikacje, 1078
rozproszony, 578
symetryczny, 545
synchronizacja, 547
szeregowanie, 551
szeregowanie zespołów, 555
typu lider-naśladowca, 544
UMA, 532–535
wielokomputer, 558
współdzielenie przestrzeni, 554
zapętlanie, 550
ze współdzieloną pamięcią, 532
systemy plików, 66, 274
badania, 339
globalne, 588
implementacja, 290, 768
kopie zapasowe, 318
księgujące, 304, 775
Linuksa, 760, 768
MS-DOS, 334
na nośnikach typu flash, 305
o strukturze dziennika, 302
optymalizacja, 312
publikacje, 1075
spójność, 323
układ dla BIOS-u, 291
układ dla UEFI, 292
wsadowe, 34
wydajność, 325
zarządzanie, 312
systemy operacyjne
badania, 99
bezpieczeństwo, 605
czasu rzeczywistego, 62, 184
egzojądra, 95, 1041
generacje, 33–45
jako menedżery zasobów, 31
jako rozszerzona maszyna, 30
kart elektronicznych, 62
komputerów
mainframe, 60
osobistych, 61
podręcznych, 61
maszyny wirtualne, 91
mikrojądra, 88, 1042
model klient-serwer, 90
model klient-serwer z mikrojądrem, 1042
monolityczne, 85
pisane w języku C, 96
procesora, 543
projektowanie, 1027
rozproszone, 43
serwerów, 60
sieciowe, 43
smartfonów, 61
stronicowanie, 247
unijądra, 95
VM/370 z CMS, 91
wbudowane, 61
wielowarstwowe, 86, 1040
systemy zaufane, trusted systems, 610
szachownicowanie, checkerboarding, 257
szeregowanie, 170, 172, 551
bazujące na priorytetach, 180
cykliczne, 178

Kup ksi k Pole ksi k

szeregowanie
dwupoziomowe, 553
gwarantowane, 182
inteligentne, 553
loteryjne, 183
mechanizm, 185
pod kątem bezpieczeństwa, 557
równoległe, co-scheduling, 556
sprawiedliwe, 184
strategia, 185
systemów wielokomputerowych, 575
w systemie
czasu rzeczywistego, 184
interaktywnym, 178
Linux, 730
wsadowym, 176
wątków, 186
według powinowactwa, affinity scheduling, 553
zespołów, gang scheduling, 555, 556
szpiegowanie, snooping, 539
szyfrowanie dysków, 332

Ś

ścieżka, 67
bezwzględna, absolute path, 287, 761
komunikacji IPC, 808
względna, relative path, 761

T

tabela
i-węzłów, 81
procesów, 63, 115
skrótów, hash table, 223
wskaźników katalogu stron, 222
tablica
adresów Hotpatch, 1018
deskryptorów stron, 743
EPT, 498
GPT, 292, 382
MFT, 978
partycji, 292
stron, 212, 214, 498, 952
odwrócona, 222
wielopoziomowa, 220
uchwytów, 904, 905
wektora przerwań, 56
TCP, Transmission Control Protocol, 586, 754
technologia
SVM, 486
VT, 486
terminal, 402
THP, Transparent Huge Pages, 241
TLB, Translation Lookaside Buffer, 218, 948
tłumaczenie binarne, binary translation, 93, 487, 490
token, 878, 918
dostępu, access token, 1005
TPM, Trusted Platform Module, 632, 900
TPU, Tensor Processing Unit, 541
trackpad, 407
transakcja, 802
tryb
gotowości, 987
IUM, 1003
jądra, 28
LPC, 871
wirtualnego, 489
przelotu, fly-by mode, 355
surowy, raw mode, 404
ugotowany, cooked mode, 404
użytkownika, 28
wiązki, burst mode, 355
tylne drzwi, back door, 674
typy obiektów, 909, 911

U

UAC, User Account Control, 1009
uchwyt, handle, 872, 904
jądra, 904
UDP, User Datagram Protocol, 754
UEFI, Unified Extensible Firmware Interface, 58, 292
układ
DMA, 56
scalony, 36
scalony wielkiej skali integracji, 40
ultrawielordzeniowy, manycore chip, 540
wielordzeniowy, multicore chip, 48, 539, 540
wielowątkowy, 48
UMA, Uniform Memory Access, 532
UMDF, User-Mode Driver Framework, 970
UNICS, 698
unijądro, 95

UNIX
przenośny, 700
publikacje, 1080
uprawnienia, capabilities, rights, 616, 621
w Androidzie, 827, 833
URL, Uniform Resource Locator, 587
uruchamianie komputera, 58
urządzenia
domeny, 502
emulowane, 992
kontrolery, 349
parawirtualizowane, 993
sieciowe, 759
sterowniki, 367
wejścia-wyjścia, 53
blokowe, 348
odwzorowane w pamięci, 350
znakowe, 348
współdzielone, 502
z akceleracją sprzętową, 993
urządzenie JBD, 775
USB, Universal Serial Bus, 58, 368
usługa
VMMS, 992
w systemie Android, 815
usługi sieciowe, 583
uwierzytelnianie, 633
biometryczne, 642, 867
metodą wyzwanie-odpowiedź, 639
użycie obiektu fizycznego, 640
wieloskładnikowe, 867
uwięzienie, livelock, 471

V

VF, virtual functions, 503
VFS, Virtual File System, 309, 715, 760, 768
VHD, Virtual Hard Disk, 999
VM/370, 91
VMM, Virtual Machine Monitor, 482
VMMS, Virtual Machine Management Service, 992
VMware Workstation, 510, 513
ESX Server, 522
ewolucja systemu, 521
konfiguracja wirtualnego sprzętu, 516
system operacyjny hosta, 518
wirtualizacja na platformie x86, 511
VT, Virtualization Technology, 486

W

warstwa abstrakcji sprzętowej, 882
wątek POSIX, 126
wątki, 118
dyspozytora, 121
eksmisji magazynu, 960
implementacja
hybrydowa, 132
w jądrze, 131
w przestrzeni użytkownika, 128
komponenty, 124
konflikty, 133
model klasyczny, 123
obsługa wielowątkowości, 133
pakiet Pthreads, 127
pracownika, 121
prywatne zmienne globalne, 134
publikacje, 1074
robocze jądra, 751
systemowe, 1043
szeregowanie, 186
w systemie Linux, 727
w systemie Windows, 920
zwielokrotnianie, 132
wczesne wiązanie, early binding, 1047
WDF, Windows Driver Foundation, 970
WDK, Windows Driver Kit, 970
WDM, Windows Driver Model, 970
wejście-wyjście, 53, 69, 347
badania, 433
cele oprogramowania, 361
cienkie klienty, 423
funkcje oprogramowania, 371
funkcje warstw, 377
implementacja systemu, 969
implementacja w Linuksie, 756
interfejsy użytkowników, 402
jednostki MMU, 52, 211, 501
pakiety żądań, 971
procedury obsługi przerwań, 366
programowane, 363
publikacje, 1076
sterowane przerwaniami, 364
sterowniki urządzeń, 367
systemu Linux, 751, 757
systemu Windows, 964
urządzenia, 348
warstwy systemu, 377

wejście-wyjście
  wirtualizacja, 500, 502
  wykorzystanie DMA, 365
  wywołania API, 967
  wywołania systemowe, 755
wektor przerwań, 116, 357
weryfikator aplikacji, application verifier, 906
wiązanie późne, late binding, 1047
wielka blokada jądra, 734
wielobieżność, reentrancy, 1054
wielokomputer, *Patrz* system wielokomputerowy
wieloprogramowość, 37, 108, 117
wielowątkowość, 48, 123, 132, 542
wiersz pamięci podręcznej, 533
Win32 API, 83, 875
Win64 API, 83
WinAPI, 83
winda szeregująca Linuksa, 758
Windows
  10, 866
  11, 859, 867
  7, 865
  8, 865
  adresy wirtualne, 943
  App SDK, 870
  asynchroniczne wywołania procedur, 890
  awaria systemu, 894
  bezpieczeństwo, 1004
  biblioteki DLL, 913
  Defender, 1020
  hipernadzorca, 989
  hipernadzorca Hyper-V, 883, 989
  implementacja
    bezpieczeństwa, 1008
    procesów, 927
    systemu plików, 976
    systemu wejścia-wyjścia, 969
    wątków, 927
    zarządzania pamięcią, 948
  jądra i sterowniki, 938, 940
  klasy priorytetów, 932
  kompresja pamięci, 958
  kontenery, 996
  menedżer obiektów, 902
  NT, 42
  NT 4.0, 863
  obiekty dyspozytora, 891
  obsługa błędów braku stron, 949
  operacje wejścia-wyjścia, 964
  opóźnione wywołania procedur, 888
  oprogramowanie Windows Defender, 1020
  organizacja trybu jądra, 882
  pamięć podręczna, 963
  partycje pamięci, 961
  podsystemy, 870, 913
  procesy, 916, 917
  programowanie systemu, 868
  publikacje, 1080
  pule wątków, 919
  rejestr, 879
  sterowniki urządzeń, 897
  stosy urządzeń, 972
  struktura systemu, 881
  synchronizacja, 924
  system plików, 974
  szeregowanie, 930
  uruchamianie systemu, 898
  usługa Windows Update, 1017
  usługi trybu użytkownika, 913
  Vista, 864
  warstwa abstrakcji sprzętowej, 884
  warstwa jądra, 887
  warstwa wykonawcza, 893
  warstwy programowania, 868
  wątki, 916, 920
  wersje systemu, 860
  wirtualizacja, 988
  włókna, 918
  wywołania API, 921, 1007
  wywołania systemowe, 946
  zabezpieczenia systemu, 1016
  zadania, 918
  zarządzanie
    energią, 986
    pamięcią fizyczną, 954
    pamięcią, 941, 946
wirtualizacja, 481, 494
  architektury x86, 513
  badania, 523
  koszt, 492
  na poziomie systemu operacyjnego, 506
  pamięci, 496
  poziomu procesu, 488
  przestrzeni nazw, 997
  publikacje, 1077
  SR-IOV, 502
  systemów bez obsługi wirtualizacji, 490

Kup książkę Poleć książkę

techniki, 489
w systemie Windows, 988
wejścia-wyjścia, 500, 502
wymagania, 485
wirtualna
platforma sprzętowa, 516
przestrzeń adresowa, 211
wirtualny i-węzeł, 780
WoW64, 937
procesy, 938
WSA, Windows Subsystem for Android, 1001
wskaźnik, 96
stosu, 46
WSL, Windows Subsystem for Linux, 1001
współdzielenie
bibliotek, 244
plików, 591
stron, 243
bazujące na zawartości, 503
transparentne, 503
wyjątek, 357
wymiana
pamięci, swapping, 204
stron dynamiczna, 251
wyścig, 139
wywłaszczanie, preemptive, 173, 444, 458
wywołania
blokujące, 566
nieblokujące, 566
systemowe, 47, 73
różne, 82
Win32 API, 84
zarządzanie katalogami, 80
zarządzanie plikami, 80
zarządzanie procesami, 76
wzajemne wykluczanie, 141
rozwiązanie Petersona, 144
ścisła naprzemienność, 142
wyłączanie przerwań, 141
zmienne blokujące, 142

X

X Window, 409
klienty i serwery, 410
X Window System, X11, 43

Z

zabezpieczenia, 69
oparte na wirtualizacji, 867
sprzętowe, 72
zabieranie cykli, cycle stealing, 355
zabójca braku pamięci, OOM, 798
zadanie, 34
zagłodzenie, 448, 471
zakleszczenie, deadlock, 158, 443, 449
algorytm strusia, 453
badania, 472
blokowanie dwufazowe, 467
komunikacyjne, 468
modelowanie, 450
przeciwdziałanie
warunek braku wywłaszczania, 466
warunek cyklicznego oczekiwania, 466
warunek wstrzymania i oczekiwania, 465
warunek wzajemnego wykluczania, 465
przeciwdziałanie, 465
publikacje, 1077
unikanie, 460
algorytm bankiera, 462, 463
stany przydziału zasobów, 461
trajektorie zasobów, 460
usuwanie, 453
poprzez wywłaszczanie, 458
poprzez zabijanie procesów, 459
przez cofnięcie operacji, 459
warunki powstawania, 450
wykrywanie, 453–456
zasobów, 450
zasobów w sieci, 470
zamiar, intent, 819
jawny, explicit intent, 820
niejawny, implicit intent, 820
zapis z opóźnieniem, 964
zapobieganie uruchamianiu danych, DEP, 1013
zarządzanie
buforem TLB, 219
energią, 425, 986
dysk twardy, 427
inteligentne baterie, 431
interfejs sterownika, 432
komunikacja bezprzewodowa, 430
pamięć, 430
problemy sprzętowe, 426
problemy systemu operacyjnego, 427

Kup ksi k Pole ksi k

zarządzanie
  energią
    procesor, 428
    wyświetlacz, 427
    zarządzanie temperaturą, 431
  katalogami, 77, 80
  miejscem na dysku, 312
  obciążeniem, 239
  pamięcią, 52, 116, 197, 738, 741, 742, 946
    algorytmy zastępowania stron, 223
    badania, 261
    fizyczną, 743, 954
    listy jednokierunkowe, 207
    mapy bitowe, 206
    oddzielenie strategii od mechanizmu, 252
    pamięć wirtualna, 209
    podręczną, 326
    przestrzenie adresowe, 201
    publikacje, 1075
    segmentacja, 254
    wolną, 206
  plikami, 80, 116
  procesami, 76, 116, 720, 921
  systemem plików, 77, 312
  wątkami, 921
  włóknami, 921
  zadaniami, 921
  zasobami, 32
zasada najmniejszego zaskoczenia, 707
zasoby
  trajektorie, 460
  zakleszczenie, 450
  zdobywanie, 445
zasób
  bez możliwości wywłaszczania, 444
  z wywłaszczaniem, 444
zbiór roboczy, 230
zdalne wywoływanie procedur, RPC, 569, 802
zdalny bezpośredni dostęp do pamięci, 565
zdarzenia, 136
  powiadomień, 925
  synchronizacji, 925
zdarzenie, 925
zegary, clocks, 396
  budowa, 396
  programowe, 397, 400
  sterownik, 398
złośliwe oprogramowanie
  rootkit, 899
zmienne
  blokujące, 142
  warunkowe, 156, 926
znak ucieczki, escape character, 406
zrzut
  fizyczny, 319
  logiczny, 320

Kup książkę Poleć książkę

PROGRAM PARTNERSKI

GRUPY 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 Helion

Systemy operacyjne

Większość aplikacji i usług jest zależna od interakcji z systemem operacyjnym, dlatego profesjonaliści IT potrzebują głębokiej, a przede wszystkim aktualnej wiedzy w tej dziedzinie. To właśnie zrozumienie systemów operacyjnych pozwala inżynierowi IT na skuteczne diagnozowanie problemów, optymalizowanie wydajności i tworzenie solidnych rozwiązań, które oprą się próbie czasu i podniosą poziom bezpieczeństwa.

To piąte, gruntownie zaktualizowane wydanie podręcznika, który doceni każdy student informatyki i inżynier oprogramowania. Książka obejmuje szeroki zakres zagadnień, od podstawowych pojęć po zaawansowaną problematykę związaną z najnowszymi trendami w systemach operacyjnych. Wyczerpująco omawia procesy, wątki, zarządzanie pamięcią, systemy plików, operacje wejścia-wyjścia, zakleszczenia, interfejsy użytkownika, multimedia czy kompromisy wydajnościowe. Szczegółowo, jako studia przypadków, zostały tu opisane systemy: Windows 11, Unix, Linux i Android. Jasny i przystępny styl, a także liczne przykłady i ćwiczenia ułatwiają zrozumienie nawet bardzo skomplikowanych zagadnień.

W książce między innymi:

- **podstawowe pojęcia i struktura systemów operacyjnych**
- **sprzęt a funkcjonowanie systemu operacyjnego**
- **przegląd systemów operacyjnych, w tym internetu rzeczy i systemów wbudowanych**
- **systemy: Unix, Linux, Android — procesy, zarządzanie pamięcią, bezpieczeństwo**
- **Windows 11 — struktura, procesy i wątki, wirtualizacja, zabezpieczenia**
- **projektowanie systemów operacyjnych**

Mistrz oprogramowania zaczyna od systemu operacyjnego!

Dr Andrew S. Tanenbaum jest emerytowanym profesorem informatyki. Wcześniej był dziekanem Advanced School for Computing and Imaging, a także profesorem Królewskiej Holenderskiej Akademii Sztuk i Nauk. Ma duży dorobek, również programistyczny — napisał system MINIX, który był inspiracją dla Linuksa.

Dr Herbert Bos jest profesorem w zakładzie informatyki Wolnego Uniwersytetu w Amsterdamie. Głównym obszarem jego badań są zabezpieczenia systemów. Odkrywał luki bezpieczeństwa w większości głównych systemów operacyjnych, najpopularniejszych przeglądarkach i procesorach.

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-0289-3

Cena: 179,00 zł