ANDREW S.
TANENBAUM
HERBERT
BOS

SYSTEMY OPERACYJNE

WYDANIE IV

Helion

Tytuł oryginału: Modern Operating Systems (4th Edition)

Tłumaczenie: Radosław Meryk
na podstawie „Systemy operacyjne. Wydanie III” w tłumaczeniu Radosława Meryka i Mikołaja Szczepaniaka

ISBN: 978-83-283-1422-1

Authorized translation from the English language edition, entitled: MODERN OPERATING SYSTEMS; Fourth Edition, ISBN 013359162X; by Andrew S. Tanenbaum; and by Herbert Bos; published by Pearson Education, Inc, publishing as Prentice Hall.

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/sysop4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

SPIS TREŚCI

Przedmowa 23

O autorach 27

1 Wprowadzenie 29

1.1. CZYM JEST SYSTEM OPERACYJNY? 31
- 1.1.1. System operacyjny jako rozszerzona maszyna 32
- 1.1.2. System operacyjny jako menedżer zasobów 33

1.2. HISTORIA SYSTEMÓW OPERACYJNYCH 34
- 1.2.1. Pierwsza generacja (1945 – 1955) — lampy elektronowe 35
- 1.2.2. Druga generacja (1955 – 1965) — tranzystory i systemy wsadowe 35
- 1.2.3. Trzecia generacja (1965 – 1980) — układy scalone i wieloprogramowość 37
- 1.2.4. Czwarta generacja (1980 – czasy współczesne) — komputery osobiste 42
- 1.2.5. Piąta generacja (1990 – czasy współczesne) — komputery mobilne 46

1.3. SPRZĘT KOMPUTEROWY — PRZEGLĄD 47
- 1.3.1. Procesory 47
- 1.3.2. Pamięć 51
- 1.3.3. Dyski 54

1.3.4. Urządzenia wejścia-wyjścia 55
1.3.5. Magistrale 58
1.3.6. Uruchamianie komputera 61

1.4. PRZEGLĄD SYSTEMÓW OPERACYJNYCH 61
1.4.1. Systemy operacyjne komputerów mainframe 62
1.4.2. Systemy operacyjne serwerów 62
1.4.3. Wieloprocesorowe systemy operacyjne 62
1.4.4. Systemy operacyjne komputerów osobistych 63
1.4.5. Systemy operacyjne komputerów podręcznych 63
1.4.6. Wbudowane systemy operacyjne 63
1.4.7. Systemy operacyjne węzłów sensorowych 64
1.4.8. Systemy operacyjne czasu rzeczywistego 64
1.4.9. Systemy operacyjne kart elektronicznych 65

1.5. POJĘCIA DOTYCZĄCE SYSTEMÓW OPERACYJNYCH 65
1.5.1. Procesy 65
1.5.2. Przestrzenie adresowe 67
1.5.3. Pliki 68
1.5.4. Wejście-wyjście 71
1.5.5. Zabezpieczenia 71
1.5.6. Powłoka 71
1.5.7. Ontogeneza jest rekapitulacją filogenezy 73

1.6. WYWOŁANIA SYSTEMOWE 76
1.6.1. Wywołania systemowe do zarządzania procesami 79
1.6.2. Wywołania systemowe do zarządzania plikami 82
1.6.3. Wywołania systemowe do zarządzania katalogami 83
1.6.4. Różne wywołania systemowe 85
1.6.5. Interfejs Win32 API systemu Windows 85

1.7. STRUKTURA SYSTEMÓW OPERACYJNYCH 88
1.7.1. Systemy monolityczne 88
1.7.2. Systemy warstwowe 89
1.7.3. Mikrojądra 90
1.7.4. Model klient-serwer 93
1.7.5. Maszyny wirtualne 93
1.7.6. Egzojądra 97

1.8. ŚWIAT WEDŁUG JĘZYKA C 98
1.8.1. Język C 98
1.8.2. Pliki nagłówkowe 99

1.8.3. Duże projekty programistyczne 100
1.8.4. Model fazy działania 100

1.9. BADANIA DOTYCZĄCE SYSTEMÓW OPERACYJNYCH 101

1.10. PLAN POZOSTAŁEJ CZĘŚCI KSIĄŻKI 103

1.11. JEDNOSTKI MIAR 104

1.12. PODSUMOWANIE 104

2 Procesy i wątki 109

2.1. PROCESY 109
2.1.1. Model procesów 110
2.1.2. Tworzenie procesów 112
2.1.3. Kończenie działania procesów 114
2.1.4. Hierarchie procesów 115
2.1.5. Stany procesów 115
2.1.6. Implementacja procesów 117
2.1.7. Modelowanie wieloprogramowości 119

2.2. WĄTKI 120
2.2.1. Wykorzystanie wątków 121
2.2.2. Klasyczny model wątków 125
2.2.3. Wątki POSIX 129
2.2.4. Implementacja wątków w przestrzeni użytkownika 131
2.2.5. Implementacja wątków w jądrze 134
2.2.6. Implementacje hybrydowe 135
2.2.7. Mechanizm aktywacji zarządcy 135
2.2.8. Wątki pop-up 137
2.2.9. Przystosowywanie kodu jednowątkowego do obsługi wielu wątków 138

2.3. KOMUNIKACJA MIĘDZY PROCESAMI 141
2.3.1. Wyścig 141
2.3.2. Regiony krytyczne 143
2.3.3. Wzajemne wykluczanie z wykorzystaniem aktywnego oczekiwania 144
2.3.4. Wywołania sleep i wakeup 149
2.3.5. Semafory 151
2.3.6. Muteksy 154
2.3.7. Monitory 159

2.3.8. Przekazywanie komunikatów 164
2.3.9. Bariery 167
2.3.10. Unikanie blokad: odczyt-kopiowanie-aktualizacja 168

2.4. SZEREGOWANIE 169
2.4.1. Wprowadzenie do szeregowania 170
2.4.2. Szeregowanie w systemach wsadowych 176
2.4.3. Szeregowanie w systemach interaktywnych 178
2.4.4. Szeregowanie w systemach czasu rzeczywistego 184
2.4.5. Oddzielenie strategii od mechanizmu 185
2.4.6. Szeregowanie wątków 185

2.5. KLASYCZNE PROBLEMY KOMUNIKACJI MIĘDZY PROCESAMI 187
2.5.1. Problem pięciu filozofów 187
2.5.2. Problem czytelników i pisarzy 190

2.6. PRACE BADAWCZE NAD PROCESAMI I WĄTKAMI 191

2.7. PODSUMOWANIE 192

3 Zarządzanie pamięcią 201

3.1. BRAK ABSTRAKCJI PAMIĘCI 202

3.2. ABSTRAKCJA PAMIĘCI: PRZESTRZENIE ADRESOWE 205
3.2.1. Pojęcie przestrzeni adresowej 205
3.2.2. Wymiana pamięci 207
3.2.3. Zarządzanie wolną pamięcią 210

3.3. PAMIĘĆ WIRTUALNA 213
3.3.1. Stronicowanie 214
3.3.2. Tabele stron 217
3.3.3. Przyspieszenie stronicowania 219
3.3.4. Tabele stron dla pamięci o dużej objętości 223

3.4. ALGORYTMY ZASTĘPOWANIA STRON 226
3.4.1. Optymalny algorytm zastępowania stron 227
3.4.2. Algorytm NRU 228
3.4.3. Algorytm FIFO 229
3.4.4. Algorytm drugiej szansy 229
3.4.5. Algorytm zegarowy 230

3.4.6. Algorytm LRU 231
3.4.7. Programowa symulacja algorytmu LRU 231
3.4.8. Algorytm bazujący na zbiorze roboczym 233
3.4.9. Algorytm WSClock 236
3.4.10. Podsumowanie algorytmów zastępowania stron 238

3.5. PROBLEMY PROJEKTOWE SYSTEMÓW STRONICOWANIA 239
3.5.1. Lokalne i globalne strategie alokacji pamięci 239
3.5.2. Zarządzanie obciążeniem 241
3.5.3. Rozmiar strony 242
3.5.4. Osobne przestrzenie instrukcji i danych 243
3.5.5. Strony współdzielone 244
3.5.6. Biblioteki współdzielone 246
3.5.7. Pliki odwzorowane w pamięci 248
3.5.8. Strategia czyszczenia 248
3.5.9. Interfejs pamięci wirtualnej 249

3.6. PROBLEMY IMPLEMENTACJI 249
3.6.1. Zadania systemu operacyjnego w zakresie stronicowania 250
3.6.2. Obsługa błędów braku strony 250
3.6.3. Archiwizowanie instrukcji 251
3.6.4. Blokowanie stron w pamięci 253
3.6.5. Magazyn stron 253
3.6.6. Oddzielenie strategii od mechanizmu 255

3.7. SEGMENTACJA 256
3.7.1. Implementacja klasycznej segmentacji 259
3.7.2. Segmentacja ze stronicowaniem: MULTICS 260
3.7.3. Segmentacja ze stronicowaniem: Intel x86 263

3.8. BADANIA DOTYCZĄCE ZARZĄDZANIA PAMIĘCIĄ 267

3.9. PODSUMOWANIE 268

4 Systemy plików 279

4.1. PLIKI 281
4.1.1. Nazwy plików 281
4.1.2. Struktura pliku 283
4.1.3. Typy plików 284
4.1.4. Dostęp do plików 286

4.1.5. Atrybuty plików 286
4.1.6. Operacje na plikach 288
4.1.7. Przykładowy program wykorzystujący wywołania obsługi systemu plików 289

4.2. KATALOGI 291
4.2.1. Jednopoziomowe systemy katalogów 291
4.2.2. Hierarchiczne systemy katalogów 292
4.2.3. Nazwy ścieżek 292
4.2.4. Operacje na katalogach 294

4.3. IMPLEMENTACJA SYSTEMU PLIKÓW 296
4.3.1. Układ systemu plików 296
4.3.2. Implementacja plików 297
4.3.3. Implementacja katalogów 302
4.3.4. Pliki współdzielone 304
4.3.5. Systemy plików o strukturze dziennika 306
4.3.6. Księgujące systemy plików 308
4.3.7. Wirtualne systemy plików 310

4.4. ZARZĄDZANIE SYSTEMEM PLIKÓW I OPTYMALIZACJA 313
4.4.1. Zarządzanie miejscem na dysku 313
4.4.2. Kopie zapasowe systemu plików 319
4.4.3. Spójność systemu plików 324
4.4.4. Wydajność systemu plików 327
4.4.5. Defragmentacja dysków 332

4.5. PRZYKŁADOWY SYSTEM PLIKÓW 332
4.5.1. System plików MS-DOS 333
4.5.2. System plików V7 systemu UNIX 336
4.5.3. Systemy plików na płytach CD-ROM 338

4.6. BADANIA DOTYCZĄCE SYSTEMÓW PLIKÓW 343

4.7. PODSUMOWANIE 344

5 Wejście-wyjście 349

5.1. WARUNKI, JAKIE POWINIEN SPEŁNIAĆ SPRZĘT WEJŚCIA-WYJŚCIA 349
5.1.1. Urządzenia wejścia-wyjścia 350
5.1.2. Kontrolery urządzeń 351

5.1.3. Urządzenia wejścia-wyjścia odwzorowane w pamięci 352
5.1.4. Bezpośredni dostęp do pamięci (DMA) 355
5.1.5. O przerwaniach raz jeszcze 358

5.2. WARUNKI, JAKIE POWINNO SPEŁNIAĆ OPROGRAMOWANIE WEJŚCIA-WYJŚCIA 362
5.2.1. Cele oprogramowania wejścia-wyjścia 362
5.2.2. Programowane wejście-wyjście 364
5.2.3. Wejście-wyjście sterowane przerwaniami 365
5.2.4. Wejście-wyjście z wykorzystaniem DMA 366

5.3. WARSTWY OPROGRAMOWANIA WEJŚCIA-WYJŚCIA 367
5.3.1. Procedury obsługi przerwań 367
5.3.2. Sterowniki urządzeń 368
5.3.3. Oprogramowanie wejścia-wyjścia niezależne od urządzeń 372
5.3.4. Oprogramowanie wejścia-wyjścia w przestrzeni użytkownika 377

5.4. DYSKI 379
5.4.1. Sprzęt 379
5.4.2. Formatowanie dysków 384
5.4.3. Algorytmy szeregowania ramienia dysku 388
5.4.4. Obsługa błędów 391
5.4.5. Stabilna pamięć masowa 393

5.5. ZEGARY 396
5.5.1. Sprzęt obsługi zegara 397
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 407

5.7. CIENKIE KLIENTY 423

5.8. ZARZĄDZANIE ENERGIĄ 424
5.8.1. Problemy sprzętowe 425
5.8.2. Problemy po stronie systemu operacyjnego 426
5.8.3. Problemy do rozwiązania w programach aplikacyjnych 432

5.9. BADANIA DOTYCZĄCE WEJŚCIA-WYJŚCIA 433

5.10. PODSUMOWANIE 435

6 Zakleszczenia 443

6.1. ZASOBY 444
6.1.1. Zasoby z możliwością wywłaszczania i bez niej 444
6.1.2. Zdobywanie zasobu 445

6.2. WPROWADZENIE W TEMATYKĘ ZAKLESZCZEŃ 447
6.2.1. Warunki powstawania zakleszczeń zasobów 447
6.2.2. Modelowanie zakleszczeń 448

6.3. ALGORYTM STRUSIA 450

6.4. WYKRYWANIE ZAKLESZCZEŃ I ICH USUWANIE 451
6.4.1. Wykrywanie zakleszczeń z jednym zasobem każdego typu 451
6.4.2. Wykrywanie zakleszczeń dla przypadku wielu zasobów każdego typu 453
6.4.3. Usuwanie zakleszczeń 455

6.5. UNIKANIE ZAKLESZCZEŃ 457
6.5.1. Trajektorie zasobów 457
6.5.2. Stany bezpieczne i niebezpieczne 458
6.5.3. Algorytm bankiera dla pojedynczego zasobu 459
6.5.4. Algorytm bankiera dla wielu zasobów 460

6.6. PRZECIWDZIAŁANIE ZAKLESZCZENIOM 462
6.6.1. Atak na warunek wzajemnego wykluczania 462
6.6.2. Atak na warunek wstrzymania i oczekiwania 463
6.6.3. Atak na warunek braku wywłaszczania 463
6.6.4. Atak na warunek cyklicznego oczekiwania 463

6.7. INNE PROBLEMY 464
6.7.1. Blokowanie dwufazowe 465
6.7.2. Zakleszczenia komunikacyjne 465
6.7.3. Uwięzienia 468
6.7.4. Zagłodzenia 469

6.8. BADANIA NA TEMAT ZAKLESZCZEŃ 469

6.9. PODSUMOWANIE 470

7 Wirtualizacja i przetwarzanie w chmurze 477

7.1. HISTORIA 479

7.2. WYMAGANIA DOTYCZĄCE WIRTUALIZACJI 480

7.3. HIPERNADZORCY TYPU 1 I TYPU 2 483

7.4. TECHNIKI SKUTECZNEJ WIRTUALIZACJI 484
- 7.4.1. Wirtualizacja systemów bez obsługi wirtualizacji 485
- 7.4.2. Koszt wirtualizacji 487

7.5. CZY HIPERNADZORCY SĄ PRAWIDŁOWYMI MIKROJĄDRAMI? 488

7.6. WIRTUALIZACJA PAMIĘCI 491

7.7. WIRTUALIZACJA WEJŚCIA-WYJŚCIA 495

7.8. URZĄDZENIA WIRTUALNE 498

7.9. MASZYNY WIRTUALNE NA PROCESORACH WIELORDZENIOWYCH 498

7.10. PROBLEMY LICENCYJNE 499

7.11. CHMURY OBLICZENIOWE 500
- 7.11.1. Chmury jako usługa 500
- 7.11.2. Migracje maszyn wirtualnych 501
- 7.11.3. Punkty kontrolne 502

7.12. STUDIUM PRZYPADKU: VMWARE 502
- 7.12.1. Wczesna historia firmy VMware 503
- 7.12.2. VMware Workstation 504
- 7.12.3. Wyzwania podczas opracowywania warstwy wirtualizacji na platformie x86 505
- 7.12.4. VMware Workstation: przegląd informacji o rozwiązaniu 506
- 7.12.5. Ewolucja systemu VMware Workstation 515
- 7.12.6. ESX Server: hipernadzorca typu 1 firmy VMware 515

7.13. BADANIA NAD WIRTUALIZACJĄ I CHMURĄ 517

8 Systemy wieloprocesorowe 521

8.1. SYSTEMY WIELOPROCESOROWE 523
8.1.1. Sprzęt wieloprocesorowy 524
8.1.2. Typy wieloprocesorowych systemów operacyjnych 534
8.1.3. Synchronizacja w systemach wieloprocesorowych 538
8.1.4. Szeregowanie w systemach wieloprocesorowych 542

8.2. WIELOKOMPUTERY 548
8.2.1. Sprzęt wielokomputerów 548
8.2.2. Niskopoziomowe oprogramowanie komunikacyjne 552
8.2.3. Oprogramowanie komunikacyjne poziomu użytkownika 555
8.2.4. Zdalne wywołania procedur 558
8.2.5. Rozproszona współdzielona pamięć 560
8.2.6. Szeregowanie systemów wielokomputerowych 565
8.2.7. Równoważenie obciążenia 565

8.3. SYSTEMY ROZPROSZONE 568
8.3.1. Sprzęt sieciowy 570
8.3.2. Usługi i protokoły sieciowe 573
8.3.3. Warstwa middleware bazująca na dokumentach 576
8.3.4. Warstwa middleware bazująca na systemie plików 578
8.3.5. Warstwa middleware bazująca na obiektach 582
8.3.6. Warstwa middleware bazująca na koordynacji 584

8.4. BADANIA DOTYCZĄCE SYSTEMÓW WIELOPROCESOROWYCH 586

8.5. PODSUMOWANIE 587

9 Bezpieczeństwo 593

9.1. ŚRODOWISKO BEZPIECZEŃSTWA 595
9.1.1. Zagrożenia 596
9.1.2. Intruzi 598

9.2. BEZPIECZEŃSTWO SYSTEMÓW OPERACYJNYCH 599
9.2.1. Czy możemy budować bezpieczne systemy? 600
9.2.2. Zaufana baza obliczeniowa 601

9.3. KONTROLOWANIE DOSTĘPU DO ZASOBÓW 602
9.3.1. Domeny ochrony 602
9.3.2. Listy kontroli dostępu 605
9.3.3. Uprawnienia 607

9.4. MODELE FORMALNE BEZPIECZNYCH SYSTEMÓW 610
9.4.1. Bezpieczeństwo wielopoziomowe 612
9.4.2. Ukryte kanały 614

9.5. PODSTAWY KRYPTOGRAFII 619
9.5.1. Kryptografia z kluczem tajnym 620
9.5.2. Kryptografia z kluczem publicznym 621
9.5.3. Funkcje jednokierunkowe 622
9.5.4. Podpisy cyfrowe 622
9.5.5. Moduły TPM 624

9.6. UWIERZYTELNIANIE 626
9.6.1. Uwierzytelnianie z wykorzystaniem obiektu fizycznego 633
9.6.2. Uwierzytelnianie z wykorzystaniem technik biometrycznych 635

9.7. WYKORZYSTYWANIE BŁĘDÓW W KODZIE 638
9.7.1. Ataki z wykorzystaniem przepełnienia bufora 640
9.7.2. Ataki z wykorzystaniem łańcuchów formatujących 648
9.7.3. „Wiszące wskaźniki” 651
9.7.4. Ataki bazujące na odwołaniach do pustego wskaźnika 652
9.7.5. Ataki z wykorzystaniem przepełnień liczb całkowitych 653
9.7.6. Ataki polegające na wstrzykiwaniu kodu 654
9.7.7. Ataki TOCTOU 655

9.8. ATAKI Z WEWNĄTRZ 656
9.8.1. Bomby logiczne 656
9.8.2. Tylne drzwi 656
9.8.3. Podszywanie się pod ekran logowania 657

9.9. ZŁOŚLIWE OPROGRAMOWANIE 658
9.9.1. Konie trojańskie 661
9.9.2. Wirusy 663
9.9.3. Robaki 673
9.9.4. Oprogramowanie szpiegujące 676
9.9.5. Rootkity 680

9.10. ŚRODKI OBRONY 684
9.10.1. Firewalle 685
9.10.2. Techniki antywirusowe i antyantywirusowe 687
9.10.3. Podpisywanie kodu 693
9.10.4. Wtrącanie do więzienia 695
9.10.5. Wykrywanie włamań z użyciem modeli 695
9.10.6. Izolowanie kodu mobilnego 697
9.10.7. Bezpieczeństwo Javy 701

9.11. BADANIA DOTYCZĄCE BEZPIECZEŃSTWA 704

9.12. PODSUMOWANIE 705

10 Pierwsze studium przypadku: UNIX, Linux i Android 715

10.1. HISTORIA SYSTEMÓW UNIX I LINUX 716
10.1.1. UNICS 716
10.1.2. PDP-11 UNIX 717
10.1.3. Przenośny UNIX 718
10.1.4. Berkeley UNIX 719
10.1.5. Standard UNIX 720
10.1.6. MINIX 721
10.1.7. Linux 722

10.2. PRZEGLĄD SYSTEMU LINUX 725
10.2.1. Cele Linuksa 725
10.2.2. Interfejsy systemu Linux 726
10.2.3. Powłoka 728
10.2.4. Programy użytkowe systemu Linux 731
10.2.5. Struktura jądra 733

10.3. PROCESY W SYSTEMIE LINUX 735
10.3.1. Podstawowe pojęcia 735
10.3.2. Wywołania systemowe Linuksa związane z zarządzaniem procesami 738
10.3.3. Implementacja procesów i wątków w systemie Linux 742
10.3.4. Szeregowanie w systemie Linux 748
10.3.5. Uruchamianie systemu Linux 753

10.4. ZARZĄDZANIE PAMIĘCIĄ W SYSTEMIE LINUX 755
10.4.1. Podstawowe pojęcia 756
10.4.2. Wywołania systemowe Linuksa odpowiedzialne za zarządzanie pamięcią 759
10.4.3. Implementacja zarządzania pamięcią w systemie Linux 760
10.4.4. Stronicowanie w systemie Linux 766

10.5. OPERACJE WEJŚCIA-WYJŚCIA W SYSTEMIE LINUX 769
10.5.1. Podstawowe pojęcia 769
10.5.2. Obsługa sieci 771
10.5.3. Wywołania systemowe wejścia-wyjścia w systemie Linux 772
10.5.4. Implementacja wejścia-wyjścia w systemie Linux 773
10.5.5. Moduły w systemie Linux 776

10.6. SYSTEM PLIKÓW LINUKSA 777
10.6.1. Podstawowe pojęcia 777
10.6.2. Wywołania systemu plików w Linuksie 782
10.6.3. Implementacja systemu plików Linuksa 785
10.6.4. NFS — sieciowy system plików 794

10.7. BEZPIECZEŃSTWO W SYSTEMIE LINUX 800
10.7.1. Podstawowe pojęcia 800
10.7.2. Wywołania systemowe Linuksa związane z bezpieczeństwem 802
10.7.3. Implementacja bezpieczeństwa w systemie Linux 803

10.8. ANDROID 804
10.8.1. Android a Google 804
10.8.2. Historia Androida 805
10.8.3. Cele projektowe 808
10.8.4. Architektura Androida 810
10.8.5. Rozszerzenia Linuksa 811
10.8.6. Dalvik 814
10.8.7. Binder IPC 816
10.8.8. Aplikacje Androida 824
10.8.9. Zamiary 834
10.8.10. Piaskownice aplikacji 835
10.8.11. Bezpieczeństwo 836
10.8.12. Model procesów 841

10.9. PODSUMOWANIE 846

11 Drugie studium przypadku: Windows 8 855

11.1. HISTORIA SYSTEMU WINDOWS DO WYDANIA WINDOWS 8.1 855
11.1.1. Lata osiemdziesiąte: MS-DOS 856
11.1.2. Lata dziewięćdziesiąte: Windows na bazie MS-DOS-a 857
11.1.3. Lata dwutysięczne: Windows na bazie NT 857
11.1.4. Windows Vista 860
11.1.5. Druga dekada lat dwutysięcznych: Modern Windows 861

11.2. PROGRAMOWANIE SYSTEMU WINDOWS 862
11.2.1. Rdzenny interfejs programowania aplikacji (API) systemu NT 865
11.2.2. Interfejs programowania aplikacji Win32 869
11.2.3. Rejestr systemu Windows 872

11.3. STRUKTURA SYSTEMU 875
11.3.1. Struktura systemu operacyjnego 875
11.3.2. Uruchamianie systemu Windows 890
11.3.3. Implementacja menedżera obiektów 891
11.3.4. Podsystemy, biblioteki DLL i usługi trybu użytkownika 901

11.4. PROCESY I WĄTKI SYSTEMU WINDOWS 904
11.4.1. Podstawowe pojęcia 904
11.4.2. Wywołania API związane z zarządzaniem zadaniami, procesami, wątkami i włóknami 911
11.4.3. Implementacja procesów i wątków 916

11.5. ZARZĄDZANIE PAMIĘCIĄ 924
11.5.1. Podstawowe pojęcia 924
11.5.2. Wywołania systemowe związane z zarządzaniem pamięcią 928
11.5.3. Implementacja zarządzania pamięcią 929

11.6. PAMIĘĆ PODRĘCZNA SYSTEMU WINDOWS 939

11.7. OPERACJE WEJŚCIA-WYJŚCIA W SYSTEMIE WINDOWS 940
11.7.1. Podstawowe pojęcia 940
11.7.2. Wywołania API związane z operacjami wejścia-wyjścia 942
11.7.3. Implementacja systemu wejścia-wyjścia 944

11.8. SYSTEM PLIKÓW NT SYSTEMU WINDOWS 949
11.8.1. Podstawowe pojęcia 949
11.8.2. Implementacja systemu plików NTFS 950

11.9. ZARZĄDZANIE ENERGIĄ W SYSTEMIE WINDOWS 960

11.10. BEZPIECZEŃSTWO W SYSTEMIE WINDOWS 8 963
11.10.1. Podstawowe pojęcia 964
11.10.2. Wywołania API związane z bezpieczeństwem 965
11.10.3. Implementacja bezpieczeństwa 967
11.10.4. Czynniki ograniczające zagrożenia bezpieczeństwa 969

11.11. PODSUMOWANIE 972

12 Projekt systemu operacyjnego 979

12.1. ISTOTA PROBLEMÓW ZWIĄZANYCH Z PROJEKTOWANIEM SYSTEMÓW 980
12.1.1. Cele 980
12.1.2. Dlaczego projektowanie systemów operacyjnych jest takie trudne? 981

12.2. PROJEKT INTERFEJSU 983
12.2.1. Zalecenia projektowe 983
12.2.2. Paradygmaty 986
12.2.3. Interfejs wywołań systemowych 989

12.3. IMPLEMENTACJA 992
12.3.1. Struktura systemu 992
12.3.2. Mechanizm kontra strategia 996
12.3.3. Ortogonalność 997
12.3.4. Nazewnictwo 998
12.3.5. Czas wiązania nazw 999
12.3.6. Struktury statyczne kontra struktury dynamiczne 1000
12.3.7. Implementacja góra-dół kontra implementacja dół-góra 1001
12.3.8. Komunikacja synchroniczna kontra asynchroniczna 1002
12.3.9. Przydatne techniki 1004

12.4. WYDAJNOŚĆ 1009
12.4.1. Dlaczego systemy operacyjne są powolne? 1009
12.4.2. Co należy optymalizować? 1010
12.4.3. Dylemat przestrzeń-czas 1011
12.4.4. Buforowanie 1014
12.4.5. Wskazówki 1015
12.4.6. Wykorzystywanie efektu lokalności 1015
12.4.7. Optymalizacja z myślą o typowych przypadkach 1016

12.5. ZARZĄDZANIE PROJEKTEM 1017
12.5.1. Mityczny osobomiesiąc 1017
12.5.2. Struktura zespołu 1018
12.5.3. Znaczenie doświadczenia 1020
12.5.4. Nie istnieje jedno cudowne rozwiązanie 1021

12.6. TRENDY W ŚWIECIE PROJEKTÓW SYSTEMÓW OPERACYJNYCH 1021
12.6.1. Wirtualizacja i przetwarzanie w chmurze 1022
12.6.2. Układy wielordzeniowe 1022
12.6.3. Systemy operacyjne z wielkimi przestrzeniami adresowymi 1023
12.6.4. Bezproblemowy dostęp do danych 1024
12.6.5. Komputery zasilane bateriami 1024
12.6.6. Systemy wbudowane 1025

12.7. PODSUMOWANIE 1026

13 Lista publikacji i bibliografia 1031

13.1. SUGEROWANE PUBLIKACJE DODATKOWE 1031
13.1.1. Publikacje wprowadzające i ogólne 1032
13.1.2. Procesy i wątki 1032
13.1.3. Zarządzanie pamięcią 1033
13.1.4. Systemy plików 1033
13.1.5. Wejście-wyjście 1034
13.1.6. Zakleszczenia 1035
13.1.7. Wirtualizacja i przetwarzanie w chmurze 1035
13.1.8. Systemy wieloprocesorowe 1036
13.1.9. Bezpieczeństwo 1037
13.1.10. Pierwsze studium przypadku: UNIX, Linux i Android 1039
13.1.11. Drugie studium przypadku: Windows 8 1039
13.1.12. Zasady projektowe 1040

13.2. BIBLIOGRAFIA W PORZĄDKU ALFABETYCZNYM 1041

A Multimedialne systemy operacyjne 1069

A.1. WPROWADZENIE W TEMATYKĘ MULTIMEDIÓW 1070

A.2. PLIKI MULTIMEDIALNE 1074

A.2.1. Kodowanie wideo 1075
A.2.2. Kodowanie audio 1078

A.3. KOMPRESJA WIDEO 1079
A.3.1. Standard JPEG 1080
A.3.2. Standard MPEG 1082

A.4. KOMPRESJA AUDIO 1085

A.5. SZEREGOWANIE PROCESÓW MULTIMEDIALNYCH 1088
A.5.1. Szeregowanie procesów homogenicznych 1088
A.5.2. Szeregowanie w czasie rzeczywistym — przypadek ogólny 1088
A.5.3. Szeregowanie monotoniczne w częstotliwości 1090
A.5.4. Algorytm szeregowania EDF 1091

A.6. PARADYGMATY DOTYCZĄCE MULTIMEDIALNYCH SYSTEMÓW PLIKÓW 1093
A.6.1. Funkcje sterujące VCR 1094
A.6.2. Wideo niemal na życzenie 1096

A.7. ROZMIESZCZENIE PLIKÓW 1097
A.7.1. Umieszczanie pliku na pojedynczym dysku 1097
A.7.2. Dwie alternatywne strategie organizacji plików 1098
A.7.3. Rozmieszczenie wielu plików na pojedynczym dysku 1102
A.7.4. Rozmieszczanie plików na wielu dyskach 1104

A.8. BUFOROWANIE 1106
A.8.1. Buforowanie bloków 1106
A.8.2. Buforowanie plików 1108

A.9. SZEREGOWANIE OPERACJI DYSKOWYCH W SYSTEMACH MULTIMEDIALNYCH 1108
A.9.1. Statyczne szeregowanie operacji dyskowych 1108
A.9.2. Dynamiczne szeregowanie operacji dyskowych 1110

A.10. BADANIA NA TEMAT MULTIMEDIÓW 1112

A.11. PODSUMOWANIE 1112

Skorowidz 1119

2

PROCESY I WĄTKI

Zanim rozpocznie się szczegółowe studium tego, w jaki sposób systemy operacyjne są zaprojektowane i skonstruowane, warto przypomnieć, że 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. Bez abstrakcji procesów istnienie współczesnej techniki komputerowej byłoby niemożliwe. W niniejszym 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).

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.

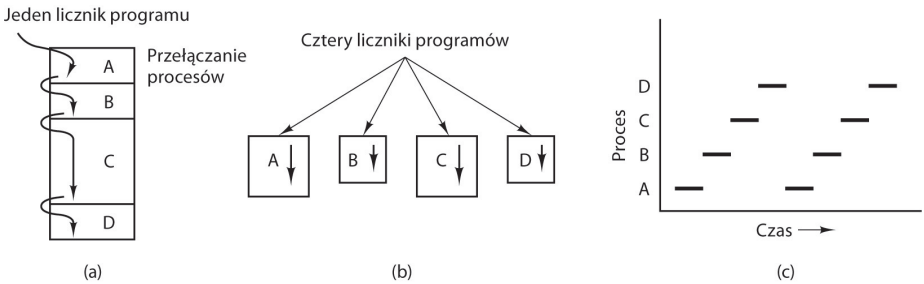
W każdym systemie wieloprogramowym procesor szybko przełącza się pomiędzy procesami, poświęcając każdemu z nich po kolei dziesiątki albo setki milisekund. Chociaż, ściśle rzecz biorąc, w dowolnym momencie procesor realizuje tylko jeden proces, w ciągu sekundy może obsłużyć ich wiele, co daje iluzję współbieżności. 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 niniejszego 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.



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, w którym 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, 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 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 Microsoft 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. 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 technicznego 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.

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 tworzy proces-dziecko za pomocą wywołania `fork`, 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.

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). Alternatywnie proces-dziecko może współużytkować pamięć procesu-rodzica, ale w tym przypadku 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. Zniszczenie 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 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, 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

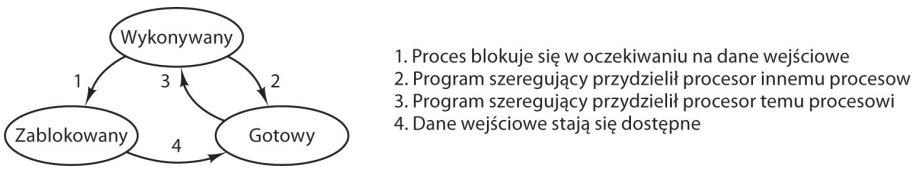
Chociaż każdy proces jest niezależnym podmiotem, posiadającym własny licznik programu i wewnętrzny stan, procesy często muszą się komunikować z innymi procesami. Jeden proces może generować wyjście, które inny proces wykorzysta jako wejście. W poleceniu powłoki:

```
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` 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).



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.

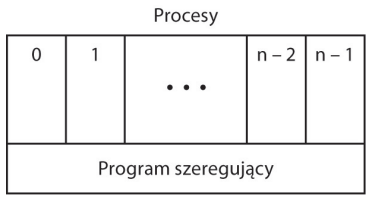
Tak jak pokazano na rysunku, pomiędzy tymi trzema stanami możliwe są cztery przejścia. Przejście nr 1 występuje wtedy, kiedy system operacyjny wykryje, że proces nie może kontynuować działania. W niektórych systemach proces może wykonać wywołanie systemowe, np. `pause`, w celu przejścia do stanu zablokowania. 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 zachodzi 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. Przyjrzymy się mu bliżej w dalszej części tego rozdziału. 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 niniejszego rozdziału.

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.

W tabeli 2.1 pokazano kilka kluczowych pól w typowym systemie. 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.

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, kiedy 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). 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. 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. Od tego momentu obsługą przerwania zajmuje się oprogramowanie — w szczególności procedura obsługi przerwania.

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.

Tabela 2.2. Szkielet działań wykonywanych przez najniższy poziom systemu operacyjnego w momencie wystąpienia przerwania

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.

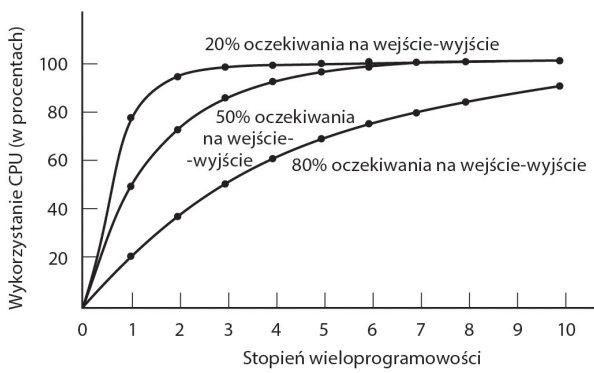
2.1.7. Modelowanie wieloprogramowości

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. Jednak przy jednym procesorze nie ma możliwości jednoczesnego działania trzech procesów. 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.

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 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.

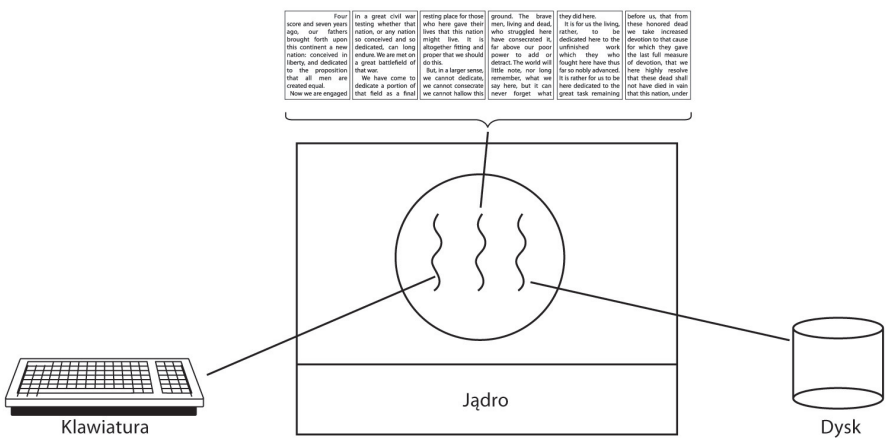
Najłatwiej przekonać się o przydatności wątków, analizując konkretne przykłady. W roli pierwszego przykładu rozważmy edytor tekstu. Edytory tekstu zazwyczaj wyświetlają na ekranie tworzony dokument sformatowany dokładnie w takiej postaci, w jakiej będzie on wyglądał na drukowanej stronie. Zwłaszcza wszystkie 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. Alternatywnie można 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. 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. Alternatywnie 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 bazującego na przerwaniach. W przypadku zastosowania trzech wątków model programowania jest znacznie prostszy. Pierwszy wątek

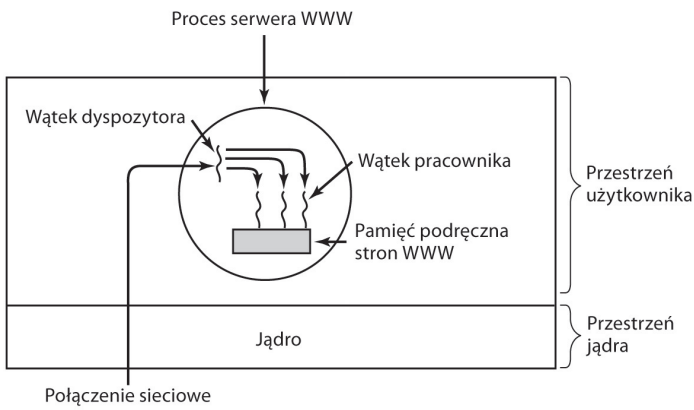
zajmuje się jedynie interakcjami z użytkownikiem. Drugi wątek przeformatowuje dokument, kiedy otrzyma takie zlecenie. Trzeci wątek okresowo zapisuje zawartość pamięci RAM na dysk.

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.

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. Kiedy użytkownik zmodyfikuje jeden element, może zajść potrzeba obliczenia wielu innych elementów. 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 ośrodków WWW niektóre strony WWW są częściej odwiedzane niż inne, np. główna strona serwisu Sony jest odwiedzana znacznie częściej od strony umieszczonej głęboko w drzewie katalogów i zawierającej specyfikację techniczną jakiegoś modelu kamery wideo. 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.

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.

Do tej pory omówiliśmy dwa możliwe projekty: wielowątkowy serwer WWW i jednowątkowy serwer WWW. Załóżmy, że wątki nie są dostępne, 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.

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, dlatego jest trudne do zaprogramowania. Dostępne modele zestawiono w tabeli 2.3.

Tabela 2.3. 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

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.

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

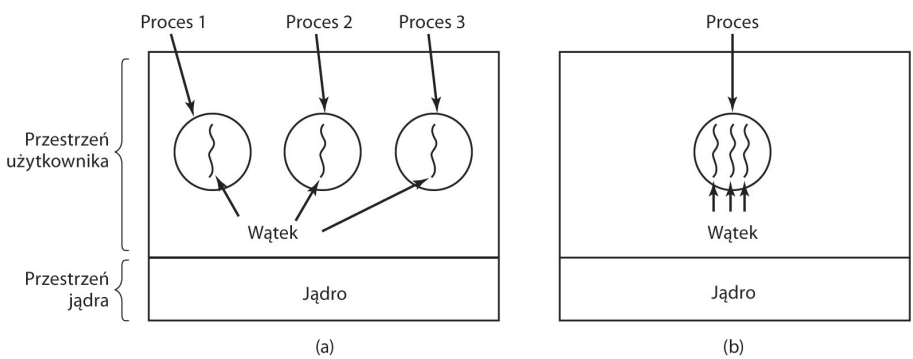
można skorzystać z wątków. Najpierw przyjrzymy się klasycznemu modelowi wątków. Następnie omówimy model wątków Linuksa, w którym linia pomiędzy wątkami i procesami jest rozmyta.

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, 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 wątek 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 2.4. 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 2.4. W pierwszej kolumnie wyszczególniono cechy wspólne dla wszystkich wątków w procesie. W drugiej kolumnie zamieszczone niektóre elementy prywatne dla każdego wątku

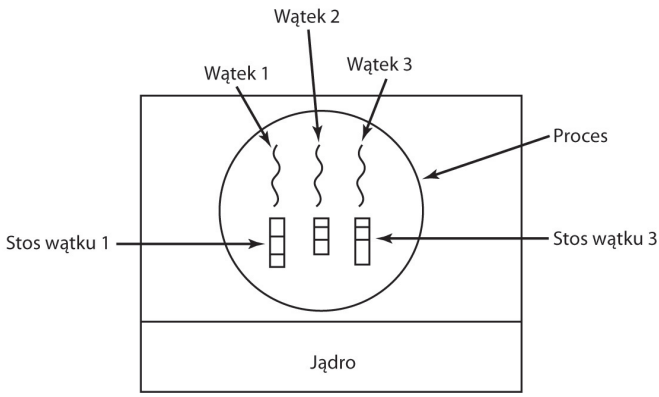
Komponenty procesu	Komponenty wątku
Przestrzeń adresowa	Licznik programu
Zmienne globalne	Rejestry
Otwarte pliki	Stos
Procesy-dzieci	Stan
Zaległe alarmy	
Sygnały i procedury obsługi sygnałów	
Informacje dotyczące statystyk	

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 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 swój 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 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 twardych połączeń sieciowych.

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 2.5.

Tabela 2.5. 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

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.

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 przypadku procesów, ponieważ zakłada się, że procesy ze sobą rywalizują i każdy z nich chce uzyskać maksymalnie dużo czasu procesora. 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 *print-hello-world(void *tid)
{
    /* 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.

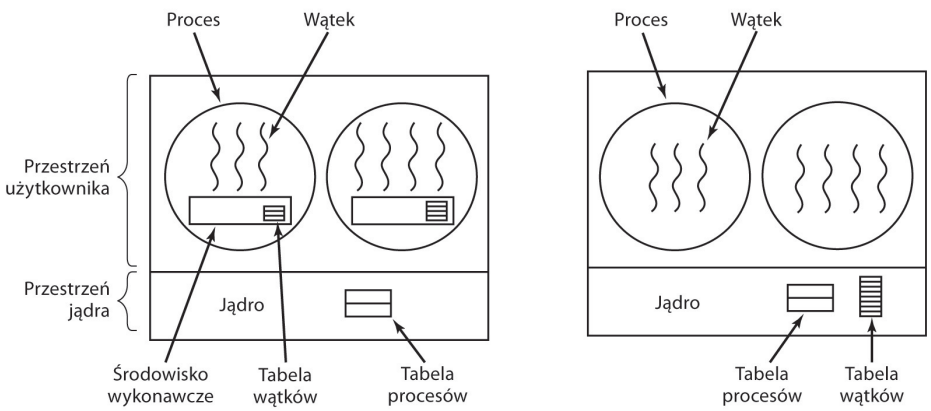
Pakiet `Pthreads` w żadnym razie nie ogranicza się do funkcji opisanych powyżej. Jest ich znacznie więcej. Niektóre z kolejnych wywołań opiszemy później, po omówieniu zagadnienia synchronizacji procesów i wątków.

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. Do tej pory zapoznaliśmy się z czterema procedurami z tej grupy: `pthread_create`, `pthread_exit`, `pthread_join` oraz `pthread_yield`. Zwykle jednak jest ich więcej.



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

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.

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.

Jest jednak jedna zasadnicza różnica w porównaniu z procesami. 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. Co więcej, może on następnie wywołać zarządcę wątków w celu wybrania innego wątku 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 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*).

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 lub skoczy do instrukcji, której nie ma w pamięci, występuje 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 druzgoczącym argumentem przeciwko wątkom zarządzanym na poziomie przestrzeni użytkownika jest fakt, że programiści, ogólnie rzecz biorąc, potrzebują wątków w aplikacjach, w których wątki blokują się często — np. w wielowątkowym serwerze WWW. 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). Tabela wątków również nie występuje w każdym procesie. 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 informacji tradycyjnie utrzymywanych przez jądro 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.

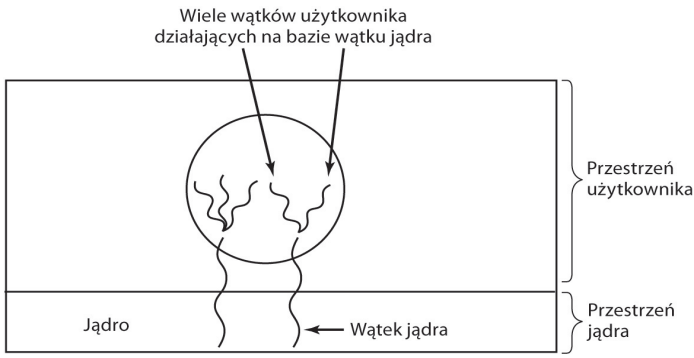
Wątki jądra nie wymagają żadnych nowych nieblokujących wywołań systemowych. Co więcej, jeśli jeden z wątków w procesie spowoduje warunek braku strony, jądro może łatwo sprawdzić, czy proces zawiera inne wątki możliwe do uruchomienia. Jeśli tak, uruchamia jeden z nich w oczekiwaniu na przesłanie z dysku 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 `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 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. Co się jednak stanie, jeśli dwa wątki (lub większa liczba wątków) zarejestrują zainteresowanie tym samym sygnałem? To tylko dwa problemy, jakie stwarzają wątki. Jest ich jednak więcej.

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.



Rysunek 2.10. Zwielokrotnianie wątków użytkownika na bazie wątków jądra

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.

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.

2.2.7. Mechanizm aktywacji zarządcy

Chociaż zarządzanie wątkami na poziomie jądra jest lepsze od zarządzania nimi na poziomie użytkownika pod pewnymi istotnymi względami, jest ono również bezdyskusyjnie wolniejsze. W związku z tym poszukiwano sposobów poprawy tej sytuacji bez konieczności rezygnacji z ich dobrych właściwości. Poniżej opiszemy jedno z zaproponowanych rozwiązań, opracowane przez zespół kierowany przez Andersona [Anderson et al., 1992] i nazywane *aktywacją zarządcy* (ang. *scheduler activations*). Podobne prace zostały opisane w [Edlera et al., 1988] oraz [Scott et al., 1990].

Celem działania mechanizmu aktywacji zarządcy jest naśladowanie funkcji wątków jądra, ale z zapewnieniem lepszej wydajności i elastyczności — cech, które zwykle charakteryzują pakiety zarządzania wątkami zaimplementowane w przestrzeni użytkownika. W szczególności wątki użytkownika nie powinny wykonywać specjalnych, nieblokujących wywołań systemowych lub sprawdzać wcześniej, czy wykonanie określonych wywołań systemowych jest bezpieczne. Niemniej jednak, kiedy wątek zablokuje się na wywołaniu systemowym lub sytuacji braku strony, powinien mieć możliwość uruchomienia innego wątku w ramach tego samego procesu, jeśli jakiś jest gotowy do działania.

Wydajność osiągnięto dzięki uniknięciu niepotrzebnych przejść pomiędzy przestrzenią użytkownika a przestrzenią jądra. Jeśli np. wątek zablokuje się w oczekiwaniu na to, aż inny wątek wykona jakieś działania, nie ma powodu informowania o tym jądra. Dzięki temu unika się kosztów związanych z przejściami pomiędzy przestrzeniami jądra i użytkownika. Środowisko wykonawcze przestrzeni użytkownika może samodzielnie zablokować wątek synchronizujący i zainicjować nowy.

Kiedy jest wykorzystywany mechanizm aktywacji zarządcy, jądro przypisuje określoną liczbę procesorów wirtualnych do każdego procesu i umożliwia środowisku wykonawczemu (przestrzeni użytkownika) na przydzielanie wątków do procesorów. Mechanizm ten może być również wykorzystany w systemie wieloprocesorowym, w którym zamiast procesorów wirtualnych są procesory fizyczne. Liczba procesorów wirtualnych przydzielonych do procesu zazwyczaj początkowo wynosi jeden, ale proces może poprosić o więcej, a także zwrócić procesory, których już nie potrzebuje. Jądro może również zwrócić wirtualne procesory przydzielone wcześniej, w celu przypisania ich procesom bardziej potrzebującym.

Podstawowa zasada działania tego mechanizmu polega na tym, że jeśli jądro dowie się o blokadzie wątku (np. z powodu uruchomienia blokującego wywołania systemowego lub braku strony), to powiadamia o tym środowisko wykonawcze procesu. W tym celu przekazuje na stos w postaci parametrów numer zablokowanego wątku oraz opis zdarzenia, które wystąpiło. Powiadomienie może być zrealizowane dzięki temu, że jądro uaktywnia środowisko wykonawcze znajdujące się pod znanym adresem początkowym. Jest to mechanizm w przybliżeniu analogiczny do sygnałów w Uniksie. Mechanizm ten określa się terminem *wezwanie* (ang. *upcall*).

Po uaktywnieniu w taki sposób środowisko wykonawcze może zmienić harmonogram działania swoich wątków. Zazwyczaj odbywa się to poprzez oznaczenie bieżącego wątku jako zablokowany oraz pobranie innego wątku z listy wątków będących w gotowości, ustawienie jego rejestrów i wznowienie działania. Później, kiedy jądro dowie się, że poprzedni wątek może ponownie działać (np. potok, z którego czytał dane, zawiera dane lub brakująca strona została pobrana z dysku), wykonuje kolejne wezwanie do środowiska wykonawczego w celu poinformowania go o tym zdarzeniu. Środowisko wykonawcze może wówczas we własnej gestii natychmiast zrestartować zablokowany wątek lub umieścić go na liście wątków do późniejszego uruchomienia.

Jeżeli wystąpi przerwanie sprzętowe, gdy działa wątek użytkownika, procesor przełącza się do trybu jądra. Jeśli przerwanie jest spowodowane przez zdarzenie, którym przerwany proces nie jest zainteresowany — np. zakończenie operacji wejścia-wyjścia innego procesu — to kiedy procedura obsługi przerwania zakończy działanie, umieszcza przerwany wątek w tym samym stanie, w jakim znajdował się on przed wystąpieniem przerwania. Jeśli jednak proces jest zainteresowany przerwaniem — np. nadejście strony wymaganej przez jeden z wątków procesu — przerwany proces nie jest wznawiany. Zamiast tego jest on zawieszany, a na tym samym wirtualnym procesorze zaczyna działać środowisko wykonawcze — stan przerwanego wątku jest w tym momencie umieszczony na stosie. W tym momencie środowisko wykonawcze podej-

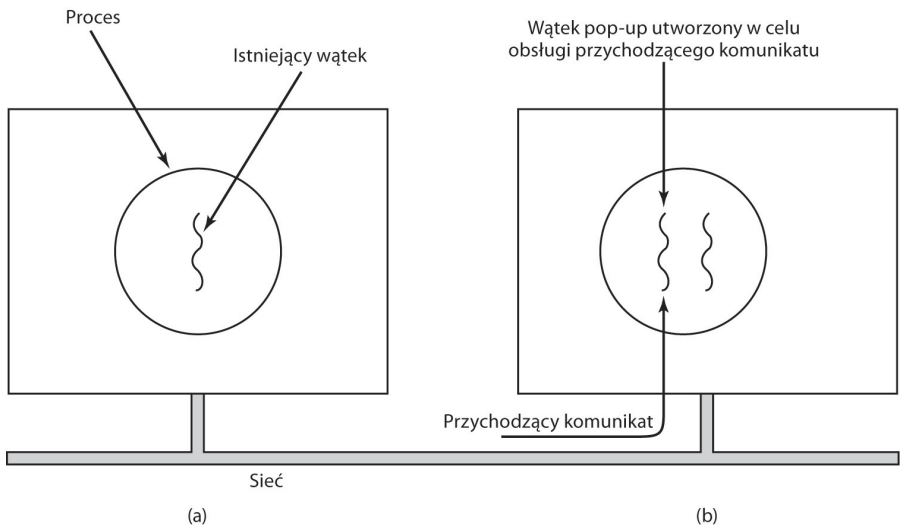
muje decyzję o tym, jakiemu wątkowi przydzielić dany procesor: przerwanemu, nowo przygotowanemu do działania czy jakiemuś innemu.

Wadą mechanizmu aktywacji zarządcy jest całkowite poleganie na wezwaniach — jest to pojęcie, które narusza wewnętrzną strukturę każdego systemu warstwowego. Zwykle warstwa n oferuje określone usługi, które może wywołać warstwa $n+1$, warstwa n nie może jednak wywoływać procedur w warstwie $n+1$. Mechanizm wezwań narusza tę podstawową zasadę.

2.2.8. Wątki pop-up

Wątki często się przydają w systemach rozproszonych. Istotnym przykładem może być sposób postępowania z nadchodzącymi komunikatami — np. żądaniami obsługi. Tradycyjne podejście polega na tym, że na komunikat oczekuje proces lub wątek zablokowany na wywołaniu systemowym `receive`. Kiedy nadejdzie komunikat, wątek ten przyjmuje go, rozpakowuje, analizuje zawartość i przetwarza.

Możliwe jest jednak całkiem inne podejście — po dotarciu komunikatu system tworzy nowy wątek do jego obsługi. Taki wątek określa się jako *wątek pop-up*. Zilustrowano go na rysunku 2.11. Zasadnicza zaleta wątków pop-up polega na tym, że ponieważ są one zupełnie nowe, nie mają żadnej historii — rejestrów, stosu, czegokolwiek, co musiałoby być odtworzone. Każdy wątek rozpoczyna się jako nowy i wszystkie są identyczne. Dzięki temu takie wątki można tworzyć szybko. Nowy wątek otrzymuje przychodzący komunikat do przetworzenia. Dzięki zastosowaniu wątków pop-up opóźnienie pomiędzy przybyciem komunikatu a rozpoczęciem jego przetwarzania jest bardzo niewielkie.



Rysunek 2.11. Tworzenie nowego wątku po przybyciu pakietu: (a) zanim nadejdzie komunikat; (b) po nadejściu komunikatu

W przypadku wykorzystania wątków pop-up potrzebne jest pewne zaawansowane planowanie. Przykładowo: w którym procesie działa wątek? Jeśli system obsługuje wątki działające w kontekście jądra, to wątek może tam działać (dlatego właśnie nie pokazaliśmy jądra na rysunku 2.11). Umieszczenie wątku pop-up w przestrzeni jądra zazwyczaj jest łatwiejsze i szybsze niż umieszczenie go w przestrzeni użytkownika. Ponadto wątek pop-up w przestrzeni jądra

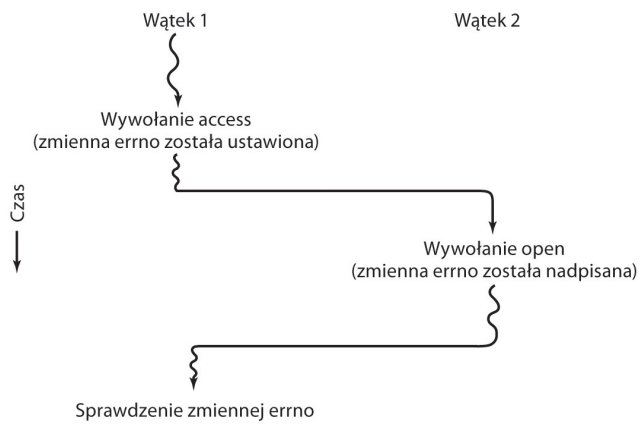
może łatwo uzyskać dostęp do wszystkich tabel i urządzeń wejścia-wyjścia jądra, co może być potrzebne do przetwarzania przerwań. Z drugiej strony działający błędnie wątek jądra może zrobić więcej szkód niż błędnie działający wątek przestrzeni użytkownika. Jeśli np. działa zbyt długo i nie ma możliwości jego wywłaszczenia, wchodzące dane mogą zostać utracone.

2.2.9. 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.

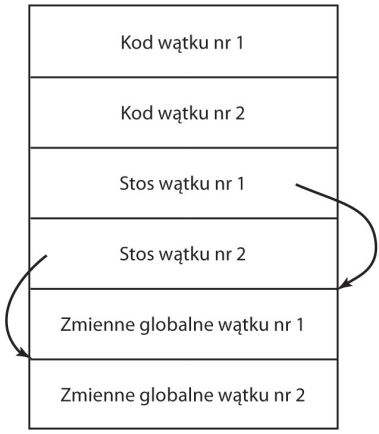
Na początek należy sobie uświadomić, że wątek, tak jak proces, zwykle składa się z wielu procedur. Mogą one mieć zmienne lokalne, zmienne globalne i parametry. Zmienne lokalne i parametry nie powodują żadnych problemów, tymczasem zmienne, które są globalne dla wątku, ale nie są globalne dla całego programu, 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 2.12 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. Wątek 2. uruchomił wywołanie systemowe `open`, które się nie powiodło. Spowodowało to nadpisanie zmiennej `errno`, a kod dostępu wątku 1. został utracony na zawsze. Kiedy wątek 1. później się uruchomi, przeczyta nieprawidłową wartość i będzie działał nieprawidłowo.



Rysunek 2.12. 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 2.13. W ten



Rysunek 2.13. Wątki mogą mieć prywatne zmienne globalne

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.

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.

Alternatywnie można stworzyć nowe procedury biblioteczne do tworzenia, ustawiania i czytania tych zmiennych globalnych na poziomie wątku. Pierwsze wywołanie może mieć następującą 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 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? Nowo utworzony wątek pop-up? 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 zmo-

dyfikować 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 następujących pozycjach: [Hauser et al., 1993], [Marsh et al., 1991] oraz [Rodrigues et al., 2010].

2.3. KOMUNIKACJA MIĘDZY PROCESAMI

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. *InterProcess 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.

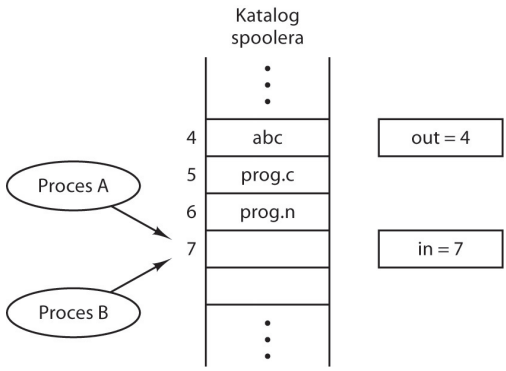
Warto również wspomnieć o tym, że dwa spośród tych problemów mają w równym stopniu zastosowanie do wątków. Pierwszy z nich — przekazywanie informacji — jest łatwy w odniesieniu do wątków, ponieważ wykorzystują one wspólną przestrzeń adresową — wątki w różnych przestrzeniach adresowych, które muszą się komunikować, można zaliczyć do tej samej klasy problemów, do których należy komunikacja pomiędzy procesami. Jednak pozostałe dwa problemy — powstrzymanie od „skakania sobie do oczu” i kolejkowanie — mają zastosowanie do procesów w takim samym stopniu, jak do wątków. Występują te same problemy i można zastosować takie same rozwiązania. Poniżej omówimy te problemy w kontekście procesów. Pamiętajmy jednak o tym, że te same problemy i rozwiązania mają zastosowanie także do wątków.

2.3.1. 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 2.14.



Rysunek 2.14. Dwa procesy w tym samym czasie chcą uzyskać dostęp do wspólnej pamięci

W przypadkach, w których mają zastosowanie prawa Murphy'ego[1], 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 *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.

[1] Jeśli coś może się nie udać, to się nie uda.

2.3.2. Regiony krytyczne

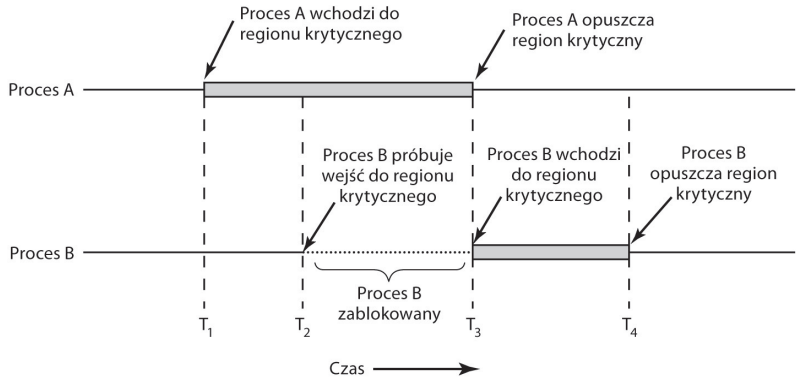
W jaki sposób uniknąć sytuacji wyścigu? Kluczem do zapobiegania kłopotom w tej sytuacji, a także w wielu innych sytuacjach dotyczących współdzielonej pamięci, współdzielonych plików oraz innych współdzielonych zasobów jest znalezienie sposobu na niedopuszczenie do tego, by więcej procesów niż jeden czytało 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 sekcji krytycznych.
2. Nie można przyjmować żadnych założeń dotyczących szybkości lub liczby procesorów.
3. Proces działający wewnątrz swojego regionu krytycznego nie może blokować innych procesów.
4. Żaden proces nie powinien oczekiwać w nieskończoność na dostęp do swojego regionu krytycznego.

W sensie abstrakcyjnym właściwości, które nas interesują, pokazano na rysunku 2.15. 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 2.15. Wzajemne wykluczanie z wykorzystaniem regionów krytycznych

2.3.3. 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.

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.

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 maszynach wyższej klasy, a w niedalekiej przyszłości można spodziewać się maszyn z ośmioma lub szesnastoma 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.

Blokowanie zmiennych

W drugiej kolejności przeanalizujmy rozwiązanie programowe. Rozważmy sytuację, w której mamy pojedynczą, współdzieloną zmienną (`blokada`), która początkowo ma wartość 0. Kiedy proces chce wejść do regionu krytycznego, najpierw sprawdza zmienną `blokada`. 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 2.3. 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 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].

W kodzie na listingu 2.3 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. Ciągłe testowanie zmiennej do czasu, aż osiągnie ona pewną wartość, nosi nazwę *aktywnego oczekiwania*. 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*).

Listing 2.3. 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();
}
```

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.

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 2.4. 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 2.4. 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 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 REGISTER,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. Procesor, który uruchamia instrukcję TSL, blokuje magistralę 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 procesorów, 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 2.5. 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.

Listing 2.5. 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. Przed wejściem do regionu krytycznego proces wywołuje funkcję enter_region. 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 bazujący na rozkazie `XCHG` zaprezentowano na listingu 2.6. Jak można zauważyć, zasadniczo jest on identyczny jak rozwiązanie z instrukcją `TSL`. Niskopoziomową synchronizację w oparciu o rozkaz `XCHG` wykorzystują wszystkie procesory x86 firmy Intel.

Listing 2.6. 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
```

2.3.4. Wywołania sleep i wakeup

Zarówno rozwiązanie Petersona, jak i rozwiązanie bazujące na rozkazach `TSL` lub `XCHG` są prawidłowe, 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 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. W związku z tym proces *H* wykonuje pętlę nieskończoną. 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. Alternatywnie zarówno operacja `sleep`, jak i `wakeup` mają po jednym parametrze — adresie pamięci wykorzystywanym 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. Jeśli nie, producent dodaje element do bufora i inkrementuje zmienną `count`.

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 2.7.

Listing 2.7. 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 8, albo nawet 32, ale zasadniczy problem i tak pozostanie.

2.3.5. 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 2.8. Aby działały prawidłowo, istotne znaczenie ma zaimplementowanie ich w sposób niepodzielny. Oczywistą metodą jest zaimplementowanie operacji `up` i `down` w postaci wywołań systemowych. 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 2.8. 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 */
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 niniejszego rozdziału.

W przykładzie z listingu 2.8 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.

2.3.6. Muteksy

Jeśli nie jest potrzebna właściwość zliczania, czasami używa się uproszczonej wersji semaforów zwanych *muteksami* (ang. *mutex*). Muteksy nadają się wyłącznie do zarządzania wzajemnym wykluczaniem niektórych współdzielonych zasobów lub fragmentu kodu. Są one łatwe i wydajne do implementacji. 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`, które można wykorzystać z pakietem obsługi wątków poziomu użytkownika pokazano na listingu 2.9. Rozwiązanie z instrukcją `XCHG` jest w zasadzie takie samo.

Listing 2.9. 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?

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. 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 początkowo blokada ma wartość 1 — co zgodnie z założeniem oznacza, że blokada jest wolna. 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. 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 ope-

racji 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.

Oprócz muteksów pakiet `Pthreads` oferuje inny mechanizm synchronizacji: *zmienne warunkowe*. 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.

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

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.

Niektóre wywołania związane 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 na rysunku). Podstawowe operacje na zmiennych warunkowych to `pthread_cond_wait` i `pthread_c`
`↪ond_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.

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. Z tego powodu muteks jest jednym z parametrów wywołania.

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

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 2.10 pokazano proste rozwiązanie problemu producent-konsument z pojedynczym buforem. Kiedy 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.

Listing 2.10. 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);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

2.3.7. Monitory

W przypadku użycia semaforów i muteksów komunikacja między procesami wydaje się łatwa. Zgadza się? Nic bardziej mylnego. Przyjrzyjmy się dokładniej kolejności operacji `down` przed wstawieniem lub usunięciem elementów z bufora w kodzie na listingu 2.8. 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 2.11 zilustrowano monitor napisany w wymyślonym języku Pidgin Pascal. Nie można tu użyć języka C, ponieważ monitory są konstrukcjami języka, a język C ich nie posiada.

Listing 2.11. 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 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`. Charles A.R. Hoare zaproponował umożliwienie działania przebudzonemu procesowi i zawieszenie drugiego z nich. Per 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 2.12. 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.

Listing 2.12. Szkielet rozwiązania problemu producent-konsument z wykorzystaniem monitorów. Tylko jedna procedura monitora jest aktywna w danym momencie. Bofor 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 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 2.13. Rozwiązanie składa się z czterech klas. Klasa zewnętrzna — `Producer ↪Consumer` — 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 2.13. 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
            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.

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 zezwala również na wymianę informacji pomiędzy maszynami. Potrzebne jest inne rozwiązanie.

2.3.8. 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. Alternatywnie może on natychmiast zwrócić sterowanie, przekazując 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 komunikaty mogą być utracone w sieci.

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 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 i Wetherall, 2010].

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. Przykładowo [Cheriton, 1984] zasugerował takie ograniczenie rozmiaru komunikatu, aby zmieścił się on w rejestrach maszyny. Przekazywanie komunikatów mogłoby się wówczas odbywać z wykorzystaniem rejestrów.

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 2.14. 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 2.14. 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 */
        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.

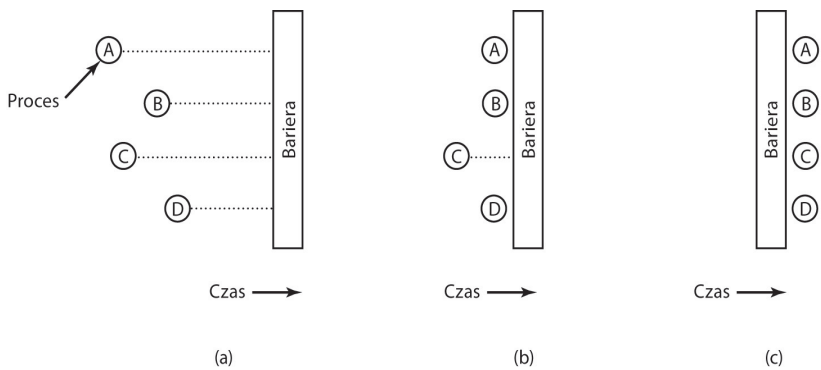
Przeciwległym ekstremum do posiadania skrzynek pocztowych jest całkowite wyeliminowanie buforowania. W przypadku zastosowania tego podejścia, jeśli zostanie wykonana operacja `send` przed wykonaniem operacji `receive`, proces wysyłający będzie zablokowany do chwili

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 *rendez-vous* (z fr. spotkanie). Jest ona łatwiejsza do zaimplementowania od mechanizmu z buforowaniem, ale mniej elastyczna, ponieważ nadawca 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].

2.3.9. 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 blokowany do czasu, aż wszystkie procesy osiągną barierę. Pozwala to grupom procesów na synchronizację. Działanie bariery zilustrowano na rysunku 2.16.



Rysunek 2.16. 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 2.16(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 2.16(b). Na koniec, kiedy ostatni proces — C — dotrze do bariery, wszystkie procesy są zwalniane, tak jak pokazano na rysunku 2.16(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.

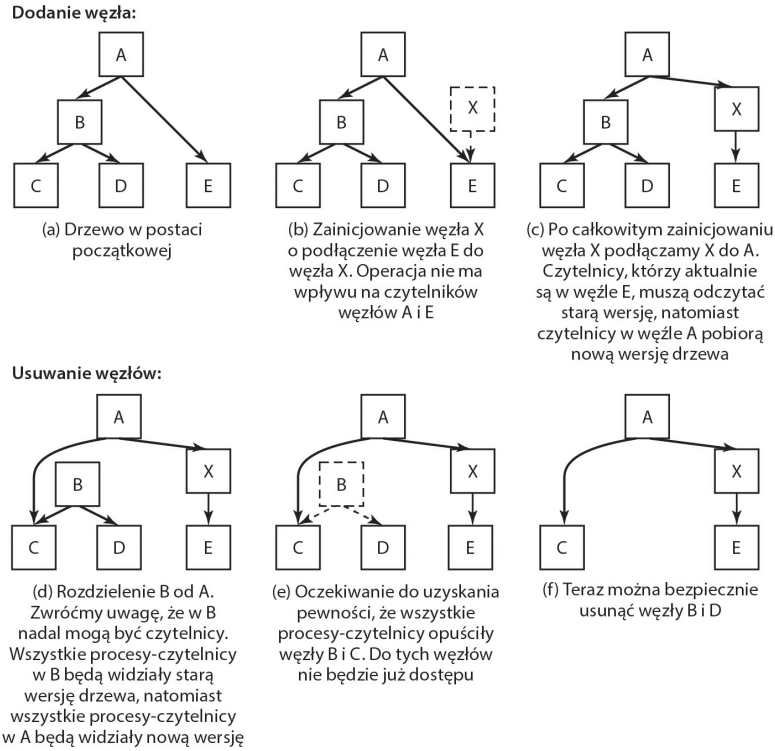
Począwszy od bieżących wartości macierzy wartości, wykonywane jest przekształcenie, w wyniku którego otrzymujemy drugą wersję macierzy. Stosując np. prawa termodynamiki, obliczamy temperatury punktów po upływie czasu T. Następnie proces jest powtarzany i w ten sposób, w miarę nagrzewania się arkusza, 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×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.

2.3.10. Unikanie blokad: odczyt-kopiowanie-aktualizacja

Najszybsze blokady to całkowity brak blokad. Pytanie brzmi: czy bez blokowania możemy pozwolić na równoczesny dostęp do odczytu i zapisu wspólnej struktury danych? Ogólnie odpowiedź jest oczywiście przecząca. Wyobraźmy sobie proces A sortujący tablicę liczb w czasie, kiedy proces 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 2.17. 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. 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*).



Rysunek 2.17. Operacja odczyt-kopiowanie-aktualizacja: wstawienie węzła do drzewa, a następnie usunięcie gałęzi — bez blokad

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.

2.4. 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 punktów.

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.

2.4.1. 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. 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. 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 wielu urządzeń mobilnych, takich jak smartfony (może z wyjątkiem najmocniejszych modeli), oraz węzłów w sieciach sensorowych. W tego rodzaju urządzeniach procesory CPU bywają słabe, a ilość pamięci jest niewielka. Ponadto ze względu na to, że w tych urządzeniach żywotność baterii jest jednym z najważniejszych ograniczeń, niektóre programy szeregujące dążą do optymalizacji zużycia energii.

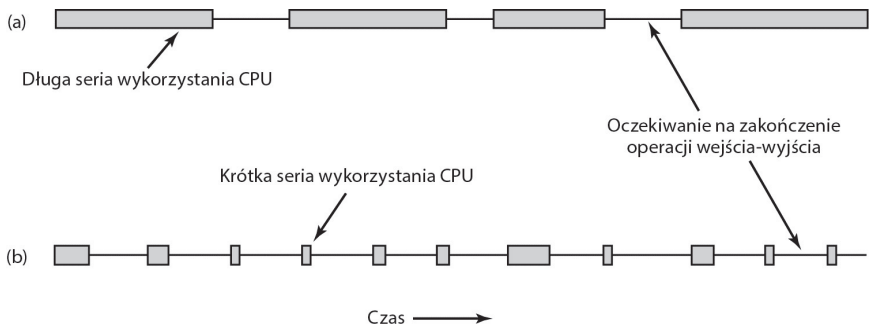
Oprócz wyboru właściwego procesu do uruchomienia program szeregujący musi również dbać o wydajne wykorzystanie procesora, ponieważ przełączanie procesów jest kosztowną operacją. Na początek musi nastąpić przełączenie z trybu użytkownika do trybu jądra. Następnie należy zapisać stan bieżącego procesu, 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). 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) żądaniami wejścia-wyjścia, co pokazano na rysunku 2.18. 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 niniejszym przykładzie, zachodzi wtedy, gdy proces wchodzi do stanu zablokowania w oczekiwaniu na to, aż urządzenie zewnętrzne zakończy pracę.

Ważną rzeczą, na którą należy zwrócić uwagę na rysunku 2.18, jest to, że niektóre procesy, np. ten z rysunku 2.18(a), poświęcają większość czasu na obliczenia, podczas gdy inne, jak ten z rysunku 2.18(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.



Rysunek 2.18. 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

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. Oczywiście w przypadku, gdy kilka procesów będzie gotowych, zarządca będzie mógł uruchomić w następnej kolejności ważniejszy proces. Podstawowa idea w tym przypadku polega na tym, że jeśli proces zorientowany na wejście-wyjście chce działać, powinien szybko otrzymać swoją szansę. Dzięki temu będzie mógł wysłać swoje żądanie operacji dyskowej, przez co zadba o to, by dysk miał co robić. Jak widzieliśmy na rysunku 2.4, kiedy procesy są zorientowane na wejście-wyjście, potrzeba ich dość dużo, aby procesor był przez cały czas zajęty.

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 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 wejścia-wyjścia na semaforze (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.

Wreszcie: 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 na końcu przydzielonego przedziału czasu proces dalej działa, jest zawieszany i program szeregujący wybiera inny proces do uruchomienia (jeśli jest dostępny). Aby była możliwa realizacja szeregowania z wywłaszczaniem, na końcu przedziału czasowego musi nastąpić przerwanie zegara. Dzięki temu program szeregujący odzyskuje kontrolę nad procesorem. Jeśli nie jest dostępne przerwanie zegara, jedyną opcją okazuje się szeregowanie bez wywłaszczania.

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 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. Takie podejście zmniejsza liczbę przełączeń procesów, a tym samym poprawia wydajność. 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 założenia zestawiono w tabeli 2.8. Poniżej omówimy je po kolei.

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.

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.

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 wejścia-wyjścia. W przypadku 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.

Często w systemach wsadowych wykorzystuje się procesor. 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 wykorzystania 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 poostrzegane 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 500 MB 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. Zagadnienia szeregowania procesów w systemach czasu rzeczywistego nie zostały omówione w tym rozdziale. Są jednak opisane w ramach dodatku dotyczącego multimedialnych systemów operacyjnych znajdującego się na końcu książki.

2.4.2. Szeregowanie w systemach wsadowych

Teraz nadszedł czas, by przejść od ogólnych problemów szeregowania do specyficznych 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.

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 2.19. 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.

Rysunek 2.19. 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 2.19(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

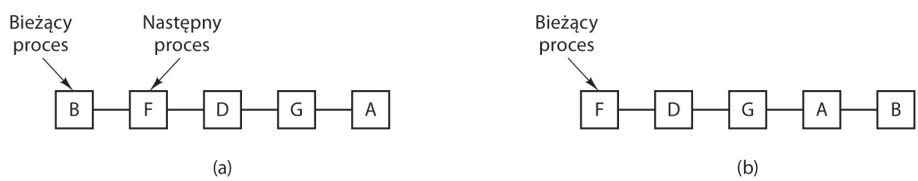
Odmianą algorytmu „najpierw najkrótsze zadanie" z wywłaszczaniem jest algorytm *„następny proces o najkrótszym pozostałym czasie działania"*. W przypadku użycia tego algorytmu program szeregujący zawsze wybiera proces, którego pozostały czas działania jest najkrótszy. W tym przypadku czas działania również 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.

2.4.3. 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 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 2.20. Kiedy proces wykorzysta swój kwant, jest umieszczany na końcu listy, co pokazano na rysunku 2.20(b).



Rysunek 2.20. 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 — zapisanie i załadowanie rejestrów i mapy pamięci, aktualizacji różnych tabel i list, opróżnienie i ponowne załadowanie pamięci podręcznej itp. Załóżmy, że to *przełączenie procesów* lub *przełączenie kontekstu*, jak się je czasami nazywa, 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.

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.

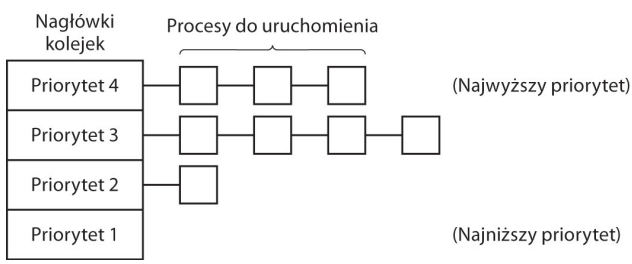
Aby nie dopuścić do tego, by procesy o wysokich priorytetach działały w nieskończoność, program szeregujący może zmniejszać priorytet działających procesów wraz z każdym cyklem

zegara. Jeśli działanie to spowoduje obniżenie priorytetu poniżej priorytetu następnego w kolejności procesu, następuje przełączenie procesów. Alternatywnie każdemu procesowi może być przydzielony maksymalny kwant czasu, przez który może on działać. Kiedy ten kwant zostanie wykorzystany, szansę na działanie otrzymuje następny proces w kolejności priorytetów.

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, porucznikó w — 60 itd. Alternatywnie w komercyjnym centrum obliczeniowym zadania o wysokim priorytecie mogą kosztować 100 dolarów na godzinę, o średnim priorytecie — 75 dolarów na godzinę, natomiast 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. Nikt nigdy go nie użył.

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 wejścia-wyjścia na długotrwałe oczekiwanie na procesor będzie oznaczało, że niepotrzebnie zajmie on pamięć przez długi czas. Prosty algorytm zapewniający dobrą obsługę dla procesów zorientowanych na 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 2.21 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 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.



Rysunek 2.21. Algorytm szeregowania z czterema klasami priorytetów

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. 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.

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.

Niżej opisaną strategię zastosowano, aby zapobiec sytuacji, w której proces potrzebujący działać przez długi czas przy pierwszym uruchomieniu, a potem zmieniający się w proces interaktywny, nie był zablokowany na zawsze. 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 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 $a = {}^1/_2$ 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 ${}^1/_8$.

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 $a = {}^1/_2$. 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. 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 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.

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 bilet loteryjny, a zasób otrzymuje proces będący w posiadaniu tego biletu. 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 bilety i w ten sposób zwiększać ich szanse na zwycięstwo. Jeśli w grze jest 100 biletów, 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 w oparciu o priorytety, gdy bardzo trudno stwierdzić, co właściwie oznacza priorytet 40, w tym przypadku reguła jest czytelna: proces posiadający procent *f* biletów otrzyma mniej więcej procent *f* wybranego zasobu.

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

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

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.

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 tego, 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 spełniałaby 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.

2.4.4. 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 zewnętrznego 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 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 zdarzenie 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. Omówienie konkretnych algorytmów szeregowania w systemach czasu rzeczywistego odłożymy do rozdziału 7., w którym będziemy omawiać multimedialne systemy czasu rzeczywistego.

2.4.5. 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.

2.4.6. Szeregowanie wątków

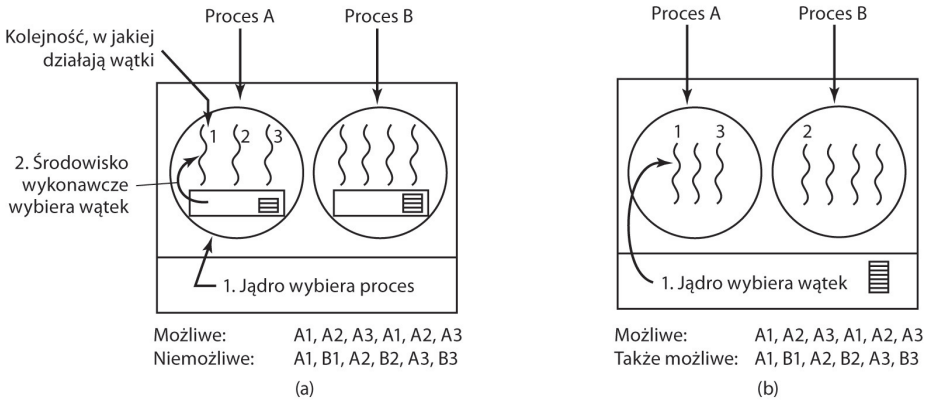
Jeśli każdy z kilku procesów składa się z kilku wątków, mamy do czynienia z dwoma poziomami współbieżności: procesami i wątkami. Szeregowanie w takich systemach różni się znacząco w zależności od tego, czy są wykorzystywane wątki na poziomie użytkownika, wątki na poziomie jądra (czy oba rodzaje).

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 2.22(a).



Rysunek 2.22. (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.

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.

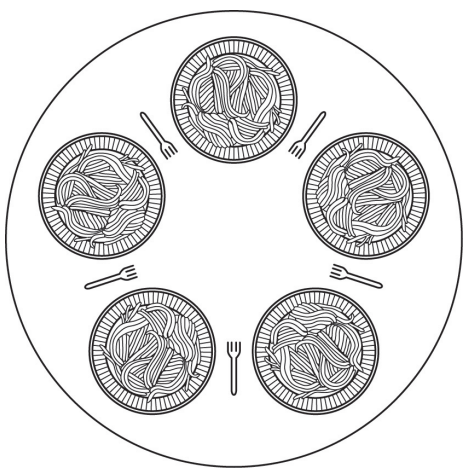
Istotną rolę odgrywa również to, że wątki zarządzane na poziomie użytkownika mogą wykorzystywać mechanizm szeregowania specyficzny dla aplikacji. Rozważmy dla przykładu serwer WWW z rysunku 2.6. Załóżmy, że wątek pracownika właśnie się zablokował, a wątek dyspozytora i dwa wątki pracowników są gotowe. 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.

2.5. KLASYCZNE PROBLEMY KOMUNIKACJI MIĘDZY PROCESAMI

Literatura dotycząca systemów operacyjnych pełna jest interesujących problemów, które były szeroko dyskutowane i analizowane przy użyciu różnych metod synchronizacji. W poniższych punktach przeanalizujemy trzy z bardziej znanych problemów.

2.5.1. Problem pięciu filozofów

W 1965 roku Dijkstra sformułował i rozwiązał problem synchronizacji, który nazwał *problemem pięciu filozofów*. Od tego czasu każdy, kto wynajdywał nowy prymityw synchronizacji, czuł się zobowiązany do zademonstrowania zalet nowego prymitywu poprzez pokazanie jego wykorzystania jako eleganckiego rozwiązania problemu pięciu filozofów. Problem dość prosto można sformułować w następujący sposób: pięciu filozofów siedzi przy okrągłym stole. Przed każdym z nich stoi talerz spaghetti. Jest ono tak śliskie, że filozofowie potrzebują dwóch widelców, aby mogli je jeść. Pomiędzy każdą parą talerzy leży jeden widelec. Rozmieszczenie filozofów przy stole pokazano na rysunku 2.23.



Rysunek 2.23. Obiad na wydziale filozofii

Życie filozofów składa się z naprzemiennych okresów jedzenia i rozmyślania (jest to pewna abstrakcja nawet w przypadku filozofów, ale inne działania nie są tutaj ważne). Kiedy filozof zaczyna czuć głód, próbuje sięgnąć po widelce z lewej i prawej strony — po jednym na raz i w dowolnej kolejności. Jeśli uda mu się zdobyć dwa widelce, je przez chwilę, a następnie odkłada widelce i kontynuuje rozmyślanie. Zasadnicze pytanie brzmi: czy potrafisz napisać program dla każdego z filozofów, który będzie wykonywał wymagane operacje i nigdy się nie zablokuje? (Wymaganie posiadania dwóch widelców jest trochę sztuczne; być może należałoby przerzucić się z kuchni włoskiej na chińską i zastąpić spaghetti ryżem, a widelce pałeczkami).

Oczywiste rozwiązanie pokazano na listingu 2.15. Procedura `take_fork` oczekuje, aż określony widelec stanie się dostępny, a następnie go podnosi. Niestety, oczywiste rozwiązanie jest błędne. Przypuśćmy, że wszystkich pięciu filozofów jednocześnie podniosło swoje lewe widelce. Żaden z nich nie będzie mógł podnieść swojego prawego widelca i powstanie zakleszczenie.

Listing 2.15. Błędne rozwiązanie problemu pięciu filozofów

```
#define N 5                     /* liczba filozofów */
void philosopher(int i)         /* i: numer filozofa — od 0 do 4 */
{
    while (TRUE) {
        think( );               /* filozof rozmyśla */
        take_fork(i);           /* podniesienie lewego widelca */
        take_fork((i+1) % N);   /* podniesienie prawego widelca; % to operator modulo */
        eat( );                 /* mniam, mniam, spaghetti */
        put_fork(i);            /* odłożenie lewego widelca na stół */
        put_fork((i+1) % N);    /* odłożenie prawego widelca na stół */
    }
}
```

Moglibyśmy zmodyfikować program w taki sposób, aby po wzięciu lewego widelca sprawdzał, czy prawy widelec jest dostępny. Jeśli nie, filozof powinien odłożyć lewy widelec, poczekać jakiś czas, a następnie powtórzyć cały proces. Propozycja ta również nie rozwiązuje problemu. Tym razem z innego powodu. Przy odrobinie pecha wszyscy filozofowie mogliby zacząć algorytm jednocześnie. Wzięliby widelce znajdujące się z lewej strony, zorientowaliby się, że po prawej stronie widelce są niedostępne, odłożyli widelce z lewej, poczekali, znów jednocześnie podnieśli widelce z lewej strony, i tak dalej, w nieskończoność. Taka sytuacja, w której wszystkie programy działają w nieskończoność bez żadnego postępu, nazywa się *zagłodzeniem* (nazywa się zagłodzeniem nawet wtedy, gdy akcja nie rozgrywa się we włoskiej czy też chińskiej restauracji).

Można by sądzić, że jeśli po nieudanej próbie podniesienia widelca z prawej strony filozofowie będą czekać przez losowy czas, zamiast zawsze taki sam, szansa na to, że system się zablokuje na długo, jest bardzo mała. Ta obserwacja okazuje się słuszna i niemal we wszystkich aplikacjach ponowienie próby za jakiś czas nie stanowi problemu. Jeśli np. w popularnej lokalnej sieci komputerowej Ethernet dwa komputery jednocześnie wyślą pakiet, każdy z nich czeka losowy czas i ponawia próbę. W praktyce takie rozwiązanie się sprawdza. W niektórych zastosowaniach potrzebne jest jednak rozwiązanie, które działa zawsze i nie może zawieść z powodu nieznanej serii liczb losowych. Wystarczy pomyśleć o systemie bezpieczeństwa w elektrowni atomowej.

Aby usprawnić kod z listingu 2.15 w taki sposób, by pozbawić go problemu zakleszczeń i zagłodzenia, wystarczy zabezpieczyć pięć instrukcji następujących po wywołaniu operacji `think` semaforem binarnym. Przed przystąpieniem do podnoszenia widelców filozof mógłby wykonać operację `down` na zmiennej `mutex`. Po odłożeniu widelców powinien on wykonać ope-

rację up na zmiennej mutex. Teoretycznie rozwiązanie to jest właściwe. Praktycznie charakteryzuje się obniżoną wydajnością: w dowolnym momencie będzie mógł jeść tylko jeden filozof. Ponieważ jest dostępnych pięć widelców, w tym samym czasie dwóch filozofów powinno mieć możliwość jedzenia.

Rozwiązanie zaprezentowane na listingu 2.16 jest wolne od zakleszczeń i umożliwia maksymalny stopień współbieżności dla dowolnej liczby filozofów. Wykorzystano w nim tablicę state, która służy do śledzenia tego, czy filozof je, myśli, czy jest głodny (próbuje wziąć widelce). Filozof może przejść do stanu jedzenia tylko wtedy, gdy żaden z jego sąsiadów nie je. Sąsiedzi filozofa o numerze *i* są zdefiniowani za pomocą makr LEFT i RIGHT. Mówiąc inaczej, jeśli *i* wynosi 2, to LEFT ma wartość 1, a RIGHT — 3.

Listing 2.16. Rozwiązanie problemu pięciu filozofów

```
#define N          5          /* liczba filozofów */
#define LEFT       (i+N-1)%N  /* numer lewego sąsiada filozofa i */
#define RIGHT      (i+1)%N    /* numer prawego sąsiada filozofa i */
#define THINKING   0          /* filozof rozmyśla */
#define HUNGRY     1          /* filozof próbuje podnieść widelce */
#define EATING     2          /* filozof je */
typedef int semaphore;        /* semafory to specjalny rodzaj danych typu int */
int state[N];                 /* tablica do śledzenia stanu filozofów */
semaphore mutex = 1;          /* wzajemne wykluczanie regionów krytycznych */
semaphore s[N];               /* jeden semafor na filozofa */
void philosopher(int i)       /* i: numer filozofa — od 0 do N–1 */
{
    while (TRUE) {            /* pętla nieskończona */
        think( );             /* filozof rozmyśla */
        take_forks(i);        /* podniesienie dwóch widelców lub zablokowanie */
        eat( );               /* mniam, mniam, spaghetti */
        put_forks(i);         /* odłożenie dwóch widelców na stół */
    }
}
void take_forks(int i)        /* i: numer filozofa — od 0 do N–1 */
{
    down(&mutex);             /* wejście do regionu krytycznego */
    state[i] = HUNGRY;        /* zarejestrowanie faktu, że filozof jest głodny */
    test(i);                  /* próba podniesienia dwóch widelców */
    up(&mutex);               /* opuszczenie regionu krytycznego */
    down(&s[i]);              /* zablokowanie, jeśli nie podniesiono widelców */
}
void put_forks(i)             /* i: numer filozofa — od 0 do N–1 */
{
    down(&mutex);             /* wejście do regionu krytycznego */
    state[i] = THINKING;      /* filozof zakończył jedzenie */
    test(LEFT);               /* sprawdzenie, czy sąsiad z lewej strony może teraz jeść */
    test(RIGHT);              /* sprawdzenie, czy sąsiad z prawej strony może teraz jeść */
    up(&mutex);               /* opuszczenie regionu krytycznego */
}
void test(i)                  /* i: numer filozofa — od 0 do N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Program wykorzystuje tablicę semaforów — po jednym dla każdego filozofa. W związku z tym, jeśli potrzebne widelce są zajęte, głodny filozof przechodzi do stanu zablokowania. Zwróćmy uwagę, że każdy proces uruchamia procedurę `philosopher` jako swój główny kod, natomiast inne procedury: `take_forks`, `put_forks` i `test` są zwykłymi procedurami, a nie oddzielnymi procesami.

2.5.2. Problem czytelników i pisarzy

Problem pięciu filozofów przydaje się do modelowania procesów, które rywalizują o wyłączny dostęp do ograniczonych zasobów, np. urządzeń wejścia-wyjścia. 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 2.17.

Listing 2.17. 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 other;                          /* 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 */
    }
}
```

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.

Aby zapobiec tej sytuacji, 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.

2.6. 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.

Wątki są nowszym mechanizmem niż procesy, ale i one są obecne już od dość długiego czasu. Pomimo to od czasu do czasu pojawia się artykuł poświęcony wątkom — np. na temat klasteryzacji wątków w systemach wieloprocesorowych [Tam et al., 2007] lub skalowania w nowoczesnych systemach operacyjnych z obsługą wielu wątków i wielu rdzeni, takich jak Linux [Boyd-Wickizer, 2010].

Szczególnie aktywny obszar to badania dotyczące rejestrowania i odtwarzania realizacji procesów [Viennot et al., 2013]. Odtwarzanie pomaga programistom wyśledzić trudne do znalezienia błędy, a ekspertom od zabezpieczeń — badać incydenty.

Wiele współczesnych badań w społeczności zajmującej się tematyką systemów operacyjnych dotyczy zagadnień bezpieczeństwa. Liczne przykłady dowodzą, że użytkownicy potrzebują lepszej ochrony przed intruzami (a czasami przed samymi sobą). Jedno z podejść polega na śledzeniu i uważnym ograniczaniu przepływu informacji w systemie operacyjnym [Giffin et

al., 2012]. Szeregowanie (zarówno w systemach jednoprocesorowych, jak i wieloprocesorowych) w dalszym ciągu jest zagadnieniem znajdującym się w kręgu zainteresowania badaczy. Niektóre badania dotyczą zagadnień energooszczędnego szeregowania na urządzeniach mobilnych [Yuan i Nahrstedt, 2006], szeregowania z obsługą hiperwątkowości [Bulpin i Pratt, 2005] oraz szeregowania z uprzedzeniami (ang. *bias-aware scheduling*) [Koufaty, 2010]. Wraz ze wzrostem zapotrzebowania na obliczenia wykonywane na słabych, ograniczonych pojemnością baterii smartfonach niektórzy badacze proponują przenieść procesy do bardziej wydajnych serwerów w chmurze [Gordon et al., 2012]. Trzeba jednak przyznać, że nie ma zbyt wielu projektantów systemu, którzy chodziliby cały dzień z miejsca na miejsce, załamując ręce z powodu braku dobrych algorytmów szeregowania wątków. Zatem wygląda na to, że taki typ badań jest bardziej inspirowany przez samych badaczy niż przez oczekiwania użytkowników. Podsumujmy: procesy, wątki i szeregowanie nie są gorącymi tematami badań, tak jak to bywało kiedyś. Badania są prowadzone nad takimi zagadnieniami 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 przypadku niektórych aplikacji przydatne jest istnienie wielu wątków sterowania w obrębie pojedynczego procesu. 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.

Procesy mogą się ze sobą komunikować z wykorzystaniem prymitywów 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 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.

Prymitywy komunikacji między procesami można wykorzystać do rozwiązywania takich problemów, jak producent-konsument, pięciu filozofów oraz czytelnik-pisarz. Nawet w przypadku stosowania tych prymitywów należy zachować ostrożność w celu uniknięcia błędów i zakleszczeń.

W niniejszym 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 bazujące na priorytetach, wielopoziomowe kolejki, szeregowanie gwarantowane oraz 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 pięć programów. Przez połowę czasu programy te są w stanie oczekiwania na wejście-wyjście. Jaki ułamek czasu procesora jest marnotrawiony?
6. Komputer jest wyposażony w 4 GB pamięci RAM, z której system operacyjny zajmuje 512 MB. Każdy proces zajmuje 256 MB (dla uproszczenia). Wszystkie procesy mają te same własności. Jaki jest maksymalny czas oczekiwania na wejście-wyjście, 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 6. stopnia (tzn. w tym samym czasie w pamięci jest sześć programów). Załóżmy, że każdy proces spędza 40% swojego czasu w oczekiwaniu na wejście-wyjście. Ile wynosi procent wykorzystania procesora?
9. 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 wykorzystać wątki do poprawy czasu pobierania.
10. W tekście rozdziału powiedziano, że model z rysunku 2.7(a) nie był odpowiedni dla serwera plików wykorzystującego cache w pamięci. Dlaczego nie? Czy każdy proces mógłby mieć własną pamięć cache?
11. Jeśli nastąpi rozwidlenie wielowątkowego procesu, problem występuje w przypadku, gdy proces-dziecko otrzyma kopie wszystkich wątków procesu-rodzica. Załóżmy, że jeden z wyjściowych wątków oczekiwał na dane wejściowe z klawiatury. Teraz na dane z klawiatury czekają dwa wątki — po jednym w każdym procesie. Czy ten problem kiedykolwiek występuje w przypadku procesów jednowątkowych?

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 2.4 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. Czy wątek może być wywłaszczony za pomocą przerwania zegara? Jeśli tak, to w jakich okolicznościach? Jeśli nie, to dlaczego?
17. 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?
18. Jaka jest największa zaleta implementacji wątków w przestrzeni użytkownika? A jaka jest największa wada tego sposobu implementacji?
19. 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?
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 w środowisku wykonawczym działa jakiś inny wątek. 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 w takich warunkach istnieje możliwość zaimplementowania pakietu obsługi wątków w przestrzeni użytkownika? Uzasadnij.

23. Czy rozwiązanie z aktywnym oczekiwaniem, w którym wykorzystano zmienną `turn` (listing 2.3), będzie działać, jeśli dwa procesy działają w systemie wieloprocesorowym ze współdzieloną pamięcią — tzn. gdy dwa procesory współdzielą pamięć?
24. 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?
25. 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?
26. 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 w oparciu o priorytety jest wykorzystywane szeregowanie cykliczne? Uzasadnij.
27. 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.
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 sytuacji wyścigu, jeśli nie ma jednoczesnych zdarzeń, tak jak to ma miejsce w tym przypadku?
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?
30. Rozważmy następujące rozwiązanie problemu wzajemnego wykluczania z udziałem dwóch procesów *P0* i *P1*. Załóżmy, że zmienna `turn` jest inicjowana wartością. 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. W jaki sposób można zaimplementować semafory w systemie operacyjnym, który może wyłączyć przerwania?
32. Pokaż sposób implementacji semaforów zliczających (tzn. semaforów zdolnych do przechowywania dowolnych wartości) z wykorzystaniem semaforów binarnych oraz standardowych instrukcji maszynowych.
33. Jeśli w systemie działają tylko dwa procesy, to czy jest sens, aby wykorzystywać barierę do ich synchronizacji? Dlaczego tak lub dlaczego nie?

34. 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.
35. W synchronizacji w obrębie monitorów wykorzystuje się zmienne warunkowe oraz dwie specjalne operacje: `wait` i `signal`. W synchronizacji w bardziej ogólnej postaci powinien występować pojedynczy prymityw, `waituntil`, do którego byłby przekazywany parametr w postaci dowolnego predykatu typu `Boolean`. Można by zatem użyć następującej operacji:

    ```
    waituntil x < 0 or y + z < n
    ```

    Prymityw `signal` przestałby być potrzebny. Ten mechanizm jest znacznie bardziej ogólny od mechanizmu Hoare'a lub Brincha Hansena, ale nie jest wykorzystywany. Dlaczego nie? *Wskazówka:* pomyśl o implementacji.
36. W restauracji fast food zatrudniono pracowników czterech typów: (1) zbieraczy zamówień, którzy przyjmują zamówienia od klientów; (2) kucharzy, którzy przygotowują jedzenie; (3) specjalistów od pakowania, którzy pakują jedzenie w torebki oraz (4) kasjerów, którzy wręczają torebki klientom i biorą od nich pieniądze. Każdego pracownika można uznać za proces sekwencyjny komunikujący się z innymi procesami. Jaką formę komunikacji międzyprocesowej wykorzystują procesy? Porównaj ten model do procesów w Uniksie.
37. 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. Zamiast tego otrzymuje kod błędu. Proces odpowiada na błąd poprzez wielokrotne ponawianie próby — tak długo, aż się powiedzie. Czy taki schemat prowadzi do sytuacji wyścigu?
38. 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 itp. 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?
39. Rozważmy następujący fragment kodu w języku C:

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

    Ile procesów potomnych zostanie stworzonych w wyniku uruchomienia tego programu?
40. 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?

41. 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?
42. Wyjaśnij, jaki wpływ mają na siebie wartość kwantu czasu i czas przełączania kontekstu w cyklicznym algorytmie szeregowania.
43. 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) $Q =$
    (b) $Q > T$
    (c) $S < Q < T$
    (d) $Q = S$
    (e) Q jest bliskie 0.
44. 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)?
45. 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 bazujące 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.
46. 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)?
47. 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?
48. Czy w systemie opisanym powyżej można dodać kolejny strumień wideo, jeśli system nadal ma być szeregowalny?
49. 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?

50. 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?

51. Załóżmy, że do rozwiązania problemu pięciu filozofów zastosowano następującą procedurę: filozof oznaczony parzystym numerem zawsze podnosi widelec z lewej strony przed podniesieniem tego z prawej, natomiast filozof oznaczony nieparzystym numerem zawsze podnosi widelec z prawej strony przed podniesieniem tego z lewej. Czy ta procedura gwarantuje działanie bez zakleszczeń?

52. 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?

53. Rozważ system, w którym pożądane jest oddzielenie strategii od mechanizmu szeregowania wątków zarządzanych na poziomie jądra. Zaproponuj sposoby osiągnięcia tego celu.

54. Dlaczego w rozwiązaniu problemu pięciu filozofów (listing 2.16) zmienną `state` w procedurze `take_forks` ustawiono na wartość `HUNGRY`?

55. Przeanalizuj procedurę `put_forks` z listingu 2.16. Przypuśćmy, że zmienną `state[i]` ustawiono na `THINKING` *po* dwóch wywołaniach funkcji `test`, a nie *przed*. W jaki sposób ta zmiana wpłynie na rozwiązanie?

56. 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. 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.

57. 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).

58. Przypuśćmy, że mamy do czynienia z systemem operacyjnym, który udostępnia semafory. Zaimplementuj system komunikatów. Napisz procedury wysyłania i odbierania komunikatów.

59. Rozwiąż problem pięciu filozofów z wykorzystaniem monitorów zamiast semaforów.

60. Przypuśćmy, że władze uniwersytetu chcą pokazać polityczną poprawność i zastosować doktrynę instytucji U.S. Supreme Court (*separate but equal is inherently unequal* — oddzielnie, choć tak samo, to i tak nie jest równość). Chcą ją zastosować zarówno dla płci, jak i dla ras i zlikwidować ugruntowaną praktykę oddzielnych łazienek dla poszczególnych płci w kampusie. Jednak w celu uszanowania tradycji postanowiono zastosować odstęp-

stwo od zasady polegające na tym, że jeśli w łazience jest kobieta, to mogą do niej wejść inne kobiety, ale nie mogą wchodzić mężczyźni, i na odwrót. Znak na drzwiach łazienki wskazuje, w jakim spośród trzech możliwych stanów się ona znajduje:

- Wolna.
- Zajęta przez kobiety.
- Zajęta przez mężczyzn.

W dowolnie wybranym języku programowania napisz następujące procedury: `kobieta_chce_wejsc`, `mezczyzna_chce_wejsc`, `kobieta_wychodzi`, `mezczyzna_wychodzi`. Możesz wykorzystać dowolne liczniki i techniki synchronizacji.

61. Przepisz program z listingu 2.3 w taki sposób, aby obsługiwał więcej niż dwa procesy.
62. 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.
63. 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.
64. Celem tego ćwiczenia jest implementacja wielowątkowego rozwiązania sprawdzania, czy podana wartość jest liczbą idealną. *N* jest liczbą idealną, jeśli suma wszystkich jej dzielników, z wyłączeniem jej samej, wynosi *N*. Przykładami takich liczb są 6 i 28. Dane wejściowe to liczba *N*. Algorytm zwraca `true`, jeśli liczba jest idealna i `false` w przeciwnym razie. Główny program będzie czytać liczby *N* i *P* z wiersza poleceń. Główny proces utworzy zbiór *P* wątków. Liczby o wartościach od 1 do *N* zostaną podzielone między te wątki w taki sposób, aby żadne dwa wątki nie przetwarzały tej samej liczby. Dla każdego numeru z tego zbioru wątek sprawdzi, czy liczba jest dzielnikiem *N*. Jeśli tak jest, doda tę liczbę do wspólnego bufora, w którym są przechowywane dzielniki *N*. Proces nadrzędny będzie czekać, aż wszystkie wątki zakończą działanie. Do rozwiązania zadania zastosuj odpowiedni prymityw synchronizacji. Proces macierzysty określi, czy liczba jest idealna, tzn. czy *N* jest sumą wszystkich jej podzielników, a następnie wyświetli odpowiedni komunikat. (*Uwaga*: Aby przyspieszyć obliczenia, można ograniczyć zakres przeszukiwanych liczb: od 1 do pierwiastka kwadratowego z *N*).

65. 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.

SKOROWIDZ

A

AAS, as a Service, 501
abstrakcja pamięci, 202
  przestrzenie adresowe, 205
  systemu plików, 786
ACE, Access Control Entries, 965
ACL, Access Control List, 605, 872
ACPI, 432, 878
adapter graficzny, 413
adres
  bazowy, 265
  liniowy, 265
  wirtualny, 262
ADSL, Asymmetric Digital Subscriber Line, 1068
AIDL, Android Interface Definition Language, 822
aktywne oczekiwanie, 144
aktywność, activity, 826
algorytm
  alokacji, 300
  bankiera, 459, 460
  bazujący na zbiorze roboczym, 233
  binarnego wykładniczego cofania, 540
  bliźniaków, 764
  CFS, 751
  deterministyczny, 566
  drugiej szansy, 229, 238
  FIFO, 229, 238
  LRU, 231, 238
  NRU, 228, 238
  odbierania ramek stron, 766
  PFF, 240
  Postarzanie, 238
  RMS, 1088
  scan-EDF, 1109
  strusia, 450
  szeregowania EDF, 1089
  windy szeregowania żądań, 390
  WSClock, 236, 238
  wykładniczego cofania binarnego, 571
  wymiany stron, 767
  zarządzania dyskiem, 387
  zastępowania ramek stron, 769
  zastępowania stron, 227, 235, 934
  zbiór roboczy, 238
  zegarowy, 230, 238
  zrzutu logicznego, 324
algorytmy
  bez wywłaszczania, 176
  heurystyczne
    inicjowane przez nadawcę, 566
    inicjowane przez odbiorcę, 567
  szeregowania, 173–176
    ramienia dysku, 388
  zastępowania stron, 226, 238
alokacja, 299
  pamięci, 239
  plików, 300
  przestrzeni dyskowej, 954
ALPC, Advanced LPC, 888, 913

Android, 46, 804
  aktywności, 826
  aplikacje, 824
  architektura, 810
  bezpieczeństwo, 836
  Binder IPC, 816
  blokady WakeLock, 812
  cele projektowe, 808
  Dalvik, 814
  dostawcy zawartości, 832
  hierarchia procesów, 810
  interakcje z usługami systemu, 811
  model procesów, 841
  odbiorcy, 831
  piaskownice aplikacji, 835
  rozszerzenia Linuksa, 811
  stan procesów, 845
  tworzenie procesu, 816
  usługi, 830
  zamiary, 834
Android 1.0, 807
APC, Asynchronous Procedure Call, 875, 882
API, Application Programming Interface, 488
  mechanizmu Binder, 821
  systemu NT, 865
  WinRT, 863
aplety, 701
aplikacje Androida, 824
architektura
  Androida, 810
  komputera, 32
  magistrali, 524
  równoległej magistrali, 59
  systemu NFS, 794
  współdzielonej magistrali, 59
  x86, 507
archiwizowanie instrukcji, 251
ARPANET, 572
ASLR, Adress Space Layout Randomization, 646, 970
asynchroniczne wywołania procedur, 882
atak
  drive-by-download, 639
  na warunek
    braku wywłaszczania, 463
    cyklicznego oczekiwania, 463
    oczekiwania, 463
    wstrzymania, 463
    wzajemnego wykluczania, 462
  Stuxnet, 628
ataki
  łańcuchy formatujące, 648
  na przepływ sterowania, 647
  odwołania do pustego wskaźnika, 652
  oprogramowanie szpiegujące, 679
  poprzez upodobnienie, 697
  przepełnienie bufora, 640
  przepełnienie liczb całkowitych, 653
  TOCTOU, 655
  wielokrotne wykorzystywanie kodu, 644
  wiszące wskaźniki, 651
  wstrzykiwanie kodu, 654
  z wewnątrz, 656
atestacja zdalna, remote atestation, 624
ATM, Automated Teller Machine, 633
atrybuty plików, 286
awaria, 395

B

badania
  dotyczące bezpieczeństwa, 704
  dotyczące systemów plików, 343
  dotyczące systemów wieloprocesorowych, 586
  dotyczące wejście-wyjścia, 433
  dotyczące zarządzania pamięcią, 267
  na temat multimediów, 1110
  na temat zakleszczeń, 469
  nad procesami i wątkami, 191
  nad wirtualizacją i chmurą, 517
balonikowanie, 494
bariery, 167
baza danych ramek stron, 936
Berkeley UNIX, 719
bezpieczeństwo, 593, 1037
  haseł, 629
  Javy, 701
  przez ukrywanie, 619
  systemów operacyjnych, 599
  w systemie Linux, 800
  wielopoziomowe, 612
bezpieczne wykonywanie apletów, 700
bezpośredni dostęp do pamięci, 355
biblioteka kernel32.dll, 917
biblioteki
  DLL, 246, 901
  dołączane dynamicznie, 246, 860
  współdzielone, 246
Binder IPC, 816
BIOS, Basic Input Output System, 61, 202, 890
BKL, Big Kernel Lock, 752
Blackberry OS, 46
blok, 313
  dwupośredni, 337
  jednopośredni, 337
  PEB, 905
  podpisu, 622
  TEB, 905
  trójpośredni, 337

blokady
WakeLock, 812
współdzielone, 781
wyłączne, 781
blokowanie
dwufazowe, 465
stron, 253
zmiennych, 145
Blu-ray, 1068
błędny sektor, 392
błędy
braku stron, 250, 930
odczytu, 392
przepełnienia bufora, 639
stron, 492
w kodzie, 638
bomby logiczne, 656
bootloader, 625
brak
abstrakcji pamięci, 202
strony, page fault, 216
BSOD, Blue Screen Of Death, 886
bufor TLB, 220, 929
buforowanie, caching, 327, 373, 1014
bloków, 1104
plików, 1104, 1106

C

C2DM, Cloud To Device Messsaging, 805
CA, Certification Authority, 623
cele
algorytmów szeregowania, 174
Linuksa, 725
CFI, Control Flow Integrity, 704
CFS, Completely Fair Scheduler, 751
chmura, 477
jako usługa, 500
obliczeniowa, 500
chwila, jiffy, 749
ciąg rozkazów NOP, 642
ciągła alokacja, 297
cienkie klienty, 423
CLI, Clear Interrupts, 487
CLR, Common Language Runtime, 862
CMS, Conversational Monitor System, 94
cofnięcie operacji, 456
COM, Component Object Model, 873, 903
Common Criteria, 887
CRT, Cathode Ray Tube, 352
CS, Connected Standby, 962
cykl życia procesu, 843
czas wiązania nazw, 999
czcionki, 420
czyszczenie, 248
czytanie bloków zawczasu, 330

`D

DAC, Discretionary Access Control, 612
DACL, Discretionary ACL, 964
Dalvik, 814
DCI, Digital Cinema Initiatives, 1075
DCT, 1079
debugowanie procesu, 927
deduplikacja, 494
definiowanie typu obiektów, 896
defragmentacja dysków, 332
DEP, Data Execution Prevention, 644
deskryptor segmentu kodu, 264
deskryptory plików, 290
DFSS, Dynamic Fair-Share Scheduling, 924
DLL, Dynamic Link Libraries, 89, 246, 860, 902
długość palców, 637
DMA, Direct Memory Access, 58, 356
DMI, Direct Media Interface, 60
domeny
ochrony, 602, 603
urządzeń, 497
DOS, Disk Operating System, 42
dostawcy zawartości, 832
dostęp
bezpośredni do pamięci, 355
do danych, 838, 1024
do dysku, 390
do pamięci, 53
zdalny bezpośredni, 554
do plików, 286
do wspólnej pamięci, 142
do zasobów, 602
dostępność, 596
dowiązanie
do usługi, 831
symboliczne, 900
DPC, Deferred Procedure Call, 881
DRM, Digital Rights Management, 44, 877, 969
drzewo katalogów, 294
duże projekty programistyczne, 100
DV, Digital Video, 1081
DVD, Digital Versatile Disk, 1068
dylemat przestrzeń-czas, 1011
dynamiczna relokacja, 206
dysk twardy, 54, 75, 379, 428
SATA, 32, 56
SSD, 55, 69
SSF, 387

dyski
AV, 393
magnetyczne, 379
dyskietka, 380
działanie wirusa, 663

E

EDF, Earliest Deadline First, 1089
efekt
jitter, 1095
lokalności, 1015
EFS, Encryption File System, 960
egzojądra, 97, 993
ekran logowania, 657
fałszywy, 658
ekrany dotykowe, 421
eksploit, 594, 639, 645
ESX Server, 515
etapy projektowania oprogramowania, 1021
Ethernet, 570
klasyczny, 571
przełączany, 571

F

fałszywe współdzielenie, 563
FAT, File Allocation Table, 300
FIFO, First-In, First-Out, 229
filmy, 1067
firewalle, 685
firma VMware, 503
flaga przerwań, 488
flagi mapy bitowej, 747
formatowanie dysków, 384
FPGA, Field-Programmable Gate Arrays, 555
framework
KMDF, 945
UMDF, 945
funkcja
gets, 641
printf, 649
Rectangle, 419
skrótu
SHA-1, 622
strcmp, 657
XOpenDisplay, 411
funkcje
atomowe, 156
jednokierunkowe, 622
kryptograficzne, 609
skrótu, 622
sterujące VCR, 1092
Win32 API, 966
futeksy, 155

G

gałęzie rejestru, 873
generowanie
przerwania, 58
wyjścia, 407
geometria dysku, 381
GID, Group ID, 800
główny rekord
rozruchowy, MBR, 387
startowy, MBR, 668
gniazda, sockets, 771, 913
GNOME, 727
GPL, GNU Public License, 724
GPT, GUID Partition Table, 388
graf, 566
graficzny interfejs użytkownika, GUI, 29, 43, 412, 776
GUI, Graphical User Interface, 29, 43, 412, 776

H

HAL, Hardware Abstraction Layer, 876
HAL Development Kit, 878
hasła, 628
jednorazowe, 631
heterogeniczne procesory wielordzeniowe, 533
hierarchia
dziedziczenia, 822
katalogów, 579
pamięci, 52, 201
procesów Androida, 115, 810
hierarchiczne systemy katalogów, 292
hipernadzorca, hypervisor, 96, 485, 876
typu 1, 483
typu 2, 483
Xen, 501
hipersześcian czterowymiarowy, 549
historia systemu
Android, 805
Linux, 716
UNIX, 716
Windows, 855

I

IAAS, Infrastructure as a Service, 500
IAT, Import Address Table, 902
IDE, Integrated Drive Electronics, 379

identyfikator
nonce, 625
PID, 79
SID, 964
UID, 67, 837
IDS, Intrusion Detection System, 687, 695
ilustracja przekosu cylindrów, 386
implementacja
bezpieczeństwa, 967
dół-góra, 1001
góra-dół, 1001
hybrydowa, 135
katalogów, 302
menedżera obiektów, 891
plików, 297
procesów, 117, 916
segmentacji, 259
systemu
NFS, 797
plików, 296
plików Linuksa, 785
plików NTFS, 950
wątków, 916
wątków w jądrze, 134
wątków w przestrzeni użytkownika, 131
wejścia-wyjścia, 773, 944
zarządzania pamięcią, 760, 929
indeks, 900
infekcja, 677
informacje o rozwiązaniu, 506
infrastruktura jako usługa, 500
instrukcja, 252
TSL, 147, 538
integralność kodu, 596, 970
Intel x86, 263
interakcje z dostawcą zawartości, 833
interfejs
API, 488, 820
Binder, 822
obiektu Binder, 820
pamięci wirtualnej, 249
programowania aplikacji, API, 865
sieciowy, 554
sterownika, 432
sterownik-jądro, 774
sterowników urządzenia, 372
Win32 API, 85
wywołań systemowych, 989
interfejsy
mechanizmu Binder, 822
sieciowe, 551
systemu Linux, 726
użytkowników, 402
Internet, 572
interpretacja, 700
intruz, 598
inwersja priorytetów, 923
iOS, 47
IP, Internet Protocol, 772
IRP, I/O Request Packets, 899, 943
ISA, Instruction-Set Architectures, 45, 59
ISR, Interrupt Service Routines, 881
istotność procesu, 845
ITO, Indium Tin Oxide, 422
i-węzły, 301, 331, 337, 790
izolowanie, 698
kodu mobilnego, 697
mechanizmu od strategii, 996

J

jądro
Linuksa, 733
systemu Windows, 876
JBD, Journaling Block Device, 793
jednostka MMU, 217, 496
jednostki miar, 104
język
AIDL, 822
C, 98
JIT, Just-In-Time, 814
JPEG, 1078
JVM, Java Virtual Machine, 97, 701

K

kanarki, 642
karta
$FORTRAN, 36
chipowa, 633
inteligentna, 635
z paskiem magnetycznym, 633
z zakodowaną wartością, 633
karty elektroniczne, 65
katalog, 291
\??, 897
\Arcname, 897
\BaseNamedObjects, 897
\Device, 897
\DosDevices, 897
\Driver, 897
\FileSystem, 897
\KnownDLLs, 897
\NLS, 897
\ObjectTypes, 897

katalog
\Security, 897
\Windows, 897
obiektów, 900
katalogi
hierarchiczne systemy, 292
implementacja, 302
jednopoziomowe systemy, 291
kategorie ważności procesów, 844
KDE, 727
keylogger, 659
klasyczny model wątków, 125
klawiatura, 402
klienty, 409
klipy wideo, 1067
klucz
publiczny, 621
symetryczny, 621
tajny, 620, 621
kod
ECC, 394
jednowątkowy, 138
kompresji, 689
mobilny, 697
PIN, 633
powłoki, shellcode, 641
z zakleszczeniem, 446
kodowanie
audio, 1076
JPEG, 1078
percepcyjne, 1083
wideo, 1073
kolejka, 900
kompilacja warunkowa, 1005
komponenty
procesu, 127
wątku, 127
kompresja
audio, 1083
plików, 958
wideo, 1077
komputer
osobisty, 42, 63
mobilny, 46
podręczny, 63
komunikacja
asynchroniczna, 1002
bezprzewodowa, 430
IPC, 823
między procesami, 141
międzyprocesowa, 913
synchroniczna, 1002
węzła z interfejsem, 554
komunikat rozgłoszeniowy, 831
konie trojańskie, 661
kontrola dostępu, 605, 612
kontroler, 352
kontrolery urządzeń, 351
kontrolowanie dostępu do zasobów, 602
kopia przy zapisie, 927
kopie
map bitowych, 420
woluminów, 941
zapasowe systemu plików, 319
korekcja błędu, 391
koszt wirtualizacji, 487
kraker, 627
kryptografia, 619, 621
z kluczem tajnym, 620
księgowanie, 959
księgujące systemy plików, 308
kwantyzacja próbek, 1076, 1079

L

lampy elektronowe, 35
LAN, Local Area Networks, 570
LFS, Log-structured File System, 306
licencja, 499
GPL, 724
licznik, 400
czasowy, 900
dozorujący, 400
limity dyskowe, quotas, 306, 318
linker, 100
Linux, 722, 725
bezpieczeństwo, 800, 802
implementacja
bezpieczeństwa, 803
procesów, 742
systemu plików, 785
wątków, 742
wejścia-wyjścia, 773
zarządzania pamięcią, 760
interfejsy systemu, 726
katalogi, 778
model wejścia-wyjścia, 775
moduły, 776
obsługa sieci, 771
operacje na plikach, 774
operacje wejścia-wyjścia, 769
powłoka, 728
procesy, 735
programy użytkowe, 731
przydzielanie pamięci, 763

sekwencja procesów, 755
stronicowanie, 766
struktura jądra, 733
synchronizacja, 752
system plików, 777
/proc, 793
ext2, 787
ext4, 792
NFS, 794
szeregowanie, 748
tablice stron, 763
tworzenie procesu, 736
uruchamianie systemu, 753
wątki, 745
wirtualny system plików, 786
wywołania systemowe, 738, 759, 803
wywołania systemu plików, 782
wywołania wejścia-wyjścia, 772
zarządzanie pamięcią, 755
lista
gotowości, standby list, 927
jednokierunkowa, 211, 299
kontroli dostępu, 605, 606
publikacji, 1031
pytań i odpowiedzi, 632
uprawnień, 608
login spoofing, 657
logowanie, 627
LOIC, 597
LRU, Least Recently Used, 231,845, 931
luki w oprogramowaniu, 638

Ł

ładowalne moduły, 777
łańcuch formatujący, 648
łączenie sterowników, 948

M

Mac OS X, 43
MAC, Mandatory Access Control, 612
macierz RAID, 382
macierze ochrony, 604, 610
magazyn stron, 253
magistrala, 58
DMI, 60
ISA, 59
PBA, 59
PCI, 511
PCIe, 59
podwójna, 355
pojedyncza, 355
SBA, 59
SCSI, 60
USB, 60
makroblok, 1082
malware, 639, 661
mapa bitowa, 210, 324, 418, 420
maskowanie czasowe, 1083
master-slave, 535–537
maszyna wirtualna, 93, 485, 498, 508
Javy, 97
MBR, Master Boot Record, 387, 668, 753
mechanizm
aktywacji zarządcy, 135
UMS, 909
zabójcy OOM, 813
menedżer
aktywności, 843
obiektów, 893, 900
wejścia-wyjścia, 899
zasobów, 33
metoda wyzwanie-odpowiedź, 632
MFT, Master File Table, 951
miękki błąd braku strony, 932
migracja, 502
migracje maszyn wirtualnych, 501
mikrojądra, 90, 489
mikrokomputer, 42
minimalizacja ruchu ramienia dysku, 330
miniport, 889
MINIX, 721
MMU, Memory Management Unit, 669
model, 695
Bella-La Paduli, 612
bezpieczeństwa, 613
Biby, 614
fazy działania, 100
klient-serwer, 93
procesów, 110, 841
programowania COM, 903
transferu, 578
WDM, 945
wejścia-wyjścia, 775
modele bezpiecznych systemów, 610
modelowanie
wieloprogramowości, 119
zakleszczeń, 448
Modern Windows, 861
modulacja PCM, 1077
moduł
jądra Binder, 817
TPM, 624, 891
VMM, 512

moduły w Linuksie, 776
monitor, 159, 402
  odwołań, 601, 700
  VMM, 478
most, bridge, 571
MPEG, 1080
MSDK, Microsoft Development Kit, 863
MS-DOS, 43, 856
MULTICS, 260
multimedia, 1067
multimedialne systemy operacyjne, 1067
muteks, 154, 900, 975
muteksy w pakiecie Pthreads, 156
muzyka, 1067
mysza, 402, 406

N

narzędzia dsniff, 597
narzędzie nmap, 597
NAT, network address translation, 685
nazwy
  plików, 281
  ścieżek, 292
NFS, Network File System, 794
  architektura systemu, 794
  implementacja systemu, 797
  protokoły systemu, 795
  struktura warstwy systemu, 797
  wersje systemu, 800
niebieski ekran śmierci, 886
nieprawidłowa strona, invalid page, 925
niskopoziomowe oprogramowanie komunikacyjne, 552
NIST, 500
NOP, 642
NT, New Technology, 858
NUMA, NonUniform Memory Access, 921
numer SID, 964
NVOD, 1094

O

obiekty dyspozytora, 883
obliczanie godziny, 399
obraz
  nieskompresowany, 1013
  skompresowany, 1013
obrona wielostrefowa, 684
obrys znaku, 421
obsługa
  błędów, 391
  błędów braku stron, 250, 255, 930
  przerwań, 57, 367
  sieci, 771
  systemu plików, 289
  wielu wątków, 123, 138
  zagnieżdżonych tablic stron, 493
  zegara, 397, 398
ochrona pliku, 801
odbiorca, receiver, 831
oddzielenie
  mechanizmu szeregowania, 185
  strategii od mechanizmu, 255
odległość Hamminga, 636
odtwarzanie po awarii, 395
odwołanie do pustego wskaźnika, 652
odwracanie priorytetów, 923
odwzorowywanie adresu liniowego, 266
odzyskiwanie pamięci, 494
ogólna warstwa blokowa, 774
ograniczenia zabezpieczeń, 970
okna tekstowe, 407
okno, 414
operacja
down, 151
  operacja RCU, 168
  up, 151
operacje
  na katalogach, 294
  na plikach, 288, 774
  wejścia-wyjścia, 58, 95, 172, 769, 942
    w Windows, 940
opóźnione wywołania procedur, 881
oprogramowanie
  do generowania wyjścia, 407
  do wprowadzania danych, 402
  DRM, 877
  jako usługa, 500
  klawiatury, 403
  komunikacyjne poziomu użytkownika, 555
  myszy, 406
  obsługi zegara, 398
  szpiegujące, 676, 679
  wejścia-wyjścia, 362
optymalizacja, 1010, 1016
ortogonalność, 997
OS X, 45
osiąganie spójności sekwencyjnej, 564
otwarty plik, 900

P

PAAS, Platform as a Service, 500
pakiet
  Pthreads, 129, 156
  SDK, 806
  WDK, 945
pakiety żądań wejścia-wyjścia, 946
pamięci o dużej pojemności, 74, 223
pamięć, 51, 430
  EEPROM, 54
  fizyczna, 935
  flash, 54
  masowa, 393
  podręczna, 328, 939
  podręczna L2, 53
  RAM, 53, 201, 396
  ROM, 53
  wirtualna, 76, 213, 262, 928
paradygmaty
  danych, 988
  interfejsu użytkownika, 986
  multimedialnych systemów wykonywania, 987
  plików, 1091
parametry dyskowe, 380
parawirtualizacja, 488, 489
partycjonowanie pamięci, 534
paski, strips, 382
paskowanie, striping, 382
patchguard, 971
PBA, parallel bus architecture, 59
PCR, Platform Configuration Register, 625
PDA, Personal Digital Assistant, 46, 63
PDE, Page Directory Entry, 933
PDP-11 UNIX, 717
PEB, Process Environment Block, 905, 917
perspektywy, views, 939
PFF, Page Fault Frequency, 240
PFN database, 935
piaskownice aplikacji, 835
PID, Process Identifier, 736, 742
pierścień, 549
PKI, Public Key Infrastructure, 623
platforma
  jako usługa, 500
  x86, 505
plik
  AndroidManifest.xml, 824
  lib.dll, 930
  ntoskrnl.exe, 891
pliki, 68, 281
  .dll, 89
  .wmf, 419
  atrybuty, 286
  binarne, 284
  deskryptory, 290
  implementacja, 297
  multimedialne, 1072, 1095
  nagłówkowe, 99
  nazwy, 281
  odwzorowane w pamięci, 248, 758
  program do kopiowania, 289
  rozszerzenia, 282
  stron, 926
  struktura, 283
  typu peer-to-peer, 677
  typy, 284
  współdzielone, 304
  wykonywalne, 285
  z pojedynczą blokadą, 782
PLT, Procedure Linkage Table, 645
plug and play, 60, 859
płyta macierzysta, motherboard, 61
pobieranie plików bez wiedzy, 677
podmiot, 605
podpisy cyfrowe, 622
podpisywanie kodu, 693
podsystemy, 901
podsystemy NT, 865
podział czasu, 39, 543
pola, 1074
  pakietu żądań wejścia-wyjścia, 947
  struktury i-węzła, 790
polecenie lseek, 784
port ALPC, 900
pośrednictwo, 1006
potok trójfazowy, 49
potoki, pipes, 45, 737
poufność, 596
powłoka, 71, 728
  uproszczona, 81
powstawanie
  przerwań, 359
  zakleszczeń, 447
poziomy RAID, 383
priorytety
  systemu Windows, 921
  wątków, 922
problem
  czytelników i pisarzy, 190
  pięciu filozofów, 187
  producent-konsument, 150, 152, 165
  relokacji, 204

problemy
do rozwiązania w programach, 432
implementacyjne RPC, 560
po stronie systemu operacyjnego, 426
projektowe systemów stronicowania, 239
sprzętowe, 425
procedura, 896
pośrednicząca klienta, 559
pośrednicząca serwera, 559
zliczająca bity, 1011
procedury obsługi przerwań, 367
proces, 845, 900
wielowątkowy, 126
procesor, 45, 47, 428, 534
superskalarny, 49
SVM, 481
procesory wielordzeniowe, 498
procesy, 65, 109, 905, 1032
bariery, 167
blokowanie zmiennych, 145
hierarchie, 115
implementacja, 117
komponenty, 127
komunikacja, 141
kończenie działania, 114
model, 110
monitory, 159
muteksy, 154
prace badawcze, 191
problemy komunikacji, 187
przekazywanie komunikatów, 164
regiony krytyczne, 143
semafory, 151
stany, 115
ścisła naprzemienność, 145
tworzenie, 112
unikanie blokad, 168
w systemie Linux, 735
wyścig, 141
wzajemne wykluczanie, 144
zorientowane na wejście-wyjście, 171
profil, 900
program
CMS, 94
gzip, 778
Hyper-V, 891
WinResume.exe, 891
zainfekowany, 689
programowane wejście-wyjście, 364
programowanie
aplikacji Win32, 869
ROP, 645
systemu Windows, 862
z wykorzystaniem wielu rdzeni, 534
zorientowane na powrót, ROP, 644
programowe zarządzanie buforem, 221
programy
agentów, 697
licencjonowane, 499
użytkowe Linuksa, 731
projekt systemu operacyjnego, 979
projektowanie systemu, 980 operacyjnego
cele, 980
etapy projektowania, 1021
implementacja, 992
interfejs, 983
nazewnictwo, 998
trendy, 1021
trudności, 981
wydajność, 1009
zarządzanie projektem, 1017
protokoły
sieciowe, 573, 574
systemu NFS, 795
protokół TCP/IP, 410
próbkowanie fali sinusoidalnej, 1076
przeciwdziałanie zakleszczeniom, 462
przeglądy kodu, code reviews, 657
przekazywanie komunikatów, 164, 165
przekos
cylindrów, 386
głowic, head skew, 385
przełączanie, 541
pakietów, 550
światów, 487
przełącznik
krzyżowy, 525, 526
pojedynczy, 549
przenośny UNIX, 718
przepełnienie
bufora, 640, 648
liczb całkowitych, 653
przeplatanie, 1074
wideo, 1096
przeplot, 387
przerwania, 57, 358, 367
nieprecyzyjne, 360
precyzyjne, 360
przestrzenie adresowe, 67
przestrzeń
adresowa, 205, 243, 930
wirtualna, 215
nazw obiektów, 895
procesów, 925

przetwarzanie w chmurze, 477, 1022, 1035
przydzielanie
  adresów wirtualnych, 925
  dedykowanych urządzeń, 376
  pamięci, 763
przynęty, honeypots, 697
przyspieszenie stronicowania, 219
PTE, Page Table Entries, 933
publikacje, 1031
  wprowadzające i ogólne, 1032
publikuj-subskrybuj, 585
pule wątków, 907
pułapka, breakpoint, 927
punkty
  kontrolne, 502
  przyłączania, 950

R

RAID, 381
RAM, Random Access Memory, 53, 201
ramka, 1073
  wideo, 1082
raportowanie błędów, 376
RDMA, Remote Direct Memory Access, 554
RDP, Remote Desktop Protocol, 924
rdzeń, 51
regiony krytyczne, 143
reguły ochrony, 703
rejestr
  bazowy, 57
  systemu Windows, 872
  bazy i limitu, 206
rekord tablicy MFT, 955, 957
relokacja, 204
remapowanie adresów pamięci, 55
replikacja, 562
reprezentacja uprawnienia, 609
RGB, Red, Green, Blue, 1074
RMS, Rate Monotonic Scheduling, 1088
robaki, 673
rodzaje
  komunikatów, 411
  rootkitów, 680
  systemów, 569
ROM, Read Only Memory, 53, 202
rootkit, 680, 971
  firmy Sony, 683
ROP, Return-Oriented Programming, 645, 970
rozgałęźnik-wampir, 571
rozmiar
  bloku, 313, 377
  maksymalny partycji, 335
  strony, 242
rozmieszczenie plików multimedialnych, 1100, 1095
rozpowszechnianie
  oprogramowania szpiegującego, 677
  wirusów, 672
rozpoznawanie tęczówki, 636
rozproszona pamięć współdzielona, 249, 560
rozszerzenia
  Joliet, 342
  plików, 282
  Rock Ridge, 341
rozszerzona maszyna, 32
rozwiązania siłowe, 1008
rozwiązanie Petersona, 146
równoważenie obciążenia, 565
RPC, Remote Procedure Calls, 560, 888, 913, 559, 862
ruch ramienia dysku, 330

S

SAAS, Software as a Service, 500
SACL, System Access Control List, 965
SAM, Security Access Manager, 873
sandboxing, 477
SATA, Serial ATA, 32, 56, 379
SBA, shared bus architecture, 59
schemat
  list, 936
  stosów urządzeń, 890
SCSI, Small Computer System Interface, 60
SDK, Software Development Kit, 806
segmentacja, 256, 259
  klasyczna, 259
  ze stronicowaniem, 260, 263
sekcja, 900
sektor dysku, 385
sekwencja ucieczki, 408
selektor x86, 264
semafor, 151, 446, 900
semantyka współdzielenia plików, 580
serwer, 125, 409
siatka, 549
sieciowy system plików, NFS, 794
sieć
  botnet, 597
  Ethernet, 570
  Internet, 572
  LAN, 570
  przełącznikowa omega, 527
  WAN, 570
  WWW, 577
silniki mutacji, 690

skanery antywirusowe, 687
skrypciarze, 599
skrytki pocztowe, mailslots, 913
słaby punkt, 594
spooler, 141, 462
sposoby konstrukcji serwera, 125
spójność systemu plików, 324
sprawca, 605
sprawdzanie błędów, 1008
sprzęt
  komputerowy, 47
  obsługi zegara, 397
  sieciowy, 570
  wielokomputerów, 548
  wieloprocesorowy, 524
spyware, 676
SSD, Solid State Disk, 55, 69
SSF, Shortest Seek First, 389
stabilny
  odczyt, 395
  zapis, 394
standard
  ISO 9660, 341
  JPEG, 1078
  MPEG, 1080
  POSIX, 738
  UNIX, 720
stany
  autoryzowane, 611
  bezpieczne, 458
  nieautoryzowane, 611
  niebezpieczne, 458
  procesów, 115, 845
  stron, 769
  STB, Set Top Box, 1069
  steganografia, 617
  sterownik
  dysku, 32
  jądra win32k.sys, 888
  urządzenia, 900
  VMM, 512
  Win32k.sys, 891
  zarządzania energią, 432
sterowniki
  filtrów system plików, 889
  urządzeń, 368, 889, 945
stos, 128
  urządzenia, device stack, 889, 946
strategia czyszczenia, 248
strategie
  alokacji pamięci, 239
  organizacji plików, 1096
  stronicowania, 926
strona, 242
  wirtualna, 926
stronicowanie, 214, 250, 259
  w Linuksie, 766
strony współdzielone, 244
struktura
  danych, 895
  danych runqueue, 750
  dziennika, 306
  napędu dyskowego, 54
  obiektu, 893
  pliku, 283
  systemu, 992
  systemu monolitycznego, 89
  systemu operacyjnego, 88
    MINIX 3, 92
    THE, 90
    VM/370, 94
    Windows, 875
  systemu plików, 951
  systemu x86, 59
  tokenu dostępu, 964
  wpisu, 218
  zespołu, 1018
struktury
  dynamiczne, 1000
  statyczne, 1000
superużytkownik, 601
SVM, Secure Virtual Machine, 481
sygnał wakeup, 151
Symbian, 47
symetryczne systemy wieloprocesorowe, 536
symulacja algorytmu LRU, 231, 232
synchronizacja, 914
  w Linuksie, 752
system
  bez abstrakcji pamięci, 203
  CTSS, 181
  DV, 1081
  MULTICS, 260
  opakowań WDF, 945
  operacyjny, 29, 31
    Android, 804
    czwarta generacja, 42
    druga generacja, 35
    historia, 34
    hosta, 511
    jako menedżer zasobów, 33
    jako rozszerzona maszyna, 32
    Linux, 722, 725
    piąta generacja, 46
    pierwsza generacja, 35
    trzecia generacja, 37

UNIX, 716
X Window, 409
plików, 69, 84, 279, 780, 1033
/proc, 793
ext2, 787
ext4, 792
FAT-16, 949
FAT-32, 949
ISO 9660, 338
kopie zapasowe, 319
Linuksa, 777
NT, 949
MS-DOS, 333
NTFS, 949
spójność, 324
wydajność, 327
system plików V7, 336
rozproszony, 523, 569
VMI, 490
VMware Workstation, 513, 515
wielokomputerowy, 569
wieloprocesorowy, 569
wieloprocesorowy SMP, 536
systemy
jednopoziomowe katalogów, 291
klient-serwer z mikrojądrem, 994
operacyjne
badania, 101
czasu rzeczywistego, 64
firmy DEC, 858
kart elektronicznych, 65
komputerów mainframe, 62
komputerów osobistych, 63
komputerów podręcznych, 63
monolityczne, 88
serwerów, 62
struktura, 88
warstwowe, 89
wbudowane, 63
węzłów sensorowych, 64
wieloprocesorowe, 62, 534
plików
księgujące, 308
na płytach CD-ROM, 338
o strukturze dziennika, 306
wirtualne, 310
rozproszone, 568
rozszerzalne, 995
stronicowania, 239
wbudowane, 1025
wielokomputerowe
szeregowanie, 565
systemy wieloprocesorowe, 1036
symetryczne, 536
synchronizacja, 538
szeregowanie, 542
wieloprocesorowe, 521, 523
NUMA, 528
typu master-slave, 535
UMA
z architekturą magistrali, 524
z przełącznikami krzyżowymi, 525
wielowarstwowe, 992
wsadowe, 35
szeregowanie, 169, 172, 919
bazujące na priorytetach, 179
cykliczne, 178, 748
EDF, 1089
gwarantowane, 182
loteryjne, 182
monotoniczne w częstotliwości, 1088
operacji dyskowych
dynamiczne, 1108
statyczne, 1106
operacji dyskowych, 1106
procesów homogenicznych, 1086
procesów multimedialnych, 1086
sprawiedliwe, 183
systemów wielokomputerowych, 565
w czasie rzeczywistym, 1086
w Linuksie, 748
w systemach czasu rzeczywistego, 184
w systemach interaktywnych, 178
w systemach wsadowych, 176
w trybie użytkownika, 907
wątków, 185
zespołów, 546
sześcian, 549
szkielet aplikacji, 411, 415
szybkość
czytania danych, 315
przesyłania danych, 351, 1071
szyfrowanie plików, 960

Ś

ścieżki, 292
ścisła naprzemienność, 145
śledzenie
wolnych bloków, 316
wykorzystania zasobów, 319
środki obrony, 684
środowisko bezpieczeństwa, 595

T

tabela FAT, 300
tabele stron, 217, 223, 493
  odwrócone, 225
  wielopoziomowe, 223
tablica
  GPT, 388
  plików, 952
  PLT, 645
  stron, 218, 931
  uchwytów, 894, 895
takt zegara, 397
TCP, Transmission Control Protocol, 575, 772
TEB, Thread Environment Block, 905, 917
technika
  ASLR, 646
  plug and play, 60
techniki
  antyantywirusowe, 687
  antywirusowe, 687
  biometryczne, 635
  skutecznej wirtualizacji, 484
technologia
  RAID, 381
  wewnętrznych połączeń, 549
teoria grafów, 566
test POST, 753
THE, 90
TLB, Translation Lookaside Buffer, 929
TLS, Thread Local Storage, 905
tłumacz binarny, 485
TOCTOU, 655
token, 964
  dostępu, 900, 965
topologie wewnętrznych połączeń, 549
torus podwójny, 549
TPM, Trusted Platform Module, 891
trafienie pamięci, 52
trajektorie zasobów, 457
transfer
  obiektów, 820
  poprzez DMA, 356
transformacja falkowa Gabora, 636
transmisja równoległa, 59
tranzystory, 35
tryb
  beztaktowy, 749
  jądra, 29
  nadzorcy, 29
  niekanoniczny, 403
  offline, 36
  piaskownicy, 477
  użytkownika, 29
tryby ochrony pliku, 801
tworzenie
  dowiązania, 779
  procesów, 112, 736, 816
tylne drzwi, back door, 656
typ master-slave, 535
typy
  obiektów, 900
  plików, 284
  sieci, 570
  usług sieciowych, 575
  wieloprocesorowych systemów operacyjnych, 534

U

UAC, User Account Control, 968
uchwyty, 894
UDF, Universal Disk Format, 298
udostępnianie zdjęcia, 829, 840
UEFI, Unified Extensible Firmware Interface, 890
UID, User ID, 67, 800
układ
  czterordzeniowy, 51
  systemu plików, 296
układy
  scalone, 37
  ultrawielordzeniowe, 532
  wielordzeniowe, 50, 530, 1022
  wielowątkowe, 50
ukryty kanał komunikacyjny, 614, 616
ukrywanie sprzętu, 1004
UMS, User-Mode Scheduling, 908
UNICS, 716
unikanie
  blokad, 168
  kanarków, 642
  wirusów, 692
UNIX, 45, 716
uprawnienia, 607
  superużytkownika, 601
uruchamianie
  aktywności, 826
  aplikacji, 827
  komputera, 61
  procesów, 842
  systemu Linux, 753
  systemu Windows, 890
  usługi, 830

urządzenia, 900
dedykowane, 376
wejścia-wyjścia, 55, 350, 352
wirtualne, 498
urząd certyfikacji, CA, 623
USB, Universal Serial Bus, 60
USENET, 588
usługa, 830
ADSL, 773
bezpołączeniowa, 573
datagramów z potwierdzeniami, 574
połączeniowa, 573
usługi
komunikacyjne, 555
sieciowe, 573
trybu użytkownika, 901
usuwanie
procesu, 828
zakleszczeń, 455
poprzez wywłaszczanie, 455
poprzez zabijanie procesów, 456
przez cofnięcie operacji, 456
uwierzytelnianie, 626
metodą wyzwanie-odpowiedź, 632
z wykorzystaniem obiektu fizycznego, 633
z wykorzystaniem technik biometrycznych, 635
uwięzienia, 468
użycie karty inteligentnej, 635

V

VAD, Virtual Address Descriptor, 929
VFS, Virtual File System, 310, 733, 778
VMI, Virtual Machine Interface, 490
VMM, Virtual Machine Monitor, 478
VMWARE, 502
VMware Workstation, 504
VMX, 512

W

WAN, Wide Area Networks, 570
war dialer, 628
warstwa
abstrakcji sprzętowej, 877
jądra, 880
middleware bazująca na
dokumentach, 576
koordynacji, 584
obiektach, 582
systemie plików, 578
wykonawcza, 884
warstwy programowania, 863
wątek, 120, 909, 900, 1032
sprzątacza, 308
wyskakujący, 558
wątki
implementacja w jądrze, 134
implementacja w przestrzeni użytkownika, 131
jądra, 995
komponenty, 127
konflikty, 138
model klasyczny, 125
pop-up, 137
POSIX, 129
prywatne zmienne globalne, 139
stron zerowych, 922
w systemie Linux, 745
wykorzystanie, 121
WDF, Windows Driver Foundation, 945
WDK, Windows Driver Kit, 945
WDM, Windows Driver Model, 945
wejście-wyjście, 71, 349, 1034
jednostki MMU, 496
oprogramowanie, 362
niezależne od urządzeń, 372
w przestrzeni użytkownika, 377
programowane, 364
sterowane przerwaniami, 365
system wieloprocesorowy ze współdzieloną pamięcią, 523
urządzenia, 350, 352
warstwy oprogramowania, 367
warstwy systemu, 378
wirtualizacja, 495
z wykorzystaniem DMA, 366
wektor przerwań, 57
wersje systemu NFS, 800
weryfikatory
integralności, 691
zachowań, 691
wewnętrzne połączenia, 549
węzły sensorowe, 64
wideo
na żądanie, 1069
niemal na życzenie, 1094
RAM, 413
wielkie przestrzenie adresowe, 1023
wielobieżność, reentrancy, 1007
wielokomputer, 523, 548
z przekazywaniem komunikatów, 523
wielokrotne
kolejki, 181
użycie kodu, 1007
wykorzystywanie kodu, 644

wieloprogramowość, 37, 119
Win32 API, 85, 859
Windows
  alokacja przestrzeni dyskowej, 954
  asynchroniczne wywołania procedur, 882
  bezpieczeństwo, 963
  biblioteki DLL, 901
  błędy braku stron, 930
  implementacja
    bezpieczeństwa, 967
    procesów, 916
    systemu plików, 950
    systemu wejścia-wyjścia, 944
    wątków, 916
    zarządzania pamięcią, 929
  katalogi, 897
  kompresja plików, 958
  komunikacja międzyprocesowa, 913
  księgowanie, 959
  łączenie sterowników, 948
  menedżer obiektów, 891
  obiekty dyspozytora, 883
  ograniczenia zabezpieczeń, 970
  operacje wejścia-wyjścia, 940
  opóźnione wywołania procedur, 881
  organizacja trybu jądra, 876
  pamięć podręczna, 939
  pliki stron, 926
  podsystemy, 901
  priorytety systemu, 921
  procedury, 896
  procesy, 904
  przestrzeń nazw obiektów, 895
  rejestr systemu, 872
  schemat stosów urządzeń, 890
  sterowniki urządzeń, 889, 945
  stosy urządzeń, 947
  struktura systemu, 875
  synchronizacja, 914
  system plików, 949
  szeregowanie, 907, 919
  szyfrowanie plików, 960
  uchwyty, 894
  uruchamianie systemu, 890
  usługi trybu użytkownika, 901
  warstwa abstrakcji sprzętowej, 877
  warstwa jądra, 880
  warstwa wykonawcza, 884
  wątki, 904, 909
  włókna, 906
  wywołania API, 911, 965
  zadania, 906
  zarządzanie
    energią, 960
    pamięcią, 924
    pamięcią wirtualną, 928
Windows 7, 44
Windows 8, 855
Windows Me, 44
Windows na bazie MS-DOS-a, 857
Windows na bazie NT, 857
Windows NT, 44
Windows Update, 971
Windows Vista, 44, 860
Windows XP, 44
wirtualizacja, 477, 489, 1022, 1035
  architektury x86, 507
  na platformie x86, 505
  pamięci, 491
  SR-IOV, 497
  wejścia-wyjścia, 495
wirtualna
  maszyna Javy, 701
  platforma sprzętowa, 510
  przestrzeń adresowa, 215, 756, 765, 925
wirtualne systemy plików, 310
wirtualny
  sprzęt, 511
  system plików, 786
wirus, 594, 663
wirus skompresowany, 689
wirusy
  polimorficzne, 689
  rezydujące w pamięci, 667
  sektora startowego, 668
  towarzyszące, 664
  w kodzie źródłowym, 671
  w makrach, 670
  w programach wykonywalnych, 665
  w sterownikach urządzeń, 670
wiszące wskaźniki, dangling pointers, 651
włókna, fibers, 906
WNF, Windows Notification Facility, 888
WNS, Windows Notification Service, 962
WOW, Windows-on-Windows, 870
wpis
  PDE, 933
  PTE, 933
wprowadzanie danych, 402
wskazówki, hints, 1015
współdzielenie
  plików, 580, 759
  przestrzeni, 545
współdzielona biblioteka, 247
wstrzykiwanie kodu, 654
wtrącanie do więzienia, 695

wydajność systemu
operacyjnego, 1009
plików, 327
wyjście VM, 492
wykonywanie danych, 643
wykorzystanie luk w oprogramowaniu, 638
wykrywanie
rootkitów, 681
włamań, 695
zakleszczeń, 451, 453
wyłączanie przerwań, 144
wymagania dotyczące wirtualizacji, 480
wymiana pamięci, 207
wysyłanie komunikatu rozgłoszeniowego, 832
wyszukiwanie pliku, 338
wyścig, 141
wyświetlacz, 427
wywłaszczanie, 444, 455
wywołania
API, 911, 965
biblioteczne, 86
blokujące, 556
funkcji Win32, 869
interfejsu NT API, 867, 943
interfejsu Win32 API, 874, 916
nieblokujące, 556
pakietu Pthreads, 157, 158
procedur, 558
rdzennego NT API, 869
RPC, 913
systemowe, 76
do zarządzania katalogami, 83
do zarządzania plikami, 82
do zarządzania procesami, 79
Linuksa, 738, 759
różne, 85
wejścia-wyjścia, 772
systemu plików, 782
wywołanie
chmod, 85
read, 124
sleep, 149
stat, 784
time, 85
wakeup, 149
Win32 API, 869
wyzwanie-odpowiedź, challenge-response, 632
wzajemne wykluczanie, 144, 462
wzorzec skanowania, 1074

X

X Window System, 45

Z

zabezpieczenia, 71
aplikacji, 836
sprzętowe, 74
zabieranie cykli, cycle stealing, 357
zabijanie procesów, 456
zabójca OOM, 813
zadania, jobs, 906
zagłodzenia, 469
zagrożenia, 596
zakleszczenie, deadlock, 443, 1035
komunikacyjne, 465
modelowanie, 448
przeciwdziałanie, 462
unikanie, 449, 457
usuwanie, 455
warunki powstawania, 447
wykrywanie, 451, 453
wywłaszczanie, 455
zalecenia projektowe
kompletność, 984
paradygmaty, 986
prostota, 984
wydajność, 985
zależności pomiędzy procesami, 844
zamiary, intents, 834
zapętlanie, 541
zapobieganie wykonywaniu danych, 643
zarządca, 135
zarządzanie
bateriami, 431
buforem TLB, 221
energią, 424, 960
katalogami, 80, 83
miejscem na dysku, 313
obciążeniem, 241
pamięcią, 118, 210, 211, 267, 1033
pamięcią, 201, 924
pamięcią fizyczną, 760, 935
pamięcią W Linuksie, 755
pamięcią wirtualną, 928
plikami, 80, 82, 118
procesami, 79, 80, 118, 911, 916
procesorem, 909
wątkami, 911, 916

zarządzanie
włóknami, 911, 916
zadaniami, 911
zarządzanie projektem, 1017
systemem plików, 80
systemem plików, 313
zarządzanie temperaturą, 431
zarządzanie wolną pamięcią, 210
zarządzanie zasobami, 909
zasada
Kerckhoffsa, 620
pola, 603
zasady projektowe, 1040
zasoby, 444
zastępowanie stron, 226, 227
zaufana baza obliczeniowa, 601
zbiór roboczy, working set, 233, 934
zdalne wywołanie procedury, RPC, 558
zdarzenie, 900
zdobywanie zasobu, 445
zegary, 396
programowe, 400
złośliwe oprogramowanie, 639, 658
zmienne globalne, 139
znak
CR, 405
EOF, 405
ERASE, 405
INTR, 405
KILL, 405
LNEXT, 405
NL, 405
QUIT, 405
START, 405
STOP, 405
zwalnianie dedykowanych urządzeń, 376

Ż

żądanie uprawnień, 838
żniwiarze gadżetów, gadget harvesters, 645

IDEALNA LEKTURA DLA KAŻDEGO PASJONATA INFORMATYKI!

System operacyjny to złożone oprogramowanie, umożliwiające działanie programów na Twoim komputerze. Właśnie trzymasz w rękach światowy bestseller poświęcony współczesnym systemom operacyjnym. Jego autor Andrew S. Tanenbaum to uznany specjalista, który przez wiele lat zajmował się projektowaniem systemów operacyjnych. Dzięki temu każda strona książki wypełniona jest po brzegi szczegółową wiedzą, podaną w przystępny sposób.

Sięgnij po tę książkę, a poznasz dogłębnie zagadnienia związane z procesami, wątkami, zarządzaniem pamięcią, systemami plików oraz operacjami wejścia-wyjścia. Ponadto zdobędziesz informacje o projektowaniu interfejsu użytkownika, multimediach oraz problemach z zakleszczaniem czy bezpieczeństwem. W dobie urządzeń mobilnych niezwykle istotne są także tematy związane z zarządzaniem energią — ich znajomość zapewni dłuższe działanie Twoich urządzeń. Kolejne wydanie tej książki zostało zaktualizowane o wiedzę na temat wirtualizacji, chmur obliczeniowych, systemów Android oraz Windows 8. Jeżeli szukasz kompleksowego podręcznika, dotykającego każdego obszaru związanego z systemami operacyjnymi — to właśnie go znalazłeś!

Dzięki tej książce:

- dowiesz się, czym są proces i wątek
- poznasz zalety i wady dostępnych systemów operacyjnych
- zanalizujesz organizację plików na Twoim dysku
- poznasz fachowe słownictwo
- zaznajomisz się z problemem zakleszczania
- poznasz techniki szeregowania zadań
- przekonasz się, jak są zorganizowane chmury obliczeniowe

Andrew S. Tanenbaum — profesor informatyki, zdobywca grantu European Research Council Advanced na badania nad niezawodnością w systemach komputerowych. Prowadził badania związane z kompilatorami, systemami operacyjnymi oraz sieciami komputerowymi. Jest cenionym publicystą.

Herbert Bos — profesor na Uniwersytecie Amsterdamskim. Naukowo zajmuje się takimi zagadnieniami jak bezpieczeństwo oraz wsparcie sieci komputerowych w systemach operacyjnych. Jest zaangażowany w projekty MINIX 3 oraz Rosetta.

Helion
36710 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci
Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl
ISBN 978-83-283-1422-1
9 788328 314221
Informatyka w najlepszym wydaniu
cena: 129,00 zł

PEARSON
ALWAYS LEARNING