helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

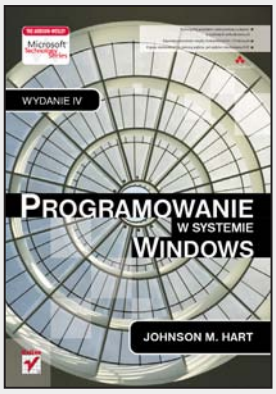
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Programowanie w systemie Windows. Wydanie IV



Autor: Johnson M. Hart
Tłumaczenie: Tomasz Walczak
ISBN: 978-83-246-2780-6
Tytuł oryginału: Windows System Programming (4th Edition)
Format: 168x237, stron: 752

- Wykorzystaj paralelizm i maksymalizuj wydajność w systemach wielordzeniowych
- Zapewnij przenośność między środowiskami 64- i 32-bitowymi
- Popraw skalowalność za pomocą wątków, pul wątków i mechanizmu IOCP

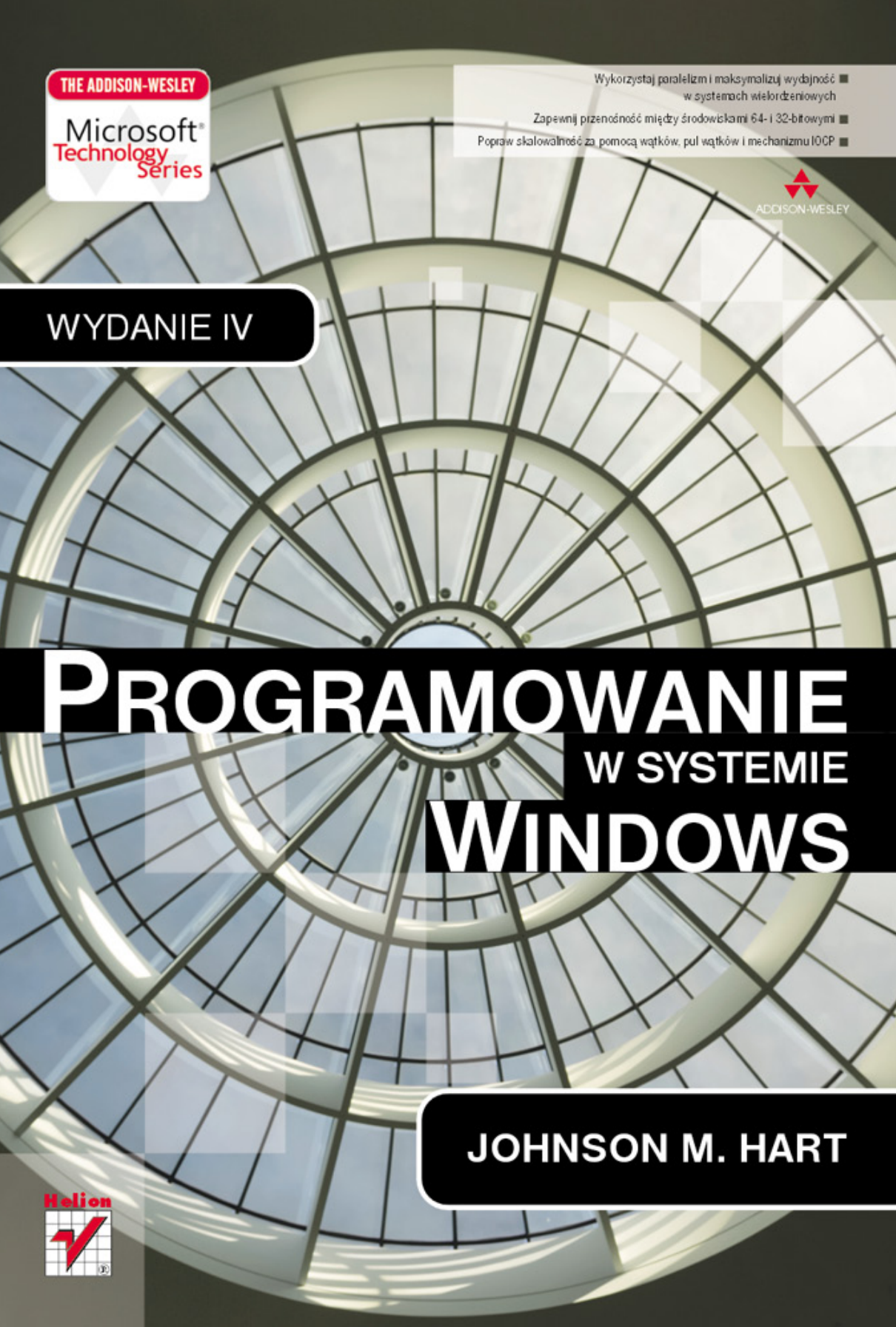
Wybierając system Windows jako docelową platformę rozwijanych aplikacji, programiści na całym świecie sugerują się najczęściej jego dużą funkcjonalnością i wymogami biznesowymi. System ten jest bowiem zgodny z wieloma kluczowymi standardami. Obsługuje między innymi biblioteki standardowe języków C i C+ oraz uwzględnia wiele otwartych standardów współdziałania. Dlatego gniazda systemu Windows są standardowym interfejsem programowania rozwiązań sieciowych z dostępem do TCP/IP i innych protokołów sieciowych. W dodatku każda nowa wersja tego systemu jest coraz bardziej zintegrowana z dodatkowymi technologiami z obszaru multimediów, sieci bezprzewodowych, usług Web Service, platformy .NET i usługi plug-and-play. Niewątpliwym atutem Windows jest także zawsze uważany za stabilny, a jednak ciągle wzbogacany o ważne dodatki interfejs API.

Jeśli zatem szukasz kompletnego, rzetelnego i aktualnego podręcznika do nauki programowania za pomocą interfejsu Windows API, właśnie go znalazłeś! Książka ta w praktyczny sposób przedstawia wszystkie mechanizmy systemu Windows potrzebne programistom, pokazując, w jaki sposób działają funkcje tego systemu i jak wchodzą w interakcje z aplikacjami. Skoncentrowano się tu na podstawowych usługach systemu, w tym na systemie plików, zarządzaniu procesami i wątkami, komunikacji między procesami, programowaniu sieciowym i synchronizacji. Autor tej książki nie zamierza jednak obciążać Cię zbędną teorią i nieistotnymi szczegółami. Podaje Ci wiedzę opartą na prawdziwych przykładach, dzięki czemu szybko i sprawnie opanujesz poruszane tu zagadnienia. Wiadomości, które tu znajdziesz, pozwolą Ci zrozumieć interfejs Windows API w takim stopniu, byś zdobył solidne podstawy do rozwijania programów na platformę .NET Microsoftu.

Oto kompletny, aktualny przewodnik po programowaniu przy użyciu interfejsu Windows API!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008


THE ADDISON-WESLEY
Microsoft
Technology
Series
Wykorzystaj paralelizm i maksymalizuj wydajność w systemach wielordzeniowych
Zapewnij przenośność między środowiskami 64- i 32-bitowymi
Popraw skalowalność za pomocą wątków, pul wątków i mechanizmu IOCP
ADDISON-WESLEY
WYDANIE IV
PROGRAMOWANIE W SYSTEMIE WINDOWS
JOHNSON M. HART
Helion

SPIS TREŚCI

Rysunki **13**

Tabele **15**

Listingi **17**

Przebiegi programów **21**

Wstęp **23**

O autorze **33**

Rozdział 1. Wprowadzenie do systemu Windows **35**

Podstawy systemów operacyjnych 36
Ewolucja systemu Windows 37
Wersje systemu Windows 37
Pozycja systemu Windows na rynku 40
System Windows, standardy i systemy o otwartym dostępie do kodu źródłowego 41
Podstawy systemu Windows 43
Przenośność 32- i 64-bitowego kodu źródłowego 46
Biblioteka standardowa języka C — kiedy korzystać z niej do przetwarzania plików? 47
Co jest potrzebne do korzystania z tej książki? 48
Przykład — proste sekwencyjne kopiowanie pliku 50
Podsumowanie 58
Ćwiczenia 61

Rozdział 2. Korzystanie z systemu plików i znakowych operacji wejścia-wyjścia w systemie Windows **63**

Systemy plików w systemie Windows 64
Reguły tworzenia nazw plików 65
Otwieranie, wczytywanie, zapisywanie i zamykanie plików 66

Przerywnik — Unicode i znaki ogólne ... 74
Strategie związane z kodowaniem Unicode ... 77
Przykład — przetwarzanie błędów ... 78
Urządzenia standardowe ... 81
Przykład — kopiowanie wielu plików do standardowego wyjścia ... 82
Przykład — proste szyfrowanie pliku ... 85
Zarządzanie plikami i katalogami ... 88
Operacje wejścia-wyjścia konsoli ... 94
Przykład — wyświetlanie danych i instrukcji ... 96
Przykład — wyświetlanie bieżącego katalogu ... 99
Podsumowanie ... 100
Ćwiczenia ... 101

Rozdział 3. Zaawansowane przetwarzanie plików i katalogów oraz rejestr ... 103

64-bitowy system plików ... 104
Wskaźniki do plików ... 104
Pobieranie rozmiaru plików ... 109
Przykład — bezpośrednie aktualizowanie rekordów ... 111
Atrybuty plików i przetwarzanie katalogów ... 115
Przykład — wyświetlanie atrybutów plików ... 121
Przykład — ustawianie znaczników czasu dla plików ... 125
Strategie przetwarzania plików ... 126
Blokowanie dostępu do plików ... 128
Rejestr ... 134
Zarządzanie rejestrem ... 137
Przykład — wyświetlanie kluczy i zawartości rejestru ... 141
Podsumowanie ... 145
Ćwiczenia ... 146

Rozdział 4. Obsługa wyjątków ... 149

Wyjątki i procedury do ich obsługi ... 150
Wyjątki zmiennoprzecinkowe ... 157
Błędy i wyjątki ... 159
Przykład — traktowanie błędów jak wyjątków ... 161
Procedury obsługi zakończenia ... 163
Przykład — stosowanie procedur obsługi zakończenia do poprawy jakości programów ... 167
Przykład — stosowanie funkcji filtrującej ... 170
Procedury sterujące konsoli ... 175
Przykład — procedura sterująca konsoli ... 176
Wektorowa obsługa wyjątków ... 178
Podsumowanie ... 180
Ćwiczenia ... 181

Rozdział 5. Zarządzanie pamięcią, pliki odwzorowane w pamięci i biblioteki DLL 183

Architektura zarządzania pamięcią w systemie Windows 184
Sterty 187
Zarządzanie pamięcią na stercie 191
Przykład — sortowanie plików za pomocą binarnego drzewa wyszukiwań 198
Pliki odwzorowane w pamięci 203
Przykład — sekwencyjne przetwarzanie pliku za pomocą plików odwzorowanych 212
Przykład — sortowanie pliku odwzorowanego w pamięci 215
Przykład — stosowanie wskaźników z bazą 218
Biblioteki DLL 224
Przykład — dołączanie w czasie wykonywania programu funkcji do konwersji plików 231
Punkt wejścia do biblioteki DLL 232
Zarządzanie wersjami bibliotek DLL 234
Podsumowanie 236
Ćwiczenia 237

Rozdział 6. Zarządzanie procesem 241

Procesy i wątki w systemie Windows 241
Tworzenie procesu 244
Dane identyfikacyjne procesów 251
Powielanie uchwytów 252
Wychodzenie z procesu i kończenie jego działania 254
Oczekiwanie na zakończenie działania procesu 256
Bloki i łańcuchy znaków środowiska 258
Przykład — równoległe wyszukiwanie wzorca 260
Procesy w środowisku wieloprocesorowym 264
Czas wykonywania procesu 265
Przykład — czas wykonywania procesu 265
Generowanie zdarzeń sterujących konsoli 267
Przykład — proste zarządzanie zadaniem 268
Przykład — korzystanie z obiektów zadań 279
Podsumowanie 283
Ćwiczenia 284

Rozdział 7. Wątki i szeregowanie 287

Wprowadzenie do wątków 287
Podstawowe informacje o wątkach 290
Zarządzanie wątkami 291
Stosowanie biblioteki języka C w wątkach 296
Przykład — wielowątkowe wyszukiwanie wzorca 298

Wpływ na wydajność 301
Wątki główne i robocze oraz inne modele działania wątków 303
Przykład — sortowanie przez scalanie z wykorzystaniem wielu procesorów 304
Wprowadzenie do paralelizmu w programach 311
Pamięć TLS 312
Priorytety oraz szeregowanie procesów i wątków 314
Stany wątków 317
Pułapki i często popełniane błędy 319
Oczekiwanie z pomiarem czasu 321
Włókna 322
Podsumowanie 325
Ćwiczenia 326

Rozdział 8. Synchronizowanie wątków 329

Dlaczego trzeba synchronizować wątki? 330
Obiekty synchronizacji wątków 339
Obiekty CRITICAL_SECTION 340
Obiekty CRITICAL_SECTION do zabezpieczania współużytkowanych zmiennych 343
Przykład — prosty system z producentem i konsumentem 345
Muteksy 351
Semafory 358
Zdarzenia 361
Przykład — system z producentem i konsumentem 364
Więcej wskazówek na temat muteksów i obiektów CRITICAL_SECTION 369
Inne funkcje Interlocked 371
Wydajność przy zarządzaniu pamięcią 373
Podsumowanie 374
Ćwiczenia 375

Rozdział 9. Blokowanie, wydajność i dodatki w systemach NT6 377

Wpływ synchronizacji na wydajność 378
Program do badania wydajności 383
Dopracowywanie wydajności systemów wieloprocesorowych za pomocą liczby powtórzeń pętli obiektów CS 384
Blokady SRW w systemach NT6 387
Zmniejszanie rywalizacji za pomocą puli wątków 390
Porty kończenia operacji wejścia-wyjścia 393
Pule wątków z systemów NT6 394
Podsumowanie — wydajność blokowania 403
Paralelizm po raz wtóry 404
Koligacja procesora 409

Wskazówki i pułapki z obszaru wydajności 411
Podsumowanie 413
Ćwiczenia 414

Rozdział 10. Zaawansowana synchronizacja wątków 417

Model zmiennej warunkowej i właściwości związane z bezpieczeństwem 418
Stosowanie funkcji SignalObjectAndWait 426
Przykład — obiekt bariery z progiem 428
Obiekt kolejki 432
Przykład — wykorzystanie kolejek w wieloetapowym potoku 436
Zmienne warunkowe z systemów Windows NT6 446
Asynchroniczne wywołania procedur 451
Kolejkowanie asynchronicznych wywołań procedur 452
Oczekiwanie z obsługą alertów 454
Bezpieczne anulowanie wątków 456
Wątki Pthreads i przenośność aplikacji 457
Stosy wątków i liczba wątków 457
Wskazówki z obszaru projektowania, diagnozowania i testowania 458
Poza interfejs Windows API 461
Podsumowanie 462
Ćwiczenia 463

Rozdział 11. Komunikacja między procesami 467

Potoki anonimowe 468
Przykład — przekierowywanie wejścia-wyjścia za pomocą potoku anonimowego 469
Potoki nazwane 472
Funkcje do obsługi transakcji z wykorzystaniem potoku nazwanego 479
Przykład — system klient-serwer do przetwarzania wiersza poleceń 483
Komentarze na temat programu klient-serwer do przetwarzania wiersza poleceń 489
Szczeliny pocztowe 491
Tworzenie i nazywanie potoków oraz szczelin pocztowych i nawiązywanie połączeń z nimi 496
Przykład — serwer możliwy do znalezienia przez klienty 496
Podsumowanie 499
Ćwiczenia 500

Rozdział 12. Programowanie sieciowe z wykorzystaniem gniazd systemu Windows 503

Gniazda systemu Windows 504
Funkcje do obsługi gniazd serwera 507

Funkcje do obsługi gniazd klienta 513
Porównanie potoków nazwanych i gniazd 515
Przykład — funkcja do odbierania komunikatów w gnieździe 516
Przykład — klient oparty na gniazdach 517
Przykład — oparty na gniazdach serwer z nowymi mechanizmami 519
Serwery wewnątrzprocesowe 529
Komunikaty oparte na wierszach, punkty wejścia bibliotek DLL i pamięć TLS 531
Przykład — bezpieczna ze względu na wątki biblioteka DLL do obsługi komunikatów w gniazdach 533
Przykład — inna strategia tworzenia bibliotek DLL bezpiecznych ze względu na wątki 538
Datagramy 541
Gniazda Berkeley a gniazda systemu Windows 543
Operacje nakładanego wejścia-wyjścia oparte na gniazdach systemu Windows 544
Dodatkowe funkcje gniazd systemu Windows 544
Podsumowanie 545
Ćwiczenia 546

Rozdział 13. Usługi systemu Windows 549

Tworzenie usług systemu Windows — przegląd 550
Funkcja main() 551
Funkcje ServiceMain() 552
Procedura sterująca usługi 557
Rejestrowanie zdarzeń 558
Przykład — nakładka na usługi 558
Zarządzanie usługami systemu Windows 565
Podsumowanie — działanie usług i zarządzanie nimi 569
Przykład — powłoka do sterowania usługą 569
Współużytkowanie obiektów jądra przy użyciu usługi 574
Uwagi na temat diagnozowania usług 575
Podsumowanie 576
Ćwiczenia 577

Rozdział 14. Asynchroniczne operacje wejścia-wyjścia i IOCP 579

Przegląd asynchronicznych operacji wejścia-wyjścia w systemie Windows 580
Operacje nakładanego wejścia-wyjścia 581
Przykład — synchronizacja z wykorzystaniem uchwytu pliku 586
Przykład — przekształcanie pliku za pomocą operacji nakładanego wejścia-wyjścia i wielu buforów 587
Wzbogacone operacje wejścia-wyjścia z procedurami zakończenia 591
Przykład — przekształcanie plików za pomocą wzbogaconych operacji wejścia-wyjścia 597

Asynchroniczne operacje wejścia-wyjścia z wykorzystaniem wątków 600
Zegary oczekujące 602
Przykład — korzystanie z zegarów oczekujących 605
Mechanizm IOCP 607
Przykład — serwer oparty na mechanizmie IOCP 612
Podsumowanie 619
Ćwiczenia 620

Rozdział 15. Zabezpieczanie obiektów systemu Windows 623

Atrybuty zabezpieczeń 623
Przegląd zabezpieczeń — deskryptor zabezpieczeń 624
Flagi kontrolne deskryptora zabezpieczeń 628
Identyfikatory zabezpieczeń 628
Zarządzanie listami ACL 630
Przykład — uprawnienia w stylu UNIX-a do plików NTFS 632
Przykład — inicjowanie atrybutów zabezpieczeń 636
Wczytywanie i modyfikowanie deskryptorów zabezpieczeń 641
Przykład — odczytywanie uprawnień do pliku 643
Przykład — modyfikowanie uprawnień do plików 645
Zabezpieczanie obiektów jądra i komunikacji 645
Przykład — zabezpieczanie procesu i jego wątków 648
Przegląd dodatkowych mechanizmów zabezpieczeń 648
Podsumowanie 650
Ćwiczenia 651

Dodatek A Używanie przykładowych programów 655

Układ plików w pakiecie Przykłady 656

Dodatek B Przenośność kodu źródłowego — Windows, UNIX i Linux 659

Strategie tworzenia przenośnego kodu źródłowego 660
Usługi systemu Windows dla systemu UNIX 660
Przenośność kodu źródłowego z funkcjami systemu Windows 661
Rozdziały 2. i 3. — zarządzanie plikami i katalogami 667
Rozdział 4. — obsługa wyjątków 672
Rozdział 5. — zarządzanie pamięcią, pliki odwzorowane w pamięci i biblioteki DLL 674
Rozdział 6. — zarządzanie procesem 675
Rozdział 7. — wątki i szeregowanie 677
Rozdziały od 8. do 10. — synchronizowanie wątków 679
Rozdział 11. — komunikacja między procesami 681
Rozdział 14. — asynchroniczne operacje wejścia-wyjścia 684
Rozdział 15. — zabezpieczanie obiektów systemu Windows 685

Dodatek C Wyniki pomiarów wydajności 687

Konfiguracje testowe 687
Pomiary wydajności 690
Przeprowadzanie testów 703

Bibliografia 705

Skorowidz 709

ROZDZIAŁ 13

USŁUGI SYSTEMU WINDOWS

Serwery z rozdziałów 11. i 12. to aplikacje konsolowe. W zasadzie serwery te mogą działać w nieskończoność i obsługiwać liczne klienty, a te nawiązują połączenie, wysyłają żądania, odbierają odpowiedzi i zrywają połączenie. Oznacza to, że serwery te mogą ciągle oferować usługi. Jednak aby serwery były w pełni efektywne, potrzebna jest możliwość zarządzania usługami.

Usługi systemu Windows (ang. *Windows Services*)[1], nazywane niegdyś usługami NT, udostępniają mechanizmy do zarządzania potrzebne do przekształcenia serwerów na usługi, które można uruchamiać na żądanie lub w czasie ładowania systemu przed zalogowaniem się użytkowników. Można też wstrzymywać i wznawiać takie usługi, jak również kończyć i śledzić ich działanie. Informacje o usługach zawiera rejestr.

Ostatecznie wszystkie serwery, na przykład te zbudowane w rozdziałach 11. i 12., należy przekształcić na usługi, zwłaszcza jeśli mają być powszechnie stosowane przez klientów lub w organizacji.

System Windows udostępnia wiele usług. Należą do nich Klient DNS, różne usługi SQL Server i Usługi terminalowe. Pełny zestaw usług można wyświetlić za pomocą przystawki *Zarządzanie komputerem* dostępnej w *Panelu sterowania*.

Program *JobShell* (listing 6.3) z rozdziału 6. umożliwia uproszczone zarządzanie serwerem. Pozwala uruchomić serwer pod kontrolą zadania i wysłać sygnał zakończenia pracy. Usługi systemu Windows są jednak dużo bardziej kompletne i niezawodne. Podstawowym przykładem w tym rozdziale jest zmodyfikowana wersja programu *JobShell* umożliwiająca kontrolowanie takich usług.

[1] Ta terminologia bywa myląca, ponieważ system Windows udostępnia wiele usług, które nie są opisanymi tu usługami systemu Windows. Znaczenie powinno wynikać z kontekstu (podobnie zrozumienie pojęcia „system Windows" przy omawianiu interfejsu API nie sprawiało problemu).

W tym rozdziale pokazano też, jak przekształcić istniejącą aplikację konsolową w usługę systemu Windows, a także jak instalować, obserwować i kontrolować usługi. Znajduje się tu także omówienie rejestrowania zdarzeń (mechanizm ten umożliwia usłudze zapis wykonanych operacji w pliku).

Tworzenie usług systemu Windows — przegląd

Usługi systemu Windows działają pod kontrolą menedżera SCM (ang. *Service Control Manager*). Narzędzie to można wykorzystać do sterowania usługami na trzy sposoby:

1. Przez użycie przystawki administracyjnej *Usługi* (*Panel sterowania/System i konserwacja/Narzędzia administracyjne*).
2. Za pomocą narzędzia *sc.exe* obsługiwanego z poziomu wiersza poleceń.
3. Przez programistyczne kontrolowanie menedżera SCM, co ilustruje listing 13.3.

Przekształcenie aplikacji konsolowej (takiej jak *serverNP* lub *serverSK*) na usługę systemu Windows wymaga wykonania trzech podstawowych kroków w celu umieszczenia programu pod kontrolą menedżera SCM.

1. Należy utworzyć nowy punkt wejścia w postaci funkcji `main()` i zarejestrować w nim usługę w menedżerze SCM przez podanie logicznych punktów wejścia i nazwy usługi.
2. Trzeba przekształcić dawny punkt wejścia w postaci funkcji `main()` na funkcję *`ServiceMain()`*, która rejestruje procedurę sterującą usługi i informuje menedżer SCM o jej stanie. Pozostały kod programu może pozostać taki sam, choć można dodać polecenia do rejestrowania zdarzeń. Zamiast nazwy *`ServiceMain()`* należy podać nazwę logicznej usługi. W procesie może działać jedna lub kilka takich usług.
3. Należy napisać procedurę sterującą usługi reagującą na polecenia od menedżera SCM.

W tych trzech punktach wspomniano kilkakrotnie o tworzeniu, uruchamianiu i kontrolowaniu usług. W następnych podrozdziałach opisano szczegóły tych operacji, a rysunek 13.1 w dalszej części rozdziału ilustruje współdziałanie różnych komponentów.

Funkcja main()

Nowa funkcja `main()` wywoływana przez menedżer SCM ma za zadanie rejestrować usługę w menedżerze i uruchamiać program rozdzielający do sterowania usługą. Wymaga to wywołania funkcji `StartServiceCtrl ↪Dispatcher` z nazwami i punktami wejścia usług logicznych.

```
BOOL StartServiceCtrlDispatcher (
   SERVICE_TABLE_ENTRY *lpServiceStartTable)
```

Jedyny parametr, `lpServiceStartTable`, to adres tablicy z elementami `SERVICE_TABLE_ENTRY`. Każdy taki element to nazwa i punkt wejścia usługi logicznej. Końcem tablicy jest para elementów `NULL`.

Funkcja zwraca wartość `TRUE`, jeśli rejestracja zakończyła się powodzeniem.

Główny wątek procesu usługi wywołującego funkcję `StartService ↪CtrlDispatcher` nawiązuje połączenie z menedżerem SCM. Menedżer ten rejestruje usługę (lub usługi), przy czym wątek wywołujący służy jako program rozdzielający do sterowania usługą. Menedżer SCM nie zwraca sterowania do wątku wywołującego do czasu zakończenia pracy przez wszystkie usługi. Warto zauważyć, że usługi logiczne nie rozpoczynają działania od razu. Uruchomienie usługi wymaga wywołania opisanej dalej funkcji `StartService`.

Listing 13.1 przedstawia typowy program główny usługi z jedną usługą logiczną.

Listing 13.1. Funkcja main — główny punkt wejścia usługi

```
#include "Everything.h"

void WINAPI ServiceMain (DWORD argc, LPTSTR argv[]);
static LPTSTR serviceName = _T("SocketCommandLineService");

/* Główna procedura uruchamiająca program rozdzielający do sterowania usługą. */
VOID _tmain (int argc, LPTSTR argv[])
{
    SERVICE_TABLE_ENTRY dispatchTable[] =
    {
        { serviceName, ServiceMain },
        { NULL, NULL }
    };
```

```
    if (!StartServiceCtrlDispatcher (dispatchTable))
        ReportError (_T("Nie mozna uruchomic programu
        ↪rozdzielajacego."), 1, TRUE);
    /* Funkcja ServiceMain() zadziała po uruchomieniu jej przez menedżer SCM. */
    /* Sterowanie jest zwracane do tego miejsca dopiero po zamknięciu wszystkich usług. */
    return;
}
```

Funkcje ServiceMain()

Tablica przydziału określa funkcje, jak pokazano to na listingu 13.1, a każda z nich reprezentuje usługę logiczną. Funkcje te to wzbogacone wersje podstawowego programu przekształconego na usługę, a menedżer SCM wywołuje każdą usługę logiczną w osobnym wątku. Usługa logiczna może z kolei uruchomić dodatkowe wątki, takie jak wątki robocze serwera generowane przez programy *serverSK* i *serverNP*. Często usługa systemu Windows obejmuje tylko jedną usługę logiczną. Na listingu 13.2 usługa logiczna to zmodyfikowana wersja głównego serwera (listing 12.2). W jednej usłudze systemu Windows można uruchomić usługi logiczne oparte na gniazdach i potokach nazwanych. Wtedy należy udostępnić dwie funkcje główne usługi.

Choć funkcja *`ServiceMain()`* jest oparta na funkcji `main()` oraz ma parametry z liczbą i łańcuchem argumentów, warto zwrócić uwagę na jedną małą różnicę. Funkcję tę należy zadeklarować jako `void WINAPI`. Nie powinna ona zwracać wartości typu `int`, jak robi to zwykła funkcja `main()`.

Rejestrowanie procedury sterującej usługi

Procedura sterująca usługi wywoływana przez menedżer SCM musi mieć możliwość kontrolowania powiązanej z nią usługi logicznej. Procedura sterująca konsoli z programu *serverSK* ustawia globalną flagę zamknięcia i ilustruje (choć na uproszczonym przykładzie), jak powinna wyglądać taka procedura. Jednak każda usługa logiczna musi przede wszystkim natychmiast zarejestrować procedurę za pomocą funkcji `RegisterServiceCtrl ↪HandlerEx`. Funkcję tę należy wywołać na początku funkcji `Service ↪Main()` i nie uruchamiać jej w dalszej części programu. Menedżer SCM po otrzymaniu żądania sterującego od usługi wywołuje omawianą procedurę.

```
SERVICE_STATUS_HANDLE
   RegisterServiceCtrlHandlerEx (
   LPCTSTR lpServiceName,
   LPHANDLER_FUNCTION_EX lpHandlerProc,
   LPVOID lpContext)
```

Parametry

Parametr `lpServiceName` to podana przez użytkownika nazwa usługi umieszczona we wpisie dotyczącym danej usługi logicznej w tablicy usług. Nazwa ta powinna odpowiadać nazwie funkcji *`ServiceMain`* zarejestrowanej za pomocą funkcji `StartServiceCtrlDispatcher`.

Parametr `lpHandlerProc` to adres rozbudowanej procedury obsługi opisanej w dalszym podrozdziale.

Parametr `lpContext` to zdefiniowane przez użytkownika dane przekazane do procedury sterującej. Umożliwiają one jednej procedurze sterującej rozróżnianie wielu korzystających z niej usług.

Jeśli wystąpił błąd, zwracana wartość (obiekt typu `SERVICE_STATUS_`
`↪HANDLE`) to `0`. Do zbadania błędów należy zastosować standardowe metody.

Ustawianie stanu usługi

Po zarejestrowaniu procedury obsługi następne zadanie polega na ustawieniu stanu usługi na wartość `SERVICE_START_PENDING`. Umożliwia to funkcja `SetServiceStatus`. Posłuży ona jeszcze w kilku innych miejscach do ustawiania różnych wartości w celu poinformowania menedżera SCM o obecnym stanie usługi. W jednym z dalszych podrozdziałów i w tabeli 13.3 opisano inne obok `SERVICE_START_PENDING` prawidłowe wartości stanu.

Procedura sterująca usługi musi ustawić stan przy każdym wywołaniu, nawet jeśli stan ten się nie zmienił.

Ponadto każdy wątek usługi może w dowolnym momencie wywołać funkcję `SetServiceStatus`, aby przekazać informacje o postępie w pracy programu, błędach i inne. Usługi często korzystają z odrębnego wątku do okresowego aktualizowania stanu. Odstęp między aktualizacjami jest określony w składowej w parametrze ze strukturą danych. Menedżer SCM może uznać, że wystąpił błąd, jeśli aktualizacja stanu nie nastąpi w podanym czasie.

```
BOOL SetServiceStatus (
   SERVICE_STATUS_HANDLE hServiceStatus,
   LPSERVICE_STATUS lpServiceStatus)
```

Parametry

Parametr `hServiceStatus` to wartość typu `SERVICE_STATUS_HANDLE` zwrócona przez funkcję `RegisterServiceCtrlHandlerEx` (dlatego należy ją wywołać przed funkcją `SetServiceStatus`).

Parametr `lpServiceStatus` wskazuje strukturę `SERVICE_STATUS`. Opisuje ona właściwości, stan i możliwości usługi.

Struktura SERVICE_STATUS

Oto definicja struktury `SERVICE_STATUS`:

```
typedef struct _SERVICE_STATUS {
   DWORD dwServiceType;
   DWORD dwCurrentState;
   DWORD dwControlsAccepted;
   DWORD dwWin32ExitCode;
   DWORD dwServiceSpecificExitCode;
   DWORD dwCheckPoint;
   DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

Parametry

Parametr `dwWin32ExitCode` to normalny kod wyjścia wątku usługi logicznej. Usługa musi ustawić ten kod na wartość `NO_ERROR` w czasie działania i przy standardowym kończeniu pracy. Mimo nazwy parametru można go używać także w aplikacjach 64-bitowych. Człon „32” pojawia się też w innych nazwach.

Parametr `dwServiceSpecificExitCode` można wykorzystać do zasygnalizowania błędu w czasie uruchamiania lub zatrzymywania usługi, jednak wartość ta zostanie zignorowana, jeśli parametr `dwWin32ExitCode` ma wartość różną od `ERROR_SERVICE_SPECIFIC_ERROR`.

Usługa powinna okresowo zwiększać wartość parametru `dwCheckPoint`, aby informować o przechodzeniu przez wszystkie etapy pracy, w tym ini-

cjowanie i zamykanie usługi. Ta wartość jest nieważna i powinna wynosić `0`, jeśli usługa nie ma do wykonania operacji uruchomienia, zatrzymania, wstrzymania lub kontynuowania.

Parametr `dwWaitHint` to czas w milisekundach między wywołaniami funkcji `SetServiceStatus` w celu zwiększenia wartości parametru `dwCheckPoint` lub zmiany wartości parametru `dwCurrentState`. Jak wcześniej wspomniano, jeśli ten czas upłynie bez wywołania funkcji `Set`
`↪ServiceStatus`, menedżer SCM może uznać, że wystąpił błąd.

Pozostałe składowe struktury `SERVICE_STATUS` opisano w osobnych punktach.

Typ usługi (dwServiceType)

Parametr `dwServiceType` musi mieć jedną z wartości opisanych w tabeli 13.1.

Tabela 13.1. Typy usług

Wartość	Znaczenie
`SERVICE_WIN32_OWN_PROCESS`	Sygnalizuje, że usługa systemu Windows działa we własnym procesie ze swoimi zasobami. *Wartość tę wykorzystano na listingu 13.2.*
`SERVICE_WIN32_SHARE_PROCESS`	Określa usługę systemu Windows współużytkującą proces z innymi usługami. Powoduje to połączenie kilku usług w jednym procesie, co pozwala zmniejszyć potrzebną ilość zasobów.
`SERVICE_KERNEL_DRIVER`	Określa sterownik urządzenia systemu Windows i jest zarezerwowana do użytku przez system.
`SERVICE_FILE_SYSTEM_DRIVER`	Określa sterownik systemu plików systemu Windows i także jest zarezerwowana.
`SERVICE_INTERACTIVE_PROCESS`	Tę flagę można połączyć tylko z dwoma wartościami `SERVICE_WIN32_X`. Usługi interaktywne powodują zagrożenie w obszarze bezpieczeństwa i należy ich unikać.

W tej książce typ usługi to prawie zawsze `SERVICE_WIN32_OWN_`
`↪PROCESS`, a ustawienie `SERVICE_WIN32_SHARE_PROCESS` to jedyna inna wartość odpowiednia dla usług w trybie użytkownika. Przedstawienie pozostałych wartości pozwala jednak pokazać, że usługi odgrywają wiele różnych ról.

Stan usługi (dwCurrentState)

Parametr `dwCurrentState` określa obecny stan usługi. W tabeli 13.2 przedstawiono różne możliwe wartości.

Tabela 13.2. Wartości określające stan usługi

Wartość	Znaczenie
`SERVICE_STOPPED`	Usługa nie jest uruchomiona.
`SERVICE_START_PENDING`	Usługa rozpoczyna działanie, ale nie jest jeszcze gotowa do odpowiadania na żądania (na przykład dlatego, że wątek roboczy nie został jeszcze uruchomiony).
`SERVICE_STOP_PENDING`	Usługa zatrzymuje działanie, ale nie zakończyła jeszcze zamykania. Na przykład globalna flaga zamknięcia została ustawiona, ale wątki robocze jeszcze nie odpowiedziały.
`SERVICE_RUNNING`	Usługa działa.
`SERVICE_CONTINUE_PENDING`	Usługa jest w trakcie wznawiania pracy po wstrzymaniu, ale jeszcze nie działa.
`SERVICE_PAUSE_PENDING`	Trwa wstrzymywanie usługi, ale nie przeszła ona jeszcze bezpiecznie w stan wstrzymania.
`SERVICE_PAUSED`	Usługa jest wstrzymana.

Akceptowane kody sterowania (dwControlsAccepted)

Parametr `dwControlsAccepted` określa kody sterowania, które usługa ma akceptować i przetwarzać za pomocą procedury sterującej usługi (zobacz następny podrozdział). W tabeli 13.3 wymieniono trzy wartości użyte w jednym z dalszych przykładów. Odpowiednie wartości należy połączyć bitowym operatorem „or” (`|`). Trzy dodatkowe wartości zawiera opis struktury `SERVICE_STATUS` w dokumentacji MSDN.

Kod specyficzny dla usługi

Po zarejestrowaniu procedury obsługi i ustawieniu stanu na wartość `SERVICE_START_PENDING` usługa może przeprowadzić swoją inicjację oraz ponownie określić stan. W przekształconej wersji programu *serverSK* po zainicjowaniu gniazd i przygotowaniu serwera do akceptowania połączeń od klientów status należy ustawić na wartość `SERVICE_RUNNING`.

Tabela 13.3. Kody sterowania akceptowane przez usługę (niepełna lista)

Wartość	Znaczenie
SERVICE_ACCEPT_STOP	Dodaje obsługę żądania SERVICE_CONTROL_STOP.
SERVICE_ACCEPT_PAUSE_CONTINUE	Dodaje obsługę żądań SERVICE_CONTROL_PAUSE i SERVICE_CONTROL_CONTINUE.
SERVICE_ACCEPT_SHUTDOWN (funkcja ControlService nie może wysłać tego kodu sterowania)	Powiadamia usługę o zamykaniu systemu. Umożliwia to systemowi przesłanie wartości SERVICE_CONTROL_SHUTDOWN do usługi. Ten kod jest przeznaczony do wyłącznego użytku przez system Windows.
SERVICE_ACCEPT_PARAMCHANGE	Umożliwia zmianę parametrów rozruchowych bez ponownego uruchamiania. Powiadomienie to SERVICE_CONTROL_PARAMCHANGE.

Procedura sterująca usługi

Procedura sterująca usługi, czyli funkcja zwrotna określona w funkcji RegisterServiceCtrlHandlerEx, ma następującą postać:

```
DWORD WINAPI HandlerEx (
   DWORD dwControl,
   DWORD dwEventType,
   LPVOID lpEventData,
   LPVOID lpContext)
```

Parametr dwControl określa wysłany przez menedżer SCM sygnał sterujący, który należy przetworzyć.

Parametr ten przyjmuje 14 wartości, w tym te wymienione w tabeli 13.3. Oto pięć wartości istotnych w omawianym przykładzie:

```
SERVICE_CONTROL_STOP
SERVICE_CONTROL_PAUSE
SERVICE_CONTROL_CONTINUE
SERVICE_CONTROL_INTERROGATE
SERVICE_CONTROL_SHUTDOWN
```

Można też stosować zdefiniowane przez użytkownika wartości z przedziału od 128 do 255, jednak nie wykorzystano ich w tym miejscu.

Parametr dwEventType ma zwykle wartość 0, jednak do zarządzania urządzeniami służą wartości niezerowe (omawianie tego zagadnienia wykracza poza zakres książki). Parametr lpEventDate udostępnia dodatkowe dane wymagane dla niektórych zdarzeń.

Ostatni parametr, lpContext, zawiera zdefiniowane przez użytkownika dane przekazane do funkcji RegisterServiceCtrlHandlerEx przy rejestrowaniu określonej procedury obsługi.

Procedura obsługi jest wywoływana przez menedżer SCM w tym samym wątku, co program główny, i jest zwykle napisana jako instrukcja switch. Ilustrują to przykłady.

Rejestrowanie zdarzeń

Usługi działają „bezwejściowo" bez interakcji z użytkownikiem, dlatego zwykle nie powinny bezpośrednio wyświetlać komunikatów o stanie. W systemach starszych niż Vista i NT6 niektóre usługi tworzyły konsolę, pole komunikatów lub okno do interakcji z użytkownikiem. Obecnie techniki te nie są już dostępne.

Rozwiązanie tego problemu polega na rejestrowaniu zdarzeń w pliku dziennika lub wykorzystaniu mechanizmu rejestrowania zdarzeń z systemu Windows. Takie zdarzenia są przechowywane w systemie Windows i można je wyświetlić za pomocą przeglądarki zdarzeń dostępnej w *Narzędziach administracyjnych* w *Panelu sterowania*.

Dalszy przykładowy program *SimpleService* (listing 13.2) rejestruje ważne zdarzenia i błędy usługi w pliku dziennika. W ćwiczeniu poproszono o zmodyfikowanie tego programu przez zastosowanie zdarzeń systemu Windows.

Przykład — nakładka na usługi

Program z listingu 13.2 przekształca dowolną funkcję _tmain na usługę. Aby było to możliwe, trzeba wykonać opisane dalej operacje. Istniejący kod serwera (czyli dawna funkcja _tmain) jest wywoływany jako wątek lub proces z poziomu funkcji ServiceSpecific. Dlatego przedstawiony tu kod to nakładka na istniejący serwer.

Listing 13.2. Program SimpleService — nakładka na serwery

```
/* Rozdział 13. SimpleService.c.
   Najprostszy przykład usługi systemu Windows.
   Jej jedyne możliwości to aktualizowanie punktów kontrolnych
   i akceptowanie podstawowych poleceń sterujących.
   Program można też uruchomić jako niezależną aplikację. */

#include "Everything.h"
#include <time.h>
#define UPDATE_TIME 1000 /* Sekunda odstępu między aktualizacjami. */

VOID LogEvent (LPCTSTR, WORD), LogClose();
BOOL LogInit(LPTSTR);
void WINAPI ServiceMain (DWORD argc, LPTSTR argv[]);
VOID WINAPI ServerCtrlHandler(DWORD);
void UpdateStatus (int, int);
int ServiceSpecific (int, LPTSTR *);

static BOOL shutDown = FALSE, pauseFlag = FALSE;
static SERVICE_STATUS hServStatus;
static SERVICE_STATUS_HANDLE hSStat; /* Uchwyt do ustawiania stanu. */

static LPTSTR serviceName = _T("SimpleService");
static LPTSTR logFileName = _T(".\\LogFiles\\SimpleServiceLog.txt");
static BOOL consoleApp = FALSE, isService;

/* Główna procedura uruchamiająca program rozdzielający do sterowania usługą. */
/* Program można też uruchomić jako niezależną aplikację konsolową. */
/* Stosowanie: simpleService [-c] */
/*             Opcja -c powoduje uruchomienie aplikacji konsolowej zamiast usługi. */

VOID _tmain (int argc, LPTSTR argv[])
{
    SERVICE_TABLE_ENTRY DispatchTable[] =
    {
        { serviceName, ServiceMain},
        { NULL,        NULL }
    };

    Options (argc, argv, _T("c"), &consoleApp, NULL);
    isService = !consoleApp;
    /* Inicjowanie pliku dziennika. */
    if (!LogInit (logFileName)) return;

    if (isService) {
        LogEvent(_T("Uruchamianie programu rozdzielajacego."),
        ↪EVENTLOG_SUCCESS);
        StartServiceCtrlDispatcher (DispatchTable);
```

```
    } else {

        LogEvent(_T("Uruchamianie aplikacji."), EVENTLOG_SUCCESS);
        ServiceSpecific (argc, argv);
    }
    LogClose();
    return;
}

/* Punkt wejścia w postaci funkcji ServiceMain wywoływanej przez program główny. */
void WINAPI ServiceMain (DWORD argc, LPTSTR argv[])
{
    LogEvent (_T("Wchodzenie do funkcji ServiceMain."),
    ↪EVENTLOG_SUCCESS);

    hServStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    hServStatus.dwCurrentState = SERVICE_START_PENDING;
    hServStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP |
        SERVICE_ACCEPT_SHUTDOWN | SERVICE_ACCEPT_PAUSE_CONTINUE;
    hServStatus.dwWin32ExitCode = NO_ERROR;
    hServStatus.dwServiceSpecificExitCode = 0;
    hServStatus.dwCheckPoint = 0;
    hServStatus.dwWaitHint = 2 * UPDATE_TIME;

    hSStat =
        RegisterServiceCtrlHandler( serviceName, ServerCtrlHandler);

    if (hSStat == 0) {
        LogEvent (_T("Nie mozna zarejestrowac procedury obslugi."),
                  EVENTLOG_ERROR_TYPE);
        hServStatus.dwCurrentState = SERVICE_STOPPED;
        hServStatus.dwWin32ExitCode = ERROR_SERVICE_SPECIFIC_ERROR;
        hServStatus.dwServiceSpecificExitCode = 1;
        UpdateStatus (SERVICE_STOPPED, -1);
        return;
    }

    LogEvent (_T("Zarejestrowano procedure sterujaca."),
    ↪EVENTLOG_SUCCESS);
    SetServiceStatus (hSStat, &hServStatus);
    LogEvent (_T("Stan SERVICE_START_PENDING."), EVENTLOG_SUCCESS);

    /* Uruchamianie zadań specyficznych dla usługi. Ogólne operacje zostały wykonane. */
    ServiceSpecific (argc, argv);

    /* Sterowanie jest zwracane do tego miejsca dopiero po zakończeniu
       działania funkcji ServiceSpecific, co oznacza zamknięcie systemu. */
    LogEvent (_T("Watki uslugi zakonczyly prace."),
    ↪EVENTLOG_SUCCESS);
    LogEvent (_T("Ustawianie stanu na SERVICE_STOPPED."),
    ↪EVENTLOG_SUCCESS);
```

```
    UpdateStatus (SERVICE_STOPPED, 0);
    LogEvent (_T("Stan ustawiono na SERVICE_STOPPED."),
    ↪EVENTLOG_SUCCESS);
    return;
}

/* Funkcja specyficzna dla usługi (główna) wywoływana w funkcji ServiceMain. */
int ServiceSpecific (int argc, LPTSTR argv[])
{
    UpdateStatus (-1, -1); /* Zmiana stanu i ustawienie następnego punktu kontrolnego. */
    /* Serwer można uruchomić jako wątek lub proces. */
    /* Załóżmy, że usługa rozpoczyna pracę w dwie sekundy. */
    UpdateStatus (SERVICE_RUNNING, -1);
    LogEvent (_T("Aktualizacja stanu. Usluga dziala."),
    ↪EVENTLOG_SUCCESS);

    /* Okresowe aktualizowanie stanu. */
    /*** Pętlę do obsługi aktualizacji można uruchomić w odrębnym wątku. ***/
    /* Należy też sprawdzić flagę pauseFlag – zobacz ćwiczenie 13-1. */
    LogEvent (_T("Uruchamianie glownej petli uslugi."),
    ↪EVENTLOG_SUCCESS);
    while (!shutDown) { /* Flaga shutDown jest ustawiana przy poleceniu zamknięcia. */
        Sleep (UPDATE_TIME);
        UpdateStatus (-1, -1); /* Załóżmy, że zmiany nie wystąpiły. */
        LogEvent (_T("Aktualizacja stanu. Brak zmian."),
        ↪EVENTLOG_SUCCESS);
    }
    LogEvent (_T ("Proces serwera zakonczyl prace."),
    ↪EVENTLOG_SUCCESS);
    return 0;
}

/* Procedura sterująca. */
VOID WINAPI ServerCtrlHandler( DWORD dwControl)
{
    switch (dwControl) {
    case SERVICE_CONTROL_SHUTDOWN:
    case SERVICE_CONTROL_STOP:
        shutDown = TRUE; /* Ustawianie flagi globalnej shutDown. */
        UpdateStatus (SERVICE_STOP_PENDING, -1);
        break;
    case SERVICE_CONTROL_PAUSE:
        pauseFlag = TRUE;
        /* Implementacja wstrzymania to temat ćwiczenia 13.1. */
        break;
    case SERVICE_CONTROL_CONTINUE:
        pauseFlag = FALSE;
        /* Implementacja kontynuacji to też ćwiczenie. */
        break;
```

```
    case SERVICE_CONTROL_INTERROGATE:
        break;
    default:
        if (dwControl > 127 && dwControl < 256) /* Zdefiniowane przez
                                                   użytkownika. */
        break;
    }
    UpdateStatus (-1, -1);
    return;
}

void UpdateStatus (int NewStatus, int Check)
/* Ustawianie stanu usługi i punktu kontrolnego (na określoną wartość lub przez zwiększenie). */
{
    if (Check < 0 ) hServStatus.dwCheckPoint++;
    else hServStatus.dwCheckPoint = Check;

    if (NewStatus >= 0) hServStatus.dwCurrentState = NewStatus;
    if (isService) {
        if (!SetServiceStatus (hSStat, &hServStatus)) {
            LogEvent (_T("Nie mozna ustawic stanu."),
            ↪EVENTLOG_ERROR_TYPE);
            hServStatus.dwCurrentState = SERVICE_STOPPED;
            hServStatus.dwWin32ExitCode =
            ↪ERROR_SERVICE_SPECIFIC_ERROR;
            hServStatus.dwServiceSpecificExitCode = 2;
            UpdateStatus (SERVICE_STOPPED, -1);
            return;
        } else {
            LogEvent (_T("Zaktualizowano stan uslugi."),
            ↪EVENTLOG_SUCCESS);
        }
    } else {
        LogEvent (_T("Zaktualizowano stan niezaleznego programu."),
        ↪EVENTLOG_SUCCESS);
    }
    return;
}

/* Proste rejestrowanie zdarzeń w pliku. */
static FILE * logFp = NULL;
/* Bardzo prosta usługa rejestrująca (korzysta z pliku). */
VOID LogEvent (LPCTSTR UserMessage, WORD type)
{
    TCHAR cTimeString[30] = _T("");
    time_t currentTime = time(NULL);
    _tcsncat (cTimeString, _tctime(&currentTime), 30);
    /* Usuwanie znaku nowego wiersza na końcu łańcucha z czasem. */
    cTimeString[_tcslen(cTimeString)-2] = _T('\0');
    _ftprintf(logFp, _T("%s. "), cTimeString);
```

```
    if (type == EVENTLOG_SUCCESS || type ==
    ➥EVENTLOG_INFORMATION_TYPE)
        _ftprintf(logFp, _T("%s"), _T("Informacja.  "));
    else if (type == EVENTLOG_ERROR_TYPE)
        _ftprintf(logFp, _T("%s"), _T("Blad.        "));
    else if (type == EVENTLOG_WARNING_TYPE)
        _ftprintf(logFp, _T("%s"), _T("Ostrzezenie. "));
    else
        _ftprintf(logFp, _T("%s"), _T("Nieznane.    "));
    _ftprintf(logFp, _T("%s\n"), UserMessage);
    fflush(logFp);
    return;
}

BOOL LogInit(LPTSTR name)
{
    logFp = _tfopen (name, _T("a+"));
    if (logFp != NULL) LogEvent (_T("Zainicjowano rejestrowanie."),
                                    EVENTLOG_SUCCESS);
    return (logFp != NULL);
}

VOID LogClose()
{
    LogEvent (_T("Zamykanie dziennika."), EVENTLOG_SUCCESS);
    return;
}
```

Opcja –c podawana w wierszu poleceń określa, że kod należy uruchomić jako niezależny program (na przykład w celach diagnostycznych). Pominięcie tej opcji powoduje wywołanie funkcji `StartServiceCtrl ➥Dispatcher`.

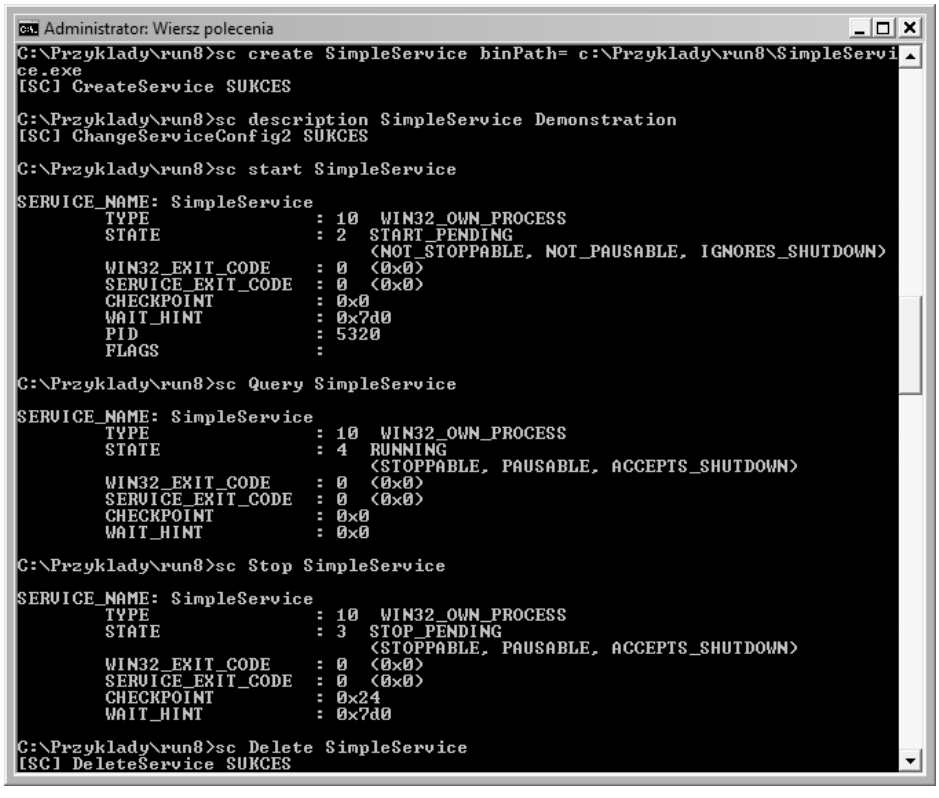
Innym dodatkiem jest plik dziennika. Dla uproszczenia jego nazwę zapisano na stałe w kodzie. Usługa rejestruje w tym pliku istotne zdarzenia. Na końcu znajdują się proste funkcje do inicjowania i zamykania dziennika oraz rejestrowania komunikatów.

W komentarzach opisano kilka innych uproszczeń i ograniczeń.

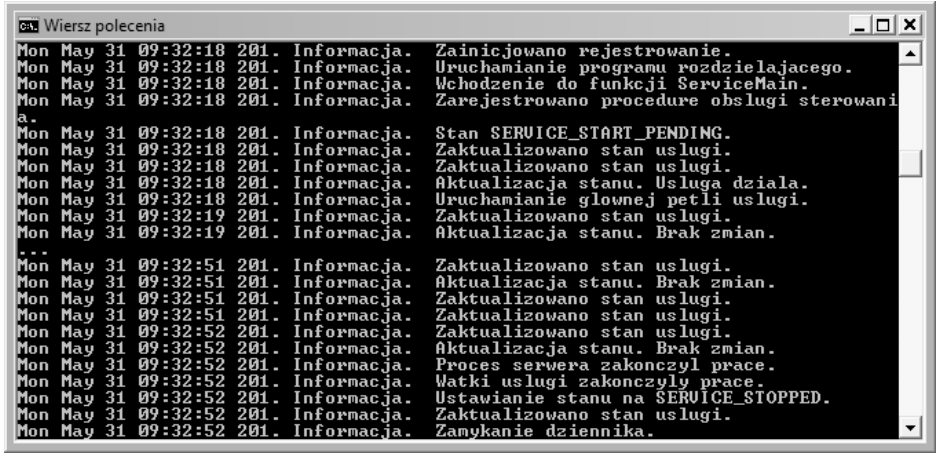
Uruchamianie prostej usługi

Przebieg programu 13.2a pokazuje, jak za pomocą uruchamianego w wierszu poleceń narzędzia *sc* utworzyć, uruchomić, odpytać, zatrzymać i zamknąć program *SimpleService*. Tylko administrator może wykonywać te operacje.

Przebieg programu 13.2b przedstawia zawartość pliku dziennika.



Przebieg programu 13.2a. Usługa SimpleService kontrolowana za pomocą narzędzia sc



Przebieg programu 13.2b. SimpleServiceLog.txt — plik dziennika

Zarządzanie usługami systemu Windows

Po napisaniu usługi następnym krokiem jest umieszczenie jej pod kontrolą menedżera SCM, aby mógł on uruchamiać i zatrzymywać usługę oraz sterować nią na inne sposoby. Choć program *sc.exe* i usługowe narzędzia administracyjne na to pozwalają, usługami można też zarządzać programistycznie, co opisano w dalszej części podrozdziału.

Trzeba wykonać kilka kroków — otworzyć menedżer SCM, utworzyć usługę pod jego kontrolą, a następnie uruchomić tę usługę. Te etapy nie służą do bezpośredniego kontrolowania usługi. Są to instrukcje dla menedżera SCM, który z kolei steruje określoną usługą.

Otwieranie menedżera SCM

Do utworzenia usługi niezbędny jest odrębny proces uruchomiony jako „administrator". Podobnie działa program *JobShell* (rozdział 6.) uruchamiający zadania. Pierwszy krok to otwarcie menedżera SCM i pobranie uchwytu, który umożliwi utworzenie usługi.

```
SC_HANDLE OpenSCManager (
   LPCTSTR lpMachineName,
   LPCTSTR lpDatabaseName,
   DWORD dwDesiredAccess)
```

Parametry

Jeśli menedżer SCM działa na komputerze lokalnym, parametr `lpMachine`
`↪Name` ma wartość `NULL`, natomiast można też uzyskać dostęp do menedżera uruchomionego na maszynach z danej sieci.

Parametr `lpDatabaseName` ma zwykle wartość `NULL`.

Parametr `dwDesiredAccess` ma standardowo wartość `SC_MANAGER_`
`↪ALL_ACCESS`, jednak można określić bardziej ograniczone uprawnienia dostępu, jak opisano to w dokumentacji dostępnej w internecie.

Tworzenie i usuwanie usługi

Aby zarejestrować usługę, należy wywołać funkcję `CreateService`.

```
SC_HANDLE CreateService (
   SC_HANDLE hSCManager,
   LPCTSTR lpServiceName,
   LPCTSTR lpDisplayName,
   DWORD dwDesiredAccess,
   DWORD dwServiceType,
   DWORD dwStartType,
   DWORD dwErrorControl,
   LPCTSTR lpBinaryPathName,
   LPCTSTR lpLoadOrderGroup,
   LPDWORD lpdwTagId,
   LPCTSTR lpDependencies,
   LPCTSTR lpServiceStartName,
   LPCTSTR lpPassword);
```

Funkcja `CreateService` umieszcza nowe usługi w rejestrze w kluczu:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

Nie należy jednak próbować pomijać funkcji `CreateService` przez bezpośrednie manipulowanie rejestrem. O tym kluczu wspomniano tylko po to, aby pokazać, w jaki sposób Windows przechowuje informacje o usługach.

Parametry

Parametr `hSCManager` to uchwyt typu `SC_HANDLE` zwrócony przez funkcję `OpenSCManager`.

Parametr `lpServiceName` to nazwa używana do późniejszego wskazywania usługi. Jest to jedna z nazw usług logicznych podanych w tablicy przydziału w wywołaniu funkcji `StartServiceCtrlDispatcher`. Warto zauważyć, że funkcję `CreateService` trzeba wywołać dla każdej usługi logicznej.

Parametr `lpDisplayName` to nazwa usługi wyświetlana użytkownikowi w narzędziu administracyjnym *Usługi* (*Panel sterowania/Narzędzia administracyjne*) i w innych miejscach. Nazwa ta pojawia się natychmiast po udanym wywołaniu funkcji `CreateService`.

Do parametru `dwDesiredAccess` można przypisać wartość `SERVICE_ALL_ACCESS` lub kombinację wartości `GENERIC_READ`, `GENERIC_WRITE` i `GENERIC_EXECUTE`. Więcej szczegółów na ten temat można znaleźć w dokumentacji MSDN.

Wartości parametru `dwServiceType` przedstawiono w tabeli 13.1.

Parametr dwStartType określa sposób uruchamiania usługi. W przykładach użyto ustawienia SERVICE_DEMAND_START, natomiast inne wartości (SERVICE_BOOT_START i SERVICE_SYSTEM_START) umożliwiają uruchamianie usług sterowników urządzeń w czasie ładowania lub uruchamiania systemu. Ustawienie SERVICE_AUTO_START informuje, że usługę należy włączyć w czasie uruchamiania komputera.

Parametr lpBinaryPathName określa pełną ścieżkę do pliku wykonywalnego. Koniecznie trzeba podać rozszerzenie *.exe*. Jeśli ścieżka obejmuje odstępy, należy użyć cudzysłowów.

Inne parametry dotyczą nazwy konta i hasła, grup służących do łączenia usług, a także zależności, jeśli istnieje kilka zależnych od siebie usług.

Wartości parametrów konfiguracyjnych istniejącej usługi można zmienić za pomocą funkcji ChangeServiceConfig i ChangeService
➥Config2. Ta druga jest prostsza i prawdopodobnie z tego powodu nie nosi nazwy ChangeServiceConfigEx. Do wskazywania usług służą ich uchwyty, a wspomniane funkcje pozwalają ustawić nowe wartości większości parametrów. Można na przykład podać nową wartość parametrów dwServiceType lub dwStartType, jednak nie można zmienić parametru dwAccess.

Istnieje też funkcja OpenService. Służy ona do pobierania uchwytu nazwanej usługi. Aby wyrejestrować usługę z menedżera SCM, należy użyć funkcji DeleteService, a do zamykania uchwytów SC_HANDLE przeznaczona jest funkcja CloseServiceHandle.

Uruchamianie usługi

Usługa po utworzeniu nie jest uruchomiona. Należy wywołać funkcję *Ser*
➥*viceMain()* przez podanie uchwytu uzyskanego za pomocą funkcji CreateService. Razem z uchwytem należy podać parametry wiersza poleceń (argc, argv) oczekiwane przez funkcję główną usługi (czyli funkcję określoną w tablicy przydziału).

```
BOOL StartService (
   SC_HANDLE hService,
   DWORD argc,
   LPTSTR argv[])
```

Kontrolowanie usługi

Usługę można kontrolować przez nakazanie menedżerowi SCM wywołania procedury sterującej usługi z określonym kodem sterowania.

```
BOOL ControlService (
   SC_HANDLE hService,
   DWORD dwControlCode,
   LPSERVICE_STATUS lpServStat)
```

Oto wartości parametru `dwControlCode` istotne w omawianych przykładach:

```
SERVICE_CONTROL_STOP
SERVICE_CONTROL_PAUSE
SERVICE_CONTROL_CONTINUE
SERVICE_CONTROL_INTERROGATE
SERVICE_CONTROL_SHUTDOWN
```

Ważne są też wartości z przedziału od `128` do `255` określone przez użytkownika. Dodatkowe nazwane kody informują usługę o zmianie wartości uruchomieniowych lub modyfikacjach w zakresie powiązania.

Wartość `SERVICE_CONTROL_INTERROGATE` nakazuje usłudze przesłanie informacji o stanie przez wywołanie funkcji `SetServiceStatus`. Technika ta ma jednak ograniczoną przydatność, ponieważ menedżer SCM otrzymuje okresowe aktualizacje.

Parametr `lpServStat` wskazuje strukturę `SERVICE_STATUS`, w której zapisywany jest aktualny stan usługi. Z tej samej struktury korzysta funkcja `SetServiceStatus`.

Sprawdzanie stanu usługi

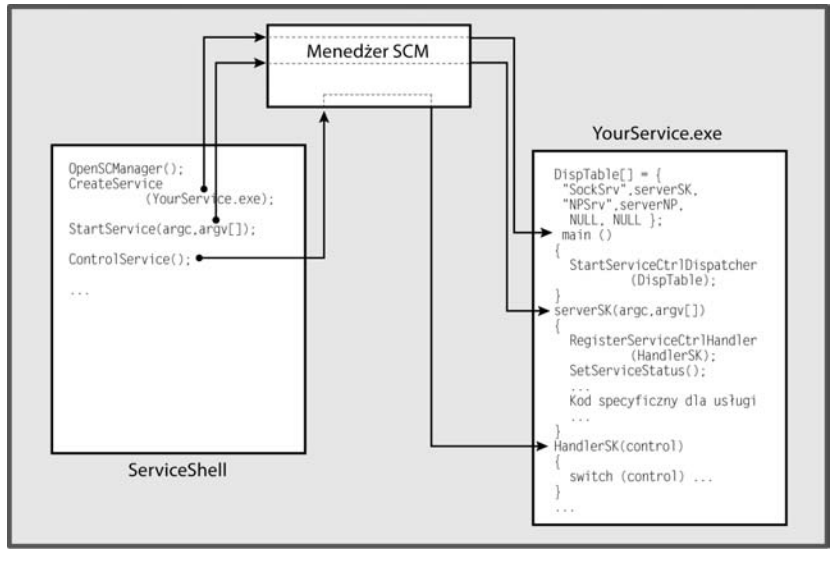
Aby pobrać aktualny stan usługi ze struktury `SERVICE_STATUS`, należy wywołać poniższą funkcję:

```
BOOL QueryServiceStatus (
   SC_HANDLE hService,
   LPSERVICE_STATUS lpServiceStatus)
```

Istnieje różnica między wywołaniem funkcji `QueryServiceStatus` (pobiera ona aktualne informacje o stanie od menedżera SCM) i wywołaniem funkcji `ControlService` z kodem sterowania `SERVICE_CONTROL_INTERROGATE`. Ta pierwsza nakazuje usłudze zaktualizować menedżer SCM, a nie aplikację.

Podsumowanie — działanie usług i zarządzanie nimi

Rysunek 13.1 przedstawia menedżer SCM i jego związki z usługami oraz programem sterującym usługi, takim jak narzędzie z listingu 13.3 z następnego podrozdziału. Usługa musi przede wszystkim zarejestrować się w menedżerze SCM, a wszystkie kierowane do niej polecenia przechodzą przez ten menedżer.



Rysunek 13.1. Kontrolowanie usług systemu Windows za pomocą menedżera SCM

Przykład — powłoka do sterowania usługą

Usługi systemu Windows można kontrolować po kliknięciu ikony *Usługi* w oknie *Narzędzia administracyjne*. Inna możliwość to sterowanie usługami za pomocą programu *sc.exe* uruchamianego w wierszu poleceń w systemie

Windows. Ponadto można sterować usługą z poziomu aplikacji, co ilustruje następny przykład — program *ServiceShell* (listing 13.3). Jest to zmodyfikowana wersja programu *JobShell* (listing 6.3) z rozdziału 6.

Listing 13.3. ServiceShell — program do sterowania usługami

```
/* Rozdział 13. */
/* ServiceShell.c – powłoka do zarządzania usługami systemu Windows.
   Jest to zmodyfikowana wersja programu do zarządzania zadaniami z rozdziału 6.
   Tu program zarządza usługami, a nie zadaniami. */
/* Ilustruje sterowanie usługą z poziomu programu.
   Zwykle należy korzystać z polecenia sc.exe lub narzędzia
   administracyjnego Usługi. */
/* Oto lista obsługiwanych poleceń:
        create   Tworzy usługę
        delete   Usuwa usługę
        start    Uruchamia usługę
        control  Kontroluje usługę */
#include "Everything.h"

static int Create (int, LPTSTR *, LPTSTR);
static int Delete (int, LPTSTR *, LPTSTR);
static int Start (int, LPTSTR *, LPTSTR);
static int Control (int, LPTSTR *, LPTSTR);

static SC_HANDLE hScm;
static BOOL debug;

int _tmain (int argc, LPTSTR argv[])
{
    BOOL exitFlag = FALSE;
    TCHAR command[MAX_COMMAND_LINE+10], *pc;
    DWORD i, locArgc; /* Lokalny argument argc. */
    TCHAR argstr[MAX_ARG][MAX_COMMAND_LINE];
    LPTSTR pArgs[MAX_ARG];

    debug = (argc > 1); /* Prosta flaga diagnostyczna. */
    /* Przygotowywanie lokalnej tablicy „argv" ze wskaźnikami do łańcuchów znaków. */
    for (i = 0; i < MAX_ARG; i++) pArgs[i] = argstr[i];

    /* Otwieranie na lokalnym komputerze menedżera SCM
       z domyślną bazą danych i pełnym dostępem. */
    hScm = OpenSCManager (NULL, SERVICES_ACTIVE_DATABASE,
                          SC_MANAGER_ALL_ACCESS);

    /* Główna pętla do przetwarzania polecenia. */
    _tprintf (_T("\nZarzadzanie uslugami systemu Windows."));
    while (!exitFlag) {
        _tprintf (_T("\nSM$"));
```

```
        _fgetts (command, MAX_COMMAND_LINE, stdin);
        /* Zastępowanie znaku nowego wiersza sekwencją końca łańcucha znaków. */
        pc = _tcschr (command, _T('\n')); *pc = _T('\0');

        if (debug) _tprintf (_T("%s\n"), command);
        /* Przekształcanie polecenia na postać „argc, argv". */
        GetArgs (command, &locArgc, pArgs);
        CharLower (argstr[0]); /* Wielkość znaków w poleceniu nie ma znaczenia. */

        if (debug) _tprintf (_T("\n%s %s %s %s"), argstr[0],
        ↪argstr[1],
                             argstr[2], argstr[3]);

        if (_tcscmp (argstr[0], _T("create")) == 0) {
            Create (locArgc, pArgs, command);
        }
        else if (_tcscmp (argstr[0], _T("delete")) == 0) {
            Delete (locArgc, pArgs, command);
        }
        else if (_tcscmp (argstr[0], _T("start")) == 0) {
            Start (locArgc, pArgs, command);
        }
        else if (_tcscmp (argstr[0], _T("control")) == 0) {
            Control (locArgc, pArgs, command);
        }
        else if (_tcscmp (argstr[0], _T("quit")) == 0) {
            exitFlag = TRUE;
        }
        else _tprintf (_T("\nNieznane polecenie."));
    }

    CloseServiceHandle (hScm);
    return 0;
}

int Create (int argc, LPTSTR argv[], LPTSTR command)
{
    /* Tworzenie nowej usługi z ustawieniem SERVICE_DEMAND_START:
       argv[1]: nazwa usługi
       argv[2]: wyświetlana nazwa
       argv[3]: binarny plik wykonywalny */

    SC_HANDLE hSc;
    TCHAR executable[MAX_PATH+1],
        quotedExecutable[MAX_PATH+3] = _T("\"");

    /* Potrzebna jest pełna ścieżka i – jeśli występują odstępy – cudzysłowy. */
    GetFullPathName (argv[3], MAX_PATH+1, executable, NULL);
    _tcscat(quotedExecutable, executable);
    _tcscat(quotedExecutable, _T("\""));
```

```
    if (debug) _tprintf (_T("\nPelna sciezka do uslugi: %s"),
                         executable);

    hSc = CreateService (hScm, argv[1], argv[2],
       SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS,
       SERVICE_DEMAND_START, SERVICE_ERROR_NORMAL,
       quotedExecutable, NULL, NULL, NULL, NULL, NULL);
    CloseServiceHandle (hSc); /* Nie trzeba zachowywać uchwytu, ponieważ
                              funkcja OpenService pobiera informacje z bazy danych usługi. */
    return 0;
}

/* Usuwanie usługi
      argv[1]: nazwa usuwanej usługi */
int Delete (int argc, LPTSTR argv[], LPTSTR command)
{
    SC_HANDLE hSc;

    if (debug) _tprintf (_T("\nPrzed usunieciem uslugi: %s"),
    ↪argv[1]);
    hSc = OpenService(hScm, argv[1], DELETE);
    DeleteService (hSc);
    CloseServiceHandle (hSc);
    return 0;
}

/* Uruchamianie nazwanej usługi
      argv[1]: nazwa uruchamianej usługi */
int Start (int argc, LPTSTR argv[], LPTSTR command)
{
    SC_HANDLE hSc;
    TCHAR workingDir[MAX_PATH+1];
    LPTSTR argvStart[] = {argv[1], workingDir};

    GetCurrentDirectory (MAX_PATH+1, workingDir);

    /* Pobieranie uchwytu usługi podanej w wierszu poleceń (argv[1]). */
    hSc = OpenService(hScm, argv[1], SERVICE_ALL_ACCESS);

    /* Uruchamia usługę z jednym argumentem – katalogiem roboczym. */
    /* Nazwa usługi pochodzi z wiersza poleceń programu (argv[1]). */
    StartService (hSc, 2, argvStart);

    CloseServiceHandle (hSc);

    return 0;
}

/* Sterowanie nazwaną usługą.
      argv[1]: nazwa kontrolowanej usługi
      argv[2]: polecenie sterujące (wielkość znaków nie ma znaczenia):
```

```
                stop
                pause
                resume
                interrogate
                user - zdefiniowane przez użytkownika
                */
static LPCTSTR commandList[] =
    { _T("stop"), _T("pause"), _T("resume"),
      _T("interrogate"), _T("user") };
static DWORD controlsAccepted[] = {
    SERVICE_CONTROL_STOP, SERVICE_CONTROL_PAUSE,
    SERVICE_CONTROL_CONTINUE, SERVICE_CONTROL_INTERROGATE, 128 };

int Control (int argc, LPTSTR argv[], LPTSTR command)
{
    SC_HANDLE hSc;
    SERVICE_STATUS sStatus;
    DWORD dwControl, i;
    BOOL found = FALSE;

    if (debug) _tprintf (_T("\nSterowanie usluga: %s"), argv[1]);

    for (i= 0;
           i < sizeof(controlsAccepted)/sizeof(DWORD) && !found; i++)
        found = (_tcscmp (commandList[i], argv[2]) == 0);
    if (!found) {
        _tprintf (_T("\nNiedozwolone polecenie %s"), argv[1]);
        return 1;
    }
    dwControl = controlsAccepted[i-1];
    if (dwControl == 128) dwControl = _ttoi (argv[3]);
    if (debug) _tprintf (_T("\ndwControl = %d"), dwControl);

    hSc = OpenService(hScm, argv[1],
       SERVICE_INTERROGATE | SERVICE_PAUSE_CONTINUE |
       SERVICE_STOP | SERVICE_USER_DEFINED_CONTROL |
       SERVICE_QUERY_STATUS );

    ControlService (hSc, dwControl, &sStatus);

    if (dwControl == SERVICE_CONTROL_INTERROGATE) {
        QueryServiceStatus (hSc, &sStatus);

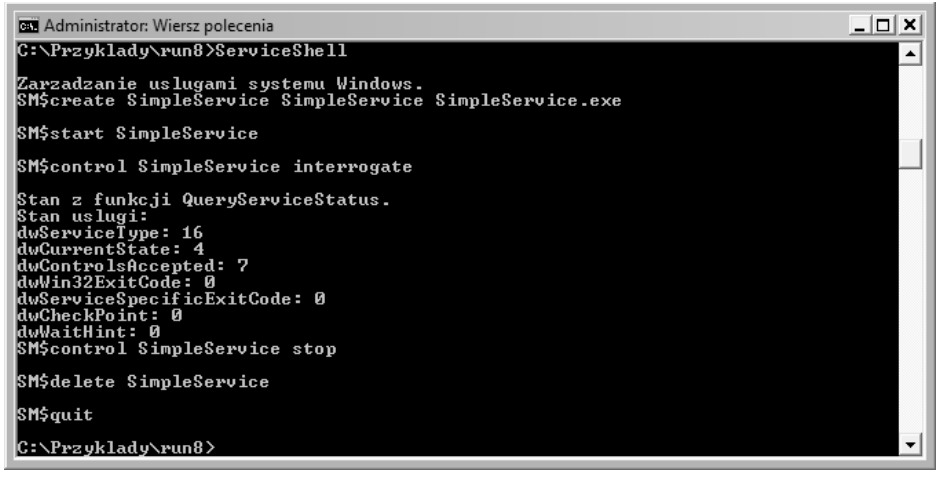
        _tprintf (_T("\nStan z funkcji QueryServiceStatus."));
        _tprintf (_T("\nStan uslugi:"));
        _tprintf (_T("\ndwServiceType: %d"), sStatus.dwServiceType);
        _tprintf (_T("\ndwCurrentState: %d"),
       ↪sStatus.dwCurrentState);
        _tprintf (_T("\ndwControlsAccepted: %d"),
       ↪sStatus.dwControlsAccepted);
```

```
        _tprintf (_T("\ndwWin32ExitCode: %d"),
        ↪sStatus.dwWin32ExitCode);
        _tprintf (_T("\ndwServiceSpecificExitCode: %d"),
                  sStatus.dwServiceSpecificExitCode);
        _tprintf (_T("\ndwExitCode: %d"), sStatus.dwCheckPoint);
        _tprintf (_T("\ndwWaitHint: %d"), sStatus.dwWaitHint);

    }
    if (hSc != NULL) CloseServiceHandle (hSc);
    return 0;
}
```

Ten przykład ma pokazać, jak kontrolować usługi z poziomu programu. Nie zastępuje on programu *sc.exe* ani narzędzia administracyjnego *Usługi*.

Przebieg programu 13.3 przedstawia działanie funkcji `SimpleService`.



Przebieg programu 13.3. Aplikacja ServiceShell — zarządzanie usługami

Współużytkowanie obiektów jądra przy użyciu usługi

Czasem usługa i aplikacje współużytkują obiekt jądra. Usługa może na przykład korzystać z nazwanego muteksu do zabezpieczania współużytkowanego obszaru pamięci stosowanego do komunikacji z aplikacjami. Ponadto w tym kontekście odwzorowanie plików także jest współużytkowanym obiektem jądra.

Występuje tu problem wynikający z tego, że aplikacje pracują w innym kontekście zabezpieczeń niż usługi, które mogą działać na koncie systemowym. Nawet jeśli ochrona nie jest potrzebna, nie należy tworzyć i (lub) otwierać współużytkowanych obiektów jądra ze wskaźnikiem atrybutów zabezpieczeń ustawionym na `NULL` (zobacz rozdział 15.). Zamiast tego potrzebna jest przynajmniej różna od `NULL` poufna lista kontroli dostępu. Oznacza to, że aplikacje i usługa muszą korzystać z różnej od `NULL` struktury atrybutów zabezpieczeń. Zwykle warto zabezpieczać obiekty, a zagadnienie to jest tematem rozdziału 15.

Warto też zauważyć, że jeśli usługa działa na koncie systemowym, może mieć trudności z dostępem do zasobów z innych maszyn, na przykład do współużytkowanych plików.

Uwagi na temat diagnozowania usług

Usługa ma działać ciągle, dlatego musi być niezawodna i w miarę możliwości wolna od defektów. Choć usługę można dołączyć do debugera i użyć dzienników zdarzeń do śledzenia jej działania, te techniki najlepiej jest stosować po zainstalowaniu usługi.

W fazie rozwijania i diagnozowania kodu często najłatwiej jest zastosować nakładkę na usługi zaprezentowaną na listingu 13.2. Umożliwia ona uruchomienie programu jako usługi lub niezależnej aplikacji. Wybór odbywa się za pomocą opcji `-c` podawanej w wierszu poleceń.

- Najpierw należy rozwinąć wersję „przedusługową" w postaci niezależnego programu. W ten sposób zbudowano na przykład program *serverSK*.
- Następnie należy dopracować program za pomocą rejestrowania zdarzeń lub pliku dziennika.
- Po stwierdzeniu, że program jest gotowy do zainstalowania jako usługa, należy uruchomić go bez podawania w wierszu poleceń opcji `-c`. Program zostanie uruchomiony jako usługa.
- Niezbędne są dodatkowe testy usługi, aby wykryć błędy logiczne i problemy z bezpieczeństwem. Usługi mogą działać na koncie systemowym i nie zawsze mają dostęp do obiektów użytkownika, a wersja niezależna czasem nie umożliwia wykrycia takich problemów.
- Normalne diagnozowanie zdarzeń i drobne prace konserwacyjne można przeprowadzić na podstawie informacji z pliku dziennika lub

dziennika zdarzeń. Nawet informacje o stanie mogą pomóc w określeniu sprawności serwera i wykryciu symptomów usterek.

- Jeśli konieczne są skomplikowane prace konserwacyjne, można zdiagnozować program jako zwykłą aplikację za pomocą opcji `-c`.

Podsumowanie

Usługi systemu Windows udostępniają standardowe możliwości w zakresie dodawania utworzonych przez użytkownika usług do komputerów z system Windows. Gotowy niezależny program można przekształcić na usługę za pomocą metod opisanych w tym rozdziale.

Usługi można tworzyć, sterować nimi i obserwować ich działanie za pomocą narzędzi administracyjnych lub programu *ServiceShell* zaprezentowanego w tym rozdziale. Menedżer SCM kontroluje zainstalowane usługi i śledzi ich działanie. W rejestrze znajdują się wpisy powiązane z wszystkimi usługami.

Co dalej?

W rozdziale 14. opisano asynchroniczne operacje wejścia-wyjścia. Obejmują one dwie techniki do wykonywania wielu operacji odczytu i zapisu równolegle z przetwarzaniem danych. Nie trzeba przy tym korzystać z wątków — potrzebny jest tylko jeden wątek użytkownika.

W większości przypadków łatwiej jest pisać programy za pomocą wielu wątków niż asynchronicznych operacji wejścia-wyjścia, a wydajność rozwiązań opartych na wątkach jest zwykle wyższa. Jednak asynchroniczne operacje wejścia-wyjścia są niezbędne przy stosowaniu portów kończenia operacji wejścia-wyjścia. Porty te są niezwykle przydatne przy budowaniu skalowalnych serwerów, które mogą obsługiwać dużą liczbę klientów.

W rozdziale 14. omówiono też zegary oczekujące.

Lektura dodatkowa

W książce *Professional NT Services* Kevin Miller dokładnie opisał usługi. W niniejszym rozdziale nie omówiono sterowników urządzeń i ich interakcji z usługami. Informacje na te tematy można znaleźć na przykład w książce *Programming the Microsoft Windows Driver Model, Second Edition* Waltera Oneya.

Ćwiczenia

13.1. Zmodyfikuj program *SimpleService* (listing 13.2) przez zastosowanie zdarzeń systemu Windows zamiast pliku dziennika. Potrzebne będą funkcje `RegisterEventSource`, `ReportEvent` i `DeregisterEventSource` (każdą z nich opisano w dokumentacji MSDN). Rozważ ponadto wykorzystanie rejestrowania zdarzeń w systemie Vista. Inna możliwość to zastosowanie systemu rejestrowania o otwartym dostępie do kodu źródłowego, takiego jak Nlog (*http://nlogproject.org/home*).

13.2. Rozbuduj program *serviceSK*, tak aby w sensowny sposób przyjmował polecenie wstrzymania. Oto sugestia — wstrzymana usługa powinna zachowywać istniejące połączenia, ale nie może przyjmować nowych połączeń. Ponadto usługa powinna zakończyć przetwarzanie odebranych żądań i wysłać odpowiedzi, jednak nie można akceptować w niej dalszych żądań od klientów.

13.3. Program *ServiceShell* przy sprawdzaniu stanu usługi po prostu wyświetla liczby. Rozwiń ten program w taki sposób, aby przedstawiał stan w bardziej czytelnej postaci.

13.4. Przekształć program *serverNP* (listing 12.3) na usługę.

13.5. Przetestuj program *serviceSK* z pliku *Przykłady*. Zmodyfikuj go, tak aby korzystał z rejestrowania zdarzeń.

PROGRAMOWANIE W SYSTEMIE WINDOWS

WYDANIE IV

THE ADDISON-WESLEY Microsoft Technology Series

Wybierając system Windows jako docelową platformę rozwijanych aplikacji, programiści na całym świecie sugerują się najczęściej jego dużą funkcjonalnością i wymogami biznesowymi. System ten jest bowiem zgodny z wieloma kluczowymi standardami. Obsługuje między innymi biblioteki standardowe języków C i C+ oraz uwzględnia wiele otwartych standardów współdziałania. Dlatego gniazda systemu Windows są standardowym interfejsem programowania rozwiązań sieciowych z dostępem do TCP/IP i innych protokołów sieciowych. W dodatku każda nowa wersja tego systemu jest coraz bardziej zintegrowana z dodatkowymi technologiami z obszaru multimediów, sieci bezprzewodowych, usług Web Service, platformy .NET i usługi plug-and-play. Niewątpliwym atutem Windowsa jest także zawsze uważany za stabilny, a jednak ciągle wzbogacany o ważne dodatki interfejs API.

Jeśli zatem szukasz kompletnego, rzetelnego i aktualnego podręcznika do nauki programowania za pomocą interfejsu Windows API, właśnie go znalazłeś! Książka ta w praktyczny sposób przedstawia wszystkie mechanizmy systemu Windows potrzebne programistom, pokazując, w jaki sposób działają funkcje tego systemu i jak wchodzą w interakcje z aplikacjami. Skoncentrowano się tu na podstawowych usługach systemu, w tym na systemie plików, zarządzaniu procesami i wątkami, komunikacji między procesami, programowaniu sieciowym i synchronizacji. Autor tej książki nie zamierza jednak obciążać Cię zbędną teorią i nieistotnymi szczegółami. Podaje Ci wiedzę opartą na prawdziwych przykładach, dzięki czemu szybko i sprawnie opanujesz poruszane tu zagadnienia. Wiadomości, które tu znajdziesz, pozwolą Ci zrozumieć interfejs Windows API w takim stopniu, byś zdobył solidne podstawy do rozwijania programów na platformę .NET Microsoftu.

Johnson M. Hart – dłużej niż ćwierć wieku pracuje jako inżynier oprogramowania, menedżer i dyrektor techniczny w takich firmach, jak Hewlett-Packard czy Apollo Komputer. Przez prawie dziesięć lat wykładał nauki komputerowe na Uniwersytecie Kentucky w USA. Obecnie jest konsultantem w dziedzinie rozwijania aplikacji dla systemu Microsoft Windows i platformy .NET oraz systemów o otwartym dostępie do kodu źródłowego. Jest również uznanym szkoleniowcem i autorem wielu artykułów na temat inżynierii oprogramowania.

W książce znajdziesz omówienie między innymi takich kwestii, jak:

- wprowadzenie do rodziny systemów operacyjnych Windows
- wykorzystanie paralelizmu i maksymalizowanie wydajności w systemach wielordzeniowych
- używanie 64-bitowej przestrzeni adresowej oraz zapewnianie przenośności między środowiskami 64- i 32-bitowymi
- zagadnienia związane z systemami plików, operacjami wejścia-wyjścia w konsoli, blokowaniem dostępu do plików i zarządzaniem katalogami
- wprowadzenie do obsługi wyjątków w systemie Windows, w tym do mechanizmu SEH
- zarządzanie i synchronizacja procesów systemu Windows
- zarządzanie pamięcią i biblioteki DLL
- szczegółowe omówienie synchronizacji wątków systemu Windows, pul wątków i wydajności
- przekształcanie aplikacji serwerowych na usługi systemu Windows
- zapewnianie przenośności kodu źródłowego oraz współdziałania aplikacji z systemów Windows, Linux i UNIX
- zabezpieczanie obiektów w systemie Windows
- poprawiające wydajność funkcje interfejsu Windows API – blokady SRW i zmienne warunkowe

Oto kompletny, aktualny przewodnik po programowaniu przy użyciu interfejsu Windows API!

helion.pl księgarnia internetowa

Cena 109,00 zł

Nr katalogowy: 5720

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900, 0 601 339900

Wydawnictwo Helion
ul. Kościuszki 1c, 44-100 Gliwice
44-100 Gliwice, skr. poczt. 462
32 230 98 63
http://helion.pl
e-mail: helion@helion.pl

ISBN 978-83-246-2780-6



Informatyka w najlepszym wydaniu