Microsoft®

Windows®
via C/C++

Wydanie V

Jeffrey Richter (Wintellect)
Christophe Nasarre

Windows® via C/C++
Edycja polska Microsoft Press
Original English language edition © 2008 by Jeffrey Richter and Christophe Nasarre
Tytuł oryginału: Windows® via C/C++

Polish edition by APN PROMISE Sp. z o. o. Warszawa 2007

APN PROMISE Sp. z o. o., biuro: 00-108 Warszawa, ul. Zielna 39
tel. (022) 351 90 00, faks (022) 351 90 99
e-mail: mspress@promise.pl

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiejkolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

Microsoft, Microsoft Press, Active Directory, Excel, InfoPath, IntelliSense, Internet Explorer, MSDN, Outlook, PowerPoint, SharePoint, SQL Server, Visual Basic, Visual Studio, Windows oraz Windows Server są zarejestrowanymi znakami towarowymi Microsoft Corporation.

Wszystkie inne nazwy handlowe i towarowe występujące w niniejszej publikacji mogą być znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm odnośnych właścicieli.

Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń. Ewentualne podobieństwo do jakiejkolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

APN PROMISE Sp. z o. o. dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji. APN PROMISE Sp. z o. o. nie jest w żadnym wypadku odpowiedzialna za jakiekolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-023-5

Przekład: Jakub Niedźwiedź
Redakcja: Marek Włodarz
Korekta: Anna Wojdanowicz
Skład i łamanie: Marek Włodarz

Spis treści

Część I: Lektura obowiązkowa . . . 1

1 Obsługa błędów . . . 3

Definiowanie własnych kodów błędów . . . 7
Aplikacja przykładowa ErrorShow . . . 8

2 Praca ze znakami i łańcuchami znaków . . . 11

Kodowanie znaków . . . 12
Znakowe i łańcuchowe typy danych ANSI i Unicode . . . 14
Funkcje Unicode i ANSI w Windows . . . 16
Funkcje Unicode i ANSI w bibliotece uruchomieniowej C . . . 18
Bezpieczne funkcje łańcuchowe w bibliotece uruchomieniowej C . . . 19
Wprowadzenie do nowych, bezpiecznych funkcji łańcuchowych . . . 20
Jak uzyskać więcej kontroli przy wykonywaniu operacji na łańcuchach? . . . 23
Funkcje łańcuchowe Windows . . . 25
Dlaczego należy korzystać z Unicode? . . . 27
Jak zalecamy pracować ze znakami i łańcuchami? . . . 28
Tłumaczenie łańcuchów pomiędzy Unicode a ANSI . . . 29
Eksportowanie funkcji ANSI i Unicode z bibliotek DLL . . . 31
Określanie, czy dany tekst jest w standardzie ANSI, czy Unicode . . . 33

3 Obiekty jądra . . . 35

Czym jest obiekt jądra? . . . 35
Zliczanie użyć . . . 37
Bezpieczeństwo . . . 37
Tabela uchwytów obiektów jądra dla danego procesu . . . 39
Tworzenie obiektu jądra . . . 40
Zamykanie obiektu jądra . . . 41
Współdzielenie obiektów jądra pomiędzy granicami procesów . . . 45
Korzystanie z dziedziczenia uchwytów do obiektów . . . 46
Nadawanie nazw obiektom . . . 50
Duplikowanie uchwytów do obiektów . . . 62

Część II: Realizacja zadań . . . 67

4 Procesy . . . 69

Pisanie pierwszej aplikacji dla Windows . . . 70
Uchwyt do instancji procesu . . . 75
Funkcja CreateProcess . . . 91

pszApplicationName i *pszCommandLine* 92
Kończenie procesu. 107
Koniec działania funkcji początkowej głównego wątku 107
Funkcja *ExitProcess* 108
Funkcja *TerminateProcess* 109
Gdy przestaną istnieć wszystkie wątki procesu 110
Gdy proces zakończy swoje działanie 110
Procesy potomne. 111
Uruchamianie osobnych procesów potomnych. 113
Gdy administrator działa z uprawnieniami zwykłego użytkownika. 113
Automatyczne podnoszenie uprawnień procesu 117
Ręczne podnoszenie uprawnień procesu 119
Jaki jest bieżący kontekst uprawnień?. 120
Wyliczanie procesów działających w systemie 122

5 Zadania 131

Nakładanie ograniczeń na procesy należące do zadania. 135
Umieszczanie procesu wewnątrz zadania 143
Kończenie wszystkich procesów należących do zadania 144
Pozyskiwanie statystyk dotyczących zadania 144
Powiadomienia o zadaniach. 147
Aplikacja przykładowa Job Lab 150

6 Podstawy wątków 153

Kiedy tworzyć wątek? 154
Kiedy nie tworzyć wątku 156
Pisanie funkcji początkowej nowego wątku 157
Funkcja *CreateThread* 158
psa 159
cbStackSize 159
pfnStartAddr i *pvParam* 160
dwCreateFlags. 161
pdwThreadID. 161
Kończenie wątku 162
Funkcja wątkowa zwraca wartość 162
Funkcja *ExitThread* 162
Funkcja *TerminateThread* 163
Gdy proces kończy działanie 164
Gdy wątek kończy działanie. 164
Wątki od środka. 165
Uwarunkowania biblioteki uruchomieniowej C/C++ 167
Ojej! przez pomyłkę wywołałem *CreateThread* zamiast *_beginthreadex* 176
Funkcje biblioteki uruchomieniowej C/C++, których nie powinno się nigdy wywoływać. 177
Zyskiwanie poczucia własnej tożsamości. 178
Konwertowanie pseudo-uchwytu na prawdziwy uchwyt 178

7 Planowanie wątków, priorytety i koligacje 181

Wstrzymywanie i wznawianie wątku 183
Wstrzymywanie i wznawianie procesu. 184
Usypianie 186
Przełączanie na inny wątek. 186
Przełączanie na inny wątek na procesorze z hiperwątkowością 187
Czasy wykonywania wątku 187
CONTEXT we właściwym kontekście 191
Priorytety wątków 195
Abstrakcyjne spojrzenie na priorytety 196
Programowanie priorytetów. 200
  Dynamiczne zwiększanie poziomów priorytetów wątków 203
  Dostrajanie planowania wątków dla procesu pierwszoplanowego 204
  Planowanie priorytetów dla żądań wejścia/wyjścia 205
  Aplikacja przykładowa Scheduling Lab. 206
Koligacje. 212

8 Synchronizacja wątków w trybie użytkownika. 217

Dostęp atomowy: rodzina funkcji blokujących 218
Wiersze pamięci podręcznej. 224
Zaawansowana synchronizacja wątków. 225
  Technika, której należy unikać 226
Sekcje krytyczne 228
  Sekcje krytyczne: uwagi drobnym drukiem 230
  Sekcje krytyczne i blokady pętlowe 233
  Sekcje krytyczne i obsługa błędów 233
Ograniczone blokady odczytu-zapisu 235
Zmienne warunkowe. 238
  Przykładowa aplikacja Queue 239
  Użyteczne wskazówki i techniki. 249

9 Synchronizacja wątków przy pomocy obiektów jądra 253

Funkcje oczekiwania 255
Efekty uboczne udanego oczekiwania. 258
Obiekty jądra dla zdarzeń. 260
  Aplikacja przykładowa Handshake 264
Obiekty jądra dla zegarów 268
  Kolejkowanie wywołań APC przez zegary 272
  Brakujące szczegóły dotyczące zegarów 273
Obiekty jądra dla semaforów 274
Obiekty jądra dla muteksów. 277
  Sprawy związane z porzucaniem. 279
  Muteksy a sekcje krytyczne 280
  Aplikacja przykładowa Queue 280
Wygodne zestawienie obiektów do synchronizacji wątków 287
Inne funkcje synchronizujące wątki 287

Asynchroniczne, sprzętowe wejście/wyjście 288
WaitForInputIdle 288
MsgWaitForMultipleObjects(Ex) 290
WaitForDebugEvent 290
SignalObjectAndWait 291
Wykrywanie zakleszczeń przy użyciu Wait Chain Traversal API 292

10 Synchroniczne i asynchroniczne operacje wejścia/wyjścia 301

Otwieranie i zamykanie urządzeń 302
Szczegółowe spojrzenie na *CreateFile* 305
Praca z urządzeniami plikowymi 312
Pobieranie rozmiaru pliku 313
Ustawianie wskaźnika pliku 314
Ustawianie końca pliku. 316
Wykonywanie synchronicznego, sprzętowego wejścia/wyjścia 316
Przerzucanie danych do urządzenia 317
Anulowanie synchronicznego wejścia/wyjścia 317
Podstawy asynchronicznego, sprzętowego wejścia/wyjścia. 319
Struktura OVERLAPPED 320
Zastrzeżenia do asynchronicznego, sprzętowego wejścia/wyjścia 322
Anulowanie zakolejkowanych żądań sprzętowego wejścia/wyjścia 324
Otrzymywanie powiadomień o zakończonym żądaniu wejścia/wyjścia 325
Sygnalizowanie obiektu jądra dla urządzenia 326
Sygnalizowanie obiektu jądra dla zdarzenia 327
Powiadamialne wejście/wyjście 330
Porty zakończeń wejścia/wyjścia 335

11 Pula wątków Windows. 355

Scenariusz 1: Asynchroniczne wywoływanie funkcji 356
Jawne sterowanie zadaniem 357
Aplikacja przykładowa Batch 358
Scenariusz 2: Wywoływanie funkcji w określonych odstępach czasu 362
Aplikacja przykładowa Timed Message Box 364
Scenariusz 3: Wywoływanie funkcji, gdy zasygnalizowany zostanie pojedynczy obiekt jądra 367
Scenariusz 4: Wywoływanie funkcji, gdy zakończy się żądanie asynchronicznego wejścia/wyjścia 369
Działania wykonywane po zakończeniu funkcji zwrotnej. 371
Niestandardowe pule wątków 372
Eleganckie niszczenie puli wątków: grupy porządkowe 375

12 Włókna. 377

Praca z włóknami. 377
Aplikacja przykładowa Counter 381

Część III: **Zarządzanie pamięcią** . . . 385

13 Architektura pamięci Windows . . . 387

Wirtualna przestrzeń adresowa procesu . . . 387
Jak jest podzielona wirtualna przestrzeń adresowa? . . . 388
Partycja przypisywania pustego wskaźnika . . . 389
Partycja trybu użytkownika . . . 389
Partycja trybu jądra . . . 392
Regiony w przestrzeni adresowej . . . 392
Przydzielanie fizycznej pamięci dla regionu . . . 393
Pamięć fizyczna a plik wymiany . . . 394
Pamięć fizyczna nieutrzymywana w pliku wymiany . . . 396
Atrybuty ochrony . . . 398
Dostęp z kopiowaniem przy zapisie . . . 399
Flagi atrybutów ochrony dla specjalnego dostępu . . . 400
Zebranie wszystkiego razem . . . 400
Regiony od środka . . . 406
Znaczenie wyrównywania danych . . . 410

14 Badanie pamięci wirtualnej . . . 415

Informacje systemowe . . . 415
Aplikacja przykładowa System Information . . . 418
Stan pamięci wirtualnej . . . 424
Zarządzanie pamięcią na maszynach NUMA . . . 425
Aplikacja przykładowa Virtual Memory Status . . . 426
Określanie stanu przestrzeni adresowej . . . 428
Funkcja *VMQuery* . . . 430
Aplikacja przykładowa Virtual Memory Map . . . 435

15 Korzystanie z pamięci wirtualnej we własnych aplikacjach . . . 439

Rezerwowanie regionu w przestrzeni adresowej . . . 439
Przydzielanie pamięci w zarezerwowanym regionie . . . 442
Jednoczesne rezerwowanie regionu i przydzielanie pamięci . . . 442
Kiedy przydzielać pamięć fizyczną? . . . 444
Oddzielanie pamięci fizycznej i zwalnianie regionu . . . 446
Kiedy oddzielać pamięć fizyczną? . . . 447
Aplikacja przykładowa Virtual Memory Allocation . . . 448
Zmienianie atrybutów ochrony . . . 454
Odnawianie zawartości pamięci fizycznej . . . 455
Aplikacja przykładowa MemReset . . . 456
Rozszerzenia okien adresowych . . . 459
Aplikacja przykładowa AWE . . . 462

16 Stos wątku . . . 469

Funkcja sprawdzania stosu z biblioteki uruchomieniowej C/C++ . . . 473
Aplikacja przykładowa Summation . . . 475

17 Pliki mapowane w pamięci 481
Mapowane w pamięci pliki wykonywalne i biblioteki DLL 482
Dane statyczne nie są współdzielone przez wiele instancji pliku wykonywalnego lub biblioteki DLL 483
Mapowane w pamięci pliki danych 493
Metoda 1: Jeden plik, jeden bufor 493
Metoda 2: Dwa pliki, jeden bufor 494
Metoda 3: Jeden plik, dwa bufory 494
Metoda 4: Jeden plik, zero buforów 495
Korzystanie z plików mapowanych w pamięci 495
Krok 1: Tworzenie lub otwieranie obiektu jądra dla pliku 495
Krok 2: Tworzenie obiektu jądra dla mapowania pliku 497
Krok 3: Mapowanie danych pliku do przestrzeni adresowej procesu 500
Krok 4: Odłączanie mapowania danych pliku od przestrzeni adresowej procesu 503
Kroki 5 i 6: Zamykanie obiektu mapowania pliku i obiektu pliku 505
Aplikacja przykładowa File Reverse 506
Przetwarzanie dużego pliku przy użyciu plików mapowanych w pamięci 512
Pliki mapowane w pamięci a spójność 513
Określanie adresu bazowego pliku mapowanego w pamięci 514
Szczegóły implementacyjne plików mapowanych w pamięci 515
Korzystanie z plików mapowanych w pamięci do współdzielenia danych pomiędzy procesami 517
Pliki mapowane w pamięci wspierane przez plik wymiany 518
Aplikacja przykładowa Memory-Mapped File Sharing 519
Rzadko przydzielane pliki mapowane w pamięci 522
Aplikacja przykładowa MMF Sparse 524

18 Sterty 535
Domyślna sterta procesu 535
Powody tworzenia dodatkowych stert 536
Ochrona składników 537
Efektywniejsze zarządzanie pamięcią 537
Dostęp lokalny 538
Unikanie kosztów synchronizacji wątków 539
Szybkie zwalnianie pamięci 539
Jak utworzyć dodatkową stertę? 539
Alokowanie bloku pamięci ze sterty 541
Zmienianie rozmiaru bloku 543
Otrzymywanie rozmiaru bloku 544
Zwalnianie bloku 544
Niszczenie sterty 544
Używanie stert w języku C++ 545
Rozmaite funkcje sterty 548

Część IV: Dynamicznie dołączane biblioteki 551

19 Podstawy DLL 553

Biblioteki DLL a przestrzeń adresowa procesu 554
Ogólny obraz sytuacji 556
  Budowanie modułu DLL 559
  Budowanie modułu wykonywalnego 564
  Uruchamianie modułu wykonywalnego 567

20 Zaawansowane techniki DLL 571

Jawne ładowanie modułu DLL i dołączanie symboli 571
  Jawne ładowanie modułu DLL 573
  Jawne wyładowywanie modułu DLL 576
  Jawne dołączanie do wyeksportowanego symbolu 579
Funkcja wejściowa biblioteki 580
  Powiadomienie DLL_PROCESS_ATTACH 581
  Powiadomienie DLL_PROCESS_DETACH 582
  Powiadomienie DLL_THREAD_ATTACH 585
  Powiadomienie DLL_THREAD_DETACH 586
  Szeregowane wywołania *DllMain* 587
  DllMain a biblioteka uruchomieniowa C/C++ 589
Ładowanie biblioteki DLL z opóźnieniem 590
  Aplikacja przykładowa DelayLoadApp 596
Przekierowania funkcji 602
Znane biblioteki DLL 602
Przekierowywanie bibliotek DLL 604
Zmienianie adresów bazowych modułów 605
Wiązanie modułów 611

21 Pamięć lokalna dla wątku 615

Dynamiczna pamięć TLS 616
  Korzystanie z dynamicznej pamięci TLS 618
Statyczna pamięć TLS 620

22 Wszczepianie bibliotek DLL i podczepianie API 623

Wszczepianie biblioteki DLL: przykład 624
Wszczepianie biblioteki DLL przy użyciu rejestru 626
Wszczepianie biblioteki DLL przy użyciu haków Windows 627
  Narzędzie Desktop Item Position Saver (DIPS) 629
Wszczepianie biblioteki DLL przy użyciu zdalnych wątków 638
  Aplikacja przykładowa Inject Library 642
  Biblioteka DLL Image Walk 647
Wszczepianie biblioteki DLL przy użyciu konia trojańskiego 649
Wszczepianie biblioteki DLL przez debuger 650
Wszczepianie kodu przy użyciu *CreateProcess* 650
Podczepianie API: przykład 651

Podczepianie API poprzez nadpisywanie kodu . . . 652
Podczepianie API poprzez manipulowanie sekcją importu modułu . . . 653
Aplikacja przykładowa Last MessageBox Info . . . 656

Część V: Strukturalna obsługa wyjątków . . . 671

23 Procedury obsługi sytuacji krańcowych . . . 673

Zrozumienie procedur obsługi sytuacji krańcowych poprzez przykład . . . 674
Funcenstein1 . . . 675
Funcenstein2 . . . 675
Funcenstein3 . . . 677
Funcfurter1 . . . 678
Czas na Quiz: *FuncaDoodleDoo* . . . 679
Funcenstein4 . . . 680
Funcarama1 . . . 681
Funcarama2 . . . 682
Funcarama3 . . . 682
Funcarama4: ostatnia granica . . . 683
Uwagi na temat bloku *finally* . . . 685
Funcfurter2 . . . 686
Aplikacja przykładowa SEH Termination . . . 687

24 Procedury obsługi wyjątków i wyjątki programowe . . . 693

Zrozumienie filtrów wyjątków i procedur obsługi wyjątków poprzez przykład . . . 694
Funcmeister1 . . . 694
Funcmeister2 . . . 694
EXCEPTION_EXECUTE_HANDLER . . . 697
Kilka użytecznych przykładów . . . 698
Globalne odkręcanie . . . 701
Zatrzymywanie globalnego odkręcania . . . 704
EXCEPTION_CONTINUE_EXECUTION . . . 705
Ostrożne korzystanie z EXCEPTION_CONTINUE_EXECUTION . . . 706
EXCEPTION_CONTINUE_SEARCH . . . 707
GetExceptionCode . . . 709
Wyjątki związane z pamięcią . . . 709
Wyjątki związane z wyjątkami . . . 710
Wyjątki związane z debugowaniem . . . 710
Wyjątki związane z liczbami całkowitymi . . . 710
Wyjątki związane z liczbami zmiennoprzecinkowymi . . . 710
GetExceptionInformation . . . 713
Wyjątki programowe . . . 717

25 Nieobsłużone wyjątki, ukierunkowana obsługa wyjątków i wyjątki C++ . . . 721

Wewnątrz funkcji *UnhandledExceptionFilter* . . . 724

Działanie numer 1: pozwalanie na zapisanie zasobu i kontynuowanie wykonania . . . 724
Działanie numer 2: powiadamianie debugera o nieobsłużonym wyjątku . . . 724
Działanie numer 3: powiadamianie globalnie ustawionej funkcji filtra . . . 724
Działanie numer 4: powiadamianie debugera o nieobsłużonym wyjątku (znowu) . . . 725
Działanie numer 5: ciche zakończenie procesu . . . 725
UnhandledExceptionFilter i interakcja z usługą WER . . . 726
Debugowanie dokładnie na czas . . . 730
Aplikacja przykładowa Spreadsheet . . . 733
Ukierunkowany wyjątek i procedury obsługi kontynuacji . . . 742
Wyjątki C++ a wyjątki strukturalne . . . 743
Wyjątki i debuger . . . 745

26 Raportowanie błędów i przywracanie aplikacji do normalnego stanu . . . 749

Konsola raportowania błędów systemu Windows . . . 749
Programowe raportowanie błędów systemu Windows . . . 752
Wyłączanie generowania i przesyłania raportów . . . 754
Dostosowywanie wszystkich raportów dotyczących problemów wewnątrz procesu . . . 755
Tworzenie i dostosowywanie raportu o problemie . . . 756
Tworzenie niestandardowego raportu dotyczącego problemu: *WerReportCreate* . . . 759
Ustawianie parametrów raportu: *WerReportSetParameter* . . . 760
Dodawanie pliku mini-zrzutu do raportu: *WerReportAddDump* . . . 761
Dodawanie arbitralnych plików do raportu: *WerReportAddFile* . . . 762
Modyfikowanie łańcuchów tekstowych w oknie dialogowym: *WerReportSetUIOption* . . . 763
Przekazywanie raportu dotyczącego problemu: *WerReportSubmit* . . . 764
Zamykanie raportu dotyczącego problemu: *WerReportCloseHandle* . . . 766
Aplikacja przykładowa Customized WER . . . 766
Automatyczne wznawianie aplikacji i odzyskiwanie danych . . . 772
Automatyczne wznawianie aplikacji . . . 772
Wsparcie dla odzyskiwania danych przez aplikację . . . 774

Część VI: Dodatki . . . 777

A Środowisko budowania aplikacji . . . 779

Plik nagłówkowy CmnHdr.h . . . 779
Opcja wersji systemu Microsoft Windows . . . 780
Opcje Unicode . . . 780
Definicje Windows i poziom ostrzeżeń 4 . . . 781
Pomocnicze makro *pragma message* . . . 781
Makro *chINRANGE* . . . 782
Makro *chBEGINTHREADEX* . . . 782

Ulepszenie *DebugBreak* dla platform *x*86. 783
Tworzenie kodów dla wyjątków programowych 784
Makro *chMB* 784
Makra *chASSERT* i *chVERIFY*. 784
Makro *chHANDLE_DLGMSG* 784
Makro *chSETDLGICONS* 784
Zmuszanie programu łączącego do szukania funkcji początkowej (*w*)*WinMain* . 784
Wsparcie dla kompozycji interfejsu użytkownika przy użyciu dyrektywy pragma 785

B Makra rozbijające komunikaty, makra pól potomnych i makra API . 791

Makra rozbijające komunikaty 792
Makra pól potomnych. 794
Makra API. 795

Część I
Lektura obowiązkowa

W tej części:

Rozdział 1: Obsługa błędów 3
Rozdział 2: Praca ze znakami i łańcuchami znaków. 11
Rozdział 3: Obiekty jądra 35

Rozdział 1

Obsługa błędów

W tym rozdziale:

Definiowanie własnych kodów błędów 7
Aplikacja przykładowa ErrorShow 8

Zanim zaczniemy zajmować się mnóstwem możliwości oferowanych przez Microsoft Windows, trzeba zrozumieć, w jaki sposób wykonywana przez różne funkcje Windows jest obsługa błędów.

Przy wywoływaniu funkcji Windows sprawdza ona przekazywane jej parametry i próbuje wykonać swoje zadanie. Jeśli przekazany zostanie niewłaściwy parametr lub żądane działanie nie będzie mogło zostać zrealizowane z jakiegoś innego powodu, wartość zwracana przez funkcję wskaże, że czegoś nie udało jej się zrobić. Tabela 1-1 przedstawia typy danych dla wartości zwracanych przez większość funkcji Windows.

Tabela 1-1 Najczęściej używane typy danych zwracane przez funkcje Windows

Typ danych	Wartość oznaczająca niepowodzenie
`VOID`	Ta funkcja raczej nie może zawieść. Bardzo mało funkcji Windows zwraca typ `VOID`.
`BOOL`	Jeśli działanie funkcji się nie powiedzie, zwracana jest wartość 0; w przeciwnym razie zwracana jest wartość niezerowa. Należy unikać sprawdzania, czy zwracana wartość jest równa `TRUE`: najlepiej zawsze sprawdzać, czy zwracana wartość jest różna od `FALSE`.
`HANDLE`	Jeśli działanie funkcji się nie powiedzie, zwracana jest zwykle wartość `NULL`; w przeciwnym razie `HANDLE` identyfikuje obiekt uchwytu, którym można manipulować. W tym przypadku trzeba być ostrożnym, ponieważ niektóre funkcje zwracają uchwyt o wartości `INVALID_HANDLE_VALUE`, który jest zdefiniowany jako –1. Dokumentacja Platform SDK dla danej funkcji jasno wyjaśni, czy funkcja zwraca `NULL`, czy też `INVALID_HANDLE_VALUE` w celu zaznaczenia niepowodzenia.
`PVOID`	Jeśli działanie funkcji się nie powiedzie, zwracana jest wartość `NULL`; w przeciwnym razie `PVOID` określa adres w pamięci dla bloku danych.
`LONG/DWORD`	W tym przypadku jest trudniej. Funkcje zwracające wartości liczbowe zwykle zwracają typ `LONG` lub `DWORD`. Jeśli z jakiegoś powodu funkcja nie może zwrócić właściwej liczby, zwykle zwraca 0 lub –1 (w zależności od funkcji). Przy wywoływaniu funkcji zwracającej `LONG/DWORD` warto przeczytać dokładnie dokumentację Platform SDK, aby upewnić się, że poprawnie wykonano sprawdzanie potencjalnych błędów.

Gdy funkcja Windows zwraca kod błędu, często warto zrozumieć, dlaczego działanie funkcji się nie powiodło. Firma Microsoft zebrała listę wszystkich możliwych kodów błędów i przypisała każdemu z nich liczbę 32-bitową.

Wewnętrznie, gdy funkcja Windows wykryje błąd, korzysta z mechanizmu zwanego pamięcią lokalną dla wątku w celu skojarzenia odpowiedniego kodu błędu z wątkiem wywołującym funkcję (mechanizm ten jest omawiany w rozdziale 21 „Pamięć lokalna dla wątku"). Mechanizm ten pozwala wątkom działać niezależnie od siebie bez wzajemnego wpływania na swoje kody błędów. Gdy funkcja kończy swoje działanie, zwracana przez nią wartość wskazuje, że wystąpił błąd. Aby dowiedzieć się, jaki to dokładnie błąd, należy wywołać funkcję `GetLastError`:

```
DWORD GetLastError();
```

Funkcja ta po prostu zwraca 32-bitowy kod błędu dla danego wątku przez ostatnie wywołanie funkcji.

Mając już ten 32-bitowy, liczbowy kod błędu, trzeba go przetłumaczyć na coś bardziej użytecznego. Plik nagłówkowy WinError.h zawiera listę wszystkich kodów błędów zdefiniowanych przez Microsoft. Przedstawię tutaj fragment tego pliku, aby można było zobaczyć, jak on wygląda:

```
// MessageId: ERROR_SUCCESS
//
// MessageText:
//
//  The operation completed successfully.
//
#define ERROR_SUCCESS                    0L

#define NO_ERROR 0L                            // dderror
#define SEC_E_OK                         ((HRESULT)0x00000000L)

//
// MessageId: ERROR_INVALID_FUNCTION
//
// MessageText:
//
//  Incorrect function.
//
#define ERROR_INVALID_FUNCTION           1L    // dderror

//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND             2L

//
// MessageId: ERROR_PATH_NOT_FOUND
//
// MessageText:
//
//  The system cannot find the path specified.
```

```
//
#define ERROR_PATH_NOT_FOUND            3L

//
// MessageId: ERROR_TOO_MANY_OPEN_FILES
//
// MessageText:
//
//  The system cannot open the file.
//
#define ERROR_TOO_MANY_OPEN_FILES       4L

//
// MessageId: ERROR_ACCESS_DENIED
//
// MessageText:
//
//  Access is denied.
//
#define ERROR_ACCESS_DENIED             5L
```

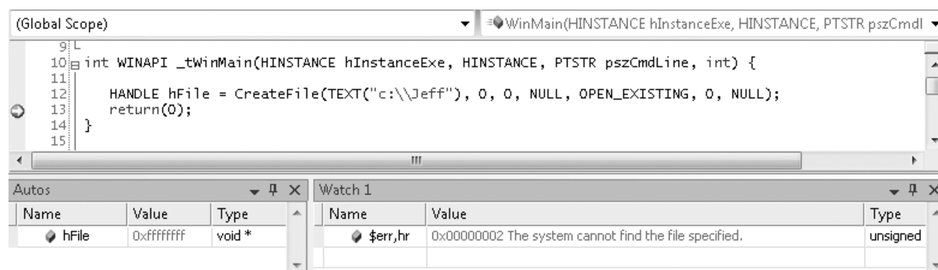
Jak widać, każdy błąd jest reprezentowany przez trzy elementy: identyfikator (makro, którego można używać w kodzie źródłowym do porównywania z wartością zwracaną przez `GetLastError`), tekst komunikatu (tekst w języku angielskim opisujący dany błąd) oraz numer (którego bezpośredniego użycia należy unikać, a zamiast niego stosować identyfikator). Trzeba mieć na uwadze, że wybrałem do pokazania tylko niewielki fragment pliku nagłówkowego WinError.h; cały plik ma długość ponad 39000 wierszy!

Gdy działanie funkcji Windows się nie powiedzie, należy od razu wywołać `GetLastError`, ponieważ zwracana wartość prawdopodobnie zostanie nadpisana, jeśli wywołana będzie kolejna funkcja Windows. Trzeba też mieć na uwadze, że funkcja Windows, która zakończy się powodzeniem, może też nadpisać tę wartość przez `ERROR_SUCCESS` (wartość oznaczającą powodzenie działania funkcji).

Działanie niektórych funkcji Windows może się powieść z kilku powodów. Na przykład próba utworzenia obiektu jądra dla nazwanego zdarzenia może się udać, ponieważ obiekt zostanie faktycznie utworzony albo ponieważ istnieje już obiekt zdarzenia o takiej samej nazwie. W aplikacji może istnieć potrzeba sprawdzenia faktycznej przyczyny powodzenia funkcji. Do zwrócenia tej informacji firma Microsoft wybrała zastosowanie tego samego mechanizmu informującego o kodzie ostatniego błędu. Gdy więc funkcja zakończy się powodzeniem, można uzyskać dodatkowe informacje, wywołując `GetLastError`. W razie funkcji zachowujących się w ten sposób dokumentacja Platform SDK jasno stwierdza, że można w tym celu użyć `GetLastError`. Na przykład dokumentacja funkcji `CreateEvent` informuje, że w przypadku istnienia wcześniej zdarzenia o tej samej nazwie zwracany jest kod `ERROR_ALREADY_EXISTS`.

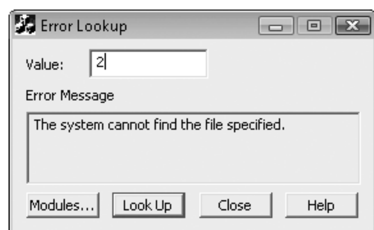
Uważam, że podczas debugowania sprawdzanie kodu ostatniego błędu w danym wątku jest niezwykle użyteczne. Debuger w Microsoft Visual Studio obsługuje bardzo użyteczną funkcję – można skonfigurować okno Watch tak, żeby zawsze pokazywało numer kodu ostatniego błędu w danym wątku oraz tekstowy opis tego błędu. Robi się to, zaznaczając wiersz w oknie Watch i wpisując **$err,hr**, jak można zobaczyć na rysunku 1-1. Widać na nim, że wywołana została funkcja `CreateFile`. Funkcja ta zwróciła dane typu `HANDLE` o wartości `INVALID_HANDLE_VALUE` (–1), co oznacza, że nie udało się otworzyć podanego pliku. Okno Watch pokazuje kod ostatniego błędu (kod błędu, który zostałby zwrócony przez

funkcję `GetLastError`, gdyby została wywołana), którego wartość równa jest 0x00000002. Dzięki określeniu `hr` okno Watch dodatkowo informuje, że kod błędu 2 oznacza, iż system nie może znaleźć podanego pliku („The system cannot find the file specified"). Można zauważyć, że jest to ten sam opis, który pojawia się w pliku nagłówkowym WinError.h dla kodu błędu o numerze 2.



Rysunek 1-1 Użycie *`$err,hr`* w oknie Watch programu Visual Studio w celu wyświetlania kodu ostatniego błędu w bieżącym wątku.

Visual Studio zawiera też niewielkie narzędzie o nazwie Error Lookup. Można z niego skorzystać, aby zamienić liczbę oznaczającą kod błędu na jego tekstowy opis.



Jeśli w pisanej aplikacji zostanie wykryty błąd, można zechcieć wyświetlić użytkownikowi opis tekstowy błędu. System Windows oferuje funkcję zamieniającą kod błędu na jego tekstowy opis. Funkcja ta nazywa się `FormatMessage`:

```
DWORD FormatMessage(
   DWORD dwFlags,
   LPCVOID pSource,
   DWORD dwMessageId,
   DWORD dwLanguageId,
   PTSTR pszBuffer,
   DWORD nSize,
   va_list *Arguments);
```

`FormatMessage` ma dość bogatą funkcjonalność i jest preferowanym sposobem konstruowania napisów tekstowych, które mają być wyświetlane użytkownikowi. Jednym z powodów dużej użyteczności tej funkcji jest łatwość jej współpracy z wieloma językami. Funkcja ta przyjmuje jako parametr identyfikator języka i zwraca właściwy tekst. Oczywiście, najpierw trzeba samemu przetłumaczyć odpowiednie łańcuchy znaków i wbudować przetłumaczoną tablicę komunikatów jako zasób w moduł .exe lub DLL, a następnie funkcja sama wybierze prawidłową wersję komunikatu. Aplikacja przykładowa ErrorShow (pokazana

w dalszej części tego rozdziału) demonstruje, jak wywoływać tę funkcję w celu zamiany kodu błędu zdefiniowanego przez Microsoft na jego tekstowy opis.

Od czasu do czasu ktoś zadaje mi pytanie, czy Microsoft tworzy ogólną listę zawierającą wszystkie możliwe kody błędów, które mogą być zwrócone z każdej funkcji Windows. Odpowiedź niestety brzmi: nie. Co więcej Microsoft prawdopodobnie nigdy nie stworzy takiej listy – po prostu zbudowanie i utrzymywanie jej w miarę pojawiania się nowych wersji systemu byłoby zbyt trudne.

Problem ze zbudowaniem takiej listy wynika z tego, że można wywołać jedną funkcję Windows, ale wewnętrznie funkcja ta może wywoływać inną funkcję, itd. Każda z wywołanych funkcji może spowodować błąd z wielu różnych powodów. Czasami w razie niepowodzenia danej funkcji, funkcja na wyższym poziomie może sobie z nim poradzić i mimo tego wykonać swoje zadanie. Do stworzenia takiej ogólnej listy trzeba by prześledzić ścieżkę wykonywania każdej funkcji i zbudować listę wszystkich możliwych kodów błędów. Jest to trudne. A w miarę tworzenia nowych wersji systemu ścieżki wykonywania funkcji mogłyby ulegać zmianie.

Definiowanie własnych kodów błędów

Pokazałem już, jak funkcje Windows informują funkcje nadrzędne o swoich błędach. Microsoft umożliwia też programistom skorzystanie z tego mechanizmu we własnych funkcjach. Załóżmy, że piszemy funkcję, która będzie wywoływana przez innych. Działanie funkcji może się nie powieść z różnych powodów, trzeba więc zwrócić z niej informację o błędzie.

W tym celu wystarczy ustawić kod ostatniego błędu w danym wątku, a następnie zwrócić z funkcji wartość `FALSE`, `INVALID_HANDLE_VALUE`, `NULL` lub inną odpowiednią w danej sytuacji. Aby ustawić kod ostatniego błędu w wątku, wystarczy wywołać

```
VOID SetLastError(DWORD dwErrCode);
```

przekazując w tej funkcji określony kod błędu w postaci liczby 32-bitowej. Zwykle dobrze jest korzystać z kodów istniejących już w WinError.h – o ile jakiś kod odpowiada dobrze błędowi, który ma być zgłoszony. Jeśli żaden z kodów w pliku WinError.h nie odpowiada ściśle danemu błędowi, można utworzyć własny kod. Kod błędu jest liczbą 32-bitową, którą można podzielić na pola przedstawione w Tabeli 1-2.

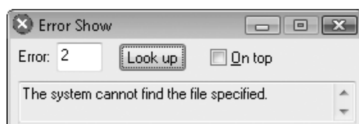
Tabela 1-2 Pola kodu błędu

Bity:	31–30	29	28	27–16	15–0
Zawartość	Kategoria błędu	Microsoft/ klient	Zarezerwowany	Kod funkcjonalności	Kod wyjątku
Znaczenie	0 = Powodzenie 1 = Informacja 2 = Ostrzeżenie 3 = Błąd	0 = kod zdefiniowany przez Microsoft 1 = kod zdefiniowany przez klienta	Musi być 0	Pierwsze 256 wartości jest zarezerwowanych przez Microsoft	Kod zdefiniowany przez Microsoft/ klienta

Pola te zostaną szczegółowo omówione w rozdziale 24 „Procedury obsługi wyjątków i wyjątki programowe”. Na razie jedynym istotnym polem, którego trzeba być świadomym, jest bit 29. Firma Microsoft obiecuje, że wszystkie kody błędów tworzone przez nią będą mieć w tym bicie wartość 0. Tworząc własne kody błędów, trzeba umieścić w tym bicie wartość 1. W ten sposób można mieć gwarancję, że własny kod błędu nie będzie nigdy w konflikcie z kodem błędu zdefiniowanym przez Microsoft istniejącym obecnie lub utworzonym w przyszłości. Pole funkcjonalności jest wystarczająco duże do pomieszczenia 4096 możliwych wartości. Pierwszych 256 spośród nich zostało zarezerwowanych przez Microsoft, pozostałe wartości mogą być definiowane we własnych aplikacjach.

Aplikacja przykładowa ErrorShow

Aplikacja ErrorShow (01-ErrorShow.exe) demonstruje jak uzyskać opis tekstowy dla kodu błędu. Kod źródłowy i pliki zasobów dla tej aplikacji można znaleźć w folderze 01-ErrorShow w materiałach dodatkowych do tej książki, które można pobrać ze strony WWW pod adresem http://wintellect.com/Books.aspx. Aplikacja ta pokazuje po prostu, jak działa okno Watch debugera oraz narzędzie Error Lookup. Po uruchomieniu programu pojawi się następujące okno:



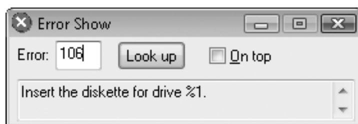
W polu edycyjnym można wpisać dowolny numer błędu. Po kliknięciu przycisku Look Up opis tekstowy błędu zostanie wyświetlony w przewijanym polu u dołu okna. Jedyną interesującą cechą tej aplikacji jest sposób wywołania `FormatMessage`. Oto, jak skorzystałem z tej funkcji:

```
// Pobierz kod błędu
DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

HLOCAL hlocal = NULL;   // Bufor na łańcuch tekstowy z komunikatem o błędzie

// Skorzystaj z domyślnych ustawień lokalizacyjnych systemu,
// ponieważ sprawdzamy komunikaty Windows
// Uwaga: ta kombinacja MAKELANGID ma wartość 0
DWORD systemLocale = MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL);

// Pobierz opis tekstowy kodu błędu
BOOL fOk = FormatMessage(
   FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
   FORMAT_MESSAGE_ALLOCATE_BUFFER,
   NULL, dwError, systemLocale,
   (PTSTR) &hlocal, 0, NULL);

if (!fOk) {
   // Czy to błąd związany z siecią?
   HMODULE hDll = LoadLibraryEx(TEXT("netmsg.dll"), NULL,
      DONT_RESOLVE_DLL_REFERENCES);
```

```
   if (hDll != NULL) {
      fOk = FormatMessage(
         FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_IGNORE_INSERTS |
         FORMAT_MESSAGE_ALLOCATE_BUFFER,
         hDll, dwError, systemLocale,
         (PTSTR) &hlocal, 0, NULL);
      FreeLibrary(hDll);
   }
}

if (fOk && (hlocal != NULL)) {
   SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
   LocalFree(hlocal);
} else {
   SetDlgItemText(hwnd, IDC_ERRORTEXT,
      TEXT("No text found for this error number."));
```

Pierwszy wiersz pobiera numer kodu błędu z pola edycyjnego. Następnie tworzony jest uchwyt do bloku pamięci i inicjowany wartością `NULL`. Funkcja `FormatMessage` wewnętrznie alokuje ten blok pamięci i zwraca z powrotem jego uchwyt.

Przy wywoływaniu `FormatMessage` przekazywana jest flaga `FORMAT_MESSAGE_FROM_SYSTEM`. Informuje ona funkcję `FormatMessage`, że chcemy uzyskać łańcuch dla kodu błędu zdefiniowanego przez system. Przekazywana jest też flaga `FORMAT_MESSAGE_ALLOCATE_BUFFER`, aby nakazać funkcji zaalokowanie wystarczająco dużego bloku pamięci na pomieszczenie opisu tekstowego danego błędu. Uchwyt do tego bloku zostanie zwrócony w zmiennej `hlocal`. Flaga `FORMAT_MESSAGE_IGNORE_INSERTS` pozwala uzyskać teksty z kodami % dla parametrów używanych przez Windows do wstawiania informacji kontekstowych, jak pokazano na następującym zrzucie ekranu:



Jeśli flaga ta nie byłaby przekazana, trzeba by dostarczyć wartości zmiennych podstawianych w miejsce kodów % w parametrze `Arguments`; ale nie jest to możliwe w przypadku aplikacji Error Show, ponieważ zawartość komunikatów nie jest znana z góry.

Trzeci parametr określa numer błędu, który chcemy sprawdzić. Czwarty w jakim języku chcemy uzyskać opis tekstowy. Ponieważ interesują nas komunikaty dostarczane bezpośrednio przez Windows, identyfikator języka tworzony jest z dwóch stałych, których połączenie daje w efekcie wartość 0 – co oznacza domyślny język systemu operacyjnego. Wówczas nie możemy bezpośrednio zakodować określonego języka, ponieważ nie wiemy z góry, jaki będzie język instalacji systemu operacyjnego, w której program będzie uruchamiany.

Jeśli funkcja `FormatMessage` zakończy się powodzeniem, opis tekstowy błędu będzie znajdować się w bloku pamięci, z którego jest następnie kopiowany do przewijanego pola u dołu okna dialogowego. Jeśli funkcja `FormatMessage` zakończy się niepowodzeniem, próbujemy wyszukać kod komunikatu w module NetMsg.dll, aby sprawdzić, czy dany kod jest związany z siecią (szczegóły dotyczące wyszukiwania modułów DLL na dysku można znaleźć w rozdziale 20 „Zaawansowane techniki DLL"). Używając uchwytu do modułu NetMsg.dll, ponownie wywołujemy `FormatMessage`. Każdy moduł DLL, (lub .exe) może

mieć własny zbiór kodów błędów, który można zdefiniować, używając narzędzia Message Compiler (MC.exe) w celu dodania zasobów do modułu. Narzędzie Error Lookup z programu Visual Studio pozwala wyszukiwać kody w innych modułach dzięki przyciskowi Modules.

Rozdział 2

Praca ze znakami i łańcuchami znaków

W tym rozdziale:

Kodowanie znaków . . . 12
Znakowe i łańcuchowe typy danych ANSI i Unicode . . . 14
Funkcje Unicode i ANSI w Windows . . . 16
Funkcje Unicode i ANSI w bibliotece uruchomieniowej C . . . 18
Bezpieczne funkcje łańcuchowe w bibliotece uruchomieniowej C . . . 19
Dlaczego należy korzystać z Unicode? . . . 27
Jak zalecamy pracować ze znakami i łańcuchami? . . . 28
Tłumaczenie łańcuchów pomiędzy Unicode a ANSI . . . 29

W miarę jak system Microsoft Windows staje się coraz bardziej popularny na całym świecie, coraz ważniejsze dla programistów jest przygotowywanie aplikacji na różne rynki międzynarodowe. Dawniej wersje amerykańskie oprogramowania często pojawiały się nawet sześć miesięcy przed wersjami międzynarodowymi. Rosnące wsparcie międzynarodowe dla systemu operacyjnego ułatwia produkowanie aplikacji na rynki międzynarodowe i zmniejsza opóźnienia pomiędzy dystrybucją amerykańskich a międzynarodowych wersji oprogramowania.

System Windows zawsze wspierał programistów w lokalizowaniu aplikacji. Aplikacja może uzyskać informacje specyficzne dla określonego kraju z różnych funkcji i może sprawdzać ustawienia z Panelu sterowania w celu określenia preferencji użytkownika. Windows obsługuje różne kroje czcionek w aplikacjach. Wreszcie Windows Vista wspiera standard Unicode w wersji 5.0 (aby zapoznać się z Unicode 5.0 można przeczytać artykuł „Extend The Global Reach Of Your Applications With Unicode 5.0" pod adresem *http://msdn.microsoft.com/msdnmag/issues/07/01/Unicode/default.aspx*).

Błędy przepełnienia bufora (typowe przy manipulowaniu łańcuchami znaków) stały się główną drogą ataków na aplikacje a nawet na części systemu operacyjnego. W poprzednich latach firma Microsoft podjęła wiele wewnętrznych i zewnętrznych wysiłków mających na celu podniesienie poprzeczki bezpieczeństwa w świecie Windows. Druga część tego rozdziału przedstawia nowe funkcje dostarczane przez Microsoft w bibliotece wykonawczej języka C. Przy przetwarzaniu łańcuchów znaków należy korzystać z tych nowych funkcji dla ochrony kodu przed przepełnieniami bufora.

Postanowiłem przedstawić ten rozdział na początku książki, ponieważ zalecam, aby aplikacje zawsze używały łańcuchów znaków w standardzie Unicode i aby zawsze przetwarzać

łańcuchy znaków przy użyciu nowych, bezpiecznych funkcji łańcuchowych. Jak będzie się można przekonać, kwestie związane z bezpiecznym używaniem łańcuchów Unicode będą omawiane niemal w każdym rozdziale i we wszystkich aplikacjach przykładowych w tej książce. Mając bazę kodu nieużywającego Unicode, najlepiej przenieść go do standardu Unicode, ponieważ polepszy to wydajność aplikacji oraz przygotuje ją do lokalizacji na inne języki. Pomoże to również przy współpracy z COM i .NET Framework.

Kodowanie znaków

Prawdziwym problemem przy lokalizacji oprogramowania było zawsze manipulowanie różnymi zbiorami znaków. Przez lata łańcuchy tekstowe były kodowane jako ciągi jednobajtowych znaków z zerem na końcu. Wielu z nas weszło to w krew. Wywołanie funkcji `strlen` zwraca liczbę znaków w zakończonej zerem tablicy jednobajtowych znaków ANSI.

Problem w tym, że niektóre języki i systemy pisma (klasyczny przykład to japońskie kandzi) mają tak dużo symboli w swoich zbiorach znaków, że pojedynczy bajt oferujący co najwyżej 256 różnych symboli po prostu nie wystarcza. Utworzono więc dwubajtowe zbiory znaków (DBCS) wspierające te języki i systemy pisma. W dwubajtowym zbiorze znaków każdy znak łańcucha składa się z 1 lub 2 bajtów. Na przykład, jeśli chodzi o kandzi, gdy wartość pierwszego znaku znajduje się pomiędzy 0x81 a 0x9F lub pomiędzy 0xE0 a 0xFC, trzeba odczytać następny bajt, aby określić pełny znak łańcucha. Praca z dwubajtowymi zbiorami znaków jest koszmarem dla programisty, ponieważ niektóre znaki mają szerokość 1 bajta, a inne 2 bajtów. Na szczęście można już zapomnieć o zbiorach znaków DBCS i skorzystać ze wsparcia dla łańcuchów Unicode oferowanego przez funkcje Windows oraz funkcje biblioteki uruchomieniowej języka C.

Unicode jest standardem zapoczątkowanym przez firmy Apple i Xerox w roku 1988. W 1991 roku powstało konsorcjum na rzecz rozwoju i promocji Unicode. W skład konsorcjum wchodzą takie firmy, jak Apple, Compaq, Hewlett-Packard, IBM, Microsoft, Oracle, Silicon Graphics, Sybase, Unisys i Xerox (pełna i aktualna lista członków konsorcjum dostępna jest pod adresem http://www.Unicode.org). Ta grupa firm odpowiada za obsługę standardu Unicode. Pełny opis standardu Unicode można znaleźć w książce The Unicode Standard opublikowanej przez wydawnictwo Addison-Wesley (książka ta jest dostępna poprzez adres http://www.Unicode.org).

W systemie Windows Vista każdy znak Unicode jest kodowany przy użyciu notacji UTF-16 (UTF jest skrótem od Unicode Transformation Format – format transformacji Unicode). UTF-16 koduje każdy znak w postaci 2 bajtów (czyli 16 bitów). W tej książce mówiąc o Unicode, zawsze odnosimy się do kodowania UTF-16, o ile nie zostanie wskazane co innego. Windows używa UTF-16, ponieważ znaki z większości języków używanych na świecie mogą być łatwo reprezentowane przez wartości 16-bitowe, co pozwala programom na proste przeglądanie łańcuchów znak po znaku i obliczanie ich długości. Jednakże 16 bitów nie wystarcza do przedstawienia wszystkich znaków z pewnych języków. W przypadku tych języków UTF-16 obsługuje substytuty stanowiące sposób wykorzystania 32 bitów (czyli 4 bajtów) do przedstawiania pojedynczego znaku. Ponieważ niewiele aplikacji musi korzystać ze znaków w tych językach, UTF-16 jest dobrym kompromisem pomiędzy oszczędnością pamięci a ułatwieniem kodowania. Warto zwrócić uwagę, że platforma .NET Framework zawsze koduje wszystkie znaki i łańcuchy przy użyciu UTF-16, więc korzystanie z UTF-16 w aplikacjach Windows zwiększy wydajność i ograniczy wykorzystanie

pamięci, jeśli trzeba będzie przekazywać znaki lub łańcuch pomiędzy kodem rodzimym a kodem zarządzanym.

Istnieją inne standardy UTF dotyczące reprezentowania znaków, wśród nich:

UTF-8 UTF-8 koduje niektóre znaki w postaci 1 bajta, niektóre znaki w postaci 2 bajtów, inne w postaci 3 bajtów, a jeszcze inne w postaci 4 bajtów. Znaki o wartościach mniejszych niż 0x0080 są kompresowane do 1 bajta, co dotyczy znaków używanych w Stanach Zjednoczonych. Znaki pomiędzy 0x0080 a 0x07FF są konwertowane na 2 bajty, co dotyczy języków europejskich i bliskowschodnich. Znaki o wartościach większych lub równych 0x0800 są konwertowane na 3 bajty, co dotyczy języków wschodnioazjatyckich. Na koniec pary substytutów zapisywane są w postaci 4 bajtów. UTF-8 jest niezwykle popularnym formatem kodowania, ale jest mniej efektywny niż UTF-16, jeśli koduje się wiele znaków o wartościach większych lub równych 0x0800.

UTF-32 UTF-32 koduje każdy znak jako 4 bajty. To kodowanie jest przydatne, jeśli trzeba napisać prosty algorytm przechodzenia przez kolejne znaki (używane w dowolnym języku), a nie chce się mieć do czynienia ze znakami o różnej liczbie bajtów. W wypadku UTF-32 nie trzeba myśleć o substytutach, ponieważ każdy znak jest 4-bajtowy. Oczywiście kodowanie UTF-32 nie jest efektywne, jeśli chodzi o wykorzystanie pamięci. Dlatego jest rzadko używane do zapisywania łańcuchów tekstowych w pliku lub przesyłania ich przez sieć. Ten format kodowania zwykle jest używany wewnętrznie w samym programie.

Obecnie punkty kodowania Unicode1 są zdefiniowane dla następujących alfabetów (zwanych pismami): arabskiego, chińskiego bopomofo, cyrylicy (rosyjskiego), greckiego, hebrajskiego, japońskiego kana, koreańskiego hangul, łacińskiego i wielu innych. W każdej wersji Unicode pojawiają się nowe znaki w istniejących pismach, a nawet nowe pisma, na przykład fenickie. W zbiorach znaków zawarto też znaczną liczbę znaków interpunkcyjnych, symboli matematycznych, symboli technicznych, strzałek, piktogramów, znaków diakrytycznych i innych znaków.

Wszystkie 65536 znaków podzielono na regiony. Tabela 2-1 przedstawia niektóre regiony i przypisane im grupy znaków.

Tabela 2-1 Zbiory znaków Unicode i alfabety

Kod 16-bitowy	Znaki	Kod 16-bitowy	Alfabet/pisma
0000–007F	ASCII	0300–036F	ogólne znaki diakrytyczne
0080–00FF	znaki Latin1	0400–04FF	cyrylica
0100–017F	łacińskie europejskie	0530–058F	ormiański
0180–01FF	łacińskie rozszerzone	0590–05FF	hebrajski
0250–02AF	standard zapisu fonetycznego	0600–06FF	arabski
02B0–02FF	zmodyfikowane litery	0900–097F	dewanagari

Znakowe i łańcuchowe typy danych ANSI i Unicode

Na pewno wszyscy są świadomi, że język C wykorzystuje typ danych `char` do reprezentowania 8-bitowych znaków ANSI. Domyślnie przy deklarowaniu stałej łańcuchowej w kodzie źródłowym kompilator C zamienia znaki łańcucha na tablicę 8-bajtowych znaków typu `char`:

```
// Znak 8-bitowy
char c = 'A';

// Tablica do 99 znaków 8-bitowych oraz kończącego łańcuch 8-bitowego zera.
char szBuffer[100] = "A String";
```

Kompilator C/C++ firmy Microsoft definiuje wbudowany typ danych `wchar_t`, który reprezentuje 16-bitowy znak Unicode (UTF-16). Ponieważ wcześniejsze wersje kompilatora Microsoft nie oferowały tego wbudowanego typu danych, kompilator definiuje ten typ danych tylko wtedy, gdy określony jest przełącznik kompilatora `/Zc:wchar_t`. Domyślnie przy kompilowaniu projektu C++ w Microsoft Visual Studio ten przełącznik kompilatora jest włączony. Zalecane jest korzystanie z niego, ponieważ lepiej pracować ze znakami Unicode poprzez wbudowany typ pierwotny rozumiany wewnętrznie przez kompilator.

Uwaga Przed wsparciem ze strony wbudowanego kompilatora plik nagłówkowy C definiował typ danych `wchar_t` w następujący sposób:

```
typedef unsigned short wchar_t;
```

A oto, jak zadeklarować znak i łańcuch znaków Unicode:

```
// Znak 16-bitowy
wchar_t c = L'A';

// Tablica do 99 znaków 16-bitowych oraz kończącego łańcuch 16-bitowego zera.
wchar_t szBuffer[100] = L"A String";
```

Wielka litera `L` przed stałą łańcuchową informuje kompilator, że dany łańcuch znaków ma zostać skompilowany jako łańcuch Unicode. Gdy kompilator będzie umieszczał ten łańcuch w sekcji danych programu, zakoduje każdy znak przy użyciu UTF-16, wstawiając w tym prostym przypadku bajty zerowe pomiędzy każdy ze znaków ASCII.

Zespół Windows w firmie Microsoft definiuje własne typy danych, aby odizolować się nieco od języka C. Tak więc plik nagłówkowy Windows – WinNT.h definiuje następujące typy danych:

```
typedef char     CHAR;   // Znak 8-bitowy
typedef wchar_t WCHAR;   // Znak 16-bitowy
```

Co więcej plik nagłówkowy WinNT.h definiuje garść wygodnych typów danych przydatnych do pracy ze wskaźnikami do znaków i do łańcuchów:

```
// Wskaźnik do 8-bitowego znaku(-ów)
typedef CHAR *PCHAR;
typedef CHAR *PSTR;
typedef CONST CHAR *PCSTR
```

```
// Wskaźnik do 16-bitowego znaku(-ów)
typedef WCHAR *PWCHAR;
typedef WCHAR *PWSTR;
typedef CONST WCHAR *PCWSTR;
```

Uwaga W pliku nagłówkowym WinNT.h można znaleźć następującą definicję:

```
typedef __nullterminated WCHAR *NWPSTR, *LPWSTR, *PWSTR;
```

Przedrostek `__nullterminated` jest *przypisem nagłówkowym*, który opisuje, w jaki sposób typy mają być używane jako parametry funkcji i zwracane przez nie wartości. W wersji Enterprise programu Visual Studio można włączyć opcję Code Analysis (analiza kodu) we właściwościach projektu. Spowoduje to dodatnie przełącznika `/analyze` do wiersza poleceń kompilatora, który będzie wykrywać w kodzie wywołania funkcji w sposób niezgodny z semantyką zdefiniowaną przez przypisy. Warto zwrócić uwagę, że tylko wersje Enterprise kompilatora obsługują przełącznik `/analyze`. Aby kod w tej książce był bardziej czytelny, przypisy nagłówkowe są pomijane. Warto przeczytać stronę dokumentacji „Header Annotations" w portalu MSDN pod adresem *http://msdn2.microsoft.com/En-US/library/aa383701.aspx*, aby dowiedzieć się więcej na temat języka przypisów nagłówkowych.

We własnym kodzie źródłowym nie ma większego znaczenia, których typów danych się używa, ale zalecam bycie konsekwentnym w celu tworzenia kodu łatwiejszego w utrzymaniu. Osobiście, jako programista Windows, zawsze używam typów danych Windows, ponieważ odpowiadają one typom danych używanych w dokumentacji MSDN, co wszystkim ułatwia czytanie kodu.

Możliwe jest pisanie kodu źródłowego w taki sposób, żeby mógł być kompilowany z użyciem znaków i łańcuchów albo w standardzie ANSI, albo Unicode. W pliku nagłówkowym WinNT.h zdefiniowane są następujące typy i makra:

```
#ifdef UNICODE

typedef WCHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST WCHAR *PCTSTR;
#define __TEXT(quote) quote         // r_winnt

#define __TEXT(quote) L##quote

#else

typedef CHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST CHAR *PCTSTR;
#define __TEXT(quote) quote

#endif

#define   TEXT(quote) __TEXT(quote)
```

Te typy i makra (oraz kilka rzadziej używanych, niepokazanych tutaj) są stosowane do tworzenia kodu źródłowego, który może być kompilowany z użyciem albo znaków i łańcuchów ANSI, albo Unicode, jak na przykład:

```
// Jeśli zdefiniowano UNICODE, to znak 16-bitowy; w przeciwnym razie znak
// 8-bitowy
TCHAR c = TEXT('A');
```

```
// Jeśli zdefiniowano UNICODE, to tablica znaków 16-bitowych; w przeciwnym razie
// znaki 8-bitowe
TCHAR szBuffer[100] = TEXT("A String");
```

Funkcje Unicode i ANSI w Windows

Od czasu Windows NT wszystkie wersje Windows są tworzone od podstaw z użyciem Unicode. To znaczy wszystkie funkcje podstawowe do tworzenia okien, wyświetlania tekstu, przetwarzania łańcuchów tekstowych, itd. wymagają łańcuchów Unicode. Jeśli funkcja Windows zostanie wywołana z przekazanym jej łańcuchem ANSI (łańcuchem znaków 1-bajtowych), funkcja ta zamieni ten łańcuch na Unicode, a następnie przekaże go systemowi operacyjnemu. Jeśli oczekiwane jest zwrócenie z funkcji łańcucha ANSI, system zamieni łańcuch Unicode na ANSI przed zwróceniem go aplikacji. Wszystkie te konwersje dokonywane są w sposób niewidoczny dla programisty. Oczywiście do przeprowadzenia tych konwersji wymagane są od systemu czas i pamięć.

Gdy Windows udostępnia funkcję wymagającą łańcucha tekstowego jako parametru, zwykle dostarczane są dwie wersje tej samej funkcji – na przykład funkcja `CreateWindowEx` przyjmująca łańcuchy Unicode i druga funkcja `CreateWindowEx` przyjmująca łańcuchy ANSI. W istocie obie mają następujące prototypy:

```
HWND WINAPI CreateWindowExW(
   DWORD dwExStyle,
   PCWSTR pClassName,    // Łańcuch Unicode
   PCWSTR pWindowName,   // Łańcuch Unicode
   DWORD dwStyle,
   int X,
   int Y,
   int nWidth,
   int nHeight,
   HWND hWndParent,
   HMENU hMenu,
   HINSTANCE hInstance,
   PVOID pParam);

HWND WINAPI CreateWindowExA(
   DWORD dwExStyle,
   PCSTR pClassName,    // Łańcuch ANSI
   PCSTR pWindowName,   // Łańcuch ANSI
   DWORD dwStyle,
   int X,
   int Y,
   int nWidth,
   int nHeight,
   HWND hWndParent,
   HMENU hMenu,
   HINSTANCE hInstance,
   PVOID pParam);
```

`CreateWindowExW` jest wersją przyjmującą łańcuchy Unicode. Wielka litera W na końcu nazwy funkcji oznacza „wide” – szeroki. Znaki Unicode mają szerokość 16 bitów, więc często mówi się o nich jako o znakach szerokich. Wielka litera A na końcu `CreateWindowExA` wskazuje, że funkcja przyjmuje łańcuchy znaków ANSI.

Zwykle jednak umieszczamy w kodzie po prostu wywołanie `CreateWindowEx`, a nie wywołujemy bezpośrednio `CreateWindowExW` ani `CreateWindowExA`. W pliku nagłówkowym WinUser.h, funkcja `CreateWindowEx` jest w rzeczywistości zdefiniowana jako makro:

```
#ifdef UNICODE
#define CreateWindowEx CreateWindowExW
#else
#define CreateWindowEx CreateWindowExA
#endif
```

To, czy zdefiniowano `UNICODE` podczas kompilacji kodu źródłowego, określa, która wersja `CreateWindowEx` zostanie wywołana. Podczas tworzenia nowego projektu w Visual Studio, `UNICODE` jest definiowane domyślnie. Jakiekolwiek wywołania `CreateWindowEx` są więc domyślnie rozwijane przez makro w wywołania `CreateWindowExW` – wersji Unicode funkcji `CreateWindowEx`.

W systemie Windows Vista kod źródłowy Microsoft dla funkcji `CreateWindowExA` stanowi po prostu warstwa tłumacząca, która alokuje pamięć na zamianę łańcuchów ANSI na Unicode, a następnie wywołuje funkcję `CreateWindowExW`, przekazując jej przekonwertowane łańcuchy. Gdy funkcja `CreateWindowExW` skończy działanie, funkcja `CreateWindowExA` zwolni swoje bufory pamięciowe i zwróci uchwyt okna. W przypadku funkcji wypełniających bufory łańcuchami system musi konwertować łańcuchy Unicode na ich odpowiedniki nieużywające Unicode, zanim aplikacja będzie mogła przetworzyć łańcuch tekstowy. Ponieważ system musi wykonywać wszystkie te konwersje, aplikacja wymaga więcej pamięci i działa wolniej. Można sprawić, aby aplikacja działała efektywniej, pisząc ją od początku z wykorzystaniem Unicode. Znane też były przypadki wykrycia błędów w tych funkcjach tłumaczących Windows, więc unikanie ich również wyeliminuje pewne potencjalne błędy.

W razie tworzenia dynamicznie dołączanych bibliotek (DLL), z których będą korzystać inni programiści, warto rozważyć użycie techniki udostępniania dwóch eksportowanych funkcji w DLL – wersji ANSI i wersji Unicode. W wersji ANSI należy po prostu przydzielić pamięć, wykonać niezbędne konwersje i wywołać wersję Unicode funkcji. Zademonstruję ten proces później w tym rozdziale w części „Eksportowanie funkcji ANSI i Unicode z bibliotek DLL".

Pewne funkcje w Windows API, takie jak `WinExec` i `OpenFile`, istnieją jedynie dla wstecznej zgodności z programami 16-bitowymi, które obsługiwały tylko łańcuchy ANSI. W dzisiejszych programach należy unikać tych metod. Wywołania `WinExec` i `OpenFile` powinny być zastąpione wywołaniami funkcji `CreateProcess` i `CreateFile`. Tak czy owak te stare funkcje wewnętrznie po prostu wywołują nowe. Dużym problemem w wypadku starych funkcji jest to, że nie przyjmują łańcuchów Unicode i zwykle oferują mniejszą funkcjonalność. Przy wywoływaniu tych funkcji trzeba przekazywać łańcuchy ANSI. W systemie Windows Vista większość funkcji ma zarówno wersję Unicode jak i wersję ANSI. Jednakże coraz częściej Microsoft tworzy funkcje oferujące jedynie wersje Unicode – na przykład `ReadDirectoryChangesW` i `CreateProcessWithLogonW`.

Gdy firma Microsoft przenosiła COM z 16-bitowej wersji Windows do Win32, podjęto decyzję, że wszystkie metody interfejsów COM wymagające łańcuchów znaków będą przyjmować jedynie łańcuchy Unicode. To była świetna decyzja, ponieważ model COM zwykle jest używany w celu umożliwienia różnym składnikom wzajemnej wymiany danych,

a Unicode jest najlepszym sposobem na przekazywanie łańcuchów znaków. Korzystanie z Unicode w całej aplikacji ułatwia też współdziałanie z COM.

Gdy kompilator zasobów kompiluje wszystkie zasoby aplikacji, tworzy wynikowy plik będący binarną reprezentacją tych zasobów. Wartości łańcuchowe w zasobach (tabele łańcuchów, szablony okien dialogowych, menu, itd.) zawsze są zapisywane przy użyciu łańcuchów Unicode. W Windows Vista system dokonuje wewnętrznych konwersji, jeśli aplikacja nie definiuje makra `UNICODE`. Na przykład, jeśli `UNICODE` nie jest zdefiniowane podczas kompilowania modułu źródłowego, wywołanie `LoadString` tak naprawdę wywoła funkcję `LoadStringA`. `LoadStringA` wczyta łańcuch Unicode z zasobów i przekonwertuje go na ANSI. Reprezentacja łańcucha w kodowaniu ANSI zostanie zwrócona z funkcji do aplikacji.

Funkcje Unicode i ANSI w bibliotece uruchomieniowej C

Podobnie jak funkcja Windows biblioteka uruchomieniowa języka C oferuje jeden zbiór funkcji do manipulowania znakami i łańcuchami ANSI oraz drugi do manipulowania znakami i łańcuchami Unicode. Jednakże w przeciwieństwie do Windows funkcje ANSI nie tłumaczą łańcuchów na Unicode i nie wywołują wewnętrznie funkcji w wersjach Unicode. Oczywiście wersje Unicode też wykonują same swoją robotę; nie wywołują wewnętrznie wersji ANSI.

Przykładem funkcji uruchomieniowej C, która zwraca długość łańcucha ANSI, jest `strlen`, a przykładem jej odpowiednika, który zwraca długość łańcucha Unicode jest `wcslen`. Prototypy obu tych funkcji można znaleźć w pliku nagłówkowym String.h. Aby pisać kod, który będzie można skompilować albo w wersji ANSI, albo Unicode, trzeba też dołączyć plik TChar.h definiujący następujące makro:

```
#ifdef _UNICODE
#define _tcslen     wcslen
#else
#define _tcslen     strlen
#endif
```

Następnie w kodzie należy wywoływać `_tcslen`. Jeśli zdefiniowano `_UNICODE`, makro rozwinie to wywołanie do `wcslen`; w przeciwnym razie wywołanie zostanie rozwinięte do `strlen`. Domyślnie przy tworzeniu nowego projektu C++ w Visual Studio, identyfikator `_UNICODE` jest zdefiniowany (tak samo, jak zdefiniowany jest identyfikator `UNICODE`). Biblioteka uruchomieniowa C zawsze poprzedza identyfikatory, które nie są częścią standardu C++, znakami podkreślenia, natomiast zespół Windows nie stosuje takiej konwencji. Dlatego w swoich aplikacjach warto się upewnić że zdefiniowane są obie definicje `UNICODE` i `_UNICODE` albo nie jest zdefiniowana żadna z nich. Dodatek A „Środowisko budowania aplikacji" opisze szczegółowo plik nagłówkowy CmnHdr.h używany przez wszystkie przykłady kodu z tej książki w celu uniknięcia tego rodzaju problemów.

Bezpieczne funkcje łańcuchowe w bibliotece uruchomieniowej C

Każda funkcja modyfikująca łańcuch znaków przedstawia potencjalne niebezpieczeństwo: jeśli docelowy bufor dla łańcucha nie będzie wystarczająco duży do pomieszczenia łańcucha wynikowego, wystąpi nadpisanie pamięci poza buforem. Oto przykład:

```
// Poniższy kod wstawia 4 znaki do 3-znakowego bufora, co prowadzi do naruszenia
// pamięci
WCHAR szBuffer[3] = L"";
wcscpy(szBuffer, L"abc"); // Końcowe 0 też jest znakiem!
```

Problem z funkcjami `strcpy` i `wcscpy` (oraz większością innych funkcji przetwarzających łańcuchy znaków) polega na tym, że nie przyjmują one argumentu określającego maksymalny rozmiar bufora i dlatego funkcja nie wie, że nadpisuje pamięć poza buforem. Skoro funkcja nie zdaje sobie z tego sprawy, nie może zgłosić błędu, nie wiadomo więc, że bufor został przepełniony. Oczywiście najlepiej byłoby, gdyby działanie funkcji po prostu się nie powiodło, nie uszkadzając dodatkowej pamięci.

Tego typu zachowanie było w przeszłości często wykorzystywane przez szkodliwe oprogramowanie. Obecnie Microsoft zapewnia zbiór nowych funkcji zastępujących niebezpieczne funkcje manipulujące łańcuchami (takie jak pokazana wcześniej funkcja `wcscat`) z biblioteki uruchomieniowej C, które wielu z nas poznało i polubiło przez lata. Aby pisać bezpieczny kod, nie powinno się już używać żadnych ze znanych funkcji uruchomieniowych C, które modyfikują łańcuchy znaków (funkcje takie jak `strlen`, `wcslen` i `_tcslen` są jednak dobre, ponieważ nie próbują modyfikować przekazywanych do nich łańcuchów, chociaż zakładają, że przekazywany łańcuch jest zakończony zerem, co może nie być prawdą). Zamiast tego należy wykorzystywać nowe, bezpieczne funkcje łańcuchowe zdefiniowane w dostarczanym przez Microsoft pliku StrSafe.h.

Uwaga Firma Microsoft wewnętrznie przepisała swoje biblioteki klas ATL i MFC tak, aby używały nowych, bezpiecznych funkcji łańcuchowych, dlatego jeśli korzysta się z tych bibliotek, wystarczy przebudować swoje aplikacje z użyciem nowych ich wersji, aby uzyskać bezpieczniejsze aplikacje.

Ponieważ książka ta nie jest poświęcona programowaniu w C/C++, szczegółów dotyczących korzystania z tej biblioteki należy szukać w następujących źródłach informacji:

- Artykuł w piśmie MSDN Magazine „Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries" autorstwa Martyna Lovella dostępny pod adresem http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx
- Prezentacja wideo Martyna Lovella w serwisie Channel9 dostępna pod adresem http://channel9.msdn.com/Showpost.aspx?postid=186406
- Dokumentacja dotycząca bezpiecznych łańcuchów znaków w serwisie MSDN Online dostępna pod adresem http://msdn2.microsoft.com/en-us/library/ms647466.aspx
- Lista wszystkich bezpiecznych funkcji zastępujących funkcje z biblioteki uruchomieniowej C w serwisie MSDN Online, którą można znaleźć pod adresem http://msdn2.microsoft.com/en-us/library/wd3wzwts(VS.80).aspx

Warto jednak w tym rozdziale omówić kilka szczegółów. Zacznę od opisania wzorców stosowanych przez nowe funkcje. Następnie wspomnę o pułapkach, które można napotkać przy przenoszeniu kodu używającego starszych funkcji do odpowiadających im bezpiecznych wersji jak na przykład przy stosowaniu `_tcscpy_s` zamiast `_tcscpy`. Następnie pokażę, kiedy ciekawsze byłoby zamiast tego wywoływanie nowych funkcji `StringC*`.

Wprowadzenie do nowych, bezpiecznych funkcji łańcuchowych

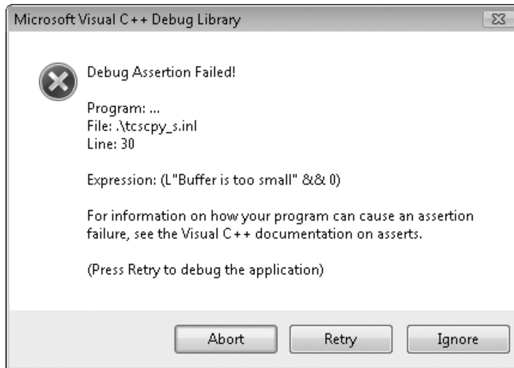
Gdy dołącza się plik StrSafe.h, dołączany jest też plik String.h, a istniejące dotychczas funkcje manipulujące łańcuchami z biblioteki uruchomieniowej C, takie jak te kryjące się za makrem `_tcscpy`, są oznakowywane podczas kompilacji ostrzeżeniami o ich dezaktualizacji. Ważne jest, że dołączenie StrSafe.h musi pojawiać się za wszystkimi innymi plikami dołączanymi w kodzie źródłowym. Zalecam wykorzystanie ostrzeżeń kompilatora do jawnego zamieniania wszystkich wystąpień zdezaktualizowanych funkcji na ich bezpieczniejsze zamienniki – i myślenie za każdym razem o możliwych przepełnieniach bufora, a w skrajnych sytuacjach łagodne kończenie działania aplikacji.

Każda istniejąca funkcja, jak `_tcscpy` lub `_tcscat`, ma odpowiadającą jej nową funkcję zaczynającą się od tej samej nazwy a zakończoną przyrostkiem `_s` (od słowa secure – bezpieczny). Wszystkie te nowe funkcje mają pewną wspólną cechę, która wymaga wyjaśnienia. Zacznijmy od przyjrzenia się ich prototypom na poniższym wycinku kodu pokazującym obok siebie definicje dwóch typowych funkcji łańcuchowych:

```
PTSTR   _tcscpy  (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscpy_s(PTSTR strDestination, size_t numberOfCharacters,
   PCTSTR strSource);

PTSTR   _tcscat  (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscat_s(PTSTR strDestination, size_t numberOfcharacters,
   PCTSTR strSource);
```

Gdy jako parametr przekazywany jest zapisywalny bufor, musi być też podany jego rozmiar. Oczekiwana jest tu wartość w postaci liczby znaków mieszczących się w danym buforze, którą można łatwo obliczyć, korzystając z makra `_countof` (zdefiniowanego w stdlib.h).

Wszystkie bezpieczne funkcje (`_s`) sprawdzają poprawność swoich argumentów zaraz na samym początku działania. Sprawdzenie jest dokonywane w celu upewnienia się, że wskaźniki nie są równe `NULL`, że liczby całkowite należą do odpowiedniego zakresu, że wartości wyliczeniowe są poprawne i że bufory są wystarczająco duże do pomieszczenia danych wynikowych. Jeśli któreś z tych sprawdzeń się nie powiedzie, funkcja ustawia wartość lokalnej dla wątku zmiennej uruchomieniowej C `errno`, a funkcja zwraca wartość `errno_t` wskazującą sukces lub porażkę. Jednakże w rzeczywistości funkcje te nie kończą normalnie swojego działania; zamiast tego aplikacje zbudowane w trybie debugowania wyświetlają nieprzyjazne dla użytkownika okno dialogowe podobne do pokazanego na rysunku 2-1. Następnie aplikacja kończy działanie. Aplikacje zbudowane w trybie wersji gotowej do opublikowania od razu kończą swoje działanie automatycznie.



Rysunek 2-1 Okno dialogowe wyświetlane przy wystąpieniu błędu

Środowisko uruchomieniowe języka C pozwala na zdefiniowanie własnej funkcji, która będzie wywoływana w przypadku wykrycia nieprawidłowego parametru. W funkcji tej można zapisać gdzieś informacje o błędzie, dołączyć debugera lub zrobić cokolwiek innego. Aby to umożliwić, trzeba najpierw zdefiniować funkcję odpowiadającą następującemu prototypowi:

```
void InvalidParameterHandler(PCTSTR expression, PCTSTR function,
   PCTSTR file, unsigned int line, uintptr_t /*pReserved*/);
```

Parametr `expression` opisuje niespełnione oczekiwanie w kodzie implementacji środowiska uruchomieniowego C, na przykład `(L"Buffer is too small" && 0)`. Jak widać nie jest to zbyt przyjazne dla użytkownika i nie powinno być wyświetlane. Komentarz ten odnosi się też do następnych trzech parametrów, ponieważ `function`, `file` i `line` opisują odpowiednio nazwę funkcji, plik kodu źródłowego i numer wiersza kodu źródłowego, w którym wystąpił błąd.

Uwaga Wszystkie te argumenty będą miały wartość `NULL`, jeśli nie zdefiniowano `DEBUG`. Ta procedura jest przydatna do śledzenia błędów tylko przy testowaniu wersji aplikacji zbudowanych w trybie debugowania. W wersjach zbudowanych w trybie do publikacji można by zastąpić okno dialogowe bardziej przyjaznym dla użytkownika komunikatem wyjaśniającym, że wystąpił niespodziewany błąd wymagający zamknięcia aplikacji – a następnie można by gdzieś zapisać informacje o błędzie lub uruchomić powtórnie aplikację. Jeśli stan pamięci aplikacji uległ uszkodzeniu, jej wykonywanie należy przerwać. Zaleca się jednak zaczekanie na sprawdzenie `errno_t` w celu zdecydowania, czy dany błąd można naprawić, czy nie.

Następnym krokiem jest zarejestrowanie tej procedury przez wywołanie `_set_invalid_parameter_handler`. Jednakże ten krok nie wystarczy, ponieważ okno dialogowe będzie się nadal pojawiać. Trzeba wywołać `_CrtSetReportMode(_CRT_ASSERT, 0);` na początku aplikacji, aby wyłączyć okna dialogowe wywoływane przez środowisko uruchomieniowe języka C.

Teraz przy wywoływaniu zastępczych funkcji łańcuchowych można sprawdzać zwracaną wartość `errno_t`, aby zrozumieć, co się stało. Jedynie wartość `S_OK` oznacza, że wywołanie się powiodło. Inne możliwe zwracane wartości, które można znaleźć w errno.h, jak `EINVAL`, dotyczą niepoprawnych argumentów, na przykład wskaźników `NULL`.

Przyjrzyjmy się przykładowi łańcucha kopiowanego do bufora za małego o jeden znak:

```
TCHAR szBefore[5] = {
   TEXT('B'), TEXT('B'), TEXT('B'), TEXT('B'), '\0'
};

TCHAR szBuffer[10] = {
   TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'),
   TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), '\0'
};

TCHAR szAfter[5] = {
   TEXT('A'), TEXT('A'), TEXT('A'), TEXT('A'), '\0'
};

errno_t result = _tcscpy_s(szBuffer, _countof(szBuffer),
   TEXT("0123456789"));
```

Tuż przed wywołaniem `_tcscpy_s` każda zmienna ma zawartość pokazaną na rysunku 2-2.

Watch 1

Name	Value	Type
szBefore	0x0012f9f4 "BBBB"	wchar_t [5]
szBuffer	0x0012f9d8 "---------"	wchar_t [10]
szAfter	0x0012f9c4 "AAAA"	wchar_t [5]

Rysunek 2-2 Stan zmiennych przed wywołaniem *_tcscpy_s*

Ponieważ łańcuch "1234567890", który ma być skopiowany do `szBuffer` ma dokładnie taki sam 10-znakowy rozmiar jak bufor, nie ma dość miejsca na skopiowanie końcowego znaku `'\0'`. Można by oczekiwać, że wartością `result` będzie teraz `STRUNCATE`, a ostatni znak `'9'` nie zostanie skopiowany, ale tak nie jest. Zwracana jest wartość `ERANGE`, a stan każdej ze zmiennych pokazano na rysunku 2-3.

Watch 1

Name	Value	Type
szBefore	0x0012f9f4 "BBBB"	wchar_t [5]
szBuffer	0x0012f9d8 ""	wchar_t [10]
szAfter	0x0012f9c4 "AAAA"	wchar_t [5]

Rysunek 2-3 Stan zmiennych po wywołaniu *_tcscpy_s*

Występuje pewien efekt uboczny, którego nie widać, jeśli nie spojrzy się na pamięć związaną z `szBuffer`, jak pokazano na rysunku 2-4.

```
Memory 1
Address: 0x0012F9D8    Columns: Auto
0x0012F9D8  00 00 fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd cc cc cc cc cc cc cc  ..ýýýýýýýýýýýýýýýýýýÌ
0x0012F9F3  cc 42 00 42 00 42 00 42 00 00 00 cc cc cc cc cc cc 6b fc c3 8e 18 fa 12 00 1a 25  ÌB.B.B.B...ÌÌÌÌÌÌküÃŽ
0x0012FA0E  40 00 11 01 00 00 00 00 00 00 40 fa 12 00 15 27 40 00 04 0c 05 00 e9 03 00 00 52  @.........@ú...'@....
0x0012FA29  0c 06 00 00 00 00 00 11 01 00 00 cc cc cc cc cc cc cc cc 11 01 00 00 6c fa 12 00  ...........ÌÌÌÌÌÌÌÌ..
```

Rysunek 2-4 Zawartość pamięci *szBuffer* po nieudanym wywołaniu funkcji

Pierwszy znak `szBuffer` został ustawiony na `'\0'`, a wszystkie pozostałe bajty zawierają teraz wartość `0xfd`. Łańcuch wynikowy został więc skrócony do pustego łańcucha, a pozostałe bajty bufora zostały ustawione na wartość wypełniającą (`0xfd`).

Uwaga Jeśli ktoś zastanawia się, dlaczego pamięć za wszystkimi definicjami zmiennych została wypełniona wartościami `0xcc` na rysunku 2-4, odpowiedź wynika z implementacji przez kompilator sprawdzeń automatycznie wykrywających przepełnienie bufora w trakcie działania aplikacji (`/RTCs`, `/RTCu` lub `/RTC1`). Jeśli kod zostanie skompilowany bez tych flag `/RTCx`, to obraz pamięci będzie przedstawiał wszystkie zmienne `sz*` obok siebie. Należy jednak pamiętać, aby zawsze kompilować aplikacje z włączonymi sprawdzeniami w celu wykrywania ewentualnych przepełnień bufora na wczesnym etapie cyklu programowania.

Jak uzyskać więcej kontroli przy wykonywaniu operacji na łańcuchach?

Poza nowymi, bezpiecznymi funkcjami łańcuchowymi biblioteka uruchomieniowa C ma kilka nowych funkcji zapewniających więcej kontroli nad wykonywaniem manipulacji na łańcuchach. Na przykład można określać wartości wypełniające lub sposób obcinania łańcuchów. Oczywiście biblioteka C oferuje zarówno wersje ANSI (A) jak i wersje Unicode (W) poszczególnych funkcji. Oto prototypy kilku spośród tych funkcji (istnieje ich więcej, ale nie zostały tutaj pokazane):

```
HRESULT StringCchCat(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCatEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);

HRESULT StringCchCopy(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCopyEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);

HRESULT StringCchPrintf(PTSTR pszDest, size_t cchDest,
   PCTSTR pszFormat, ...);
HRESULT StringCchPrintfEx(PTSTR pszDest, size_t cchDest,
   PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags,
   PCTSTR pszFormat,...);
```

Można zauważyć, że wszystkie pokazane metody zawierając w swoich nazwach „Cch”. Jest to skrót od „Count of characters” – liczba znaków, wartość tę zwykle uzyskuje się poprzez użycie makra `_countof`. Istnieje też zbiór funkcji mających w nazwie „Cb”, na przykład `StringCbCat(Ex)`, `StringCbCopy(Ex)` oraz `StringCbPrintf(Ex)`. Funkcje te oczekują, że argument oznaczający rozmiar jest podawany jako liczba bajtów, a nie liczba znaków. Zwykle używa się operatora `sizeof` do uzyskania tej wartości.

Wszystkie te funkcje zwracają `HRESULT` o wartości równej jednej z przedstawionych w tabeli 2-2.

Tabela 2-2 Wartości HRESULT zwracane przez bezpieczne funkcje łańcuchowe

Wartość HRESULT	Opis
S_OK	Powodzenie. Bufor docelowy zawiera łańcuch źródłowy zakończony '\0'.
STRSAFE_E_INVALID_PARAMETER	Błąd. Jako parametr przekazano wartość NULL.
STRSAFE_E_INSUFFICIENT_BUFFER	Błąd. Podany bufor docelowy był za mały do pomieszczenia całego łańcucha źródłowego.

W przeciwieństwie do funkcji bezpiecznych (z przyrostkiem _s), funkcje te w przypadku zbyt małego bufora dokonują obcięcia łańcucha. Można wykryć taką sytuację, gdy zwracana jest wartość STRSAFE_E_INSUFFICIENT_BUFFER. Jak można sprawdzić w pliku StrSafe.h, wartość tego kodu wynosi 0x8007007a i jest traktowana jako niepowodzenie przez makra SUCCEEDED/FAILED. Jednakże tutaj część bufora źródłowego, która zmieściła się w docelowym buforze zapisywalnym została skopiowana, a ostatni dostępny znak został ustawiony na '\0'. W poprzednim przykładzie zmienna szBuffer zawierałaby łańcuch "012345678" gdyby użyto StringCchCopy zamiast _tcscpy_s. Opcja obcinania łańcuchów może być w danym przypadku pożądana lub nie w zależności od tego, co się próbuje osiągnąć i dlatego jest traktowana domyślnie jako niepowodzenie. W razie ścieżki budowanej przez scalanie różnych fragmentów informacji obcięty wynik nie będzie użyteczny. Jeśli budowany jest komunikat dla użytkownika, obcinanie może być do przyjęcia. Od programisty zależy, jak potraktuje obcięty wynik.

Można też zauważyć, że dla wielu funkcji wcześniej pokazanych istnieją też wersje rozszerzone (Ex). Te wersje rozszerzone przyjmują trzy dodatkowe parametry, które opisano w tabeli 2-3.

Tabela 2-3 Parametry wersji rozszerzonych

Parametry i wartości	Opis
size_t* pcchRemaining	Wskaźnik do zmiennej określającej liczbę nieużywanych znaków w buforze docelowym. Skopiowany, końcowy znak '\0' nie jest liczony. Na przykład, jeśli jeden znak zostanie skopiowany do bufora 10-znakowego, zwrócone zostanie 9, choć bez obcinania nie będzie można użyć więcej niż 8 znaków. Liczba ta nie jest zwracana, jeśli pcchRemaining ma wartość NULL.
LPTSTR* ppszDestEnd	Jeśli parametr ppszDestEnd jest różny od NULL, wskazuje znak '\0' kończący łańcuch znaków w buforze docelowym.
DWORD dwFlags	Jedna lub więcej spośród poniższych wartości połączonych operatorem '|'.
STRSAFE_FILL_BEHIND_NULL	Jeśli funkcja zakończy się powodzeniem, dolny bajt dwFlags jest używany do wypełniania reszty bufora docelowego za końcowym znakiem '\0' (zobacz komentarz dotyczący STRSAFE_FILL_BYTE tuż za tą tabelą, aby dowiedzieć się więcej).

Tabela 2-3 Parametry wersji rozszerzonych

Parametry i wartości	Opis
`STRSAFE_IGNORE_NULLS`	Traktuje wskaźniki `NULL` do łańcuchów tak jak puste łańcuchy `(TEXT(""))`.
`STRSAFE_FILL_ON_FAILURE`	Jeśli funkcja zakończy się niepowodzeniem, dolny bajt `dwFlags` jest używany do wypełnienia całego bufora docelowego oprócz pierwszego znaku `'\0'` używanego do ustawienia wyniku w postaci pustego łańcucha znaków (zobacz komentarz dotyczący `STRSAFE_FILL_BYTE` tuż za tą tabelą, aby dowiedzieć się więcej). W razie niepowodzenia `STRSAFE_E_INSUFFICIENT_BUFFER` każdy znak w zwracanym łańcuchu jest zastępowany wartością bajta wypełniacza.
`STRSAFE_NULL_ON_FAILURE`	Jeśli funkcja zakończy się niepowodzeniem, pierwszy znak bufora docelowego jest ustawiany na `'\0'` w celu zdefiniowania łańcucha pustego `(TEXT(""))`. W wypadku niepowodzenia `STRSAFE_E_INSUFFICIENT_BUFFER` jakikolwiek obcięty łańcuch jest nadpisywany.
`STRSAFE_NO_TRUNCATION`	Tak jak w przypadku `STRSAFE_NULL_ON_FAILURE`, jeśli funkcja zakończy się niepowodzeniem, bufor docelowy jest ustawiany na pusty łańcuch `(TEXT(""))`. W razie niepowodzenia `STRSAFE_E_INSUFFICIENT_BUFFER` jakikolwiek obcięty łańcuch jest nadpisywany.

Uwaga Nawet jeśli użyto flagi `STRSAFE_NO_TRUNCATION`, znaki z łańcucha źródłowego są kopiowane aż do ostatniego dostępnego znaku w buforze docelowym. Następnie zarówno pierwszy jak i ostatni znak bufora docelowego są ustawiane na `'\0'`. Nie jest to takie istotne, chyba że ze względów bezpieczeństwa nie chce się zachowywać nieużywanych danych.

Trzeba wspomnieć o jeszcze jednym szczególe związanym z uwagą. Na rysunku 2-4 użyto wartości `0xfd` do zastąpienia wszystkich znaków za końcowym znakiem `'\0'` aż do końca bufora docelowego. W przypadku wersji `Ex` tych funkcji można wybrać, czy chce się skorzystać z tej operacji wypełniania (kosztownej, zwłaszcza gdy bufor docelowy jest duży), i jaka wartość bajtowa ma być użyta jako wypełniacz. Jeśli do parametru `dwFlag` dodane zostanie `STRSAFE_FILL_BEHIND_NULL`, pozostałe znaki zostaną ustawione na `'\0'`. Gdy zamiast `STRSAFE_FILL_BEHIND_NULL` użyte zostanie makro `STRSAFE_FILL_BYTE`, podana wartość bajtowa zostanie użyta do wypełnienia pozostałych miejsc w buforze docelowym.

Funkcje łańcuchowe Windows

System Windows również oferuje różne funkcje do manipulowania łańcuchami znaków. Wiele z tych funkcji, jak `lstrcat` i `lstrcpy`, się zdezaktualizowało, ponieważ nie wykrywają one problemów z przepełnieniem bufora. Plik ShlwApi.h definiuje kilka przydatnych funkcji łańcuchowych do formatowania wartości związanych z systemem operacyjnym, takich

jak `StrFormatKBSize` i `StrFormatByteSize`. Opis systemowych funkcji obsługujących łańcuchy można znaleźć pod adresem *http://msdn2.microsoft.com/en-us/library/ms538658.aspx*.

Często porównuje się łańcuchy w celu sprawdzenia ich identyczności lub posortowania. Najlepszymi funkcjami do tego celu są `CompareString(Ex)` i `CompareStringOrdinal`. Funkcji `CompareString(Ex)` używa się do porównywania łańcuchów w sposób poprawny językowo. Oto prototyp funkcji `CompareString`:

```
int CompareString(
   LCID locale,
   DWORD dwCmdFlags,
   PCTSTR pString1,
   int cch1,
   PCTSTR pString2,
   int cch2);
```

Funkcja ta porównuje dwa łańcuchy. Pierwszy jej parametr `CompareString` określa identyfikator lokalizacji (LCID) będący 32-bitową wartością identyfikującą określony język. `CompareString` wykorzystuje wartość LCID do porównywania dwóch łańcuchów, sprawdzając znaczenie poszczególnych znaków w danym języku. Językowo poprawne porównanie daje wyniki bardziej istotne dla użytkownika. Jednakże tego typu porównanie jest wolniejsze niż porównanie porządkowe. Identyfikator lokalizacji dla danego wątku można uzyskać, wywołując funkcję Windows `GetThreadLocale`:

```
LCID GetThreadLocale();
```

Drugi parametr `CompareString` określa flagi modyfikujące metodę używaną przez funkcję do porównywania dwóch łańcuchów znaków. Tabela 2-4 przedstawia możliwe flagi.

Tabela 2-4 Flagi używane przez funkcję CompareString

Flaga	Znaczenie
`NORM_IGNORECASE` `LINGUISTIC_IGNORECASE`	Ignoruj różnice pomiędzy małymi a wielkimi literami.
`NORM_IGNOREKANATYPE`	Nie odróżniaj znaków hiragana i katakana.
`NORM_IGNORENONSPACE` `LINGUISTIC_IGNOREDIACRITIC`	Ignoruj znaki diakrytyczne.
`NORM_IGNORESYMBOLS`	Ignoruj symbole.
`NORM_IGNOREWIDTH`	Nie odróżniaj znaku jednobajtowego od tego samego znaku w postaci dwubajtowej.
`SORT_STRINGSORT`	Traktuj znaki interpunkcyjne tak samo jak symbole.

Pozostałe cztery parametry funkcji `CompareString` określają oba łańcuchy znaków i ich długości w znakach (nie w bajtach). Jeśli wartość ujemna zostanie przekazana dla parametru `cch1`, funkcja przyjmie, że łańcuch `pString1` jest zakończony zerem i sama obliczy długość tego łańcucha. To samo odnosi się do parametru `cch2` i łańcucha `pString2`. Jeśli potrzebne są bardziej zaawansowane opcje językowe, należy przyjrzeć się funkcji `CompareStringEx`.

Do porównywania łańcuchów używanych w celach programowych (na przykład jako ścieżki dostępu, klucze/wartości rejestru, elementy/atrybuty XML, itd.) należy korzystać z funkcji `CompareStringOrdinal`:

```
int CompareStringOrdinal(
  PCWSTR pString1,
  int cchCount1,
  PCWSTR pString2,
  int cchCount2,
  BOOL bIgnoreCase);
```

Funkcja ta wykonuje porównywanie na poziomie kodów znaków bez uwzględniania lokalizacji językowej i dlatego działa szybko. Ponieważ łańcuchy programowe nie są zwykle wyświetlane użytkownikom, funkcja ta jest wówczas bardziej odpowiednia. Obsługuje ona łańcuchy Unicode.

Wartości zwracane przez funkcje `CompareString` i `CompareStringOrdinal` różnią się od wartości zwracanych przez funkcje porównywania łańcuchów `*cmp` z biblioteki uruchomieniowej języka C. `CompareString(Ordinal)` zwraca `0` dla oznaczenia niepowodzenia `CSTR_LESS_THAN` (zdefiniowane jako `1`) dla wskazania, że `pString1` jest mniejszy niż `pString2`, `CSTR_EQUAL` (zdefiniowane jako `2`) dla wskazania, ze `pString1` jest równy `pString2` oraz `CSTR_GREATER_THAN` (zdefiniowane jako `3`) dla wskazania, że `pString1` jest większy niż `pString2`. Dla wygody w razie powodzenia działania funkcji można odjąć `2` od zwróconej wartości, aby uzyskać wynik zgodny z wynikiem funkcji bibliotecznych C (`-1`, `0` i `+1`).

Dlaczego należy korzystać z Unicode?

Przy programowaniu aplikacji zdecydowanie polecamy korzystanie ze znaków i łańcuchów Unicode. Oto kilka powodów dlaczego:

- Unicode ułatwia lokalizowanie aplikacji na rynki światowe.
- Pozwala na rozprowadzanie pojedynczego pliku binarnego (.exe lub DLL) obsługującego wszystkie języki.
- Unicode poprawia efektywność aplikacji, ponieważ kod wykonuje się szybciej i zużywa mniej pamięci. Windows wykorzystuje wewnętrznie znaki i łańcuchy Unicode, więc przy przekazywaniu znaku lub łańcucha ANSI Windows musi przydzielić pamięć i zamienić znak lub łańcuch ANSI na jego odpowiednik w Unicode.
- Korzystanie z Unicode zapewnia, że aplikacja może łatwo wywoływać wszystkie aktualne funkcje Windows, ponieważ niektóre z nich oferują wersje działające tylko ze znakami i łańcuchami Unicode.
- Korzystanie z Unicode zapewnia tworzonemu kodowi łatwą integrację z COM (wymagającym używania znaków i łańcuchów Unicode).
- Korzystanie z tej funkcji zapewnia tworzonemu kodowi łatwą integrację z .NET Framework (również wymagającym używania znaków i łańcuchów Unicode).
- Korzystanie z Unicode zapewnia, że kod może łatwo manipulować własnymi zasobami (gdzie łańcuchy są zawsze zapisywane jako Unicode).

Jak zalecamy pracować ze znakami i łańcuchami?

W oparciu o to, co można było przeczytać w tym rozdziale, w części tej podsumujemy, o czym należy zawsze pamiętać przy tworzeniu własnego kodu. Następnie przedstawimy garść wskazówek dotyczących lepszego manipulowania łańcuchami Unicode i ANSI. Dobrze jest zacząć konwertować aplikacje tak, aby były gotowe na Unicode nawet, jeśli nie planuje się użycia Unicode od razu. Oto kilka podstawowych wskazówek, którymi należy się kierować:

- Należy zacząć myśleć o łańcuchach tekstowych jak o tablicach znaków, a nie jak o tablicach bajtów lub wartości `char`.
- Trzeba korzystać z ogólnych typów danych (takich jak `TCHAR/PTSTR`) dla znaków i łańcuchów tekstowych.
- Należy korzystać z określonych typów danych (takich jak `BYTE` i `PBYTE`) dla bajtów, wskaźników do bajtów i buforów danych.
- Powinno się korzystać z makra `TEXT` lub `_T` do tworzenia stałych znakowych lub łańcuchowych, ale unikać mieszania obu z nich dla zachowania spójności i lepszej czytelności.
- Należy wykonywać globalne zamiany (na przykład zastąpienie `PSTR` przez `PTSTR`).
- Trzeba uwzględniać zmiany w problemach arytmetycznych związanych z łańcuchami. Na przykład funkcje zwykle oczekują przekazania wielkości bufora w znakach, a nie w bajtach. To oznacza, że należy przekazywać `_countof(szBuffer)` zamiast `sizeof(szBuffer)`. Jeśli trzeba przydzielić blok pamięci na łańcuch i znana jest liczba znaków w łańcuchu, to należy mieć na uwadze, że pamięć alokuje się w bajtach. Oznacza to, że należy wywołać `malloc(nCharacters * sizeof(TCHAR))`, a nie `malloc(nCharacters)`. Z wszystkich moich wskazówek ta jest najtrudniejsza do zapamiętania, a kompilator nie oferuje ostrzeżeń lub błędów w razie pomyłki. To dobra okazja do zdefiniowania własnych makr, jak na przykład:

  ```
  #define chmalloc(nCharacters)  (TCHAR*)malloc(nCharacters * sizeof(TCHAR)).
  ```
- Należy unikać funkcji z rodziny `printf`, zwłaszcza z użyciem typów pól %s i %S do konwersji łańcuchów ANSI na Unicode i vice versa. Zamiast tego powinno się korzystać z funkcji `MultiByteToWideChar` i `WideCharToMultiByte`, jak pokazano w dalszej części tego rozdziału.
- Zawsze należy definiować oba symbole `UNICODE` i `_UNICODE` lub nie definiować żadnego z nich.

Oto podstawowe zasady, którymi należy się kierować, wykorzystując funkcje manipulujące łańcuchami:

- Należy zawsze pracować z bezpiecznymi funkcjami manipulacji na łańcuchach (na przykład tymi zakończonymi na _s lub poprzedzonymi przedrostkiem StringCch. Trzeba korzystać z tych ostatnich w celu obsługi jawnego obcinania łańcuchów, a w przeciwnych wypadkach należy raczej korzystać z tych poprzednich.
- Nie powinno się używać niebezpiecznych funkcji manipulujących łańcuchami, pochodzących z biblioteki uruchomieniowej języka C (patrz poprzednia zasada). Bardziej ogólnie: nie należy korzystać ani implementować żadnych procedur manipulowania

buforem, które nie mają jako parametru rozmiaru bufora docelowego. Biblioteka uruchomieniowa C zawiera zastępniki dla funkcji manipulowania buforem takie jak `memcpy_s`, `memmove_s`, `wmemcpy_s` lub `wmemmove_s`. Wszystkie te metody są dostępne, gdy zdefiniowany jest symbol `__STDC_WANT_SECURE_LIB__`, co jest domyślnym ustawieniem w pliku CrtDefs.h. Nie należy więc usuwać definicji `__STDC_WANT_SECURE_LIB__`.

- Trzeba korzystać z flag kompilatora `/GS` (http://msdn2.microsoft.com/en-us/library/aa290051(VS.71).aspx) oraz `/RTCs` w celu automatycznego wykrywania przepełnień bufora.
- Nie należy korzystać z udostępnianych przez Kernel32 metod manipulowania łańcuchami, takich jak `lstrcat` i `lstrcpy`.
- Istnieją dwa rodzaje łańcuchów tekstowych, które porównuje się w kodzie. Łańcuchy programistyczne to nazwy plików, ścieżki dostępu, elementy i atrybuty XML oraz klucze/wartości rejestru. W tym przypadku warto korzystać z `CompareStringOrdinal`, jako że jest to najszybszy sposób i nie musi być brane pod uwagę ustawienie językowe. Łańcuchy te pozostają takie same bez względu, gdzie na świecie działa dana aplikacja. Łańcuchy użytkownika są typowymi łańcuchami pojawiającymi się w interfejsie użytkownika. W ich wypadku należy wywoływać `CompareString(Ex)`, co bierze pod uwagę lokalizację przy porównywaniu łańcuchów.

Nie ma wyboru: będąc profesjonalnym programistą, nie można pisać kodu w oparciu o niebezpieczne funkcje manipulujące buforem. Z tego powodu cały kod w tej książce opiera się na tych bezpieczniejszych funkcjach z biblioteki uruchomieniowej języka C.

Tłumaczenie łańcuchów pomiędzy Unicode a ANSI

Do zamiany wielobajtowych łańcuchów znakowych na szerokoznakowe łańcuchy należy wykorzystać funkcję `MultiByteToWideChar`. Funkcję tę pokazano poniżej:

```
int MultiByteToWideChar(
   UINT uCodePage,
   DWORD dwFlags,
   PCSTR pMultiByteStr,
   int cbMultiByte,
   PWSTR pWideCharStr,
   int cchWideChar);
```

Parametr `uCodePage` identyfikuje numer strony kodowej związanej z łańcuchem wielobajtowym. Parametr `dwFlags` umożliwia określenie dodatkowych opcji wpływających na znaki z symbolami diakrytycznymi. Zwykle flag się nie używa i 0 przekazywane jest w parametrze `dwFlags` (więcej szczegółów dotyczących możliwych wartości tej flagi można znaleźć w serwisie MSDN online pod adresem http://msdn2.microsoft.com/en-us/library/ms776413.aspx). Parametr `pMultiByteStr` określa łańcuch, który ma być zamieniany, a `cbMultiByte` określa długość łańcucha (w bajtach). Funkcja automatycznie określa długość łańcucha źródłowego, jeśli wartość –1 zostanie przekazana dla parametru `cbMultiByte`.

Wersja Unicode łańcucha wynikającego z konwersji jest zapisywana do bufora zlokalizowanego w pamięci pod adresem określonym przez parametr `pWideCharStr`. W parametrze `cchWideChar` trzeba podać maksymalny rozmiar tego bufora (w znakach). Przy wywołaniu `MultiByteToWideChar` i przekazaniu wartości `0` jako parametru `cchWideChar` funkcja nie

dokona konwersji, a zamiast tego zwróci liczbę szerokich znaków (łącznie z końcowym znakiem `'\0'`), które muszą być zapewnione przez bufor, aby konwersja się powiodła. Zazwyczaj konwertuje się łańcuch wielobajtowych znaków na jego odpowiednik Unicode, wykonując następujące kroki:

1. Wywołanie `MultiByteToWideChar` z przekazaną wartością `NULL` dla parametru `pWideCharStr`, wartością `0` dla parametru `cchWideChar` oraz wartością –1 dla parametru `cbMultiByte`.
2. Zaalokowanie bloku pamięci wystarczająco dużego, żeby pomieścić przekonwertowany łańcuch Unicode. Rozmiar ten jest obliczany na podstawie wartości zwróconej przez poprzednie wywołanie `MultiByteToWideChar` pomnożonej przez `sizeof(wchar_t)`.
3. Ponowne wywołanie `MultiByteToWideChar`, tym razem z przekazanym adresem bufora dla parametru `pWideCharStr` oraz rozmiarem obliczonym w oparciu o wartość zwróconą przez pierwsze wywołanie `MultiByteToWideChar` pomnożoną przez `sizeof(wchar_t)` dla parametru `cchWideChar`.
4. Użycie przekonwertowanego łańcucha znaków.
5. Zwolnienie bloku pamięci zajmowanego przez łańcuch Unicode.

Funkcja `WideCharToMultiByte` konwertuje łańcuch szerokich znaków na jego odpowiednik wielobajtowy, jak pokazano poniżej:

```
int WideCharToMultiByte(
   UINT uCodePage,
   DWORD dwFlags,
   PCWSTR pWideCharStr,
   int cchWideChar,
   PSTR pMultiByteStr,
   int cbMultiByte,
   PCSTR pDefaultChar,
   PBOOL pfUsedDefaultChar);
```

Funkcja ta jest podobna do funkcji `MultiByteToWideChar`. Parametr `uCodePage` ponownie identyfikuje stronę kodową, która ma być związana z nowo przekonwertowanym łańcuchem. Parametr `dwFlags` pozwala określić dodatkowe opcje konwersji. Flagi wpływają na znaki z symbolami diakrytycznymi i znaki, których system nie jest w stanie przekonwertować. Zwykle taki poziom kontroli konwersji nie będzie potrzebny i wystarczy przekazać wartość `0` jako parametr `dwFlags`.

Parametr `pWideCharStr` określa adres pamięci dla konwertowanego łańcucha, a parametr `cchWideChar` określa długość tego łańcucha (w znakach). Funkcja sama określi długość łańcucha źródłowego, jeśli dla parametru `cchWideChar` zostanie przekazana wartość –1.

Wielobajtowa wersja łańcucha wynikająca z konwersji jest zapisywana w buforze określanym przez parametr `pMultiByteStr`. W parametrze `cbMultiByte` trzeba podać maksymalny rozmiar tego bufora (w bajtach). Przekazanie wartości `0` jako parametru `cbMultiByte` funkcji `WideCharToMultiByte` spowoduje, że funkcja zwróci rozmiar wymagany dla bufora docelowego. Zwykle konwertuje się łańcuch szerokich znaków na łańcuch znaków wielobajtowych, korzystając z sekwencji kroków podobnej do omówionej wcześniej przy konwertowaniu łańcucha wielobajtowego na łańcuch szerokich znaków, z wyjątkiem tego, że zwracana wartość bezpośrednio oznacza liczbę bajtów wymaganych dla powodzenia konwersji.

Można zauważyć, że funkcja `WideCharToMultiByte` przyjmuje o dwa parametry więcej niż funkcja `MultiByteToWideChar`: `pDefaultChar` i `pfUsedDefaultChar`. Parametry te są używane przez funkcję `WideCharToMultiByte` jedynie wtedy, gdy natrafi ona na szeroki znak, który nie jest reprezentowany na stronie kodowej określonej przez parametr `uCodePage`. Jeśli szerokiego znaku nie można przekonwertować, funkcja użyje znaku określanego przez parametr `pDefaultChar`. Jeśli parametr ten jest równy `NULL`, co się najczęściej stosuje, funkcja wykorzysta systemowy znak domyślny. Tym znakiem jest zwykle znak zapytania. Takie zachowanie może być niepożądane w razie nazw plików, ponieważ znak zapytania ma w nich znaczenie specjalne.

Parametr `pfUsedDefaultChar` wskazuje zmienną logiczną, której wartość funkcja ustawi na `TRUE`, jeśli przynajmniej jeden znak łańcucha szerokich znaków nie będzie mógł być przekonwertowany na swój odpowiednik wielobajtowy. Funkcja ustawi wartość tej zmiennej na `FALSE`, jeśli wszystkie znaki zostaną przekonwertowane bez problemów. W większości sytuacji jako wartość tego parametru przekazuje się `NULL`.

Kompletny opis użycia tych funkcji można znaleźć w dokumentacji Platform SDK.

Eksportowanie funkcji ANSI i Unicode z bibliotek DLL

Tych dwóch funkcji można używać do łatwego tworzenia wersji Unicode i ANSI własnych funkcji. Na przykład można by utworzyć dynamicznie dołączaną bibliotekę zawierającą funkcję odwracającą kolejność wszystkich znaków w łańcuchu. Wersję Unicode tej funkcji można by napisać w ten sposób:

```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength) {

   // Pobierz wskaźnik do ostatniego znaku łańcucha.
   PWSTR pEndOfStr = pWideCharStr + wcsnlen_s(pWideCharStr , cchLength) - 1;
   wchar_t cCharT;
   // Powtarzaj aż do dotarcia do środkowego znaku w łańcuchu.
   while (pWideCharStr < pEndOfStr) {
      // Zapisz znak w zmiennej tymczasowej.
      cCharT = *pWideCharStr;

      // Wstaw ostatni znak w miejsce pierwszego znaku.
      *pWideCharStr = *pEndOfStr;

      // Wstaw tymczasowy znak w miejsce ostatniego znaku.
      *pEndOfStr = cCharT;

      // Przejdź do kolejnego znaku z lewej strony.
      pWideCharStr++;

      // Przejdź do kolejnego znaku z prawej strony.
      pEndOfStr--;
   }

   // Znaki łańcucha są odwrócone, zwróć informację o powodzeniu działania funkcji.
   return(TRUE);
}
```

Wersję ANSI tej funkcji można by napisać tak, że nie będzie ona w ogóle wykonywać sama zamiany kolejności znaków w łańcuchu. Zamiast tego może konwertować łańcuch

ANSI na Unicode, przekazywać łańcuch Unicode do funkcji `StringReverseW`, a następnie konwertować odwrócony łańcuch znaków z powrotem na ANSI. Funkcja ta wyglądałaby następująco:

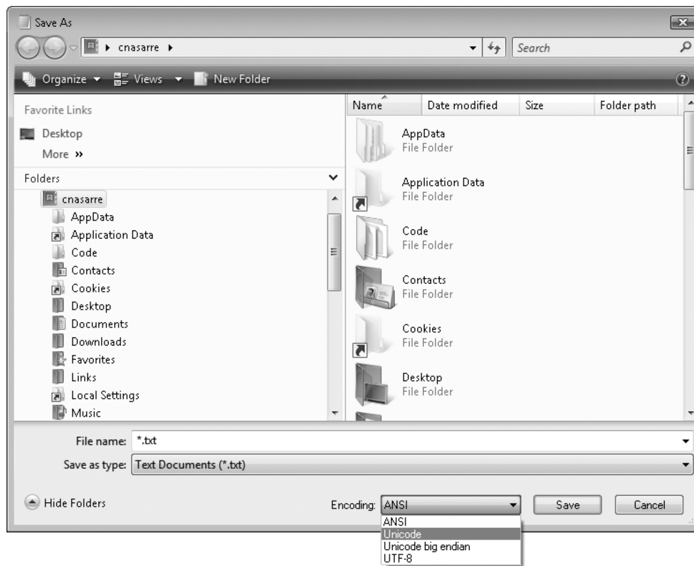
```
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength) {
   PWSTR pWideCharStr;
   int nLenOfWideCharStr;
   BOOL fOk = FALSE;

   // Oblicz liczbę znaków potrzebnych do przechowywania
   // wersji łańcucha z szerokimi znakami.
   nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
      pMultiByteStr, cchLength, NULL, 0);

   // Przydziel pamięć z domyślnej sterty procesu
   // o rozmiarze odpowiednim dla łańcucha szerokich znaków.
   // Nie zapominaj, że MultiByteToWideChar zwraca
   // liczbę znaków, a nie liczbę bajtów, więc
   // trzeba ją pomnożyć przez rozmiar szerokiego znaku.
   pWideCharStr = (PWSTR)HeapAlloc(GetProcessHeap(), 0,
      nLenOfWideCharStr * sizeof(wchar_t));

   if (pWideCharStr == NULL)
      return(fOk);

   // Konwertuj łańcuch wielobajtowy na łańcuch szerokich znaków.
   MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, cchLength,
      pWideCharStr, nLenOfWideCharStr);

   // Wywołaj wersję Unicode tej funkcji do wykonania
   // faktycznej pracy.
   fOk = StringReverseW(pWideCharStr, cchLength);

   if (fOk) {
      // Przekonwertuj łańcuch szerokich znaków z powrotem
      // na łańcuch wielobajtowy.
      WideCharToMultiByte(CP_ACP, 0, pWideCharStr, cchLength,
         pMultiByteStr, (int)strlen(pMultiByteStr), NULL, NULL);
   }

   // Zwolnij pamięć zawierającą łańcuch szerokich znaków.
   HeapFree(GetProcessHeap(), 0, pWideCharStr);

   return(fOk);
}
```

Na koniec w pliku nagłówkowym rozprowadzanym z dynamicznie dołączaną biblioteką należy zdefiniować prototypy obu funkcji w następujący sposób:

```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength);
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength);

#ifdef UNICODE
#define StringReverse StringReverseW
#else
#define StringReverse StringReverseA
#endif // !UNICODE
```

Określanie, czy dany tekst jest w standardzie ANSI, czy Unicode

Aplikacja Notatnik (Notepad) w systemie Windows pozwala otwierać zarówno pliki Unicode, jak i ANSI oraz tworzyć pliki w obu wersjach. Rysunek 2-5 przedstawia okno dialogowe Zapisz jako Notatnika. Warto zwrócić uwagę na różne sposoby zapisania pliku tekstowego.



Rysunek 2-5 Okno dialogowe Zapisz jako programu Notatnik w systemie Windows Vista

Dla wielu aplikacji (takich jak kompilatory) otwierających i przetwarzających pliki tekstowe byłoby wygodnie, gdyby po otwarciu pliki mogły określić, czy dany plik tekstowy zawiera znaki ANSI, czy Unicode. Funkcja `IsTextUnicode` eksportowana przez AdvApi32.dll i zadeklarowana w WinBase.h może pomóc w dokonaniu tego rozróżnienia:

```
BOOL IsTextUnicode(CONST PVOID pvBuffer, int cb, PINT pResult);
```

Problem z plikami tekstowymi polega na tym, że nie ma ścisłych i szybkich reguł dotyczących badania ich zawartości. Znacznie utrudnia to określenie, czy dany plik zawiera znaki ANSI, czy Unicode. Funkcja `IsTextUnicode` wykorzystuje ciąg statystycznych i deterministycznych metod w celu ustalenia rodzaju zawartości bufora. Ponieważ nie jest to ścisłe sprawdzenie, możliwe jest, że funkcja `IsTextUnicode` zwróci nieprawidłowy wynik.

Pierwszy parametr `pvBuffer` określa adres bufora, którego zawartość ma być sprawdzona. Typem danych jest wskaźnik void, ponieważ nie wiadomo z góry, czy mamy do czynienia z tablicą znaków ANSI, czy tablicą znaków Unicode.

Drugi parametr `cb` określa liczbę bajtów w buforze wskazywanym przez `pvBuffer`. Znowu, ponieważ nie wiemy, co jest w buforze, `cb` jest liczbą bajtów, a nie liczbą znaków. Należy zwrócić uwagę, że nie trzeba podawać całej długości bufora. Oczywiście, im więcej bajtów zbada funkcja `IsTextUnicode` tym dokładniejszy wynik będzie mogła uzyskać.

Trzeci parametr `pResult` jest adresem liczby całkowitej, którą trzeba zainicjować przed wywołaniem funkcji `IsTextUnicode`. Liczbę tę inicjuje się, aby wskazać, jakie sprawdzenia ma wykonać funkcja `IsTextUnicode`. Można też przekazać wartość `NULL` dla tego parametru, aby funkcja `IsTextUnicode` wykonała wszystkie możliwe sprawdzenia (więcej szczegółów na ten temat można znaleźć w dokumentacji Platform SDK).

Jeśli funkcja `IsTextUnicode` uzna, że bufor zawiera tekst Unicode, zwróci wartość `TRUE`; w przeciwnym razie zwróci wartość `FALSE`. Jeśli zażądano określonych sprawdzeń w liczbie wskazywanej przez parametr `pResult`, funkcja ustawi bity tej liczby całkowitej przed zwróceniem wyniku odpowiednio do wyników poszczególnych sprawdzeń.

Aplikacja przykładowa FileRev zaprezentowana w rozdziale 17 „Pliki mapowane w pamięci” demonstruje zastosowanie funkcji `IsTextUnicode`.

Rozdział 3

Obiekty jądra

W tym rozdziale:

Czym jest obiekt jądra? ... 35
Tabela uchwytów obiektów jądra dla danego procesu ... 39
Współdzielenie obiektów jądra pomiędzy granicami procesów ... 45

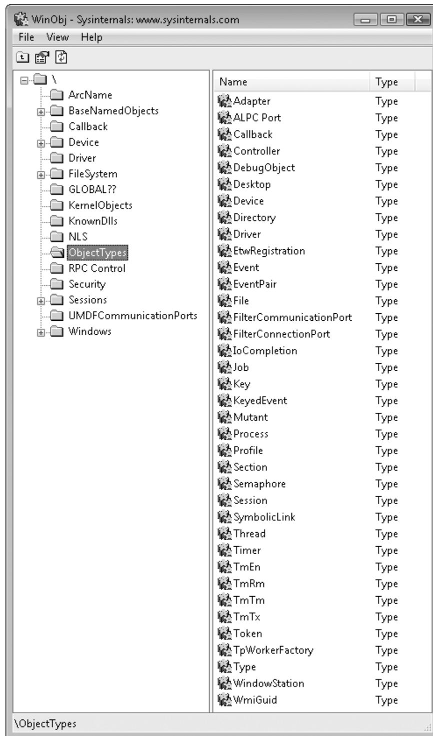
Zaczniemy poznawanie interfejsu programowania aplikacji (API) dla systemu Microsoft Windows od zapoznania się z obiektami jądra i ich uchwytami. Ten rozdział będzie zajmować się dość abstrakcyjnymi pojęciami – nie zamierzam tutaj omawiać szczegółów konkretnych obiektów jądra. Omówię natomiast cechy dotyczące wszystkich obiektów jądra.

Wolałbym zacząć od bardziej konkretnego tematu, ale porządne zrozumienie obiektów jądra jest istotne, by zostać biegłym programistą Windows. Obiekty jądra są używane przez system i aplikacje do zarządzania wieloma zasobami, takimi jak procesy, wątki i pliki (oraz wieloma innymi). Pojęcia przedstawione w tym rozdziale będą pojawiać się w pozostałych rozdziałach tej książki. Zdaję sobie jednak sprawę, że trudno będzie sobie przyswoić część materiału omawianego tutaj, dopóki nie zacznie się manipulować obiektami jądra przy użyciu konkretnych funkcji. Podczas czytania dalszych rozdziałów książki warto więc będzie od czasu do czasu powrócić do tego rozdziału.

Czym jest obiekt jądra?

Będąc programistą Windows, regularnie tworzy się, otwiera i przetwarza obiekty jądra. System tworzy i przetwarza różne typy obiektów jądra, na przykład obiekty plików, mapowań plików, portów wejścia/wyjścia, zadań, skrzynek pocztowych, muteksów, procesów, semaforów, wątków, zegarów, fabryki puli wątków. Darmowe narzędzie WinObj z firmy Sysinternals (dostępne pod adresem http://www.microsoft.com/technet/sysinternals/utilities/winobj.mspx) pozwala na przeglądanie listy wszystkich typów obiektów jądra. Trzeba je uruchomić z poziomu konta administratora, aby móc oglądać listę taką, jak pokazana na rysunku na stronie następnej.

Obiekty te są tworzone przez wywoływania różnych funkcji, których nazwy nie zawsze odpowiadają wprost typom używanych obiektów jądra. Na przykład funkcja `CreateFileMapping` powoduje utworzenie przez system mapowania pliku odpowiadającego obiektowi prezentowanemu jako `Section` przez narzędzie WinObj. Każdy obiekt jądra jest po prostu blokiem pamięci przydzielanym przez jądro systemu operacyjnego i dostępnym tylko poprzez jądro. Ten blok pamięci jest strukturą danych, której elementy przechowują informacje o danym obiekcie. Niektóre elementy (deskryptor zabezpieczeń, licznik użycia, itd.) są takie same we wszystkich typach obiektów, ale większość jest specyficzna dla danego typu obiektu. Na przykład obiekt procesu ma identyfikator procesu, priorytet bazowy i kod wyjścia, natomiast obiekt pliku ma wartość przesunięcia, tryb udostępniania i tryb otwarcia.



Ponieważ struktury danych obiektów jądra dostępne są tylko poprzez jądro, nie da się w aplikacjach odnaleźć położenia tych struktur danych w pamięci i bezpośrednio zmieniać ich zawartości. Microsoft wprowadził to ograniczenie celowo, aby zapewnić strukturom obiektów jądra możliwość zachowania spójnego stanu. Ograniczenie to pozwala firmie Microsoft na dodawanie, usuwanie lub zmienianie elementów tych struktur bez uszkadzania aplikacji.

Skoro nie można zmieniać tych struktur bezpośrednio, jak aplikacje mogą manipulować obiektami jądra? Windows oferuje zestaw funkcji do manipulowania tymi strukturami w określony sposób. Obiekty jądra są zawsze dostępne poprzez te funkcje. Przy wywoływaniu funkcji tworzącej obiekt jądra funkcja ta zwraca uchwyt identyfikujący dany obiekt. Uchwyt ten można traktować jako pewną wartość, która może być używana przez dowolny wątek procesu. Uchwyt jest wartością 32-bitową w przypadku systemu 32-bitowego, a 64-bitową w przypadku systemu 64-bitowego. Uchwyt ten można przekazywać do różnych funkcji Windows, aby system wiedział, którym obiektem jądra dana funkcja ma się zająć. Na temat uchwytów powiem nieco więcej w dalszej części tego rozdziału.

Aby działanie systemu operacyjnego było sprawne, wartości uchwytów są względne na poziomie danego procesu. Gdyby więc wartość uchwytu została przekazana do innego procesu (przy użyciu jakiejś formy komunikacji między procesami), wywołania dokonywane przez ten inny proces z użyciem wartości uchwytu z procesu pierwotnego się nie powiodą lub – co gorsza – będą odwoływać się do zupełnie innego obiektu jądra znajdującego się akurat na tej samej pozycji w tabeli uchwytów danego procesu. W części „Współdzielenie obiektów jądra pomiędzy granicami procesów" przyjrzymy się trzem mechanizmom pozwalającym wielu procesom na współdzielenie tego samego obiektu jądra.

Zliczanie użyć

Obiekty jądra są w posiadaniu jądra, a nie procesu. Innymi słowy, jeśli proces wywoła funkcję tworzącą obiekt jądra, a następnie proces się zakończy, obiekt jądra niekoniecznie zostanie zniszczony. Na ogół obiekt zostanie faktycznie zniszczony, ale jeśli inny proces będzie wykorzystywać obiekt jądra utworzony w poprzednim procesie, jądro będzie wiedziało, że nie ma niszczyć obiektu, dopóki ten inny proces nie przestanie z niego korzystać. Trzeba zawsze pamiętać, że obiekt jądra może przetrwać dłużej niż proces, który go utworzył.

Jądro wie, ile procesów korzysta z jego obiektu określonego, ponieważ każdy obiekt zawiera licznik użycia. Licznik użycia jest jednym z elementów danych wspólnym dla wszystkich typów obiektów jądra. Gdy obiekt jest tworzony, jego licznik użycia jest ustawiany na 1. Gdy inny proces uzyska dostęp do istniejącego obiektu jądra, licznik użycia jest zwiększany o 1. Gdy proces kończy działanie, jądro automatycznie zmniejsza o 1 liczniki użycia dla wszystkich swoich obiektów otwartych w danym procesie. Jeśli licznik użycia obiektu spadnie do 0, jądro niszczy ten obiekt. Dzięki temu żaden obiekt jądra nie pozostanie w systemie, jeśli żaden proces nie będzie się do niego odwoływać.

Bezpieczeństwo

Obiekty jądra mogą być chronione przez deskryptory zabezpieczeń. Deskryptor zabezpieczeń opisuje, kto jest właścicielem obiektu (zwykle jego twórca), która grupa użytkowników może mieć dostęp do obiektu, a która ma zabroniony dostęp do obiektu. Deskryptory zabezpieczeń są zwykle używane przy pisaniu aplikacji serwerowych. Jednakże w przypadku systemu Microsoft Windows Vista funkcja ta staje się bardziej widoczna dla aplikacji klienckich z prywatnymi przestrzeniami nazw, co zobaczymy później w tym rozdziale i w następnym w części zatytułowanej „Gdy administrator działa z uprawnieniami zwykłego użytkownika".

Prawie wszystkie funkcje, które tworzą obiekty jądra, przyjmują jako jeden z argumentów wskaźnik do struktury `SECURITY_ATTRIBUTES`, jak pokazano tutaj na przykładzie funkcji `CreateFileMapping`:

```
HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);
```

Większość aplikacji po prostu przekazuje `NULL` dla tego argumentu, co sprawia, że obiekt jest tworzony z domyślnymi zabezpieczeniami opartymi na kontekście bezpieczeństwa dla bieżącego procesu. Można jednak zaalokować strukturę `SECURITY_ATTRIBUTES`, zainicjować ją i przekazać jej adres jako ten parametr. Struktura `SECURITY_ATTRIBUTES` ma postać:

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength;
   LPVOID lpSecurityDescriptor;
   BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Choć struktura ta nosi związaną z bezpieczeństwem nazwę `SECURITY_ATTRIBUTES`, tak naprawdę zawiera tylko jeden element mający jakiś związek z zabezpieczeniami: `lpSecurityDescriptor`. Chcąc ograniczyć dostęp do tworzonego obiektu jądra, trzeba utworzyć deskryptor zabezpieczeń, a następnie zainicjować strukturę `SECURITY_ATTRIBUTES` w następujący sposób:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);         // Używane dla zgodności z różnymi wersjami
struktury
sa.lpSecurityDescriptor = pSD;   // Adres zainicjowanego deskryptora zabezpieczeń
sa.bInheritHandle = FALSE;       // To omówimy później
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,
   PAGE_READWRITE, 0, 1024, TEXT("MyFileMapping"));
```

Omówienie elementu `bInheritHandle` odłożę na później do części „Korzystanie z dziedziczenia uchwytów do obiektów", ponieważ element ten nie ma nic wspólnego z bezpieczeństwem.

Chcąc uzyskać dostęp do istniejącego obiektu jądra (zamiast tworzyć nowy), trzeba określić operacje, jakie zamierza się wykonać na tym obiekcie. Na przykład, jeśli chce się uzyskać dostęp do istniejącego obiektu jądra dla mapowania pliku, aby móc z niego odczytać dane, trzeba wywołać `OpenFileMapping`, jak pokazano poniżej:

```
HANDLE hFileMapping = OpenFileMapping(FILE_MAP_READ, FALSE,
   TEXT("MyFileMapping"));
```

Poprzez przekazanie `FILE_MAP_READ` jako pierwszego parametru funkcji `OpenFileMapping` wskazujemy, że chcemy czytać z tego mapowania pliku po uzyskaniu do niego dostępu. Funkcja `OpenFileMapping` dokonuje najpierw sprawdzenia zabezpieczeń, zanim zwróci prawidłową wartość uchwytu. Jeśli zalogowany użytkownik ma prawa dostępu do istniejącego obiektu jądra dla mapowania pliku, funkcja `OpenFileMapping` zwróci prawidłowy uchwyt. Jeśli jednak nastąpi odmowa dostępu, funkcja `OpenFileMapping` zwróci `NULL`, a wywołanie `GetLastError` zwróci wartość 5 (`ERROR_ACCESS_DENIED`). Nie należy zapominać, że jeśli zwrócony uchwyt zostanie użyty do wywołania funkcji wymagających innych uprawnień niż `FILE_MAP_READ`, również wystąpi błąd „odmowy dostępu". Ponieważ większość aplikacji nie wykorzystuje tych funkcji zabezpieczeń, nie będę się dalej zagłębiał w te kwestie.

Chociaż wiele aplikacji nie musi się przejmować zabezpieczeniami, wiele funkcji Windows wymaga przekazywania im wymaganych informacji dotyczących zabezpieczenia dostępu. Kilka aplikacji zaprojektowanych dla poprzednich wersji Windows nie zadziałało poprawnie w Windows Vista, ponieważ przy implementacji nie uwzględniały wystarczająco kwestii zabezpieczeń.

Na przykład wyobraźmy sobie aplikację, która po uruchomieniu wczytuje pewne dane z podklucza rejestru. Aby wykonać to poprawnie, kod powinien wywołać funkcję `RegOpenKeyEx,` przekazując jej wartość `KEY_QUERY_VALUE` jako żądany tryb dostępu.

Jednakże wiele aplikacji było pierwotnie projektowanych dla systemów sprzed Windows 2000 bez jakiegokolwiek zastanawiania się nad zabezpieczeniami. Niektórzy mogli wywoływać funkcję `RegOpenKeyEx,` przekazując jej `KEY_ALL_ACCESS` jako żądany tryb dostępu. Programiści korzystali z tego podejścia, ponieważ było to prostsze rozwiązanie i nie wymagało zastanawiania się, jaki tryb dostępu jest wymagany. Problem w tym, że dla danego podklucza rejestru może być możliwy odczyt, ale nie zapis dla użytkownika nie będącego administratorem. Wtedy, jeśli aplikacja będzie działać w systemie Windows Vista,

wywołanie RegOpenKeyEx z parametrem KEY_ALL_ACCESS się nie powiedzie, a bez odpowiedniego wykrywania błędów, aplikacja może działać zupełnie nieprzewidywalnie.

Gdyby programista pomyślał choć trochę o zabezpieczeniach i zmienił parametr KEY_ALL_ACCESS na KEY_QUERY_VALUE (co w tym przypadku w zupełności wystarczy), aplikacja działałaby w każdej wersji systemu operacyjnego.

Zaniedbywanie ustawiania właściwych flag dostępu jest jednym z błędów najczęściej popełnianych przez programistów. Używanie odpowiednich flag ułatwi z pewnością przenoszenie aplikacji pomiędzy różnymi wersjami Windows. Trzeba też sobie zdawać sprawę, że każda nowa wersja Windows przynosi ze sobą nowy zestaw ograniczeń, które nie istniały w poprzednich wersjach. Na przykład w Windows Vista trzeba uwzględniać funkcję kontroli konta użytkownika (User Access Control – UAC). Domyślnie funkcja ta zmusza aplikacje do działania w ograniczonym kontekście zabezpieczeń, nawet jeśli bieżący użytkownik należy do grupy administratorów. Funkcją UAC zajmiemy się dokładniej w rozdziale 4 „Procesy".

Oprócz korzystania z obiektów jądra aplikacja może korzystać z innych typów obiektów systemowych, takich jak menu, okna, kursory myszy, pędzle czy czcionki. Są to obiekty użytkownika lub interfejsu urządzenia graficznego (Graphical Device Interface – GDI), a nie obiekty jądra. Zaczynając programowanie dla Windows, można mieć kłopoty przy próbie odróżnienia obiektu użytkownika lub obiektu GDI od obiektu jądra. Na przykład, czy ikona jest obiektem użytkownika, czy obiektem jądra? Najlepszym sposobem określenia, czy dany obiekt jest obiektem jądra, jest zbadanie funkcji tworzącej ten obiekt. Prawie wszystkie funkcje tworzące obiekty jądra mają parametr pozwalający określić atrybut zabezpieczeń, tak jak funkcja CreateFileMapping pokazana wcześniej.

Żadna z funkcji tworzących obiekty użytkownika lub GDI nie ma parametru PSECURITY_ATTRIBUTES. Spójrzmy na przykład na funkcję CreateIcon:

```
HICON CreateIcon(
   HINSTANCE hinst,
   int nWidth,
   int nHeight,
   BYTE cPlanes,
   BYTE cBitsPixel,
   CONST BYTE *pbANDbits,
   CONST BYTE *pbXORbits);
```

Artykuł MSDN pod adresem http://msdn.microsoft.com/msdnmag/issues/03/01/GDILeaks udostępnia mnóstwo szczegółów dotyczących obiektów GDI i użytkownika.

Tabela uchwytów obiektów jądra dla danego procesu

Gdy proces jest inicjowany, system alokuje dla niego tabelę uchwytów. Ta tabela uchwytów jest używana tylko dla obiektów jądra, a nie dla obiektów użytkownika bądź obiektów GDI. Szczegóły dotyczące struktury i zarządzania tablicą uchwytów nie są udokumentowane. Normalnie unikałbym omawiania nieudokumentowanych części systemu operacyjnego. Jednak robię tu wyjątek, ponieważ sądzę, że kompetentny programista Windows musi rozumieć, jak zarządzana jest tabela uchwytów dla procesu. Ponieważ informacje te nie są udokumentowane, nie wszystkie szczegóły będą ściśle poprawne, a implementacja wewnętrzna na pewno różni się w różnych wersjach Windows.

Tabela 3-1 pokazuje, jak wygląda tabela uchwytów dla procesu. Jak widać jest to po prostu tablica struktur danych. Każda struktura zawiera wskaźnik od obiektu jądra, maskę dostępu i jakieś dodatkowe flagi.

Tabela 3-1 Struktura tabeli uchwytów dla procesu

Indeks	Wskaźnik do bloku pamięci dla obiektu jądra	Maska dostępu (wartość DWORD tworząca bity flagi)	Flagi
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
…	…	…	…

Tworzenie obiektu jądra

Gdy proces jest inicjowany, jego tabela uchwytów jest pusta. Gdy wątek procesu wywoła funkcję tworzącą obiekt jądra, na przykład funkcję `CreateFileMapping`, jądro zaalokuje blok pamięci dla obiektu i zainicjuje go. Następnie przeszuka tabelę uchwytów dla procesu w poszukiwaniu pustego miejsca. Ponieważ na początku tabela (taka jak tabela 3-1) jest pusta, jądro znajdzie strukturę pod indeksem 1 i zainicjuje ją. Wskaźnik zostanie ustawiony na adres pamięci wewnętrznej zawierającej strukturę danych obiektu jądra, maska dostępu zostanie ustawiona na pełny dostęp, a flagi zostaną ustawione (flagi omówię w części „Korzystanie z dziedziczenia uchwytów do obiektów").

Oto kilka funkcji tworzących obiekty jądra (w żadnym razie nie jest to pełna lista):

```
HANDLE CreateThread(
   PSECURITY_ATTRIBUTES psa,
   size_t dwStackSize,
   LPTHREAD_START_ROUTINE pfnStartAddress,
   PVOID pvParam,
   DWORD dwCreationFlags,
   PDWORD pdwThreadId);

HANDLE CreateFile(
   PCTSTR pszFileName,
   DWORD dwDesiredAccess,
   DWORD dwShareMode,
   PSECURITY_ATTRIBUTES psa,
   DWORD dwCreationDisposition,
   DWORD dwFlagsAndAttributes,
   HANDLE hTemplateFile);

HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);

HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTES psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);
```

Wszystkie funkcje tworzące obiekty jądra zwracają uchwyty względne dla danego procesu, które mogą być używane przez dowolne wątki działające w tym samym procesie. Wartość uchwytu powinna być podzielona przez 4 (lub przesunięta w prawo o dwa bity w celu zignorowania ostatnich dwóch bitów używanych wewnętrznie przez Windows) w celu otrzymania faktycznego indeksu w tablicy uchwytów dla procesu identyfikującego, gdzie przechowywana jest informacja o obiekcie jądra. Podczas debugowania aplikacji i badania faktycznych wartości uchwytów do obiektów jądra widoczne będą niewielkie wartości typu 4, 8, itp. Należy pamiętać, że znaczenie wartości uchwytu jest nieudokumentowane i może ulec zmianie.

Gdy wywoływana jest funkcja przyjmująca jako argument uchwyt do obiektu jądra, przekazuje się dla niej wartość zwróconą przez wywołaną wcześniej jedną z funkcji `Create*`. Wewnętrznie funkcja zaglądnie do tabeli uchwytów dla procesu w celu uzyskania adresu danego obiektu jądra i wykona operację na strukturze danych tego obiektu.

Jeśli przekazany zostanie nieprawidłowy uchwyt, działanie funkcji się nie powiedzie, a `GetLastError` zwróci wartość 6 (`ERROR_INVALID_HANDLE`). Ponieważ wartości uchwytów są faktycznie używane jako indeksy w tabeli uchwytów dla procesu, uchwyty są powiązane z konkretnymi procesami i nie mogą być używane w innych procesach. Próba taka jedynie odwołałaby się do innego obiektu jądra przechowywanego pod tym samym indeksem w tabeli uchwytów dla drugiego procesu bez jakiejkolwiek wiedzy, jakiego rodzaju byłby to obiekt.

Jeśli wywołana zostanie funkcja tworząca obiekt jądra, a jej wywołanie się nie powiedzie, zwracana jest zwykle wartość uchwytu 0 (`NULL`), dlatego pierwszą, poprawną wartością uchwytu jest 4. Żeby tak się stało, systemowi musiałoby brakować pamięci lub musiałby wystąpić jakiś poważny problem z zabezpieczeniami. Niestety kilka funkcji zwraca wartość uchwytu –1 (`INVALID_HANDLE_VALUE` zdefiniowane w WinBase.h) w razie niepowodzenia. Na przykład, jeśli `CreateFile` nie może otworzyć podanego pliku, zwraca `INVALID_HANDLE_VALUE` zamiast `NULL`. Trzeba bardzo uważać przy sprawdzaniu wartości zwracanej przez funkcję tworzącą obiekt jądra. W szczególności jedynie przy wywoływaniu funkcji `CreateFile` trzeba porównać zwracaną wartość z `INVALID_HANDLE_VALUE`. Poniższy kod jest nieprawidłowy:

```
HANDLE hMutex = CreateMutex(…);
if (hMutex == INVALID_HANDLE_VALUE) {
   // Ten kod nigdy nie będzie wykonywany, ponieważ
   // CreateMutex zwraca NULL w przypadku niepowodzenia.
}
```

Podobnie poniższy kod jest również nieprawidłowy:

```
HANDLE hFile = CreateFile(…);
if (hFile == NULL) {
   // Ten kod nigdy nie będzie wykonywany, ponieważ CreateFile
   // zwraca INVALID_HANDLE_VALUE (-1) w przypadku niepowodzenia.
}
```

Zamykanie obiektu jądra

Niezależnie od tego, w jaki sposób obiekt jądra został utworzony, można wskazać systemowi, że zakończono przetwarzanie tego obiektu poprzez wywołanie funkcji `CloseHandle`:

```
BOOL CloseHandle(HANDLE hobject);
```

Wewnętrznie funkcja ta najpierw sprawdza tabelę uchwytów dla procesu, który ją wywołał, aby upewnić się, że przekazana wartość uchwytu identyfikuje jakiś obiekt, do którego ten proces faktycznie ma dostęp. Jeśli uchwyt będzie poprawny, system pobierze adres struktury danych obiektu jądra i zmniejszy licznik użycia w tej strukturze. Jeśli licznik dojdzie do zera, obiekt jądra zostanie zniszczony i usunięty z pamięci.

Jeśli do funkcji `CloseHandle` przekazany zostanie nieprawidłowy uchwyt, może nastąpić jedna z dwóch rzeczy. Jeśli proces działa normalnie, `CloseHandle` zwróci `FALSE`, a `GetLastError` zwróci `ERROR_INVALID_HANDLE`. Jeśli natomiast proces jest debugowany, system wyrzuci wyjątek 0xC0000008 („Podano nieprawidłowy uchwyt"), co pozwoli na debugowanie błędu.

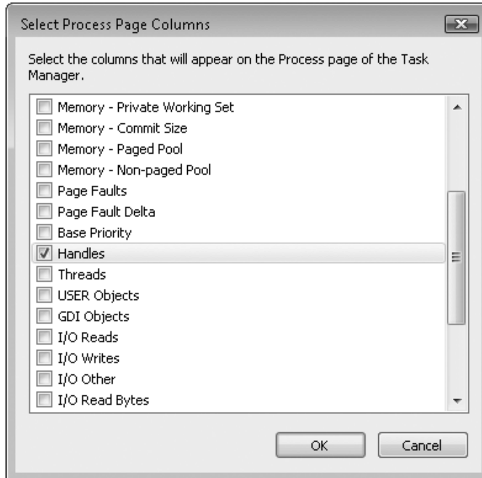
Tuż przed powrotem funkcji `CloseHandle` dane obiektu w tabeli uchwytów dla procesu są czyszczone – uchwyt staje się niepoprawny w danym procesie i nie należy próbować z niego korzystać. Czyszczenie następuje niezależnie od tego, czy sam obiekt jądra został zniszczony! Po wywołaniu `CloseHandle` nie ma już dostępu do obiektu jądra; jeśli jednak licznik obiektu nie został zmniejszony do zera, to obiekt nie został zniszczony. Tak ma być; oznacza to tylko, że jeden lub kilka innych procesów nadal korzysta z tego obiektu. Gdy inne procesy przestaną korzystać z obiektu (poprzez wywołanie `CloseHandle`), obiekt zostanie zniszczony.

Powiedzmy, że zapomnieliśmy wywołać `CloseHandle` – czy nastąpi wyciek pamięci? No cóż, i tak, i nie. Możliwy jest wyciek zasobów (takich jak obiekty jądra) z procesu podczas jego działania. Jednakże gdy proces kończy działanie, system operacyjny zapewnia, że wszystkie zasoby używane przez ten proces są zwalniane – jest to gwarantowane. W przypadku obiektów jądra system podejmuje następujące działania: gdy proces kończy działanie, system automatycznie przegląda tabelę uchwytów dla tego procesu. Jeśli tabela ma jakieś prawidłowe wpisy (obiekty, które nie zostały zamknięte przed końcem działania procesu), system zamknie te uchwyty obiektów za nas. Jeśli liczniki użycia dowolnych spośród tych obiektów dojdą do zera, jądro zniszczy te obiekty.

Uwaga Zwykle gdy tworzy się obiekt jądra, otrzymany uchwyt przechowuje się w zmiennej. Po wywołaniu funkcji `CloseHandle` z tą zmienną jako parametrem należy też ustawić tę zmienną na `NULL`. Gdyby przypadkiem zmienna ta została później wywołana z funkcją Win32, mogłyby wystąpić dwie nieoczekiwane sytuacje. Jako że miejsce w tabeli uchwytów wskazywane przez tę zmienną zostało wyczyszczone, system Windows otrzymałby nieprawidłowy parametr i wystąpiłby błąd. Możliwa jest też inna sytuacja, znacznie trudniejsza do wykrycia. Gdy tworzony jest nowy obiekt jądra, system Windows wyszukuje wolne miejsce w tabeli uchwytów. Jeśli więc w dalszym toku aplikacji tworzone byłyby nowe obiekty jądra, miejsce w tabeli uchwytów wskazywane przez tę zmienną z pewnością zostanie wypełnione jednym z nowych obiektów jądra. W ten sposób dalsze wywołanie mogłoby dotyczyć obiektu jądra złego typu lub co gorsza innego obiektu jądra tego samego typu co obiekt wcześniej zamknięty. Stan aplikacji ulegnie uszkodzeniu bez szans na odratowanie.

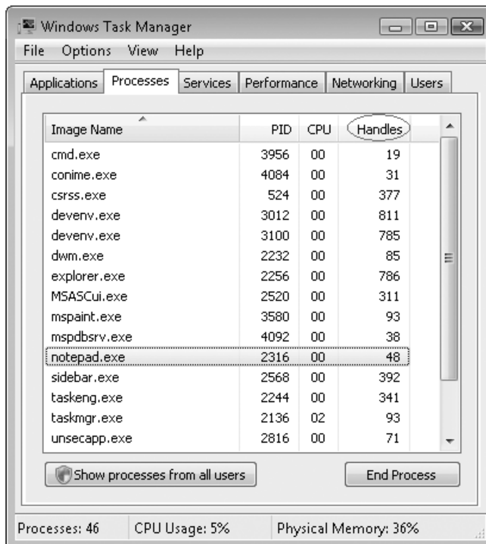
Tak więc aplikacja może gubić obiekty jądra w trakcie swojego działania, ale przy kończeniu działania procesu system gwarantuje, że wszystko zostanie prawidłowo wyczyszczone. À propos, jest to prawdą w przypadku wszystkich obiektów, zasobów takich jak obiekty GDI i bloków pamięci – gdy proces kończy działanie, system zapewnia, że proces nic po sobie nie zostawi. Prostym sposobem wykrycia, czy obiekty jądra wyciekają podczas działania aplikacji jest po prostu użycie Menedżera zadań Windows (Windows Task Manager).

Najpierw (jak pokazano na rysunku 3-1) trzeba wymusić pojawienie się kolumny Dojścia (Handles) na karcie Procesy (Processes), korzystając z okna dialogowego Wybieranie kolumn strony Proces (Select Process Page Columns) dostępnego poprzez polecenie menu Widok/Wybierz kolumny (View/Select Columns).



Rysunek 3-1 Wybieranie kolumny Dojścia (Handles) w oknie dialogowym Wybieranie kolumn strony Proces (Select Process Page Columns)

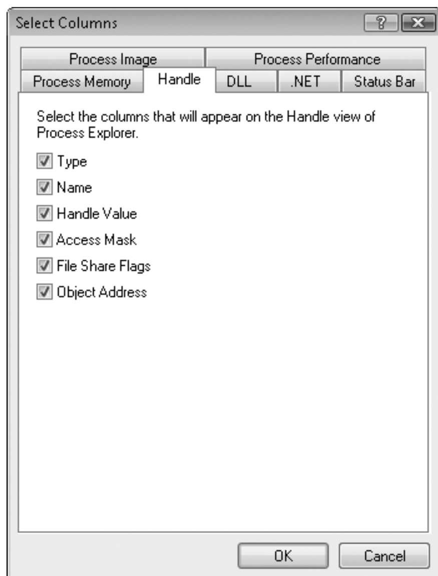
Następnie można monitorować liczbę obiektów jądra używanych przez dowolną aplikację, jak pokazano na rysunku 3-2.



Rysunek 3-2 Zliczanie uchwytów w Menedżerze zadań Windows (Windows Task Manager)

Jeśli liczba w kolumnie Dojścia (Handles) stale rośnie, następnym krokiem do zidentyfikowania, które obiekty jądra nie są zamykane, jest wykorzystanie darmowego narzędzia Process

Explorer z firmy Sysinternals (dostępnego pod adresem http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx). Najpierw należy kliknąć prawym przyciskiem myszy nagłówek w dolnym okienku Handles. Następnie w oknie dialogowym Select Columns, pokazanym na Rysunku 3-3, należy zaznaczyć wszystkie kolumny.



Rysunek 3-3 Wybieranie szczegółów dla widoku Handle w programie Process Explorer

Gdy to zostanie zrobione, należy zmienić opcję Update Speed na Paused w menu View. Następnie trzeba zaznaczyć żądany proces w górnym okienku i nacisnąć F5, aby uzyskać aktualną listę obiektów jądra. Po wykonaniu działań, które trzeba sprawdzić w aplikacji, należy ponownie nacisnąć F5 w programie Process Explorer. Każdy nowy obiekt jądra będzie wyświetlany na zielonym tle, co widać jako ciemniejszy odcień szarego na rysunku 3-4.

Należy zwrócić uwagę, że pierwsza kolumna podaje typ obiektu jądra, który nie został zamknięty. Aby zwiększyć szanse na znalezienie wycieków, druga kolumna podaje nazwę obiektu jądra. Jak zobaczymy w dalszej części tego rozdziału, łańcuch tekstowy nadawany jako nazwa obiektowi jądra pozwala współdzielić taki obiekt z innymi procesami. Oczywiście pomaga też łatwiej ustalić, które obiekty nie zostały zamknięte w oparciu o ich typ (pierwsza kolumna) i nazwę (druga kolumna). Jeśli wycieka duża liczba obiektów, prawdopodobnie nie mają one nazw, ponieważ można tworzyć tylko jedną instancję nazwanego obiektu.

Rysunek 3-4 Wykrywanie nowych obiektów jądra w programie Process Explorer

Współdzielenie obiektów jądra pomiędzy granicami procesów

Często wątki działające w różnych procesach muszą współdzielić obiekty jądra. Oto kilka powodów dlaczego:

- Obiekty mapowania plików pozwalają współdzielić bloki danych pomiędzy dwoma procesami działającymi na tej samej maszynie.
- Skrzynki pocztowe i nazwane potoki pozwalają przesyłać bloki danych pomiędzy procesami działającymi na różnych maszynach połączonych siecią.
- Muteksy, semafory i zdarzenia pozwalają wątkom w różnych procesach na synchronizowanie wykonywania, jak w przypadku aplikacji, która musi powiadomić inną aplikację o zakończeniu jakiegoś zadania.

Ponieważ uchwyty do obiektów jądra są powiązane z procesami, wykonywanie tych zadań jest trudne. Jednakże firma Microsoft miała kilka dobrych powodów, aby zaprojektować uchwyty w ten sposób. Najważniejszym z nich była wydajność. Gdyby uchwyty do obiektów jądra były wartościami na poziomie systemowym, jeden proces mógłby łatwo otrzymać uchwyt do obiektu używanego przez inny proces i spowodować zamieszanie w tym procesie. Innym powodem jest bezpieczeństwo. Obiekty jądra są chronione zabezpieczeniami, a proces musi uzyskać pozwolenie na przetwarzanie obiektu przed próbą manipulowania nim. Twórca obiektu może uniemożliwić nieautoryzowanemu użytkownikowi korzystanie z obiektu, po prostu zabraniając mu dostępu do niego.

W poniższej części rozdziału przyjrzymy się trzem różnym mechanizmom pozwalającym procesom współdzielić obiekty jądra: wykorzystaniu dziedziczenia uchwytów do obiektów, nadawaniu nazw obiektom oraz duplikowaniu uchwytów do obiektów.

Korzystanie z dziedziczenia uchwytów do obiektów

Dziedziczenie uchwytów do obiektów może być używane tylko wtedy, gdy jeden z procesów jest podrzędny względem drugiego. W tym scenariuszu jeden lub kilka uchwytów do obiektów jądra jest dostępnych w procesie nadrzędnym, a proces ten, uruchamiając proces podrzędny, daje mu dostęp do swoich obiektów jądra. Aby ten typ dziedziczenia zadziałał, proces nadrzędny musi wykonać kilka kroków.

Najpierw gdy proces nadrzędny tworzy obiekt jądra, musi wskazać systemowi, że uchwyt tego obiektu może być dziedziczony. Czasami niektórzy używają w tym miejscu terminu dziedziczenie obiektów. Jednakże Windows w istocie obsługuje dziedziczenie uchwytów do obiektów. Innymi słowy, dziedziczone są uchwyty, a nie same obiekty.

Aby utworzyć uchwyt, który będzie mógł być dziedziczony, proces nadrzędny musi zaalokować i odpowiednio zainicjować strukturę `SECURITY_ATTRIBUTES` oraz przekazać adres tej struktury do określonej funkcji `Create`. Następujący kod tworzy obiekt muteks i zwraca do niego uchwyt, który może być dziedziczony:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;   // Spraw, aby zwracany uchwyt mógł być dziedziczony.

HANDLE hMutex = CreateMutex(&sa, FALSE, NULL);
```

Ten kod inicjuje strukturę `SECURITY_ATTRIBUTES`, wskazując, że obiekt powinien zostać utworzony z użyciem domyślnych zabezpieczeń i że możliwe powinno być dziedziczenie zwracanego uchwytu.

Teraz dochodzimy do flag przechowywanych w elementach tabeli uchwytów dla procesu. Każdy element tabeli uchwytów ma bit flagi wskazujący, czy dany uchwyt może być dziedziczony. Jeśli w parametrze `PSECURITY_ATTRIBUTES` przekazana była wartość `NULL` przy tworzeniu obiektu jądra, zwracany uchwyt nie może być dziedziczony, a wartość tego bitu jest równa zero. Ustawienie elementu `bInheritHandle` na `TRUE` powoduje, że ten bit flagi będzie ustawiony na 1.

Wyobraźmy sobie tablicę uchwytów dla procesu wyglądającą tak jak pokazano w tabeli 3-2.

Tabela 3-2 Tabela uchwytów dla procesu zawierająca dwa prawidłowe wpisy

Indeks	Wskaźnik do bloku pamięci dla obiektu jądra	Maska dostępu (wartość DWORD tworząca bity flagi)	Flagi
1	0xF0000000	0x????????	0x00000000
2	0x00000000	(N/A)	(N/A)
3	0xF0000010	0x????????	0x00000001

Tabela 3-2 wskazuje, że ten proces ma dostęp do dwóch obiektów jądra (uchwyty 1 i 3). Uchwyt 1 nie może być dziedziczony, a uchwyt 3 może.

Następnym krokiem do wykonania przy korzystaniu z dziedziczenia uchwytów do obiektów jest uruchomienie procesu potomnego przez proces nadrzędny. Dokonuje się tego przy użyciu funkcji `CreateProcess`:

```
BOOL CreateProcess(
   PCTSTR pszApplicationName,
   PTSTR pszCommandLine,
   PSECURITY_ATTRIBUTES psaProcess,
   PSECURITY_ATTRIBUTES psaThread,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
   PVOID pvEnvironment,
   PCTSTR pszCurrentDirectory,
   LPSTARTUPINFO pStartupInfo,
   PPROCESS_INFORMATION pProcessInformation);
```

Funkcję tę zbadamy szczegółowo w następnym rozdziale, ale na razie chciałbym się skupić na parametrze `bInheritHandles`. Zwykle przy tworzeniu nowego procesu przekazuje się wartość `FALSE` dla tego parametru. Wartość ta informuje system, że proces potomny nie ma dziedziczyć uchwytów (nawet tych, które mogą być dziedziczone) znajdujących się w tabeli uchwytów procesu nadrzędnego.

Jeśli jednak przekaże się wartość `TRUE` dla tego parametru, proces potomny odziedziczy wartości uchwytów, które mogą być dziedziczone z procesu nadrzędnego. System operacyjny utworzy nowy proces potomny, ale nie pozwoli mu rozpocząć wykonywania kodu od razu. Oczywiście system utworzy nową, pustą tabelę uchwytów dla procesu potomnego – tak samo jak w przypadku każdego nowego procesu. Ale ponieważ przekazana została wartość `TRUE` dla parametru `bInheritHandles` funkcji `CreateProcess`, system zrobi jeszcze jedną rzecz: przejrzymy tablicę uchwytów procesu nadrzędnego i dla każdego elementu zawierającego poprawny uchwyt, który może być dziedziczony, przekopiuje go dokładnie do tabeli uchwytów procesu potomnego. Element jest kopiowany do dokładnie tej samej pozycji w tabeli uchwytów procesu potomnego co w procesie nadrzędnym. Jest to ważne, ponieważ oznacza, że wartość uchwytu identyfikującego obiekt jądra jest identyczna zarówno w procesie nadrzędnym jak i potomnym.

Oprócz kopiowania elementu tabeli uchwytów system zwiększa licznik użycia obiektu jądra, ponieważ jest on teraz używany przez dwa procesy. Aby obiekt jądra został zniszczony, zarówno proces nadrzędny jak i potomny muszą albo wywołać `CloseHandle` dla tego obiektu, albo zakończyć działanie. Proces potomny nie musi kończyć działania jako pierwszy – to samo dotyczy procesu nadrzędnego. Właściwie proces nadrzędny może zamknąć swój uchwyt do obiektu od razu po powrocie z funkcji `CreateProcess` bez wpływania na zdolność procesu potomnego do manipulowania danym obiektem.

Tabela 3-3 pokazuje tabelę uchwytów procesu potomnego tuż przed rozpoczęciem wykonywania tego procesu. Widać, że elementy 1 i 2 nie są zainicjowane i dlatego nie są prawidłowymi uchwytami do użycia przez proces potomny. Jednakże indeks 3 identyfikuje obiekt jądra. W rzeczywistości identyfikuje obiekt jądra pod adresem 0xF0000010 – ten sam obiekt co w tabeli uchwytów dla procesu nadrzędnego.

Tabela 3-3 Uchwyty procesu potomnego po odziedziczeniu uchwytu z procesu nadrzędnego

Indeks	Wskaźnik do bloku pamięci dla obiektu jądra	Maska dostępu (wartość DWORD tworząca bity flagi)	Flagi
1	0x00000000	(brak)	(brak)
2	0x00000000	(brak)	(brak)
3	0xF0000010	0x????????	0x00000001

Jak zobaczymy w rozdziale 13 „Architektura pamięci Windows", zawartość obiektów jądra jest przechowywana w przestrzeni adresowej jądra współdzielonej przez wszystkie procesy działające w systemie. Dla systemów 32-bitowych jest to obszar pomiędzy następującymi adresami pamięci: 0x80000000 a 0xFFFFFFFF. Dla systemów 64-bitowych jest to pamięć pomiędzy adresami: 0x00000400'00000000 a 0xFFFFFFF'FFFFFFFF. Maska dostępu jest identyczna z maską w procesie nadrzędnym, flagi również są takie same. Oznacza to, że gdyby proces potomny uruchamiał własny proces potomny z tym samym parametrem `bInheritHandles` funkcji `CreateProcess` ustawionym na `TRUE`, ten nowy proces potomny również odziedziczyłby uchwyt obiektu jądra mający tę samą wartość uchwytu, tę samą maskę dostępu i te sami flagi, a licznik użycia obiektu znowu zostałby zwiększony o 1.

Należy mieć świadomość, że dziedziczenie uchwytu do obiektu może być zastosowane tylko w momencie tworzenia procesu podrzędnego. Jeśli proces nadrzędny utworzy nowe obiekty jądra z uchwytami, które mogą być dziedziczone, działający już proces podrzędny nie odziedziczy tych nowych uchwytów.

Dziedziczenie uchwytów do obiektów ma jedną bardzo dziwną cechę: proces potomny nie ma pojęcia o tym, że odziedziczył jakieś uchwyty. Dziedziczenie uchwytów do obiektów jądra jest przydatne tylko wtedy, gdy proces potomny będzie świadom faktu, że ma się spodziewać dostępu do danego obiektu jądra, gdy będzie uruchomiony przez inny proces. Zwykle proces nadrzędny i potomny są pisane przez tę samą firmę; jednakże ktoś inny będzie mógł pisać aplikacje podrzędne, jeśli udokumentowane zostanie, czego proces potomny może się spodziewać.

Zdecydowanie najczęściej używanym sposobem określania przez proces potomny oczekiwanej wartości uchwytu do obiektu jądra jest przekazywanie jej do procesu potomnego poprzez argument wiersza polecenia. Kod inicjujący proces potomny przetwarza wiersz polecenia (zwykle przy użyciu `_stscanf_s`), a następnie wyciąga z niego wartość uchwytu. Mając wartość uchwytu proces ma taki sam dostęp do obiektu, jak jego proces nadrzędny. Jedynym powodem, dla którego dziedziczenie uchwytów działa, jest to, że wartość uchwytu do współdzielonego obiektu jądra jest identyczna w obu procesach. Dlatego właśnie proces nadrzędny jest w stanie przekazać wartość uchwytu poprzez argument wiersza polecenia.

Oczywiście można skorzystać z innych form komunikacji między procesami w celu przekazania wartości uchwytu dziedziczonego obiektu jądra z procesu nadrzędnego do procesu potomnego. Jedną z technik jest oczekiwanie przez proces nadrzędny na zakończenie inicjalizacji przez proces potomny (przy użyciu funkcji `WaitForInputIdle` omawianej w rozdziale 9 „Synchronizacja wątków przy pomocy obiektów jądra"); następnie proces nadrzędny może przesłać komunikat do okna utworzonego przez wątek w procesie potomnym.

Inną techniką jest dodanie przez proces nadrzędny zmiennej środowiskowej do danego bloku środowiska. Nazwa zmiennej musi być znana procesowi podrzędnemu, a wartością zmiennej powinna być wartość uchwytu do obiektu jądra, który ma być dziedziczony. Gdy proces nadrzędny będzie uruchamiać proces potomny, odziedziczy on zmienne środowiskowe procesu nadrzędnego i może łatwo wywołać `GetEnvironmentVariable` w celu uzyskania wartości uchwytu do odziedziczonego obiektu. To podejście świetnie się nadaje w sytuacjach, gdy proces potomny będzie uruchamiać inne procesy potomne, ponieważ zmienne środowiskowe będą mogły być ponownie odziedziczone. Specjalny przypadek dziedziczenia konsoli procesu nadrzędnego przez proces potomny opisano dokładnie w bazie wiedzy Microsoft pod adresem http://support.microsoft.com/kb/190351.

Zmienianie flag uchwytu

Czasami można się spotkać z sytuacją, w której proces nadrzędny tworzy obiekt jądra z uchwytem, który może być dziedziczony, a następnie uruchamia dwa procesy potomne. Proces nadrzędny chce, aby tylko jeden proces potomny dziedziczył uchwyt do obiektu jądra. Innymi słowy, czasami może być potrzebna kontrola nad tym, który proces potomny będzie dziedziczył uchwyty do obiektów jądra. Aby zmodyfikować flagę dziedziczenia uchwytu do obiektu jądra, można wywołać funkcję `SetHandleInformation`:

```
BOOL SetHandleInformation(
   HANDLE hObject,
   DWORD dwMask,
   DWORD dwFlags);
```

Jak widać, funkcja ta przyjmuje trzy parametry. Pierwszy, `hObject`, identyfikuje prawidłowy uchwyt. Drugi parametr, `dwMask`, informuje funkcję, która flaga (lub flagi) ma być zmieniona. Obecnie dwie flagi są związane z każdym uchwytem:

```
#define HANDLE_FLAG_INHERIT
   0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE
   0x00000002
```

Można połączyć obie flagi bitowym operatorem LUB, jeśli zmieniane mają być obie flagi równocześnie. Trzeci parametr funkcji `SetHandleInformation`, `dwFlags`, wskazuje wartość, na jaką flagi mają być ustawione. Na przykład, aby włączyć flagę dziedziczenia dla uchwytu do obiektu jądra, należy:

```
SetHandleInformation(hObj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

Aby wyłączyć tę flagę, należy wywołać:

```
SetHandleInformation(hObj, HANDLE_FLAG_INHERIT, 0);
```

Flaga `HANDLE_FLAG_PROTECT_FROM_CLOSE` informuje system, że nie powinien pozwalać na zamknięcie danego uchwytu:

```
SetHandleInformation(hObj, HANDLE_FLAG_PROTECT_FROM_CLOSE,
   HANDLE_FLAG_PROTECT_FROM_CLOSE);
CloseHandle(hObj);   // Wzbudzany jest wyjątek
```

Gdy aplikacja uruchomiona jest pod kontrolą debugera, jeśli wątek będzie próbował zamknąć chroniony uchwyt, `CloseHandle` wzbudzi wyjątek. Poza kontrolą debugera, `CloseHandle` po prostu zwróci `FALSE`. Ochrony uchwytu przed zamknięciem używa się rzadko. Jednakże flaga ta może być przydatna, jeśli proces uruchamia proces potomny, który z kolei uruchamia następny proces potomny. Proces nadrzędny może oczekiwać, że ten ostatni proces odziedziczy uchwyt obiektu przekazany bezpośredniemu potomkowi. Istnieje jednak możliwość, że bezpośredni potomek zamknie uchwyt przed wywołaniem własnego potomka. Gdyby tak się stało, proces nadrzędny może nie być w stanie komunikować się z potomkiem niższego poziomu, ponieważ nie odziedziczy on obiektu jądra. Oznaczając uchwyt jako „chroniony przed zamknięciem" dajemy potomkowi niższego poziomu większe szanse na odziedziczenie uchwytu do prawidłowego obiektu.

To podejście ma jednak pewną wadę. Bezpośredni proces potomny mógłby wywołać następujący kod, aby wyłączyć flagę `HANDLE_FLAG_PROTECT_FROM_CLOSE`, a następnie zamknąć uchwyt:

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE, 0);
CloseHandle(hObj);
```

Proces nadrzędny zakłada, że proces potomny nie wykona takiego kodu. Oczywiście proces nadrzędny zakłada też, że proces potomny uruchomi swojego potomka, więc założenie to nie jest obarczone znacznym ryzykiem.

Dla uzupełnienia wspomnę też o funkcji `GetHandleInformation`:

```
BOOL GetHandleInformation(
   HANDLE hObject,
   PDWORD pdwFlags);
```

Funkcja ta zwraca bieżące ustawienia flag dla określonego uchwytu, zapisując je w zmiennej DWORD wskazywanej przez parametr `pdwFlags`. Aby sprawdzić, czy uchwyt może być dziedziczony, należy użyć następującego kodu:

```
DWORD dwFlags;
GetHandleInformation(hObj, &dwFlags);
BOOL fHandleIsInheritable = (0 != (dwFlags & HANDLE_FLAG_INHERIT));
```

Nadawanie nazw obiektom

Drugą dostępną metodą współdzielenia obiektów jądra ponad granicami procesów jest nadawanie nazw obiektom. Wiele (chociaż nie wszystkie) obiektów jądra może mieć nazwy. Na przykład wszystkie poniższe funkcje tworzą nazwane obiekty jądra:

```
HANDLE CreateMutex(
   PSECURITY_ATTRIBUTES psa,
   BOOL bInitialOwner,
   PCTSTR pszName);

HANDLE CreateEvent(
   PSECURITY_ATTRIBUTES psa,
   BOOL bManualReset,
   BOOL bInitialState,
   PCTSTR pszName);

HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTES psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);

HANDLE CreateWaitableTimer(
   PSECURITY_ATTRIBUTES psa,
   BOOL bManualReset,
   PCTSTR pszName);

HANDLE CreateFileMapping(
   HANDLE hFile,
   PSECURITY_ATTRIBUTES psa,
   DWORD flProtect,
```

```
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   PCTSTR pszName);

HANDLE CreateJobObject(
   PSECURITY_ATTRIBUTES psa,
   PCTSTR pszName);
```

Wszystkie one mają taki sam ostatni parametr, `pszName`. Jeśli wartość `NULL` zostanie przekazana dla tego parametru, oznacza, że system ma utworzyć obiekt jądra bez nazwy (anonimowy). Jeśli utworzy się obiekt bez nazwy, można współdzielić go między procesami albo korzystając z dziedziczenia (jak omówiono w poprzedniej części tego rozdziału), albo używając funkcji `DuplicateHandle` (omówionej w następnej części). Aby współdzielić obiekt poprzez nazwę, trzeba mu ją nadać.

Jeśli dla parametru `pszName` przekazana zostanie wartość inna niż `NULL`, powinien być to adres nazwy zapisanej w łańcuchu zakończonym zerem. Nazwa ta może mieć maksymalną długość `MAX_PATH` znaków (zdefiniowane jako 260). Niestety Microsoft nie oferuje żadnej pomocy w przypisywaniu nazw obiektom jądra. Na przykład, jeśli spróbuje się utworzyć obiekt o nazwie „JeffObj", nie ma gwarancji, że obiekt o takiej nazwie już nie istnieje. Co gorsza wszystkie takie obiekty należą do jednej przestrzeni nazw, nawet jeśli nie są tego samego typu. Z tego powodu następujące wywołanie `CreateSemaphore` zawsze zwróci `NULL` – ponieważ istnieje już muteks o takiej samej nazwie:

```
HANDLE hMutex = CreateMutex(NULL, FALSE, TEXT("JeffObj"));
HANDLE hSem = CreateSemaphore(NULL, 1, 1, TEXT("JeffObj"));
DWORD dwErrorCode = GetLastError();
```

Gdyby zbadać wartość `dwErrorCode` po wykonaniu powyższego kodu, można zobaczyć kod 6 (`ERROR_INVALID_HANDLE`). Ten kod błędu nie wyjaśnia wiele, ale co można zrobić?

Gdy już wiemy, jak nadać obiektowi nazwę, zobaczmy jak współdzielić obiekt w ten sposób. Powiedzmy, że proces A wywołuje następującą funkcję:

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

To wywołanie funkcji tworzy nowy obiekt jądra (muteks) i przypisuje mu nazwę „JeffMutex". Należy zwrócić uwagę, że uchwyt `hMutexProcessA` nie jest uchwytem, który może być dziedziczony – i w tej sytuacji nie musi być.

Nieco później jakiś proces uruchamia proces B. Proces B nie musi być potomkiem procesu A; może być wywołany z poziomu Eksploratora Windows lub jakiejkolwiek innej aplikacji. Fakt, że proces B nie musi być potomkiem procesu A, daje przewagę korzystaniu z nazwanych obiektów nad dziedziczeniem. Gdy proces B rozpocznie swoje działanie, wykonuje następujący kod:

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

Gdy proces B wywoła `CreateMutex`, system najpierw sprawdza, czy istnieje już obiekt jądra o nazwie „JeffMutex". Ponieważ obiekt o tej nazwie faktycznie istnieje, jądro sprawdza typ obiektu. Ponieważ próbujemy utworzyć muteks, a obiekt o nazwie „JeffMutex" też jest muteksem, system sprawdza, czy proces ma pełen dostęp do obiektu. Jeśli tak, system znajduje puste miejsce w tabeli uchwytów dla procesu B i wypełnia je wskazaniem na istniejący obiekt jądra. Jeśli typy obiektów nie są zgodne albo jeśli dostęp jest zabroniony, wywołanie `CreateMutex` się nie powiedzie (zwróci `NULL`).

Uwaga Funkcje tworzące obiekty jądra (takie jak `CreateSemaphore`) zawsze zwracają uchwyty z pełnymi prawami dostępu. Chcąc ograniczyć prawa dostępu dla uchwytu, można skorzystać z rozszerzonych wersji funkcji tworzących obiekty jądra (z przyrostkiem `Ex`), które przyjmują dodatkowy parametr `dwDesiredAccess` typu DWORD. Na przykład można zezwolić lub zabronić wywoływania funkcji `ReleaseSemaphore` dla danego uchwytu do semafora poprzez użycie lub nie opcji `SEMAPHORE_MODIFY_STATE` w wywołaniu `CreateSemaphoreEx`. Szczegóły dotyczące określonych uprawnień odpowiadających poszczególnym rodzajom obiektów jądra można znaleźć w dokumentacji Windows SDK pod adresem *http://msdn2.microsoft.com/en-us/library/ms686670.aspx*.

Gdy powiedzie się wywołanie `CreateMutex` z poziomu procesu B, muteks nie jest w rzeczywistości tworzony. Proces B otrzymuje po prostu wartość uchwytu identyfikującego istniejący już w jądrze obiekt muteks. Ponieważ nowa pozycja w tabeli uchwytów dla procesu B odwołuje się do tego obiektu, licznik użycia obiektu muteks jest zwiększany o 1. Obiekt nie zostanie zniszczony, dopóki procesy A i B nie zamkną swoich uchwytów do obiektu. Warto zwrócić uwagę, że wartości uchwytów w obu procesach najprawdopodobniej będą różne. To jest w porządku. Proces A będzie korzystać ze swojej wartości uchwytu, a proces B ze swojej przy manipulowaniu tym samym obiektem jądra.

Istnieje alternatywna metoda współdzielenia obiektów przez nazwę. Zamiast wywoływać funkcję `Create*` proces może wywołać jedną z funkcji `Open*` pokazanych poniżej:

```
HANDLE OpenMutex(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenEvent(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenSemaphore(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenWaitableTimer(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenFileMapping(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);

HANDLE OpenJobObject(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

Uwaga Kiedy obiekty jądra są współdzielone poprzez nazwę, trzeba wiedzieć o pewnym niezwykle ważnym szczególe. Gdy proces B wywołuje `CreateMutex`, przekazuje do tej funkcji informację o atrybucie zabezpieczeń oraz jeszcze jeden parametr. Parametry te są ignorowane, jeśli obiekt o podanej nazwie już istnieje! Aplikacja może stwierdzić, czy faktycznie utworzyła nowy obiekt jądra, czy raczej otwarła istniejący obiekt, wywołując `GetLastError` zaraz po wywołaniu funkcji `Create*`:

```
HANDLE hMutex = CreateMutex(&sa, FALSE, TEXT(„JeffObj”));
if (GetLastError() == ERROR_ALREADY_EXISTS) {
   // Otwarto uchwyt do istniejącego obiektu.
   // sa.lpSecurityDescriptor oraz drugi parametr
   // (FALSE) są ignorowane.
} else {
   // Utworzono zupełnie nowy obiekt.
   // sa.lpSecurityDescriptor oraz drugi parametr
   // (FALSE) są używane do skonstruowania obiektu.
}
```

Warto zwrócić uwagę, że wszystkie te funkcje mają taki sam prototyp. Ostatni parametr, `pszName`, określa nazwę obiektu jądra. Nie można przekazać wartości `NULL` dla tego parametru, trzeba przekazać adres łańcucha znaków zakończonego zerem. Funkcje te przeszukują przestrzeń nazw obiektów jądra, próbując znaleźć pasującą nazwę. Jeśli nie istnieje żaden obiekt jądra o podanej nazwie, funkcje te zwracają `NULL`, a `GetLastError` zwraca 2 (`ERROR_FILE_NOT_FOUND`). Jeśli jednak istnieje obiekt jądra o podanej nazwie, ale ma inny typ, funkcje te zwracają `NULL`, a `GetLastError` zwraca 6 (`ERROR_INVALID_HANDLE`). Jeśli jest to obiekt tego samego typu, system sprawdza, czy żądany dostęp jest dozwolony (poprzez parametr `dwDesiredAccess`). Jeśli tak, tabela uchwytów dla procesu wywołującego jest uaktualniana, a licznik użycia obiektu jest zwiększany o 1. Zwrócony uchwyt będzie mógł być dziedziczony, jeśli przekazano wartość `TRUE` dla parametru `bInheritHandle`.

Główna różnica pomiędzy wywołaniem funkcji `Create*` a wywołaniem funkcji `Open*` jest taka, że jeśli obiekt jeszcze nie istnieje, funkcja `Create*` go utworzy, natomiast funkcja `Open*` po prostu się nie powiedzie.

Jak wspominałem wcześniej, Microsoft nie oferuje prawdziwej pomocy dotyczącej tworzenia unikalnych nazw obiektów. Innymi słowy problemem może być, gdyby użytkownik uruchomił dwa programy, a każdy z nich próbował utworzyć obiekt o nazwie „MyObject”. Aby zapewnić unikalność nazw, zalecam używanie identyfikatorów GUID i tekstowych reprezentacji GUID dla nazw obiektów. Inny sposób zapewnienia unikalności nazw można będzie zobaczyć w części zatytułowanej „Prywatne przestrzenie nazw”.

Nazwane obiekty są często używane w celu uniemożliwienia jednoczesnego działania wielu instancji tej samej aplikacji. Aby to osiągnąć, po prostu należy wywoływać jakąś funkcję `Create*` w funkcji `_tmain` lub `_tWinMain` w celu utworzenia nazwanego obiektu (nie ma znaczenia, jaki to będzie typ obiektu). Po powrocie z funkcji `Create*` należy wywołać `GetLastError`. Jeśli `GetLastError` zwróci `ERROR_ALREADY_EXISTS`, to znaczy, że działa już inna instancja aplikacji i nowa instancja może zakończyć swoje działanie. Oto przykładowy kod ilustrujący to zagadnienie:

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine,
   int nCmdShow) {
   HANDLE h = CreateMutex(NULL, FALSE,
      TEXT("{FA531CC1-0497-11d3-A180-00105A276C3E}"));
```

```
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      // Uruchomiona jest już instancja tej aplikacji.
      // Zamknij obiekt i od razu zakończ działanie aplikacji.
      CloseHandle(h);
      return(0);
   }

   // To jest pierwsza uruchomiona instancja tej aplikacji.
   ...
   // Przed zakończeniem działania zamknij obiekt.
   CloseHandle(h);
   return(0);
}
```

Przestrzenie nazw usług terminalowych

Warto zwrócić uwagę, że usługi terminalowe zmieniają nieco powyższy scenariusz. Maszyna z uruchomionymi usługami terminalowymi ma wiele przestrzeni nazw dla obiektów jądra. Istnieje jedna globalna przestrzeń nazw używana przez obiekty jądra, które mają być dostępne dla wszystkich sesji klienckich. Ta przestrzeń nazw jest głównie używana przez usługi. Oprócz tego każda sesja kliencka ma własną przestrzeń nazw. To rozwiązanie pozwala dwóm lub więcej sesjom, w których uruchomiono tę samą aplikację na nie wchodzenie sobie w drogę – jedna sesja nie będzie miała dostępu do obiektów drugiej, nawet jeśli obiekty będą miały takie same nazwy. Scenariusze te nie są tylko związane z maszynami serwerowymi, ponieważ funkcje zdalnego pulpitu i szybkiego przełączania użytkowników również są zaimplementowane przy wykorzystaniu sesji usług terminalowych.

Uwaga Zanim jakikolwiek użytkownik się zaloguje, usługi są uruchamiane w pierwszej sesji. W Windows Vista, w odróżnieniu od poprzednich wersji Windows, zaraz jak użytkownik się zaloguje, aplikacje są uruchamiane w nowej sesji – różnej od sesji 0 poświęconej usługom. W ten sposób zasadnicze składniki systemu, które zwykle uruchamiane są z wysokimi uprawnieniami, są bardziej odizolowane od jakiegokolwiek szkodliwego oprogramowania uruchomionego przez któregoś użytkownika.

Dla programistów usług konieczność uruchamiania ich w sesji innej niż aplikacje klienckie wpływa na konwencje nazw dla współdzielonych obiektów jądra. Obecnie obowiązkowe jest tworzenie obiektów, które mają być współdzielone z aplikacjami użytkowników, w globalnej przestrzeni nazw. Ta sama kwestia występuje, jeżeli trzeba napisać usługę mającą się komunikować z aplikacjami, które mogą działać, gdy kilku różnych użytkowników zalogowało się do różnych sesji poprzez szybkie przełączanie użytkowników – usługa nie może zakładać, że działa w tej samej sesji co aplikacja użytkownika. Więcej szczegółów na temat izolacji sesji 0 i jaki to ma wpływ na programistów usług można przeczytać w artykule "Impact of Session 0 Isolation on Services and Drivers in Windows Vista" znajdującym się pod adresem *http://www.microsoft.com/whdc/system/vista/services.mspx*.

Jeśli trzeba wiedzieć, w której sesji usług terminalowych działa dany proces, wystarczy użyć funkcji `ProcessIdToSessionId` (eksportowanej przez kernel32.dll i zadeklarowanej w WinBase.h), co pokazano w poniższym przykładzie:

```
DWORD processID = GetCurrentProcessId();
DWORD sessionID;
```

```
if (ProcessIdToSessionId(processID, &sessionID)) {
   tprintf(
      TEXT("Process '%u' runs in Terminal Services session '%u'"),
      processID, sessionID);
} else {
   // ProcessIdToSessionId może się nie powieść, jeżeli nie ma się wystarczających
   // praw dostępu do procesu, dla którego przekazywany jest identyfikator jako
parametr.
   // W tym przypadku nie ma to znaczenia, ponieważ używamy własnego identyfikatora
procesu.
   tprintf(
      TEXT("Unable to get Terminal Services session ID for process '%u'"),
      processID);
}
```

Nazwane obiekty jądra dla usługi zawsze trafiają do globalnej przestrzeni nazw. Domyślnie w usługach terminalowych nazwany obiekt jądra dla aplikacji trafia do przestrzeni nazw dla danej sesji. Jednakże możliwe jest wymuszenie trafienia nazwanego obiektu do globalnej przestrzeni nazw poprzez poprzedzenie nazwy przedrostkiem „Global\" jak w poniższym przykładzie:

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, TEXT("Global\\MyName"));
```

Można też wprost określić, że dany obiekt jądra ma trafić do przestrzeni nazw bieżącej sesji poprzedzając jego nazwę przedrostkiem „Local\" jak w poniższym przykładzie:

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, TEXT("Local\\MyName"));
```

Microsoft traktuje Global i Local jako zarezerwowane słowa, których nie powinno się używać w nazwach obiektów poza wymuszaniem umieszczenia ich w odpowiednich przestrzeniach nazw. Microsoft traktuje też słowo Session jako słowo kluczowe. Jednakże nie jest możliwe utworzenie obiektu z nazwą w innej sesji przy użyciu przedrostka Session – wówczas wywołanie funkcji się nie powiedzie a `GetLastError` zwróci `ERROR_ACCESS_DENIED`.

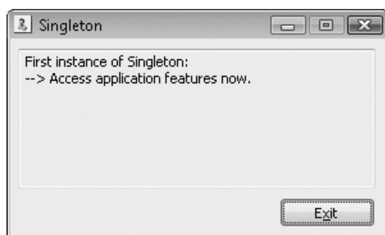
Uwaga We wszystkich zarezerwowanych słowach kluczowych wielkość liter ma znaczenie.

Prywatne przestrzenie nazw

Gdy tworzy się obiekt jądra, można zabezpieczyć dostęp do niego, przekazując wskaźnik do struktury `SECURITY_ATTRIBUTES`. Jednakże przed pojawieniem się Windows Vista nie była możliwa ochrona nazwy współdzielonego obiektu przed przechwyceniem. Dowolny proces, nawet mający najmniejsze przywileje, jest w stanie utworzyć obiekt o danej nazwie. Jeśli wziąć poprzedni przykład, w którym aplikacja używała nazwanego obiektu muteks do wykrycia, czy inna jej instancja nie jest już uruchomiona, łatwo można by napisać inną aplikacją tworzącą obiekt jądra o takiej samej nazwie. Jeśli zostanie ona uruchomiona wcześniej, nasza aplikacja przykładowa po uruchomieniu będzie od razu kończyć działanie, sądząc, że działa już jakaś inna jej instancja. Jest to podstawowy mechanizm stojący za kilkoma atakami typu Denial of Service (DoS). Obiekty jądra nie mające nazw nie są podatne na takie ataki i często aplikacje używają obiektów bez nazw, chociaż nie mogą być one współdzielone pomiędzy procesami.

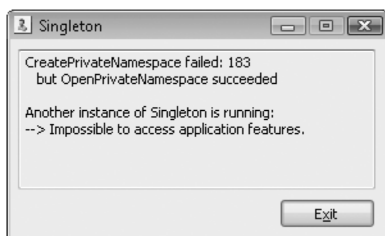
Chcąc mieć pewność, że nazwy obiektów jądra tworzone przez własne aplikacje nie będą wchodziły w konflikty z nazwami w innych aplikacjach ani nie będą podatne na ataki, można zdefiniować własny przedrostek i używać go jako prywatnej przestrzeni nazw, tak jak to ma miejsce w przypadku przedrostków Global i Local. Proces serwera odpowiedzialny za tworzenie obiektów jądra definiuje deskryptor chroniący samą przestrzeń nazw.

Aplikacja Singleton (jej kod źródłowy Singleton.cpp zostanie podany w dalszej części tego rozdziału) pokazuje, jak korzystać z prywatnych przestrzeni nazw do zaimplementowania tego samego wzorca singletonu przedstawionego wcześniej, ale w bezpieczniejszy sposób. Po uruchomieniu programu pojawia się okno pokazane na rysunku 3-5.



Rysunek 3-5 Uruchomiona pierwsza instancja programu Singleton

Jeśli ten sam program zostanie uruchomiony, podczas gdy pierwszy jego egzemplarz nadal działa, okno pokazane na rysunku 3-6 wyjaśni, że wykryto poprzednią instancję tego samego programu.



Rysunek 3-6 Druga instancja programu Singleton uruchomiona podczas gdy pierwsza nadal działa

Funkcja `CheckInstances` w poniższym kodzie źródłowym pokazuje, jak utworzyć granicę, skojarzyć z nią identyfikator zabezpieczeń (SID) odpowiadający grupie lokalnych administratorów (Local Administrators) a następnie stworzyć lub otworzyć prywatną przestrzeń nazw, której nazwa zostanie użyta jako przedrostek przez obiekt jądra. Deskryptor granicy otrzymuje nazwę, ale co ważniejsze otrzymuje też SID uprzywilejowanej grupy użytkowników. W ten sposób Windows zapewnia, że tylko aplikacje działające w kontekście użytkownika będącego częścią tej uprzywilejowanej grupy będą w stanie utworzyć tę samą przestrzeń nazw w tych samych granicach i w ten sposób uzyskać dostęp do obiektów jądra utworzonych wewnątrz tej granicy, która jest poprzedzona nazwą prywatnej przestrzeni nazw.

Jeśli szkodliwa aplikacja działająca z niskimi uprawnieniami utworzy ten sam deskryptor granicy, ponieważ skradziono na przykład nazwę i SID, przy próbie utworzenia lub otworzenia prywatnej przestrzeni nazw chronionej przez konto o wysokich uprawnieniach dane

wywołanie się nie powiedzie, a `GetLastError` zwróci `ERROR_ACCESS_DENIED`. Jeśli szkodliwa aplikacja ma wystarczające uprawnienia, aby utworzyć lub otworzyć przestrzeń nazw, przejmowanie się tym nie jest już istotne, ponieważ aplikacja ta ma już wystarczającą kontrolę, żeby spowodować więcej zniszczeń niż tylko przechwycenie nazwy obiektu jądra.

Singleton.cpp

```
******************************************************************************
Moduł:  Singleton.cpp
Uwagi: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
*****************************************************************************

#include "stdafx.h"
#include "resource.h"

#include "..\CommonFiles\CmnHdr.h"     /* Zobacz dodatek A. */
#include <windowsx.h>
#include <Sddl.h>          // do zarządzania SID
#include <tchar.h>
#include <strsafe.h>

//////////////////////////////////////////////////////////////////////////////
// Główne okno dialogowe
HWND     g_hDlg;

// Muteks, granica i przestrzeń nazw używane do wykrycia poprzedniej
//działającej instancji
HANDLE   g_hSingleton = NULL;
HANDLE   g_hBoundary = NULL;
HANDLE   g_hNamespace = NULL;

// Sprawdzanie, czy przestrzeń nazw została utworzona lub otwarta
BOOL     g_bNamespaceOpened = FALSE;

// Nazwy granicy i prywatnej przestrzeni nazw
PCTSTR   g_szBoundary = TEXT("3-Boundary");
PCTSTR   g_szNamespace = TEXT("3-Namespace");
#define DETAILS_CTRL GetDlgItem(g_hDlg, IDC_EDIT_DETAILS)
//////////////////////////////////////////////////////////////////////////////
// Dodaje łańcuch tekstowy do pola edycyjnego "Details"
void AddText(PCTSTR pszFormat, ...) {

   va_list argList;
   va_start(argList, pszFormat);

   TCHAR sz[20 * 1024];

   Edit_GetText(DETAILS_CTRL, sz, _countof(sz));
   _vstprintf_s(
      _tcschr(sz, TEXT('\0')), _countof(sz) - _tcslen(sz),
      pszFormat, argList);
   Edit_SetText(DETAILS_CTRL, sz);
   va_end(argList);
}
//////////////////////////////////////////////////////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
   switch (id) {
      case IDOK:
```

```
      case IDCANCEL:
         // Użytkownik kliknął przycisk Exit
         // lub zamknął okno dialogowe klawiszem ESCAPE
         EndDialog(hwnd, id);
         break;
   }
}
///////////////////////////////////////////////////////////////////////////////
void CheckInstances() {

   // Utwórz deskryptor granicy
   g_hBoundary = CreateBoundaryDescriptor(g_szBoundary, 0);

   // Utwórz SID odpowiadający grupie lokalnych administratorów
   BYTE localAdminSID[SECURITY_MAX_SID_SIZE];
   PSID pLocalAdminSID = &localAdminSID;
   DWORD cbSID = sizeof(localAdminSID);
   if (!CreateWellKnownSid(
      WinBuiltinAdministratorsSid, NULL, pLocalAdminSID, &cbSID)) {
      AddText(TEXT("AddSIDToBoundaryDescriptor failed: %u\r\n"),
         GetLastError());
      return;
   }

   // Skojarz SID z deskryptorem granicy
   // --> tylko aplikacje działające w kontekście administratora będą
   // w stanie uzyskać dostęp do obiektów jądra w tej samej przestrzeni nazw
   if (!AddSIDToBoundaryDescriptor(&g_hBoundary, pLocalAdminSID)) {
      AddText(TEXT("AddSIDToBoundaryDescriptor failed: %u\r\n"),
         GetLastError());
      return;
   }

   // Utwórz przestrzeń nazw jedynie dla lokalnych administratorów
   SECURITY_ATTRIBUTES sa;
   sa.nLength = sizeof(sa);
   sa.bInheritHandle = FALSE;
   if (!ConvertStringSecurityDescriptorToSecurityDescriptor(
      TEXT("D:(A;;GA;;;BA)"),
      SDDL_REVISION_1, &sa.lpSecurityDescriptor, NULL)) {
      AddText(TEXT("Security Descriptor creation failed: %u\r\n"),
                   GetLastError());
      return;
   }

   g_hNamespace =
      CreatePrivateNamespace(&sa, g_hBoundary, g_szNamespace);

   // Nie zapominaj o zwolnieniu pamięci dla deskryptora zabezpieczeń
   LocalFree(sa.lpSecurityDescriptor);
   // Sprawdź wynik utworzenia prywatnej przestrzeni nazw
   DWORD dwLastError = GetLastError();
   if (g_hNamespace == NULL) {
      // Nie ma nic do zrobienia, jeśli dostęp zabroniony
      // --> ten kod musi działać na koncie administracyjnym
      if (dwLastError == ERROR_ACCESS_DENIED) {
         AddText(TEXT("Access denied when creating the namespace.\r\n"));
         AddText(TEXT("   You must be running as Administrator.\r\n\r\n"));
```

```
         return;

      } else {
         if (dwLastError == ERROR_ALREADY_EXISTS) {
         // Jeśli inna instancja utworzyła już tę przestrzeń nazw,
         // zamiast tego musimy ją otworzyć.
          AddText(TEXT("CreatePrivateNamespace failed: %u\r\n"), dwLastError);
            g_hNamespace = OpenPrivateNamespace(g_hBoundary, g_szNamespace);
            if (g_hNamespace == NULL) {
               AddText(TEXT("   and OpenPrivateNamespace failed: %u\r\n"),
                  dwLastError);
               return;
            } else {
               g_bNamespaceOpened = TRUE;
               AddText(TEXT("   but OpenPrivateNamespace succeeded\r\n\r\n"));
            }
         } else {
            AddText(TEXT("Unexpected error occurred: %u\r\n\r\n"),
               dwLastError);
            return;
         }
      }
   }

   // Spróbuj utworzyć obiekt muteks z nazwą
   // opartą na prywatnej przestrzeni nazw
   TCHAR szMutexName[64];
   StringCchPrintf(szMutexName, _countof(szMutexName), TEXT("%s\\%s"),
      g_szNamespace, TEXT("Singleton"));

   g_hSingleton = CreateMutex(NULL, FALSE, szMutexName);
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      // Istnieje już instancja tego obiektu
      AddText(TEXT("Another instance of Singleton is running:\r\n"));
      AddText(TEXT("--> Impossible to access application features.\r\n"));
   } else  {
      // Obiekt Singleton jest tworzony po raz pierwszy
      AddText(TEXT("First instance of Singleton:\r\n"));
      AddText(TEXT("--> Access application features now.\r\n"));
   }
}
////////////////////////////////////////////////////////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_SINGLETON);

   // Zapamiętaj uchwyt okna głównego okna dialogowego
   g_hDlg = hwnd;

   // Sprawdź, czy działają już inne instancje
   CheckInstances();

   return(TRUE);
}
////////////////////////////////////////////////////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
```

```
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
   }

   return(FALSE);
}
///////////////////////////////////////////////////////////////////////////////
int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
   UNREFERENCED_PARAMETER(hPrevInstance);
   UNREFERENCED_PARAMETER(lpCmdLine);

   // Pokaż główne okno
   DialogBox(hInstance, MAKEINTRESOURCE(IDD_SINGLETON), NULL, Dlg_Proc);

   // Nie zapominaj o wyczyszczeniu i zwolnieniu zasobów jądra
   if (g_hSingleton != NULL) {
      CloseHandle(g_hSingleton);
   }

   if (g_hNamespace != NULL) {
      if (g_bNamespaceOpened) {  // Otwarta przestrzeń nazw
         ClosePrivateNamespace(g_hNamespace, 0);
      } else { // Utworzona przestrzeń nazw
         ClosePrivateNamespace(g_hNamespace, PRIVATE_NAMESPACE_FLAG_DESTROY);
      }
   }

   if (g_hBoundary != NULL) {
      DeleteBoundaryDescriptor(g_hBoundary);
   }

   return(0);
}

//////////////////////////////// Koniec pliku ////////////////////////////////
```

Zbadajmy poszczególne kroki funkcji `CheckInstances`. Po pierwsze, utworzenie deskryptora granicy wymaga identyfikatora tekstowego dla nazwania zakresu, w którym zdefiniowana będzie prywatna przestrzeń nazw. Nazwa ta jest przekazywana jako pierwszy parametr następującej funkcji:

```
HANDLE CreateBoundaryDescriptor(
   PCTSTR pszName,
   DWORD dwFlags);
```

Obecne wersje Windows nie używają drugiego parametru i dlatego należy zawsze przekazywać w nim 0. Sygnatura funkcji zakłada, że zwracaną wartością jest uchwyt do obiektu jądra; jednakże tak nie jest. Zwracana wartość jest wskaźnikiem do struktury zawierającej definicję granicy. Z tego powodu nie należy nigdy przekazywać zwróconej wartości uchwytu do funkcji `CloseHandle`; zamiast tego należy ją przekazywać do funkcji `DeleteBoundaryDescriptor`.

Następnym krokiem jest skojarzenie identyfikatora SID uprzywilejowanej grupy użytkowników poprzez wywołanie następującej funkcji:

```
BOOL AddSIDToBoundaryDescriptor(
   HANDLE* phBoundaryDescriptor,
   PSID pRequiredSid);
```

W przykładowej aplikacji, identyfikator SID grupy lokalnych administratorów jest tworzony poprzez wywołanie `AllocateAndInitializeSid` z `SECURITY_BUILTIN_DOMAIN_RID` i `DOMAIN_ALIAS_RID_ADMINS` jako parametrami opisującymi grupę. Lista dobrze znanych grup jest zdefiniowana w pliku nagłówkowym WinNT.h.

Uchwyt do deskryptora granicy jest przekazywany jako drugi parametr, gdy wywoływana jest następująca funkcja tworząca prywatną przestrzeń nazw:

```
HANDLE CreatePrivateNamespace(
   PSECURITY_ATTRIBUTES psa,
   PVOID pvBoundaryDescriptor,
   PCTSTR pszAliasPrefix);
```

Adres struktury `SECURITY_ATTRIBUTES` przekazywany jako pierwszy parametr do tej funkcji jest używany przez Windows do zezwalania lub nie zezwalania aplikacji wywołującej `OpenPrivateNamespace` na dostęp do przestrzeni nazw i otwieranie lub tworzenie obiektów w tej przestrzeni nazw. Dostępne są dokładnie takie same opcje jak w przypadku folderu w systemie plików. W ten sposób zapewniamy poziom filtrowania przy otwieraniu przestrzeni nazw. Identyfikator SID dodany do deskryptora granicy jest używany do zdefiniowania, kto jest w stanie przekroczyć tę granicę i utworzyć przestrzeń nazw. W przykładowej aplikacji struktura `SECURITY_ATTRIBUTE` jest tworzona przez wywołanie funkcji `ConvertStringSecurityDescriptorToSecurityDescriptor`, która przyjmuje łańcuch znaków o skomplikowanej składni jako pierwszy parametr. Składnia tego łańcucha znaków jest udokumentowana pod adresem http://msdn2.microsoft.com/en-us/library/aa374928.aspx oraz http://msdn2.microsoft.com/en-us/library/aa379602.aspx.

Typem danych parametru `pvBoundaryDescriptor` jest `PVOID`, chociaż `CreateBoundaryDescriptor` zwraca typ `HANDLE` – to nawet Microsoft traktuje zwracany obiekt jako pseudo-uchwyt. Przedrostek, który ma być używany do tworzenia własnych obiektów jądra, jest podawany jako trzeci parametr. Jeśli spróbuje się utworzyć prywatną przestrzeń nazw, która już istnieje, `CreatePrivateNamespace` zwróci `NULL`, a `GetLastError` zwróci `ERROR_ALREADY_EXISTS`. Wtedy trzeba otworzyć istniejącą, prywatną przestrzeń nazw, korzystając z następującej funkcji:

```
HANDLE OpenPrivateNamespace(
   PVOID pvBoundaryDescriptor,
   PCTSTR pszAliasPrefix);
```

Należy zwrócić uwagę, że obiekty `HANDLE` zwracane przez `CreatePrivateNamespace` i `OpenPrivateNamespace` nie są uchwytami do obiektów jądra; zamyka się je, wywołując `ClosePrivateNamespace`:

```
BOOLEAN ClosePrivateNamespace(
   HANDLE hNamespace,
   DWORD dwFlags);
```

Jeśli tworzona jest przestrzeń nazw, która nie ma być widoczna po zamknięciu, należy przekazać `PRIVATE_NAMESPACE_FLAG_DESTROY` jako drugi parametr, w przeciwnym razie należy przekazać 0. Granica jest zamykana albo gdy proces kończy działanie, albo jeśli zostanie

wywołana funkcja `DeleteBoundaryDescriptor` z pseudo-uchwytem granicy jako jedynym parametrem. Przestrzeń nazw nie może być zamykana, podczas gdy używany jest obiekt jądra. Jeśli przestrzeń nazw zostanie zamknięta, podczas gdy istnieje w niej jakiś obiekt, możliwe będzie utworzenie innego obiektu jądra z tą samą nazwą w tej samej ponownie utworzonej przestrzeni nazw w tej samej granicy, co umożliwi ponownie ataki DoS.

Podsumowując, prywatna przestrzeń nazw jest po prostu katalogiem, w którym tworzone są obiekty jądra. Tak jak inne katalogi prywatna przestrzeń nazw ma skojarzony ze sobą deskryptor zabezpieczeń, który jest ustawiany przy wywoływaniu `CreatePrivateNamespace`. Jednakże w przeciwieństwie do katalogów w systemie plików, przestrzeń nazw nie zawiera elementu nadrzędnego ani nazwy – deskryptor granicy jest używany jako nazwa, przez którą można odwoływać się do przestrzeni nazw. Z tej przyczyny obiekty jądra utworzone z przedrostkiem w oparciu o prywatną przestrzeń nazw pojawiają się w programie Process Explorer z firmy Sysinternals z przedrostkiem „...\” zamiast oczekiwanego „nazwa_przestrzeni_nazw\”. Przedrostek „...\” ukrywa tę informację, dając dodatkową ochronę przed potencjalnymi hakerami. Nazwa nadawana prywatnej przestrzeni nazw jest aliasem widocznym tylko wewnątrz danego procesu. Inne procesy (a nawet ten sam proces) mogą otwierać tę samą prywatną przestrzeń nazw i nadawać jej inny alias.

Przy tworzeniu zwykłych katalogów sprawdzane są uprawnienia na poziomie katalogu nadrzędnego w celu określenia, czy można utworzyć podkatalog. Przy tworzeniu przestrzeni nazw wykonywane jest sprawdzanie granicy – poświadczenie bieżącego wątku musi zawierać wszystkie identyfikatory SID, które składają się na tą granicę.

Duplikowanie uchwytów do obiektów

Ostatnia technika współdzielenia obiektów jądra ponad granicami procesów wymaga użycia funkcji `DuplicateHandle`:

```
BOOL DuplicateHandle(
   HANDLE hSourceProcessHandle,
   HANDLE hSourceHandle,
   HANDLE hTargetProcessHandle,
   PHANDLE phTargetHandle,
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   DWORD dwOptions);
```

Mówiąc po prostu, funkcja ta kopiuje element z tabeli uchwytów dla jednego procesu do tabeli uchwytów dla innego procesu. `DuplicateHandle` ma kilka parametrów, ale sposób ich użycia jest jasny. W najogólniejszej sytuacji użycie `DuplicateHandle` może się wiązać z trzema różnymi procesami działającymi w systemie.

W wywołaniu `DuplicateHandle` pierwszy i trzeci parametr – `hSourceProcessHandle` i `hTargetProcessHandle` – są uchwytami do obiektów jądra. Same uchwyty muszą być utworzone względem procesu wywołującego funkcję `DuplicateHandle`. Ponadto parametry te muszą identyfikować procesy; funkcja nie powiedzie się, jeśli przekazane zostaną uchwyty do innych rodzajów obiektów jądra. Obiekty jądra dla procesów omówimy bardziej szczegółowo w rozdziale 4; na razie wystarczy wiedzieć, że obiekt procesu jest tworzony za każdym razem, gdy nowy proces jest uruchamiany w systemie.

Drugi parametr, `hSourceHandle`, jest uchwytem do obiektu jądra dowolnego typu. Jednakże wartość uchwytu nie jest określona względem procesu wywołującego `DuplicateHandle`.

Uchwyt ten musi być określony względem procesu identyfikowanego przez uchwyt `hSourceProcessHandle`. Czwarty parametr, `phTargetHandle`, jest adresem zmiennej `HANDLE`, która jako wartość otrzyma uchwyt z tego elementu tabeli uchwytów dla procesu identyfikowanego przez `hTargetProcessHandle`, do którego zostanie skopiowana informacja dotycząca uchwytu źródłowego.

Trzy ostatnie parametry `DuplicateHandle` pozwalają określić wartość maski dostępu i flagi dziedziczenia, które powinny być zastosowane w docelowym elemencie przechowującym ten uchwyt obiektu jądra. Parametr `dwOptions` może mieć wartość 0 (zero) lub być dowolną kombinacją następujących dwóch flag: `DUPLICATE_SAME_ACCESS` i `DUPLICATE_CLOSE_SOURCE`.

Podanie `DUPLICATE_SAME_ACCESS` informuje `DuplicateHandle`, że docelowy uchwyt ma mieć taką samą maskę dostępu jak uchwyt z procesu źródłowego. Użycie tej flagi powoduje, że `DuplicateHandle` ignoruje parametr `dwDesiredAccess`.

Podanie `DUPLICATE_CLOSE_SOURCE` powoduje zamknięcie uchwytu w procesie źródłowym. Ta flaga ułatwia przekazywanie obiektu jądra do innego procesu Gdy flaga ta jest użyta, licznik użycia obiektu jądra nie ulega zmianie.

Użyję przykładu w celu pokazania, jak działa `DuplicateHandle`. Na potrzeby tej demonstracji proces S jest procesem źródłowym, który obecnie ma dostęp do pewnego obiektu jądra, a proces T procesem docelowym, który uzyska dostęp do tego obiektu jądra. Proces C jest katalizatorem, który uruchomi wywołanie `DuplicateHandle`. W tym przykładzie będę używać konkretnych liczb dla wartości uchwytów tylko w celu zademonstrowania, jak ta funkcja działa. W aplikacjach rzeczywistych wartości uchwytów przechowywane byłyby w zmiennych, które byłyby przekazywane jako argumenty do funkcji.

Tabela uchwytów dla procesu C (tabela 3-4) zawiera dwie wartości uchwytów: 1 i 2. Wartość uchwytu 1 określa obiekt jądra dla procesu S, a wartość uchwytu 2 określa obiekt jądra dla procesu T.

Tabela 3-4 Tabela uchwytów dla procesu C

Indeks	Wskaźnik do bloku pamięci dla obiektu jądra	Maska dostępu (wartość DWORD tworząca bity flagi)	Flagi
1	0xF0000000 (obiekt jądra dla procesu S)	0x????????	0x00000000
2	0xF0000010 (obiekt jądra dla procesu T)	0x????????	0x00000000

Tabela 3-5 jest tabelą uchwytów procesu S, która zawiera jeden element z uchwytem o wartości 2. Ten uchwyt może identyfikować dowolny typ obiektu jądra – nie musi to być obiekt procesu.

Tabela 3-5 Tabela uchwytów dla procesu S

Indeks	Wskaźnik do bloku pamięci dla obiektu jądra	Maska dostępu (wartość DWORD tworząca bity flagi)	Flagi
1	0x00000000	(brak)	(brak)
2	0xF0000020 (dowolny obiekt jądra)	0x????????	0x00000000

Tabela 3-6 pokazuje, co zawiera tabela uchwytów dla procesu T, zanim proces C wywoła funkcję `DuplicateHandle`. Jak widać, tabela uchwytów dla procesu T zawiera tylko pojedynczy element z wartością uchwytu 2, pierwsza pozycja w tabeli jest obecnie nieużywana.

Tabela 3-6 Tabela uchwytów dla procesu T przed wywołaniem DuplicateHandle

Indeks	Wskaźnik do bloku pamięci dla obiektu jądra	Maska dostępu (wartość DWORD tworząca bity flagi)	Flagi
1	0x00000000	(brak)	(brak)
2	0xF0000030 (dowolny obiekt jądra)	0x????????	0x00000000

Jeśli teraz proces C wywoła `DuplicateHandle`, używając następującego kodu, zmieni się tylko tabela uchwytów dla procesu T, jak pokazano w tabeli 3-7:

```
DuplicateHandle(1, 2, 2, &hObj, 0, TRUE, DUPLICATE_SAME_ACCESS);
```

Tabela 3-7 Tabela uchwytów dla procesu T po wywołaniu DuplicateHandle

Indeks	Wskaźnik do bloku pamięci dla obiektu jądra	Maska dostępu (wartość DWORD tworząca bity flagi)	Flagi
1	0xF0000020	0x????????	0x00000001
2	0xF0000030 (dowolny obiekt jądra)	0x????????	0x00000000

Drugi element z tabeli uchwytów dla procesu S został przekopiowany do pierwszego elementu tabeli uchwytów dla procesu T. Funkcja `DuplicateHandle` wypełniła też zmienną `hObj` z procesu C wartością 1 będącą indeksem elementu w tabeli uchwytów dla procesu T, w którym został umieszczony nowy element.

Ponieważ przekazano flagę `DUPLICATE_SAME_ACCESS` do funkcji `DuplicateHandle`, maska dostępu dla tego uchwytu w tabeli dla procesu T będzie identyczna z maską dostępu w elemencie tabeli dla procesu S. Przekazanie flagi `DUPLICATE_SAME_ACCESS` powoduje również, że funkcja `DuplicateHandle` ignoruje swój parametr `dwDesiredAccess`. Na koniec warto zwrócić uwagę, że bit dziedziczenia został włączony, ponieważ przekazano wartość `TRUE` w parametrze `bInheritHandle` funkcji `DuplicateHandle`.

Tak jak w przypadku dziedziczenia jedną z dziwnych spraw związanych z funkcją `DuplicateHandle` jest to, że proces docelowy nie jest powiadamiany, że nowy obiekt jądra jest teraz dla niego dostępny. Proces C musi więc jakoś poinformować proces T, że ma teraz dostęp do obiektu jądra i musi użyć jakiejś formy komunikacji między procesami w celu przekazania wartości uchwytu z `hObj` do procesu T. Oczywiście użycie argumentu wiersza polecenia lub modyfikacja zmiennych środowiskowych procesu T są wykluczone, ponieważ proces jest już uruchomiony. Musi zostać użyty komunikat okienkowy lub jakiś inny mechanizm komunikacji między procesami.

To, co właśnie opisałem, to najogólniejszy przypadek użycia `DuplicateHandle`. Jak widać jest to bardzo elastyczna funkcja. Jednakże jest rzadko używana w sytuacjach angażujących trzy różne procesy (częściowo ponieważ jest bardzo mało prawdopodobne, że proces C będzie znał wartość uchwytu obiektu używanego przez proces S). Zwykle funkcja `DuplicateHandle` jest wywoływana, gdy mamy do czynienia jedynie z dwoma procesami.

Wyobraźmy sobie sytuację, gdzie jeden proces ma dostęp do obiektu, do którego dostęp chce też uzyskać inny proces albo przypadek, w którym jeden proces chce nadać dostęp do obiektu jądra innemu procesowi. Na przykład powiedzmy, że proces S ma dostęp do jakiegoś obiektu jądra i chce, aby proces T uzyskał dostęp do tego obiektu. W tym celu należy wywołać `DuplicateHandle` w następujący sposób:

```
// Cały poniższy kod jest wykonywany przez proces S.

// Utwórz obiekt muteks dostępny dla procesu S.
HANDLE hObjInProcessS = CreateMutex(NULL, FALSE, NULL);


// Pobierz uchwyt do obiektu jądra dla procesu T.
HANDLE hProcessT = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
   dwProcessIdT);

HANDLE hObjInProcessT;   // Niezainicjowany uchwyt określony względem procesu T.

// Daj procesowi T dostęp do obiektu muteks.
DuplicateHandle(GetCurrentProcess(), hObjInProcessS, hProcessT,
   &hObjInProcessT, 0, FALSE, DUPLICATE_SAME_ACCESS);

// Użyj jakiegoś mechanizmu komunikacji między procesami,
// aby przekazać wartość uchwytu hObjInProcessS do procesu T.
...
// Nie musimy się już komunikować z procesem T.
CloseHandle(hProcessT);
...
// Gdy proces S nie musi już korzystać z obiektu muteks, powinien go zamknąć.
CloseHandle(hObjInProcessS);
```

Wywołanie `GetCurrentProcess` zwraca pseudo-uchwyt, który zawsze identyfikuje proces wywołujący – tutaj proces S. Gdy nastąpi powrót z `DuplicateHandle`, `hObjInProcessT` będzie zawierać uchwyt w kontekście procesu T identyfikujący ten sam obiekt co uchwyt `hObjInProcessS` używany przez kod w procesie S. Proces S nie powinien nigdy wykonywać kodu:

```
// Proces S nigdy nie powinien próbować zamykać zduplikowanego uchwytu.
CloseHandle(hObjInProcessT);
```

Gdyby proces S wykonał ten kod, wywołanie to mogłoby się powieść lub nie powieść. Ale nie w tym tkwi problem. Wywołanie powiodłoby się, gdyby proces S miał akurat dostęp do obiektu jądra z taką samą wartością uchwytu co `hObjInProcessT`. Dałoby to nieoczekiwany efekt w postaci zamknięcia jakiegoś nieokreślonego obiektu jądra, a następna próba uzyskania dostępu do tego obiektu przez proces S na pewno spowodowałaby niepożądane zachowanie aplikacji (mówiąc delikatnie).

Oto inny sposób użycia funkcji `DuplicateHandle`: załóżmy, że proces ma dostęp do odczytu i do zapisu do obiektu mapowania pliku. W pewnym momencie wywoływana jest funkcja, która ma uzyskać dostęp do obiektu mapowania pliku, jedynie go odczytując. Aby zwiększyć sprawność naszej aplikacji, możemy skorzystać z funkcji `DuplicateHandle` do utworzenia nowego uchwytu do istniejącego obiektu i zapewnić, że ten nowy uchwyt będzie miał dostęp tylko do odczytu do tego obiektu. Następnie moglibyśmy przekazać ten uchwyt tylko do odczytu do tej funkcji; w ten sposób kod funkcji nigdy nie będzie w stanie przypadkowo zapisać czegoś w obiekcie mapowania pliku.

Poniższy kod ilustruje ten przykład:

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE,
   LPTSTR szCmdLine, int nCmdShow) {

   // Utwórz obiekt mapowania pliku; uchwyt ma dostęp do odczytu i do zapisu.
   HANDLE hFileMapRW = CreateFileMapping(INVALID_HANDLE_VALUE,
      NULL, PAGE_READWRITE, 0, 10240, NULL);

   // Utwórz inny uchwyt do obiektu mapowania pliku;
   // ten uchwyt ma dostęp tylko do odczytu.
   HANDLE hFileMapRO;
   DuplicateHandle(GetCurrentProcess(), hFileMapRW, GetCurrentProcess(),
      &hFileMapRO, FILE_MAP_READ, FALSE, 0);

   // Wywołaj funkcję, która powinna jedynie odczytywać z mapowania pliku.
   ReadFromTheFileMapping(hFileMapRO);

   // Zamknij obiekt mapowania pliku tylko do odczytu.
   CloseHandle(hFileMapRO);

   // Nadal możemy odczytywać/zapisywać w obiekcie mapowania pliku, używając
hFileMapRW.
   ...
   // Gdy kod główny nie korzysta już z dostępu do tego mapowania pliku,
   // zamknij go.
   CloseHandle(hFileMapRW);
}
```