3-8272-5702-6
Windows 2000 developer's guide

Kapitel 3

Threads und Prozesse

Dieses Kapitel zeigt, wie man Threads und Prozesse erzeugt, verwaltet und in Multithreading-Anwendungen unter Windows 2000 einsetzt. Der größte Teil dieses Kapitels behandelt die effiziente Synchronisierung zwischen Threads und Prozessen in Multithreading-Anwendungen mit den Möglichkeiten, die Windows 2000 bietet. Auf der Begleit-CD finden Sie mehrere Beispielprojekte, die die Konzepte des Multithreading und der Synchronisierung verdeutlichen.

3.1 Prozesse und Threads

Windows 2000 startet einen Prozess, wenn eine Anwendung gestartet wird. Der Prozess besitzt den Speicher, Ressourcen und Ausführungsthreads, die mit der betreffenden Instanz der Ausführung eines ausführbaren Programms verbunden sind. Beim Starten eines Prozesses wird auch ein primärer Thread gestartet. Solange wenigstens ein Thread mit einem Prozess verbunden ist, setzt der Prozess seine Ausführung fort.

Ein Thread – auch als *Ausführungsthread* bezeichnet – ist die kleinste Ausführungseinheit in Windows 2000. Ein Thread ist immer mit einem Prozess verbunden und lebt immer innerhalb eines bestimmten Prozesses. Auch wenn viele Prozesse nur einen einzelnen Thread haben, der für die gesamte Lebensdauer des Prozesses existiert, kann ein Prozess durchaus auch mehrere Threads in seinem Leben haben, wie Abbildung 3.1 zeigt.

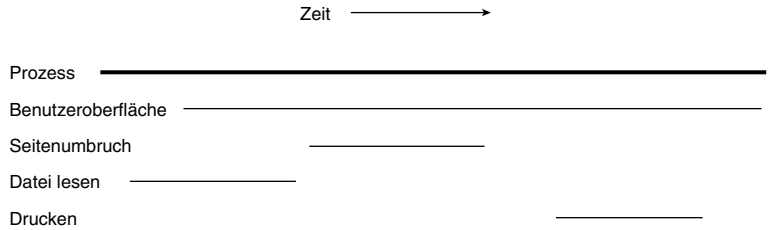


Bild 3.1: Ein typischer Prozess in Windows 2000 hat in seinem Leben viele Threads

Threads werden entsprechend ihrer Priorität und innerhalb der gleichen Priorität in kreisförmiger Art (Round-Robin) geplant. Windows 2000 unterscheidet 31 verschiedene Prioritätsebenen (siehe Abbildung 3.2).

Ein Thread kann sich in einem von sechs Zuständen befinden:

- Waiting (Wartend)
- Ready (Bereit)
- Running (Laufend)
- Standby (Bereitschaft)

- Terminated (Beendet)
- Transition (Übergang)

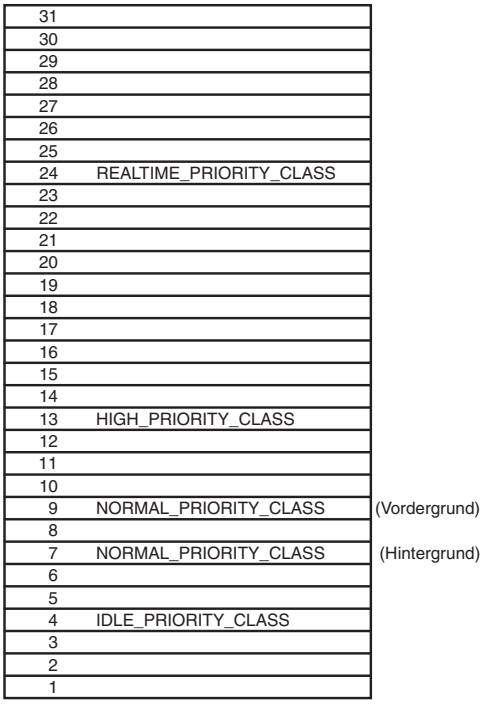


Bild 3.2: In Windows 2000 sind 31 unterschiedliche Prioritätsebenen verfügbar

Die drei letzten Zustände in der Liste sind eigentlich »Subzustände«. In diesen Subzuständen befindet sich der Thread entweder im Zustand *wartend*, wie im Fall eines Threads in Bereitschaft, oder im Zustand *laufend*, wie im Fall des Übergangszustandes. Ein Thread im Zustand *beendet* hat die Ausführung beendet und wird gerade vom Prozess entfernt.

Zu jedem beliebigen Zeitpunkt befindet sich nur genau ein Thread pro Systemprozessor im Zustand *laufend*. Alle anderen Threads nehmen den Zustand *wartend* oder *bereit* ein. Ein laufender Thread kann seine Ausführung so lange fortsetzen, bis eine der folgenden Bedingungen eintritt:

- Der Thread überschreitet seine ihm maximal zugestandene Ausführungszeit, das sogenannte *Quantum*.
- Ein als wartend markierter Thread höherer Priorität geht in den Wartezustand über.
- Der laufende Thread muss auf ein Ereignis oder ein Objekt warten.

Wenn ein Thread sein Zeitquantum überschreitet, markiert Windows 2000 den Thread als *bereit* und sucht nach dem Thread im Zustand *bereit* mit der höchsten Priorität im System. Gibt es andere Threads in diesem Zustand mit der gleichen Prioritätsebene wie der bislang laufende Thread, markiert das System den nächsten Thread als *laufend* und startet seine Ausführung. Auf diese Weise werden Threads mit der gleichen Prioritätsebene in einer karusselartigen Reihenfolge der Warteschlange bedient. Wenn ein Thread auf das Signal eines Ein-/Ausgabeereignisses oder eines anderen Objekts wartet, wird ein anderer Thread ausgewählt, um nach dem gleichen Planungsschema zu laufen.

Damit Windows-basierte Anwendungen reaktionsschneller sind, wird der Thread, der das Vordergrundfenster besitzt, normalerweise um zwei Prioritätsebenen nach oben gesetzt. Diese Verstärkung wirkt nur, wenn der Thread mit der üblichen Prioritätsklasse läuft. Gehört der Thread zu einer der Prioritätsklassen *Leerlauf*, *Hoch* oder *Echtzeit*, hat die Umschaltung in den Vordergrund keinen Einfluss auf seine Prioritätsebene.

Wenn ein Thread infolge einer großen Zahl höher priorisierter Aufgaben nicht laufen kann, gilt er als »verhungert«. Threads mit einer extrem niedrigen Priorität erhalten gelegentlich einen »Prioritätsschub«, damit sie für einen Ausführungszyklus eingeplant werden. In den meisten Fällen läßt sich damit garantieren, dass alle Threads wenigstens eine Chance bekommen, sich um CPU-Zyklen zu bewerben.

3.2 Prozesse verwalten

Einen Prozess startet man normalerweise über den Windows 2000-Explorer, über das START-Menü oder durch die Eingabe des Programmnamens auf der Befehlszeile. Windows 2000 bietet auch eine Reihe von Funktionen, über die sich Prozesse erzeugen und verwalten lassen.

3.2.1 Die Funktion CreateProcess

Wie bereits erwähnt, lässt sich ein Prozess nach verschiedenen Methoden starten. Um einen Prozess aus einer anderen Windows 2000-Anwendung heraus zu starten, rufen Sie die Funktion `CreateProcess` auf:

```
STARTUPINFO si;
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
PROCESS_INFORMATION pi;
BOOL fCreated = CreateProcess(_T("C:\\foo.exe"),
                              NULL,
                              NULL,
                              NULL,
                              FALSE,
```

```
                                CREATE_NEW_CONSOLE,
                                NULL,
                                _T("C:\\"),
                                &si,
                                &pi);
HANDLE hProcess = pi.hProcess;  // Prozess-Handle
```

Die zehn Parameter der Funktion `CreateProcess` haben folgende Bedeutung:

- `lpApplicationName`: Der Pfad zur ausführbaren Datei, die zu starten ist. Wenn dieser Parameter `NULL` ist, bestimmt das System den Dateinamen aus dem nächsten Parameter.
- `lpCommandLine`: Die Befehlszeile zum Starten des Prozesses. Dieser Parameter kann `NULL` sein, wenn keine Befehlszeilenargumente an den neuen Prozess zu übergeben sind.
- `lpProcessAttributes`: Ein Zeiger auf eine `SECURITY_ATTRIBUTES`-Struktur, in der die Sicherheitsattribute für den primären Thread, den der neue Prozess erzeugt, festgelegt sind. Beim Wert `NULL` erhält der Thread einen Standardsicherheitsdeskriptor vom Betriebssystem.
- `lpThreadAttributes`: Ein Zeiger auf eine `SECURITY_ATTRIBUTES`-Struktur mit den Sicherheitsattributen für den primären Thread, den der neue Prozess erzeugt hat. Beim Wert `NULL` erhält der Thread einen Standardsicherheitsdeskriptor vom Betriebssystem.
- `bInheritHandles`: Dieses Flag gibt an, ob der neue Prozess Handles vom aufrufenden Prozess erbt – im Beispiel `FALSE`.
- `dwCreationFlags`: Diese Flags steuern die Prioritätsklasse und das Erstellen des Prozesses. Im Beispiel hat dieser Parameter den Wert `CREATE_NEW_CONSOLE`. Weitere mögliche Werte finden Sie im Anschluss an diese Parameterbeschreibung.
- `lpEnvironment`: Ein Zeiger auf einen Umgebungsblock für den neuen Prozess. Wenn dieser Parameter NULL ist, verwendet der neue Prozess die Umgebung des übergeordneten (aufrufenden) Prozesses.
- `lpCurrentDirectory`: Das aktuelle Laufwerk und Verzeichnis für den neuen Prozess. Wenn dieser Parameter `NULL` ist, erzeugt das Betriebssystem den neuen Prozess mit demselben Laufwerk und aktuellen Verzeichnis wie den übergeordneten Prozess.
- `lpStartupInfo`: Ein Zeiger auf eine `STARTUPINFO`-Struktur, über die sich verschiedene Attribute für den neuen Prozess setzen lassen, beispielsweise Titel und Position des Fensters. Die `STARTUPINFO`-Struktur sollten Sie wie im Beispiel initialisieren. Insbesondere ist das Feld `cb` mit der Größe der Struktur `STARTUPINFO` zu initialisieren.

- `lpProcessInformation`: Ein Zeiger auf eine `PROCESS_INFORMATION`-Struktur, die Windows 2000 mit Identifizierungsinformationen über den neuen Prozess füllt.

Der Parameter `dwCreationFlags` kann einen oder mehrere der folgenden Werte annehmen:

- `CREATE_DEFAULT_ERROR_MODE`: Verhindert, dass der neue Prozess den vom übergeordneten Prozess gesetzten Fehlermodus erbt.
- `CREATE_NEW_CONSOLE`: Der neue Prozess erhält eine eigene Konsole, statt die Konsole des übergeordneten Prozesses zu erben. Dieses Flag lässt sich nicht zusammen mit dem Flag `DETACHED_PROCESS` verwenden.
- `CREATE_NEW_PROCESS_GROUP`: Legt fest, dass der neue Prozess die Wurzel einer neuen Prozessgruppe ist.
- `CREATE_SEPARATE_WOW_VDM`: Dieses Flag ist nur beim Starten einer Windows-basierten 16-Bit-Anwendung gültig. Es legt fest, dass der neue Prozess seine eigenen Fenster in der virtuellen DOS-Maschine (VDM) von Windows erhält (WOW = Windows on Windows).
- `CREATE_SHARED_WOW_VDM`: Dieses Flag ist nur beim Starten einer Windows-basierten 16-Bit-Anwendung gültig. Wenn es gesetzt ist, läuft der neue Prozess in einer gemeinsam genutzten virtuellen DOS-Maschine (VDM).
- `CREATE_SUSPENDED`: Das Betriebssystem erstellt den primären Thread des neuen Prozesses in einem suspendierten Zustand.
- `CREATE_UNICODE_ENVIRONMENT`: Legt fest, dass der Umgebungsblock mit Unicode- anstelle von ANSI-Zeichen arbeitet.
- `DEBUG_PROCESS`: Wenn dieses Flag gesetzt ist, gilt der übergeordnete Prozess als Debugger, während der neue Prozess den zu debuggenden Prozess darstellt, d.h. der übergeordnete Prozess empfängt Ereignisse, die der neue Prozess generiert.
- `DEBUG_ONLY_THIS_PROCESS`: Der neue Prozess wird nicht mit demselben Debugger wie der übergeordnete Prozess debuggt.
- `DETACHED_PROCESS`: Verhindert, dass der neue Prozess auf das Konsolenfenster des übergeordneten Prozesses zugreift. Dieses Flag lässt sich nicht zusammen mit dem Flag `CREATE_NEW_CONSOLE` verwenden.
- `IDLE_PRIORITY_CLASS`: Bezeichnet einen Prozess, dessen Threads nur laufen, wenn sich das System im Leerlauf befindet.
- `BELOW_NORMAL_PRIORITY_CLASS`: Bezeichnet einen Prozess, dessen Threads auf einer Prioritätsstufe zwischen `IDLE_PRIORITY_CLASS` und `NORMAL_PRIORITY_CLASS` laufen sollen. Diese Zwischenstufe steht erstmalig mit Windows 2000 zur Verfügung.

- `NORMAL_PRIORITY_CLASS`: Bezeichnet einen Prozess ohne bestimmte Anforderungen an den Zeitplan. Diesen Wert der Thread-Priorität sollten Sie für alle üblichen Fälle verwenden.
- `ABOVE_NORMAL_PRIORITY_CLASS`: Bezeichnet einen Prozess, dessen Threads auf einer Prioritätsstufe zwischen `NORMAL_PRIORITY_CLASS` und `HIGH_PRIORITY_CLASS` laufen sollten. Diese Zwischenstufe steht erstmalig mit Windows 2000 zur Verfügung.
- `HIGH_PRIORITY_CLASS`: Bezeichnet einen Prozess, dessen Threads zeitkritische Aufgaben innerhalb einer minimalen Zeitverzögerung durchführen müssen. Dieses Flag sollten Sie mit Vorsicht einsetzen, weil die damit markierten Threads den Vorrang gegenüber den meisten anderen Threads im System erhalten. Untergeordnete Prozesse erben diese Prioritätsstufe nicht.
- `REALTIME_PRIORITY_CLASS`: Bezeichnet einen Prozess, dessen Threads die höchst mögliche Priorität erhalten sollen. Threads, die auf dieser Stufe laufen, überholen im wahrsten Sinne des Wortes Teile des Betriebssystems. Da Threads mit diesem Attribut schneller ausgeführt werden als die virtuelle Speicherverwaltung und andere interne Prozesse, sind Deadlocks bei diesen Threads möglich, wodurch sich das System fehlerhaft verhalten kann. Untergeordnete Prozesse erben diese Prioritätsstufe nicht.

Die Funktion `CreateProcess` gibt `TRUE` zurück, wenn sie den neuen Prozess starten konnte. Lässt sich der Prozess nicht starten, gilt das normalerweise als Fehler im neuen Prozess. Soweit es `CreateProcess` betrifft, ist die Arbeit damit beendet.

Die Struktur `PROCESS_INFORMATION` enthält Informationen über den neuen Prozess und seinen primären Thread. Die Struktur hat vier Elemente:

- `hProcess`: Liefert einen Handle für den neuen Prozess.
- `hThread`: Liefert einen Handle auf den primären Thread des neuen Prozesses.
- `dwProcessId`: Liefert eine ID-Nummer für den neuen Prozess. Dieser Wert ist nur für die Lebensdauer des Prozesses gültig. Sobald dieser Prozess beendet ist, kann Windows 2000 den Wert an einen neuen Prozess zuweisen.
- `dwThreadId`: Liefert eine ID-Nummer für den primären Thread des neuen Prozesses.

3.2.2 Prozesse beenden

Einen Prozess kann man nach drei Verfahren beenden. Zu bevorzugen ist der Aufruf der Funktion `ExitProcess` in einem der Threads, die im Prozess laufen:

```
ExitProcess(NO_ERROR);
```

Durch Aufruf von `ExitProcess` kann der Prozess ordnungsgemäß herunterfahren – beispielsweise die Einsprungpunkt-Funktionen der vom Prozess geladenen DLLs aufrufen, Signalisieren von Threads, die auf das Beenden des Prozesses warten, und Schließen aller Objekt-Handles des Prozesses.

Nachdem der Prozess beendet ist, gibt die Funktion `GetExitCodeProcess` den als Parameter an `ExitProcess` übergebenen Wert zurück. Wenn Sie die Funktion `GetExitCodeProcess` aufrufen, bevor der Prozess beendet ist, lautet der Rückgabewert `STILL_ACTIVE`.

Einen Prozess kann man auch durch Aufruf von `TerminateProcess` beenden. Dieser Funktion übergibt man den Handle des zu beendenden Prozesses:

```
TerminateProcess(hProcess, NO_ERROR);
```

Nach Möglichkeit sollte man auf dieses Verfahren verzichten. Es entlädt keine DLLs und führt auch keine anderen Aufräumarbeiten des Systems aus, die normalerweise beim Beenden eines Prozesses erforderlich sind.

Schließlich kann man einen Prozess beenden, indem man alle Threads des Prozesses beendet. Diese Methode verwenden die meisten Win32-Programme. Das Betriebssystem beendet den Prozess, sobald alle Threads ihre Arbeit beendet haben.

3.3 Threads verwalten

Im Gegensatz zu vielen bekannten Multithreading-Betriebssystemen bietet Windows 2000 echte Threads. Bei den meisten Multithreading-Betriebssystemen muss man einen neuen Prozess erzeugen, wenn man einen neuen Ausführungspfad erstellt. Ein Windows 2000-Thread ist hinsichtlich der Computerressourcen wesentlich kostengünstiger zu erstellen als ein typischer UNIX-Prozess und bietet einen weiten Bereich von Verwaltungsoptionen. Die nächsten Abschnitte erläutern, wie man Threads in einer Windows 2000-Anwendung erzeugt und verwaltet.

3.3.1 Threads erzeugen

Ein Thread läßt sich nach vier Verfahren erzeugen:

- Starten eines neuen Prozesses
- Aufrufen der Win32-API-Funktion `CreateThread`
- Aufrufen der Funktion `_beginthread` aus der C-Laufzeitbibliothek
- Aufrufen der Funktion `_beginthreadex` aus der C-Laufzeitbibliothek

Wenn man einen neuen Prozess mit der Funktion `CreateProcess` startet, steht die Thread-ID im Feld `dwThreadId` der Struktur `PROCESS_INFORMATION`. Das Betriebssystem füllt diese Struktur beim Rücksprung der Funktion. Der Handle des Threads ist im Feld `hThread` dieser Struktur enthalten.

Einen neuen Thread mit CreateThread starten

Einen Thread erzeugen Sie durch Aufruf von `CreateThread` wie im folgenden Beispiel:

```
long WINAPI ThreadEntry(LPARAM lparam)
{
    // ...
}
unsigned long nThreadID;
HANDLE hThread = CreateThread(NULL,
                              0,
                              (LPTHREAD_START_ROUTINE)ThreadEntry,
                              (void*)szHello,
                              0,
                              &nThreadID);
```

Die Funktion `CreateThread` übernimmt sechs Parameter:

- `lpThreadAttributes`: Ein Zeiger auf eine `SECURITY_ATTRIBUTES`-Struktur mit den Sicherheitsattributen für den neuen Thread. Wenn man `NULL` für diesen Parameter übergibt, erhält der neue Thread vom Betriebssystem einen Standardsicherheitsdeskriptor.
- `dwStackSize`: Die anfängliche Stack-Größe für den neuen Thread. Bei 0 für diesen Parameter erhält der neue Thread die gleiche Größe wie der primäre Thread. In der Regel ist das ein brauchbarer Standardwert, da Windows 2000 die Stack-Größe bei Bedarf erhöht.
- `lpStartAddress`: Die Adresse einer Startfunktion. Der neue Thread beginnt die Ausführung in dieser Funktion. Er terminiert, sobald er aus dieser Funktion zurückspringt.
- `lpParameter`: Ein 32-Bit-Parameter, der an die Startfunktion des neuen Threads übergeben wird.
- `dwCreationFlags`: Diese Flags steuern das Erstellen des Threads. Wenn Sie den Wert CREATE_SUSPENDED angeben, erzeugt das Betriebssystem den Thread in einem suspendierten Zustand. Ist der Wert gleich 0, startet der Thread sofort nach der Erstellung. Um einen suspendierten Thread weiter auszuführen, ist die Funktion `ResumeThread` aufzurufen.
- `lpThreadId`: Ein Zeiger auf eine 32-Bit-Variable, die bei Rückkehr von `CreateThread` die Thread-ID enthält.

Wenn Threads Funktionen der C-Laufzeitbibliothek verwenden, sollte man auf `CreateThread` verzichten. Andernfalls können auf Grund der Interaktion von `CreateThread` mit der C-Laufzeitbibliothek kleine Speicherlücken entstehen, wenn der Thread terminiert. Falls Sie ohne Funktionen der C-Laufzeitbibliothek nicht auskommen, nehmen Sie am besten `_beginthread` oder `_beginthreadex`, auf die der nächste Abschnitt eingeht.

Thread-Funktionen der C-Laufzeitbibliothek

Zum Erstellen von Threads bietet der Visual C++-Compiler zwei Erweiterungen der C-Laufzeitbibliothek:

- `_beginthread`
- `_beginthreadex`

Hinweis

Diese Funktionen erfordern es, dass man eine der Multithread-Versionen der C-Laufzeitbibliothek verwendet. Wenn Sie mit der integrierten Entwicklungsumgebung von Visual C++ arbeiten, können Sie die Laufzeitbibliothek über PROJEKT / EINSTELLUNGEN im Dialogfeld PROJEKTEINSTELLUNGEN festlegen. Auf der Registerkarte C++ wählen Sie im Dropdown-Listenfeld KATEGORIE den Eintrag Code Generation und dann im Dropdown-Listenfeld LAUFZEIT-BIBLIOTHEK die vom Projekt verwendete Bibliothek.

Mit beiden Funktionen können Sie einen Thread erstellen, der sicher mit der C-Laufzeitbibliothek zusammenarbeiten kann. Am Einfachsten lässt sich die Funktion `_beginthread` verwenden, wie Listing 3.1 in einem Beispiel zeigt.

Listing 3.1: Einen Worker-Thread mit _beginthread starten

```
/* Kompilieren Sie diesen Code mit der Laufzeitbibliothek Multithreaded
*/
#include <windows.h>
#include <tchar.h>
#include <process.h>
#ifndef UNICODE
    #include <stdio.h>
#endif

void threadFunc(void* pv);

int _tmain()
{
    ULONG tid = 0;
    _tprintf(_T("Hello aus dem Haupt-Thread\n"));
    tid = _beginthread(threadFunc, 0, NULL);
    if(tid == -1)
```

```
    {
        /* Fehler behandeln */
    }
    WaitForSingleObject((HANDLE)tid, INFINITE);
    _tprintf(_T("Verabschiedung aus dem Haupt-Thread\n"));
    return 0;
}

void threadFunc(void* pv)
{
    _tprintf(_T("Hello aus dem Worker-Thread\n"));
    _tprintf(_T("Arbeiten, arbeiten, arbeiten\n"));
    _tprintf(_T("Verabschiedeung aus dem Worker-Thread\n"));
}
```

Die drei Parameter der Funktion `_beginthread` haben folgende Bedeutung:

- `start_address`: Die Adresse der Startfunktion, mit der der neue Thread die Ausführung beginnt.
- `stack_size`: Die Größe des Stacks für den neuen Thread oder 0, wenn die C-Laufzeitbibliothek automatisch den Stack verwalten soll.
- `arglist`: Ein `void*`-Parameter mit einem optionalen Argument, das an die Startfunktion des Threads übergeben wird.

Die Funktion `_beginthread` gibt einen Handle auf den gestarteten Thread zurück oder `-1`, wenn sich der Thread nicht starten lässt.

Gegenüber der Funktion `CreateThread` fehlen bei `_beginthread` einige Optionen, beispielsweise die Fähigkeit, einen Thread im suspendierten Zustand zu erzeugen, und die Möglichkeit, einen Sicherheitsdeskriptor für den neuen Thread bereitzustellen.

Wie Listing 3.2 zeigt, ist die Funktion `_beginthreadex` etwas komplizierter als `_beginthread`, hält sich aber mehr an das Modell der Win32-Funktion `CreateThread`.

Listing 3.2: Einen Worker-Thread mit _beginthreadex starten

```
#include <windows.h>
#include <tchar.h>
#include <process.h>
#ifndef UNICODE
    #include <stdio.h>
#endif

unsigned __stdcall threadFunc(void* pv);

int _tmain()
```

```
{
    ULONG hThread = 0;
    unsigned threadId = 0;

    /* Sicherheitsdeskriptor für Worker-Thread initialisieren */
    SECURITY_DESCRIPTOR* psd;
    psd = (SECURITY_DESCRIPTOR*)LocalAlloc(LPTR,
                                SECURITY_DESCRIPTOR_MIN_LENGTH);
    InitializeSecurityDescriptor(psd,SECURITY_DESCRIPTOR_REVISION);
    SetSecurityDescriptorDacl(psd, TRUE, NULL, FALSE);

    _tprintf(_T("Hello aus dem Haupt-Thread\n"));
    /* Thread in einem suspendierten Zustand erstellen */
    hThread = _beginthreadex(psd,
                             0,
                             threadFunc,
                             NULL,
                             CREATE_SUSPENDED,
                             &threadId);
    if(!hThread)
    {
        /* Fehler behandeln */
    }
    /* Worker-Thread aktivieren */
    ResumeThread((HANDLE)hThread);
    WaitForSingleObject((HANDLE)hThread, INFINITE);
    _tprintf(_T("Verabschiedung aus dem Haupt-Thread\n"));

    CloseHandle((HANDLE)hThread);
    LocalFree(psd);
    return 0;
}

unsigned __stdcall threadFunc(void* pv)
{
    _tprintf(_T("Hello aus dem Worker-Thread\n"));
    _tprintf(_T("Arbeiten, arbeiten, arbeiten\n"));
    _tprintf(_T("Verabschiedung aus dem Worker-Thread\n"));
    return NO_ERROR;
}
```

Die sechs Parameter der Funktion `_beginthreadex` haben folgende Bedeutung:

- `security`: Ein Zeiger auf eine `SECURITY_DESCRIPTOR`-Struktur für den neuen Thread oder `NULL`, um einen Standardsicherheitsdeskriptor zu verwenden.
- `stack_size`: Die Größe des Stacks für den neuen Thread oder 0, wenn die C-Laufzeitbibliothek den Stack automatisch verwalten soll.

- `start_address`: Die Adresse einer Startfunktion, mit der der neue Thread die Ausführung beginnt.
- `arglist`: Ein Zeiger auf eine Liste von Argumenten, die an den Thread zu übergeben sind, oder `NULL`, wenn keine Argumente verwendet werden.
- `initflag`: Der anfängliche Startzustand des Threads – entweder `CREATE_SUSPENDED`, um den Thread in einem suspendierten Zustand zu erzeugen, oder 0, um den Thread sofort im laufenden Zustand zu erzeugen.
- `thrdaddr`: Die Adresse einer Variablen vom Typ `unsigned int`, die das Betriebssystem mit der Thread-ID für den neuen Thread bei Rückkehr der Funktion füllt.

Die Funktion `_beginthread` gibt einen Handle auf den gestarteten Thread zurück oder 0, wenn sich der Thread nicht starten ließ.

3.3.2 Threads beenden

Einen Thread kann man auf drei Wegen beenden. Der »normale« Weg führt über einen Aufruf der Funktion `ExitThread` aus dem endenden Thread. Im Aufruf von `ExitThread` übergibt man einen Beendigungscode als Parameter:

```
ExitThread(NO_ERROR);
```

Das zweite Verfahren erlaubt dem Thread, seine Thread-Funktion zu beenden. Die Funktion `ExitThread` wird implizit aufgerufen, wenn man aus der Startfunktion des Threads zurückspringt. Zum Beispiel führt die folgende Thread-Funktion eine `for`-Schleife fünfmal aus und gibt dann `NO_ERROR` als Beendigungscode des Threads zurück:

```
long WINAPI ThreadFunc(long lParam)
{
    for(int n = 0; n < 5; n++)
    {
        // Mit Thread arbeiten
    }
    return NO_ERROR;
}
```

Schließlich kann man einen Thread mit der Funktion `TerminateThread` beenden. Allerdings ist dieses Verfahren nicht zu empfehlen. Beim Aufruf von `TerminateThread` hat ein Thread keine Möglichkeit, teilweise ausgeführte Arbeiten ordnungsgemäß abzuschließen. Der Thread kann auch kritische Abschnitte besitzen, die nicht freigegeben werden. Der Aufruf von `TerminateThread` sieht folgendermaßen aus:

```
HANDLE hThread = CreateThread(....);
.
.
TerminateThread(hThread, NO_ERROR);
```

Die Funktion `TerminateThread` hat zwei Parameter: den Thread-Handle und einen Beendigungscode für den Thread.

Wenn mit einer dieser Methoden der letzte Thread eines Prozesses beendet wird, terminiert der Prozess.

Der Beendigungscode für einen Thread läßt sich mit der Funktion `GetExitCodeThread` ermitteln:

```
DWORD dwResult;
GetExitCodeThread(hThread, &dwResult);
if(dwResult != NO_ERROR)
{
    // Fehler behandeln
}
```

Die Funktion `GetExitCodeThread` hat zwei Parameter: den Thread-Handle und die Adresse einer 32-Bit-Variablen, die das Betriebssystem mit dem Beendigungscode des Threads füllt. Wenn man die Funktion `GetExitCodeThread` für einen noch aktiven Thread aufruft, lautet der Rückgabewert `STILL_ACTIVE`.

3.3.3 Thread-Prioritäten abrufen und ändern

Unter Windows 2000 lässt sich die Priorität eines Threads dynamisch ändern. Allerdings sollte man dabei mit Umsicht vorgehen, denn eine zu hoch gesetzte Priorität des Threads kann die Leistung des gesamten Systems negativ beeinflussen.

Windows 2000 bietet sechs Prioritätsklassen:

- `IDLE_PRIORITY_CLASS`
- `BELOW_NORMAL_PRIORITY_CLASS`
- `NORMAL_PRIORITY_CLASS`
- `ABOVE_NORMAL_PRIORITY_CLASS`
- `HIGH_PRIORITY_CLASS`
- `REALTIME_PRIORITY_CLASS`

Für jeden Thread kann man die Ausführung unabhängig von seiner Prioritätsklasse in einem Bereich von fünf Prioritätsebenen festlegen:

- `THREAD_PRIORITY_LOWEST`: Zwei Ebenen unter der Prioritätsklasse des Prozesses.

- `THREAD_PRIORITY_BELOW_NORMAL`: Eine Ebene unter der Prioritätsklasse des Prozesses.
- `THREAD_PRIORITY_NORMAL`: Die gleiche Ebene wie die Prioritätsklasse des Prozesses.
- `THREAD_PRIORITY_ABOVE_NORMAL`: Eine Ebene über der Prioritätsklasse des Prozesses.
- `THREAD_PRIORITY_HIGHEST`: Zwei Ebenen über der Prioritätsklasse des Prozesses.

Darüber hinaus lassen sich an die Funktion `SetThreadPriority` zwei spezielle Prioritätsebenen übergeben:

- `THREAD_PRIORITY_IDLE`: Setzt die Priorität des Threads immer auf 1. Ist allerdings die Prioritätsklasse auf `REALTIME_PRIORITY_CLASS` eingestellt, gilt die Prioritätsebene 16.
- `THREAD_PRIORITY_TIME_CRITICAL`: Setzt die Priorität des Threads immer auf 15. Ist jedoch die Prioritätsklasse auf `REALTIME_PRIORITY_CLASS` eingestellt, gilt die Prioritätsebene 31. Das ist die einzige Möglichkeit, mit der man einen Thread mit der Prioritätsstufe 31 laufen lassen kann.

3.3.4 Lokaler Thread-Speicher

Automatische Variablen – zum Beispiel Variablen, die man innerhalb des Funktionsrumpfes deklariert – werden im Kontext des momentan laufenden Threads angelegt. Das bedeutet, dass jeder Thread über seine eigene Kopie aller automatischen Variablen verfügt. Normalerweise ist das eine gute Sache, weil man den Zugriff auf die auf dem Stack angelegten Variablen nicht synchronisieren muss.

Schwieriger wird es, wenn man auf globale Variablen zurückgreifen muss: Erstens ist der Zugriff auf die Variable zu synchronisieren, so dass mehrere Threads nicht gleichzeitig versuchen, die Variable zu modifizieren. Zweitens nutzen alle Threads eine normale globale Variable gemeinsam – außer einem Array gibt es keine Möglichkeit, Daten nur auf einen bestimmten Thread bezogen zu speichern.

Windows 2000 behandelt dieses Problem in einer speziellen Weise. Mit *lokalem Thread-Speicher* kann man thread-spezifisch verwaltete Variablen erzeugen, wobei der Programmierer selbst nur sehr wenig zur Verwaltung beitragen muss. Man unterscheidet zwei Arten des lokalen Thread-Speichers:

- Statischer lokaler Thread-Speicher
- Dynamischer lokaler Thread-Speicher

Die nächsten Abschnitte gehen auf die Vorteile beider Verfahren näher ein.

Statischer lokaler Thread-Speicher

Statischer lokaler Thread-Speicher ist leicht zu verwenden. Es gibt weder Funktionsaufrufe, noch sind spezielle Vorkehrungen zu treffen oder verzwickte Compilerschalter einzustellen.

Der statische lokale Thread-Speicher ist in einem speziellen Speicherbereich mit dem Namen `.tls` im Adressraum des jeweiligen Prozesses untergebracht. Wenn man einen neuen Thread erzeugt, erstellt Windows 2000 eine neue Kopie des lokalen Thread-Speichers für den neuen Thread und zerstört den Speicherblock, wenn der Thread beendet wird. Ein Thread kann nur auf die thread-lokalen Variablen zugreifen, die für diesen Thread bestimmt sind.

Eine Variable im lokalen Thread-Speicher deklariert man mit `__declspec(thread)` als Teil der Variablendeklaration:

```
__declspec(thread) int nMeals = 0;
```

Mit einer `typedef`-Anweisung lässt sich der Code verständlicher formulieren:

```
typedef __declspec(thread) TLS;
TLS int nMeals = 0;
```

Eine Variable im lokalen Thread-Speicher verwendet man wie jede andere Variable. Man kann sie lesen, beschreiben oder ihre Adresse bestimmen, genau wie bei anderen globalen Variablen. Den Zugriff auf die Variable muss man nicht synchronisieren, da sie nur genau ein Thread verwenden kann.

Eine Variable im lokalen Thread-Speicher kann man im globalen Gültigkeitsbereich oder als statische Variable in einer Funktion deklarieren, allerdings nie als reine automatische Variable.

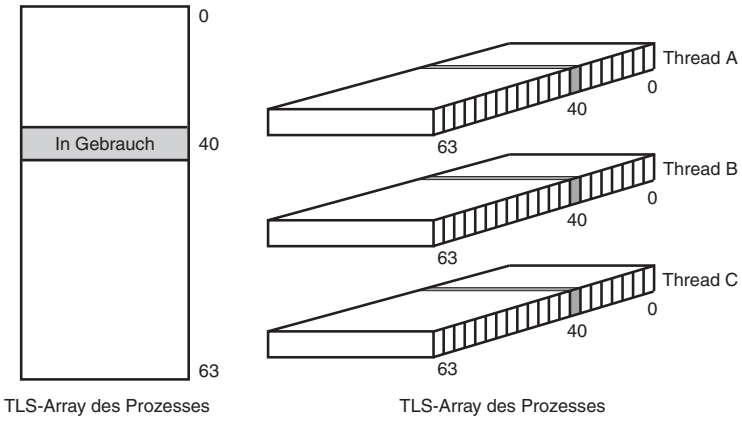
Dynamischer lokaler Thread-Speicher

Gegenüber dem statischen lokalen Thread-Speicher ist der dynamische lokale Thread-Speicher zwar etwas komplizierter, lässt sich aber flexibler einsetzen, weil man ihn nach Bedarf reservieren und freigeben kann. Eine Anwendung legt den dynamischen lokalen Thread-Speicher zur Laufzeit an.

Windows 2000 verwaltet den dynamischen lokalen Thread-Speicher, und jeder Prozess greift auf den lokalen Thread-Speicher über Indizes zu, wie es Abbildung 3.3 zeigt.

Alle Threads eines Prozesses verfügen über eigene Kopien des lokalen Thread-Speichers. Windows 2000 verwaltet ein Statusflag für die Indizes des lokalen Thread-Speichers und reserviert einen neuen Bereich für jeden Prozess, der einen neuen Speicherindex anfordert. Den lokalen Thread-Speicher verwaltet man mit vier Funktionen:

- `TlsAlloc` fordert einen neuen Index für lokalen Thread-Speicher von Windows 2000 an.
- `TlsSetValue` ruft ein Thread auf, um einen 32-Bit-Wert in seiner Kopie des lokalen Thread-Speichers abzulegen.
- `TlsGetValue` ruft ein Thread auf, um einen vorher im lokalen Thread-Speicher abgelegten 32-Bit-Wert abzurufen.
- `TlsFree` benachrichtigt Windows 2000, dass ein Prozess einen bestimmten Index für lokalen Thread-Speicher nicht mehr benötigt.



(TLS = Thread Local Storage, Lokaler Thread-Speicher)

Bild 3.3: Lokaler Thread-Speicher verwendet Indizes, die Windows 2000 verwaltet

Jeder Thread im Prozess kann auf seine eigene Kopie des lokalen Thread-Speichers zugreifen. Werte anderer Threads kann er nicht verändern.

3.3.5 Thread-ID bestimmen

Beim Erstellen eines Threads weist ihm das Betriebssystem eine ID (Identifizierungsnummer) zu. Die Thread-ID kennzeichnet den Thread eindeutig im gesamten System – keine zwei Prozesse enthalten Threads mit derselben Thread-ID. Die Thread-ID ist ein nützliches Tag, das man in vielen thread-bezogenen API-Aufrufen verwendet.

Die Thread-ID kann man nach zwei Verfahren bestimmen:

- Wenn man den Thread mit der Funktion `CreateThread`, `_beginthread` oder `_beginthreadex` erzeugt, speichert man die zurückgegebene Thread-ID.
- Man ruft die Funktion `GetCurrentThreadId` auf.

Wenn man lediglich die Thread-IDs für eine kleine Zahl von Worker-Threads wissen will, ist es recht einfach, die Thread-ID beim Erstellen des Threads zu speichern. Allerdings versagt diese Lösung, wenn die Thread-ID des primären Threads für den Prozess zu bestimmen ist. Weiterhin ist ein nicht unerheblicher Buchführungsaufwand notwendig, um eine größere Zahl von Worker-Threads zu verfolgen.

Die Funktion `GetCurrentThreadId` hat keine Parameter und liefert die Thread-ID für den Thread, der die Funktion aufgerufen hat:

```
DWORD dwThreadID = GetCurrentThreadId();
```

Für den eigenen Thread kann man auch einen Pseudo-Handle mit der Funktion `GetCurrentThread` abrufen:

```
HANDLE hThread = GetCurrentThread();
```

Ein *Pseudo-Handle* ist kein wirklicher Thread-Handle, dient aber als Platzhalter für alle Funktionen, die einen echten Thread-Handle benötigen. Außerdem muss man einen Pseudo-Handle nicht schließen.

Mit der Funktion `GetCurrentProcess` kann man einen Pseudo-Handle für den aktuellen Prozess abrufen:

```
HANDLE hProcess = GetCurrentProcess();
```

3.3.6 Thread-Affinität

Unter Thread-Affinität versteht man die Verbindung zwischen einem bestimmten Thread und einem anderen Objekt. Kapitel 16 behandelt die Thread-Affinität bei COM-Objekten. Im vorliegenden Kapitel geht es um die Thread-Affinität in Bezug auf bestimmte CPUs.

Unter Windows 2000 kann man Thread-Affinität für einen Prozess oder Thread spezifizieren, so dass dieser auf einer Untermenge der verfügbaren Prozessoren laufen kann. Legt man die Thread-Affinität für einen Thread oder Prozess fest, lässt sich *in bestimmten Fällen* die Leistung verbessern, indem man die Verwendung des CPU-Caches optimiert.

Die APIs der Thread- und Prozess-Affinität funktionieren nur bei Mehrprozessorsystemen – wenn Sie mit einer Einzel-CPU-Maschine arbeiten, sind alle Threads an eine einzelne CPU gebunden. Beachten Sie, dass auf einer SMP-Maschine die für Ihre Threads oder Prozesse definierte Thread-Affinität nicht verhindert, dass andere Threads die von Ihnen bevorzugten CPUs benutzen – bei unpassendem Einsatz können die APIs der Thread- und Prozessor-Affinität die Leistung der Anwendung negativ beeinflussen.

Die Thread-Affinitäts-Maske ist eine `DWORD`-Variable, bei der Bit 0 (das niederwertigste Bit) die erste CPU, das Bit 1 die zweite CPU usw. repräsentiert.

Die Affinitätsmaske lässt sich für den gesamten Prozess oder für einen bestimmten Thread im Prozess setzen. Um die Thread-Affinität für einen Thread zu setzen, rufen Sie die Funktion `SetThreadAffinityMask` wie in Listing 3.3 gezeigt auf.

Listing 3.3: Ausführung eines Threads auf der zweiten CPU erzwingen

```
#include <windows.h>
#include <tchar.h>

int _tmain()
{
    // Pseudohandle für Thread ermitteln
    HANDLE hThread = GetCurrentThread();
    DWORD dwAffinity = 0x02; // Nur auf zweitem Prozessor ausführen!
    DWORD dwOldAffinity = SetThreadAffinityMask(hThread, dwAffinity);
    _tprintf(_T("Alte Affinitätsmaske: %x\n"), dwOldAffinity);
    .
    .
    .
    return 0;
}
```

Die Funktion `SetThreadAffinityMask` hat zwei Parameter:

- `hThread`: Ein Handle auf den Thread, dessen Affinitätsmaske die Funktion setzt.
- `dwThreadAffinityMask`: Die neue Affinitätsmaske für den Thread.

Die Funktion `SetThreadAffinityMask` gibt die vorherige Affinitätsmaske des Threads zurück.

Mit der Funktion `SetProcessAffinityMask` kann man die Affinitätsmaske für alle Threads in einem Prozess festlegen, wie Listing 3.4 zeigt.

Listing 3.4: Ausführung eines Prozesses auf der zweiten CPU erzwingen

```
#include <windows.h>
#include <tchar.h>

int _tmain()
{
    // Pseudohandle für Prozess ermitteln
    HANDLE hProcess = GetCurrentProcess();
    DWORD  dwAffinity = 0x02;  // Nur auf zweitem Prozessor ausführen!
    BOOL fSetAffinity = SetProcessAffinityMask(hProcess,
                                               dwAffinity);
    .
```

```
    .
    .
    return 0;
}
```

Die Funktion `SetProcessAffinityMask` hat zwei Parameter:

- `hProcess`: Ein Handle auf den Prozess, dessen Affinitätsmaske die Funktion setzt.
- `dwProcessAffinityMask`: Die neue Affinitätsmaske für die Threads des Prozesses.

Die Funktion `SetProcessAffinityMask` gibt bei einem Fehler den Wert `FALSE` und im Erfolgsfall einen Wert ungleich Null – d.h., nicht die vorherige Affinitätsmaske – zurück. Die Affinitätsmaske für den Prozess können Sie mit der Funktion `GetProcessAffinityMask` abrufen:

```
DWORD dwProcAffinity;
DWORD dwSysAffinity;
HANDLE hProcess = GetCurrentProcess();
BOOL fGetAffinity = GetProcessAffinityMask(hProcess,
                                           &dwProcAffinity,
                                           &dwSysAffinity);
```

Die Funktion `GetProcessAffinityMask` hat drei Parameter:

- `hProcess`: Ein Handle auf den Prozess, dessen Affinitätsmaske man abrufen möchte.
- `lpProcessAffinityMask`: Ein Zeiger auf einen `DWORD`-Wert, den die Funktion mit der Affinitätsmaske des aktuellen Prozesses füllt.
- `lpSystemAffinityMask`: Ein Zeiger auf einen `DWORD`-Wert, den die Funktion mit der Affinitätsmaske für das System füllt. In dieser Affinitätsmaske ist für jeden verfügbaren Prozessor das entsprechende Bit gesetzt.

Wie bereits erwähnt, kann sich die Blockierung eines Threads oder Prozesses auf einen bestimmten Prozessor negativ auf die Gesamtleistung des Systems auswirken. Selbst wenn ein Prozessor verfügbar ist, kann der Thread nicht laufen, falls die Affinitätsmaske keinem der verfügbaren Prozessoren die Ausführung des Threads gestattet.

Mit der Funktion `SetThreadIdealProcessor` können Sie den bevorzugten Prozessor für einen Thread angeben. Das System verhindert die Ausführung des Threads nicht, plant aber die Ausführung wann immer möglich für den spezifizierten Prozessor ein. Eine Maske bevorzugter Prozessoren lässt sich nicht festlegen – man muss einen einzigen Prozessor als den idealen Prozessor spezifizieren:

```
HANDLE hProcess = GetCurrentThread();
DWORD  dwPreferredProc = 0x02;  // Prozessor 3 bevorzugen
DWORD  dwPrevious = SetThreadIdealProcessor(hProcess,
                                            dwPreferredProc);
if(dwPrevious == -1)
    ReportError();
```

Die zwei Parameter der Funktion `SetThreadIdealProcessor` haben folgende Bedeutung:

- `hThread`: Ein Handle auf den Thread, dessen bevorzugter Prozessor festzulegen ist.
- `dwIdealProcessor`: Der bevorzugte Prozessor für den Thread.

Die Funktion `SetThreadIdealProcessor` gibt den vorherigen bevorzugten Prozessor zurück oder `MAXIMUM_PROCESSORS`, wenn der Thread keinen bevorzugten Prozessor hat. Bei einem Fehler liefert die Funktion den Wert `-1`.

3.3.7 Zusammenarbeit mehrerer Threads

Die Verwaltung von Threads und Prozessen in einer Anwendung ist kein spezielles Merkmal von Windows 2000. Nachdem man eine Anwendung in mehr als einen Thread aufgeteilt hat, muss man sich mit Problemen beschäftigen, mit denen man bei Einzelthread-Programmierung überhaupt nicht in Berührung gekommen wäre.

Zum Beispiel muss man einen einfachen Vorgang wie das Lesen und Schreiben einer globalen Variablen in geeigneter Weise synchronisieren. Oftmals können scheinbar »harmlose« Codefragmente gefährliche Bereiche verstecken:

```
/* Globale Variable */
int g_nRequestsOutstanding;

void QueueNewRequest(void)
{
    ++g_nRequestsOutstanding;
    DoSomethingInterestingToQueueRequest();
}

void SatisfyRequest(void)
{
    RemoveOutstandingRequestInAnEfficientManner();
    --g_nRequestsOutstanding;
}
```

Dieses Codefragment zeigt zwei verschiedene Funktionen, die in dieselbe Variable, `g_nRequestsOutstanding`, schreiben. Wenn zwei (oder mehrere) Threads gleichzeitig versuchen, den Wert einer Variablen zu verändern, enthält die Vari-

able am Ende einen falschen Wert. In praxi gibt es zwei verschiedene Arten von Fehlerszenarien:

- Auf Mehrprozessormaschinen kann ein Multithreaded Prozess ohne weiteres zwei Threads gleichzeitig ausführen. Wenn der Code den Zugriff auf Variablen durch mehrere Threads nicht synchronisiert, scheitert der Prozess mit hoher Wahrscheinlichkeit.
- Auf Einprozessormaschinen können zwar nicht mehrere Threads in einem Prozess gleichzeitig laufen, allerdings lässt sich ein Thread an jedem Punkt während der Ausführung des Programms unterbrechen. Findet die Unterbrechung gerade beim Schreiben in eine Variable statt, kann ein anderer Thread in die Variable schreiben, während der erste Thread auf seine Ausführung wartet.

Listing 3.5 zeigt das Beispiel eines Programms, das nicht Thread-sicher ist. Die Konsolenanwendung `BadCount` finden Sie auf der Begleit-CD. Die Anwendung erzeugt zwei Threads, die eine globale Variable bis zum Wert 50000 inkrementieren und sich diese Arbeit teilen sollen.

Listing 3.5: Das Programm BadCount zeigt den falschen Weg, um globale Daten gemeinsam zu nutzen

```
/*
 * BadCount.c - nicht synchronisierter Zugriff auf eine globale Variable
 * über zwei Threads. Mit der Laufzeitbibliothek Multithreaded kompilie-
ren.
 */
#include <windows.h>
#include <tchar.h>
#ifndef UNICODE
    #include <stdio.h>
#endif

/* Globaler Zähler */
int  g_nTheCounter = 0;
void IncCounter();

DWORD WINAPI threadFunc(LPVOID);
int _tmain()
{
    DWORD dwThreadId;
    /* Worker-Thread für Hälfte der Arbeit erstellen */
    HANDLE hThread = CreateThread(NULL,
                                  0,
                                  threadFunc,
                                  NULL,
                                  0,
                                  &dwThreadId);
```

```
    if(!hThread) return 0;
    /*
     * Globale Variable 25000mal inkrementieren, dann auf den
     * Worker-Thread warten, um Ausführung zu beenden.
     */
    IncCounter();
    WaitForSingleObject(hThread, INFINITE);
    /*
     * Wert der globalen Variablen anzeigen, nachdem sie 50000mal
     * inkrementiert wurde.
     */
    _tprintf(_T("Wert des globalen Zählers: %d\n"),
             g_nTheCounter);
    return 0;
}

/*
 *  Worker-Thread-Funktion - Inkrementiert die globale Variable
 *  25000mal und kehrt dann zurück.
 */
DWORD WINAPI threadFunc(LPVOID lpv)
{
    IncCounter();
    return NO_ERROR;
}

/*
 * Gemeinsame Funktion für beide Threads, um die globale
 * Zählervariable zu inkrementieren.
 */
void IncCounter()
{
    int n;
    for(n = 0; n < 25000; ++n)
    {
        ++g_nTheCounter;
    }
}
```

Wie schlecht ist `BadCount`? Das hängt vom jeweiligen Computer ab. Auf der SMP-Maschine des Autors (ein IBM 704 mit zwei Pentium Pro CPUs) liegt der am Ende des Programms gemeldete Zählerstand bei etwa 30000. Auf einer anderen Workstation mit einer einzelnen Alpha CPU lautet das Ergebnis dagegen wie erwartet 50000. Das Problem besteht zwar auch auf einem Einprozessorsystem, nur tritt es dort seltener zutage. Beim Multithreading sind also folgende Punkte zu beachten:

- Nur weil ein Programm auf Ihrem Computer läuft, heißt das nicht, dass es in Ordnung ist.
- SMP-Maschinen sind ausgezeichnete Testwerkzeuge.
- Probleme der Synchronisierung lassen sich nur sehr schwer einkreisen.

Jedes Betriebssystem, das mehrere Ausführungsthreads unterstützt, muss eine Möglichkeit bieten, damit Sie derartige Probleme der Synchronisierung behandeln können. Windows 2000 zeichnet sich durch eine große Zahl von Verwaltungsoptionen aus und ist zum Bersten voll mit Methoden, die Sie bei der Verwaltung von Threads und Prozessen unterstützen. Auf diese Methoden geht der Abschnitt »Synchronisierung« später in diesem Kapitel näher ein.

3.3.8 Wann ein Thread zu erzeugen ist

Verschiedene Probleme lassen sich leicht lösen, indem man eine Anwendung in zwei oder mehrere Threads aufteilt. Eine Anwendung kommt für Multithreading in Frage, wenn sie eines der drei folgenden Kriterien erfüllt:

- Sie verbringt einen großen Teil der Zeit damit, auf abgeschlossene Ein-/Ausgabeoperationen zu prüfen.
- Sie hat eine Anzahl von »Hintergrund-Tasks«, die asynchron auszuführen sind.
- Sie hat andere Tasks, die sich unabhängig voneinander ohne viel Aufwand für die Synchronisierung ausführen lassen.

Wenn man mit mehreren Threads arbeitet, lassen sich Anwendungen mit den genannten Eigenschaften leichter programmieren. Allerdings kann es auch schief gehen, wenn man erstmalig mit Threads zu tun hat. Sehen Sie sich dazu den nächsten Abschnitt an.

3.3.9 Wann kein Thread zu erzeugen ist

Manche Arten von Anwendungen bieten sich von vornherein nicht für Multithreading an. Wenn eine Anwendung eines der folgenden Kriterien erfüllt, kommt Sie für Multithreading höchstwahrscheinlich nicht in Frage:

- Sie ist praktisch aus einem Stück »geschnitzt«.
- Sie verbraucht keine vorhandenen Computerressourcen.
- Es bestehen komplizierte Probleme der Synchronisierung.
- Zwischen den Tasks bestehen viele Abhängigkeiten.

Man sollte immer bedenken, dass eine Anwendung mit mehreren Threads zwar einige Probleme löst, dabei gleichzeitig aber neue Probleme aufwirft, beispielsweise Synchronisierung, Thread- und Objekt-Lebensdauer sowie erhöhte Kom-

plexität in der Testphase. Darüber hinaus gilt: Ein Programm mit mehreren Threads läuft auf einer Einprozessormaschine in den meisten Fällen nicht schneller, sondern oftmals sogar langsamer.

3.4 Synchronisierung

Wie bereits angesprochen, können Probleme auftreten, wenn zwei oder mehrere Threads eine gemeinsame Variable verwenden. Solange mehrere Threads eine Variable nur lesen, verläuft alles problemlos. Versucht allerdings ein Thread, eine gemeinsame Variable zu verändern, ist der Zugriff auf diese Variable zu synchronisieren.

Ein Synchronisierungselement ist ein Objekt, das Sie bei der Verwaltung einer Multithreaded Anwendung unterstützt. In Windows 2000 sind fünf Arten von Synchronisierungselementen verfügbar:

- *Ereignisse* sind Objekte, die der Programmierer erzeugt und die signalisieren, dass eine Variable oder Routine für den Zugriff verfügbar ist.
- *Kritische Abschnitte* sind Codeabschnitte, auf die ein einzelner Thread zu einem bestimmten Zeitpunkt zugreifen kann.
- *Mutexe* sind Windows 2000-Objekte, die sicherstellen, dass nur ein einzelner Thread Zugriff auf eine geschützte Variable oder geschützten Code hat.
- *Semaphoren* sind Mutexen ähnlich, verhalten sich aber wie Zähler, die einer festgelegten Anzahl von Threads den Zugriff auf eine geschützte Variable oder geschützten Code erlauben.
- *Atomare Operationen auf API-Ebene* stellt Windows 2000 bereit, um den Inhalt einer Variablen in einer einzigen Operation inkrementieren, dekrementieren oder austauschen zu können.

Diese Synchronisierungselemente sind jeweils für spezielle Situationen geeignet. Darauf gehen die nächsten Abschnitte ein.

3.4.1 Sperroperationen von Win32

Mit den einfachsten Grundformen der Synchronisierung manipuliert oder testet man den Wert von einer oder zwei Variablen. Die Win32-API umfasst sieben Funktionen, die auch bei mehreren CPUs garantiert atomar und Thread-sicher sind:

- `InterlockedIncrement`: Inkrementiert eine 32-Bit-Variable und gibt den neuen Wert zurück.
- `InterlockedDecrement`: Dekrementiert eine 32-Bit-Variable und gibt den neuen Wert zurück.

- `InterlockedExchange`: Ändert den Wert einer 32-Bit-Variablen auf einen neuen Wert und gibt den vorherigen Wert zurück.
- `InterlockedExchangeAdd`: Inkrementiert den Wert einer 32-Bit-Variablen um einen angegebenen Betrag und gibt den vorherigen Wert zurück.
- `InterlockedExchangePointer`: Ändert den Wert einer 32-Bit-Variablen auf einen neuen Wert und gibt den vorherigen Wert zurück. In 64-Bit-Versionen von Windows 2000 hat dieser Parameter eine Größe von 64 Bit.
- `InterlockedCompareExchange`: Setzt den Wert einer 32-Bit-Variablen in Abhängigkeit von einer Bedingung auf einen neuen Wert und gibt den anfänglichen Wert zurück.
- `InterlockedCompareExchangePointer`: Setzt den Wert einer 32-Bit-Variablen in Abhängigkeit von einer Bedingung auf einen neuen Wert und gibt den anfänglichen Wert zurück. In 64-Bit-Versionen von Windows 2000 haben die Parameter eine Größe von 64 Bit.

Auf Intel-SMP-Systemen müssen die an diese Funktionen übergebenen Variablen auf 32 Bit ausgerichtet sein. Bei 64-Bit-Versionen von Windows 2000 muss man die an `InterlockedExchangePointer` und `InterlockedCompareExchangePointer` übergebenen Variablen auf 64 Bit ausrichten.

Die gebräuchlichsten Funktionen sind `InterlockedIncrement` und `InterlockedDecrement`. Das Programm `GoodCount` in Listing 3.6 ist eine neue Version des in Listing 3.5 präsentierten Beispiels `BadCount`. `GoodCount` verwendet `InterlockedIncrement`, um die globale Variable atomar zu inkrementieren, und läuft auf allen Windows 2000-Systemen einwandfrei.

Listing 3.6: Das Programm GoodCount zeigt den richtigen Weg, um globale Daten gemeinsam zu nutzen

```
/*
 * GoodCount.c - synchronisierter Zugriff auf eine globale Variable über
 * zwei Threads. Mit der Laufzeitbibliothek Multithreaded kompilieren.
 */
#include <windows.h>
#include <tchar.h>
#ifndef UNICODE
    #include <stdio.h>
#endif

/* Globaler Zähler */
long g_nTheCounter = 0;
void IncCounter();

DWORD WINAPI threadFunc(LPVOID);
int _tmain()
```

```
{
    DWORD dwThreadId;
    /* Worker-Thread für Hälfte der Arbeit erstellen */
    HANDLE hThread = CreateThread(NULL,
                                  0,
                                  threadFunc,
                                  NULL,
                                  0,
                                  &dwThreadId);
    if(!hThread) return 0;
    /*
     * Globale Variable 25000mal inkrementieren, dann auf den
     * Worker-Thread warten, um Ausführung zu beenden.
     */
    IncCounter();
    WaitForSingleObject(hThread, INFINITE);
    /*
     * Wert der globalen Variablen anzeigen, nachdem sie 50000mal
     * inkrementiert wurde.
     */
    _tprintf(_T("Wert des globalen Zählers: %d\n"),
             g_nTheCounter);
    return 0;
}

/*
 *  Worker-Thread-Funktion - Inkrementiert die globale Variable
 *  25000mal und kehrt dann zurück.
 */
DWORD WINAPI threadFunc(LPVOID lpv)
{
    IncCounter();
    return NO_ERROR;
}

/*
 * Gemeinsame Funktion für beide Threads, um die globale
 * Zählervariable zu inkrementieren. Diese Version von IncCounter
 * verwendet die Funktion InterlockedIncrement.
 */
void IncCounter()
{
    int n;
    for(n = 0; n < 25000; ++n)
    {
        InterlockedIncrement(&g_nTheCounter);
    }
}
```

3.4.2 Kritische Abschnitte

Ein *kritischer Abschnitt* ist ein Codeabschnitt, der nur von einem Thread zu einem bestimmten Zeitpunkt verwendet werden darf. Wenn zwei oder mehrere Threads gleichzeitig auf einen kritischen Abschnitt zugreifen wollen, erhält nur ein Thread die Steuerung des kritischen Abschnitts. Alle anderen Threads werden blockiert, d.h. gehen in den Wartezustand, bis der kritische Abschnitt freigegeben wird (siehe Abbildung 3.4).

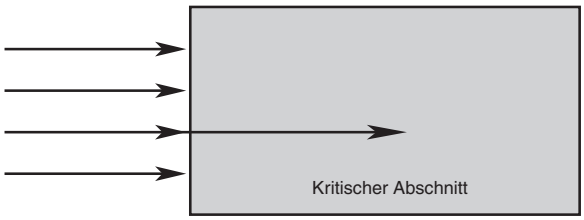


Bild 3.4: Ein kritischer Abschnitt erlaubt nur genau einem Thread die Ausführung zu einem bestimmten Zeitpunkt

Verglichen mit anderen Verfahren zur Synchronisierung, auf die das Kapitel später eingeht, verbrauchen kritische Abschnitte so gut wie keine Computerressourcen. Allerdings kann man einen kritischen Abschnitt im Gegensatz zu den anderen Grundformen der Synchronisierung von Windows 2000 nur innerhalb eines einzelnen Prozesses verwenden.

Einen kritischen Abschnitt schützt man durch eine Variable vom Typ `CRITICAL_SECTION`. Diese Variable ist vor ihrer Verwendung zu initialisieren und muss im Gültigkeitsbereich aller Threads liegen, die auf die Variable zugreifen. Während ihrer Verwendung darf die Variable nicht den Gültigkeitsbereich verlassen. Aus diesem Grund deklariert man Variablen für kritische Abschnitte oftmals im globalen Gültigkeitsbereich.

Die `CRITICAL_SECTION`-Variable initialisiert man mit der Funktion `InitializeCriticalSection`:

```
CRITICAL_SECTION cs;
InitializeCriticalSection(&cs);
```

Um einen kritischen Abschnitt in Besitz zu nehmen, muss ein Thread die Funktion `EnterCriticalSection` aufrufen:

```
EnterCriticalSection(&cs);
```

Wenn der kritische Bereich frei ist, wird er als belegt markiert und der Thread setzt seine Ausführung unmittelbar fort. Sollte der kritische Bereich bereits belegt sein, blockiert der Thread, bis der Abschnitt wieder frei ist.

Hat ein Thread die Arbeit mit der geschützten Variablen oder Funktion abgeschlossen, gibt man den kritischen Abschnitt über die Funktion `LeaveCriticalSection` wieder frei:

```
LeaveCriticalSection(&cs);
```

Ein Thread, der auf dieser `CRITICAL_SECTION`-Variablen blockiert ist, kann die Steuerung der Variablen übernehmen.

Eine einzelne `CRITICAL_SECTION`-Variable kann mehrere damit im Zusammenhang stehende Variablen oder Funktionen schützen. Später in diesem Kapitel sichert eine einzige `CRITICAL_SECTION`-Variable ab, dass nur einer von acht Threads mit den Steuerelementen in einem Dialogfeld arbeitet.

Nachdem ein Thread die Kontrolle über einen kritischen Bereich erhalten hat, hindert er andere Threads daran, die Kontrolle des kritischen Abschnitts zu übernehmen. Um einen flüssigen Ablauf zu erreichen, sollten Threads einen kritischen Abschnitt nur während einer sehr kurzen Zeitdauer belegen.

Jeder Aufruf von `EnterCriticalSection` ist durch einen Aufruf von `LeaveCriticalSection` auszugleichen. Wenn Sie den Aufruf von `LeaveCriticalSection` vergessen, bleiben Threads, die auf den Eintritt in den kritischen Abschnitt warten, blockiert – entweder bis der Prozess endet oder für immer. Solange innerhalb eines kritischen Abschnitts nicht nur einfachste Arbeiten zu erledigen sind, sollte man mit `__try`- und `__finally`-Blöcken absichern, dass die Aufrufe von `EnterCriticalSection` und `LeaveCriticalSection` ausgeglichen sind. Dazu ein Beispiel:

```
void DoSomething()
{
    __try
    {
        EnterCriticalSection(&csOutput);
        .
        .
        // Hier etwas Interessantes erledigen
        .
    }
    __finally
    {
        LeaveCriticalSection(&csOutput);
    }
}
```

In diesem Codefragment kommt die Funktion `LeaveCriticalSection` immer zum Aufruf, selbst wenn die Ausführung innerhalb des kritischen Abschnitts eine Ausnahme auslöst. Man kann einen kritischen Abschnitt auch in eine C++-Klasse einhüllen und in dieser Klasse sicherstellen, dass der kritische Abschnitt korrekt initialisiert und abgebaut wird.

Ein und derselbe Thread darf `EnterCriticalSection` mehrmals mit derselben `CRITICAL_SECTION`-Variablen aufrufen. Das hängt damit zusammen, dass sich in einer großen Anwendung alle möglichen verschachtelten kritischen Abschnitte nur schwer ermitteln lassen. Beispielsweise ruft der folgende Code die Funktion `EnterCriticalSection` zweimal mit derselben `CRITICAL_SECTION`-Variablen auf:

```
void CIsdnTerminal::HandleKeyPress()
{
    EnterCriticalSection(&m_csAction);
    if(ReceivedRelease() == FALSE)
        TranslateKey();
    LeaveCriticalSection(&m_csAction);
}
BOOL CIsdnTerminal::ReceivedRelease()
{
    BOOL fResult;
    EnterCriticalSection(&m_csAction);
    if(m_state == CLEARING)
        fResult = TRUE;
    else
        fResult = m_fReleaseStored;
    LeaveCriticalSection(&m_csAction);
    return fResult;
}
```

Dieses Beispiel gleicht jeden Aufruf von `EnterCriticalSection` durch einen Aufruf von `LeaveCriticalSection` aus. Ruft ein Thread, der eine `CRITICAL_SECTION`-Variable besitzt, die Funktion `EnterCriticalSection` mit derselben Variablen auf, wird ein interner Zähler inkrementiert und der Thread kann seine Arbeit ohne Blockierung fortsetzen. Sobald der interne Zähler wieder der Stand Null erreicht hat, können andere Threads die Kontrolle über den kritischen Abschnitt übernehmen.

Wenn der kritische Abschnitt nicht mehr benötigt wird, gibt man die von ihm belegten Ressourcen mit der Funktion `DeleteCriticalSection` frei:

```
DeleteCriticalSection(&cs);
```

Mit der Funktion `TryEnterCriticalSection` lässt sich der Zustand eines kritischen Abschnitts testen, ohne die Ausführung des Threads zu blockieren:

```
BOOL fAcquired = TryEnterCriticalSection(&cs);
if(fAcquired)
{
    // In geschützte Variable schreiben
    nAgeAlexandria = 6;
    nAgeMackenzie = 2;
}
```

```
else
{
    // Kritischer Abschnitt war nicht in Besitz zu bringen
}
```

Die Funktion `TryEnterCriticalSection` gibt `TRUE` zurück, wenn der Thread die Kontrolle über den kritischen Abschnitt hat. Der Rückgabewert ist `FALSE`, falls ein anderer Thread den kritischen Abschnitt bereits in Besitz genommen hat.

Mit der Funktion `TryEnterCriticalSection` kann man auch versuchen, den kritischen Abschnitt für einen bestimmten Thread zu übernehmen. Falls der kritische Abschnitt verfügbar ist, nimmt ihn der Thread in Besitz. Andernfalls gibt die Funktion `TryEnterCriticalSection` den Wert `FALSE` zurück und der Thread kann sich inzwischen mit anderen Arbeiten beschäftigen.

3.4.3 Ereignisse verwalten

Kritische Abschnitte bieten sich an, wenn Daten oder Funktionen gegenüber mehreren Threads zu schützen sind. In einer Multithreading-Anwendung muss man aber manchmal auch einen anderen Thread darüber benachrichtigen, dass ein Ereignis aufgetreten ist.

In Windows 2000 ist ein *Ereignis* ein vom Betriebssystem verwaltetes Synchronisierungsobjekt. Jedem Ereignis kann ein Name zugeordnet sein. Damit können mehrere Prozesse denselben Ereignis-Handle gemeinsam nutzen.

Ereignisse setzt man ein, wenn ein Thread auf einen anderen Thread warten muss, um eine Aufgabe fertigzustellen, oder wenn ein Thread in den inaktiven Zustand übergehen und auf einen anderen Thread warten muss, um ein aufgetretenes Ereignis anzuzeigen.

Ein ausgelöstes Ereignis kann zwei Zustände einnehmen:

- *Signalisiert*: Eine Warteanforderung für dieses Ereignis wird realisiert.
- *Nicht signalisiert*: Eine Warteanforderung auf diesem Thread wird nicht realisiert und der wartende Thread wird blockiert.

Normalerweise zeigt man mit einem Ereignis an, dass eine bestimmte Aufgabe fertiggestellt ist. Ein Thread, der auf die Fertigstellung dieser Aufgabe wartet, setzt die Verarbeitung fort, sobald das Ereignis signalisiert wird (siehe Abbildung 3.5).

Ereignis-Handles erstellen und schließen

Mit der Funktion `CreateEvent` erzeugt man ein Ereignis:

```
HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, "Ereignisname");
```

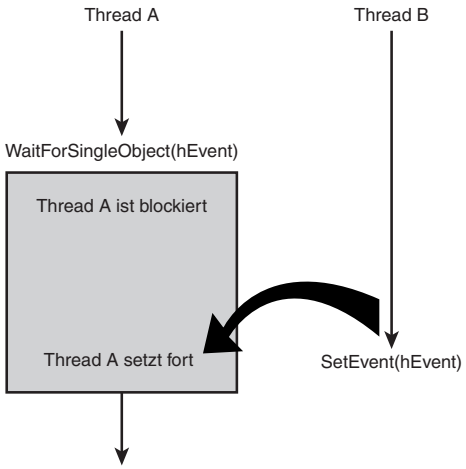


Bild 3.5: Mehrere Threads verwenden Ereignisse zur Synchronisierung

Die vier Parameter der Funktion `CreateEvent` haben folgende Bedeutung:

- `lpEventAttributes`: Ein Zeiger auf eine `SECURITY_ATTRIBUTES`-Struktur. Wenn keine sicherheitsrelevanten Aspekte zu berücksichtigen sind, kann man `NULL` für diesen Parameter übergeben.
- `bManualReset`: Zeigt an, ob das Ereignis manuell zurückzusetzen ist. Auf manuell zurückzusetzende Ereignisse geht dieses Kapitel weiter hinten ein. Normalerweise ist dieser Parameter `FALSE`, d.h. das System setzt den Zustand automatisch auf nicht signalisiert zurück.
- `bInitialState`: Spezifiziert den anfänglichen Zustand des Ereignisses. Wenn dieser Parameter TRUE ist, hat das Ereignis den Zustand signalisiert. Der Wert FALSE gibt an, dass dieses Ereignis in einen nicht signalisierten Zustand gesetzt werden sollte.
- `lpName`: Ein optionaler Name für das Ereignis. Da alle Ereignis-, Mutex- und Dateizuordnungsobjekte im selben Namensbereich liegen, ist darauf zu achten, Namen nicht versehentlich wiederzuverwenden. In den meisten Fällen gibt man hier `NULL` an, insbesondere wenn Ereignisse nicht über Prozessgrenzen hinweg gemeinsam genutzt werden.

Wenn die Funktion das Ereignis erfolgreich erzeugen konnte, gibt sie einen Handle auf das Ereignis zurück, bei einem Fehler ist der Rückgabewert `NULL`.

Handles für Ereignisse sind Windows 2000-Objekte, genau wie Handles von Dateien, Threads und Prozessen. Wenn Sie den Ereignis-Handle nicht mehr benötigen, rufen Sie die Funktion `CloseHandle` auf:

```
CloseHandle(hEvent);
```

Handles für Ereignisse signalisieren und zurücksetzen

Einen Ereignis-Handle setzt man mit der Funktion `SetEvent` in den Zustand signalisiert:

```
SetEvent(hEvent);
```

Die Funktion `SetEvent` übernimmt einen Handle als einzigen Parameter.

Einen automatisch zurückzusetzenden Ereignis-Handle setzt das Betriebssystem zurück, nachdem eine entsprechende Warteanforderung realisiert wurde. Manuell zurückzusetzende Ereignisse muss man explizit mit der Funktion `ResetEvent` zurücksetzen:

```
ResetEvent(hEvent);
```

Automatisch zurückzusetzende Ereignisse bieten sich an, wenn man Initialisierungen oder andere Ereignisse ausführt, die an eine große Zahl von Threads zu melden sind. In diesem Fall ist es sinnvoll, einen Thread im Zustand signalisiert zu belassen, selbst wenn er eine Warteanforderung erfüllt hat. Somit lassen sich alle wartenden Threads bedienen, indem man lediglich ein einziges Ereignis setzt.

Signalisiert man eine Gruppe von Threads mit manuell zurückzusetzenden Ereignissen, lässt sich nur schwer bestimmen, wann alle wartenden Threads freigegeben sind. Windows 2000 bietet zu diesem Zweck die Funktion `PulseEvent`:

```
PulseEvent(hEvent);
```

Die Funktion `PulseEvent` signalisiert die Ereignis-Handles und setzt sie zurück, nachdem alle wartenden Threads freigegeben wurden.

Auf einen Handle warten

Muss ein Thread auf die Signalisierung eines Ereignisses warten, lässt sich das nach verschiedenen Verfahren realisieren. Mit den beiden folgenden Funktionen setzt man den Thread in einen Wartezustand:

- `WaitForSingleObject` verwendet man, wenn der Thread auf einen zu signalisierenden Handle wartet.
- `WaitForMultipleObjects` verwendet man, um ein Array von Handles zu testen.

Die Funktion `WaitForSingleObject` prüft einen als Parameter übergebenen Handle und springt sofort zurück, wenn der Handle signalisiert ist:

```
DWORD dwResult = WaitForSingleObject(hEvent, INFINITE);
```

Die zwei Parameter der Funktion `WaitForSingleObject` haben folgende Bedeutung:

- `hHandle`: Der zu testende Handle.
- `dwMilliseconds`: Ein Wert für die Zeitüberschreitung (in Millisekunden). Der Wert `INFINITE` gibt an, dass die Funktion praktisch ohne Zeitüberschreitung arbeitet.

Bei manuell zurückzusetzenden Ereignissen verbleibt das Ereignis im signalisierten Zustand, selbst wenn die Warteanforderung des ersten Threads bedient wurde. Alle anderen Ereignisse werden durch `WaitForSingleObject` in ihren nicht signalisierten Zustand gesetzt, nachdem eine Warteanforderung bedient wurde.

Die Funktion `WaitForSingleObject` gibt folgende Werte zurück:

- `WAIT_OBJECT_0`, wenn der Handle signalisiert ist. Beachten Sie, dass das letzte Zeichen eine Null und nicht der Buchstabe `O` ist.
- `WAIT_TIMEOUT`, wenn der Handle nicht signalisiert ist und die Zeitüberschreitung abgelaufen ist.
- `WAIT_ABANDONED` wird nur zurückgegeben, wenn ein Thread auf einen Mutex-Handle wartet.

Hüten Sie sich davor, Aufrufe von `WaitForSingleObject` zu verschachteln. Gelegentlich muss ein Thread darauf warten, dass mehrere Handles signalisiert werden. Verschachtelte Aufrufe von `WaitForSingleObject` können gefährlich sein, wenn sich mehrere Threads um mehrere Objekte bewerben, wie es Abbildung 3.6 zeigt.

In Abbildung 3.6 verfügt sowohl Thread A als auch Thread B über eine Ressource, die für die Fortsetzung erforderlich ist.

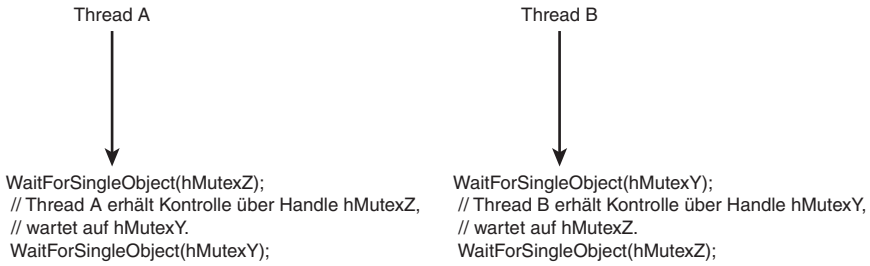


Bild 3.6: Beispiel eines Deadlocks

Leider kann kein Thread die Ausführung fortsetzen, es sei denn, der andere Thread gibt die von ihm bereits in Besitz genommene Ressource wieder frei. Zum Glück bietet Windows 2000 die Möglichkeit, auf mehrere Synchronisierungsobjekte gleichzeitig zu warten. Darauf geht der nächste Abschnitt näher ein.

Auf mehrere Handles warten

Die Funktion `WaitForMultipleObjects` ist der Funktion `WaitForSingleObject` ähnlich, außer das sie mit einem Array von Handles arbeitet. Alle Handles werden gleichzeitig überwacht und keiner ist gegenüber den anderen bevorzugt. Der Aufruf von `WaitForMultipleObjects` sieht folgendermaßen aus:

```
HANDLE hEvents[2];
.
.
.
DWORD dw = WaitForMultipleObjects(2, hEvents, TRUE, INFINITE);
```

Die vier Parameter der Funktion `WaitForMultipleObjects` haben folgende Bedeutung:

- `nCount`: Die Anzahl der zu prüfenden Objekt-Handles.
- `lpHandles`: Die Basisadresse des Handle-Arrays.
- `fWaitAll`: Gibt an, ob alle Handles signalisiert sein müssen oder lediglich eines. Im Beispiel kennzeichnet `TRUE`, dass alle Ereignis-Handles signalisiert sein müssen.
- `dwMilliseconds`: Eine Zeitüberschreitung in Millisekunden oder `INFINITE` bei keiner Zeitüberschreitung.

Wenn man für den dritten Parameter von `WaitForMultipleObjects` den Wert `TRUE` übergibt, behalten alle Ereignis-Handles so lange ihren Zustand bei, bis alle Handles verfügbar sind. Das ist notwendig, um Deadlock-Bedingungen zu vermeiden, falls Threads die Kontrolle über eine Untermenge der für die Fortsetzung erforderlichen Threads übernehmen können.

Die Rückgabewerte für `WaitForMultipleObjects` sind etwas komplexer als bei `WaitForSingleObject`:

- `WAIT_OBJECT_0` bis `WAIT_OBJECT_0 + ( Anzahl der Handles - 1)`, wenn der Handle signalisiert ist. Der Rückgabewert liefert den Index des niedrigsten nummerierten Handles, der signalisiert ist.
- `WAIT_TIMEOUT`, wenn der Handle nicht signalisiert ist und die Zeitbegrenzung abgelaufen ist.
- `WAIT_ABANDONED_0` bis `WAIT_ABANDONED_0 + (Anzahl der Handles - 1)` wird nur zurückgegeben, wenn ein Thread auf einen Mutex-Handle wartet.

Wenn Sie genau wissen wollen, welcher Handle signalisiert ist, können Sie das mit einem Codefragment wie dem folgenden ermitteln:

```
HANDLE rghEvents[63];
.
.
.
DWORD dw = WaitForMultipleObjects(63, rghEvents, FALSE, 0x10000);

if((dw >= WAIT_OBJECT_0)&&(dw <= WAIT_OBJECT_0 + 63))
{
    int ndx = dw - WAIT_OBJECT_0;
    HANDLE hSignaled = rghEvents[ndx];
    .
    // hSignaled verwenden
    .
}
```

Ereignis-Handles über Prozessgrenzen hinweg

Wie bereits erwähnt, kann man Ereignis-Handles zwischen Prozessen verwenden. Ein Ereignis-Handle lässt sich vererben, im Aufruf von `CreateProcess` an einen neuen Prozess übergeben oder mit `DuplicateHandle` duplizieren.

Wenn Sie ein Ereignisobjekt mit einem optionalen Namen erstellt haben, können Sie mit `CreateEvent` oder `OpenEvent` einen Handle auf dieses Ereignisobjekt von einem anderen Prozess erhalten. Nehmen wir an, man erstellt ein Objekt mit dem Namen »Shared«:

```
HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, "Shared");
```

Ein anderer Prozess kann einen Handle auf dasselbe Ereignis erhalten, wenn man `CreateEvent` mit den gleichen Parametern aufruft. Die Funktion `OpenEvent` kann auch von einem anderen Prozess verwendet werden:

```
HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, "Shared");
```

Auf der Begleit-CD finden Sie zwei kleine Konsolenanwendungen, die Beispiele für die Verwendung eines Ereignis-Handles über Prozessgrenzen hinweg liefern. Das Projekt `WaitEvent` erzeugt einen Ereignis-Handle und wartet auf seine Signalisierung. Das Projekt `SigEvent` öffnet eine Kopie dieses Ereignis-Handles und signalisiert sie. Dadurch kann `WaitEvent` die Ausführung fortsetzen.

Listing 3.7 zeigt den vollständigen Quellcode für das Programm `WaitEvent`.

Listing 3.7: Das Programm WaitEvent

```
#include <windows.h>
#include <tchar.h>
#ifndef UNICODE
```

```
    #include <stdio.h>
#endif
int _tmain()
{
    HANDLE hEvent = CreateEvent(NULL, FALSE,
                                FALSE, _T("WaitEvent"));

    _tprintf(_T("Auf SigEvent warten\n"));
    WaitForSingleObject(hEvent, INFINITE);
    _tprintf(_T("Event-Handle abgefangen. Fertig.\n"));
    CloseHandle(hEvent);
    return 0;
}
```

Den vollständigen Quellcode für `SigEvent` zeigt Listing 3.8.

Listing 3.8: Das Programm SigEvent

```
#include <windows.h>
#include <tchar.h>
#ifndef UNICODE
    #include <stdio.h>
#endif

int _tmain()
{
    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS,
                              FALSE, _T("WaitEvent"));
    _tprintf(_T("WaitEvent-Handle signalieren\n"));
    SetEvent(hEvent);
    return 0;
}
```

Kompilieren Sie beide Programme. Starten Sie `WaitEvent` zuerst. Im Konsolenfenster erscheint folgende Ausgabe:

```
Auf SigEvent warten
```

Wenn Sie das Programm `SigEvent` in einem weiteren Konsolenfenster starten, sieht die Anzeige wie folgt aus:

```
WaitEvent-Handle signalisieren
```

Das Programm `SigEvent` signalisiert den Ereignis-Handle und ermöglicht es somit dem Programm `WaitEvent`, die Ausführung abzuschließen. Das Konsolenfenster von `WaitEvent` zeigt folgendes an:

```
Event-Handle abgefangen. Fertig.
```

Abbildung 3.7 zeigt die beiden Konsolenfenster mit den gleichzeitig laufenden Programmen.

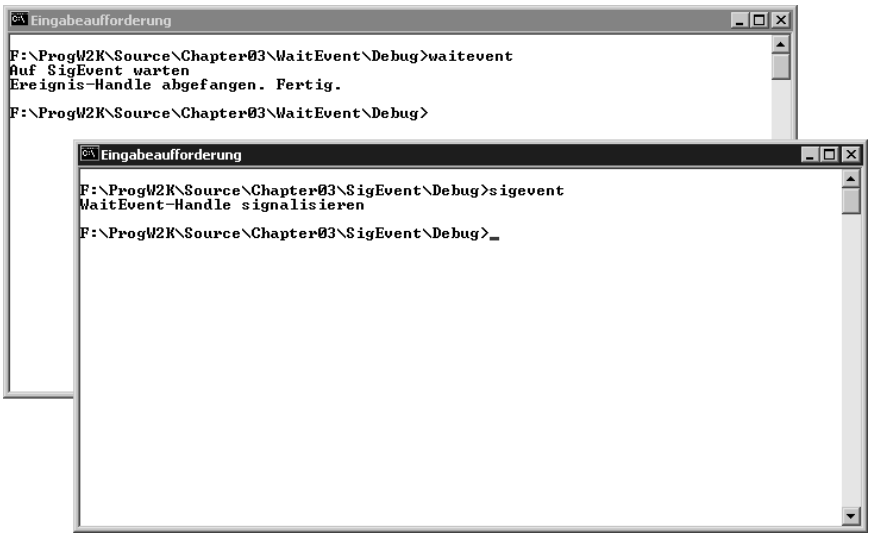


Bild 3.7: Ausführung der Programme WaitEvent und SigEvent

3.4.4 Mutexe und gegenseitiger Ausschluss

Ein Mutex ist ein Windows 2000-Objekt, mit dem sich ein gegenseitiger Ausschluss – Mutex steht für Mutual Exclusion – realisieren lässt. Wie bei kritischen Abschnitten kann nur ein Thread den Besitz eines Mutex-Objekts erlangen. Allerdings sind Mutexe im Gegensatz zu kritischen Abschnitten Windows 2000-Objekte, die das Betriebssystem verwaltet. Man kann sie auch benennen und zwischen Prozessen gemeinsam nutzen.

Um in den Besitz eines Mutex zu gelangen, muss ein Thread eine Warteoperation auf dem Mutex-Handle ausführen. Ein Mutex gilt als »signalisiert«, wenn es verfügbar ist, und als »nicht signalisiert«, wenn es in Gebrauch ist.

Ein traditionelles Beispiel zur Synchronisierung, das den gegenseitigen Ausschluss demonstriert, ist das Problem der »speisenden Philosophen«: Mehrere Philosophen sitzen an einem Tisch, auf dem zwischen jeweils benachbarten Philosophen nur eine Gabel liegt. Ein Philosoph muss zwei Gabeln in seinen Besitz bringen, bevor er essen kann, und er muss beide Gabeln ablegen, bevor er denken kann. Jeder Philosoph verbringt eine willkürliche Zeitdauer mit Denken und Essen.

Das Problem besteht darin, das Programm so zu entwerfen, dass die folgenden Bedingungen erfüllt sind:

- Kein Philosoph erleidet Hunger. Im übertragenen Sinne entzieht man ihm keine CPU-Zyklen (Nahrung).
- Kein Philosoph muss fortwährend essen, ohne denken zu dürfen.
- Vor allem aber ist darauf zu achten, dass keine Deadlock-Situation entsteht.

Der gegenseitige Ausschluss im Problem der speisenden Philosophen findet dann statt, wenn zwei Philosophen auf dieselbe Gabel zugreifen wollen. Durch gegenseitigen Ausschluss kann man sicherstellen, dass nur einer der Philosophen eine freie Gabel aufnehmen kann.

Wenn man einen Algorithmus für die Auswahl einer Gabel entwirft, muss man darauf achten, dass keine Deadlocks entstehen. Wenn die Philosophen eine Gabel aufnehmen können, aber darauf warten müssen, dass die andere Gabel frei wird, hat am Ende jeder Philosoph nur eine Gabel in der Hand – eine perfekte Deadlock-Situation.

Eine Lösung für das Problem der speisenden Philosophen finden Sie auf der Begleit-CD. Das Projekt `Philo` ist eine Konsolenanwendung, die die von den Philosophen benutzten Gabeln mit Mutex-Objekten darstellt. Will ein Philosoph essen, ruft das Programm `WaitForMultipleObjects` auf, um auf beide Mutex-Handles gleichzeitig zu warten. Da kein Mutex erworben wird, solange nicht beide bereit sind, können keine Deadlock-Situationen eintreten.

Listing 3.9 zeigt den Quellcode, der der Lösung der Begleit-CD entspricht. Diese Version des Quellcodes modelliert 63 Philosophen. Die Anzahl der Philosophen können Sie über den Wert von `nMaxPhil` ändern.

Listing 3.9: »Das Problem der speisenden Philosophen« mit Synchronisierungsobjekten von Windows 2000

```
#include <windows.h>
#include <tchar.h>
#ifndef UNICODE
    #include <stdio.h>
#endif

long  g_fDone = FALSE;
typedef struct tagCPhilosopher
{
    int     m_nID;
    HANDLE  m_hSticks[2];
    int     m_nMeals;
}CPhilosopher;

#define nMaxPhil 63
```

```
void SayEat(CPhilosopher* pPhilo)
{
    _tprintf(_T("Philosoph %d isst eine Zeit lang\n"), pPhilo->m_nID);
    pPhilo->m_nMeals++;
}

void SayThink(int nPhilo)
{
    _tprintf(_T("Philosoph %d denkt eine Zeit lang\n"), nPhilo);
}

void SayDone(CPhilosopher* pPhilo)
{
    _tprintf(_T("Philosoph %d hatte %d Mahlzeiten\n"),
             pPhilo->m_nID,
             pPhilo->m_nMeals);
}

long WINAPI WaitToEat(long lParam)
{
    CPhilosopher* pPhilo = (CPhilosopher*)lParam;
    DWORD dwEatTime = 1000 + GetCurrentThreadId();

    _tprintf(_T("Philosoph %d ist am Leben!\n"), pPhilo->m_nID);
    while(g_fDone == FALSE)
    {
        // Warten auf meine beiden Gabeln.
        WaitForMultipleObjects(2,
                               pPhilo->m_hSticks,
                               TRUE,
                               INFINITE);
        //
        // Warten erfüllt - Ich habe beide Gabeln
        SayEat(pPhilo);
        Sleep(dwEatTime);
        SayThink(pPhilo->m_nID);
        // Beide Gabeln ablegen.
        ReleaseMutex(pPhilo->m_hSticks[1]);
        ReleaseMutex(pPhilo->m_hSticks[0]);
    }
    return 0;
}

int _tmain()
{
    unsigned long  nThread;
    unsigned int   ndxStick;
```

```
    unsigned int   ndxPhilo;

    HANDLE          rghSticks[nMaxPhil];
    CPhilosopher    rgPhilos[nMaxPhil];

    for(ndxStick = 0; ndxStick < nMaxPhil; ndxStick++)
        rghSticks[ndxStick] = CreateMutex(NULL,FALSE,NULL);

    // Die Philosophen-Threads mit WaitToEat als
    // Thread-Funktion anstoßen.
    for(ndxPhilo = 0; ndxPhilo < nMaxPhil; ndxPhilo++)
    {
        HANDLE hThread;
        rgPhilos[ndxPhilo].m_nID = ndxPhilo;
        rgPhilos[ndxPhilo].m_nMeals = 0;
        rgPhilos[ndxPhilo].m_hSticks[0] = rghSticks[ndxPhilo];
        if(ndxPhilo < nMaxPhil-1)
          rgPhilos[ndxPhilo].m_hSticks[1] = rghSticks[ndxPhilo+1];
        else
          rgPhilos[ndxPhilo].m_hSticks[1] = rghSticks[0];
        hThread = CreateThread(NULL,
                               0,
                               (LPTHREAD_START_ROUTINE)WaitToEat,
                               (void*)&rgPhilos[ndxPhilo],
                               0,
                               &nThread);
        CloseHandle(hThread);
    }
    Sleep(100000);  // Ausführen für 100 Sekunden
    g_fDone = TRUE;
    Sleep(8000);    // 8 Sekunden auf Beendigung des Threads warten.

    for(ndxPhilo = 0; ndxPhilo < nMaxPhil; ndxPhilo++)
    {
        SayDone(&rgPhilos[ndxPhilo]);
        CloseHandle(rghSticks[ndxPhilo]);
    }
    return 0;
}
```

3.5 Semaphoren

Die *Semaphoren* gehören zu den ersten Synchronisierungsobjekten, die man in der Computerliteratur findet. Sie sind eine Erfindung von Edsger Dijkstra als Tool für das Multithreading, das Mitte der sechziger Jahre ein neues Feld darstellte.

Eine Semaphore ist wie ein Zähler, der als Wächter über einen Codeabschnitt oder eine Ressource agiert. Man bezeichnet Semaphoren manchmal auch als Dijkstra-Zähler.

Die Semaphore verwaltet einen internen Zähler, der bei Aktionen auf der Semaphore dekrementiert oder inkrementiert wird. Wenn der interne Zähler der Semaphore Null erreicht, muss jeder neue Thread, der den Zähler dekrementieren will, warten, bis ihn ein anderer Thread inkrementiert hat. Auf einer Semaphore lassen sich zwei Operationen ausführen:

- P oder auch DOWN: Kennzeichnet, dass eine Ressource nicht verfügbar ist. P steht für »proberen te verlagen«, ein niederländischer Begriff, den man etwa mit »Versuch zu vermindern« übersetzen kann.
- V oder auch UP: Kennzeichnet, dass eine Ressource verfügbar geworden ist. V steht für das niederländische Wort »verhogen«, zu deutsch »erhöhen«.

3.5.1 Semaphoren und Wartefunktionen

Bevor man eine von einer Semaphore überwachte Ressource verwendet, muss ein Thread immer eine Warteoperation auf dem Handle der Semaphore ausführen. Damit lässt sich der Thread blockieren, falls die Semaphore nicht signalisiert ist. Weiterhin ist es möglich, dass der interne Zähler der Semaphore dekrementiert wird, nachdem die Warteoperation des Threads abgeschlossen ist. Das ist die im letzten Abschnitt beschriebene »P«-Funktion oder der »Versuch zu vermindern«.

Wenn ein Thread die Benutzung einer kontrollierten Ressource beendet hat, gibt er die Semaphore über die Funktion `ReleaseSemaphore` frei. Diese Funktion erhöht den internen Zähler der Semaphore und ermöglicht einem anderen wartenden Thread, die Kontrolle der Semaphore zu übernehmen. Das ist die im letzten Abschnitt erwähnte »V«-Funktion (»erhöhen«).

Abbildung 3.8 zeigt, wie eine Semaphore initialisiert, verwendet und freigegeben wird.

Eine Semaphore kann man mit den weiter vorn in diesem Kapitel behandelten Wartefunktionen verwenden. Eine Semaphore gilt als signalisiert, wenn ihr Zähler größer als Null ist, und als nicht signalisiert, solange der Zähler auf Null steht.

Der interne Zähler der Semaphore sollte immer die Anzahl der für die Konsumtion verfügbaren Ressourcen widerspiegeln. Alle Threads, die von einer Semaphoren-gesteuerten Ressource Besitz ergreifen, müssen eine Wartefunktion auf dem Semaphoren-Objekt ausführen. Wenn der Semaphoren-Zähler gleich Null ist, sind keine Ressourcen verfügbar. Damit blockiert der Thread bis entweder seine Zeitbegrenzung abgelaufen ist oder eine Ressource verfügbar wird.

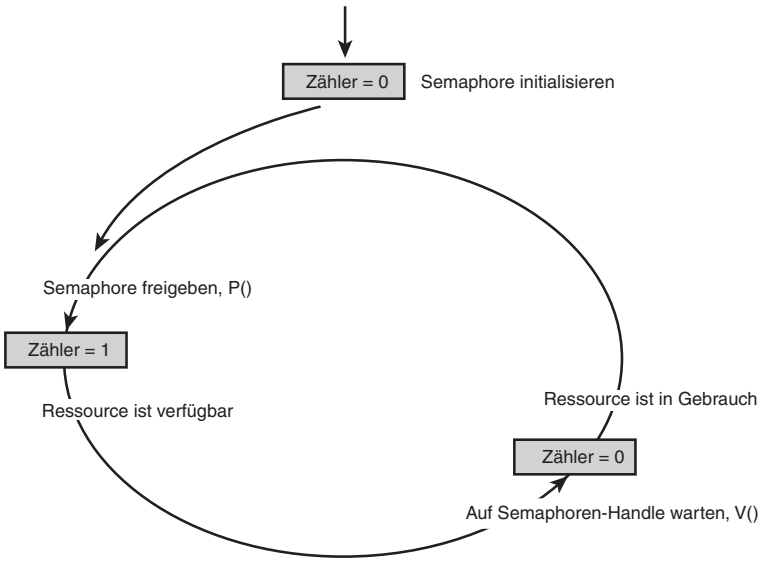


Bild 3.8: Ressourcen mit einer Semaphore steuern

3.5.2 Anwendungen für Semaphoren

Eine Semaphore ist flexibler als ein kritischer Abschnitt, da sich mit ihr eine Ressourcenmenge überwachen lässt, statt nur einem einzelnen Thread den Zugriff auf einen bestimmten Teil des Codes zu erlauben. Eine typische Aufgabenstellung in Multithreading-Umgebungen ist zum Beispiel die Zuweisung von knappen Ressourcen an eine Gruppe von Konsumenten, wie es Abbildung 3.9 zeigt.

In Abbildung 3.9 steuert eine Ressource eine Gruppe von Friseuren. Die Semaphore erhält anfangs den Maximalwert 3 und zeigt damit an, dass drei Friseurressourcen verfügbar sind. Wenn ein Kunde den Friseurladen betritt, wird die Semaphore dekrementiert (siehe Abbildung 3.10).

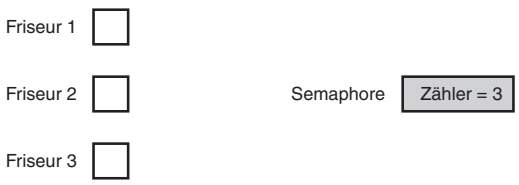


Bild 3.9: Drei freie Friseure durch eine Semaphore gesteuert

Mit jedem neuen Kunden verringert sich der interne Zähler der Semaphore um 1, erreicht schließlich den Wert Null und zeigt damit an, dass keine Friseure mehr frei sind (siehe Abbildung 3.11).

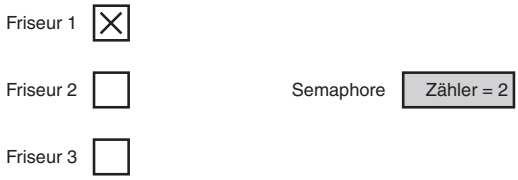


Bild 3.10: Die Konsumtion einer Friseurressource dekrementiert den Semaphorenzähler

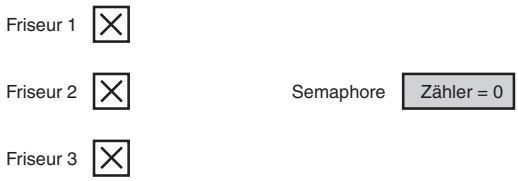


Bild 3.11: Zugriff auf die Friseure mit einer Semaphore blockieren

Hat ein Friseur einen Kunden bedient, gibt dieser Kunde die Semaphore frei. Der interne Zähler erhöht sich damit um 1 und einer der freien Friseure kann einen weiteren Kunden bedienen (siehe Abbildung 3.12).

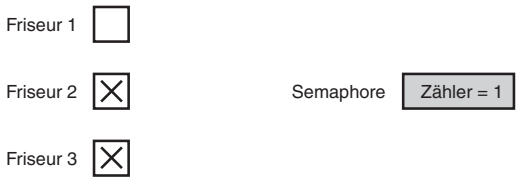


Bild 3.12: Durch die Freigabe einer Semaphore kann ein neuer Kunde bedient werden

Das sind die grundlegenden Schritte im Lebenszyklus einer Produzent-/Konsument-Anwendung. Die Friseure »produzieren« Haarschnitte, und die Kunden »konsumieren« sie der Reihe nach. Später zeigt dieses Kapitel eine dialogbasierte Anwendung, die Semaphore in einer ähnlichen Art einsetzt.

3.5.3 Semaphoren erstellen

Eine Semaphore erzeugt man mit der Funktion `CreateSemaphore`:

```
HANDLE hsemBarbers = CreateSemaphore(NULL,
                                     0,
                                     5,
                                     NULL);
if(hsemBarbers == NULL)
```

```
{
    // Fehler behandeln
}
```

Die vier Parameter der Funktion `CreateSemaphore` haben folgende Bedeutung:

- `lpSemaphoreAttributes`: Ein Zeiger auf eine `SECURITY_ATTRIBUTES`-Struktur. Auf diese Struktur geht das Kapitel später näher ein.
- `lInitialCount`: Ein Anfangswert für den Zähler der Semaphore. Dieser Wert muss größer oder gleich Null sein.
- `lMaximumCount`: Ein Maximalwert für den Zähler der Semaphore. Dieser Wert muss mindestens 1 bzw. mindestens so groß wie der Anfangswert des Zählers sein.
- `lpName`: Ein optionaler Name für die Semaphore. Denken Sie daran, dass Namen nicht doppelt vorkommen dürfen, weil alle Ereignis-, Mutex- und Dateizuordnungsobjekte denselben Namensbereich verwenden. In den meisten Fällen ist dieser Wert `NULL`.

3.5.4 Semaphoren-gesteuerte Ressourcen zurückgeben

Wenn eine Semaphoren-gesteuerte Ressource verfügbar wird, ist der Ressourcenzähler mit der Funktion `ReleaseSemaphore` zu inkrementieren:

```
BOOL fReleased = ReleaseSemaphore(hSemBarbers,
                                  1,
                                  NULL);
if(fReleased == FALSE)
{
    // Fehler behandeln
}
```

Die drei Parameter der Funktion `ReleaseSemaphore` haben folgende Bedeutung:

- `hSemaphore`: Ein Handle auf die freizugebende Semaphore.
- `lReleaseCount`: Die Anzahl der Schritte, um die der Semaphoren-Zähler zu inkrementieren ist.
- `lpPreviousCount`: Ein optionaler Zeiger auf eine Variable vom Typ `long`, die den vorherigen Zählerstand aufnimmt. Ist der Zählerstand nicht erforderlich, können Sie für diesen Parameter `NULL` übergeben.

3.6 Semaphoren und kritische Abschnitte in einer Anwendung

Auf der Begleit-CD ist das Programm `SkiLift` als Beispiel für den Einsatz von Semaphoren in einer Windows 2000-Anwendung enthalten. `SkiLift` ist eine dialogfeldbasierende Anwendung, die den Status einer Gruppe von Skifahrern anzeigt, die einen Skilift benutzen.

Eine Semaphore steuert den Zugriff auf den Skilift, wobei maximal vier Skifahrer gleichzeitig den Lift benutzen können. Nachdem sie den Gipfel des Berges erreicht haben, gelangt jeder Skifahrer nach einer zufälligen Zeit zum Ausgangspunkt des Skilifts am Fuß des Berges zurück. Abbildung 3.13 zeigt die laufende `SkiLift`-Anwendung.

Bild 3.13: Die Anwendung SkiLift steuert den Zugriff auf den Lift mit Semaphoren

Das Hauptdialogfeld für das Projekt `SkiLift` enthält zwei Schaltflächen und 20 Kontrollkästchen (siehe Abbildung 3.14).

Tabelle 3.1 zeigt die Eigenschaften für die Kontrollkästchen und die Schaltflächen. Alle hier nicht aufgeführten Eigenschaften behalten ihre Standardwerte. Die Kontrollkästchen sind in der richtigen Reihenfolge zuzuweisen, d.h. `IDC_CHECK1` bis `IDC_CHECK8` in der ersten Spalte, `IDC_CHECK9` bis `IDC_CHECK12` in der mittleren und `IDC_CHECK13` bis `IDC_CHECK20` in der letzten Spalte des Dialogfelds.

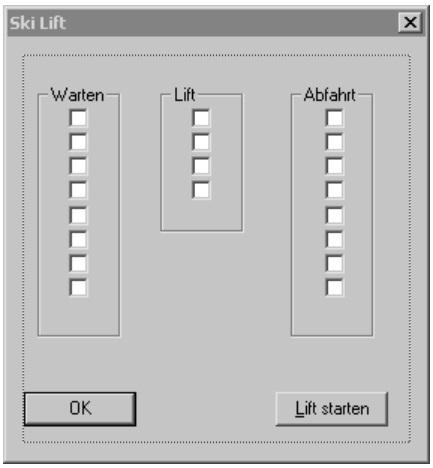


Bild 3.14: Steuerelemente im Hauptdialogfeld des Projekts SkiLift

Steuerelement	Ressourcen-ID	Beschriftung
Schaltfläche START	IDC_STARTSTOP	&Lift starten
statisches Steuerelement Status	IDC_STATUS	
Kontrollkästchen	IDC_CHECK1 bis IDC_CHECK20	(leer)

Tab. 3.1: Eigenschaftswerte für Steuerelemente im Dialogfeld von SkiLift

Listing 3.10 gibt die Header-Datei `SkiLift.h` für das Projekt `SkiLift` wieder.

Listing 3.10: Die Header-Datei SkiLift.h

```
struct CSkier
{
    int  m_nID;
    HWND m_hWnd;
};

BOOL InitApplication(HINSTANCE hInst);
BOOL InitInstance(HINSTANCE hInst, int nCmdShow);
LRESULT CALLBACK WndProc(HWND   hWnd,
                         UINT   uMsg,
                         WPARAM wParam,
                         LPARAM lParam);

BOOL CALLBACK DialogProc(HWND   hWndDlg,
                         UINT   uMsg,
                         WPARAM wParam,
```

```
                        LPARAM lParam);
void  OnStart();
DWORD WINAPI SkiThread(PVOID pv);
DWORD WINAPI Cleaner(PVOID pv);
void  WaitForLift(CSkier* pSkier);
void  GetOnLift(CSkier* pSkier);
void  SkiAWhile(CSkier* pSkier);
void  SetCheckBoxArray(UINT low, UINT high, BOOL fSet);
```

Die Hauptquelldatei für das Projekt `SkiLift` ist `SkiLift.cpp`. Listing 3.11 gibt die wichtigsten Teile dieser Quelldatei wieder.

Listing 3.11: Die Multithreading-Windows-Anwendung SkiLift

```
BOOL       g_fLiftStopped = FALSE;
CSkier     g_rgSkier[8];
HANDLE     g_hChairLift = NULL;
HINSTANCE  g_hInstance = NULL;
HWND       g_hWnd = NULL;
LPCTSTR    g_szAppName = _T("SkiLift");

CRITICAL_SECTION g_cs;
    .
    .
    .
    .
void OnStart()
{
    HWND hWndStart = GetDlgItem(g_hWnd, IDC_STARTSTOP);
    if(g_fLiftStopped)
    {
        // Liftbetrieb eröffnen - Initialisierungen
        // durchführen
        g_fLiftStopped = FALSE;
        const int nMaxOnLift = 4;
        const int nMaxSkiers = 8;
        srand( (unsigned)time( NULL ) );
        SetWindowText(hWndStart, _T("$Lift stoppen"));
        // Kritischen Abschnitt für die Benutzeroberfläche
        // initialisieren und eine Semaphore mit vier Slots für
        // den Skilift erzeugen.
        InitializeCriticalSection(&g_cs);
        g_hChairLift = CreateSemaphore(NULL,
                                       0,
                                       nMaxOnLift,
                                       _T("ChairLift"));
        // 8 Skifahrer erzeugen, jeder mit eigenem Thread.
        // Jeder der 8 Fahrer kennt seine Skinummer und sein
```

```
            // Fenster-Handle.
            for(int n = 0; n < nMaxSkiers; n++)
            {
                g_rgSkier[n].m_nID = n;
                g_rgSkier[n].m_hWnd = g_hWnd;
                DWORD dwThreadId;
                HANDLE hThread;
                hThread = CreateThread(NULL,
                                       0,
                                       (LPTHREAD_START_ROUTINE)SkiThread,
                                       (void*)&g_rgSkier[n],
                                       0,
                                       &dwThreadId);
                CloseHandle(hThread);
            }
            // Nach Erzeugen aller Skifahrer, alle Ressourcen der
            // Semaphore freigeben.
            ReleaseSemaphore(g_hChairLift, nMaxOnLift, NULL);
        }
        else
        {
            // Lift anhalten. Einen "Aufräumthread" erstellen, der
            // alle Synchronisierungsressourcen bereinigt. Die Start/Stop-
            // Schaltfläche wird deaktiviert, bis der Bereinigungsvorgang
            // abgeschlossen ist.
            DWORD dwThreadId;
            g_fLiftStopped = TRUE;
            EnableWindow(hWndStart, FALSE);
            SetWindowText(hWndStart, _T("&Lift starten"));
            HANDLE hCleanThread;
            hCleanThread = CreateThread(NULL,
                                        0,
                                        (LPTHREAD_START_ROUTINE)Cleaner,
                                        (void*)g_hWnd,
                                        0,
                                        &dwThreadId);
            CloseHandle(hCleanThread);
        }
    }


    DWORD WINAPI SkiThread(PVOID pv)
    {
        CSkier* pSkier = (CSkier*)pv;
        while(g_fLiftStopped == FALSE)
        {
            WaitForLift(pSkier);
```

```
        GetOnLift(pSkier);
        SkiAWhile(pSkier);
    }
    return NO_ERROR;
}

DWORD WINAPI Cleaner(PVOID pv)
{
    Sleep(10000);
    CloseHandle(g_hChairLift);
    // Schaltfläche Start wiederherstellen und Statusfenster
    // bereinigen.
    HWND hWndStart = GetDlgItem(g_hWnd, IDC_STARTSTOP);
    EnableWindow(hWndStart, TRUE);
    HWND hWndLabel = GetDlgItem(g_hWnd, IDC_STATUS);
    SetWindowText(hWndLabel, _T(""));
    DeleteCriticalSection(&g_cs);
    return NO_ERROR;
}

void WaitForLift(CSkier* pSkier)
{
    TCHAR szMsg[256];
    wsprintf(szMsg,
             _T("Skifahrer %d stellt sich am Lift an"),
             pSkier->m_nID);
    // Statusfenster mit Infos über diesen Fahrer aktualisieren und
    // ein Kontrollhäkchen in die Spalte Warten setzen.
    EnterCriticalSection(&g_cs);
    SetCheckBoxArray(IDC_CHECK1, IDC_CHECK8, TRUE);
    SetDlgItemText(g_hWnd, IDC_STATUS, szMsg);
    Sleep(100);
    LeaveCriticalSection(&g_cs);
    // Auf Liftsessel-Semaphore warten. Ist eine Ressource noch nicht
    // verfügbar, blockiert hier der Thread, bis eine Ressource frei
ist.
    // Wenn ein Sessel frei ist, wird er diesem Thread zugewiesen,
    // und die Ausführung setzt in der Funktion GetOnLift fort.
    WaitForSingleObject(g_hChairLift, INFINITE);
}

void GetOnLift(CSkier* pSkier)
{
    TCHAR szMsg[256];
    wsprintf(szMsg,
             _T("Skifahrer %d besteigt den Lift"),
             pSkier->m_nID);
```

```
    // Statusfenster mit Infos über diesen Fahrer aktualisieren,
    // Kontrollhäkchen aus Warten-Spalte entfernen und Kontroll-
    // häkchen in die Spalte Lift setzen.
    EnterCriticalSection(&g_cs);
    SetCheckBoxArray(IDC_CHECK1, IDC_CHECK8,  FALSE);
    SetCheckBoxArray(IDC_CHECK9, IDC_CHECK12, TRUE);
    SetDlgItemText(g_hWnd, IDC_STATUS, szMsg);
    Sleep(100);
    LeaveCriticalSection(&g_cs);
    // Kurze (feste) Zeitspanne im Liftsessel verbringen, dann eine
    // Liftsessel-Ressource zurückgeben. Ausführung setzt in der Funkti-
on
    // SkiAWhile fort.
    Sleep(2000);

    EnterCriticalSection(&g_cs);
    SetCheckBoxArray(IDC_CHECK9, IDC_CHECK12,  FALSE);
    LeaveCriticalSection(&g_cs);
    ReleaseSemaphore(g_hChairLift, 1, NULL);
}

void SkiAWhile(CSkier* pSkier)
{
    TCHAR szMsg[256];
    wsprintf(szMsg,
             _T("Skifahrer %d beginnt Abfahrt"),
             pSkier->m_nID);
    // Einen Skiahrer von den Lift-Kontrollkästchen zu den
    // Abfahrt-Kontrollkästchen verschieben. Nach Aktualisierung
    // der Statusanzeige das Statusfenster für 100 Millisekunden
    // halten, damit es der Benutzer lesen kann.
    EnterCriticalSection(&g_cs);
    SetCheckBoxArray(IDC_CHECK13, IDC_CHECK20, TRUE);
    SetDlgItemText(g_hWnd, IDC_STATUS, szMsg);
    Sleep(100);
    LeaveCriticalSection(&g_cs);
    // Abfahrt simulieren durch inaktiven Zustand für eine zufällige
    // Zeitspanne zwischen 0 und 1999 Millisekunden.
    Sleep( rand() % 2000 );
    // Abfahrt beendet, eines der Abfahrt-Kontrollkästchen
    // ausschalten.
    EnterCriticalSection(&g_cs);
    SetCheckBoxArray(IDC_CHECK13, IDC_CHECK20, FALSE);
    LeaveCriticalSection(&g_cs);
}

void SetCheckBoxArray(UINT low, UINT high, BOOL fSet)
```

```
{
    UINT nResIndex;
    if(fSet)
    {
        // Kontrollhäkchen setzen, ein deaktiviertes Kontrollkästchen
        // suchen.
        for(nResIndex = low; nResIndex <= high; nResIndex++)
            if(!IsDlgButtonChecked(g_hWnd, nResIndex))
                break;
    }
    else
    {
        // Ein Kontrollhäkchen entfernen
        for(nResIndex = high; nResIndex >= low; nResIndexó)
            if(IsDlgButtonChecked(g_hWnd, nResIndex))
                break;
    }
    CheckDlgButton(g_hWnd, nResIndex, fSet);
}
```

Listing 3.10 definiert eine Struktur namens `CSkier`, die jeden Skifahrer im Projekt `SkiLift` repräsentiert. In Listing 3.11 verwendet das Projekt `SkiLift` ein Array mit acht `CSkier`-Strukturen. Eine `CRITICAL_SECTION`-Variable synchronisiert die Steuerung der Dialogfeldelemente. Ein Semaphoren-Handle steuert den Zugriff auf den Sessellift, so dass nur vier Skifahrer den Skilift gleichzeitig benutzen können. Schließlich zeigt ein Flag an, wann der Skilift in Betrieb ist. Beim Stoppen des Skilifts wird `g_fLiftStopped` auf `TRUE` gesetzt, und alle laufenden Threads stellen die Benutzung des Skilifts ein.

Insgesamt steuern sieben Funktionen die Multithreading-Aspekte der Benutzeroberfläche. Die Funktion `SkiThread` dient als »Thread-Start« und wird von Threads, die Skifahrer im Projekt `SkiLift` repräsentieren, aufgerufen. Diese Threads rufen dann die Funktionen `WaitForLift`, `GetOnLift` und `SkiAWhile` in einer Schleife auf. Die Funktion `SetCheckBoxArray` aktiviert oder deaktiviert die Kontrollkästchen, die den aktuellen Status des Programms anzeigen.

Die Funktion `Cleaner` führt Aufräumungsarbeiten aus, bevor die Anwendung terminiert.

Kompilieren und starten Sie die Anwendung `SkiLift`. Klicken Sie auf die Schaltfläche LIFT STARTEN. In der linken Spalte erscheinen acht Kontrollhäkchen für acht Skifahrer, die in einer Schlange stehen und auf einen Skilift warten. Vier der Kontrollhäkchen wandern sofort zur mittleren Spalte der Skifahrer, die den Skilift besteigen. Wenn diese Skifahrer die Bergstation erreichen, betreten vier weitere Skifahrer den Skilift.

Jeder Skifahrer-Thread läuft eine zufällige Zeit, bis er sich erneut an der Talstation anstellt. Nach einigen Durchläufen ist die Last ausgeglichen. Allerdings erweist sich der Skilift immer als Nadelöhr- genau wie im richtigen Leben.

3.7 Auftragsobjekte

Windows 2000 führt ein neues Objekt für die Prozeßsteuerung ein: das *Auftragsobjekt*. Ein Auftragsobjekt ist eine Sammlung von Prozessen, die sich als Einheit verwalten lassen. Wie ein Beispiel zeigt, lassen sich Quoten für die gesamte Ausführungszeit und Prozessorzeit sowie Optionen zur Steuerungsplanung für das Auftragsobjekt festlegen. Ebenso kann man Prozessoraffinität einstellen, Interaktion von Prozessen im Auftrag mit der Zwischenablage steuern und Sicherheitsparameter für das Auftragsobjekt definieren.

Die Funktion `CreateJobObject` erzeugt ein Auftragsobjekt:

```
HANDLE hJobs = CreateJobObject(NULL, _T("MyJobObject"));
```

Die Funktion `CreateJobObject` hat zwei Parameter:

- `lpJobAttributes`: Ein Zeiger auf eine `SECURITY_ATTRIBUTES`-Struktur, die `NULL` sein kann, wenn untergeordnete Prozesse das Auftragsobjekt nicht erben sollen.
- `lpName`: Der Name des Auftragsobjekts. Dieser Name darf keinem anderen Kernelobjekt wie Ereignissen, Mutexen, Semaphoren, Dateizuordnungen und Waitable Timern entsprechen.

Nachdem man ein Auftragsobjekt erstellt hat, können andere Prozesse einen Handle auf das Auftragsobjekt über die Funktion `OpenJobObject` erhalten:

```
HANDLE hJobs = OpenJobObject(MAXIMUM_ALLOWED, FALSE, _T("MyJobObject"));
```

Die Funktion `OpenJobObject` hat drei Parameter:

- `dwDesiredAccess`: Der gewünschte Zugriffsmodus auf das Auftragsobjekt. Im Beispiel fordert `MAXIMUM_ACCESS` uneingeschränkte Zugriffsrechte an. Andere mögliche Werte folgen weiter unten.
- `bInheritHandle`: Zeigt an, ob der zurückgegebene Handle des Auftragsobjekts an neu erzeugte untergeordnete Prozesse vererbbar sein soll (`TRUE`) oder nicht (`FALSE`).
- `lpName`: Der Name des zu öffnenden Auftragsobjekts.

Die möglichen Zugriffsebenen beim Öffnen eines Auftragsobjekts sind:

- `MAXIMUM_ALLOWED`: Fordert die vollen Zugriffsrechte an, die für den Aufrufer gültig sind.
- `JOB_OBJECT_ASSIGN_PROCESS`: Fordert Rechte an, um dem Aufrufer zu erlauben, Prozesse an das Auftragsobjekt zuzuweisen.
- `JOB_OBJECT_SET_ATTRIBUTES`: Fordert Rechte an, um dem Aufrufer zu erlauben, Attribute für das Auftragsobjekt zu setzen.

- JOB_OBJECT_QUERY: Fordert für den Aufrufer die Rechte an, die Attribute des Auftragsobjekts abzurufen.
- JOB_OBJECT_TERMINATE: Fordert Rechte an, die dem Aufrufer erlauben, das Auftragsobjekt zu terminieren.
- JOB_OBJECT_SET_SECURITY_ATTRIBUTES: Fordert Rechte an, die dem Aufrufer erlauben, Sicherheitsattribute für das Auftragsobjekt zuzuweisen.
- JOB_OBJECT_ALL_ACCESS: Fordert volle Zugriffsrechte auf das Auftragsobjekt an.

Nachdem ein Prozess erstellt ist, weist man ihn mit der Funktion AssignProcessToJobObject einem Auftragsobjekt zu:

```
BOOL fAssigned = AssignProcessToJobObject(hJob, hProcess);
```

Die Funktion AssignProcessToJobObject hat zwei Parameter:

- hJob: Ein Handle auf das Auftragsobjekt, dem der Prozess zuzuordnen ist.
- hProcess: Ein Handle auf den Prozess, der dem Auftragsobjekt zuzuordnen ist.

3.7.1 Attribute von Auftragsobjekten steuern

Jedes Auftragsobjekt verfügt über eine große Zahl von Attributen, die man steuern kann. Alle Attribute für Auftragsobjekte setzt man über die Funktion SetInformationJobObject:

```
BOOL fSet = SetInformationJobObject(g_hJobs,
                                    JobObjectBasicLimitInformation,
                                    &jbli,
                                    sizeof(jbli));
```

Die Funktion SetInformationJobObject hat vier Parameter:

- hJob: Ein Handle auf das Auftragsobjekt.
- JobObjectInformationClass: Ein Aufzählungswert, der angibt, welche der fünf unterschiedlichen Sätze von Attributen des Auftragsobjekts zu ändern sind (Optionen für diesen Wert folgen weiter unten).
- lpJobObjectInformation: Ein Zeiger auf eine Struktur mit Attributinformationen für das Auftragsobjekt. Diese Struktur hat einen von sieben Typen, auf die das Kapitel weiter unten eingeht.
- cbJobObjectInformationLength: Die Größe der als dritter Parameter übergebenen Struktur.

Es gibt fünf verschiedene Sätze von Informationen über Auftragsobjekte. Jeder Satz wird durch einen spezifischen Aufzählungswert dargestellt, den man als

zweiten Parameter an `SetInformationJobObject` übergibt. Zu den fünf Informationstypen gehört jeweils eine spezifische Struktur, die auszufüllen und als dritter Parameter zu übergeben ist:

- `JobObjectAssociateCompletionPortInformation`: Attributinformationen über die Verbindung des Auftragsobjekts mit einem E-/A-Abschlussport werden in einer `JOBOBJECT_ASSOCIATE_COMPLETION_PORT`-Struktur übergeben.
- `JobObjectBasicLimitInformation`: Attributinformationen über Quoten, Prozessoraffinität und Zeitplanung werden in einer `JOBOBJECT_BASIC_LIMIT_INFORMATION`-Struktur übergeben.
- `JobObjectBasicUIRestrictions`: Attributinformationen über die Verwendung der Zwischenablage, Zugriff auf Benutzer-Handles und die Fähigkeit, bestimmte Windows-API-Funktionen aufzurufen, werden in einer `JOBOBJECT_BASIC_UI_RESTRICTIONS` übergeben.
- `JobObjectEndOfJobTimeInformation`: Attributinformationen über die durchzuführenden Aktionen bei Zeitüberschreitung des Auftragsobjekts werden in einer `JOBOBJECT_END_OF_JOB_TIME_INFORMATION`-Struktur übergeben.
- `JobObjectSecurityLimitInformation`: Attributinformationen über Sicherheitseinschränkungen werden in einer `JOBOBJECT_SECURITY_LIMIT_INFORMATION`-Struktur übergeben.

3.7.2 Beispiel für ein Auftragsobjekt

Auf der Begleit-CD ist das Projekt `JobObj` als Beispiel für den Einsatz von Auftragsobjekten in Windows 2000 enthalten. `JobObj` ist eine Konsolenanwendung, die ein Auftragsobjekt mit den Spielen Solitär und FreeCell erzeugt. `JobObj` empfängt Rückmeldungen zum Status des Auftragsobjekts über einen E-/A-Abschlussport.

Listing 3.12 zeigt die Hauptquelldatei des Projekt `JobObj`.

Listing 3.12: Main.cpp – Die Hauptquelldatei für das Projekt JobObj

```
#define _WIN32_WINNT 0x500
#include <windows.h>
#include <tchar.h>
#ifndef UNICODE
    #include <stdio.h>
#endif

HANDLE CreateSolitaireProcess(void);
HANDLE CreateFreecellProcess(void);
void AssociateJobAndCompletionPort(void);
void SetJobLimits(void);
```

```
HANDLE      g_hCompletionPort;
OVERLAPPED  g_ov;
HANDLE      g_hJobs = NULL;

int _tmain()
{
    g_hJobs = CreateJobObject(NULL, _T("Projekt JobObj"));
    AssociateJobAndCompletionPort();
    SetJobLimits();

    HANDLE hSolitaire = CreateSolitaireProcess();
    HANDLE hFreecell = CreateFreecellProcess();

    DWORD dwCurrentProcesses = 0;
    AssignProcessToJobObject(g_hJobs, hSolitaire);
    AssignProcessToJobObject(g_hJobs, hFreecell);

    /* Nachrichten für Abschlussereignisse anzeigen. */
    bool done = false;
    while(!done)
    {
        DWORD dwMsgId, dwKey;
        LPOVERLAPPED pov;
        LPCTSTR psz;
        GetQueuedCompletionStatus(g_hCompletionPort,
                                  &dwMsgId,
                                  &dwKey,
                                  &pov,
                                  INFINITE);
        switch(dwMsgId)
        {
            case JOB_OBJECT_MSG_END_OF_JOB_TIME:
                psz = _T("JOB_OBJECT_MSG_END_OF_JOB_TIME");
            break;
            case JOB_OBJECT_MSG_END_OF_PROCESS_TIME:
                psz = _T("JOB_OBJECT_MSG_END_OF_PROCESS_TIME");
            break;
            case JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT:
                psz = _T("JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT");
            break;
            case JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO:
                psz = _T("JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO");
                done = true;
            break;
            case JOB_OBJECT_MSG_NEW_PROCESS:
                psz = _T("JOB_OBJECT_MSG_NEW_PROCESS");
                dwCurrentProcesses++;
```

```
            break;
            case JOB_OBJECT_MSG_EXIT_PROCESS:
                psz = _T("JOB_OBJECT_MSG_EXIT_PROCESS");
                dwCurrentProcessesó;
                if(!dwCurrentProcesses) done = true;
            break;
        }
        _tprintf(_T("Auftragsfertigstellung - %s\n"), psz);
    }
    CloseHandle(g_hJobs);
    CloseHandle(g_hCompletionPort);
    MessageBox(NULL, _T("Beendet"), _T("JobObj"), MB_OK);
    return 0;
}

/*
 * Erzeugt einen Prozess für das Spiel Solitär und gibt den
 * Handle des Prozesses zurück.
 */
HANDLE CreateSolitaireProcess(void)
{
    TCHAR   lpszSolitairePath[_MAX_PATH];
    PROCESS_INFORMATION pi;
    STARTUPINFO         si;

    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);

    GetSystemDirectory(lpszSolitairePath, _MAX_PATH);
    lstrcat(lpszSolitairePath, _T("\\SOL.EXE"));

    CreateProcess(lpszSolitairePath,
                  NULL,
                  NULL,
                  NULL,
                  FALSE,
                  0,
                  NULL,
                  NULL,
                  &si,
                  &pi);
    return pi.hProcess;
}


/*
 * Erzeugt einen Prozess für das Spiel FreeCell und gibt
```

```
 * den Handle des Prozesses zurück.
 */
HANDLE CreateFreecellProcess(void)
{
    TCHAR   lpszFreecellPath[_MAX_PATH];
    PROCESS_INFORMATION pi;
    STARTUPINFO         si;

    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);

    GetSystemDirectory(lpszFreecellPath, _MAX_PATH);
    lstrcat(lpszFreecellPath, _T("\\FREECELL.EXE"));

    CreateProcess(lpszFreecellPath,
                  NULL,
                  NULL,
                  NULL,
                  FALSE,
                  0,
                  NULL,
                  NULL,
                  &si,
                  &pi);
    return pi.hProcess;
}

/*
 * Den Handle des Auftragsobjekts mit einem E-/A-Abschlussport
 * verbinden und eine "End of Job"-Aktion setzen, um eine Benachrichti-
gung
 * an den Abschlussport zu senden.
 */
void AssociateJobAndCompletionPort(void)
{
    JOBOBJECT_ASSOCIATE_COMPLETION_PORT jacp;
    g_hCompletionPort= CreateIoCompletionPort(INVALID_HANDLE_VALUE,
                                              NULL,
                                              0x42,
                                              0);
    jacp.CompletionKey = NULL;
    jacp.CompletionPort = g_hCompletionPort;

    ZeroMemory(&g_ov, sizeof(OVERLAPPED));
    SetInformationJobObject(g_hJobs,
                        JobObjectAssociateCompletionPortInformation,
                        &jacp,
                        sizeof(jacp));
```

```
    JOBOBJECT_END_OF_JOB_TIME_INFORMATION  jeot;
    jeot.EndOfJobTimeAction = JOB_OBJECT_POST_AT_END_OF_JOB;
    SetInformationJobObject(g_hJobs,
                            JobObjectEndOfJobTimeInformation,
                            &jeot,
                            sizeof(jeot));
}

/*
 * Eine (sehr kleine) Zeitbegrenzung für die Prozesse im Auftragsobjekt
 * setzen. Im Beispiel sind das 500000 Nanosekunden, was etwa zwei
 * Minuten in einem Hundeleben entspricht.
 */
void SetJobLimits(void)
{
    JOBOBJECT_BASIC_LIMIT_INFORMATION jbli;
    ZeroMemory(&jbli, sizeof(jbli));

    jbli.PerJobUserTimeLimit.QuadPart = 500000;
    jbli.LimitFlags = JOB_OBJECT_LIMIT_JOB_TIME;

    BOOL f0 = SetInformationJobObject(g_hJobs,
                            JobObjectBasicLimitInformation,
                            &jbli,
                            sizeof(jbli));
}
```

Kompilieren Sie die Anwendung `JobObj` und starten Sie sie von der Eingabeaufforderung. Das Auftragsobjekt startet die Anwendungen Solitär und FreeCell. Beim Starten der Prozesse erscheinen Informationen im Konsolenfenster. Weitere Meldungen erhalten Sie, wenn die Zeitbegrenzung des Auftragsobjekts abläuft und wenn die Prozesse Solitär oder FreeCell terminieren.

3.8 Zusammenfassung

Dieser Abschnitt hat sich mit Threads, Synchronisierung und Sicherheitsobjekten befaßt. Außerdem haben Sie Methoden kennengelernt, um Threads und Prozesse zu erstellen und zu verwalten, und es wurde auf das Für und Wider von Multithreading-Anwendungen eingegangen.

Schließlich haben einige Beispielprojekte gezeigt, wie man Thread- und Synchronisierungsobjekte im Verbund einsetzt, um Windows 2000-Anwendungen zu erstellen.