Panikschalter

Asynchrone Unterbrechung bei InterBase 6.5

von Karsten Strobel

Sisyphos, Sohn des Aiolos und Herrscher von Korinth, erhielt in der Unterwelt als Strafe für Betrügereien die Aufgabe, einen schweren Marmorstein einen Berg hinaufzuwälzen, was aber stets kurz vor der Vollendung scheiterte, indem ihm der Stein entglitt und hinunterrollte, sodass Sisyphos immer und immer wieder neu beginnen musste. Dies könnte man als die Erfindung der Endlosschleife verstehen, deren Spätwirkungen auch und gerade heute – im IT-Zeitalter – noch peinsam spürbar sind.

Mit der Ende 2001 erschienenen Version 6.5 spendiert Borland seinen InterBase-Kunden endlich die lang erwartete und ersehnte Funktion zum Abbrechen laufender Abfragen. Wenn Sie sich bei der Formulierung eines SQL-Statements einmal gründlich vertun, kann es Ihnen leicht passieren, dass Sie auf eine Antwort des InterBase-Servers sehr lange – im Extremfall etliche Tage oder gar noch wesentlich länger – warten müssen. Meist passiert solches, wenn Sie ein SELECT-Kommando auf zwei oder mehrere große Tabellen anwenden und den Server dazu zwingen, jeden Kandidaten der einen Tabelle mit jedem aus der anderen zu kombinieren, zum Beispiel weil für eine gewünschte Beziehung kein geeigneter Index zur Verfügung steht. Oft werden lange Laufzeiten auch durch unindiziertes Sortieren großer Datenmengen verursacht. Erlebt hat solches schon fast jeder, der mit SQL-Datenbanken arbeitet.

Quellcode

Den Quellcode finden Sie auf der aktuellen Profi-CD sowie auf unserer homepage unter
www.derentwickler.de

Habe ich spendieren gesagt? Nun, dass stimmt nicht ganz. Dieses neue Feature ist nur in der so genannten „Certified"-Version verfügbar – sprich in der lizenzkostenpflichtigen Erscheinungsform von InterBase. Den Weg in den Open Source Code hat diese Neuerung indes nicht gefunden, denn derlei Mehrnutzen möchte Borland seit dem Neubeginn der kommerziellen Vermarktung ihrer Datenbanksoftware natürlich gerne versilbern.

Bei der inzwischen längst abgelösten Classic Architecture von InterBase konnte man sich bei exzessiv langen Laufzeiten noch behelfen, indem man den für die betreffende Datenbanksitzung zuständigen Serverprozess ganz einfach „hart" terminierte, auch wenn damit alle noch nicht endgültig gespeicherten Änderungen dieses Clients verloren gingen. Seit der Einführung der Superserver Architecture, bei der nicht mehr Prozesse sondern Threads zur Bedienung der Clients abgespalten werden, gibt es diese Möglichkeit nicht mehr. Im Extremfall muss man den Datenbankprozess insgesamt herunterfahren, um eine „ausgeflippte" Abfrage loszuwerden. Im Wiederholungsfall zieht man da leicht den Unmut der Kollegen auf sich, denn diese Notmaßnahme bedeutet für alle Anwender zumindest eine unfreiwillige Arbeitsunterbrechung und in vielen Fällen auch einen begrenzten Datenverlust.

Funktion ohne Verwendung

Mit der neuesten Version bietet Borland dem geschundenen Sisyphos nun endlich einen „Nothammer" namens Asynchronous Cancel an. Wer sich aber nach der viel versprechenden Lektüre der Release Notes [1] in der installierten Software auf die Suche nach einer entsprechenden Programmoption begibt, sieht sich bitter enttäuscht. Weder in IBConsole noch in gfix.exe oder anderswo findet sich ein Hinweis auf den neuen Notausgang. Borland hat nämlich nur die API erweitert, aber keine Anstalten gemacht, die Funktion auch an der Oberfläche anzubieten.

Vielleicht liegt das daran, dass sich Borlands Lösungsansatz nicht so einfach in vorhandenen Anwendungen nachrüsten lässt. Um ein laufendes Statement zu unterbrechen, muss man die schon früher bekannte API-Funktion *isc_dsql_free_statement()* mit der neu eingeführten Option *DSQL_CANCEL* aufrufen. Um das abzuwürgende Statement zu identifizieren, braucht man außerdem das Statementhandle. Das ist ein 32-Bit-Wert, den man zuvor durch Aufruf von *isc_dsql_allocate_statement()* erhalten hat. Da Sie wahrscheinlich bei der Client-Programmierung (etwa mit Delphi oder C-Builder) nur selten in direkten Kontakt mit der InterBase-API kommen und dieses mühsame Geschäft gerne einer Komponentenbibliothek wie IBExpress oder IBObjects überlassen, kann es sein, dass Ihnen diese Begriffe zunächst einmal fremd vorkommen. Ihre Komponenten besorgen das Allokieren eines SQL-Statements automatisch, zum Beispiel, wenn Sie die *Pre-*

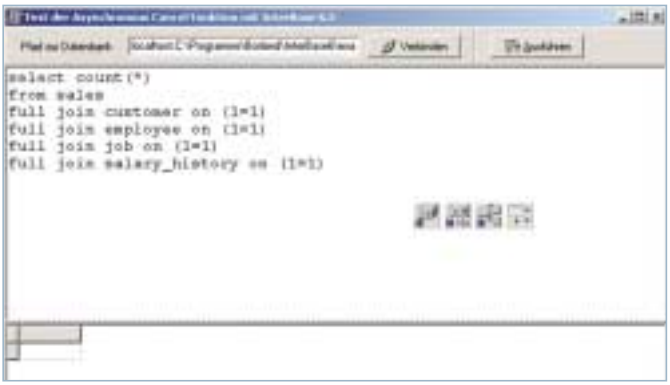

Abb. 1: ISQL-Nachbau als Testanwendung

pare-Methode einer Query-Komponente aufrufen. Nach dem Prepare – also *vor* dem eigentlichen Ausführen des Statements – steht Ihnen das Statementhandle über eine Objekteigenschaft zur Verfügung (allerdings leider nicht, wenn Sie die BDE verwenden). Die Ausführung eines Statements erfolgt bei InterBase immer synchron, d.h. der aufrufende Client bleibt so lange blockiert, bis der Server den Aufruf erledigt hat oder ein Fehler aufgetreten ist.

Wie, bitteschön, soll der Client also den *isc_dsql_free_statement()*-Aufruf zwecks Abbruch eines Statements absetzen, wenn er doch während der Ausführung eben dieses Statements blockiert ist? Die Antwort klingt zwar einleuchtend, ist aber etwas schwierig umzusetzen: Man muss einen separaten Thread verwenden, der eine eigene Verbindung zur Datenbank aufbaut und das Statement in diesem Kontext abbrechen. Erforderlich ist dabei natürlich, dass dieser „Abbruchthread", wie ich ihn nennen möchte, das Handle des im „Arbeitsthread" laufenden Statements kennt.

Bekanntlich verfügt eine Windowsanwendung immer über mindestens einen – und oft nur diesen einen – Thread, der für die Darstellung der Benutzeroberfläche (Fenster, Steuerelemente, usw.) verantwortlich ist. Viele Anwendungen beschränken sich darauf, alle Verarbeitungsschritte ebenfalls diesem Thread zu überlassen. Dadurch kommt es beim Ausführen umfangreicherer Abläufe, wie z.B. langwierigen Algorithmen oder eben der Ausführung synchroner Datenbankanfragen, zum „Einfrieren" der Programmoberfläche. Selbst die Darstellung des eigenen Fensters ist blockiert, solange der Hauptthread mit anderen Aufgaben zu beschäftigt ist, um die Aufforderung zum Neuzeichnen aus der Nachrichtenwarteschlange entgegenzunehmen. Schon alleine deshalb kann es durchaus sinnvoll sein, Threads für lang dauernde Arbeitsschritte einzusetzen.

Wir haben schon festgestellt, dass für den asynchronen Abbruch von DSQL-Statements ein Arbeitsthread und ein Abbruchthread vonnöten sind. Wenn wir von einer konventionellen Clientanwendung ausgehen, können wir die eine oder die andere Aufgabe dem Hauptthread überlassen. Also brauchen wir einen zusätzlichen Thread, der entweder die Ausführung der Statements übernimmt, oder – falls wir diesen Job beim Hauptthread belassen – darauf lauert, gestartete Statements abbrechen zu dürfen. Leider bringt die Multithread-Programmierung auch eine Menge neuer Probleme mit sich, von denen die Synchronisierung dieser voneinander entkoppelten Programmfäden das größte ist. So charmant die Idee auch klingt, die Datenbankabfragen einem Subthread zu überantworten – es würde wohl den Rahmen dieses Artikels sprengen und uns über das Ziel hinausschießen lassen. Deshalb möchte ich das Pferd anders herum aufzäumen und Ihnen eine Lösung anbieten, bei der der Hauptthread weiterhin die Datenbankstatements abarbeitet und der Abbruchthread ein Eigenleben führt. Auch lässt sich dieser Ansatz viel leichter in bestehende Anwendungen integrieren.

Mein Lösungsvorschlag besteht im Wesentlichen aus der Pascal Unit *thrdIBCancelDlgUnit.pas*, die Sie, zusammen mit den hier verwendeten Beispielprojekten, auf der Profi-CD oder zum Download im Web finden. Diese Unit implementiert eine Klasse für einen Thread, der von der Anwendung in einer einzigen Instanz gestartet wird und der dann aktiv bleibt und darauf wartet, dass es etwas zu tun gibt. Vom Hauptthread benötigt dieser Abbruchthread immer kurz vor dem Start eines DSQL-Statements ein Signal. Daraufhin wartet der Abbruchthread noch eine einstellbare Zeit und stellt dann einen kleinen Dialog dar, mit dem der Benutzer den Abbruch per Mausklick anfordern kann, falls er den Eindruck hat, dass seine Anfrage nicht mehr in erträglicher Zeit beantwortet werden wird.

Gekonnt gestümpert

Doch zunächst zu meiner minimalistischen Beispielanwendung, mit der wir den Abbruchthread testen können. Abbildung 1 zeigt das Fenster dieser Anwendung, mit der man ein beliebiges SELECT-Statement ausführen und das Ergebnis in einem TDBGrid anzeigen lassen kann. Im Editor habe ich schon ein geeignetes Statement vorbereitet, das sich auf Borlands Beispieldatenbank EMPOLYEE.GDB anwenden lässt:

```
select count(*)
from sales
full join customer on (1=1)
full join employee on (1=1)
full join job on (1=1)
full join salary_history on (1=1)
```

Ein solches Kommando würde man in der Praxis wohl kaum formulieren, aber für unsere Zwecke ist es ideal. Zwar enthalten die hier angesprochenen Tabellen *sales*, *customers*, *employees* usw. jeweils nur ein paar Dutzend Einträge, aber durch die Verknüpfung dieser Tabellen mittels *FULL JOIN* fordern wir ein kartesisches Produkt dieser Tabellen an, also eine Kombination „jedes mit jedem", und fordern mit *SELECT COUNT(*)* die Datenbank auf, die Einträge der Ergebnismenge zu zählen. Das Ergebnis dürfte sich gemäß dem Umfang der Beispieltabellen auf 33 x 15 x 42 x 31 x 49 = 31.580.010 belaufen. Das Statement zwingt die Datenbank aber, alle Kombinationen tatsächlich herzustellen. Meine armes Notebook braucht dafür über 75 Minuten! Natürlich gäbe es viel effizientere Methoden,

das Ergebnis zu berechnen, aber mein Ziel war es ausnahmsweise, eine möglichst *ineffiziente* Form zu finden.

Das Beispielprojekt steht Ihnen als IBX- und als IBO-Version zur Verfügung. Die Unterschiede sind minimal, daher beschränke ich mich jetzt auf die IBX-Variante, aber alles hier Gesagte trifft auch auf die IBO-Version zu. Listing 1 zeigt drei Ereignismethoden aus dem Quellcode des Hauptformulars.

Der Thread wird im AfterConnect Event des Verbindungsobjektes, also sofort nach dem Anmelden der Anwendung bei der Datenbank, erzeugt. Da der Thread eine eigene Verbindung herstellen soll, benötigt er natürlich auch die Login-Parameter (Datenbankpfad, User, Password, Role), die ihm beim Aufruf des Konstruktors als Mitgift gegeben werden. Außerdem erhält er noch eine Zeitkonstante (hier 5000, in Millisekunden), die ihn anweist, nach dem Starten eines Statements nicht sofort aktiv zu werden, sondern zunächst diese Zeit abzuwarten, ob die Abfrage nicht innerhalb dieser kurzen Zeit ohnehin erledigt werden kann. Später mehr dazu.

Listing 1: Threaderzeugung und -steuerung

```
procedure TfrmTestAsyncCancelIBX.
                IBDatabaseAfterConnect(Sender: TObject);
begin
 FthrdCancelDlg :=
  TthrdCancelDlg.Create(edDBPath.Text,
          IBDatabase.Params.Values['user_name'],
          FGrabPassword,
          IBDatabase.Params.Values['sql_role_name'],
          5000);
end;

procedure TfrmTestAsyncCancelIBX.btnAusfuehrenClick
                                    (Sender: TObject);
begin
 qryISQL.Close;
 qryISQL.Prepared := false;
 qryISQL.SQL.Assign(memISQL.Lines);
 qryISQL.Prepare;
 FthrdCancelDlg.StartingStatement
                                (qryISQL.StmtHandle);
 try
  qryISQL.Open;
 finally
  FthrdCancelDlg.StatementDone;
 end;
end;

procedure TfrmTestAsyncCancelIBX.
            IBDatabaseAfterDisconnect(Sender: TObject);
begin
 try
  FthrdCancelDlg.SaferWaitFor;
 finally
  FthrdCancelDlg := nil;
 end;
end;
```

Testament machen

In der *OnClick*-Methode des AUSFÜHREN-Buttons des Hauptfensters wird der eingegebene Code zunächst in die SQL-Eigenschaft einer Query-Komponente übernommen und die Anweisung mittels Prepare an den Server zur Vorkontrolle geschickt. Wird hier kein syntaktischer Fehler oder dergleichen festgestellt (d.h. wenn dieser Aufruf ohne Exception zurückkehrt), können wir auf das Statement-Handle zugreifen und es dem Abbruchthread durch Aufruf der Methode *FthrdCancelDlg.StartingStatement(handle)* übergeben. Dies dient als Ankündigung, dass das mit diesem Handle verbundene Statement nun gleich ausgeführt werden soll (*qryISQL.Open*), womit der Abbruchthread vorsorglich in Alarmbereitschaft versetzt wird. Die *Open*-Methode ist diejenige, die je nach Formulierung des Statements wenig oder viel Zeit in Anspruch nehmen kann. Nach der Rückkehr aus *Open* wird mittels *FthrdCancelDlg. StatementDone()* Entwarnung gegeben und zwar unabhängig davon, ob die Abfrage erfolgreich gewesen ist oder nicht. Erledigt ist erledigt. Beim Trennen der Datenbankverbindung wird der Thread durch Aufruf von *FthrdCancelDlg.SaferWaitFor()* wieder beendet und zur Selbstzerstörung veranlasst.

Verwechseln Sie das Problem der langen Laufzeit einer DSQL-Abfrage bitte nicht mit dem *Fetch-All*-Problem. Wenn Sie in IBConsole etwa *SELECT * FROM grosse_tabelle* eingeben, dann dauert die Ausführung dieses Befehls nur sehr kurze Zeit. Als Ergebnis kommt ein Cursorhandle zurück, der Ihnen den Inhalt der Tabelle in unsortierter Reihenfolge anbietet. Die Clientanwendung (also IBConsole) kann dann Datensatz für Datensatz abholen (*fetch*). Leider verhält sich IBConsole, wie viele andere Anwendungen,

Anzeige

Listing 2: Synchronisation von Threads

```
TthrdCancelDlg = class(TThread)
private
 Flpdt: PDLGTEMPLATE;
 FWakeupSema: THandle;
 FHwndDlg: Hwnd;
 FRunningStatement: isc_stmt_handle;
 FShowUpDelay: integer;
 FGDS_Handle: THandle;
 FDBPath, FUsername, FPassword, FRole: string;
 Fisc_attach_database: Tisc_attach_database;
 Fisc_dsql_free_statement: Tisc_dsql_free_statement;
 Fisc_detach_database: Tisc_detach_database;
 procedure DefineDialog;
 function CancelStatement(ACancel_stmt_handle: isc_stmt_handle): Boolean;
 function LoadGDS: Boolean;
 procedure UnloadGDS;
public
 constructor Create(ADBPath, AUsername, APassword, ARole: string; AShowUpDelay:
                                                                    Integer);
 destructor Destroy; override;
 procedure Execute; override;

 //Die folgenden Methoden sind zum asynchronen Aufruf durch den Hauptthread
                                                                  vorgesehen
 procedure StartingStatement(Astmt_handle: isc_stmt_handle);
 procedure StatementDone;
 function SaferWaitFor: DWORD;
end;

implementation

constructor TthrdCancelDlg.Create(ADBPath, AUsername, APassword, ARole: string;
 AShowUpDelay: Integer);
begin
 FDBPath := ADBPath;
 FUsername := AUsername;
 FPassword := APassword;
 FRole := ARole;
 FShowUpDelay := AShowUpDelay;
 DefineDialog;
 FWakeupSema := CreateSemaphore(nil, 0, 1000, nil);
 FreeOnTerminate := true;
 inherited Create(false);
end;

//...

procedure TthrdCancelDlg.Execute;
var
 PseudoHWnd: THandle;
 DlgResult: LRESULT;
 stmt: isc_stmt_handle;
 timeout, wfso: DWORD;
begin
 PseudoHWnd := AllocateHWnd(nil);
 try
  while not Terminated do
  begin
   timeout := INFINITE;
   stmt := nil;

   repeat
    wfso := WaitForSingleObject(FWakeupSema, timeout);
    if Terminated then Exit;
    if not Assigned(stmt) or (stmt <> FRunningStatement) then
    begin
     stmt := FRunningStatement;
     if Assigned(stmt) then timeout := FShowUpDelay else timeout := INFINITE;
     continue; //wait again
    end;
   until wfso = WAIT_TIMEOUT;

   DlgResult := DialogBoxIndirectParam(hInstance, Flpdt^, PseudoHWnd, @DialogProc,
                                                                  Integer(Self));
   InterlockedExchange(Integer(FHwndDlg), 0);
   if DlgResult = IDCANCEL then CancelStatement(FRunningStatement);
  end;
 finally
  DeallocateHWnd(PseudoHWnd);
 end;
end;

procedure TthrdCancelDlg.StartingStatement(Astmt_handle: isc_stmt_handle);
begin
 InterlockedExchange(Integer(FRunningStatement), Integer(Astmt_handle));
 ReleaseSemaphore(FWakeupSema, 1, nil);
end;

procedure TthrdCancelDlg.StatementDone;
var h: THandle;
begin
 StartingStatement(nil);
 h := InterlockedExchange(Integer(FHwndDlg), 0);
 if h <> 0 then EndDialog(h, 0);
end;

function TthrdCancelDlg.SaferWaitFor: DWORD;
var H: THandle;
begin
 //duplicate handle to be able to do GetExitCodeThread at the end of this method
 Win32Check(DuplicateHandle(GetCurrentProcess, Handle, GetCurrentProcess, @H,
          0, false, DUPLICATE_SAME_ACCESS));
 Terminate;
 StatementDone;
 WaitForSingleObject(H, INFINITE);
 Win32Check(GetExitCodeThread(H, Result));
 Win32Check(CloseHandle(H));
end;
```

ziemlich stur, wenn Sie im Ergebnis-Grid auf den letzten Datensatz springen (z.B. mit Strg+End). Auch wenn die Tabelle zigmillionen Datensätze enthält, versucht IBConsole alles abzuholen und im Hauptspeicher zwischenzuspeichern bzw. im Grid anzuzeigen. Befehl ist Befehl. Ein Abbrechen ist auch hier leider nicht vorgesehen, wenngleich dies sehr einfach machbar wäre. Der Client bräuchte ja nur mit seinen ständigen Fetch-Aufrufen aufzuhören. Ein Asynchronous Cancel hilft in diesem Fall nicht mehr, da das eigentliche Statement längst abgearbeitet ist. Unser Beispiel-SELECT auf die Employee-Tabellen ist ein anderer Fall. Hier dauert das Statement sehr lange; wenn es endlich fertig ist, holt der Client das Ergebnis (den Count-Wert) mit einem einzigen Fetch ab.

Listing 2 zeigt die Klassendefinition des Abbruchthreads, seine Methoden *Create* und *Execute* und die Implementierung der Methoden, mit denen die Aktivitäten des Threads von außen gesteuert werden. Diese drei letzten Methoden – wir haben gerade schon gesehen, wie sie verwendet werden – sind für den Aufruf durch den Hauptthread vorgesehen und bilden quasi die Schnittstelle des im Untergrund tätigen Threads zu den Vorgängen an der Oberfläche. Für die Synchronisation ist eine Semaphore , die im Konstruktor mit *FWakeupSema := CreateSemaphore()* angelegt wird, vorgesehen. Mit dem WinAPI-Aufruf *ReleaseSemaphore(FWakeupSema,, 1, nil)* „weckt“ der Hauptthread den Abbruchthread auf, wenn es etwas zu tun gibt.

Zurück zu den Wurzeln

Natürlich soll der Abbruchthread nicht einfach nach fünf Sekunden jedes Statement stoppen, falls es in dieser Zeit nicht fertig geworden ist. Diese Entscheidung muss natürlich beim Anwender liegen. Also braucht der Thread auch irgendeine Art Schnittstelle zum Anwender. Die normale Programmoberfläche steht unter der Kontrolle des Hauptthreads und der ist zum fraglichen Zeitpunkt blockiert und daher außerstande, irgendwelche Anweisungen entgegenzunehmen. Das Einfachste wäre sicherlich, den Abbruchthread ein eigenes Fenster öffnen zu lassen und darin eine Abbruch-Schaltfläche anzubieten. Leider können wir dafür nicht die Dienste der VCL in Anspruch nehmen, da diese nicht Thread-sicher programmiert ist. Die komfortable Methode fällt somit aus, aber ein Rückgriff auf die Urtechniken der Windowsprogrammierung bietet einen Ausweg: Die WinAPI-Funktion *DialogBox()* ist wohl nach *MessageBox()* die simpelste Art, ein Fenster anzuzeigen. Das Layout des Dialogs wird dabei aus den eingebundenen Programmressourcen geladen (erinnern Sie sich an *Borland Pascal für Windows*?). Noch spartanischer kann man es haben, wenn man stattdessen *DialogBoxIndirectParam()* verwendet und die Beschreibung des gewünschten Dialogs zuvor im Speicher aufbaut. Die Methode *DefineDialog()*, die vom Konstruktor der Thread-Klasse einmalig aufgerufen wird und die diese Struktur vorbereitet, bereitet eine solche Datenstruktur vor. Abbildung 2 zeigt das damit beschriebene Fenster, das der Abbruchthread zur Anzeige bringen soll.

Wenn nun nach ein paar Sekunden Wartezeit der Abbruchdialog angezeigt wird, ist eigentlich noch nicht entschieden, ob das Statement wirklich abgebrochen werden wird oder ob sich

Anzeige

Listing 3: Abbruch per InterBase API-Aufruf

```
function TthrdCancelDlg.CancelStatement(ACancel_stmt_handle: isc_stmt_handle):
                                                                    Boolean;

 procedure BuildPBString(var PB: array of char; var PBLen: Integer; item: byte; contents:
                                                                    string);
 //Add a string value to a parameter block
 var len: Integer;
 begin
  PB[PBLen] := char(item);
  inc(PBLen);
  len := Length(Contents);
  PB[PBLen] := char(len);
  inc(PBLen);
  StrPCopy(@PB[PBLen],Contents);
  inc(PBLen,len);
 end;

var
 DPB: TParamBlock; //parameter block for database connection
 DPBLen: Integer; //length of Paramblock
 ISC_Result: ISC_STATUS;
 StatusVector: ISC_STATUS_VECTOR;
 DBHandle: isc_db_handle;
 s: string;
 IsLocal: boolean;
begin
 if Assigned(ACancel_stmt_handle) then Result := LoadGDS
                else Result := false;
 if Result then
 try
  ZeroMemory(@StatusVector, SizeOf(StatusVector));
  DBHandle := nil;
  ZeroMemory(@DPB, sizeof(DPB));
  DPB[0] := char(isc_dpb_version1);
  DPBLen := 1;
  BuildPBString(DPB, DPBLen, isc_dpb_user_name, FUsername);
  BuildPBString(DPB, DPBLen, isc_dpb_password, FPassword);
  if FRole <> '' then
   BuildPBString(DPB, DPBLen, isc_dpb_sql_role_name, FRole);

  s := FDBPath;
  IsLocal := (Length(s) > 2) and (s[2] = ':') and (Pos(':',Copy(s, 3, Length(s))) = 0);
  if IsLocal then s := 'localhost:' + s;
  ISC_Result := Fisc_attach_database(@StatusVector, Length(s), PChar(s),
   @DBHandle, DPBLen, @DPB);
  if ISC_Result <> 0 then Result := false
  else
   try
    ISC_Result := Fisc_dsql_free_statement(@StatusVector, @ACancel_stmt_handle,
                                                                    DSQL_CANCEL);
    if ISC_Result <> 0 then Result := false;
   finally
    Fisc_detach_database(@StatusVector, @DBHandle);
   end;
 finally
  UnloadGDS;
 end;
end;
```

der Benutzer in Geduld übt und auf das reguläre Ende seiner Anfrage wartet. Im letzten Fall kehrt der *Open*-Aufruf des Hauptthreads ohne Beanstandung zurück und das Ergebnis kann angezeigt werden. Was passiert aber dann mit dem Abbruchdialog, der ja immer noch angezeigt wird? Er sollte sich natürlich von selbst schließen und der Abbruchthread soll sich unverrichteter Dinge wieder auf die Lauer legen und auf neue Beute warten. Also brauchen wir eine zweite Methode, um den Dialog programmgesteuert durch den Hauptthread schließen zu lassen. Wir haben gesehen, dass nach der Rückkehr aus dem *Open*-Aufruf immer die Methode *StatementDone* der Thread-Klasse aufgerufen wird. Dort wird, sofern der Abbruchdialog bereits geöffnet worden ist, die WinAPI-Funktion *EndDialog()* aufgerufen. Dabei nutzen wir eine sympathische Eigenschaft der DialogBox-Technik aus. Anhand des Handles dieses Dialogs ist es nämlich möglich, ihn auch aus einem anderen Thread heraus zu schließen. Das Dialogfenster schließt sich ganz von selbst und kehrt mit dem Statuscode zum Ausrufer zurück, der beim *EndDialog()*-Aufruf übergeben worden ist. Damit kann der Abbruchthread zwischen echtem Tastendruck und Zwangsschließung unterscheiden.

Werfen wir einen Blick auf die *Execute*-Methode, die für die eigentliche Steuerung des Abbruchthreads zuständig ist. Wie üblich wird der Ablauf von einer ständigen Schleife bestimmt, die erst dann verlassen wird, wenn dem Thread die Lebensberechtigung entzogen wird (*while not Terminated*). Im Inneren dieser Schleife gibt es eine weitere (*repeat until wfso=WAIT_TIMEOUT*), nach deren Eingang sich der Thread zunächst einmal mit *WaitForSingleObject()* schlafen legt. Das dabei erwartete Ereignis ist das Auslösen der Semaphore bzw. ein Timeout. Beim ersten Aufruf ist die Timeout-Grenze auf Unendlich (*INFINITE*) eingestellt. Das Warten hat ein Ende, wenn der Hauptthread mit seinem *StartingStatement()*-Aufruf die Semaphore erhöht. *WaitForSingleObject()* wird dann erneut aufgesucht, diesmal mit dem als Karenzzeit vorgesehenen Timeout-Wert. Erst wenn diese Zeit abgelaufen ist, wird die innere *repeat*-Schleife verlassen.

Sollten Sie mit der Thread-Programmierung und den dabei wichtigen Synchronisationsmechanismen nicht vertraut sein, es aber werden wollen, möchte ich Ihnen das Buch „Delphi Win32-Lösungen“ von Andreas Kosch [2] empfehlen.

Wir haben nun lange genug um den heißen Brei herumprogrammiert. Sobald der Abbruchdialog geöffnet ist und sich der Anwender entschließt, den gleichnamigen Button zu drücken, ruft die *Execute*-Methode *CancelStatement()* auf und übergibt das vorher vom Hauptthread mitgeteilte Statementhandle als Parameter. Listing 3 zeigt den Quellcode dieser Methode.

Weil diese im Kontext des Abbruchthreads aufgerufen wird – der Hauptthread ist ja zu diesem Zeitpunkt blockiert – muss hier eine eigene Verbindung zur Datenbank aufgebaut werden, da InterBase die Nutzung einzelner Verbindungen durch mehrere Threads nicht duldet. Damit nicht genug, muss der Abbruchthread auch die InterBase-Clientbibliothek (gds32.dll) mit *LoadLibrary()* laden, obwohl die Komponentenbibliothek (IBX bzw. IBO) dies für den Hauptthread schon erledigt hat. Dadurch erhält die DLL die Gelegenheit, einige Variablen für den neu hinzukommenden Thread zu initialisieren. Außerdem darf der Thread keine lokale Datenbank-

Abb. 2: Vom Thread angezeigter Abbruchdialog

Abb. 3: Meldung nach dem Abbruch

verbindung herstellen, auch wenn der Datenbankdienst auf dem selben Rechner läuft wie die Clientanwendung. Die konkurrierenden Threads würden sich sonst gegenseitig blockieren, mit der Folge, dass der Abbruchthread den Befehl zum Unterbrechen der laufenden Abfrage erst durchbringen kann, nachdem diese beendet wurde – und das ist ja nun wirklich nicht der Sinn der Sache. Daher wird in *CancelStatement()* der Verbindungsstring untersucht und die Zeichen *localhost:* vorangestellt, falls der Hauptthread eine lokale Verbindung verwendet. Wenn diese Spielregeln beachtet werden, dann ist Multithreadprogrammierung mit InterBase durchaus möglich.

Im Ganzen ruft *CancelStatement()* nur drei Funktionen der gds32.dll auf, nämlich *isc_attach_database()* um die Verbindung zur Datenbank herzustellen, *isc_dsql_free_statement()* mit dem Statementhandle und der neu eingeführten Optionskonstante *DSQL_CANCEL* als Parameter, um den Abbruch anzufordern und schließlich *isc_detach_database()*, um die Verbindung wieder zu trennen. Da diese Operationen den Einsatz von Komponenten nicht rechtfertigen und um etwaige zusätzliche Probleme mit der Thread-Sicherheit zu vermeiden, verwende ich hier direkte API-Programmierung. Die Methode *LoadGDS* erledigt den für jeden Thread obligatorischen *LoadLibrary()*-Aufruf und besorgt mit *Get ProcAddress()* die Zeiger auf die drei erwähnten InterBase-API-Funktionen. In *UnloadGDS()* verabschiedet sich der Thread mittels *FreeLibrary()* wieder von der gds32.dll.

Sobald der Abbruchthread der Datenbank das Signal zur Aufgabe gibt, wird das laufende Statement mit einem entsprechenden Statuscode beendet. Der Hauptthread wertet dies als Fehlermeldung und löst eine Exception aus (siehe Abbildung 3). Wenn Ihnen diese Meldung für einen gewollten Abbruch zu unfreundlich erscheint, können Sie natürlich diese Exception fangen und eine besser verständliche Information ausgeben. Das einmal abgebrochene Statement lässt sich nicht wieder ausführen; erst nach einem erneuten Prepare des SQL-Codes lässt sich ein neuer Anlauf starten.

Aus Platzgründen ist es nicht möglich, den ganzen Quellcode der hier vorgestellten Lösung abzudrucken. Ich habe versucht, die wesentlichen Teile herauszugreifen. Wenn Sie das hier Beschriebene bis ins letzte Detail nachvollziehen möchten, dann empfehle ich Ihnen, das Beispielprojekt auszuprobieren und mit dem Debugger nachzuvollziehen. Erfolg werden Sie aber nur in Verbindung mit der InterBase Version 6.5 haben. Auf der Borland-Webseite können Sie eine Evaluierungsversion herunterladen [3].

Fazit

Zwar können Sie die neuen Fähigkeiten von InterBase 6.5 nutzen, um Ihre Anwendung etwas toleranter gegenüber verunglückter SQL-Anweisungen zu machen. Aber den vollen Nutzen aus dem Asynchronous Cancel Feature können Sie nur schöpfen, wenn Sie selbst Hand anlegen. Borland liefert in diesem Fall nur das Schloss, nicht aber den Schlüssel. Dies könnte mit InterBase 7.0 anders werden, denn gerüchteweise hört man, dass hier umfangreichere Management und Auditing-Funktionen Einzug halten sollen. Nach der Borland-Konferenz wird man vielleicht mehr wissen.

Links & Literatur

[1] InterBase 6.5 Release Notes, Seite 15 ff.
[2] Andreas Kosch: Delphi Win32-Lösungen, Software & Support Verlag
[3] www.borland.com/interbase/tryitnow

Anzeige