Annales UMCS Informatica AI 1 (2003) 303-308

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Concurrent programming and futures

Andrzej Daniluk*

Institute of Physics, Maria Curie-Skłodowska University,
Pl. M.Curie-Skłodowskiej 1, 20-031 Lublin, Poland

Abstract

If we are manage complexity, we must create a model of the universe. The goal of model is to create a meaningful abstraction of the real world. Such an abstraction should be simpler than the real world but should also reflect accurately the real world so that we can use the model to predict the behavior of things in the real world. The object-oriented software design is about building good models. It consists of two significant pieces: a modelling language and process. In the present paper a new and little-known approach for concurrent programming is presented.

1. Introduction

The modelling language is the least important aspect of object-oriented analysis and design; unfortunately, it tends to attract the most attention. A modelling language is nothing more than a convention for how we'll draw our model on paper. The process of object-oriented analysis and design is much more complex and important than the modelling language; as an industry, we've decided to use the UML (Universal Modelling Language) as a commercial product from Rational Software, Inc. The process of software design is iterative. It means that as we develop software, we go through the entire process repeatedly as we strive for enhanced understanding of the requirements. The design directs the implementation, but the details uncovered during implementation feed back into the design. Most important, we do not try to develop any sizable project in a single, orderly, straight line; rather, we iterate over pieces of the projects, constantly improving our design and refining our implementation [1-2]. The goal of this work is to produce code that meets the stated requirements and that is reliable, extensible, and maintainable.

* *E-mail address*: adaniluk@tytan.umcs.lublin.pl

2. Concurrent programming

The future of programming is concurrent programming. Not too long ago, sequential, command-line programming gave way to graphical, event-driven programming and now single-threaded programming is yielding to multithreaded programming.

Borland C++ Builder and Borland Delphi includes features to support concurrent programming-not as much support as we find in Ada, but more than in most traditional programming languages. In addition to the language features, we can use the Windows API and its threads, process, futures, and so on [1, 3].

3. Threads and processes

A thread is a flow of control in a program. A program can have many threads, each with its own stack, its own copy of the processor register, and related information. On a multiprocessor system, each processor can run a separate thread. On a uniprocessor system, Windows and Linux create the illusion that threads are running concurrently, though only one thread at a time gets to run.

A process is a collection of threads all running in a single address space. Every process has at least one thread, called the main thread. Threads in the same process can share resources such as open files and can access any valid memory address in the process address space.

4. The TThread class and BeginThread() function

The easiest way to create a multithread application in C++ Builder or Delphi is to write a thread class that inherits from TThread. TThread class is not part of the C++ Builder and Delphi languages, but is declared in the Classes.hpp or Classes.pas units [1, 3-4]. Example 4.1 shows the declaration for the TThread class.

Example 4.1. Using the TThread class

```
#ifndef TPrintThreadH
#define TPrintThreadH
//---------------------------------------------------------------------------
#include <StdCtrls.hpp>
#include <ExtCtrls.hpp>
#include <Dialogs.hpp>
#include <Forms.hpp>
#include <Controls.hpp>
#include <Graphics.hpp>
#include <Classes.hpp>
#include <SysUtils.hpp>
#include <Messages.hpp>
#include <Windows.hpp>
```

```
#include <System.hpp>
//---------------------------------------------------------------------------
class TTPrintThreadForm : public TForm
{
__published:
TButton *Button1;
TLabel *Label1;
TBevel *Bevel1;
TBevel *Bevel2;
TBevel *Bevel3;
TLabel *Label2;
TLabel *Label3;
void __fastcall ButtonPrint(TObject *Sender);
...

private:
int ThreadsRunning;
        ...
void __fastcall ThreadDone(TObject *Sender);

public:
    void __fastcall PrintArray(TPaintBox *Box, const int *A, const int A_Size);
    ...
};
//---------------------------------------------------------------------------
typedef int TPrintArray[225];

typedef TPrintArray *PPrintArray;
//---------------------------------------------------------------------------
extern TTPrintThreadForm *TPrintThreadForm;
//---------------------------------------------------------------------------
#endif
```

If we don't want to write a class, we can use BeginThread() function. We are wrappers for the Windows API calls CreateThread() function, but we must use C++ Builder's and Delphi's functions instead of the Windows API directly. C++ Builder and Delphi keeps a global flag, IsMultiThread, which is true if our program calls BeginThread() or starts a thread using TThread class. C++ Builder and Delphi check this flag to ensure thread safety when allocating memory. If we call the CreateThread() function directly be sure to set IsMultiThread to true. Example 4.2 shows the declaration for the BeginThread() function.

Example 4.2. Using the BeginThread() function for calculate of sinl() function

```
#include <vcl.h>
#include <math.h> #pragma hdrstop
#include "Unit_26.h"
```

```
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;
LONG Run_Thread = 0;
UINT Thread_ID;
HANDLE hEvent;

SECURITY_ATTRIBUTES  security_attrib = {
   sizeof(SECURITY_ATTRIBUTES),
   NULL,
   TRUE
};
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
       : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
 Memo1->Lines->Clear();
}
//---------------------------------------------------------
int __fastcall Count_Sin(LPVOID Parameter)
{
 long double y, x=0.0;
 for(;;)
   {
    y = sinl(x*M_PI/180);
    Form1->Memo1->Lines->Add(FloatToStr(x)+
                          "      "+FloatToStr(y));
    WaitForSingleObject((LPVOID)Run_Thread, 100);
     x += 1.0; // x = x + 1
   }
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Run_Thread = BeginThread(&security_attrib, 4096,
             Count_Sin, this, CREATE_SUSPENDED,
             Thread_ID);
  if(Run_Thread == (int)INVALID_HANDLE_VALUE)
     {
       MessageBox(0, "Thread ERROR", "ERROR", MB_OK);
     }
     else {
```

```
      ResumeThread((LPVOID)Run_Thread);
      hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
      if (hEvent) {
        WaitForSingleObject(hEvent, 100 /*ms*/ );
        CloseHandle(hEvent);
        Button1->Enabled = FALSE;
      }
    }
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
   SuspendThread((LPVOID)Run_Thread);
}
//---------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
   ResumeThread((LPVOID)Run_Thread);
}
//---------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  CloseHandle((LPVOID)Run_Thread);
  Application->Terminate();
}
//---------------------------------------------------------
```

5. Futures

Writing a concurrent program can be more difficult than writing a sequential program. We need to think about race conditions, synchronization, shared variables, and others. Futures help reduce the intellectual clutter of using threads. A future is an object that promises to deliver a value sometime in the future. The application does its work in a main thread and calls upon futures to fetch or compute information concurrently. The future does its work in a separated thread, and when the main thread needs the information, it gets from the future object. For example, we can define the future class by inheriting from TFuture and overriding the Compute method. The Compute method does whatever work is necessary and returns its result as a Variant type [2]. Example 5.1 shows the declaration for the TFuture class.

Example 5.1 Declaration of the TFuture class

```
class TFuture
{
 private:
```

```
    System::TObject* __fastcall ExceptObject(void);
    void * fExceptAddr;
    HANDLE fHandle;
    bool fTerminated;
    LongWord fThreadID;
    LongWord fTimeOut;
    Variant fValue;
    bool __fastcall GetIsReady(void);
    bool __fastcall GetValue(void);
  protected:
    void __fastcall RaiseException(void);
  public:
    virtual void __fastcall AfterConstruction(void);
    Variant __fastcall Compute(void);
    void __fastcall Terminate(void);
    __property HANDLE handle={read=fHandle};
    __property bool IsReady={read=GetIsReady};
    __property bool Terminated={read=fTerminated, write=fTerminated};
    __property LongWord ThreadID={read=fThreadID};
    __property LongWord TimeOut={read=fTimeOut, write=fTimeOut};
    __property Variant Value={read=GetValue};
};
```

When the application needs the future, it reads the Value property and the GetValue method waits until the thread is finished. If the thread raised an exception, the future object reraises the same exception object at the original exception address. If in application everything goes as planned, the future value is returned as a Variant type.

6. Summary

In this paper the technique of programming by futures is presented. The major advantage of using futures is their simplicity. We can often implement the TFuture-derived class as a simple, linear subroutine. Using a future is as simple as accessing a property. All the synchronization is handled automatically by TFuture. Concurrent programming can be tricky, but with care and caution, we can write application that uses threads and process correctly, efficiently, and effectively. Using a future is as simple as accessing a property. It is worth noticing that the synchronization is handled automatically by futures.

References

[1] Lischner R., *Delphi in a Nutshell*, O'Reilly & Associates, Inc., (2000).
[2] Liberty J., *Teach Yourself C++ in 21 Days (4th Edition)*, SAMS, (2000).
[3] Daniluk A., *C++ Builder 6. Ćwiczenia zaawansowane*, Helion, (2003), in Polish.
[4] Daniluk A., *C++ Builder. Kompendium programisty*, Helion, (2003), in Polish.