Jacek Matulewski
http://www.phys.uni.torun.pl/~jacek/

Tworzenie aplikacji Windows

Aplikacje wielowątkowe

Ćwiczenia

Toruń, 13 grudnia 2002

Najnowsza wersja tego dokumentu znajduje się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad4_watki.pdf

Źródła opisanych w tym dokumencie programów znajdują się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad4_watki.zip

I. Spis treści

I. Spis treści ... 2
II. Klasa TThread ... 3
1. Tworzenie klasy pochodnej względem TThread ... 3
2. Wywoływanie wątków ... 6
3. Sprawdzanie własności Terminated ... 11
4. Zdarzenie TThread.OnTerminate ... 12
5. Zmiana priorytetu wątku ... 13
III. Kontrola wątku za pomocą funkcji WinAPI ... 14
1. Wymuszenie zakończenia wątku – TerminateThread ... 14
2. Synchronizacja wątku – sekcje krytyczne ... 14

II. Klasa TThread

1. Tworzenie klasy pochodnej względem TThread

Tworzenie aplikacji wielowątkowych w Delphi/C++ Builderze oznacza w istocie projektowanie klasy dziedziczącej z klasy `TThread`. Do tworzenia i kontroli wątków można oczywiście wykorzystać funkcje WinAPI, ale nie może się to równać łatwości i elegancji wykorzystania tego obiektu VCL.

Zbudujmy aplikację wielowątkową, której zadaniem będzie sortowanie listy liczb w kolejności rosnącej czterema metodami: sortowanie bąbelkowe, sortowanie przez wybór, sortowanie przez wstawianie i szybkie sortowanie (ang. *quick sort*). Wyniki umieszczane będą w osobnych ListBoxach[1]. Pierwowzorem tego projektu jest przykład dostarczany razem z produktami Borlanda (Delphi 7 umieszcza go w katalogu C:\Program Files\Borland\Delphi7\Demos\Threads).

W osobnym pliku (`SortThread.h`) napiszmy deklarację prostej klasy potomnej względem `TThread`:

```
#ifndef sorttype
#define sorttype double
#endif

#include <vcl/classes.hpp> //TThread
#include <vcl/stdctrls.hpp> //TListBox

enum sortmethods {smNone=-1, smBubble=0, smSelection, smInsert, smQuick};

class TSortThread : public TThread
{
public:
	TSortThread(sorttype*,int,bool);
	__fastcall ~TSortThread(void);
	sortmethods SortMethod;
	TListBox* OutputListBox;
	TColor OutputListBoxColor;
	void __fastcall Execute();
private:
	sorttype* Data;
	int DataAmount;
};
```

Klasa nie ma konstruktora domyślnego. Konstruktor przyjmuje trzy argumenty – wskaźnik do `sorttype` (tutaj zdefiniowane na stałe jako `double`, ale można zmienić np. na `int`), którym przekazywana jest tablica danych, ilość elementów w tablicy oraz wartość logiczna, która jest przekazywana do konstruktora klasy bazowej `TThread` (o jej znaczeniu poniżej).
Należy zwrócić uwagę na własność `SortMethod`. Jest ona zadeklarowana jako zmienna typu wyliczeniowego, która może przyjmować cztery możliwe wartości odpowiadające czterem metodom sortowania: `smBubble` (0), `smSelection` (1), `smInsert` (2), `smQuick` (3). Publiczną własnością jest również wskaźnik do TListBox, który może przechowywać adres obiektu `ListBox` aplikacji, w którym zapisany zostanie wynik (tj. posortowana tablica danych). Kreator nie sortuje danych, choćby dlatego, że użytkownik musi mieć możliwość przypisania `OutputListBox`. Klasa posiada osobną metodę publiczną `Execute()`, która wykonuje właściwe sortowanie. Prywatne własności klasy to wskaźnik do typu `sorttype`, który przechowywać będzie adres tablicy oraz wielkość tej tablicy.

[1] `TListBox` posiada oczywiście własność `Sorted`, która posortuje umieszczone w niej elementy metodą quicksort, ale z oczywistych względów tutaj z tego nie skorzystamy.

W pliku głównym (SortThread.cpp) musimy zdefiniować trzy metody naszej klasy, konstruktor, destruktor i Execute() [2]. Execute() jest metodą czysto wirtualną (wskaźnik do tej metody równy jest 0), co powoduje, że TThread jest klasą abstrakcyjną. Koniecznie musimy więc w naszej klasie potomnej napisać własną metodę Execute(), aby mógł powstać obiekt tej klasy.

Konstruktor wywołuje konstruktor metody bazowej przed rozpoczęciem wykonywania własnego kodu. Służy do tego konstrukcja

```
KlasaPotomna::KlasaPotomna(arg_konstr_pot):KlasaBazowa(arg_konstr_baz),
zmienna(wartość)
{
}
```

pozwalająca na wywołanie metody klasy bazowej lub potomnej, a także na inicjację własności klasy. W naszym przykładzie wskaźnik do tablicy danych ustawiany jest na NULL i inicjowany jest kolor ListBoxa. Następnie ustawiana jest własność klasy bazowej TThread->FreeOnTerminate polecająca zwolnienie pamięci zajmowaną przez obiekt natychmiast po zakończeniu metody Execute(). Jest to niezwykle wygodne, bo zwalnia nas od śledzenia niezależnego wątku z wątku pierwotnego aplikacji, tylko po to, żeby zastosować operator delete lub uruchomić metodę Free(). Zaniedbanie usunięcia z pamięci obiektu związanego z wątkiem powodowałoby wycieki pamięci (przynajmniej do zamknięcia całej aplikacji) związane z rezerwacją pamięci na dane, której dokonamy w konstruktorze klasy.

```
TSortThread::TSortThread(sorttype* OriginalData,
                          int _DataAmount,bool CreateSuspended)
:TThread(CreateSuspended),Data(NULL),OutputListBoxColor(clBtnFace)
{
//Ustawienia TThreat
FreeOnTerminate=true; //Usuwa z pamieci po zakonczeniu Execute

//Rezerwacja pamieci i kopiowanie danych
DataAmount=_DataAmount;
Data=(sorttype *)calloc(DataAmount,sizeof(sorttype));
for (int i=0; i<DataAmount; i++) Data[i]=OriginalData[i];
}
```

Po linii rezerwującej poleceniem C++ calloc blok pamięci i umieszczającej jego adres we wskaźniku Data kolejne polecenia kopiują dane do zarezerwowanego bloku pamięci. Własność DataAmount klasy przechowuje ilość elementów tablicy.

Destruktor zwalnia zajmowaną pamięć poleceniem C++ free():

```
__fastcall TSortThread::~TSortThread()
{
free(Data);
Data=NULL;
//ShowMessage("Wątek zakończył działanie("+(AnsiString) SortMethod+")");
}
```

Na końcu destruktora można umieścić wywołanie funkcji WinAPI ShowMessage z informacją o zakończeniu wątku. Wywołuje ona okno modalne, ale nie blokuje to działania aplikacji, ponieważ wszystkie działające procesy odbywają się w niezależnych wątkach.

Najistotniejsza ze względu na funkcjonalność naszej klasy jest oczywiście metoda Execute(). W zależności od ustalonej metody (przypominam, że przed uruchomieniem Execute(), a po stworzeniu klasy programista musi przypisać OutputListBox i ustalić metodę sortowania, tj. ustalić wartość SortMethod). wywoływana jest jedna z funkcji umieszczonych w osobnych plikach włączanych dyrektywami #include.

[2] Należy oczywiście pamiętać również o poleceniu #include "SortThread.h"

Ponieważ nie jest naszym celem nauka metod numerycznych potraktujemy te funkcje jak czarne skrzynki. Wszystkie przyjmują dwa argumenty (wskaźnik do tablicy z danymi i ilość danych)[3].

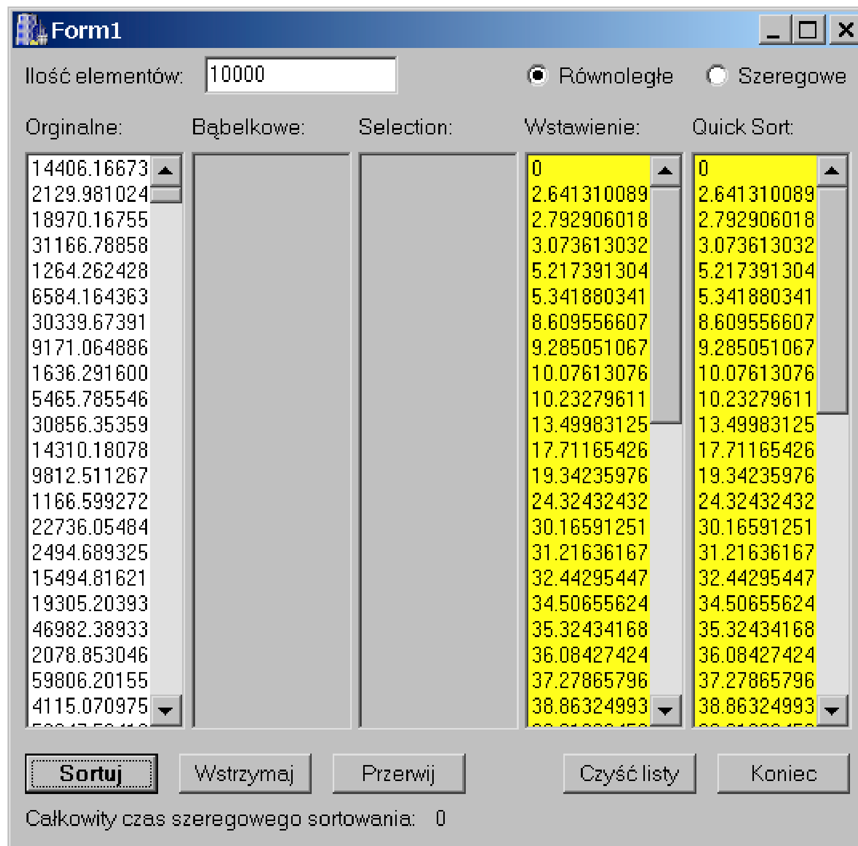
```
//Pliki zawierajace szblony funkcji sortujacych
#include "bubblesort.c"
#include "selectionsort.c"
#include "insertsort.c"
#include "quicksort.c"

void __fastcall TSortThread::Execute()
{
if (Data==0 || SortMethod==smNone) return;

//Sortowanie
switch (SortMethod)
      {
    case smBubble: bubblesort(Data,DataAmount); break;
    case smSelection: selectionsort(Data,DataAmount); break;
    case smInsert: insertsort(Data,DataAmount); break;
    case smQuick: quicksort(Data,DataAmount); break;
    }

//Zapisywanie wynikow
OutputListBox->Color=OutputListBoxColor;
OutputListBox->Items->Clear();
for (int i=0; i<DataAmount; i++) OutputListBox->Items->Add(Data[i]);
OutputListBox->Refresh();
}
```

Po wywołaniu metody sortującej, której wykonanie zajmuje większość czasu życia wątku wyniki mogą zostać zapisane są do `OutputListBox`. Jednak zajmuje to sporo czasu i zamiast kopiowania danych można poprzestać jedynie na poinformowaniu o zakończeniu przez zmianę koloru ListBox na żółty.

[3] Większość z tych funkcji jest w istocie szablonami. Można więc dość łatwo wybierać typ sortowanych danych. Pliki te znajdują się w źródłach, do których link znajduje się na stronie tytułowej tego skryptu.

2. Wywoływanie wątków

Stwórzmy projekt aplikacji z formą analogiczną do przedstawionej na rysunku. Umieśćmy na niej pięć komponentów `ListBox` (jeden na dane wejściowe i cztery na wyniki). Musimy też ustalać ilość elementów tablicy – przeznaczymy do tego komponent `TEdit` oraz przyciski jak na ilustracji.



Zanim zabierzemy się za sortowanie napiszmy metodę `ClearListBoxes()` czyszczącą ListBoxy. Najprościej będzie stworzyć tę metodę jako metodę zdarzeniową `Button3` „Czyść listy". Dla każdego ListBoxa powinna ustalać kolor na `clBtnFace` i wywoływać metodę `TListBox->Clear()`:

```
void __fastcall TForm1::ClearListBoxes(TObject *Sender)
{
ListBox1->Color=clBtnFace;
ListBox2->Color=clBtnFace;
ListBox3->Color=clBtnFace;
ListBox4->Color=clBtnFace;
ListBox5->Color=clBtnFace;
ListBox1->Clear();
ListBox2->Clear();
ListBox3->Clear();
ListBox4->Clear();
ListBox5->Clear();
}
```

Aby wygodnie kontrolować wątki wygodnie jest zadeklarować cztery wskaźniki do obiektów `TSortThreads` jako własności prywatne formy[4]. Umożliwi to nam wygodne operowanie metodami `TForm1` do ich kontroli. Wcześniej musimy oczywiście zadeklarować wykorzystanie tej klasy, a więc uwzględnić plik

[4] Ze względu na elegancję i wygodę programowania lepiej byłoby zadeklarować tablicę czterech wskaźników. Wiele czynności można by wówczas wykonać w pętlach. Jednak program straciłby wówczas na przejrzystości.

`SortThread.cpp` (`SortThread.h` jest podpinany z tego pierwszego). Na początku nagłówka należy więc dodać:

```
#include "SortThread.cpp"
```

a w sekcji `private`:

```
private:
      //Wskazniki do obiekty watkow
      TSortThread* BubbleSortThread;
      TSortThread* SelectionSortThread;
      TSortThread* InsertSortThread;
      TSortThread* QuickSortThread;
```

Ze względów bezpieczeństwa po deklaracji wskaźników powinniśmy mieć nawyk przypisania im „zerowych" adresów w konstruktorze formy:

```
__fastcall TForm1::TForm1(TComponent* Owner)
      : TForm(Owner)
{
BubbleSortThread=NULL;
SelectionSortThread=NULL;
InsertSortThread=NULL;
QuickSortThread=NULL;
}
```

Ustalamy typ sortowanych danych na double. Aby umożliwić sobie ewentualne zmiany zapiszmy `#define sorttype double`.

Tworzymy prywatny wskaźnik tablicy do przechowywania nieposortowanej listy liczb oraz liczby elementów w tej liście:

```
sorttype* Dane;
int Dane_ile;
```

One również powinny być zainicjowane w konstruktorze.

Do tworzenia danych przeznaczymy prywatną metodę `void __fastcall GenerujDane()` (należy ją oczywiście zadeklarować w nagłówku):

```
void __fastcall TForm1::GenerujDane()
{
//Czyszczenie ListBoxow
ClearListBoxes(this);

//Ile elementow (odczyt z Edit1)
Dane_ile=Edit1->Text.ToInt();
Dane=(sorttype *)calloc(Dane_ile,sizeof(sorttype));

//Tworzenie listy liczb pseudolosowych
randomize();
for (int i=0; i<Dane_ile; i++)
      {
      if (typeid(sorttype)==typeid(double))
Dane[i]=Dane_ile*random(Dane_ile)/(1.0+random(Dane_ile));
    if (typeid(sorttype)==typeid(int)) Dane[i]=random(Dane_ile);
    }

//Zapis liczb do ListBox1
ListBox1->Color=clWindow;
ListBox1->Items->Clear();
```

```
for (int i=0; i<Dane_ile; i++) ListBox1->Items->Add(Dane[i]);
ListBox1->Refresh();
}
```

Po wyczyszczeniu ListBoxów (wywołujemy stworzoną wcześniej metodę `ClearListBoxes()`) do `ListBox1` zapisywana jest pewna ilość liczb pseudolosowych. Ilość ta określona jest przez zawartość `Edit1->Text`[5]. Kopia tablicy tych liczb umieszczona jest też w pamięci o adresie przechowywanym we wskaźniku A (dostęp do kolejnych liczb jest identyczny jak w przypadku tablicy `[]`). Przy generowaniu danych sprawdzany jest typ kryjący się pod `sorttype`. Operator `typeid` wymaga dodania do projektu moduły typeinfo (`#include <typeinfo.h>`)

Mając dane możemy zabrać się za sortowanie. Napiszemy prywatną metodę `Sortowanie(bool)`, która utworzy obiekty TSortThread i uruchomi wątki.

Teraz tworzymy cztery obiekty wątków klasy TSortThread. Schemat jest dla każdej metody sortowania identyczny. Najpierw tworzymy obiekt klasy `TSortThread` o odpowiedniej nazwie. Ustalamy metodę sortowania (korzystając ze stałych zadeklarowanych w typie wyliczeniowym `sortmethods`) i przydzielamy ListBox do zapisania wyników. W przypadku pierwszej metody sortowania bąbelkowego wygląda to następująco:

```
BubbleSortThread=new TSortThread(Dane,Dane_ile,CreateSuspend);
BubbleSortThread->SortMethod=smBubble;
BubbleSortThread->OutputListBox=ListBox2;
```

Trzeci argument konstruktora `CreateSuspend`, przekazywany w kreatorze jako jedyny argument do konstruktora klasy bazowej TThread decyduje, czy metoda `Execute()` ma być wykonana natychmiast (tak jest jeżeli argument ten ma wartość `false`), czy po jej stworzeniu działanie wątku zawiesić (`true`). My poczekamy z uruchomieniem sortowania do czasu przygotowania wszystkich wątków i uruchomimy je równocześnie.

Podjęcie działania wątku (przy parametrze `CreateSuspend=true`) umożliwia metoda `Resume()`. Nie należy wywoływać metody `Execute()`, ponieważ spowoduje to tylko wykonanie kodu zawartego w samej metodzie `Execute()` i odbędzie się to w głównym wątku aplikacji. Przerwanie działania wątku spowoduje metoda `Suspend()`.

Metodę `Execute` można wykorzystać do wymuszenia szeregowego sortowania po kolei każdą metodą. Wówczas dziedziczenie klasy `TSortThread` z klasy `TThread` nie jest w ogóle wykorzystane.

```
void __fastcall TForm1::Sortowanie(bool rownolegle)
{
//Sprawdzenie czy dane wejsciowe istnieja
if (Dane==NULL)
      {
      ShowMessage("Brak danych");
      return;
      }

//Sprawdzenie czy poprzednie sortowanie zostalo zakonczone
if (BubbleSortThread!=NULL || SelectionSortThread!=NULL ||
    InsertSortThread!=NULL || QuickSortThread!=NULL)
      {
        ShowMessage("Wątki nadal działają!");
        return;
        }

//Tworzenie watkow
bool CreateSuspend=true;
```

[5] Aby odczytać liczbę z tekstu umieszczonego w `Edit1` najlepiej skorzystać z metody `AnsiString.ToInt()`, a więc `Edit1->Text.ToInt()`.

```
BubbleSortThread=new TSortThread(Dane,Dane_ile,CreateSuspend);
BubbleSortThread->SortMethod=smBubble;
BubbleSortThread->OutputListBox=ListBox2;

SelectionSortThread=new TSortThread(Dane,Dane_ile,CreateSuspend);
SelectionSortThread->SortMethod=smSelection;
SelectionSortThread->OutputListBox=ListBox3;

InsertSortThread=new TSortThread(Dane,Dane_ile,CreateSuspend);
InsertSortThread->SortMethod=smInsert;
InsertSortThread->OutputListBox=ListBox4;

QuickSortThread=new TSortThread(Dane,Dane_ile,CreateSuspend);
QuickSortThread->SortMethod=smQuick;
QuickSortThread->OutputListBox=ListBox5;

//Ustalanie koloru w zaleznosci od trybu (rownolegly/szeregowy)
TColor kolor=clLime;
if (rownolegle) kolor=clYellow;
BubbleSortThread->OutputListBoxColor=
SelectionSortThread->OutputListBoxColor=
InsertSortThread->OutputListBoxColor=
QuickSortThread->OutputListBoxColor=kolor;

if (rownolegle)
      {
      //Uruchamianie watkow rownolegle
      BubbleSortThread->Resume();
      SelectionSortThread->Resume();
      InsertSortThread->Resume();
      QuickSortThread->Resume();
      }
      else
      {
      //Uruchamianie watkow szeregowe
      Screen->Cursor=crHourGlass; //kursor=klepsydra
      time_t starttime,stoptime;
      time(&starttime); //pomiar czasu poczatkowego

      BubbleSortThread->Execute(); delete BubbleSortThread; BubbleSortThread=NULL;
      SelectionSortThread->Execute(); delete SelectionSortThread; SelectionSortThread=NULL;
      InsertSortThread->Execute(); delete InsertSortThread; InsertSortThread=NULL;
      QuickSortThread->Execute();  delete QuickSortThread; QuickSortThread=NULL;

      time(&stoptime); //pomiar czasu koncowego
      Label9->Caption=(AnsiString)(double)(stoptime-starttime);
      Screen->Cursor=crDefault; //przywrocenie domyslnego kursora
      }

}
```

Funkcja `time()` i typ `time_t` wymaga dodania nagłówka `time.h` (`#include <time.h>`).

Możemy wreszcie stworzyć metodę zdarzeniową do klawisza `Button1` „Sortuj”:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
GenerujDane();
Sortowanie(RadioButton1->Checked);
}
```

Proste jest obsłużenie klawisza `Button2` „Wstrzymaj" zawieszającego działanie wątków:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
//Sprawdzenie czy obiekty watkow istnieja
if (BubbleSortThread==NULL || SelectionSortThread==NULL ||
    InsertSortThread==NULL || QuickSortThread==NULL)
      {
        ShowMessage("Obiekty watkow nie istnieja!");
        return;
        }

//Zawieszenie/Wznawianie dzialania watkow
BubbleSortThread->Suspended=!BubbleSortThread->Suspended;
SelectionSortThread->Suspended=!SelectionSortThread->Suspended;
InsertSortThread->Suspended=!InsertSortThread->Suspended;
QuickSortThread->Suspended=!QuickSortThread->Suspended;
}
```

i klawisza Button3 „Przerwij" przerywającego działanie wątków:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
//Sprawdzenie czy obiekty watkow istnieja
if (BubbleSortThread==NULL || SelectionSortThread==NULL ||
    InsertSortThread==NULL || QuickSortThread==NULL)
      {
        ShowMessage("Obiekty watkow nie istnieja!");
        return;
        }

//Przerywanie dzialania watkow
BubbleSortThread->Terminate();
SelectionSortThread->Terminate();
InsertSortThread->Terminate();
QuickSortThread->Terminate();
}
```

To ostatnie jednak nie zadziała, jeżeli metoda `TSortThread::Execute()` nie będzie sprawdzać własności `TThread::Terminated`.

Pomimo braku możliwości przerwania działania wątków możemy już sprawdzić działanie naszej aplikacji. Za pomocą Menadżera zadań Windows można sprawdzić ilość wątków po uruchomieniu programu i po naciśnięciu przycisku „Sortuj" (na szybszych komputerach niezbędne będzie też naciśnięcie przycisku „Wstrzymaj").

Uwaga!
Aplikacja posiada pewien błąd. Po zakończeniu sortowania i usunięciu wątków wskaźniki nie są czyszczone co powoduje, że nie można wywołać sortowania jeszcze raz. Błąd ten poprawimy w kolejnych paragrafach.

Zadanie
Dodać przycisk „Wznów" wznawiający pracę wątku po wstrzymaniu (metoda `Resume()`)

Zadanie
Wątki poboczne mogą modyfikować obiekty formy głównej jeżeli przekaże się im wskaźniki do tych obiektów. Jest jeszcze jeden sposób – można wykorzystać metodę `TThread.Synchronize(TThreadMethod)`, która wywołuje zdefiniowaną przez nas metodę wątku `TSortThread` bez argumentów w kontekście wątku głównego (deklaracja powinna być następująca: `void __fastcall SetListBoxItems(void);`). Niezbędne jest zdefiniowanie tej metody i umieszczenie w niej kodu odwołującego się bezpośrednio do ListBoxów umieszczonych na `Form1` (należy dodać `#include "Unit1.h"`), nawet kosztem elegancji powyższego rozwiązania. Wreszcie w końcu metody `Execute` należy zastąpić zapisywanie posortowanych

wyników wywołaniem `Synchronize(SetListBoxItems);`. Szczegóły można znaleźć w dołączonych do skryptu źródłach.

3. Sprawdzanie własności Terminated

Wywołanie `TThread::Terminate()` nie powoduje natychmiastowego zakończenia wątku (taką możliwość daje opisana w następnym rozdziale funkcja WinAPI `TerminateThread()`, ale nie jest to sposób godny polecenia). Powoduje ona tylko ustawienie własności `Terminated` na `true`. Programista musi sam zadbać o to, żeby w metodzie `Execute()` wartość ta była sprawdzana i sam musi zadbać o kończenie wątku. Daje mu to możliwość zwolnienia zasobów i wykonania niezbędnych działań lub zignorowania wezwania do zakończenia wątku.

Chcąc zmodyfikować `TSortThread` tak, aby badała własność `Terminated` musimy funkcje sortujące uczynić metodami klasy i w najgłębiej zagnieżdżonej pętli dodać warunek `if (Terminated) Free();`. Zadeklarujmy metody prywatne w pliku nagłówkowym `TSortThread.h`:

```
void bubblesort(sorttype*,int const);
void selectionsort(sorttype*,int const);
void insertsort(sorttype*,int const);
```

(modyfikację quicksort pozostawiam jako ćwiczenie – zob. opis zadania na końcu paragrafu) i dodajmy odpowiednie implementacje w pliku `TSortThread.cpp`:

```
//Sortowanie babelkowe
void TSortThread::bubblesort(sorttype* A, int const A_ile)
{
  for (int i=A_ile; i>=0; i--)
    for (int j=0; j<i-1; j++)
      if (A[j]>A[j+1])
      {
      sorttype tmp=A[j];
      A[j]=A[j+1];
      A[j+1]=tmp;
      if (Terminated) Free(); //reakcja na Terminate()
      }
return;
}

//Metoda wyboru
void TSortThread::selectionsort(sorttype* A, int const A_ilel)
{
  for (int i=0; i<A_ilel; i++)
    for (int j=A_ilel-1; j>i; j--)
      if (A[i]>A[j])
            {
        sorttype tmp=A[i]; A[i]=A[j]; A[j]=tmp;
        if (Terminated) Free(); //reakcja na Terminate()
        }
}

//Sortowanie przez wstawianie
void TSortThread::insertsort(sorttype* A, int const A_ile)
{
  for (int i=0; i<=A_ile-1; i++)
      {
      sorttype tmp=A[i];
    int j=i;
    while(j>0 && A[j-1]>=tmp)
      {
```

```
        A[j]=A[j-1];
        --j;
        if (Terminated) Free(); //reakcja na Terminate()
        }
    A[j]=tmp;
    }
}
```

Trzy zmodyfikowane wątki sprawdzają warunek na tyle często, że ich reakcja na wezwanie do przerwania pracy jest natychmiastowa.

Zadanie
Zmodyfikować także metodę szybkiego sortowania. Należy zauważyć, że umieszczona w pliku `quicksort.c` funkcja sortująca korzysta ze standardowej funkcji C++ `qsort` (biblioteka `stdlib.h`). Warunek sprawdzający wartość `Terminated` można wstawić tylko do funkcji porównującej co jest możliwe jedynie wtedy, jeśli byłaby ona metodą obiektu. Należy wobec tego wykonać rzutowanie wskaźnika do metody na wskaźnik do funkcji.

Zadanie
Przygotować szablon klasy `TSortThread`, w którym parametrem jest typ sortowanych liczb.

4. Zdarzenie TThread.OnTerminate

Aby zmierzyć czas w jakim wątki działały równolegle wykorzystamy zdarzenie `OnTerminate` wywoływane po zakończeniu metody `TThread::Execute()`. Zadeklarujmy globalną prywatną zmienną typu `time_t` o nazwie `parallel_starttime`. W metodzie `TForm1::Sortowanie()` przed wywołaniem metod Resume() dla poszczególnych wątków (ma to miejsce, jeżeli sortowanie przebiega równolegle) zmierzmy czas poleceniem `time(¶llel_starttime);`.

Zadeklarujmy takżę metodę `void __fastcall ThreadTerminate(TObject* Sender);`, która jest zgodna z typem `TNotifyEvent`, a więc będzie mogła pełnić rolę metody zdarzeniowej. Metodę tą zdefiniujmy następująco:

```
void __fastcall TForm1::ThreadTerminate(TObject* Sender)
{
//Zostanie zapis z ostatniego
time_t stoptime;
time(&stoptime);
Label10->Caption=(AnsiString)(double)(stoptime-parallel_starttime);
}
```

W miejscu gdzie pobraliśmy czas parallel_starttime musimy teraz przypisać do zdarzeń OnTerminate każdego z wątków powyższą metodę:

```
BubbleSortThread->OnTerminate=ThreadTerminate;
SelectionSortThread->OnTerminate=ThreadTerminate;
InsertSortThread->OnTerminate=ThreadTerminate;
QuickSortThread->OnTerminate=ThreadTerminate;
```

W ten sposób po zakończeniu każdego wątku do `Label10` zapisywana będzie ilość sekund jaka upłynęła od pobrania parallel_starttime. Po zakończeniu wszystkich wątków w zmiennej zostanie czas dla najdłużej trwającego wątku.

Korzystając z tego zdarzenia można też sprawdzić czy wszystkie cztery wątki zostały zakończone i ustalić wartość wskaźników na NULL. Obiekty zostały już usunięte z pamięci dzięki ustaleniu wartości `FreeOnTerminate` na `true`, ale wskaźniki nadal zawierają ich adresy. Ponadto metody `Sortowanie()`, `Button2Click()` i `Button3Click()` po wartości wskaźników rozpoznają istnienie obiektów. Dodajmy do metody zdarzeniowej odpowiednio obsługiwany licznik:

```
static licznik=0;
licznik++;
if (licznik==4)
      {
      BubbleSortThread=NULL;
      SelectionSortThread=NULL;
      InsertSortThread=NULL;
      QuickSortThread=NULL;
      licznik=0;
      }
```

5. Zmiana priorytetu wątku

Podobnie jak można sterować priorytetem procesu (zob. III.3 w części WinAPI) można także modyfikować priorytet wątku. Z tym, że tym razem mamy wsparcie w postaci własności klasy `TThread` i w odróżnieniu od czterostopniowej skali dla procesów wątki mogą przyjmować jeden z siedmiu priorytetów.

Przed uruchomieniem wątków w metodzie `Sortowanie()` dodajmy ustalenie różnych priorytetów. Aby wyraźnie zobaczyć efekt modyfikacji najlepiej ustalić najniższy priorytet dla najszybszej metody i najwyższy dla najwolniejszej:

```
BubbleSortThread->Priority=tpHighest;
QuickSortThread->Priority=tpLowest;
```

III. Kontrola wątku za pomocą funkcji WinAPI

1. Wymuszenie zakończenia wątku – TerminateThread

Aby zamknąć wątek bez „pytania go o zdanie” można wykorzystać funkcję WinAPI `TerminateThread()`. Aby wskazać na właściwy wątek można wykorzystać własność `TThread::Handle`. Należy pamiętać, że „zabity” w ten sposób wątek nie zadba o siebie. Na programiście pozostanie obowiązek zwolnienia zajmowanej przez obiekt pamięci.

```
void __fastcall TForm1::Button6Click(TObject *Sender)
{
//Sprawdzenie czy obiekty watkow istnieja
if (BubbleSortThread==NULL || SelectionSortThread==NULL ||
    InsertSortThread==NULL || QuickSortThread==NULL)
      {
      ShowMessage("Obiekty watkow nie istnieja!");
      return;
      }

//Zabijanie wszystkich watkow
TerminateThread((void*)BubbleSortThread->Handle,0);
TerminateThread((void*)SelectionSortThread->Handle,0);
TerminateThread((void*)InsertSortThread->Handle,0);
TerminateThread((void*)QuickSortThread->Handle,0);

//Sami musimy zadbac o usuniecie obiektow
delete BubbleSortThread; BubbleSortThread=NULL;
delete SelectionSortThread; SelectionSortThread=NULL;
delete InsertSortThread; InsertSortThread=NULL;
delete QuickSortThread; QuickSortThread=NULL;
}
```

2. Synchronizacja wątku – sekcje krytyczne

Zasadniczymi problemami programowania współbieżnego są zakresy zmiennych (które powinny być lokalne w wątku, a które dzielone) oraz wydzielenie fragmentów kodu, które nie powinny być wykonywane równocześnie (np. przy zapisie do pliku). O zasięg zmiennych musimy zadbać wybierając miejsce ich deklaracji. W kontroli i synchronizacji wątków wspomaga nas natomiast WinAPI[6].

Najczęściej wykorzystywanym narzędziem są sekcje krytyczne (struktura WinAPI `CRITICAL_SECTION`). Ich zadaniem jest pilnowanie, aby fragment kodu nie był wykonywany równolegle przez wiele wątków. Wejście do sekcji krytycznej (`EnterCriticalSection()`) przez wątek uniemożliwia wejście do niej przez inny wątek do czasu opuszczenia sekcji przez pierwszy (`LeaveCriticalSection()`).

Sekcja krytyczna musi być zadeklarowana globalnie, aby była dostępna zarówno dla macierzystej aplikacji (wątku głównego) i jej wszystkich wątków pobocznych. Dodajmy więc w głównym pliku przed włączeniem nagłówka deklarację `CRITICAL_SECTION CrSec;`.
Sekcja krytyczna musi być zainicjowana przed użyciem najlepiej w wątku głównym. Dodajmy w metodzie `Button1Click()` przed wygenerowaniem danych odpowiednią inicjację:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
//Inicjacja sekcji krytycznej
InitializeCriticalSection(&CrSec);
```

[6] **Uwaga!** W nowszych wersjach C++ Buildera i Delphi znajduje się obiekt `TCriticalSection`, który może uprościć realizację synchronizacji wątków sekcjami krytycznymi.

```
GenerujDane();
Sortowanie(RadioButton1->Checked);
}
```

Od początku zadbajmy o jego usunięcie po wykonaniu wszystkich wątków. Na szczęście mamy już odpowiednio przygotowane miejsce w metodzie zdarzeniowej `TThread::OnTerminate` tam, gdzie zerowaliśmy wskaźniki wątków:

```
void __fastcall TForm1::ThreadTerminate(TObject* Sender)
{
//Zostanie zapis z ostatniego
time_t stoptime;
time(&stoptime);
Label10->Caption=(AnsiString)(double)(stoptime-parallel_starttime);

static licznik=0;
licznik++;
if (licznik==4)
      {
      BubbleSortThread=NULL;
      SelectionSortThread=NULL;
      InsertSortThread=NULL;
      QuickSortThread=NULL;
      licznik=0;

      //Usuniecie sekcji krytycznej
      DeleteCriticalSection(&CrSec);
      }
}
```

Uruchomienie i opuszczenie sekcji krytycznej zależy tylko od programisty. Dla przykładu wymuśmy szeregowe wykonanie wątków. W tym celu najrozsądniej jest umieścić wejście do sekcji krytcznej na samym początku metody `Execute()`:

```
void __fastcall TSortThread::Execute()
{
if (Data==0 || SortMethod==smNone) return;

//Wejscie do sekcji krytycznej
EnterCriticalSection(&CrSec);
ShowMessage("Wejście w sekcję krytyczną ("+(AnsiString)SortMethod+")");

//Sortowanie

...
```

Wyjście z sekcji krytycznej powinno odbyć się na samym końcu działania wątku tj. polecenie powinno znaleźć się w ostatniej linii destruktora:

```
__fastcall TSortThread::~TSortThread()
{
free(Data);
Data=NULL;
//ShowMessage("Wątek zakończył działanie ("+(AnsiString)SortMethod+")");

//Opuszczenie sekcji krytycznej
ShowMessage("Wyjście w sekcji krytycznej ("+(AnsiString)SortMethod+")");
LeaveCriticalSection(&CrSec); //To musi byc na samym koncu dzialania watku
}
```

Do synchronizacji wątków można posłużyć się również mutexem, który jest bardzo podobny do sekcji krytycznej z tą różnicą, że zasięg jego działania jest globalny w całym systemie, a nie tylko w obrębie procesu (tj działa tak jak sekcje krytyczne dla wielu równocześnie uruchomionych kopii aplikacji). Podobnie semafor, który tym różni się od mutexu, że dopuszcza zadaną ilość procesów współdzielących wykonywany fragment kodu.