МЕТОДИЧЕСКИЕ УКАЗАНИЯ к курсовой по дисциплине «ОПЕРАЦИОННЫЕ СИСТЕМЫ» для студентов 3-го курса специальности 230102 АСОИУ

ИСПОЛЬЗОВАНИЕ ПОТОКОВ В ПРИЛОЖЕНИЯХ WINDOWS

Введение

Часто разработчику приходится сталкиваться с проблемой, когда необходимо одновременное выполнение нескольких задач одного приложения. Для решения этой и других проблем разработчику на Delphi предоставлены в распоряжение средства, позволяющие реализовать так называемую многопоточностъ. Многопоточность используется для:

* обхода медленных процессов. Когда используется только один поток, приложение может приостановить свое выполнение на то время, пока им завершается какой-либо медленный процесс (доступ к диску, связь с другим компьютером по сети и т. д.). Центральный процессор компьютера в данный момент находится в режиме ожидания и практически не выполняет никаких команд. С использованием многопоточности ваше приложение может продолжать выполнение других потоков, пока один из потоков ожидает завершение медленного процесса;
* организации поведения приложения. Благодаря использованию потоков, вы можете организовать выполнение частей приложения так, как вам захочется. Например, вы можете для каждой задачи приложения (если каждой задаче выделен свой поток) распределить приоритеты выполнения. Таким образом, задача, имеющая наибольший приоритет, будет занимать больше процессорного времени, что очень важно для решения критических задач;
* поддержки мультипроцессорной обработки. Если в компьютере, на котором запущено многопоточное приложение, имеется несколько процессоров, то можно значительно увеличить скорость выполнения вашего приложения, направляя на каждый процессор свой поток.

Общий обзор потоков

Поток (Thread) - это объект операционной системы, заключенный в процесс и реализующий какую-либо задачу. Каждое приложение (процесс) Win32 имеет по крайней мере один поток, который называется главным (основным, стандартным). Каждый процесс может содержать несколько потоков.

Возможность введения многопоточности появилось с приходом вытесняющей многозадачности. В ранних, 16-разрядных версиях Windows использовалась кооперативная многозадачность.

Кооперативная многозадачность - поддержка "параллельной" работы нескольких приложений операционной системы, при которой передача управления операционной системе происходит от приложения. Данный вид многозадачности оказался весьма неэффективным. В случае "зависания" одного приложения, "зависала" вся система.

Вытесняющая многозадачность - вид многозадачности, когда управления потоками возложено полностью на операционную систему. Операционная система распределяет процессорное время для каждого потока в зависимости от его приоритетов. При этом, если зависает один из потоков, операционная система продолжает выделять процессорное время для выполнения других потоков.

В большинстве приложений вы можете использовать объект потока, который позволяет вам использовать потоки в ваших приложениях. Объекты потоков инкапсулируют в себе основные свойства и методы, необходимые для написания многопоточных приложений. Итак, любой поток - это объект, получающий определенное процессорное время. Всякое приложение Windows является процессом операционной системы. Каждый процесс состоит хотя бы из одного потока, который называется главным. Вообще, Windows не ограничивает число потоков для каждого процесса.

Для того чтобы использовать объекты потоков в вашем приложении, вам нужно создать потомок класса TThread. Класс TThread был создан для облегчения написания приложений с несколькими потоками. Он гарантирует совместимость при работе с библиотекой визуальных компонентов (VCL) Delphi. Вообще, при создании многопоточных приложений необходимо следовать приведенным ниже рекомендациям:

* остерегайтесь создавать слишком много потоков - это может привести к большой загруженности операционной системы и процессора. Рекомендуемое ограничение числа активных потоков в одном процессе - 16 (для однопроцессорной системы);
* используйте синхронизацию в случае, когда несколько потоков пытаются получить доступ к одному ресурсу;
* большинство методов, которые обращаются к объектам VCL и изменяют содержимое формы, должны вызываться из главного VCL-потока или использовать объект синхронизации, такой как TMultiReadExclusive-WriteSynchronizer.

Определение объекта TThread находится в модуле classes и имеет следующий вид (листинг 1): Листинг 1

TThread = class private

FHandle: THandle; FThreadID: THandle; FTerminated: Boolean; FSuspended: Boolean; FFreeOnTerminate: Boolean; FFinished: Boolean; FReturnValue: Integer; FOnTerminate: TNotifyEvent; Method: TThreadMethod; FSynchronizeException: TObject; procedure CallOnTerminate; function GetPriority: TThreadPriority; procedure SetPriority(Value: TThreadPriority); procedure SetSuspended(Value: Boolean); protected procedure DoTerminate; virtual; procedure Execute; virtual; abstract; procedure Synchronize(Method: TThreadMethod);

property ReturnValue: Integer read FReturnValue write FReturnValue;

property Terminated: Boolean read FTerminated;

public

constructor Create(CreateSuspended: Boolean);

destructor Destroy; override;

procedure Resume;

procedure Suspend;

procedure Terminate;

function WaitFor: LongWord;

property FreeOnTerminate: Boolean read FFreeOnTerminate write FFreeOnTerminate;

property Handle: THandle read FHandle;

property Priority: TThreadPriority read GetPriority write SetPriority; property Suspended: Boolean read FSuspended write SetSuspended; property ThreadID: THandle read FThreadID; property OnTerminate: TNotifyEvent read FOnTerminate write FQnTerminate; end;

Из вышеприведенного листинга можно определить, что объект TThread является прямым потомком объекта TObject, следовательно, он не является визуальным компонентом. Его метод Execute - абстрактный, а значит, сам объект TThread тоже является абстрактным, и вы не сможете создать экземпляр этого класса. Таким образом, вам придется определять классы- потомки данного класса для работы с потоками.

Для создания потомка класса TThread выберите в главном меню Delphi команду File/New (Файл/Новый), затем в появившемся окне щелкните на Thread Object (Объект потока). В открывшемся диалоговом окне напишите имя для вашего нового объекта потока. После всего этого Delphi создаст новый модуль и в окне редактора кода появится новая вкладка.

Если вы проделали все вышеописанное и назвали новый объект потока TMyThread, то в новом модуле, сгенерированном Delphi, вы можете увидеть код - заготовку для вашего нового объекта потока (листинг 2). Листинг 2

unit Unit2;

interface

uses

Classes; type

TMyThread = class(TThread)

private

{ Private declarations } protected

procedure Execute; override; end;

implementation { TMyThread } procedure TMyThread.Execute; begin

{ Place thread code here } end; end.

В автоматически сгенерированном файле модуля вы можете:

* инициализировать поток;
* заполнить метод Execute объекта потока, разместив там функции и процедуры;
* написать код гарантированного разрушения потока (например, строку FreeOnTerminate:=True;).

Инициализация потоков

Если хотите написать код инициализации для вашего нового объекта потока, вам необходимо добавить новый конструктор в описание вашего нового класса потока, после чего вы можете добавлять код инициализации вместо

кода реализации класса. Здесь вы можете также указать, какой приоритет вы устанавливаете для данного потока, как должен вести себя данный поток по завершении своей работы.

Приоритеты потоков

Приоритеты указывают операционной системе, сколько процессорного времени выделяется для данного потока. Для критических задач можно установить наивысший приоритет, для менее значимых - более низкий приоритет. Приоритет каждого потока складывается из двух составляющих: - класса приоритета - приоритета процесса, породившего поток; - относительного приоритета - приоритета самого потока.

Класс приоритета процесса может принимать одно из четырех значений: Idle, Normal, High и Realtime, которые выражаются числовыми значениями от 4 до 24. По умолчанию, любое приложение получает приоритет Normal. В табл. 1 представлены классы приоритетов процессов.

Таблица 1. Классы приоритетов процессов

|  |  |  |
| --- | --- | --- |
| Значение класса | Приоритет | Числовое значение |
| Idle | Низший приоритет для выполнения фоновых задач | 4 |
| Normal | Стандартный приоритет, который имеют большинство приложений Windows | 7-9 |
| High | Приоритет высокого уровня, приложение получает больше процессорного времени, чем имеющее класс Normal | 13 |
| Reaitime | Наивысший уровень приоритета | 24 |

Для определения текущего и установки требуемого класса приоритета используются функции GetPrioriryClass И SetPriorityClass соответственно. Для установки высокого класса

приоритета (High) для вашего приложения можно, например, использовать следующий код: If not SetPriorityClass(GetCurrentProcess, HIGH\_PRIORITY\_CLASS) then ShowMessage ('Ошибка установки класса приоритета').

Для установки значения относительного приоритета у потоков имеется свойство Priority. Данное свойство может принимать семь значений. Используется как бы шкала значений приоритетов от самого низкого до наивысшего. Значения приоритетов потоков представлены в табл. 2.

Таблица 2. Приоритеты потоков

|  |  |  |
| --- | --- | --- |
| Значение  относительного  приоритета | Приоритет | Числовое значение |
| TpIdle | Данный поток выполняется, когда система не занята и не выполняются никакие другие потоки. Windows не будет прекращать работу других потоковдля выполнения потока, имеющего приоритет tpldle | -15 |
| TpLowest | Низший приоритет выполнения. Данный поток занимает минимум процессорного времени | -2 |
| TpLower | Низкий приоритет. Данный потокзанимает немного больше процессорного времени, чем имеющий приоритет tpLowest | -1 |
| TpNormal | Нормальный приоритет. Все потоки по умолчанию имеют приоритет tpNormal | 0 |
| TpHigher | Высокий приоритет. Данный поток имеет приоритет выше нормального | 1 |
| TpHighest | Высший приоритет. Данный поток имеет приоритет выше, чем tpHigher | 2 |
| TpTimeCritical | Наивысший приоритет. Поток с данным приоритетом занимает максимум процессорного времени | 15 |

Использование высших и наивысших приоритетов может привести к замедлению работы других потоков. Применение данных видов приоритетов целесообразно использовать в случае, когда возникает острая необходимость скорейшего выполнения одного из процессов. Приведенный ниже пример показывает конструктор потока с самым низшим приоритетом, предназначенным для решения фоновых задач, которые не должны влиять на выполнение основных процессов:

constructor TMyThread.Create(CreateSuspended: Boolean); {

inheritedCreate(CreateSuspended);

Priority := tpldle; }

Поведение потока при завершении его работы

Обычно при завершении своей работы поток просто освобождается. Однако иногда бывает необходимо, чтобы завершение работы и освобождение потока было согласовано с другими потоками. Например, вы можете ожидать какое-либо значение, возвращаемое одним потоком перед выполнением другого потока. Для реализации этого, вы не должны освобождать первый поток, пока второй не получит значение, возвращаемое первым. Для того чтобы управлять завершением работы потока, имеется свойство потока FreeOnTerminate: По умолчанию данное свойство установлено в true. При этом поток освобождается по завершении своей работы. Если же установить данное свойство в false, то вы можете сами явно завершить работу потока.

Кроме того, в Delphi имеется возможность прекращения выполнения одного потока из другого потока подачей команды о прекращении. Когда один поток пытается прекратить работу другого потока, он вызывает метод Terminate. В результате свойство Terminate потока будет установлено в true, что можно проверить во время выполнения метода Execute: procedure TMyThread.Execute;

begin

while not Terminated do {выполнять какие-либо задачи}; end;

Пример создания многопоточного приложения в Delphi

Мы создадим простое приложение, которое состоит из трех потоков. Для начала запустим Delphi и выберем в главном меню Delphi пункт File/New Application (Файл/Новое приложение). После чего, разместим на главной форме приложения (Forml) поле для редактирования (Edit), индикатор хода выполнения работы (Progress Bar) и системный таймер (Timer).

Теперь добавим новый объект потока через пункт главного меню Delphi File/New/Thread Object (Файл/Новый/Объект потока). Введем имя для нового потомка класса TThread, например TMyThreadl. Delphi автоматически добавит модуль Unit2 в наше приложение. В описании объекта TMyThreadl добавим раздел public, в котором опишем глобальную переменную count. В результате должно получиться следующее: type

TMyThreadl = class(TThread) private

{ Private declarations } protected

procedure Execute; override; public

count:integer; end;

Далее запишем в метод Execute объекта TMyThreadl код, который должен выполняться в потоке. Пусть это будет код, который генерирует случайные числа и присваивает их глобальной переменной count. Для того чтобы генерация случайных чисел была бесконечной, зациклим ее с помощью так называемого "бесконечного цикла":

procedure TMyThreadl.Execute; begin

while true do begin

count:=random(maxint);

end;

end;

Теперь создадим второй объект потока, который должен заполнять индикатор хода работы (Progress Bar). По аналогии с первым объектом потока, при помощи главного меню Delphi создадим объект потока с именем TMyThread2. Во вновь добавленном модуле units определим глобальную переменную prcount:

type

TMyThread2 = class(TThread) private

{ Private declarations } protected

procedure Execute; override; public

prcount:integer; end;

После чего в процедуре Execute объекта TMyThread2 запишем следующий код, также зациклив его:

procedure TMyThread2.Execute; begin

while true do begin

prcount:=prcount+l;

if prcount>100 then prcount:=0;

end;

end;

Теперь сохраним наш проект и модули под теми именами, которые нам предложит Delphi (Projectl, Unitl, Unit2, Unit3). Добавим в модуле uniti в описание класса формы в разделе private объекты потоков Threadl - потомок класса TMyThreadl и Thread2 - потомок класса TMyThread2:

type

TForml = class(TForm) ProgressBarl: TProgressBar; Editl: TEdit; Timerl: TTimer;

procedure FormCreate(Sender: TObject); private

Threadl:TMyThreadl; Thread2:TMyThread2; public

{ Public declarations } end;

Далее, дважды щелкнем на любом свободном пространстве формы Forml. При этом откроется "заготовка" для метода создания формы FormCreate формы Forml. В обработчике FormCreate напишем следующий код:

procedure TForml.FormCreate(Sender: TObject); begin

Threadl:=TMyThreadl.Create(False);

Threadl.priority:=tpLowest;

Thread2:=TMyThread2.Create(False);

Thread2.priority:=tpNormal;

end;

Здесь мы создаем объект потока с использованием конструктора Create и устанавливаем приоритеты потоков (tpLowest и tpNormal).

Единственный параметр, который передается в методе Create класса TThread- CreateSuspended. Данный параметр может принимать логическое значение истины или лжи (true, false). Этот параметр показывает, создавать поток в приостановленном состоянии (true) или запускать его сразу же (false). То есть метод Execute будет вызван автоматически сразу, если параметр CreateSuspended равен false (как в нашем случае). Если же мы хотим запускать поток на исполнения в другое время, необходимо установить параметр CreateSuspended в true и воспользоваться методом Resume для вызова метода Execute на исполнение. Запишем в блоке uses модуля unitl используемые модули unit2 и unit3:

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Unit2, Unit3, StdCtrls, ComCtrls;

Нам осталось отразить результаты вычислений, производимых потоками на форме. Для этого создадим обработчик события OnTimer, дважды щелкнув на объекте Timerl. Запишем в обработчик события OnTimer следующий код: procedure TForml.TimerlTimer(Sender: TObject); begin

editl.text:=inttostr(threadl.count);

progressbarl.position:=thread2.prcount;

end;

Приложение, которое использует потоки, готово! Теперь, если запустить приложение с помощью главного меню Delphi: Run/Run (Запуск/Запуск), то мы можем видеть, как два созданных нами потока работают, причем с разной скоростью (в зависимости от приоритета).

Определение времени, занимаемого потоком

Кроме рассмотренных выше методов активизации и завершения потоков, используется то обстоятельство, что потоки могут приостанавливать и восстанавливать свою работу. Для этого используются методы Suspend и Resume. Под управлением не многопоточной операционной системы определить количество времени, требуемого на выполнение каких-либо вычислений, достаточно просто. Для этого можно использовать функцию GetTickCount. Например, чтобы определить, сколько времени требуется для генерации случайного числа в Delphi, достаточно написания следующего кода: Var

StartTime, Summa: longint;

R: integer;

begin

StartTime:=GetTickCount; R:=Random(MaxInt) ; Summa:=GetTickCount - StartTime; end;

В многопоточной операционной системе узнать время, необходимое для выполнения каких-либо операций, труднее, т. к. вычисления вашего потока могут быть прерваны в любом месте для выполнения других потоков. Для этого в по-настоящему многопоточных операционных системах (например, Windows NT) имеется функция GetThreadTimes, которая предоставляет информацию о времени работы потока:

function GetThreadTimes (hThread: THandle; var IpCreationTime, IpExitTime, IpKernelTime, IpUserTime: TFileTime): BOOL; stdcall; Рассмотрим параметры, передаваемые в данную функцию:

* hThread - название потока, для которого определяется время выполнения;
* IpCreationTime - время создания потока;
* IpExitTime - время завершения работы потока (определяется только при завершении работы потока);
* lpKernelTime - время, затраченное операционной системой для выполнения собственного кода;
* lpUserTime - время, затраченное на выполнение приложений пользователя. Как.мы видим, последние четыре параметра имеют тип TFiieTime:

type

TFiieTime = record dwLowDateTime: DWORD; dwHighDateTime: DWORD; end;

To есть, Элементы записи TFiieTime: dwLowDateTime И dwHighDateTime В сумме образуют 64-разрядное значение. Это значение определяет количество интервалов (каждый интервал равен 100 наносекунд), которые прошли начиная с 1 января 1601 года.

Использование главного VCL-потока

Метод Execute является потоковой функцией. Вы можете представить поток как приложение, которое запускается из вашего приложения, за исключением одной детали: и ваше приложение, и поток находятся в одном процессе. Написание потоковой функции немного сложнее, чем .создание обычного приложения, т. к. вам придется следить за тем, чтобы разные потоки вашего приложения не пересекались в оперативной памяти. С другой стороны, поскольку потоки находятся в одном процессе, вы можете использовать память потоками совместно, для их взаимодействия.

Когда вы используете в своем приложении объекты из библиотеки визуальных компонентов, то вам нужно иметь в виду, что свойства и методы объектов из VCL не гарантируют потоковую безопасность. В случае, когда все объекты обращаются к своим свойствам и выполняют свои методы в одном потоке, вам не нужно беспокоиться о том, что объекты будут создавать друг другу помехи.

Для использования главного VCL-потока создайте отдельную процедуру, которая выполняет необходимые действия. А затем вызовите эту процедуру из вашего потокового модуля (внутри метода Execute), используя синхронизацию. Например:

procedure TMyThread.PushTheButton; begin

Buttonl.Click; end;

procedure TMyThread.Execute; begin

Synchronize(PushTheButton); end;

Метод синхронизации защитит ваше приложение от ситуации "гонки ". Ситуация "гонки" возникает, когда два или более потока пытаются получить доступ к общему ресурсу и изменить его состояние.

В некоторых случаях вы можете обходиться без метода синхронизации, например:

* компоненты доступа к данным (Data access) являются потокобезопасными в том случае, когда каждый поток обращается к собственной базе данных. Есть одно исключение, когда вы пытаетесь получить доступ к базе данных Microsoft Access. Access работает с библиотекой Microsoft ADO, которая не является потокобезопасной.
* объекты для работы с графикой являются потокобезопасными. Это такие классы, как TFont, TPen, TBrush, TBitmap, TMetaf ile И Ticon;
* вместо объектов списков (List), которые не являются потокобезопасными, вы можете использовать потокобезопасный потомок объекта TList - TThreadList.

Координация потоков

При работе с потоками, на этапе первоначального знакомства с ними, неизбежны ошибки. Особенно неприятны ошибки, связанные с конфликтами потоков, обращающихся к разделяемым ресурсам, а также глобальным переменным. Иногда бывает необходимо, чтобы потоки работали слаженно, т. е., например, после выполнения какого-либо события в одном потоке вам необходимо, чтобы возникало определенное постоянное событие в другом потоке. Все это можно объединить под общим названием координации потоков. Рассмотрим, как можно эффективно управлять потоками для достижения решения практических задач. Для начала определим, какие способы хранения локальных переменных потока нам предоставляет Delphi. Таких способов три:

* хранение локальных переменных в стеке потока. Так как любой поток приложения получает свой собственный стек, то он будет иметь собственные локальные переменные;
* сохранение локальных переменных в объекте потомка класса TThread;
* хранение локальных переменных на уровне операционной системы, используя в описании локальных переменных слово threadvar.

Первый способ является самым простым и очевидным, а также и самым эффективным. Доступ к локальным переменным, расположенным в стеке потока, самый быстрый. Рассмотрим два других способа хранения локальных переменных потока. Второй способ проще и эффективнее, чем третий. Рассмотрим его на примере: type

TMyThreadl = class(TThread) private

i,j,k,1: integer; // локальные переменные потока типа integer

a,b,c: char; // локальные переменные потока типа char

end;

Эффективность объявления локальных переменных в потомке класса TThread очевидна, т. к. доступ к любому полю объекта осуществляется очень быстро (примерно в 10-11 раз быстрее, чем при использовании описания threadvar).

Третий способ хранения локальных переменных (с помощью threadvar), служит для создания в потоках локальных копий глобальных переменных. Глобальные переменные могут использоваться всеми потоками приложения, при этом может возникнуть ситуация, когда при изменении глобальной переменной одним потоком происходит изменение этой же глобальной переменной другим потоком. В результате значение, установленное первым потоком, бесследно исчезает, приводя к нежелательным последствиям (в данном примере произойдет обработка ошибочного значения глобальной переменной первым потоком). Для исключения такой ситуации Win32 API предлагает средство хранениялокалъных данных потоков (thread-local storage). С помощью этого средства можно создавать локальные копии глобальных переменных для любого потока. При этом требуется всего лишь заменить одно слово var при объявлении глобальных переменных на слово threadvar.

Синхронизация потоков

При дальнейшей работе с потоками вам придется подстраивать работу потоков таким образом, чтобы они не мешали друг другу. Подобная настройка потоков называется синхронизацией потоков.

Ожидание завершения работы потока

К основным понятиям механизма синхронизации потоков относятся функции и объекты ожидания. В Win32 API имеется несколько функций, называемых функциями ожидания, позволяющих приостановить работу потока до тех пор, пока не будет изменено состояние какого-либо объекта операционной системы, который является объектом ожидания. К таким объектам относятся следующие объекты ядра Win32 API:

* критические секции (critical section);
* события (Events);
* мьютексы или взаимные исключения (Mutex);
* семафоры (Semaphore);
* таймер (Timer).

Кроме вышеперечисленных объектов, можно использовать и другие объекты, например потоки, изменения в файловой системе, консольный ввод и др.

Использование критической секции Если в вашем приложении имеются объекты, которые не являются потоко-безопасными, Delphi позволяет использовать так называемые критические секции. Критическая секция - это область глобальной памяти, которую необходимо защищать от одновременного использования ее разными потоками.

Критическая секция работает наподобие "ворот", которые закрываются для всех потоков в то время, когда один поток "зашел" в эти "ворота". Она позволяет работать с данной областью памяти только одному потоку, блокируя доступ всех остальных потоков. Для использования критической секции в своем приложении вы должны создать глобальный экземпляр класса TCriticaisection. Данный класс имеет несколько методов, из которых для нас важны два: Acquire (или Enter) и Release (ПЛИ Leave).

Метод Acquire (Enter) связывает критическую секцию с потоком, вызвавшим данный метод, блокируя доступ к этой секции всем остальным потокам.

Метод Release (Leave) снимает блокировку и позволяет другим потокам обращаться к критической секции. При использовании критических секций будьте внимательны, т. к., если в вашем приложении несколько потоков имеют доступ к критической секции, они все должны работать с методом Acquire (Enter) для обращения к ней. Иначе может возникнуть проблема совместного доступа к ресурсу.

Для примера рассмотрим небольшую программу, которая содержит глобальную переменную критической секции LockXY. Данная переменная блокирует доступ к глобальным переменным X и Y.

LockXY.Acquire; // блокирование других потоков

try

Y := sin(X); finally

LockXY.Release; end;

Любые потоки, которые используют глобальные переменные X или Y, должны содержать примерно такой же код.

События

Данный объект ожидания является простейшим для решения задачи синхронизации. Объект типа событие (TEvent) может принимать одно из двух состояний: активное или пассивное.

Когда объект ожидания находится в активном состоянии, его видят многие потоки одновременно. В результате такой объект можно использовать для управления работой сразу многих потоков.

Объект класса TEvent имеет среди других своих методов два метода: SetEvent, переводящий объект в активное состояние, и ResetEvent, переводящий объект в пассивное состояние. Кроме того, этот объект содержит метод waitFor:

type TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError); function WaitFor(Timeout: DWORD): TWaitResult;

Данный метод имеет один параметр Timeout, задающий количество миллисекунд, в течение которых объект ожидает активизацию события. Этот метод возвращает значение wrSignaled в том случае, если активизация события произошла, и wrTimeout - в противном случае. Если в качестве параметра Timeout передать значение INFINITE, то ожидание активизации события будет бесконечно долгим. Для создания объекта события можно воспользоваться функцией CreateEvent: CreateEvent (lpEventAttributes, bManualReset, bInitialState, lpName); где:

* IpEventAttributes - определяет дескриптор безопасности для нового события. В Windows NT, если данный параметр равен NULL, то событие получает дескриптор безопасности, установленный по умолчанию. В Windows 9х данный параметр игнорируется;
* bManualReset - определяет ручной или автоматический сброс состояния события. Если значение данного параметра true, то вы должны использовать функцию ResetEvent для ручного перевода события в пассивное состояние. Если значение false, то Windows автоматически осуществит такой перевод;
* bInitialState - определяет начальное состояние события (активное или пассивное). Если значение параметра true, то состояние активное, иначе - пассивное;
* LpName - определяет имя/события для обращения к нему из других потоков. Имя может содержать любые символы, кроме обратной косой черты (\). Имя чувствительно к регистру букв. В случае, когда имя повторяет имя созданного события, происходит обращение к уже существующему событию. Если значение имени равно NULL, то событие создается без имени. Если событие имеет такое же имя, как семафор, мьютекс или другой объект, генерируется ошибка ERROR\_INVALID HANDLE.

Мьютексы (взаимные исключения) Данный тип объекта ожидания разрешает доступ только одному потоку в определенный момент времени. Мьютекс сигнализирует, когда он не при надлежит потоку и прекращает сигнализировать, когда он принадлежит потоку. После того как поток перестает обращаться к мьютексу, любой другой поток получает возможность доступа к нему.

Данный объект полезен в случае обращения к памяти с целью ее изменения различными потоками. Для создания объекта типа мьютекс потоки должны обращаться к функции CreateMutex!

CreateMutex (lpMutexAttributes, blnitialOwner, IpName); где:

* IpMutexAttributes - указатель на структуру SECURITY\_ATTRIBUTES, которая определяет, может ли обработчик наследоваться потоками-потомками. В Windows NT, при IpMutexAttributes = NULL, мьютекс получает дескриптор безопасности, установленный по умолчанию. В

Windows 9x данный параметр функции игнорируется;

* blnitialOwner - определяет начального владельца мьютекса. Если данный параметр равен true, то владельцем мьютекса становится вызвавший его поток. В противном случае, владелец мьютекса не определен;
* lpName - определяет имя мьютекса. Имя предназначено для обращения к мьютексу из других процессов. Имя может содержать любые символы, кроме обратной косой Черты (\). Имя чувствительно к регистру букв. В случае, когда имя повторяет имя уже созданного мьютекса, происходит обращение к уже существующему мьютексу. Если значение имени равно NULL, то мьютекс создается без имени. Если мьютекс имеет такое же имя, как семафор, событие или другой объект, генерируется ошибка ERROR\_INVALID\_HANDLE.

Семафоры

Иногда бывает необходимо ограничить число потоков, обращающихся к объекту ожидания. При этом число таких потоков может быть разным (не только один). Объект типа семафор может устанавливаться на доступ к нему определенного числа потоков. При этом все потоки, которые вновь обращаются к семафору, но, при этом, их количество уже превышает предельно допустимое количество, - приостанавливаются. При отключении присоединенных к семафору потоков "ждущие" потоки подключаются к нему.

Семафоры полезны при контролировании доступа к разделяемым ресурсам, которые могут поддерживать ограниченное число пользователей. Например, приложение может создать ограниченное число окон, отображаемых на экране. В данном случае можно использовать семафор, установленный на максимальное число окон, которые могут вывести приложение. Потоки используют функцию CreateSemaphore для создания объекта типа семафор: CreateSemaphore (IpSemaphoreAttributes, UnitialCount, IMaximumCount, IpName); где:

* lpSemaphoreAttributes - определяет дескриптор безопасности для нового семафора, данный параметр игнорируется для Windows 9x. Если параметр равен NULL, то семафор получает дескриптор безопасности по умолчанию;
* lInitialCount - определяет начальное значение счетчика для семафора. Оно должно быть больше нуля и меньшим или равным значению lMaximumCount;
* lMaximumCount - определяет максимальное значение счетчика для семафора (максимальное число потоков). Оно должно быть больше нуля;
* lpName - определяет имя семафора. Имя может содержать в своем составе любые знаки и символы, за исключением символа обратной косой черты (\). Имя является чувствительным к строчным и прописным буквам. Если семафор с данным именем уже был определен, то будет осуществлен доступ к уже существующему семафору. Если значение IpName равно NULL, то семафор будет создан без имени. Если семафор будет иметь такое же имя, как и другой объект (например, мьютекс, взаимное исключение, или другой), то произойдет ошибка и при вызове функции GetLastError будет возвращено' значение ERROR\_INVALID\_HANDLE. Данная ошибка произойдет по простой причине, т. к. все эти объекты разделяют одно и то же пространство имен.

Запуск и остановка потоков

Потоки могут быть запущены и остановлены сколько угодно раз в процессе их выполнения. Для временной остановки запущенного потока можно обратиться к методу потока suspend. Для продолжения выполнения приостановленного потока вызовите метод потока Resume. Вы можете использовать вложенные вызовы вышеперечисленных методов, т. к. метод Suspend увеличивает внутренний счетчик потока, a Resume уменьшает. Поток не будет выполняться до тех пор, пока счетчик не обратиться в ноль, т. е., если вы вызвали пять раз метод Suspend, а затем четыре раза Resume, вам понадобится еще один (пятый) вызов метода Resume для продолжения выполнения потока.

Функции WinAPI по управлению потоками

В современных операционных системах широко используются нити (потоки), как части процесса, способные выполняться параллельно, конкурируя за главный ресурс - процессор, но

делающие общую работу. Процесс в современной операционной системе - это владелец основных ресурсов кооперативно работающих нитей. У этих нитей общее виртуальное адресное пространство и общие дескрипторы и соответствующие им хэндлы. Практически нити одного процесса используют общий сегмент данных, откуда каждая нить может брать и записывать данные. Процессором нити пользуются по очереди, так что он не принадлежит ни одной из них.

Главная нить процесса создается автоматически при создании процесса. Если процесс нуждается в дополнительных нитях, то его программа вызывает системные функции создания нити. Обычно нить своей работой реализует действия одной из процедур программы. Теоретически любой нити процесса доступны все части программы, но реально работа организуется так, чтобы нити отвечала отдельная процедура.

В операционной системе Windows накопилось несколько функций порождения нитей. Самой ранней является функция CreateThread, но из-за обнаруженных у нее и нескольких последующих версий недостатков (в редких, но возможных ситуациях), были введены еще несколько, отличающихся по порядку и , частично, по набору аргументов. В настоящее время наиболее безопасной функцией порождения нити является \_beginthreadNT, описанной в заголовочном файле process.h и имеющей следующий прототип:

unsigned long \_beginthreadNT(void (\*proc)(void\*), unsigned stack\_size,void \*arglist, void \*security\_attrib, unsigned long flag,unsigned long\* tid).

Первым аргументом идет адрес процедуры нити, в стандартном случае эта процедура должна быть функцией без возвращаемого значения (функция типа void) и нетипизированным аргументом (формально типа void\*). Процедура нити, созданной функцией \_beginthreadNT , должна оканчивать специальной функцией \_endthread().

Вторым аргументом указывается размер стека, округляемый практически до значения, кратного 4096.

Третий аргумент arglist задает указатель на строку аргументов, в простейшем случае отсутствия аргументов он может быть задан значением NULL.

Аргумент security attrib используется в более сложных программах, реализующих всю мощь встроенной защиты в WindowsNT. В обычных случаях его также можно задавать равным NULL.

Пятый аргумент flag служит для задания режима приостановленной при создании нити, что обозначается символической константой CREATE\_SUSPENDED. В противном случае (запуск тут же функционирующей нити) этот параметр-флаг устанавливается равным нулю.

Последний параметр функции предназначен для возвращаемого значения идентификатора нити. В случае неудачи при создании нити возвращается число -1, в остальных случаях - хэндл созданной нити, который как переменная может быть описан с типом HANDLE.

Следующий ниже пример иллюстрирует построение и использование нитей.

#include <windows.h> #include <process.h> #include <stdio.h> #include <conio.h>

char lbuk[]="abcdefghijklmnoprqstuwxy";

void procthread1(void \*arg) { int k,j;

for (k=0;k<24;k++) { gotoxy(2 0,k+1);

textcolor(LIGHTBLUE); for (j=0;j<(int)arg;j++) cprinf("%c",lbuk[k]);

}

}

void procthread2(void \*arg) { int k,j;

for (k=0;k<24;k++){ gotoxy(40,k+1);

textcolor(LIGHTGREEN); for (j=0;j<(int)arg;j++) cprinf("%c",lbuk[k]);

}

void procthread3(void \*arg) { int k,j;

for (k=0;k<24;k++) { gotoxy(60,k+1); textcolor(RED); for (j=0;j<(int)arg;j++) cprinf("%c",lbuk[k]);

}

}

void main()

{ HANDLE hthread1,hthread2,hthread3;

unsigned long threadid1,threadid2,threadid3; int k;

hthread1=(HANDLE)\_beginthreadNT(procthread1,4096,(void \*)2,NULL,0,&threadid1); hthread2=(HANDLE)\_beginthreadNT(procthread2,4096,(void \*)3,NULL,0,&threadid2); hthread3=(HANDLE)\_beginthreadNT(procthread3,4096,(void \*)4,NULL,0,&threadid3); for (k=0;k<24;k++) { gotoxy(1,k+1);

textcolor(WHITE); for (j=0;j<(int)arg;j++) cprinf("%c",lbuk[k]);

}

getchar();

CloseHandle(hthreadl); CloseHandle(hthread2);

CloseHandle(hthread3); }

Для уничтожения нитей в операционных системах Windows служит системная функция TerminateThread, которая имеет следующий прототип:

BOOL TerminateThread(HANDLE hThread,DWORD dwExitCode).

Обычно эту функцию используют только тогда, когда управление нитью потеряно и она ни на что не реагирует.

Код возврата из нити, завершившейся самостоятельно или прекращенной приказом TerminateThread, может быть получен в другой нити того же процесса путем вызова вспомогательной функции GetExitCodeThread со следующим прототипом: BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode). Если задача, указанная аргументом hThread в момент запроса кода еще работает, то функция возвращает значение STILL\_ACTIVE в качестве значения второго аргумента.

Код возврата из нормально завершающейся нити формирует системная функция ExitThread, имеющая прототип

voidExitThread(DWORD dwExitCode).

Задание

Разработать программу, которая создает в отдельном потоке случайный массив А из N целых чисел в диапазоне от -999 до 999 выводит на экран эти числа. Создание и вывод элементов массива производится через заданное время T, N и T вводятся пользователем до запуска процесса. Массив обрабатывается двумя другими потоками ВиС, работающими параллельно с потоком, создающим массив. Все потоки выводят результаты своей работы в текстовые окна, каждый поток в свое окно.

|  |  |  |
| --- | --- | --- |
| № вар. | Задание В | Задание С |
| 1 | Сортировка массива в порядке убывания | Деление массива на подмассивы отрицательных и положительных чисел и вывод сначала отрицательных чисел, а затем положительных |
| 2 | Сортировка массива в порядке возрастания | Перестановка чисел A[j] и A[n-j], где n- текущее число элементов массива |
| 3 | Вычисление среднего арифметического и среднего квадратичного значений | Сортировка чисел в порядке убывания абсолютного значения |
| 4 | Вычисление минимального и максимального значений | Вычисление отклонений каждого значения от среднего арифметического |
| 5 | Вычисление количества чисел в каждой сотне возможных значений | Сортировка чисел в порядке возрастания абсолютного значения |
| 6 | Вычисление квадратов разностей чисел и их среднего арифметического | Вычисление, сколько раз повторяется каждое число в массиве |
| 7 | Вычисление произведений вида A[j]\*A[n-j]? где n-текущее число элементов массива | Определение разности сумм четных и нечетных элементов массива |
| 8 | Определение суммарного количества цифр (от 0 до 9), встречающихся во всех числах | Подсчет количества чисел, попадающих в заданный пользователем набор интервалов с постоянным шагом (гистограмма) |
| 9 | Вычисление общего количества цифр всех элементов массива | Определение и вывод чисел, являющихся простыми числами |
| 10 | Определение и вывод чисел, состоящих из различных цифр | Определение длины плоской ломаной линии, соединяющей точки, координаты которых заданы каждой парой чисел |
| 11 | Определение длины пространственной ломаной линии, соединяющей точки, координаты которых заданы каждой тройкой чисел | Вычисление отклонений всех чисел от округленного среднего арифметического |
| 12 | Определение и вывод элементов массива, являющихся квадратами целых числа | Вычисление отклонений всех чисел от максимального значения |
| 13 | Вычисление отклонений всех чисел от минимального значения | Определение и вывод чисел, являющихся целыми степенями двойки |
| 14 | Случайное перемешивание элементов массива | Определение и вывод чисел, состоящих только из одинаковых цифр |
| 15 | Определение и вывод чисел, содержащими нули как цифры | Определение количеств чисел, меньших и больших среднего арифметического |
| 16 | Определение и вывод элементов массива, являющихся палиндромами | Сортировка и перестановка трех чисел массива: первого, среднего и последнего, в порядке возрастания |
| 17 | Определение и вывод элементов массива, имеющих совпадающие первую и последнюю цифры | Вычисление модулей отклонений чисел от среднего арифметического и их сортировка в порядке возрастаний |
| 18 | Определение максимального расстояния от центра системы координат точек, заданных каждой парой чисел | Определение отклонений каждого числа от моды (полусуммы наибольшего и наименьшего значений) |
| 19 | Определение минимального расстояния от центра системы координат точек, заданных каждой парой чисел | Вычисление нормализованных значений (отношение числа к среднему квадратическому отклонению) |
| 20 | Определение среднего расстояния от центра системы координат точек, заданных каждой парой чисел | Вычисление отношений чисел к размаху (разности наибольшего и наименьшего значений), выраженных в процентах |

Содержание пояснительной записки

Титульный лист Задание на курсовую работу

1. Описание общего алгоритма и интерфейса программы
2. Описание методов решения и алгоритмов задач, реализуемых каждым потоком
3. Описание методов создания, уничтожения и синхронизации потоков, примененных в программе
4. Листинг программы Заключение

Список использованной литературы